Università degli Studi di Padova

Facoltà di Scienze MM.FF.NN.

Corso di Informatica

tesi di laurea

Dai requisiti all'installazione

Processi e strumenti in un team di sviluppo software

Laureando: Manuel Paccagnella
Relatore: Claudio E. Palazzi

16 gennaio 2010

Licenza

[1] http://bitbucket.org/manuelp/tesi/wiki/Home
[2] http://creativecommons.org/licenses/by-nc-nd/2.5/it/

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

Il presente capitolo ha lo scopo di introdurre il contesto in cui si è svolto lo stage presentando l'azienda ospitante, e di illustrarne lo scopo, gli obiettivi e il piano di lavoro stabilito inizialmente per il loro raggiungimento.

1.1 Azienda ospitante

L'azienda ospitante lo stage è la *Visionest*[1] di Padova:

> Visionest S.r.l. - The Business Innovator
> Via G.B. Ricci 6/A
> 35131 Padova - Italy

Nata come società di consulenza, opera in diversi ambiti:

- **Ingegneria e informatizzazione dei processi**
- **Gestione strategica della conoscenza**
- **Sicurezza & Conformità**
- **Consulenza IT**

Ad oggi può vantare esperienze con, tra gli altri: Piaggio-Aprilia, Telecom Italia Mobile, SOGEI-Ministero delle Finanze, Istituto Universitario Europeo, Università degli Studi di Padova e Verona, Regione Veneto.

[1] http://www.visionest.com/

Da pochi anni si è proposta anche come *software house*, che si occupa principalmente di software *web-based* a supporto di realtà aziendali, ed ha attualmente come progetti attivi:

- **VisionHR**: software di gestione delle risorse umane basato sulle tecnologie: *Struts, Hibernate* e *Ant*.
- **A-Plus**: *framework* sviluppato *in-house* per la produzione di strumenti di BPM[2] e BDM[3]. Le tecnologie utilizzate sono: *Spring, Hibernate, JBPM, Drools, Alfresco* e *Maven*.
- **Finanza 3000**: Strumento di gestione finanziaria realizzato in parallelo e su *A-Plus*, che ha come principale (ma non esclusivo) cliente l'ente *Veneto Sviluppo*.

1.2 Scopo

L'obiettivo dello stage è quello di analizzare e fornire soluzioni per migliorare i processi e l'infrastruttura in un team di sviluppo software.

Si cercherà di avvicinarsi ad una metodologia di sviluppo *agile*, in modo da avere dei processi efficienti ed efficaci che consentano di rispondere meglio e con maggiore reattività alle esigenze dei clienti, oltre che a ottenere dei prodotti di qualità maggiore.

In particolar modo si cercherà di incorporare dei rapidi *cicli di feedback* e di mettere il team in condizioni di analizzarli per poter migliorare autonomamente e con continuità i processi, adattandoli alle proprie capacità e al progetto in sviluppo.

1.3 Obiettivi

1.3.1 Obbligatori

Principalmente il lavoro dovrà prevedere l'analisi dei processi e dell'infrastruttura attualmente impiegati dall'azienda e l'elaborazione e la proposta di miglioramenti da apportare per rendere l'organizzazione più *agile* e *scalabile*.

[2] *Business Process Management.*
[3] *Business Document Management.*

1.3.2 Desiderabili

E' desiderabile l'installazione e la messa in opera dei nuovi strumenti proposti una volta che questi sono stati approvati.

1.3.3 Opzionali

Potrà rendersi necessario o conveniente la realizzazione di script e piccoli strumenti per facilitare l'integrazione degli strumenti nell'infrastruttura e nel *workflow* attualmente adottato dal team di sviluppo.

1.4 Aree di intervento

Il lavoro oggetto dello stage si dividerà essenzialmente in due filoni:

- **Analisi dei processi**: una prima parte dedicata all'analisi del *workflow* attualmente adottato dal team di sviluppo, dalla raccolta dei requisiti al *deployment*. L'analisi coprirà sia i processi sia gli strumenti utilizzati. Successivamente verranno suggeriti dei miglioramenti da apportare per compiere dei passi avanti e convergere sempre di più a una metodologia di sviluppo agile.
- **Infrastruttura**: una seconda area di intervento sarà la selezione, la presentazione agli sviluppatori e l'implementazione di strumenti utili all'infrastruttura di sviluppo, in modo da supportare al meglio i miglioramenti che verranno introdotti.

Le aree di intervento interessate saranno quindi:

- **Processi**: dalla *issue* al *deploy*, con particolare interesse verso i metodi di comunicazione con il cliente (*gestione di requisiti, issue e documenti*).
- **Strumenti**: *SCM, CI, WBS e Issue Tracking*, oltre che la loro integrazione nel workflow del team di sviluppo nel modo meno intrusivo possibile (il che potrà comprendere anche presentazioni e *training* nell'utilizzo).
- **Comunicazione**: grande importanza riveste la comunicazione, partendo dalle relazioni all'interno del team di sviluppo; quindi verrà prestata particolare attenzione agli aspetti di *telepresenza, suddivisione del lavoro (gestione dei task), status reporting e awareness*, elementi critici visto che tipicamente gli sviluppatori non si trovano nello

stesso luogo fisico, ma sono distribuiti geograficamente per esigenze di servizio.

1.5 Piano di lavoro

Il piano di lavoro è stato suddiviso in 4 fasi tenendo conto delle aree di intervento e del tempo necessario per lo studio e l'implementazione.

1.5.1 Documentazione

- Dal: 28/09/2009
- Al: 06/10/2009
- Durata: 60 ore

Questa prima fase sarà dedicata allo studio delle metodologie agili, delle problematiche da affrontare nella migrazione verso di esse, e degli strumenti a supporto.

1.5.2 Rilevazione

- Dal: 07/10/2009
- Al: 16/10/2009
- Durata: 60 ore

Questa fase sarà dedicata allo studio dell'attuale metodo di lavoro adottato dal team di sviluppo (sia per quanto riguarda i processi che per gli strumenti), principalmente attraverso interviste ai membri del team stesso.

1.5.3 Proposizione

- Dal: 19/10/2009
- Al: 06/11/2009
- Durata: 120 ore

La fase più onerosa e che comprenderà lo studio, la proposta e l'implementazione di miglioramenti metodologici e infrastrutturali (quindi streamlining dei processi e integrazioni all'infrastruttura).

1.5.4 Conclusioni

- Dal: 09/11/2009
- Al: 18/11/2009
- Durata: 60 ore

La fase finale consisterà nella presentazione delle opportunità di miglioramento e nell'adozione dei nuovi strumenti da parte del team.

1.6 Struttura della tesi

La presente tesi di laurea è suddivisa in quattro sezioni principali:

Introduzione Il presente capitolo, il cui obiettivo è presentare lo stage, i suoi obiettivi e l'azienda presso la quale è stato portato a termine.

Teoria In questo capitolo verrà illustrata brevemente la storia dello sviluppo del software, introducendo poi il concetto di *Agile Software Development* con i suoi principi ispiratori, il ciclo di vita e le pratiche più importanti. L'obiettivo principale è illustrare l'origine, le ragioni, i vantaggi e l'applicabilità dello sviluppo agile del software.

Assessment Questa sezione presenta la fase di *assessment* di una realtà produttiva nel capo del *software development*, prima dal punto di vista teorico illustrando in cosa consiste, quali sono gli obiettivi, i requisiti e le "tecniche" più adatte. Successivamente si affronta l'*assessment* da un punto di vista pratico, illustrando come è stato applicato all'azienda ospitante.

Evoluzione Questo capitolo si occupa della progettazione e della proposizione delle modifiche infrastrutturali e processuali necessarie all'evoluzione dell'organizzazione in accordo ai suoi *business values*. Una prima parte si occupa della parte teorica che sarà di supporto alla scelta delle misure da adottare, seguita dall'applicazione pratica nel contesto aziendale oggetto dello stage. Le misure proposte sono suddivise in base all'applicazione (progetti già avviati o nuovi), inoltre vengono presentate delle idee per stabilire un piano di miglioramento a lungo termine. L'*apprendimento continuo* viene trattato come un requisito necessario al miglioramento continuo dell'organizzazione e del processo produttivo.

Ed infine un'appendice:

Effettuare una presentazione Le capacità di comunicare e presentare efficacemente prodotti, idee o soluzioni sono estremamente utili in un contesto aziendale per interagire con i colleghi, i clienti e il *management*. In questa appendice vengono trattati dei suggerimenti sulla preparazione di una presentazione efficace. I concetti presentati sono stati utilizzati per preparare ed effettuare la presentazione del lavoro svolto all'azienda ospitate e saranno impiegati anche per preparare la discussione della presente tesi.

1.7 Convenzioni tipografiche

Nel presente lavoro sono state utilizzate le seguenti convenzioni tipografiche:

- *Enfatizzato*: per evidenziare parti importanti, nomi e parole inglesi quando ritenuto opportuno.
- **Grassetto**: per titoli e parti particolarmente importanti.
- `Teletype`: per identificare i nomi degli strumenti proposti.

Capitolo 2

Teoria

Nel presente capitolo verranno trattati brevemente i fondamenti teorici che sono stati alla base del lavoro svolto durante lo stage. Quello dello sviluppo del software è un'argomento molto vasto e in evoluzione, perciò il presente capitolo non ha alcuna pretesa di completezza.

2.1 Panoramica storica sul *software development*

Quello dell'ingegneria del software[1] è un campo molto giovane: la disciplina è nata verso la fine degli anni 60, in contrasto con ad esempio quella elettronica che nasce negli anni 50 ma come branca dell'elettrotecnica che risale all'inizio del XX secolo. Come disciplina ingegneristica, per quanto riguarda la sua comprensione della natura dello sviluppo del software e del modo migliore per farlo, è ancora immatura e non ha avuto il tempo di produrre una comprensione utile, completa e condivisa.

2.1.1 Nascita dell'ingegneria del software

Inizialmente non esisteva quasi alcuna concezione di ingegnerizzazione metodica e condivisa del software, e tutto veniva fatto con un approccio di tipo *code&fix*[2]. Tuttavia, ci si rese conto che questo modo di lavorare portava al fallimento della maggior parte dei progetti. Questo fornì la spinta per

[1] Che lo standard IEEE 610.12-1990[16] definisce: *"l'applicazione di un approccio sistematico, disciplinato e quantificabile nello sviluppo, funzionamento e manutenzione del software."*

[2] Nessuna progettazione o pianificazione, si iniziava subito la codifica e senza alcun criterio che non fosse il giudizio di ogni singolo sviluppatore. Era praticamente un far west, caotico e disorganizzato.

la nascita nel 1968 della disciplina nella conferenza NATO a Gramisch, in Germania, per ricercare una soluzione.

Con l'andare del tempo vennero formalizzati linguaggio, processi e un *corpus* di conoscenze[6] il cui scopo era quello di permettere la realizzazione di software in modo *affidabile* e *ripetibile*.

2.1.2 Le crisi del software

Tuttavia, tra gli anni 60 e 90 l'industria del software attraversò un lungo periodo di crisi durante il quale vennero identificati molti problemi connessi allo sviluppo. Non si trattava più solo di problemi di produttività, ma anche di progetti che superavano tempi e costi, cancellati o che addirittura causavano danni a proprietà e perdite di vite umane[17].

A causa di queste crisi ci si lanciò nella ricerca di un *silver bullet*[3]. Venne ricercato in molte aree, su tutte:

- **Strumenti**: OOP, tool CASE, documentazione, programmazione strutturata, ecc.
- **Metodi formali**: dimostrazioni formali di correttezza, analisi simbolica, ecc.
- **Processi**: RUP, CMMI, ecc.

Ognuno era alla ricerca della singola tecnologia, strumento, metodo o processo che avrebbe risolto la crisi permettendo la produzione di software di qualità, in modo ripetibile ed entro i tempi e i costi preventivati. La ricerca e il dibattito infiammò per molti anni, fino a quando Fred Brooks pubblicò nel 1986 il suo articolo *No Silver Bullet - Essence and Accident in Software Engineering*[18]. Un pò alla volta ci si rese conto che lo sviluppo del software è un campo talmente complesso da presentare delle difficoltà intrinseche non eliminabili da nessun singolo rimedio. Ci si rese conto che *non esistono silver bullet*.

Durante questo periodo, molti credettero che la soluzione definitiva sarebbe stata rappresentata da processi formali ben definiti e documentati, da seguire religiosamente. L'idea che ne stava alla base era che tutti i problemi erano dovuti alla disorganizzazione e alla mancanza di controllo sul lavoro svolto

[3] Un singolo rimedio semplice e di estrema efficacia, la soluzione a tutti i problemi. La denominazione "proiettile d'argento" deriva dal folklore e viene usato in varie storie come arma per contrastare minacce del male invulnerabili a tutte le altre armi come licantropi (dalla leggenda della bestia di Gévaudan), streghe (da una favola dei fratelli Grimm), ecc. L'argento è il simbolo di giustizia, legge e ordine.

dai componenti del team di sviluppo. Il processo di produzione di software di qualità sarebbe dovuto essere facilmente riproducibile e controllabile una volta codificati i processi e i ruoli di ogni persona.

Le caratteristiche principali di questo approccio al *software development*, che è stato soprannominato *heavyweight*, sono:

- Processi definiti e documentati dettagliatamente in modo normativo
- Interscambiabilità delle persone: l'eccessiva attenzione ai processi ha portato a svalutare il ruolo delle persone, passate da elementi fondamentali e meri esecutori senza alcuna importanza specifica
- Rigida suddivisione in ruoli: tentativo di replicare il funzionamento delle catene di montaggio e di diminuire al minimo la dipendenza da singole persone (e quindi l'importanza)
- Ciclo di vita sequenziale (*Waterfall*[4])
- Pesante burocrazia: produzione e manutenzione di una grande quantità di documentazione, tanto da far guadagnare all'approccio l'aggettivo *document-driven*

2.1.3 Nascita dell'Agile

Tuttavia, nonostante l'istituzione di standard ISO, IEEE, e certificazioni a livello industriale, questo approccio non ha avuto il risultato sperato. Secondo il rapporto CHAOS 1995[19] dello Standish Group[5], i livelli di successo e fallimento di progetti software in tutto il mondo nel 1995 rimanevano sorprendentemente bassi. Le percentuali sono riportate in tabella 2.1.

Successi	16.2%
A rischio	52.7%
Fallimenti	31.1%

Tabella 2.1: Dati del rapporto CHAOS 1995

Il modo di lavorare che molte aziende adottarono sperando di trovare la soluzione ai loro problemi si dimostrò in molti casi troppo pesante (*heavyweight*), burocratico, lento, "disumanizzante" (troppa poca importanza

[4]Ciclo di vita nel quale le fasi dello sviluppo (analisi funzionale, progettazione, codifica, testing, integrazione e *deployment*) vengono eseguite una sola volta e in modo strettamente sequenziale.

[5]Che si occupa di analisi statistiche sullo stato dell'industria del software a livello mondiale.

sulle persone), rischioso[6] e inconsistente con il modo in cui gli sviluppatori svolgono il loro lavoro in modo efficace. Si resero sempre più evidenti gli svantaggi e le inefficienze di questo modo di sviluppare.

Il problema dell'ingegneria come metafora dello sviluppo del software è che lavorare con la conoscenza e le informazioni non è un lavoro di tipo industriale, e la programmazione non è una catena di montaggio. Per sua natura, lo sviluppo è un'attività *creativa*[7] che ha luogo per di più in un ambiente in continuo mutamento e non completamente controllabile.

Verso la metà degli anni 90 è cominciata ad emergere una generale insoddisfazione verso l'approccio all'ingegneria del software propugnato dai sostenitori dei processi come *silver bullet*. Come reazione alle metodologie *heavyweight*, hanno cominciato a emergere e diffondersi le metodologie *lightweight* che mirano ad essere meno burocratiche e pesanti, riportando l'attenzione sulle persone e sullo scopo di produrre software flessibile, affidabile, di qualità e che risponde alle esigenze dei clienti in modo *efficace* ed *efficiente*, il tutto in un contesto[8] non completamente definito o controllabile e in continuo mutamento.

Il modo principale con il quale si è cercato di raggiungere questo scopo è attraverso un ciclo di vita *iterativo* e *incrementale* che incorpora il *feedback* proveniente da varie fonti nel processo di sviluppo, rendendo il team e il progetto stesso in grado di adattarsi ai cambiamenti (di requisiti, componenti del team, tecnologie, ecc) in modo tempestivo ed efficiente.

Nel 2001, 17 prominenti figure[9] con vasta esperienza nel campo dello sviluppo del software, si sono incontrati allo *ski resort* Snowbird nello Utah per discutere modi di creare software in modo più leggero, rapido e incentrato sulle persone. Da quella discussione sono emersi i termini *Agile Software Development*, "metodi agili" e l'*Agile Manifesto*[22] (di seguito riportato) che è considerato la definizione canonica di sviluppo agile e dei principi che lo animano.

[6] Il ciclo di vita sequenziale, prevede che i requisiti siano tutti fissati con precisione prima della progettazione e che questi non cambino. Una situazione che nella stragrande maggioranza dei casi si è rivelata impossibile. Anzi, i requisiti sono, secondo lo Standish Group[19], una delle principali ragioni di fallimento dei progetti.

[7] Direi che è a cavallo tra scienza e arte, di cui fanno parte attività che non possono essere codificate senza snaturarle, come ad esempio il *design*.

[8] Requisiti, team, clienti, tecnologie, ecc.

[9] 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.

Manifesto for Agile Software Development

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.

Successivamente, alcune delle persone firmatarie dell'*Agile Manifesto* hanno formato l'organizzazione no-profit *The Agile Alliance*[10] il cui scopo è promuovere questi principi.

Il nome *Agile* quindi, identifica un'atteggiamento e una filosofia più che una specifica metodologia di sviluppo. In base a questi principi sono nate poi diverse metodologie, ad esempio:

- Extreme Programming (XP)
- Scrum[36, 37]
- Adaptive Software Development
- Feature-Driven Development[21]
- Crystal Clear
- Dynamic Systems Development Method (DSDM)
- Lean Software Development[38, 39, 40]

Nel corso del tempo, grazie anche a questo nuovo approccio allo sviluppo, ci sono stati dei miglioramenti, illustrati nella tabella 2.2 e graficamente in figura 2.1.

Dai numeri si evidenzia sicuramente un miglioramento, ma notiamo anche che la percentuale di fallimenti è la più alta degli ultimi dieci anni. Secondo il rapporto CHAOS 2009[20], la ragione principale è ancora l'*eccessiva focalizzazione sui processi e sulle "cerimonie" e troppa poca sull'effettiva esecuzione e le buone pratiche (i fondamentali).*

[10] http://www.agilealliance.com/

	1994	1996	1998	2000	2002	2004	2006	2009
Successi	16%	27%	26%	28%	34%	29%	35%	32%
A Rischio	53%	33%	46%	49%	51%	53%	46%	44%
Fallimenti	31%	40%	28%	23%	15%	18%	19%	24%

Tabella 2.2: Dati dello Standish Group negli anni

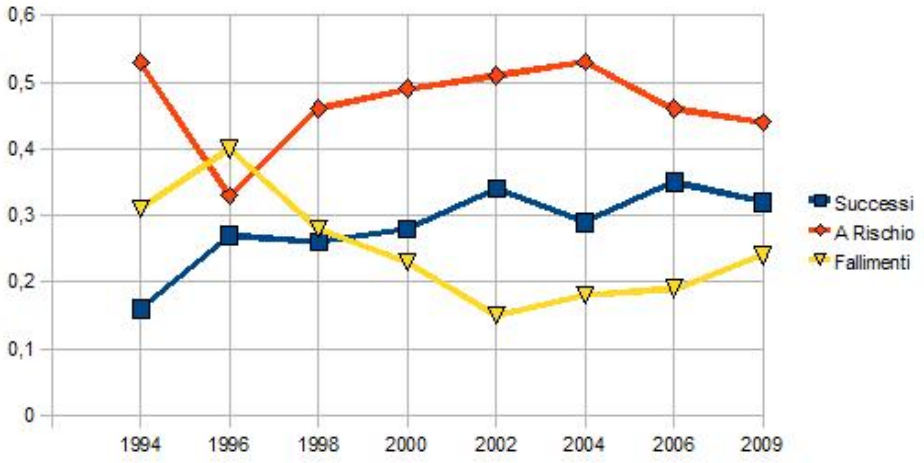


Figura 2.1: Grado di successo dell'industria negli anni secondo lo Standish Group

Persino uno dei pionieri dell'originale visione dell'ingegneria del software, Tom DeMarco, ha ammesso[23] che l'enfasi posta inizialmente sul controllo (processi e ruoli) e sulle metriche era eccessiva e che invece vanno *"gestite le persone e monitorati tempi e costi"*, attraverso una metodologia che ricalca i principi dell'agile.

Come se ciò non fosse abbastanza, lo Standish Group in ogni suo rapporto elenca i fattori di successo che hanno contribuito alla realizzazione del software che è stato consegnato entro i tempi e i costi preventivati e che soddisfaceva il cliente. Diamo un'occhiata alla classifica dei fattori di successo dal rapporto del 2006, riportati in figura 2.2: vediamo che tra i primi posti figura un processo di tipo agile.

Infatti, un'altra indagine della Indutrial Logic quantifica i miglioramenti apportati dalle metodologie agili per quanto riguarda aspetti come: riduzione dei costi, dei tempi, del personale richiesto e di difetti. Vedi figura 2.3.

L'Agile sicuramente non è la risposta definitiva (il *silver bullet*) al problema del *software development*, ma è indubbiamente un passo in avanti rispetto al passato. E' sempre bene essere pragmatici e mantenersi aggiornati (vedi sezione 4.6) per sapere in che direzione l'industria si sta muovendo, in modo

Figura 2.2: Fattori di successo nel 2006 secondo lo Standish Group

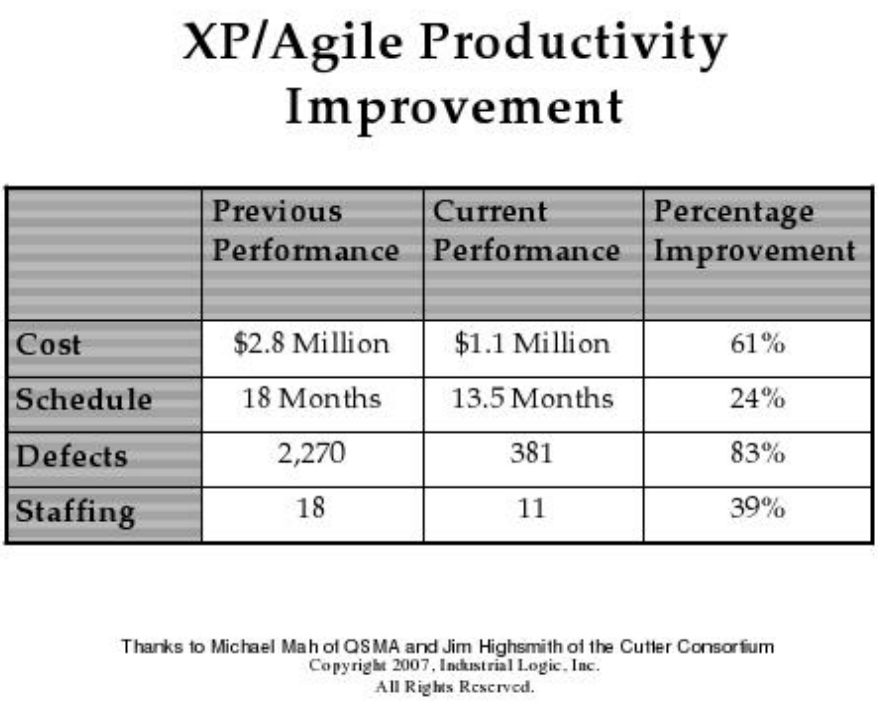
da poter approfittare prima degli altri di nuovi progressi al fine di acquisire un vantaggio competitivo sui concorrenti.

2.2 Principi guida dell'Agile

Quelli che seguono, sono i principi alla base dell'*Agile Manifesto*[22] come espressi dai suoi creatori nella *pagina*[11] del sito corrispondente.

> Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
>
> Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

[11] http://agilemanifesto.org/principles.html



XP/Agile Productivity Improvement

	Previous Performance	Current Performance	Percentage Improvement
Cost	$2.8 Million	$1.1 Million	61%
Schedule	18 Months	13.5 Months	24%
Defects	2,270	381	83%
Staffing	18	11	39%

Thanks to Michael Mah of QSMA and Jim Highsmith of the Cutter Consortium

Figura 2.3: Miglioramenti introdotti dall'agile

Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

Business people and developers must work together daily throughout the project.

Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

Working software is the primary measure of progress.

Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

Continuous attention to technical excellence and good design enhances agility.

Simplicity –the art of maximizing the amount of work not done– is essential.

> The best architectures, requirements, and designs emerge from self-organizing teams.
>
> At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Riassumendo, le caratteristiche fondamentali dell'approccio agile allo sviluppo del software sono:

- **Ciclo di vita iterativo e incrementale**, con release frequenti (ritmo che fornisce *feedback* frequente)
- **Adattamento costante ad un contesto sempre mutevole**
- **Miglioramento continuo**
- Semplicità
- Pragmaticità
- Efficienza ed efficacia
- Enfasi sulle persone
- Importanza delle buone pratiche e la loro esecuzione
- Importanza del *design* (continuo)
- Soddisfazione del cliente
- Professionalità ed ricerca dell'eccellenza
- Collaborazione tra sviluppatori, management e cliente
- Auto-organizzazione del team di sviluppo (no micro-management)

2.3 Ciclo di vita

Uno dei motivi principali dei molti fallimenti riscontrati negli anni sono i rischi connessi con i requisiti (come confermato dal rapporto CHAOS[19]), che nella maggior parte dei casi[12] non sono né fissi né chiari fin dall'inizio. Solitamente, invece, non sono chiari nemmeno al cliente e tendono a cambiare durante il corso dello sviluppo.

[12]Non sempre, ad esempio con progetti aerospaziali è possibile usare questo ciclo di vita, sebbene rimangano i difetti riguardanti l'impossibilità di quantificare le risorse che verranno impiegate per l'integrazione e il testing. Infatti persino la NASA[24] impiega lo sviluppo iterativo e incrementale per creare software complessi per il suo space shuttle.

Il software, a differenza alle controparti fisiche come veicoli e strutture architettoniche (la cui ingegnerizzazione si è tentato di emulare), nella maggior parte dei casi ha un costo di modifica più basso e una variabilità molto maggiore, tali da rendere impossibile qualsiasi formalizzazione completa e a priori dei requisiti. Nel caso ad esempio di un'auto, invece, i costi e le difficoltà di modifica sono talmente alti e soprattutto evidenti che è ragionevole e comprensibile stabilire ogni particolare a priori fino ad un buon livello di dettaglio.

Queste caratteristiche influiscono anche su come il cliente percepisce concretamente il costo: se io ho firmato il progetto di una casa di quattro stanze, sono cosciente che, a casa finita, mi terrò quelle stanze e sarà per me molto costoso cambiare idea. Nel software questa difficoltà, anche quando c'è, è molto difficile da percepire, per cui anche richieste in realtà costosissime (come ad esempio passare da un'architettura di tipo client/server classica ad una *web-based*) sono percepite dal cliente come banali. E accade frequentemente anche il contrario: modifiche percepite come difficili dal cliente sono in realtà molto semplici (ad esempio cambiare il colore di uno sfondo).

L'inadeguatezza del ciclo di vita sequenziale alla maggior parte dei progetti software, quindi, deriva proprio dalla falsa assunzione che si possano specificare completamente e definitivamente i requisiti.

Uno dei principi guida dell'agile è quindi l'adattamento ad un contesto mutevole raccogliendo presto e spesso *feedback* da una varietà di fonti (base di codice, sviluppatori, management, clienti, ecc). Questo *feedback* viene utilizzato essenzialmente per ridurre i rischi connessi ai requisiti, adattarsi ad un contesto in continuo mutamento e per permettere il *miglioramento continuo*. Di conseguenza, il ciclo di vita tipicamente impiegato da un metodo di lavoro di tipo agile è quello **iterativo e incrementale**.

In breve, lo sviluppo secondo questo ciclo di vita è organizzato attorno alle *features*, che vengono realizzare in modo *iterativo*[13] producendo continuamente degli *incrementi* nelle funzionalità del software che si sta realizzando.

2.3.1 Gestione per funzionalità

L'approccio "tradizionale" all'ingegneria del software vede la definizione precisa di ruoli, processi, procedure e attività che vengono eseguite di norma in sequenza, dando vita al ciclo di vita *Waterfall*. Questo ciclo di vita è adatto ad alcuni progetti (come ad esempio le applicazioni aerospaziali) in cui i requisiti non cambiano e sono estremamente dettagliati. Si tratta di un esempio di **gestione per attività**, nella quale queste vengono eseguite in

[13] Tutte le attività dello sviluppo vengono eseguite per ogni funzionalità, le quali vengono realizzate in sequenza.

modo sequenziale e in parallelo per tutti i requisiti. Uno schema di questo tipo di gestione dello sviluppo è mostrato in figura 2.4.

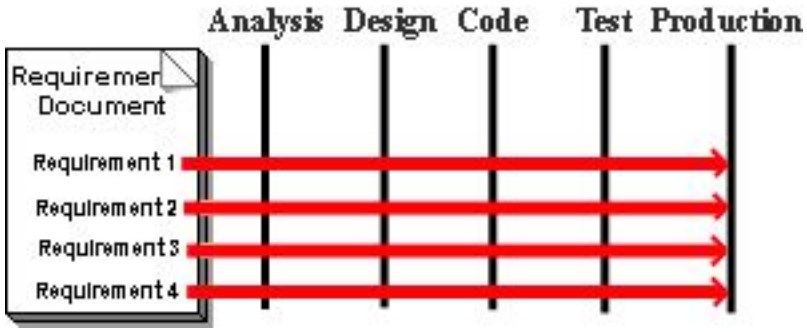


Figura 2.4: Gestione dello sviluppo per attività. Fonte: [21]

Nel mondo del software di tutti i giorni, però, si ha a che fare con prodotti sempre nuovi e clienti che non hanno quasi mai le idee abbastanza chiare su quello di cui hanno veramente bisogno. Quindi è opportuno non perdere tempo a cercare di costruire una specifica completa e dettagliata del sistema, che rifletterà per forza di cose una comprensione parziale, incompleta e a volte errata dei requisiti.

> "*Il futuro è ciò che accade mentre siamo occupati a pianificarlo.*"
> –Proverbio ebraico

Piuttosto è utile fornire presto all'utente qualcosa di tangibile, in modo da ottenere del *feedback* con il quale rifinire il prodotto e quindi creare qualcosa che soddisfi pienamente le esigenze del cliente stesso (anche quando, come spesso succede, o il cliente non è in grado di fornire delle specifiche adeguate, o non è nemmeno cosciente dei propri reali bisogni).

Si tratta di **gestione per funzionalità** in cui la parte centrale è riservata alle *feature* richieste dal cliente, che vengono sviluppate in sequenza ed effettuando tutte le attività richieste alla sua realizzazione. Questo è consentito dalla capacità di auto-organizzazione del team che esegue tutte le attività come, quando e quanto richiesto a seconda del contesto (vedi figura 2.5).

In altre parole, invece di sviluppare un prodotto eseguendo le attività in modo sequenziale e realizzando le funzionalità in parallelo (in questo modo si otterrà un prodotto presentabile solo al termine dell'intero ciclo di sviluppo), si sviluppano le funzionalità in sequenza eseguendo le attività in parallelo. Il vantaggio principale è il poter implementare un ciclo di vita iterativo e incrementale, realizzando presto e spesso del software rilasciabile tramite il quale raccogliere *feedback* dal cliente.

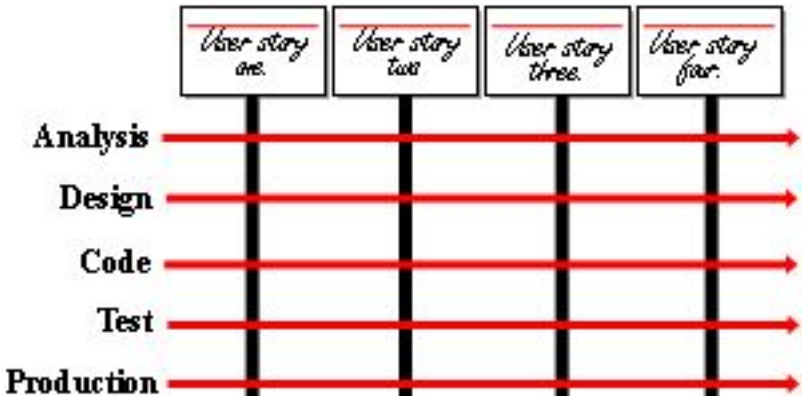


Figura 2.5: Gestione dello sviluppo per requisiti. Fonte: [21]

Questo approccio, oltre a facilitare il *planning* in un ciclo di vita iterativo e incrementale, mira a massimizzare le fonti di *feedback* con il quale correggere e migliorare i processi. In particolare, le attività di sviluppo che tradizionalmente nel ciclo di vita sequenziale erano effettuate in sequenza una sola volta per tutte le funzionalità, vengono invece distribuite lungo tutto il ciclo di sviluppo e svolte *continuamente*[14].

Ad esempio, le attività di *testing* e integrazione, nel ciclo di vita *Waterfall* erano svolte alla fine e costituivano dei grossi rischi, in quanto le richieste in termini di tempo e risorse non erano quantificabili. In un ciclo di vita agile invece, queste attività vengono svolte continuamente lungo tutto lo sviluppo, minimizzando i rischi e allo stesso tempo fornendo *feedback* prezioso per l'adattamento e il miglioramento continuo.

2.3.2 Planning

Nella pianificazione rivestono chiaramente un ruolo centrale le funzionalità, che vengono prioritizzate e realizzare iterativamente passando attraverso tutte le fasi del ciclo di vita (analisi dei requisiti, progettazione, codifica, testing, *deployment*) per ognuna di esse.

Vengono pianificate delle iterazioni (normalmente di lunghezza compresa tra 1 e 4 settimane) durante le quali vengono sempre realizzate le funzionalità più importanti. Al termine di ogni iterazione si dimostra quanto fatto al cliente, che ha la possibilità di fornire *feedback*, rettificare e aggiungere idee. Quindi al termine di ogni iterazione la lista di funzionalità può essere modificata e ri-prioritizzata. In questo modo si può fermare in qualunque momento lo sviluppo, con la consapevolezza che le funzionalità più importanti (per il cliente) sono state realizzate.

[14] Secondo l'approccio *poco&spesso*.

Con una certa frequenza inoltre, vengono effettuate delle *retrospettive* (vedi sezione 2.4.13) per migliorare i processi. Una schematizzazione di un'implementazione di questo ciclo di vita da parte della metodologia *Scrum*[36, 37] è mostrata in figura 2.6.

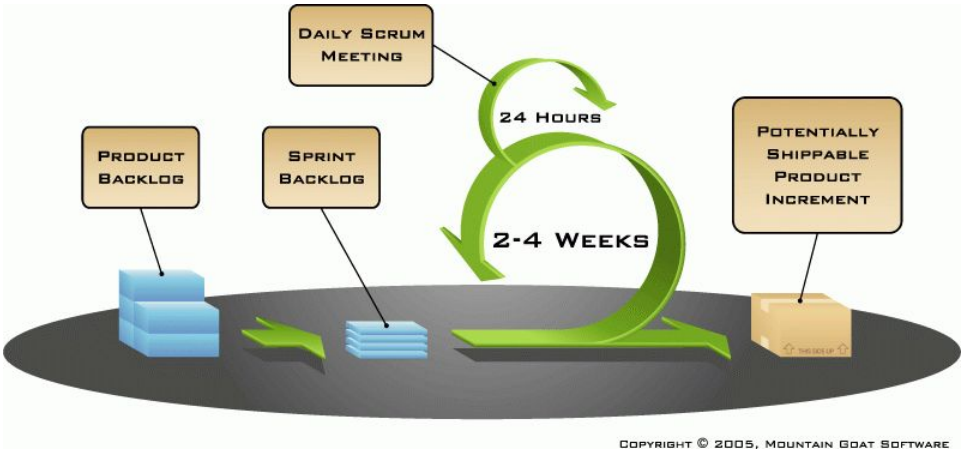


Figura 2.6: Ciclo di vita agile secondo Scrum

I vantaggi principali di una gestione dello sviluppo incentrato sulle funzionalità piuttosto che sulle attività sono i seguenti:

- Produce un prodotto che soddisfa le reali esigenze del cliente.
- Consente di rispondere con rapidità ai cambiamenti (priorità, *business values*, requisiti, ecc).
- Fornisce la possibilità di cambiare direzione nello sviluppo in un qualsiasi momento senza grandi costi.
- Permette un maggior controllo al cliente su quanto viene realizzato. Egli può sempre scegliere quali sono le prossime funzionalità che verranno realizzate. In questo modo, il costo ha più senso perché il cliente è in grado di decidere come sarà il software scegliendo le *feature* in modo incrementale in una sorta di catalogo (il *backlog*).
- Consente di poter fermare lo sviluppo quando si desidera con la sicurezza di aver comunque realizzato le funzionalità più importanti.
- Permette al cliente di prendere decisioni su quanto deve essere fatto in modo oculato e flessibile.
- Consente al cliente di tenere sempre sotto controllo quanto spende.
- Il cliente può vedere presto e spesso i progressi che vengono portati avanti (con i suoi soldi).

- Permette ai manager la pianificazione che massimizza in ogni momento il *business value* prodotto.
- Consente di non dover adottare il micro-management e di poter quindi realizzare piani più flessibili.
- Consente il miglioramento continuo dei processi.
- Stimola la professionalità e l'apprendimento consentendo l'auto-organizzazione del team di sviluppo.

In breve un ciclo di vita iterativo e incrementale, rispetto a quello classico *Waterfall*, consente maggiore libertà, flessibilità, sicurezza e qualità.

Per mantenere questo ritmo, è necessario affiancare a questo ciclo di vita le opportune pratiche per mantenere la base di codice sempre in salute (come: *TDD*, *continuous integration*, ecc. Vedi sezione 2.4) in modo da essere pronti in qualunque momento a supportare nuove funzionalità o modificare quelle preesistenti.

2.4 Pratiche

Quella che segue è una rapida spiegazione di alcune delle pratiche normalmente associate all'Agile ma nella maggior parte dei casi già esistenti prima della sua formulazione.

Per una trattazione più approfondita consultare la bibliografia a pagina 95, a cominciare da [45].

2.4.1 TDD

Questa pratica comprende il *test-first development* e il *refactoring* (vedi sezione 2.4.3). In breve lo sviluppo procede seguendo quello che viene definito il *Red-Green-Refactor Cycle*, illustrato in figura 2.7 (fonte: *The Agile In a Flash*[15]).

In poche parole:

1. Prendere una funzionalità che il sistema (o un suo componente) deve esibire, e scrivere un test utilizzando tale funzionalità (che ancora non esiste). Poi eseguire il test e verificarne il fallimento (fase *Red*).

[15] http://agileinaflash.blogspot.com/2009/02/red-green-refactor.html

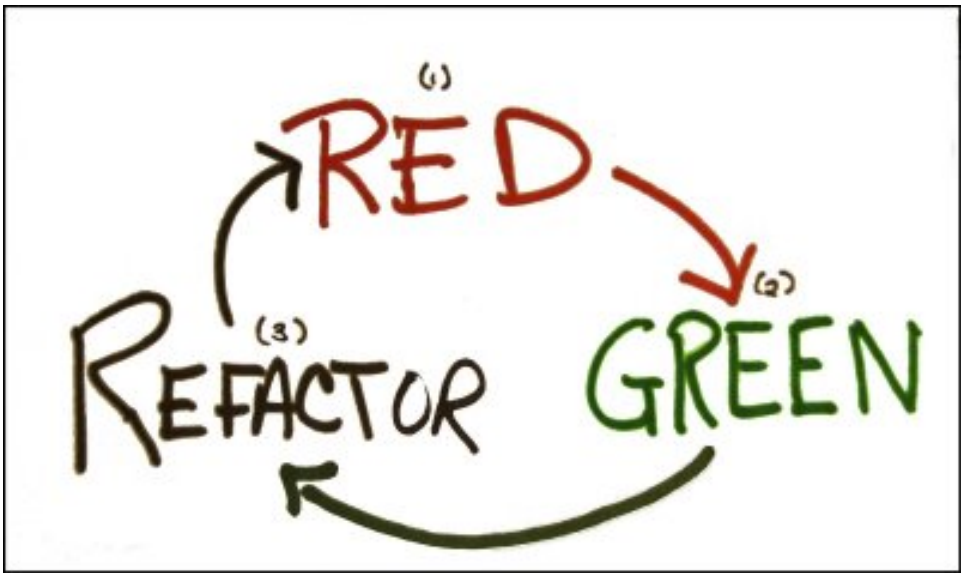


Figura 2.7: Ciclo red-green-refactor del TDD

2. Scrivere il codice strettamente necessario per far passare quel test, né più né meno[16] (fase *Green*)
3. Se necessario, eliminare la duplicazione tramite il *refactoring* (vedi sezione 2.4.3) (fase *Refactor*)

A tutti gli effetti, quindi, si utilizza questa pratica per guidare il *design* e creare una *suite* di test di regressione utile per supportare i probabili cambiamenti futuri e la manutenzione.

I vantaggi sono molteplici:

- Il TDD forza a scrivere molti test, e scrivere più test rende gli sviluppatori più produttivi[25]
- La necessità di effettuare il *debugging*[17] è praticamente eliminata.
- Lo scrivere i test prima del codice vero e proprio a partire dagli *use case* (o dalle *user stories*, vedi sezione 2.4.8) forza a mettersi nei panni dell'*utente* di tale codice focalizzandosi così sull'interfaccia. Il che produce dei sistemi con delle API più usabili e modulari.
- I test costituiscono una fonte molto efficace di *feedback* che consente di catturare molto presto gli errori.

[16] Seguendo il principio *YAGNI*: "You Aren't Gonna Need It" ovvero "Non ne avrai bisogno".

[17] Che è un'attività spesso lunga, tediosa, frustrante e di entità non prevedibile.

- Avere una *suite* di test di regressione consente di poter effettuare la manutenzione e la modifica del software riducendo molto la possibilità di introdurre nuovi *bug*.

- Sebbene la scrittura dei test comporti[26] un aumento del tempo necessario allo sviluppo mediamente del 15-20%, la densità dei difetti si riduce del 40-90% rispetto a progetti simili che non impiegano il TDD. Ma considerando anche il tempo richiesto per eliminare quel 40-90% di difetti in più, secondo un altro studio[27] tipicamente lo sviluppo di un progetto è *più rapido* utilizzando il TDD. Vedi anche la figura 2.3.

- Permette di realizzare sistemi più modulari, flessibili ed estendibili perché spinge gli sviluppatori a pensare al software in termini di piccole unità da sviluppare e testare separatamente. Il risultato sono classi (se si utilizza il paradigma OO[18]) che rispetta i principi della buona programmazione ad oggetti (ovvero quelli che Robert C. Martin definisce *SOLID*[19]).

- Dato che il codice viene scritto solo se ci sono dei test che falliscono evidenziandone la necessità, il TDD consente di avere una percentuale di copertura dei test molto alta, aumentando di conseguenza la sicurezza nel buon funzionamento del software.

- I test costituiscono a tutti gli effetti degli esempi di utilizzo dei componenti del sistema, e costituiscono quindi una vera e propria *documentazione eseguibile*.

Per una trattazione esaustiva e chiara di questa pratica, consultare [3].

2.4.2 BDD

Il valore principale del TDD non sta tanto nel *testing* (pur essendo un importante "effetto collaterale"), quanto nella sua capacità di guidare il *design* di un sistema software in un modo che ne permetta la crescita, la manutenzione e la flessibilità nel modo più efficiente possibile.

Nel 2003 Dan North si accorse che le difficoltà che molti riscontravano nell'applicazione del TDD, coscientemente o meno, derivavano dall'eccessiva focalizzazione sull'aspetto di *testing*. Le difficoltà erano causate principalmente da una prospettiva errata, e impedivano di ottenere tutti i benefici che la pratica consente.

[18] *Object Oriented*.
[19] http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

Il *Behaviour-Driven Development*[28] è nato inizialmente come un *re-branding*[20] del TDD per insegnare correttamente la sua applicazione. Per enfatizzare il suo uso per la progettazione, si è sostituita la parola *test* con *specifica*. Grazie ad un semplice cambiamento di vocabolario[29] (punto di vista), domande che sorgevano in chi non aveva capito come applicare la tecnica, come ad esempio:

- Che cosa testo ora?
- Come faccio a testare questo componente?

Trovano "magicamente" risposta spostando l'attenzione dai test al *comportamento* che si desidera il sistema esibisca. Quelli che prima erano i test, ora sono delle *specifiche di comportamento*.

Successivamente, nel corso degli anni il BDD si è evoluto in una metodologia vera e propria, includendo l'*ATDP*[21] e il *DDD*[22]. Ai fini pratici, l'uso dei test di accettazione (automatizzati) per guidare il *design* nello stesso modo in cui si usano i test di unità comporta l'aggiunta di un ciclo a livello più alto di quello del TDD, raffigurato in figura 2.8.

In breve:

1. Si parte con una *feature* da implementare (contenuta in una *user story* ad esempio, vedi sezione 2.4.8) e si scrive un test di accettazione per quella storia sotto forma di *scenario*.
2. La specifica di questo scenario (che è anche un test) deve fallire (fase *Red* del ciclo più esterno). Principalmente in questa fase e nella successiva si prendono decisioni di *design* sul sistema.
3. Si passa nel ciclo più interno (quello del TDD) e si scrive un *esempio*[23] di utilizzo del codice a livello più granulare (fase *Red* del ciclo del TDD).
4. Si scrive il minimo codice necessario per far passare l'esempio (fase *Green* del TDD).
5. Eventualmente, rimuovere la duplicazione introdotta (fase *Refactor* del TDD).

[20] Il BDD viene anche definito: "il TDD fatto come si deve."
[21] *Acceptance Test Driven Planning*, nel quale i test di accettazione vengono utilizzati per sviluppare il software allo stesso modo dei test di unità.
[22] *Domain-Driven Design*, che (detto molto superficialmente) si basa sull'uso di un linguaggio condiviso tra sviluppatori e *business*.
[23] L'equivalente di un test di unità nel TDD.

6. Eseguire lo scenario e verificarne il successo (fase *Green* del ciclo esterno).
7. Eventualmente effettuare del *refactoring*.
8. Continuare con il successivo scenario. Se la *feature* è completa, passare alla successiva.

Per maggiori dettagli sul *Behaviour-Driven Development*, consultare [4].

2.4.3 Refactoring

La pratica del *refactoring*, secondo Martin Fowler si definisce come:

> *Modificare la struttura di un software mantenendone inalterate la funzionalità.*

In altre parole si tratta di eliminare la duplicazione e apportare tutte le modifiche del caso per rendere il programma più semplice, flessibile e facile da capire, assicurandosi al tempo stesso che passi tutti i suoi test. Quindi è necessaria una buona *test suite* per poter effettuare il *refactoring* con sicurezza.

E' una pratica necessaria[49] per mantenere la base di codice in salute e flessibile (in grado di supportare modifiche successive e nuove funzionalità) mano a mano che la comprensione del dominio del problema aumenta nel corso del progetto.

Chiaramente si tratta di qualcosa di fondamentale per l'agilità, senza il quale il costo di modifica salirebbe rapidamente in modo esponenziale e la base di codice con l'andare del tempo diventerebbe sempre più caotica, trasformandosi velocemente in quella che viene definita "*a big ball of mud*"[30].

2.4.4 Continuous Integration

Con il termine *integrazione continua* si indica il processo di integrare con *frequenza* nella base di codice le modifiche a cui ogni sviluppatore lavora, eseguendo l'intera *test suite* per verificarne il corretto funzionamento. Laddove in un ciclo di vita sequenziale l'integrazione si effettuava in una fase apposita al termine dello sviluppo (fase la cui durata e costo non erano prevedibili), con questa pratica viene distribuita lungo tutto il ciclo di sviluppo facendola spesso e con costanza. In questo modo il costo associato all'integrazione praticamente si annulla.

Trattandosi di un processo prettamente meccanico, di solito lo si effettua in modo automatico con l'ausilio di un server di *continuous integration* che preleva frequentemente l'ultima versione del software[24], la compila e esegue i test, notificando[25] immediatamente eventuali fallimenti.

Naturalmente è possibile utilizzare un server di *continuous integration* anche senza test, nel qual caso però i benefici sono grandemente ridotti perché sarà in grado di notificare solo errori di compilazione.

L'utilità principale è quella di scoprire eventuali problemi con tempestività, così da poterli sistemare in modo relativamente economico invece che aspettare che diventino sempre più grandi e costosi. E' sempre meglio scoprire subito un *bug* piuttosto che in produzione.

Quando ci sono diversi sviluppatori a lavorare al progetto è possibile che si "pestino i piedi a vicenda" e le modifiche che hanno apportato al sistema entrino in qualche modo in conflitto, o che si presentino dei problemi quando il codice viene compilato o eseguito su una macchina diversa da quella del programmatore che l'ha prodotto[26], in questi casi l'integrazione continua offre un importante e tangibile aiuto.

E' necessario che la base di codice venga tenuta sempre in salute attraverso pratiche come il TDD e il *refactoring* e che nel *repository* di riferimento ci sia sempre un versione funzionante (con l'intera suite di test che passa), in modo da minimizzare le notifiche dei fallimenti da parte del server di CI[27]. Diversamente, se gli esiti negativi diventano un'abitudine cominceranno ad essere ignorati dagli sviluppatori e in breve si avrà un software fuori controllo. Lo stesso discorso si applica anche alla *suite* di test ovviamente. Una caratteristica importante dell'integrazione continua quindi è che funziona da *enforcement* della politica di software sempre rilasciabile.

In breve i vantaggi principali sono:

- Il testing è automatico, non dipende da interventi manuali.
- I fallimenti vengono notificati tempestivamente, non ci sono sorprese.
- *Enforcing* della *policy*: codice sempre rilasciabile.

[24] L'uso di un sistema di SCM (*Source Code Management*) è indispensabile, punto.

[25] Naturalmente è possibile scegliere a chi inviare le notifiche, ed è bene restringerle agli interessati.

[26] Troppo spesso si sente la scusa: "ma da me funziona!" Una variante della CI comprende anche il *deployment* del prodotto che viene compilato e testato sulle piattaforme *target* per verificarne il corretto funzionamento in ogni ambiente supportato.

[27] Acronimo di *Continuous Integration*

Riassumendo, grazie alla CI i problemi di integrazione vengono minimizzati ricostruendo il sistema molte volte al giorno, così che i problemi sono risolti quando si presentano invece di accumularsi alla fine dell'iterazione.

2.4.5 Collective Code Ownership

Si tratta della possibilità per ogni sviluppatore di leggere e modificare ogni parte della base di codice in ogni momento. In altre parole si condivide il diritto di modifica sul codice insieme alla *responsabilità* di assicurarsi che ogni cambiamento funzioni come deve. Questo è reso possibile dalla pratica abilitante del *testing automatizzato* (in particolare dal BDD/TDD).

I vantaggi sono:

- Una maggiore familiarità con tutte le aree del codice per ogni programmatore.
- La possibilità di migliorare il software in modo rapido ed efficiente evitando colli di bottiglia (che si hanno quando a conoscere bene una certa parte del sistema è una sola persona, e questa può non essere disponibile quando si rivela necessario).
- Stimola la produzione di codice di migliore qualità e più leggibile.

Anche se non è necessario che ognuno conosca e comprenda ogni singola linea di codice, *una buona conoscenza della struttura e del funzionamento dell'applicazione è essenziale.* Quindi il progetto deve essere opportunamente **modulare** e ogni componente del team deve avere una comprensione ad alto livello del funzionamento di ogni modulo e di come questo interagisce con il resto dell'applicazione.

2.4.6 Code Reviews

Una *code review* è una sessione di *analisi statica*[28] di codice da parte di uno sviluppatore diverso da quello che lo ha scritto. Tipicamente l'analisi è ristretta, secondo certi criteri, a parti specifiche dell'applicazione.

Questo è un incentivo ulteriore per ogni sviluppatore a scrivere il miglior codice possibile in modo che sia leggibile, elegante e funzionante (altri programmatori leggeranno quanto ho scritto).

L'efficacia dell'analisi statica nella rimozione di difetti è assodata:

[28] Ovvero la lettura del codice alla ricerca di errori comuni ed "esecuzione mentale" per trovare quelli meno ovvi.

> "*Formal code inspections are about **twice** as efficient as any known form of testing in finding deep and obscure programming bugs and are **the only known method to top 80 percent in defect-removal efficiency**.*"
> –Caper Jones in *Estimating Software Costs*

Di norma chi revisiona il codice dispone di una *lista di controllo* (tipicamente non-scritta, ma per i novizi può essere utile averla nero su bianco) contenente i particolari da controllare. Un elenco minimale potrebbe essere:

- Tutti i gestori di eccezioni sono non vuoti?
- Tutte le operazioni su database sono eseguite all'interno di transazioni?
- Posso leggere e comprendere il codice?
- Ci sono errori ovvi?
- Il codice avrà qualche effetto indesiderato sulle altre parti dell'applicazione?
- C'è del codice duplicato (nella stessa sezione o in altre parti del software)?
- Ci sono dei miglioramenti o dei *refactoring* ovvi che è possibile fare per migliorarlo?

Sebbene sia possibile pianificare a piacere delle sessioni di revisione del codice, l'approccio agile suggerisce di distribuire queste revisioni durante tutto il processo di sviluppo (approccio "poco&spesso", in modo incrementale e iterativo).

Non sempre è possibile applicare questa pratica in modo formale e continuativo, sebbene sia una delle misure più efficaci per la rimozione di *bug* (e quindi nel minimizzare la manutenzione e massimizzare il tempo da poter dedicare a nuovi progetti). Qualora non fosse possibile è comunque vantaggioso prevedere delle sessioni di revisione informali e irregolari, in cui concentrarsi sulle parti del prodotto più complicate e che con maggiori probabilità contengono errori.

Se invece fosse possibile introdurre questa pratica in modo regolare, gli stili principali con cui viene fatto sono definiti: "The Pick-Up Game" e *Pair Programming*.

The Pick-Up Game

Non appena il codice per una determinata funzionalità o *task* è stato scritto e testato ed è pronto per essere immesso nel sistema di versionamento, viene preso e revisionato da un altro sviluppatore (è utile ruotare le persone che revisionano il codice di un certo sviluppatore). E' una tecnica molto efficace.

Queste *commit reviews* sono revisioni veloci e informali, sono utili a controllare che il codice sia pronto (abbastanza buono) per essere inserito nel ramo principale di sviluppo.

2.4.7 Pair Programming

E' una tecnica popolarizzata dall'*Extreme Programming* e prevede che due persone lavorino sullo stesso computer: una scrive codice (*driver*) e la seconda (*navigator*) revisiona il codice in *real-time*. A intervalli regolari, i ruoli si scambiano. Le coppie non sono fisse e si formano giorno dopo giorno a seconda del *task* e delle disponibilità.

Il *pair programming* permette quella che si può definire *continuous code review* perché mentre il *driver* è concentrato su dettagli a basso livello, il *navigator* può pensare al quadro generale, fornire prospettive differenti e controllare che il codice che viene scritto non contenga errori ovvi o introduca duplicazione.

Un valore aggiunto di questa pratica è che promuove nel modo più efficace lo scambio di conoscenze e il *mentoring*. In effetti è uno dei metodi migliori per addestrare un nuovo sviluppatore a lavorare nel team, insegnandogli i processi e le tecniche che dovrà adottare.

2.4.8 User Stories

La tecnica delle *user stories*[5, 54, 55] (storie) è un approccio *lightweight* alla gestione[29] agile dei requisiti. Sono piccole *card* (fisiche o virtuali) che contengono una frase o due ed esprimono un requisito (abbastanza piccolo da poter essere implementato da un paio di persone in una singola iterazione).

La *storia* è l'unità fondamentale di lavoro che è visibile sia dall'interno che dall'esterno del progetto. Essa ha valore per l'utente finale ed è solitamente descritta nel linguaggio del *business domain*, inoltre rappresenta una funzionalità abbastanza piccola da poter essere stimata a grandi linee in termini di tempo richiesto per la sua realizzazione.

[29] Raccolta, prioritizzazione e sviluppo.

Le caratteristiche fondamentali di una storia, e quindi di una funzionalità adatta ad essere implementata in un ciclo di vita iterativo e incrementale, sono:

- Devono fornire *business value*.
- Devono essere *stimabili*, quindi il team deve avere abbastanza informazioni per poter azzardare una stima sulla complessità e le tempistiche.
- Deve essere sufficientemente piccola da poter essere implementata in una iterazione, diversamente va ulteriormente partizionata.
- Deve essere *testabile*, ovvero i test automatici e/o manuali necessari per dichiararla implementata devono essere chiari.

L'intento delle *user stories* ovviamente non è quello di raccogliere esaustivamente i requisiti (cosa che comunque è quasi impossibile, vedi sezione 2.3) ma sono più che altro uno strumento di pianificazione. Le *card* che si utilizzano sono più un promemoria per una conversazione con il cliente che una specifica.

Di solito le *user stories* vengono raggruppate e ordinate per priorità in un *backlog* (vedi sezione 2.4.9). Le iterazioni e le *release* vengono pianificate a partire dal *backlog* distribuendo le storie, e per l'implementazione di ognuna di esse si può procedere ad esempio con il *BDD* scrivendo gli scenari e gli esempi per guidare lo sviluppo.

Le storie sono il mezzo principale con il quale il cliente può influenzare lo sviluppo del progetto, modificando le priorità al termine di ogni iterazione.

Formato

Non esiste un formato standard trattandosi di uno strumento piuttosto semplice, ma nel corso del tempo se ne è sviluppato uno "di fatto" che mette in evidenza gli elementi fondamentali e il punto di vista corretto (come il BDD insegna): l'utente, il suo ruolo, e ciò che vuole per ottenere un certo beneficio. Il formato è conosciuto con il nome di *Connextra* ed è mostrato in tabella 2.3.

As I [*role*]
I want [*something*]
so that [*benefit, outcome*]

Tabella 2.3: Formato Connextra di una user-story

Vantaggi

La tecnica delle user-stories è una tecnica agile perché non è eccessivamente burocratica e permette di adattarsi rapidamente a cambiamenti nei requisiti. I vantaggi principali sono:

- Sono molto piccole: rappresentano una parte di funzionalità del sistema che possono essere implementate in un periodo che va da pochi giorni a poche settimane.
- Consente agli sviluppatori e al cliente di discutere i requisiti lungo tutto il corso di sviluppo del progetto.
- Hanno bisogno di molto poca manutenzione.
- Devono essere considerate solo al momento dell'uso.
- *Permettono di mantenere uno stretto contatto con il cliente*, il che è uno dei fattori fondamentali per il successo di un progetto (vedi figura 2.2).
- Consentono di spezzare il progetto in piccoli incrementi (essenziale per un ciclo di vita iterativo e incrementale).
- Adatte a progetti nei quali i requisiti sono volatili o poco chiari, alcune iterazioni di esplorazione guidano il processi di rifinitura.
- Possono rendere più semplice la stima dello sforzo necessario.

2.4.9 Backlog

La definizione di *backlog* è: *"an evolving, prioritized queue of business and technical functionality that needs to be developed into a system."*

Quindi si tratta di una lista prioritizzata di *user stories*[30] utile alla pianificazione di iterazioni e *release*.

2.4.10 Metriche

Le metriche possono essere un importante aiuto per ottenere informazioni sullo sviluppo altrimenti difficilmente visibili.

Una delle metriche più importanti per quanto riguarda lo stato di avanzamento del progetto è la *RTF*[53] (*Running-Tested Features*), la quale indica

[30] O funzionalità, o requisiti, come si preferisce definirle.

il numero di funzionalità attualmente realizzate, testate e funzionanti. L'aspetto interessante di questa metrica è che per massimizzarla occorre agilità, quindi può funzionare anche da misura del grado di agilità del team.

Burn-Down Chart

Trattasi di un grafico con in ascissa il tempo (usualmente in giorni) e in ordinata una stima della quantità di lavoro necessaria per terminare il progetto. Normalmente, le stime vengono fatte utilizzando delle unità di misura arbitrarie (ad esempio gli *Story Points*) per quantificare la difficoltà *relativa* delle storie in modo da facilitare la pianificazione.

Giorno dopo giorno, viene calcolata la velocità media[31] che tramite una proiezione permette di stimare il giorno in cui lo sviluppo verrà terminato. Mano a mano che il lavoro procede, la proiezione diventerà sempre più precisa. Per un esempio, vedi figura 2.9.

2.4.11 Daylog

Il *daylog* non è altro che un luogo in cui ogni sviluppatore può annotare i problemi che incontra durante il suo lavoro e le soluzioni che trova per risolverli. Disporre di uno strumento del genere consente di non dover risolvere più volte lo stesso problema, risparmiando tempo quando e se si dovesse ripresentare.

L'utilità del *daylog* è ancora maggiore quando questo è condiviso da tutti gli sviluppatori, costituendo una vera e propria *knowledge base*.

2.4.12 Stand-up meetings

Per il successo di un progetto la comunicazione è essenziale: non sono con il cliente, ma anche tra gli sviluppatori per rimanere al corrente dello stato di avanzamento del progetto.

Gli *stand-up meetings* sono appunto dei brevi *meeting* tra sviluppatori in cui ci si aggiorna sullo stato di avanzamento del progetto e su eventuali problemi da affrontare.

Il *format* prevede che venga dato un tempo massimo per ogni persona (circa due minuti), durante i quali risponderà alle seguenti domande per mantenere l'intervento focalizzato:

[31] Tramite una metrica chiamata *velocity* che indica la velocità con la quale il *team* è in grado di sviluppare il progetto in numero di *story points* al giorno. Normalmente, viene calcolata automaticamente da qualche software.

- Che cosa ho fatto ieri?
- Cosa pianifico di fare oggi?
- Quali difficoltà ho incontrato?

Questi *meeting* per essere utili devono essere corti, e per impedire che durino troppo vengono fatti *in piedi* (da qui lo "stand-up" nel nome). La durata ottimale è tra 10-15 minuti fino a un massimo di 30, se ci sono discussioni più approfondite da fare le si fa al di fuori dello *stand-up meeting* e solo con gli interessati.

Tipicamente vengono programmati quotidianamente, ma per team piccoli fare uno stand-up meeting ogni giorno può essere troppo, in questi casi si può pensare di farne un giorno si e uno no o un paio di volte a settimana.

Al meeting partecipano e hanno parola solo i componenti del team (gli sviluppatori ed eventualmente il cliente). E' utile avere un *coordinatore* (può essere uno dei componenti del team a rotazione) che si assicura che i tempi e il *focus* sia rispettato, inoltre può annotare i problemi che possono emergere.

I vantaggi degli stand-up meetings sono molteplici:

- Aiutano a partire focalizzati sull'obiettivo del giorno.
- Se una persona ha un problema, ha occasione di portarla all'attenzione del team e ricevere aiuto.
- Aiutano ad avere un'idea dello stato di avanzamento del progetto.
- Evidenziano delle eventuali aree in cui ci sono difficoltà e potrebbero essere necessarie teste aggiuntive.
- Accelerano lo sviluppo promuovendo la condivisione di codice e idee.
- Incoraggiano la produttività e motivazione: vedere gli altri che riportano dei progressi motiva a fare lo stesso.

2.4.13 Project Retrospectives

Essenziale per il miglioramento continuo è la raccolta e l'analisi del *feedback* per rilevare eventuali problemi e ottimizzazioni da adottare.

Le *retrospettive* sono dei brevi *meeting* che si effettuano al termine di ogni iterazione o *release*[32], il cui scopo è analizzare ciò che è stato fatto alla ricerca di possibili miglioramenti.

[32] O *milestone* se si preferisce.

Il *format* è simile a quello degli *stand-up meetings*, nel quale però ci si concentra sui fattori più ad alto livello. Le domande a cui si vuole trovare risposta sono:

- Cosa ha funzionato?
- Cosa ha bisogno di miglioramenti?
- Che cosa non funziona?

2.4.14 Coding Standards

L'uso di standard di codifica (principalmente per quanto riguarda lo stile, l'indentazione, l'uso di commenti e le convenzioni nel nominare variabili, classi e metodi) promuove la leggibilità e la collaborazione tra sviluppatori.

2.4.15 40-Hours Workweek

Il lavoro dello sviluppatore è un'attività cognitiva piuttosto impegnativa, che richiede molta creatività e disciplina. L'energia mentale è una risorsa limitata che va ripristinata attraverso una quantità sufficiente di *riposo*. Se non si dà modo ai programmatori di ripristinare le proprie energie, la stanchezza si fa sempre più presente fino a pregiudicare la produttività e a *degradare sensibilmente la qualità del codice prodotto*. In breve, uno sviluppatore esausto, per quanto in gamba, finirà con l'introdurre più *bug* di quanti il team è in grado di risolvere.

La misura più importante per evitare il *sovraccarico* è limitare le ore di lavoro a 40 alla settimana, 8 al giorno per 5 giorni. Anche se si adotta un ambiente di lavoro di tipo *ROWE* (vedi sezione 4.2), conviene rispettare comunque il limite massimo di 40 ore settimanali per evitare la degradazione di produttività.

Ci possono essere degli straordinari, ma come tali dovrebbero essere qualcosa di *straordinario* non la regola.

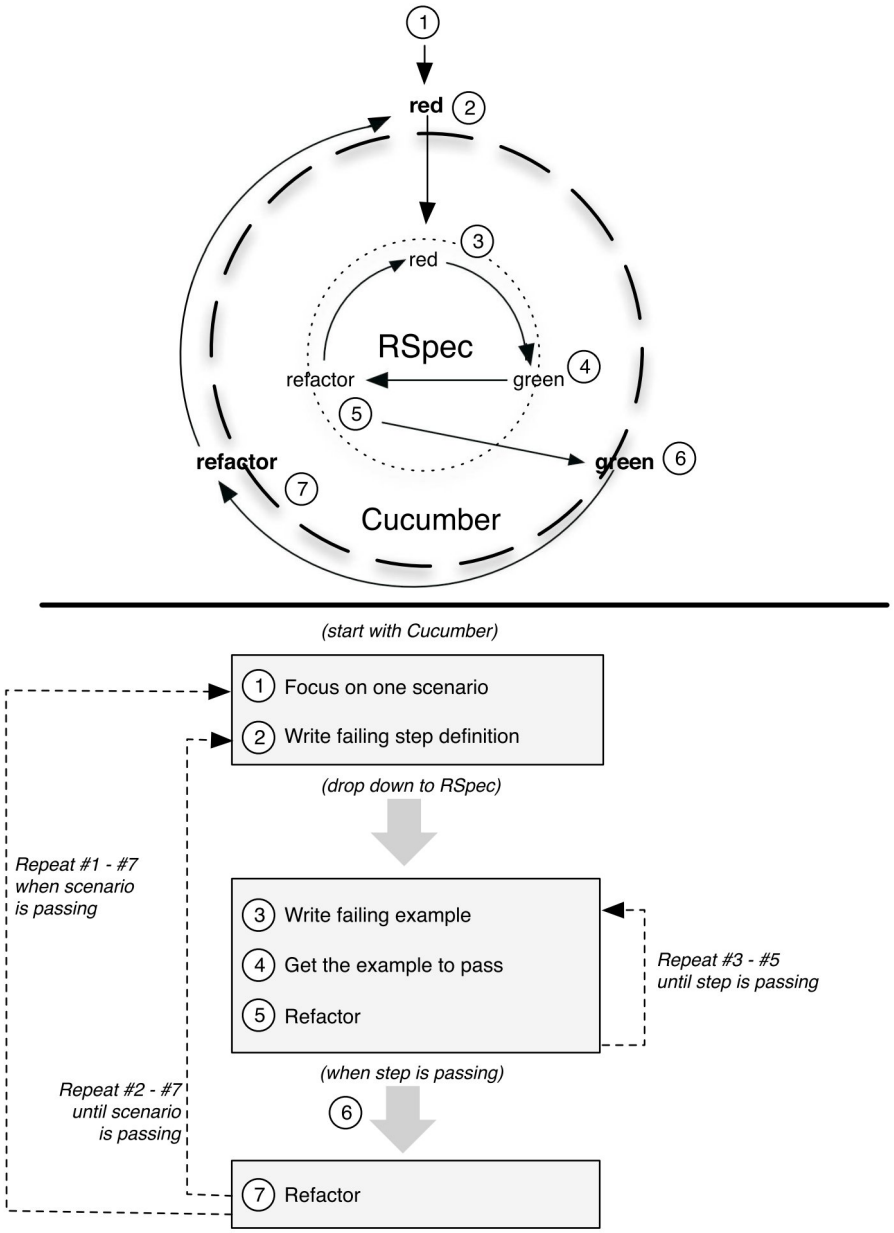


Figura 2.8: Ciclo del BDD - Fonte:[4]

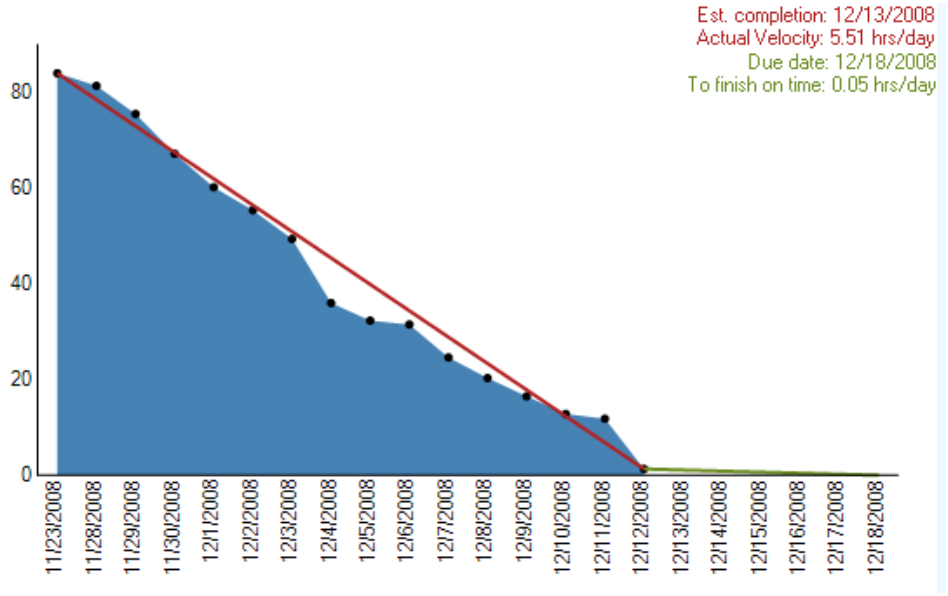


Figura 2.9: Esempio di burn-down chart

Capitolo 3

Assessment

La prima cosa da fare in ambito medico prima di prescrivere delle cure è quella di effettuare una diagnosi e chiarire il quadro clinico del paziente, accertandosi di eventuali allergie, intolleranze e particolarità. Questo è necessario per evitare di somministrare delle cure inadatte e perfino dannose al paziente. Alle stesso modo, prima di elaborare e proporre miglioramenti in una realtà aziendale, occorre procedere con un *assessment*.

I motivi principali per i quali un'azienda può voler migliorare il proprio modo di lavorare sono molti. Ad esempio possono riguardare: l'efficienza, la riduzione dei costi, la qualità dei prodotti, la soddisfazione dei clienti, cambiamenti dell'assetto organizzativo, ecc. E' necessario essere coscienti di questi obiettivi per poter proporre dei cambiamenti in linea con essi.

L'*assessment* ha lo scopo di raccogliere informazioni sulla particolare realtà aziendale ed elaborare un quadro generale dello *status quo*, in modo da poter intervenire in modo efficace. Le informazioni principali da raccogliere sono in generale:

- Obiettivi dell'azienda e *business values*
- Contesto in cui si trova ad operare l'organizzazione
- Processi attualmente in uso
- Problemi e ostacoli allo sviluppo
- Competenze del team

3.1 Teoria

Quella che segue è una breve introduzione agli elementi "teorici" di un processo di assessment, per come sono stati compresi e applicati da chi scrive.

3.1.1 Competenze

L'assessment, per essere efficace, deve essere eseguito da una persona che possegga determinati requisiti:

- **Dovrebbe essere esterno all'azienda**: una persona che lavora all'interno dell'azienda da diverso tempo con molta probabilità è "assuefatta" al metodo di lavoro attuale. Inoltre è invischiata negli aspetti organizzativi e "politici" dell'azienda stessa, che diventano un ostacolo alla sua libertà di proporre cambiamenti di una certa importanza, in quanto non è un agente imparziale. Sarebbe in grado di evidenziare problemi immediati che riguardano direttamente il proprio lavoro, ma dato il punto di vista è più difficile giungere ad una visione generale equilibrata e completa.
- **Pragmaticità**: le proposte di miglioramenti devono riflettere la propria esperienza, il contesto, gli obiettivi aziendali e le competenze del team di sviluppo. Per ogni scelta ci devono essere delle ragioni valide per la sua convenienza e applicabilità alla situazione specifica attuale.
- **Esperienza nell'applicazione di buone pratiche e principi**
- **Capacità di problem-solving**: una volta individuate le aree migliorabili, occorre "progettare" delle contromisure e un piano d'azione volti al miglioramento, che rispettino i vincoli del contesto in cui opera l'organizzazione.
- **Abilità interpersonali**: è essenziale essere in grado di interfacciarsi con tutto il personale dell'azienda per effettuare interviste, illustrare miglioramenti e comunicare con efficacia le pratiche e i principi che ne stanno alla base.

3.1.2 Interviste

Gli strumenti principali che si hanno a disposizione per l'attività di assessment sono:

1. Osservazione diretta

2. Interviste con gli *stakeholders*

In particolare, le interviste sono uno strumento essenziale per raccogliere informazioni e impressioni sui processi attualmente in uso, che non sarebbero immediatamente evidenti ad una prima osservazione dell'esterno. Per avere una comprensione completa, occorre osservare una realtà produttiva sia dall'esterno (osservazione) sia dall'interno (interviste).

E' necessario capire la *cultura*, gli *equilibri* e le *pratiche* che si sono venute a creare all'interno del team di sviluppo. Inoltre, è opportuno chiarire le *dinamiche* instauratesi con gli altri *stakeholders* (management, clienti, ecc).

Quindi è fondamentale intervistare prima di tutti i componenti del team di sviluppo (sviluppatori, analisti, progettisti, ecc) e del management. Successivamente, se possibile, si può anche procedere al coinvolgimenti di alcuni clienti dell'azienda per ottenere il punto di vista degli acquirenti: utile per avere una visione più completa.

In figura 3.1 sono raffigurate le aree principali da coprire durante una intervista.

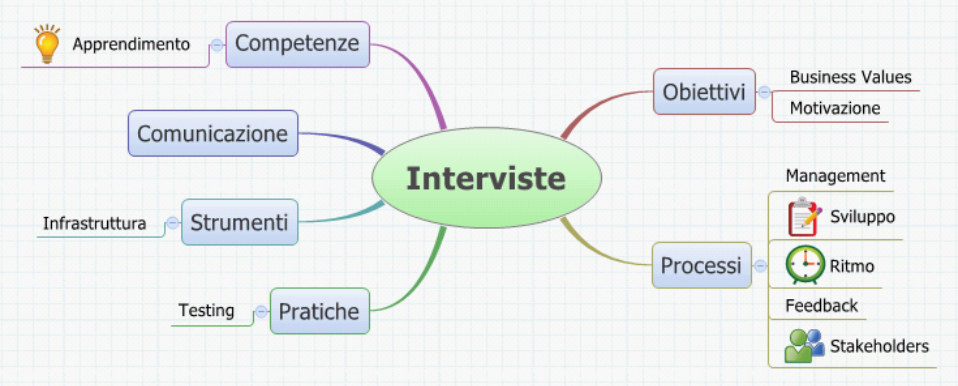


Figura 3.1: Argomenti principali di una intervista

Obiettivi

E' importante stabilire quali sono gli obiettivi dell'azienda (ciò che vuole migliorare, e in che direzione) e i *business values*. Dato che ogni organizzazione è diversa dall'altra, al fine di proporre dei veri miglioramenti occorre stabilire la scala di valori in base alla quale verranno misurate le proposte.

Ad esempio, un'azienda potrebbe privilegiare la trasparenza verso i clienti e la qualità del prodotto finito. Un'altra potrebbe considerare maggiormente la velocità di sviluppo e la flessibilità nel realizzare il software. Un'altra ancora potrebbe voler minimizzare i costi prima di tutto.

Gli obiettivi dell'azienda perciò sono qualcosa di fondamentale da conoscere per poter focalizzare le interviste, e proporre degli interventi mirati che possano avere la maggior efficacia possibile col minore sforzo implementativo.

Processi

Conoscere i processi (formali o informali) attualmente adottati, permette di disporre di una base da cui partire per il miglioramento senza rischiare di proporre modifiche che sconvolgano totalmente e in modo improvviso il metodo di lavoro, destabilizzando la cultura aziendale attuale.

In particolare è utile conoscere i processi che riguardano:

- Lo sviluppo di un progetto
- La presa in carico di un nuovo progetto
- La gestione di richieste di *bugfix* e nuove features
- La chiusura del progetto (contrattuale e *deployment*)

In generale è utile conoscere i processi e i punti di vista sia degli sviluppatori che del management (e, se possibile, anche dei clienti). Inoltre, è opportuno capire se si implementa un qualche tipo di *ritmo*[1] e una qualche forma di *miglioramento continuo*.

Pratiche

Nel corso del tempo, anche se si tratta di un campo piuttosto giovane, si sono venute a formare delle *best practices*[2] generalmente riconosciute che permettono di ottenere un prodotto di migliore qualità e nel modo più efficiente possibile.

E' utile capire se e quali *better practices* vengono impiegate dal team di sviluppo, se sono idonee al contesto (azienda, team, progetto, ecc), se sono vantaggiose e se vengono implementate nel modo corretto.

[1] Ovvero un ciclo di sviluppo che preveda delle iterazioni più o meno regolari.

[2] Io preferisco definirle "better practices", dato che non esistono pratiche sempre positive e adatte ad ogni situazione. Il contesto è critico nel determinare quanto una pratica ha senso ed è conveniente, da qui la futilità di ricercare *"best" practices. There is no Silver Bullet!*[18]

Strumenti

Spesso, le pratiche adottate da un team di sviluppo hanno più o meno bisogno di strumenti di supporto. Questi strumenti, per loro natura o per come vengono utilizzati, possono fornire un aumento di produttività o costituire un ostacolo.

Quindi, capire l'infrastruttura che l'azienda ha a disposizione ed eventuali vincoli alla sua modifica sono informazioni essenziali nella ricerca e proposta di nuove soluzioni. Bisogna indagare su quali strumenti vengono utilizzati attualmente, se sono adatti allo scopo, perché sono stati scelti e se si sono rivelati inadeguati per quale motivo.

Comunicazione

La comunicazione è un'ingrediente **essenziale** in un buon team di sviluppo, e più è semplice meglio è. Tuttavia, nonostante la comunicazione *face-to-face* sia la migliore, non deve essere la sola opzione. Ai fini di documentazione, di coordinamento e di planning è opportuno disporre di mezzi di comunicazione diversi a seconda delle esigenze. Indagare sulle forme di comunicazione attualmente impiegate dall'azienda e su come queste influiscano sul processo di sviluppo.

Questa è una di quelle aree in cui l'essere esterni al progetto fa la differenza.

Formazione e competenze

Data la natura dell'industria, la formazione riveste un ruolo particolarmente importante: le tecnologie evolvono ad una velocità pressante e rimanere aggiornati è essenziale per mantenere un vantaggio competitivo sui concorrenti e per migliorare sempre di più il proprio modo di lavorare, rendendolo via via più efficace ed efficiente.

Inoltre, le competenze dei singoli sviluppatori influiscono sulla produttività generale del team in cui lavorano e dell'intera azienda (vedi sezione 4.1.1). Capire quali sono i punti di forza e i talloni d'Achille di ogni componente del team permette di ottimizzare il lavoro e coprire eventuali mancanze in modo mirato e *preventivo*.

Quindi, data l'importanza strategica dell'apprendimento continuo, conviene indagare sul ruolo che questo riveste nell'attuale organizzazione del lavoro.

3.1.3 Modellazione

Una volta raccolte abbastanza informazioni tramite interviste e osservazione dirette, il prossimo passo è quello di elaborarle per costruire una visione generale da utilizzare per mettere in evidenza problemi e aree migliorabili, e per pianificare i passi successivi.

Un buon inizio, oltre all'eventuale uso di mappe mentali e appunti per organizzare il *corpus* di informazioni raccolte, è quello di realizzare delle rappresentazioni grafiche dei processi e dell'infrastruttura rilevate.

Processi

Quello della rappresentazione di *workflow* e processi è un ambito di studi abbastanza importante da aver richiesto la creazione e standardizzazione di un linguaggio grafico per la loro modellazione. L'utilità è presto detta: progettazione, analisi, documentazione, standardizzazione e ottimizzazione di processi. Specialmente in ambito *enterprise*.

Nel nostro caso, la modellazione dei processi attuali (come emersi dalle interviste e dall'osservazione) ha lo scopo di renderli espliciti e in una forma adatta ad essere analizzata ed esposta. Grazie ad una rappresentazione grafica, si è in grado di ottenere una visione d'insieme e di notare particolari o problemi che non sarebbero stati altrettanto evidenti studiando i processi in altra forma. Senza contare che il solo fatto di creare questi diagrammi costituisce un lavoro di *sintesi* che consente di capire meglio le modalità di lavoro dell'azienda.

Un linguaggio standard emergente per la modellazione di processi è il *BPMN*[3] [7, 31, 32]. E' stata utilizzata questa notazione nel corso dello stage per la modellazione dei processi interni correnti e proposti.

Infrastruttura

I processi rappresentano sequenze di attività e di decisioni eseguite principalmente da un gruppo di persone che cooperano verso un obiettivo comune. Ma le persone sono una parte dell'insieme e si servono normalmente di strumenti per portare a termine questi compiti. *L'insieme degli strumenti e delle soluzioni tecniche utilizzate dal team al fine di raggiungere un determinato obiettivo costituiscono l'infrastruttura di progetto.*

E' utile indagare su questo aspetto perché, sebbene gli strumenti non siano la panacea[18] dello sviluppo del software, influiscono sulla produttività e una buona infrastruttura può aiutare a incrementare l'efficienza e l'efficacia

[3] http://www.bpmn.org/

tanto quanto un insieme di strumenti di supporto inadeguati rallenta e rende più difficile ottenere dei prodotti di qualità.

Può essere utile, specialmente in presenza di un'infrastruttura articolata, creare dei diagrammi che contengano i vari elementi e le interazioni tra di essi. Questi diagrammi possono essere creati utilizzando dei semplici simboli attraverso dei programmi generici, oppure utilizzando i *deployment diagrams* dello standard *UML*[4].

3.2 Aree migliorabili

Partendo dalla comprensione e dalle competenze di base delineate nella sezione 3.1, attraverso l'osservazione diretta e le interviste a buona parte dei componenti dell'azienda presso cui ho effettuato lo stage, sono emerse alcune aree potenzialmente migliorabili che vengono di seguito sintetizzate:

- **Processi**: la sostanziale mancanza di processi espliciti (sono per lo più impliciti e si sono formati in modo relativamente spontaneo) limita le possibilità di organizzazione e miglioramento.
- **Tracciamento**: la mancanza di un metodo e degli strumenti adatti per la gestione efficiente di tempo e *task* rende inutilmente difficoltose l'organizzazione (molte interruzioni, stress, mancanza di tempo) e la pianificazione (difficoltà con la prioritizzazione dei compiti). Inoltre, il management non ha una visione chiara e puntuale dello stato di avanzamento e di salute dei progetti, il che rende difficile reagire con tempestività alle difficoltà che inevitabilmente sorgono. Senza qualche forma di tracciamento ne soffrono la produttività e la qualità.
- **Responsabilità**: la relativa mancanza di una figura *diversa per ogni progetto* investita della responsabilità di prendere decisioni, a volte costituisce un collo di bottiglia e crea disagi.
- **Documentazione**: la condivisione della conoscenza è prevalentemente orale. Se da una parte la comunicazione faccia a faccia è la migliore, dall'altra non dovrebbe essere l'unico mezzo di trasmissione di informazioni. Mancando un luogo centralizzato, strutturato e indicizzato per contenere le informazioni utili a tutti si è soggetti alla disponibilità delle persone depositarie di tali conoscenze che diventano un collo di bottiglia (e che vengono interrotte sottraendole al loro lavoro).
- **Expertise**: a causa della mancanza di tempo è molto difficile implementare l'apprendimento continuo (approfondimento di buone pra-

[4] http://www.uml.org/

tiche, strumenti, tecniche e di una cultura della qualità). Di conseguenza risulta abbastanza difficile coltivare un vantaggio competitivo e implementare un piano di miglioramento a lungo termine.

- **Testing**: è prevalentemente manuale, e quindi non si dispone di una *suite* di test di regressione che supporti la progettazione e il *refactoring*. Inoltre non si ottengono i relativi vantaggi sulla riduzione dei difetti (vedi sezione 2.4.1).

Il modo di lavorare attuale, sebbene funzioni non è dei più efficienti, e presenta seri problemi di scalabilità: finché ci sono poche persone (2 o 3) a lavorare ad un progetto i difetti sono in qualche misura gestibili, ma con un aumento nel numero di sviluppatori la situazione diventerebbe molto presto intollerabile.

Le aree problematiche evidenziate sono a mio avviso espressione di una causa di base. Dato che:

- Lo sviluppo del software è molto più che conoscere un linguaggio di programmazione o delle librerie: servono delle competenze di organizzazione, delle buone pratiche e un buon metodo di lavoro (meta-programmazione).
- Raramente qualcosa riesce bene la prima volta che la si fa.
- Senza esperienza nelle competenze di base non si possiedono le capacità di analisi del feedback e quindi di miglioramento.
- Non c'è stato un processo di base che abbia fornito delle linee guida sul modo di procedere.

Il team di sviluppo per ogni progetto si è "inventato" un processo informale senza alcuna guida o supporto, che si è rivelato poco efficiente e che non è stato possibile migliorare oltre un certo punto. Il che è perfettamente comprensibile considerando la mancanza di una base di esperienza a livello aziendale nei processi e nelle buone pratiche, di tempo e l'impossibilità di ottenere e analizzare feedback attraverso il tracciamento di metriche rilevanti.

3.3 Modello dei processi

Durante la fase di *assessment* è stata modellata la situazione dei processi, principalmente attraverso diagrammi BPMN (vedi sezione 3.1.3). Per la

creazione dei diagrammi è stato scelto di utilizzare il software *web-based* *Oryx*[5].

Dato che l'azienda analizzata sperimenta attivamente con i processi cercando il miglioramento, la situazione era leggermente diversa per ogni progetto in corso di sviluppo. La mia opera di sintesi si è riferita ad un caso abbastanza virtuoso e relativamente più esplicito degli altri.

Quello illustrato in figura 3.2 rappresenta la presa in carico di un progetto e il suo normale sviluppo con una visuale dettagliata di tipo B2B[6]. Si tratta essenzialmente di un ciclo di vita iterativo, con iterazioni relativamente irregolari e dalla durata abbastanza lunga, nelle quali il *testing* è effettuato prevalentemente a mano. Gli aspetti principali da mettere in evidenza sono:

1. Viene effettuata una fase di *pre-analisi* durante la quale si dialoga con il cliente per una prima specifica di massima dei requisiti, sufficiente per effettuare una stima e formulare (attraverso un dialogo tra management e team di sviluppo) un'offerta economica.

2. Al termine della fase di pre-analisi, se l'offerta economica viene accettata si svolge la fase principale di analisi dei requisiti da parte degli analisti, i quali producono una documentazione (relativamente snella) utile agli sviluppatori.

3. Viene pianificato lo sviluppo (attorno alle funzionalità, vedi sezione 2.3.1) attraverso delle *milestone* (stabilite contrattualmente) che costituiscono degli incrementi. Il mezzo principale di pianificazione utilizzato è la *FBS*[7] (WBS[8] secondo il gergo interno).

4. Il raggiungimento di una *milestone* comporta un'attività di *testing* (prevalentemente manuale) al termine della quale si presenta al cliente un verbale che attesta l'avvenuta realizzazione delle funzionalità previste per quella particolare *milestone*.

5. Data la tipologia di software e la filosofia aziendale, si preferisce il *training on-site* dal cliente piuttosto che la produzione di manualistica (comunque prevista per offerte rivolte a più clienti contemporaneamente).

Il processo di gestione delle richieste di nuove funzionalità o *bugfix* è illustrato in figura 3.3.

[5] http://oryx-editor.org/

[6] *Business to Business*, nel quale viene esplicitato il ruolo degli *stakeholders* principali.

[7] *Feature Breakdown Structure*. Si tratta di una suddivisione gerarchica di requisiti, macro-requisiti e *task* utile[21, 33, 34, 35] alla pianificazione del lavoro. Particolarmente

L'attività di sviluppo di una *issue* (richiesta di nuova funzionalità o di *bugfix*) è espansa in figura 3.4. Qui il processo presenta degli aspetti degni di nota:

- Uso in parallelo di due strumenti diversi (*Eventum* e la WBS, vedi sezione 3.4) per coordinare lo sviluppo, che vanno mantenuti in sincrono manualmente. Si tratta di un'attività scomoda, lenta e prona ad errori.
- Il *testing* viene eseguito manualmente e dopo lo sviluppo della funzionalità, quindi non si ottengono i vantaggi del *TDD* per quanto riguarda la progettazione e la realizzazione di una *suite* di test di regressione. Di conseguenza si perde in termini di qualità, flessibilità del *design* e produttività.

3.4 Infrastruttura

In questo caso l'infrastruttura non si è rivelata così complessa da richiedere un diagramma. Di seguito invece verranno elencati gli strumenti principali utilizzati per l'organizzazione, la pianificazione e lo sviluppo:

- **FBS**: si tratta di un foglio di calcolo conservato in un disco rigido condiviso in LAN, contenente i requisiti e la suddivisione in *task* con tanto di pianificazione per quanto riguarda la tempistica e diverse metriche. Va tenuto aggiornato manualmente.
- **Eventum**: un *issue tracker*[9]. Laddove la FBS viene utilizzata principalmente nella fase di sviluppo vera e propria, *Eventum* viene utilizzato per interfacciarsi con i clienti. Però, dato che la FBS non consente una collaborazione efficiente, viene utilizzato l'*issue tracker* per gestire parte dello sviluppo ordinario.
- **Mercurial**: viene utilizzato come strumento di SCM[10], ma senza alcun *repository* di riferimento. Le modifiche vengono condivise tra gli sviluppatori attraverso *bundle* spediti per email.

La *FBS* ed *Eventum* costituiscono quindi viste diverse sugli stessi dati (funzionalità, bug e task), che vanno mantenute sincronizzate manualmente. Si tratta di un processo inefficiente, tedioso e prono ad errori.

indicata in un ciclo di vita iterativo e incrementale.

[8] *Work Breakdown Structure*. Simile alla FBS ma organizzata attorno alle attività di sviluppo[35].

[9] Strumento il cui scopo è quello di permettere la gestione di *issue* (tipicamente richieste di nuove feature o *bugfix*) attraverso una interfaccia web.

[10] *Source Code Management*.

3.5 Comunicazione

La comunicazione all'interno dell'azienda si svolge principalmente faccia a faccia, oralmente, e quando ciò non è possibile tramite telefono, IM[11] o email. Quindi il livello di collaborazione è soddisfacente.

Però non vengono utilizzati mezzi di comunicazione e conservazione delle informazioni *strutturati*, il che rende molto difficile e costoso l'inserimento di nuovi sviluppatori in un progetto già avviato. Chiunque abbia bisogno di un'informazione è costretto a richiederla, interrompendo l'interessato dal lavoro che sta effettuando. L'uso quasi esclusivo di comunicazioni "volatili" costituisce un serio collo di bottiglia non solo per gli sviluppatori già al lavoro, ma anche e soprattutto per eventuali nuovi arrivati.

> "*Verba volant, scripta manent.*"
> –Detto latino

3.6 Formazione

Nel caso in esame, la formazione è presente sotto forma di scambio di articoli, discussioni informali e suggerimenti su libri di interesse. Un'ottima cosa, ma si può fare di più e in maniera un pò più strutturata.

[11] Instant Messaging. Ad esempio: Skype, GTalk, MSN Live, ecc.

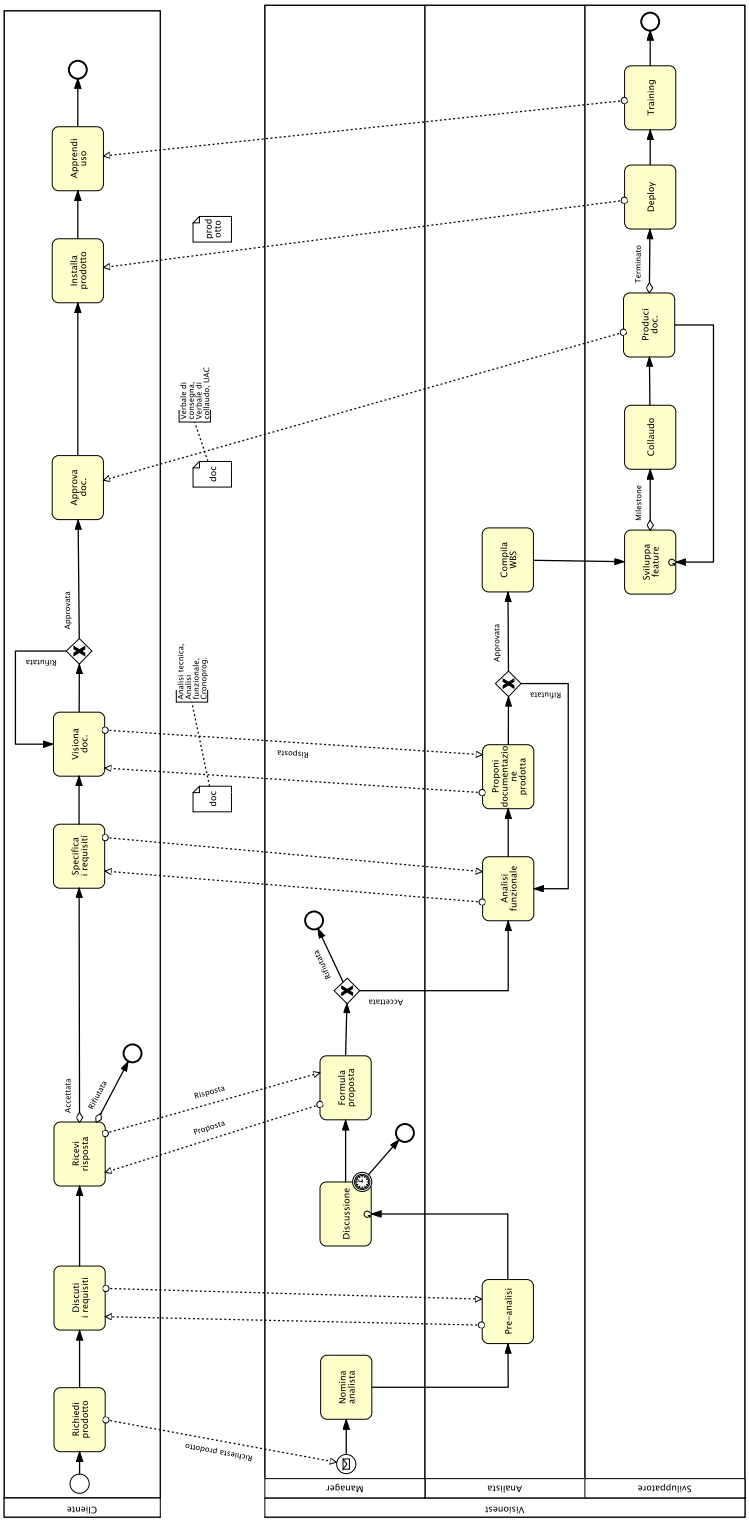


Figura 3.2: Processo B2B generale

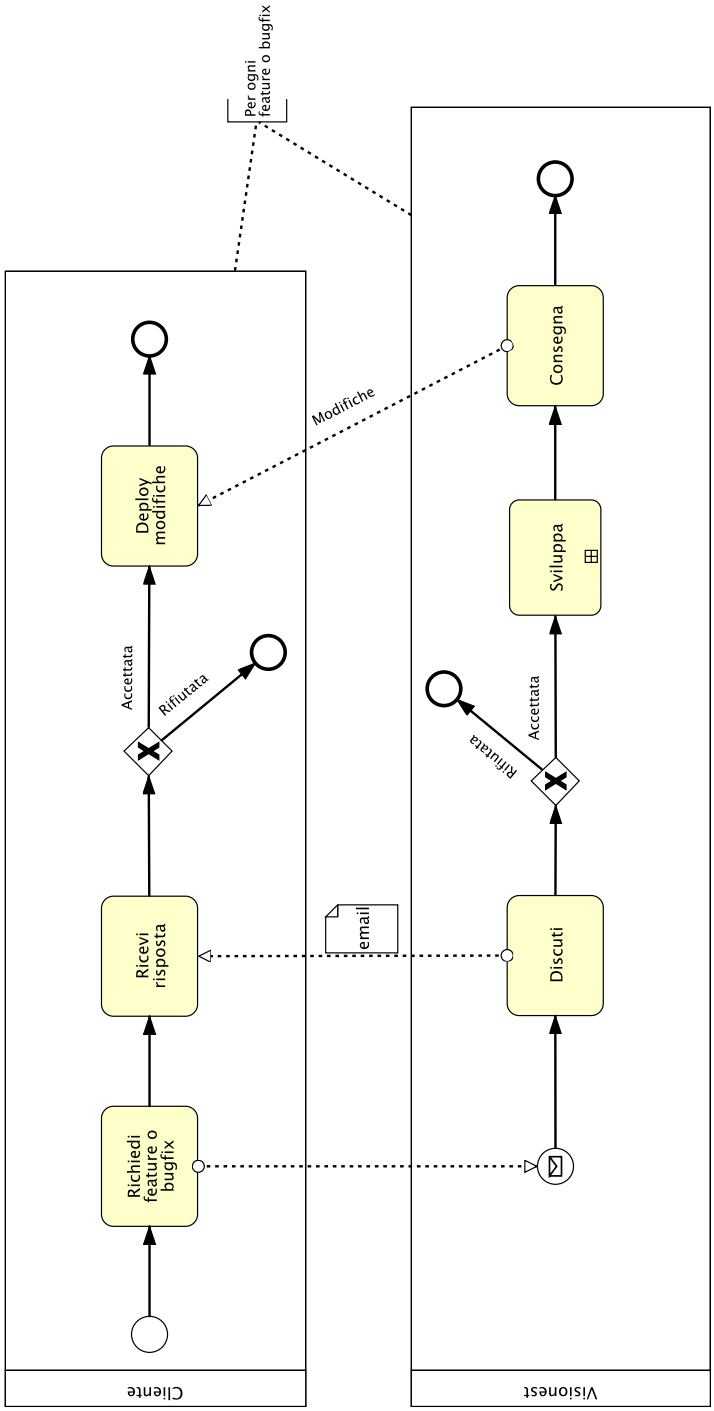


Figura 3.3: Gestione di issue

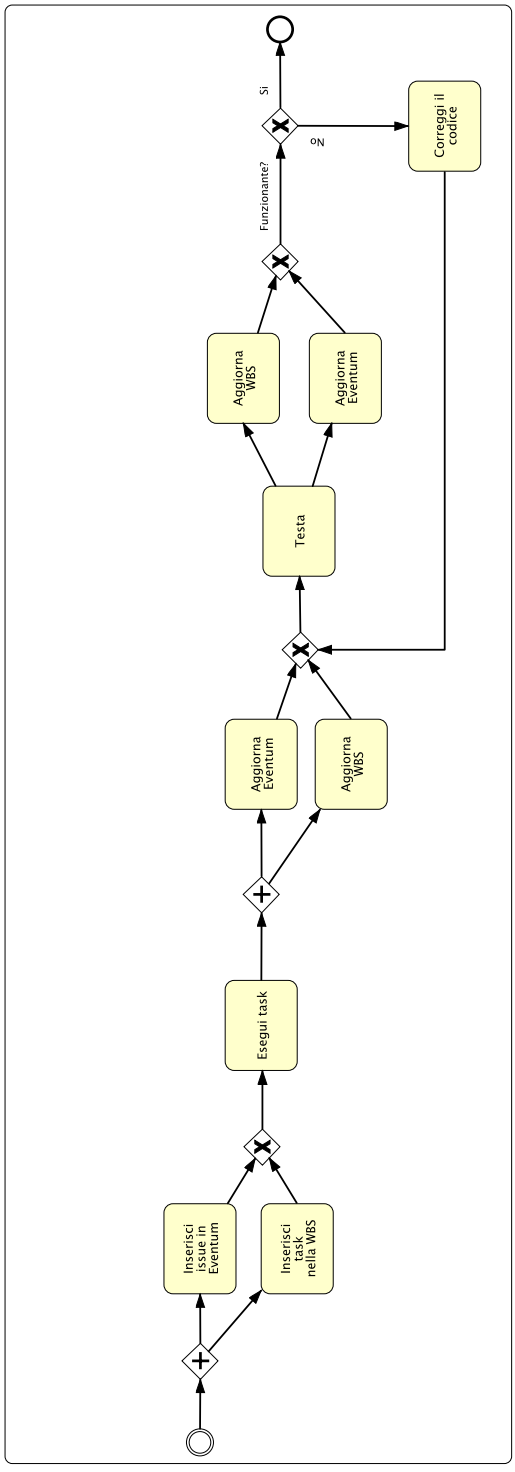


Figura 3.4: Sviluppo di una feature

Capitolo 4

Evoluzione

Dopo aver raccolto tutte le informazioni necessarie ad avere una visione completa della situazione durante la fase di *assessment*, è il momento di studiare gli interventi e le modifiche da apportare alla situazione.

Come già scritto nella sezione 3.1.2, i miglioramenti vanno studiati affinché incontrino le priorità dell'azienda soddisfacendo allo stesso tempo tutti i vincoli del particolare contesto. In questa attività, entrano in gioco in modo preponderante le capacità di *problem solving*.

Gli strumenti presentati e proposti all'azienda ospitante (vedi sezione 4.4.2), sono stati utilizzati in prima persona dall'autore per l'organizzazione e la conduzione sia dello stage che della tesi. In particolare è stato utilizzato l'SCM *Mercurial* su *BitBucket* per conservare e versionare la documentazione e gli appunti prodotti. Inoltre sono stati utilizzati l'email e la piattaforma *Twitter* come *Information Radiators* per l'*accountability* al tutor interno e al *management* sullo stato di avanzamento dei lavori.

4.1 Ruolo dei processi

Come descritto nella sezione 2.1.2, i processi non si sono rivelati il tanto ricercato *silver bullet* che avrebbe risolto tutti i problemi, nonostante l'approccio abbia prodotto delle metodologie di sviluppo molto dettagliate e burocratiche. L'Agile è nato appunto per contrastare gli *eccessi* di questa concezione, che è finita per svalutare grandemente l'aspetto umano in favore di un'aderenza "religiosa" ad un processo fissato dettagliatamente a priori. Data l'infinita varietà (e mutabilità) dei contesti[1], *non esiste un singolo processo adatto ad ogni situazione*.

[1] Progetti, clienti, team, tecnologie, vincoli, ecc. ecc.

Ma questo non significa affatto che i processi siano inutili, o che tutto debba essere fatto "a naso". L'Agile è tanto diverso dagli approcci *heavyweight* tanto quanto lo è dal vecchio *code&fix*: si tratta di una visione più equilibrata che tiene conto delle pratiche più utili che si conoscono.

4.1.1 Modello di Dreyfus

Per capire il giusto posto dei processi in un metodo di lavoro che sia agile e al tempo stesso efficiente ed efficace, occorre parlare del *modello di Dreyfus*[2, 41] di acquisizione delle competenze. In breve, secondo questo modello una persona attraversa cinque diversi stadi per passare dallo *status* di novizio a quello di esperto. Ognuno di questi stadi poi è caratterizzato da un punto di vista, da una livello di comprensione e da delle esigenze peculiari rispetto agli altri.

Nell'uso di una determinata abilità, mentre un novizio ha bisogno di istruzioni precise e di pratica deliberata, un esperto sa agire efficacemente in modo semplice ed elegante, senza sforzo apparente, praticamente inconsciamente. Tanto che l'esperto stesso in molti casi non è in grado di articolare come ha fatto, questo perché la competenza è diventata una seconda natura e ha interiorizzato le conoscenze e l'esperienza che ha accumulato nel corso tempo, mettendolo in grado di utilizzare la propria abilità senza pensarci.

Ma le differenze vanno al di là della mera conoscenza o esperienza: esperti e novizi differiscono in modo fondamentale nelle rispettive percezioni del mondo, capacità di *problem solving*, modelli mentali della realtà, modalità di apprendimento e incidenza dei fattori esterni nel proprio approccio.

Per meglio capire le differenze, trattiamo brevemente i cinque stadi di competenza secondo questo modello (vedi figura 4.1). Alla base troviamo lo stadio di novizio, e salendo via via verso l'ultimo, quello della maestria.

Stadio 1: Beginner (novizio)

Un novizio ha poca o nessuna esperienza nell'area di competenza, dove esperienza significa pratica che abbia comportato un cambiamento di prospettiva. Infatti, si può essere degli sviluppatori da dieci anni ed essere ancora dei novizi perché si è ripetuto il primo anno di pratica per nove volte. Per accumulare vera esperienza occorre la voglia di imparare, ma tipicamente un novizio è orientato ad un'obiettivo immediato, non gli interessa la visione d'insieme o l'apprendimento.

Il novizio non sa valutare il risultato delle proprie azioni, e nemmeno come reagire agli errori, così quando succede qualcosa di inaspettato è confusione e paralisi. Tuttavia, sono abbastanza efficienti se gli vengono date delle

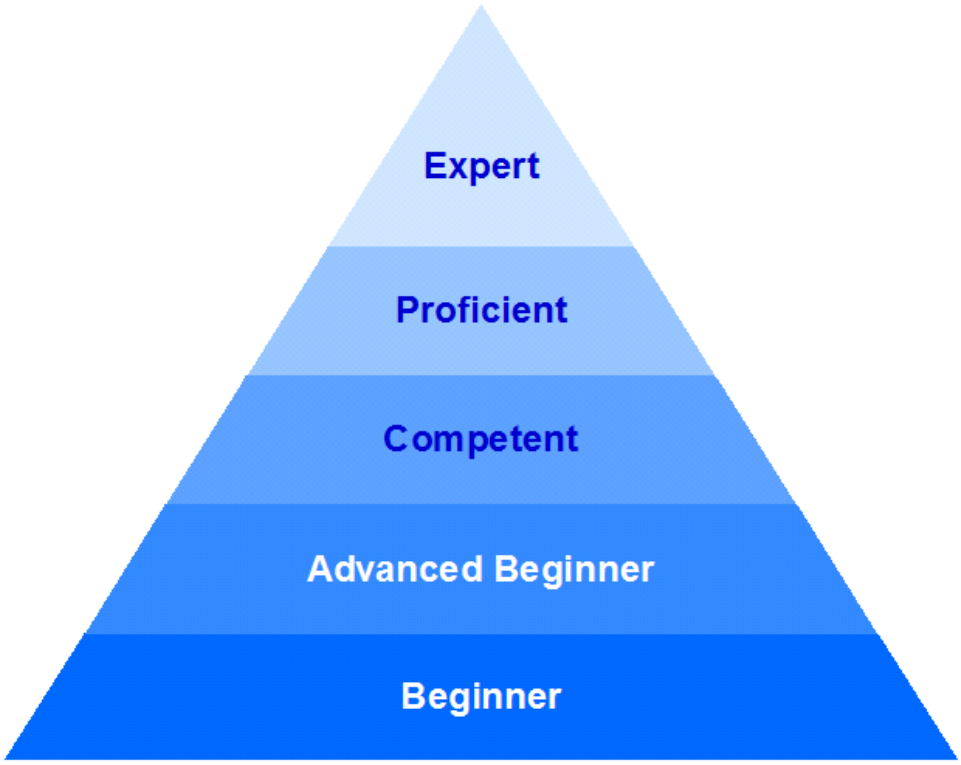


Figura 4.1: Schema del modello di Dreyfus

regole prive di contesto[2] da seguire. **I novizi hanno quindi bisogno di "ricette"**, di un *sistema* o *metodo* da seguire passo-passo, e di una guida nella sua applicazione.

Il problema con queste regole fisse è il novizio non sa *quando* vanno applicate e quando no. Le regole non possono essere mai specificate completamente e in modo non ambiguo. Si possono fornire altre regole per chiarirle, e poi altre ancora per chiarire quelle nuove e via di questo passo in una *regressione infinita*. Ad un certo punto però occorre fermarsi nelle definizioni esplicite, ed è allora che il novizio sperimenta confusione.

Stadio 2: Advanced Beginner (novizio avanzato)

Il novizio avanzato può cominciare ad allontanarsi un pò dall'insieme di regole fisse e può iniziare ad eseguire compiti per conto suo, ma ha ancora difficoltà a diagnosticare i problemi quando si presentano.

Vuole informazioni velocemente per fare ciò che gli serve, non gli interessa la teoria. Può cominciare ad usare i suggerimenti nel contesto adatto,basandosi

[2]Ad esempio: "quando succede X, fai Y."

sulla sua limitata esperienza, e riuscirà a formulare alcuni principi generali. Ma **non ha una comprensione olistica**, una visione d'insieme, **e nemmeno gli interessa**.

Il pericolo principale a questo stadio (o anche al precedente) è quello di diventare dei "teorici": la *pratica* è un elemento essenziale e imprescindibile per avanzare nei gradi di competenza in un'abilità, diversamente uno può finire a concentrarsi esclusivamente e ossessivamente in una teoria che non saprà come applicare. Senza la pratica alle spalle il "teorico", in preda alla *paralisi da analisi*, finirà per confondersi le idee, perché non avendo della esperienza alle spalle (il test della realtà) non saprà distinguere il vero dal falso e capire quando va applicato cosa. In pratica possiederà solo delle regole, dei principi e delle visioni generali, ma non sarà in grado di verificarle e non avrà fatto esperienza del ruolo fondamentale che il *contesto* gioca. Molte teorie, infatti, sembrano contrastanti quando in realtà sono corrette ma si applicano in contesti diversi.

Stadio 3: Competent (competente)

A questo stadio, uno inizia a sviluppare modelli concettuali del dominio del problema e a lavorare efficacemente con essi. **Può diagnosticare e risolvere problemi autonomamente**, ed anche affrontarne di nuovi. Può inoltre iniziare a cercare e applicare i consigli degli esperti, senza rischiare la paralisi da analisi.

Le sue azioni sono ora basate più su una personale pianificazione ed esperienza che su un set di regole predefinite. Ma senza ulteriore esperienza, non c'è ancora una vera abilità di riflessione e *auto-correzione*.

Stadio 4: Proficient (abile)

La persona abile, secondo questo modello, ha bisogno della visione generale e vuole capire il *framework* concettuale attorno all'abilità. Di conseguenza, le regole prive di contesto e i vincoli che aiutano i novizi ad accumulare esperienza sono inutili e seccanti per uno abile.

Molto importante, l'abile **è in grado di riflettere su quello che fa e sulle sue prestazioni, modificando il proprio approccio per ottenere risultati migliori**. Fino a questo stadio, questo tipo di auto-miglioramento non era disponibile. Inoltre, può imparare anche dall'esperienza degli altri oltre che dalla propria.

Insieme a questa capacità di auto-analisi e auto-miglioramento, ottiene la

possibilità di applicare le *massime*[3] degli esperti, le quali vanno applicate nel contesto giusto, che diventa chiaro solo per una persona abile o esperta.

> "*Always consider the context.*"
> –[2]

E' utile notare come le abilità di analisi del *feedback* e di miglioramento continuo siano alla base delle metodologie agili. Infatti, secondo [1]: "*Agile development uses feedback to make constant adjustments in a highly collaborative environment.*"

Ad esempio, uno dei movimenti più importanti nel campo dello sviluppo del software è quello dei *Design Patterns*[4], che sono delle *massime*, non delle ricette per novizi. Infatti l'intento dei *design pattern* è quello di documentare e trasmettere degli elementi di buon design a sviluppatori abili. Diversamente, se vengono interpretati come delle ricette, si può finire con l'inserire[2] *diciassette* design pattern in un piccolo software di reportistica senza che ce ne sia alcuna utilità pratica. Massime come queste vanno applicate *cum grano salis*, proprio ciò che manca ai novizi.

Stadio 5: Expert (esperto)

L'esperto è la fonte primaria di conoscenze e informazioni in ogni campo. Dispone di una esperienza molto vasta e varia da cui attingere, e che sa applicare con grande successo nel giusto contesto. E' spinto da una grande curiosità e voglia di imparare, e cerca continuamente nuovi e migliori modi di fare le cose.

Il tratto distintivo dell'esperto è che lavora con l'*intuizione* più che con la ragione. Spesso compie la scelta giusta, ma non sa dire come ci è arrivato. L'esperto, grazie alla molta pratica, al vasto bagaglio di esperienze e agli ampi e particolareggiati modelli mentali, è in grado di distinguere i dettagli irrilevanti da quelli importanti, formulando giudizi e soluzioni sulla base di una capacità di *pattern matching* affinata nel corso del tempo.

> "*Technical knowledge is not enough. One must transcend techniques so that the art becomes an artless art, growing out of the unconscious.*"
> –Daisetzu Teitaro Suzuki

[3] A differenza delle regole dei novizi, sono molto più vaghe e si affidano molto di più all'esperienza di chi le sente o legge. Ecco perchè sono inutili ai novizi: hanno come prerequisiti esperienza e modelli concettuali che ancora non possiedono.

[4] http://c2.com/cgi/wiki?DesignPatterns

Secondo le statistiche[2], gli esperti in un dato campo costituiscono solo l'1-5% dell'intera popolazione.

4.1.2 Applicazione del modello di Dreyfus

Come abbiamo visto, persone a livelli di competenza diversi in una data disciplina pensano e lavorano in modo diverso tra di loro. Possiamo vedere in figura 4.2 uno schema della progressione del modo di lavorare in base allo stadio di competenza nel modello di Dreyfus.

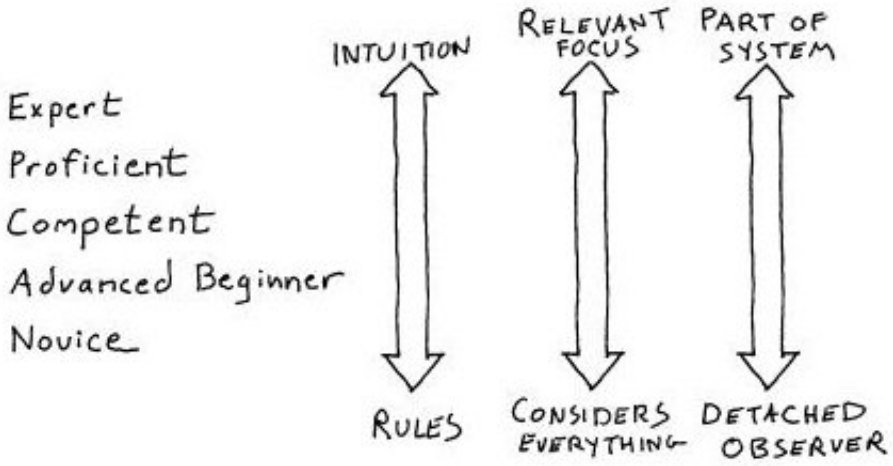


Figura 4.2: Acquisizione di competenze nel modello di Dreyfus - Fonte: [2]

Ogni team di sviluppo, con tutta probabilità sarà composto da membri con livelli di competenza anche molto diversi tra loro, e quindi non possono essere trattati come un insieme omogeneo di risorse sostituibili. Come se ciò non bastasse, dato che il modello di Dreyfus si applica per ogni competenza, e che in ogni singola attività di sviluppo rientrano molte abilità, abbiamo differenze di competenze e di *performance*[5] non solo tra diversi sviluppatori, ma anche tra diverse abilità in ognuno di essi (ciò crea un'ulteriore varianza nel contesto). Di conseguenza, il metodo più adatto in una data circostanza deve tenere conto anche di questo fatto (adattandovisi), ed è questo uno degli aspetti in cui gli approcci precedenti hanno fallito.

Spesso invece, seguendo la vecchia concezione dello sviluppo del software come una catena di montaggio dell'epoca della rivoluzione industriale, gli esperti vengono forzati a seguire "Il Processo", nonostante il loro intui-

[5] Alcuni studi[2] quantificano la differenza media tra uno sviluppatore novizio ed uno esperto con rapporti che vanno da 20:1 fino addirittura ad un 40:1.

to e giudizio in certi casi suggerisca il contrario, mentre i novizi vengono scaraventati nella "mischia" senza una adeguata guida.

E' stato fatto un esperimento[2] nel quale a dei piloti di linea esperti è stato chiesto di stilare delle regole che rappresentassero le loro *best practices*. Queste regole sono state poi date a dei novizi, che utilizzandole hanno effettivamente incrementato le proprie *performance*. Ma quando sono state fatte seguire dagli stessi esperti che le avevano scritte, il loro rendimento è sceso significativamente.

Per capire meglio l'effetto di un'acritica aderenza a dei processi prefissati, basti pensare al fenomeno dell'*obbedienza malevola*[2] che si verifica quando un dipendente, per protestare contro i propri superiori, comincia a lavorare facendo *esattamente* quello che la propria qualifica prescrive, né più né meno, e seguendo le regole alla lettera[6]. Il risultato è un gran caos e rallentamento.

> "*Practices can never be completely objectified or formalized because they must ever be worked out anew in particular relationships and in real time.*"
> –[42]

Mentre un novizio non ha esperienza né conoscenze specifiche e deve essere *guidato* passo per passo per compiere un lavoro con successo e acquisire dimestichezza, l'esperto ha già affinato le proprie capacità ad un'ottimo livello, facendole diventare una seconda natura e sviluppando un *intuito* che lo guida (il più delle volte) alla soluzione più elegante e appropriata con il minimo sforzo. Un esperto libero di esercitare il proprio giudizio, avendo una visione d'insieme abbastanza precisa, è in grado di scegliere l'approccio più adatto al contesto (pragmatismo) e migliorarlo in modo continuo adattandolo alle contingenze del particolare problema che si trova ad affrontare. Ne consegue che: abbandonare i novizi a sè stessi e costringere gli esperti a seguire processi eccessivamente rigidi e dettagliati è il modo migliore per rendere inefficaci entrambi, e quindi la ricetta per il fallimento.

> "*Use rules for novices, intuition for experts.*"
> –[2]

Una considerazione importante che si può ricavare dall'applicazione di questo modello è che un metodo di lavoro veramente agile è possibile solo con un team di persone abili o esperte. Solo loro infatti possiedono la capacità e l'esperienza per scegliere lo strumento giusto al momento giusto, e *adattare* il proprio modo di lavorare in base al contesto. E' per questo che in certe

[6] Il principio è lo stesso di quando alcuni bambini seguono letteralmente le parole dei propri genitori per fargli un dispetto...

situazioni si creano problemi nell'applicazione dei principi dell'Agile, e che in alcuni fa sorgere la percezione errata che sia in realtà un modo di lavorare indisciplinato e caotico. **Per essere agili serve disciplina**, e i novizi devono essere guidati (anche da processi espliciti) nella pratica e nell'acquisizione di esperienza in modo da elevare il loro livello di competenza fino a che possano essere veramente agili nel loro lavoro.

Per fare in modo che l'azienda incrementi la propria agilità è necessario quindi capire qual'è il livello di competenza dei suoi sviluppatori e predisporre il giusto percorso[7] per elevare il livello di *expertise*, in modo da poter fare leva sulle capacità di analisi del *feedback* e miglioramento continuo.

Il modo più efficace con il quale si può innalzare il livello di competenza dei componenti del team di sviluppo è quello di assicurarsi la presenza nello stesso di persone abili o esperte che facciano da esempio. Infatti, *modellare*[8] un esperto è il modo migliore che si conosca per acquisire esperienza e competenza. E' il metodo che viene utilizzato da molto tempo e con grande successo ad esempio nelle arti marziali e nel Jazz. Il trombettista Clark Terry descrive questo processo di crescita nel modo seguente:

1. **Imitazione**: delle pratiche esistenti.
2. **Assimilazione**: delle conoscenze non scritte e dell'esperienza nel corso del tempo.
3. **Innovazione**: capacità di andare oltre le conoscenze pre-esistenti, che ormai sono diventate una seconda natura.

Per fare leva su questo modello e svilupparsi come azienda è necessario creare un ambiente che incoraggi la crescita e che premi la competenza. Pagare gli sviluppatori tutti allo stesso modo e meno dei manager spinge molti dei programmatori migliori a lasciare il team di sviluppo per darsi al management, determinando una fuga di *expertise* che alla lunga danneggia l'azienda. La realtà, invece, è che lo sviluppo è una professione che richiede esperienza e competenza, le quali possono essere sviluppate solo in un'ambiente adatto.

Questo ambiente deve essere regolato in base ai suoi componenti: deve spingere i novizi a migliorarsi fornendogli esempi da modellare e processi da seguire, e allo stesso tempo deve incoraggiare gli esperti a esercitare le proprie competenze senza eccessivi vincoli a limitarle.

[7] Qui entra in gioco il fattore fondamentale dell'apprendimento continuo (vedi sezione 4.6).

[8] Modellare una persona significa imitarne le azioni e gli atteggiamenti, assimilandoli e facendoli propri nel corso del tempo. Funziona sia in negativo che in positivo: "Chi va con lo zoppo impara a zoppicare." Ma chi va con il vincente impara a vincere, e chi lavora gomito a gomito con un esperto in breve tempo lo diventerà a sua volta.

Gli esperti devono continuare a sviluppare, e perché questo sia possibile devono trovarvi una carriera significativa e remunerativa. Stabilire una scala di paghe e una vera carriera che rifletta il valore di uno sviluppatore esperto è il primo passo per rendere una realtà un ambiente che incoraggi la crescita.

4.1.3 Pericoli dei metodi (eccessivamente) formali

Metodi e processi che espongono un certo grado di formalizzazione possono essere utili ai novizi per imparare a muoversi con una certa produttività fin da subito, senza dover sprecare tempo e fatica con un approccio *trial&error*. Certe regole sono come delle "rotelle" che aiutano a partire, ma oltre un certo punto sono deleterie per le *performace*.

I metodi formali, portati all'eccesso e visti come *silver bullet* (vedi sezione 2.1.2), comportano dei seri problemi:

- **Svalutazione di tratti non formalizzabili**: aspetti essenziali del processo di sviluppo come la creatività e il *problem-solving* sono necessari tanto quanto non definibili. Il fatto che non possono essere fissati e inseriti in un crono programma non significa che non siano importanti.
- **Inibizione dell'innovazione**: i processi e metodi formali con tanto di legislazione restrittiva tendono a creare una mentalità da "gregge" invece di promuovere il pensiero individuale creativo e responsabile. Per contrasto, si vedano gli approcci vincenti della Apple con il suo motto "*Think different*" e di Google che dà il 20% del tempo lavorativo ai propri dipendenti perchè lavorino a progetti di loro scelta, senza restrizioni.
- **Alienazione degli esperti**: creare un ambiente particolarmente mirato ai novizi finisce per limitare e svalutare gli elementi con più esperienza.
- **Regressione infinita**: specificare troppi dettagli crea confusione e grandi difficoltà nel seguire le istruzioni. Inoltre, ogni volta che si rendono esplicite determinate assunzioni, si rivela un ulteriore strato di assunzioni non specificate.
- **Iper-semplificazione**: nessun modello o set di regole può tenere conto di tutte le variabili e le sfaccettature della realtà. L'approccio va modificato in base al contesto, ma metodi eccessivamente formali tendono a rendere difficile se non impossibile l'adattamento.
- **Scelta tra pragmatismo o formalismo**: spesso si mostra necessario deviare dai metodi formalizzati per ottenere un risultato migliore, ma

se l'iniziativa individuale e il giudizio ragionato vengono stigmatizzate in favore dell'aderenza alle direttive, si creano delle situazioni in cui le persone fanno qualcosa di palesemente inefficiente, inutile o sbagliato nonostante l'evidenza. Il che è decisamente deleterio per la morale e la produttività.

- **Mistificazione**: i discorsi diventano talmente ricchi di slogan che finiscono per perdere interamente il significato originario.

4.1.4 Riassumendo

Gli sviluppatori non sono tutti uguali: hanno livelli diversi di competenza e questo significa anche che hanno esigenze e modi di lavorare diversi. I novizi hanno bisogno di regole prive di contesto e di una guida passo per passo, ma non sono in grado di affrontare problemi nuovi. Inoltre, non avendo una visione d'insieme e la consapevolezza precisa di far parte di un sistema più grande, non sono nemmeno in condizione di capire il loro impatto su questo sistema. Gli esperti, viceversa, hanno bisogno di accedere alla visione generale e non di essere limitati da regole burocratiche e restrittive che mirano a sostituirsi al loro giudizio.

Idealmente, un'azienda dovrebbe creare un ambiente che incoraggi la crescita, affiancando gli esperti ai novizi in modo che quest'ultimi vengono guidati e imparino "per osmosi".

Tutto questo discorso sull'agilità e il modello di Dreyfus non ha lo scopo di liquidare come "il male" i processi dettagliati o il modello di sviluppo *Waterfall*. Piuttosto serve a dare il giusto posto a quei concetti che essendo degli strumenti, possono essere utili o meno a seconda di *come* e *quando* vengono impiegati. Il ciclo di vita sequenziale e le "ricette" dettagliate hanno il loro perché, come abbiamo visto, ma non sono la risposta definitiva come molti hanno teorizzato negli anni passati.

Come illustrato nella sezione 2.1.3, il trend attuale e le conoscenze che abbiamo accumulato indicano abbastanza chiaramente che *la strada più adatta per la maggior parte dei progetti è quella del ciclo di vita iterativo e incrementale con un'enfasi sulle persone e sull'applicazione di buone pratiche, e con il coinvolgimento attivo dei clienti.* Per applicare questo approccio servono le capacità di auto-analisi e di miglioramento degli esperti.

In pratica, conviene ragionare seriamente se l'agilità possa essere utile al proprio *business*.

4.2 Motivazione

Comunemente si pensa che gli incentivi e le punizioni (la carota e il bastone) aumentino le *performance*, ma sperimentalmente[14] si è scoperto che, spesso, è vero l'*opposto*.

Ad influire sul potere degli incentivi è la natura del compito da svolgere: mentre per i compiti meccanici che hanno un obiettivo e un modo per raggiungerlo ben chiari gli incentivi determinano effettivamente un aumento di prestazioni, quelli che richiedono creatività e capacità di *problem solving* (come chiaramente lo sviluppo del software), gli incentivi e le punizioni o non hanno effetto o addirittura *danneggiano* la produttività. Questo è uno dei risultati più solidi e meglio provati nel campo delle scienze sociali, e anche uno dei più ignorati dall'industria. *C'è una differenza sostanziale tra quello che la scienza sa e quello che il business fa.*

I *motivatori estrinseci* come gli incentivi economici funzionano per incrementare le prestazioni in compiti meccanici, ma sono ininfluenti o dannosi per le *performance* in attività che richiedono *anche rudimentali* abilità cognitive. Sfortunatamente, quelli estrinseci sono i motivatori più usati[9] nel mondo del business. Per promuovere la produttività e l'efficienza dei lavoratori nell'ambito dell'IT, funzionano molto meglio i *motivatori intrinseci*.

La motivazione intrinseca (ancora una volta: enfasi sulle persone) è costruita attorno al desiderio di fare qualcosa perchè è importante, piace, è stimolante e fa parte di qualcosa di più grande. Secondo Daniel Pink[14], il "sistema operativo" delle organizzazioni che prevedono attività creative, dovrebbe ruotare attorno a tre elementi fondamentali:

- **Autonomia**: la spinta a dirigere la propria vita
- **Maestria**: il desiderio di diventare sempre più abili in qualcosa che conta
- **Significato**: la soddisfazione di fare quello che va fatto al servizio di qualcosa di più grande di sè stessi

In particolare, per quanto riguarda l'autonomia, la nozione classica di *management* funziona se si vuole obbedienza, ma quando è più desiderabile il coinvolgimento, la reattività e la creatività, allora l'*auto-organizzazione* funziona meglio.

Una volta stabilita una paga adeguata al livello di competenza, è utile togliere la questione economica dal tavolo per focalizzarsi sull'autonomia e il coin-

[9] Fortunatamente le cose stanno cambiano, si vedano le politiche di Google, Apple e Blizzard ad esempio.

volgimento. Negli ultimi anni si sono sviluppati approcci molto interessanti ed estremamente efficaci in questo senso:

- *Atlassian*: una volta all'anno ha luogo il *Fedex Day*, in cui ogni dipendente è libero di lavorare a ciò che desidera per 24 ore, a patto che non riguardi il proprio lavoro di ogni giorno, per poi presentarlo al resto dell'azienda. E' stato talmente positivo da portare a...

- *Google*: il 20% del tempo lavorativo di ogni dipendente può essere dedicato a progetti a scelta, personali o meno, in completa libertà su tutto (compiti, team, tecnologie, pratiche, metodologia di sviluppo, ecc). Molti degli strumenti attualmente più utili e famosi sono nati come progetti liberi, ad esempio: GMail, Google News, AdSense e Google Earth.

- *Apple*: fin dalle sue origini ha una tradizione di innovazione e di incoraggiamento della creatività. Basti pensare al suo motto: *"Think different!"*

- *Blizzard*: la *sede*[10] del loro *Campus* è un'ambiente studiato secondo questi criteri per supportare e incoraggiare la creatività.

Un ulteriore esempio è il *ROWE*[11], creato da due consulenti statunitensi e impiegato da moltissime compagnie nel nord America. In questa tipologia di ambiente di lavoro i dipendenti non hanno orari e non è necessario che si presentino in ufficio, l'unico vincolo è che svolgano bene il proprio lavoro. Come, dove e quando lo fanno è a loro discrezione. I risultati, rispetto ad un ambiente tradizionale, sono: produttività, coinvolgimento e soddisfazione dei dipendenti aumentano, mentre il *turnover* diminuisce. Data l'importanza della comunicazione faccia a faccia, ciò non è sempre possibile nello sviluppo del software, tuttavia è un buon esempio di come l'autonomia possa determinare un aumento di rendimento degli sviluppatori.

Superare l'ideologia dalla carota e dal bastone, e portare ciò che la scienza già sa e ha provato da diverso tempo a riguardo della *motivazione* nelle aziende che sviluppano software (come già sta succedendo) è un ingrediente fondamentale per creare un'ambiente in cui le menti possono crescere e creare *con successo* dei software che siano utili, solidi, flessibili e, non meno importante, remunerativi nel senso più ampio del termine.

[10] http://us.blizzard.com/it-it/company/careers/
[11] *Result-Only Work Environment.*

4.3 Aree di intervento

Dopo aver raccolto tutte le informazioni sullo stato attuale, è il momento di pianificare un intervento per risolvere gli eventuali problemi evidenziati e individuare le azioni da intraprendere. Lo scopo è quello di migliorare la produttività dell'azienda, mettendola poi in grado, nel corso del tempo, di migliorarsi autonomamente e con continuità, adattandosi alle mutate esigenze del mercato e dei clienti.

Per poter modificare positivamente il modo di lavorare in un'azienda, è necessario adottare un *approccio integrato*, agendo su più livelli e aree contemporaneamente. Ad alto livello le aree fondamentali in cui concentrarsi sono:

- **Principi e linee guida**: sviluppatori, analisti, manager e in generale tutti i componenti dell'azienda è bene che siano consapevoli della realtà dello sviluppo, di ciò che funziona e di cosa no nel loro specifico contesto. Dovrebbe esserci una visione d'insieme sui principi e le linee guida che avviano quel circolo virtuoso che conduce poi a caratteristiche desiderabili come: esperienza, pragmatismo, agilità e auto-miglioramento.
- **Pratiche**: occorre la consapevolezza e conoscenza delle buone pratiche e di come queste influenzino lo sviluppo. Di progetto in progetto poi, andrebbero scelte su base *as needed* le pratiche più adatte al caso specifico, assemblandole in un processo da cui partire con l'adattamento.
- **Infrastruttura e strumenti** possono aumentare la produttività come essere un intralcio. In base al progetto, ai vincoli e alle pratiche adottate, dovrebbero essere utilizzati gli strumenti più adatti a supportare gli *stakeholder*.

Un esempio di applicazione di questo approccio integrato è schematizzato in figura 4.3. Ad esempio, per un progetto in cui la stabilità e il soddisfacimento dei requisiti del cliente sono particolarmente importanti e assegnato ad un team di sviluppatori competenti, è pensabile di adottare un ciclo di vita iterativo e incrementale praticando il BDD *end-to-end*[12], l'integrazione continua e le *code reviews* formali con la tecnica *"The Pickup Game"* (vedi sezione 2.4). Come strumenti di supporto si potrebbero utilizzare `Fitnesse` con `JUnit` o `easyb` per il testing, `Hudson` come server di integrazione continua, `Pivotal Tracker` come strumento di *planning* e coordinamento, ecc.

[12] Ovvero con un uso estensivo di specifiche eseguibili a guidare la progettazione e testando estensivamente l'applicazione sia a livello di unità di codice che a quello dei requisiti funzionali.

Il punto fondamentale: **è l'approccio che deve adattarsi al contesto, non viceversa**.

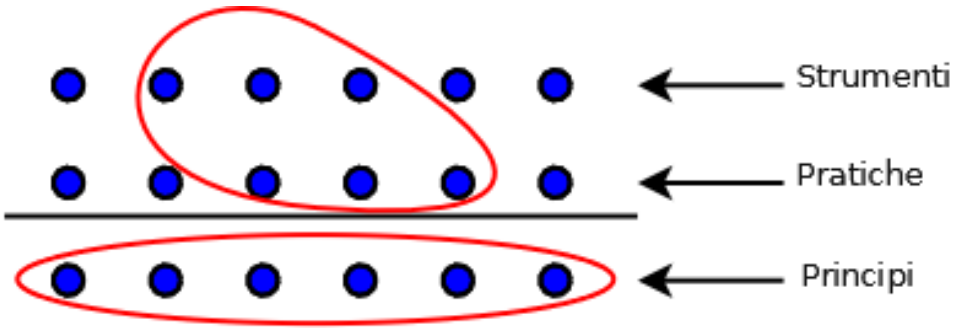


Figura 4.3: Esempio di applicazione dell'approccio integrato

Tuttavia, bisogna fare un ulteriore distinguo tra progetti già avviati e nuovi, che come vedremo tra poco rappresentano situazioni, vincoli e opportunità diverse sotto il profilo degli interventi migliorativi.

4.4 Stabilizzazione

Per i progetti già in corso di sviluppo il *team* avrà già sviluppato in modo più o meno consapevole un modo di lavorare, e stravolgerlo finirebbe per destabilizzare gli sviluppatori provocando confusione e ritardi.

In questi casi, l'approccio migliore è quello di introdurre solo delle piccole modifiche che si integrino il più possibile con il modo di fare attuale, ma che apportino comunque dei miglioramenti.

Di seguito saranno illustrate le proposte migliorative elaborate per l'azienda presso cui ho effettuato lo stage, che verranno valutate dalla stessa per un'eventuale implementazione.

4.4.1 Pratiche

Sotto il profilo delle pratiche, viste le possibilità limitate di mutamento offerte da progetti già avviati e consolidati ormai da diverso tempo (con relative pratiche), le proposte sono rivolte principalmente al miglioramento della qualità. In particolare:

- **Cultura della qualità**: a causa della mancanza di tempo dovuta a un approccio organizzativo da mettere a punto, non si è sviluppata adeguatamente la consapevolezza e la competenza nel *testing* automatizzato. L'introdurre la pratica del *TDD* in modo completo, per basi di

codice già in buona parte consolidate e che non dispongono di una *test suite* esaustiva, probabilmente non porterà tutti i vantaggi resi possibili da questa tecnica (vedi sezione 2.4). Ma si può già cominciare a ragionare in termini di testabilità, il che produce comunque un atteggiamento che porta vantaggi sotto il profilo del *design* e che costituisce al tempo stesso un utile esercizio per acquisire dimestichezza.

- **Retrospettive**: già effettuate in modo informale e irregolare. Una volta esplicitati i processi in atto, è consigliabile programmare dei brevi meeting per ragionare su ciò che è stato fatto e come farlo meglio. Trattandosi di un'applicazione delle capacità di auto-analisi, è utile per sviluppare la consapevolezza del proprio metodo di lavoro e di un'atteggiamento focalizzato alla soluzione dei problemi.

- **Code reviews**: data la mancanza di una *test suite* esaustiva, propongo l'implementazione di sessioni di revisione statica del codice per aiutare a ridurre la presenza di difetti. Principalmente da applicare in modo irregolare sulle parti di applicazione più difficoltose e che con più probabilità contengono errori. Ridurre il numero di *bug* nel codice è molto utile soprattutto per limitare la necessità di manutenzione liberando quindi tempo per l'apprendimento e la pratica del TDD, che una volta applicato correttamente, permetterà di lavorare in modo più efficiente, liberando ulteriore tempo per le fasi successive del miglioramento.

4.4.2 Infrastruttura

Le proposte per la stabilizzazione dei progetti si focalizzano però principalmente sull'infrastruttura, dalla modifica della quale si ricaverebbero i maggiori benefici. In particolare, gli elementi infrastrutturali presi in esame sono:

- **ALM**[13]: per risolvere il problema dell'organizzazione e del tracciamento[14], è utile adottare un software centralizzato e integrato che fornisca funzionalità di *planning*, *issue tracking* e che raccolga automaticamente metriche utili a stimare lo stato di salute e di avanzamento di un progetto. Introducendo un software di questo tipo, il management guadagnerebbe l'*accountability*[15] del team, la raccolta automatica di

[13] *Agile Lifecycle Management*

[14] Di *issue*, *task*, *features*, ecc.

[15] Per *accountability* si intende la possibilità dei project manager di conoscere ciò che viene fatto e a che ritmo, il che è essenziale per capire se e dove ci sono problemi da risolvere in questo senso.

metriche, funzionalità avanzate di pianificazione (e modifica della stessa in vista di cambiamenti di priorità, risorse, ecc.) e in ultima analisi una collaborazione più efficace ed efficiente. Per la proposta di un software di questo tipo è stato fatto un estensivo lavoro di ricerca e comparazione, che ha portato alla proposta degli strumenti: `VersionOne`[16], `RallyDev`[17] e `Mingle`[18].

- **SCM**[19]: data la realtà produttiva dell'azienda in esame, che non rende sempre possibile la co-locazione, è necessario un software di versionamento di tipo decentralizzato con un relativo *repository* di riferimento, in modo che non sia necessario l'accesso alla rete e vi sia una risposta univoca alla domanda *"dov'è l'ultima versione del software X?"*. Dato l'utilizzo di `Mercurial`[20] da parte dell'azienda, il luogo più naturale per accogliere questo *repository* è `BitBucket`[21], e infatti questo passo è già stato fatto durante il mio stage.

- **Continuous Integration**: per stimolare l'applicazione di buone pratiche di *testing* e per catturare tempestivamente i problemi, in modo da poterli risolvere quando ancora costa relativamente poco farlo, propongo l'installazione e l'uso di un server di *continuous integration* come `Hudson`[22]. `Hudson` ha come vantaggi una vasta comunità alle spalle, una buona robustezza e una grande facilità di installazione e gestione. L'integrazione continua effettuata con questo strumento sarebbe un'operazione trasparente dal punto di vista degli sviluppatori, e di grande utilità per la creazione di prodotti di migliore qualità.

- **Knowledge Base**: ogni progetto dovrebbe avere un luogo centralizzato, indicizzato e di facile utilizzo per contenere la documentazione relativa al suo funzionamento e alle decisioni di *design* che lo riguardano. `BitBucket` fornisce già un repository associato ad ogni progetto, in grado di contenere documentazione sotto forma di wiki. Potrebbe essere utile anche prevedere un wiki a livello aziendale (meta-progetto) per contenere documentazione appunto di carattere generale e soluzioni a problemi incontrati (un *daylog*, vedi sezione 2.4.11), costruendo una vera e propria *knowledge base* che permetterebbe di ridurre il tempo necessario per l'inserimento di un nuovo sviluppatore in un progetto già avviato.

[16] http://www.versionone.com/
[17] http://www.rallydev.com/
[18] http://www.thoughtworks-studios.com/mingle-agile-project-management
[19] Source Code Management
[20] http://mercurial.selenic.com/
[21] http://www.bitbucket.org/
[22] https://hudson.dev.java.net/

- **Information Radiators**[1]: l'uso di uno strumento da parte di ogni programmatore che permetta di pubblicare sul web in tempo reale informazioni sul proprio stato, novità e problemi sarebbe un'ulteriore passo in avanti verso la *trasparenza* e l'*accountability*, oltre che agevolare lo scambio di conoscenze. Un utente di questo strumento potrebbe essere anche il server di integrazione continua, che potrebbe così avere un ulteriore canale in *real time* per notificare eventuali problemi. Ad esempio, si potrebbe utilizzare `Status.net`[23] o `Twitter`[24].

4.5 Piano d'adozione

Oltre a proporre delle misure da adottare nell'immediato, è utile pensare a delle strategie e dei miglioramenti da proporre in un'ottica a lungo termine per nuovi progetti. Si tratta di *incrementare il know-how dell'azienda in modo che sia in grado di implementare in ogni situazione un metodo di lavoro flessibile, efficiente, efficace e adatto al contesto aziendale, con tutte le relative pratiche abilitanti finalizzate alla produttività e alla qualità.*

4.5.1 Strategie di adozione

Un buon piano d'adozione a lungo termine deve tenere conto dei requisiti relativi alle competenze dei membri dell'azienda, come visto nella sezione 4.1.1. In particolare è necessario sviluppare le competenze e le pratiche di base (TDD/BDD su tutte) con l'ausilio di un processo esplicito in aiuto ai novizi. Il tutto creando un ambiente di lavoro adatto anche agli esperti e che promuova la crescita (vedi sezione 4.2) attraverso l'apprendimento continuo (sezione 4.6).

La soluzione migliore, a mio avviso, sarebbe l'avvio di un *progetto pilota* senza eccessivi vincoli, nel quale gli sviluppatori assegnati possano fare pratica con le tecniche più utili ed efficaci, applicandole in modo da ottenere un processo più agile.

Sarebbe un'ottima cosa prevedere l'utilizzo di un *consulente* o *coach* esterno esperto che assista e supervisioni l'applicazione corretta delle pratiche. L'uso di un consulente è una soluzione che costa, ma fornirebbe due vantaggi importanti:

- La sua guida risulterebbe determinante per applicare correttamente le *better practices* più adatte alla situazione.

[23] http://status.net/
[24] http://www.twitter.com/

- Il fatto di aver pagato per i suoi servigi spingerebbe l'azienda a completare la transizione ad un metodo di lavoro migliore, diversamente è possibile che l'esperimento venga interrotto senza ottenerne quindi i benefici.

Successivamente, alla conclusione del progetto pilota, gli sviluppatori coinvolti possono aiutare il resto dell'azienda nella transizione, trasmettendo l'esperienza accumulata durante lo stesso. Questa soluzione prende il nome di *In-House Workshop*[9] ed è schematizzato in figura 4.4.

Figura 4.4: In-House Workshop

Un'altra possibilità, è quella di utilizzare i principi dell'Agile per guidare la transizione. In altre parole si tratterebbe di introdurre all'interno dell'azienda le buone pratiche di sviluppo e di management in modo iterativo e incrementale. Questo metodo, proposto da Mike Cohn, prende il nome di *Agile Transition Framework*[10] ed è schematizzato in figura 4.5. In breve si potrebbero prevedere delle iterazioni di 3-4 mesi, per le quali stabilire come obiettivo l'introduzione di un certo numero di pratiche, supportando il tutto con dei meeting per la raccolta e l'utilizzo del *feedback* in modo da adattare la transizione alla realtà aziendale.

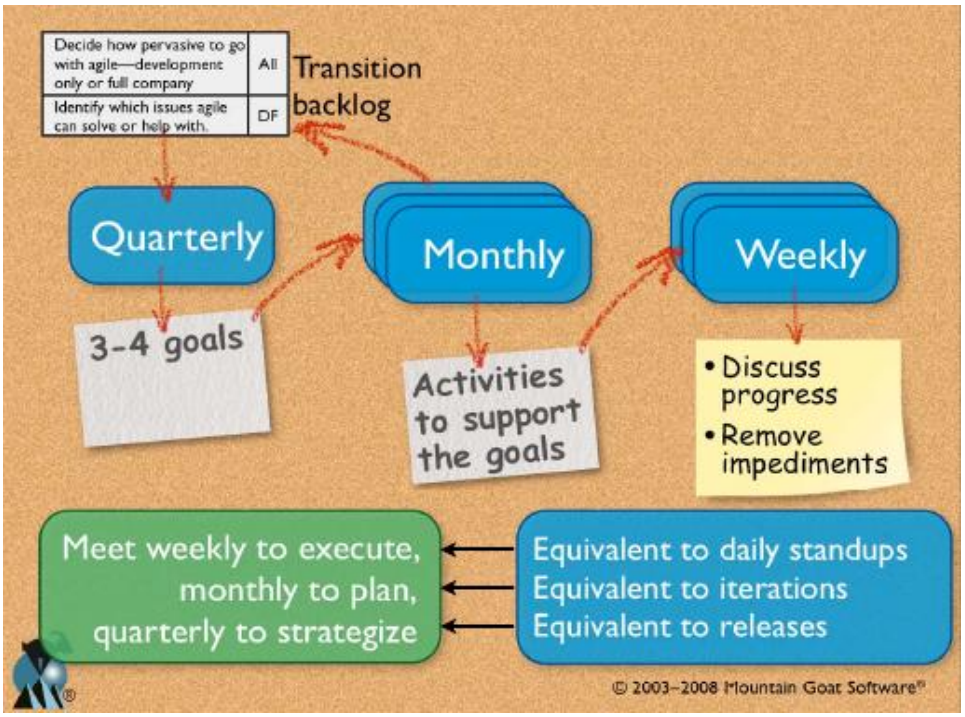


Figura 4.5: Agile Transition Framework

4.5.2 Pratiche

Le pratiche suggerite, considerando la situazione attuale, sono le seguenti:

- **BDD**: l'introduzione del TDD come pratica riguardante il *testing* e il *design*, è essenziale per la realizzazione di un software di qualità che sia ragionevolmente stabile, con pochi bug e abbastanza flessibile da poter essere modificato per adattarsi ad un cambiamento di requisiti o per accogliere una nuova *feature* richiesta da qualche cliente. In particolare, oltre all'utilizzo estensivo del *testing* per guidare la realizzazione e per disporre di una *suite* di test di regressione, suggerisco l'introduzione dei *test di accettazione automatizzati* e il testing della GUI. Come strumenti di supporto per la pratica del BDD suggerisco, ad esempio, `Fitnesse`[25], `easyb`[26], `Cucumber`[27] e `Selenium`[28].

- **Daylog**: utilizzo della *knowledge base* (vedi sezione 4.4.2) aziendale per condividere le soluzioni ai problemi incontrati. In un team

[25] http://fitnesse.org/
[26] http://www.easyb.org/
[27] http://cukes.info/
[28] http://www.seleniumhq.org/

veramente efficiente non si dovrebbe mai cercare la soluzione ad un problema per più di una volta.

- **Code Reviews**: effettuare l'analisi statica del codice prima di ogni *push* potrebbe non essere fattibile o conveniente, sarebbe opportuno comunque pensare a questa possibilità in futuro se si rivelasse necessario ridurre ulteriormente il numero di difetti riscontrati nel codice. Senza contare che questa pratica consente ai vari sviluppatori di prendere confidenza con altre aree dell'applicazione su cui non lavorano direttamente, il che serve per coltivare una comprensione migliore del funzionamento dell'intera applicazione e per evitare l'eccessiva dipendenza da un singolo sviluppatore.

- **Pair Programming**: attualmente non ha molto senso attuarla come pratica in modo continuativo (eventualmente come strumento in più per facilitare l'inserimento di nuovi elementi in un *team*), ma varrebbe la pena considerarla in un futuro in cui ci saranno più programmatori. Se attuata, consentirebbe di effettuare un'analisi statica del codice *continua* (con la conseguenza di diminuire i bug) e promuovere l'apprendimento e la diffusione delle buone pratiche.

4.5.3 Processi

Come visto nella sezione 4.3, date le esigenze e differenze nei vari progetti che vengono affrontati da una stessa un'organizzazione, in mia opinione, piuttosto che avere un singolo processo è bene avere il giusto *know how* ed esperienza nelle buone pratiche e nella loro applicazione (principalmente attraverso l'apprendimento continuo, vedi sezione 4.6), tra cui scegliere per costruire il processo più adatto alla situazione. Ogni progetto, ogni cliente e ogni situazione è diversa dall'altra e richiede un approccio adatto.

Nel caso pratico dell'azienda in cui ho lavorato, ho elaborato un processo (non troppo dissimile da quello attualmente impiegato) che ingloba le idee di sviluppo iterativo e incrementale *feature-driven* con un uso estensivo del testing attraverso la pratica del TDD/BDD.

In figura 4.6 vediamo uno schema generale, i cui particolari degni di nota sono:

- L'attività di analisi funzionale produce un elenco di requisiti (qui chiamati storie), che vengono inseriti nel sistema di ALM (vedi sezione 4.4.2) e prioritizzati in base alle priorità del cliente.

- Vengono pianificate delle *milestone* contrattuali.

- Si entra nel ritmo pianificando ed eseguendo iterazioni di lunghezza regolare, eventualmente presentando delle demo al cliente e raccogliendo *feedback* con il quale migliorare il prodotto, modificare la prioritizzazione delle storie (requisiti) e il proprio processo di sviluppo.
- Al termine dell'iterazione, il cliente ha la facoltà di decidere se continuare o meno con lo sviluppo.

Per una visione più in dettaglio e con la specifica dei ruoli coinvolti, è stato realizzato un diagramma dettagliato B2B[29] del processo di sviluppo, mostrato in figura 4.7. Rispetto allo schema generale, si evidenziano maggiori dettagli:

- L'azienda, dopo una prima fase di analisi, formula un'offerta economica che sottopone al cliente.
- Qual'ora questo accetti, si procede alla pianificazione di *milestone* e iterazioni assieme al cliente, in modo tale da poter presentare qualcosa di funzionante al termine di ognuna.
- Lo sviluppo di un'iterazione prevede di discutere con il cliente i dettagli sulle funzionalità previste per la stessa.
- Al termine di ogni *milestone*, l'azienda effettua un collaudo che deve essere controfirmato dal cliente. Quando possibile, al termine di una iterazione si può presentare una demo tramite la quale raccogliere *feedback* dal cliente, al fine di migliorare il processo di sviluppo e il prodotto stesso.
- Al termine del progetto, avviene il *deploy* e l'addestramento del personale nell'uso del prodotto.

In particolare, l'attività di sviluppo di una iterazione è espansa in figura 4.8, dove sono da evidenziare i seguenti punti:

- Lo sviluppo avviene una funzionalità alla volta, scegliendo ogni volta quella a priorità maggiore.
- Il *push* nel repository di riferimento può essere fatto solo quando tutti i test passano dando esito positivo. Questo è necessario per evitare di creare problemi agli altri componenti del team e per mantenere la base di codice sempre in salute. L'uso di un server di *countinuous integration* serve anche per l'*enforcing* di questa *policy*.

[29] *Business to Business.*

- L'arrivo di una *issue bloccante* costituisce un'evento ad alta priorità e assume la precedenza sulla normale implementazione delle funzionalità previste per quella iterazione.
- Se il tempo allocato per l'iterazione scade senza che si siano completate tutte le funzionalità, si può comunque procedere a dimostrare quanto fatto (grazie a TDD e integrazione continua che consentono di avere sempre il software funzionante) e a modificare la pianificazione.

Forma contrattuale

Come descritto nella sezione 2.3, lo sviluppo iterativo e incrementale comporta importanti vantaggi sia per l'azienda che implementa questo ciclo di vita, sia per i suoi clienti. In breve, i vantaggi per il cliente sono:

- E' coinvolto maggiormente nel processo di sviluppo e vede con frequenza i progressi (e dove vanno i suoi soldi).
- Vedendo spesso delle demo è in grado di chiarirsi le idee e aiutare il team guidandolo nella realizzazione di un prodotto che soddisfi le sue reali esigenze.
- E' in grado di prioritizzare i requisiti in base al contesto attuale (progressi compiuti, tempo e soldi disponibili) e ottenere prima quello che è più importante per sè stesso.

Anche per l'organizzazione ci sono dei vantaggi importanti nell'instaurare un ritmo nello sviluppo che porta feedback rapido e frequente:

- Realizzazione di un prodotto che risponde meglio alle esigenze del cliente: maggiore soddisfazione di quest'ultimo e migliore pubblicità.
- Riduzione del tempo da dedicare alla manutenzione, quindi maggior tempo per sviluppare nuovi progetti e meno interruzioni.
- Pianificazione più flessibile e reattiva che risponde meglio alla realtà.
- Miglioramento continuo dei processi.
- Miglioramento costante delle capacità di stimare costi, tempi e sforzi necessari.
- Sviluppo che massimizza in ogni momento il *business value* prodotto e minimizza il rischio.

- Realizzazione di un prodotto di *qualità superiore*.

Per supportare questo ciclo di vita (quando si rivela fattibile e desiderabile), però, occorre una forma contrattuale adeguata. Un contratto "a corpo"[30], ha lo svantaggio di fissare un prezzo che si basa su una stima iniziale dei costi fatta su una comprensione necessariamente incompleta e imprecisa del lavoro richiesto, e se si va oltre i tempi stimati, dato che il costo è fisso si va in perdita.

Sfortunatamente, i progetti software sono sottoposti a molti rischi *non prevedibili*, soprattutto legati ai requisiti (variazioni, incomprensioni, incompletezza, ecc) ma anche a molti altri fattori (variazioni nelle performance del team, errori, complessità accidentale e intrinseca, mancanza di conoscenze, ecc). D'altra parte una forma contrattuale a costo fisso è rassicurante per il cliente perché non riserverà sorprese in parcella.

> "*Given the inherent volatility and irreproducibility of software projects, coming up with a fixed price ahead of time pretty much guarantees a broken promise in the works.*"
> –[1]

Per poter applicare un contratto a corpo con una qualche possibilità di successo, è necessario effettuare delle stime iniziali abbastanza precise[31], le quali sono possibili solo a due condizioni:

- Dei requisiti chiari e stabili.
- Un team di sviluppo esperto che ha già larga esperienza nello specifico dominio applicativo.

Due condizioni che chiaramente è *molto* difficile poter soddisfare. Quindi nella stragrande maggioranza dei casi un contratto a costo fisso, indipendentemente dalla metodologia di sviluppo adottata, comporta dei rischi eccessivi. Su tutti: un *ROI*[32] negativo per l'azienda.

Come coniugare allora i vantaggi del contratto a corpo con uno sviluppo agile? Serve un modo per limitare il rischio sia per il cliente (in modo da non riservargli sorprese nel costo) che per l'organizzazione (in modo da non dover subire perdite). Una possibile forma contrattuale di questo tipo[1] è la seguente:

[30] Ovvero a costo fisso.
[31] Utilizzando tecniche come il *COCOMO* o la *Fixed Point Analysys*.
[32] *Return Of Investment*.

1. Offrire un primo prototipo nel giro di 6-8 settimane e con funzionalità sufficienti per mostrare quello che sarà il prodotto finale. Alla fine di questa prima iterazione, il cliente ha due scelte:
   - Può acconsentire a continuare con la prossima iterazione (e le successive funzionalità in ordine di priorità).
   - Può recedere dal contratto senza alcuna penalità e pagare solo per la realizzazione del prototipo. A questo punto il cliente può decidere di abbandonare il progetto o darlo in carico ad un'altra organizzazione.
2. Alla fine di ogni iterazione, il cliente ha la possibilità di scegliere se continuare con lo sviluppo o meno (pagando solo il lavoro svolto fino a quel momento).

Si può prevedere anche che le iterazioni abbiano costo fisso. In questo modo il cliente:

- può osservare i progressi compiuti nello sviluppo fin da subito e con continuità
- ha sempre il controllo di cosa verrà sviluppato, quanto e in che ordine
- *ha sempre il controllo su quello che spende*
- ha la certezza di poter staccare la spina in qualsiasi momento senza penalità contrattuali

E l'organizzazione:

- adotta un ritmo e un ciclo di vita incrementale e iterativo con tutti i vantaggi che questo comporta
- *non avrà mai progetti in perdita*

Conviene comunque dare una prima stima di massima (chiarendo che è misurata "a spanne" e probabilmente subirà variazioni) che sarà rifinita iterazione dopo iterazione. Con il procedere delle iterazioni e l'osservazione delle metriche (calcolate automaticamente dagli strumenti di ALM) la precisione delle stime cresce e diventa possibile farne di sempre più realistiche e precise.

In breve: *un ciclo di vita iterativo e incrementale, con iterazioni a prezzo fisso e numero variabile, in cui il cliente ha sempre il controllo su quanto viene realizzato e ha la facoltà di recedere dal contratto senza penalità,*

pagando solo il lavoro svolto fino a quel momento. Le stime sui tempi diventano sempre più precise mano a mano che si procede con lo sviluppo grazie all'uso di metriche e la *burn-down chart* (vedi sezione 2.4.10).

In particolare, secondo quanto emerso dalle interviste con il management dell'azienda presso cui ho effettuato lo stage, non sempre i clienti sono abbastanza "maturi" (dal punto di vista della comprensione delle complessità dello sviluppo), quindi se non è possibile effettuare delle demo circa ogni mese, è comunque conveniente prevedere *contrattualmente* delle *milestone*. Cosa che già si sta facendo.

4.6 Apprendimento continuo

Il campo dell'informatica evolve e cambia continuamente e molto rapidamente: occorre rimanere aggiornati per mantenere (o acquisire) un vantaggio competitivo. Ed è bene prevedere del tempo per l'apprendimento e la formazione durante l'orario lavorativo, diversamente è molto facile che non ci sia molta attività in questo senso.

L'apprendimento continuo è quindi un elemento fondamentale per mantenere il vantaggio competitivo, migliorare costantemente e creare quel tipo di ambiente di lavoro che promuova la produttività e la soddisfazione degli sviluppatori (vedi sezioni 4.1.1 e 4.2).

Alcune proposte di implementazione:

- Un'oretta al giorno (o quasi) da dedicare all'apprendimento e alla sperimentazione.
- **Una presentazione il fine settimana** fatta da chi vuole per condividere qualcosa di interessante per tutti. Naturalmente dovrebbe avere del tempo durante la settimana per prepararla. In questo modo si diffonde efficacemente la conoscenza e si fa pratica con la competenza della comunicazione, che è utile in molti ambiti (collaborazione con i colleghi, dialoghi con il management, rapporti con i clienti). Vedi appendice A.
- **Book club**: un libro, un capitolo alla volta che leggono tutti quando hanno tempo e infine una discussione (magari a pranzo) per condividere e costruire una comprensione condivisa su argomenti e tecniche rilevanti per l'azienda e i progetti in corso di sviluppo.
- **Coding Dojo**: sessioni di esercizio ed allenamento con tecniche, pratiche e strumenti per affinare le competenze. Consente di sperimentare e sbagliare in un ambito protetto, utile per esplorare nuove possibilità

(linguaggi, librerie, framework, pratiche, ecc). Sono molto utili al team e all'azienda, e consentono di coltivare l'*expertise* necessaria.

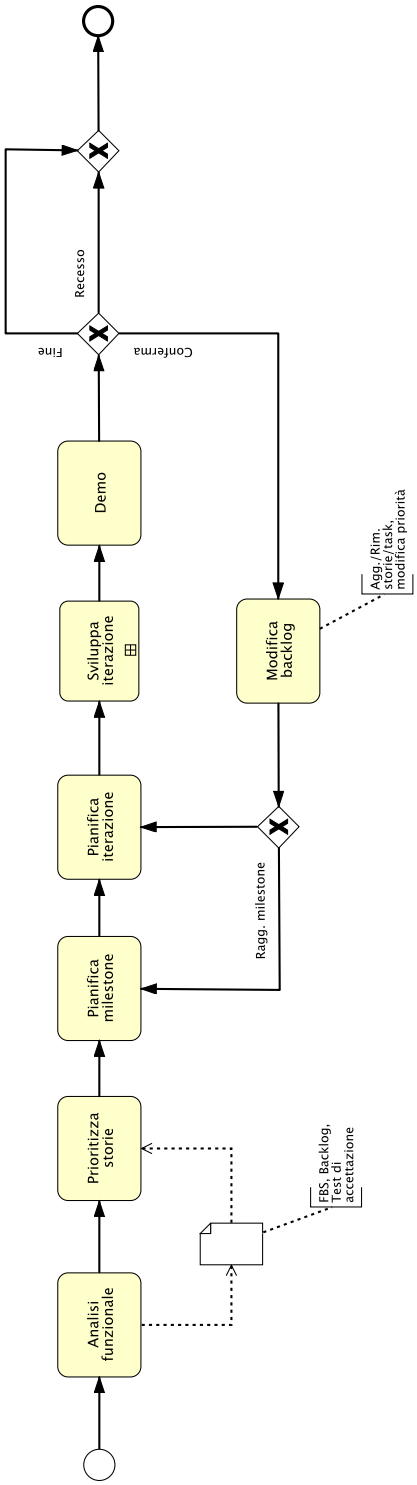


Figura 4.6: Diagramma generale del processo agile

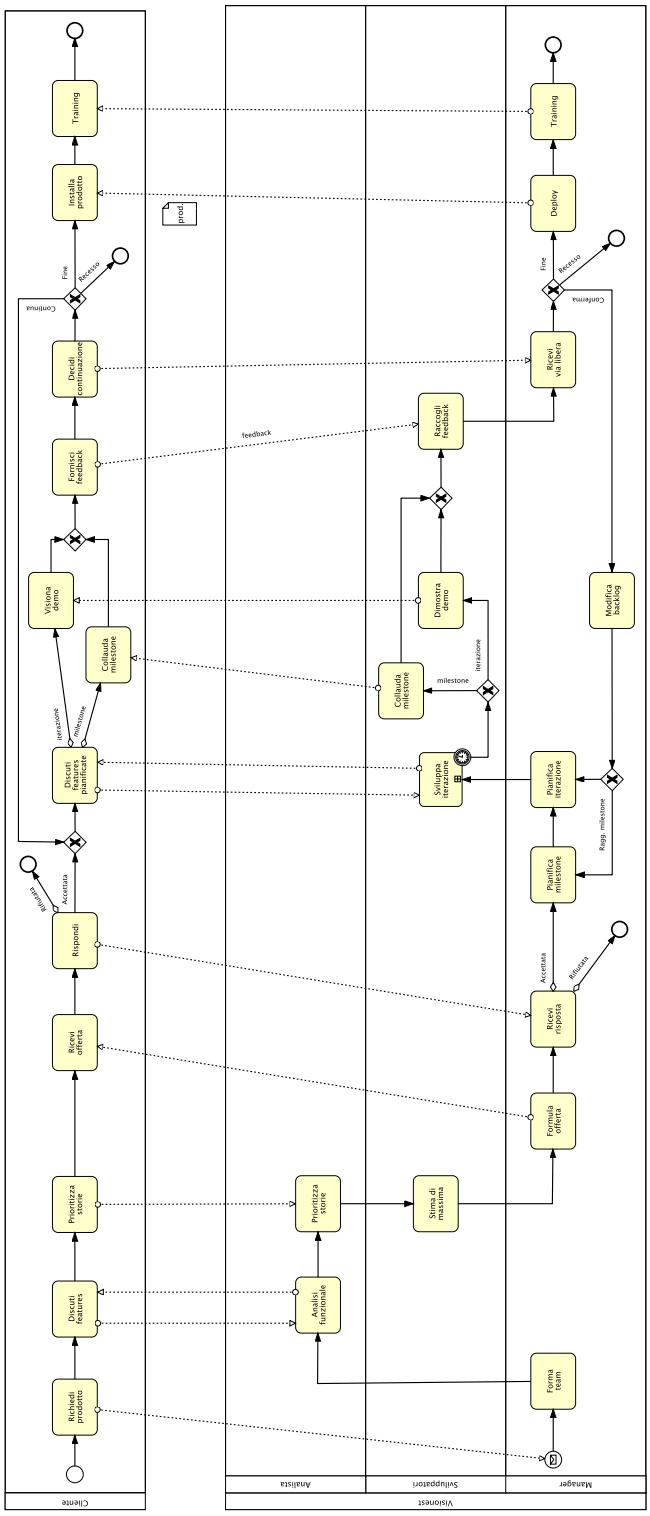


Figura 4.7: Diagramma B2B del processo agile

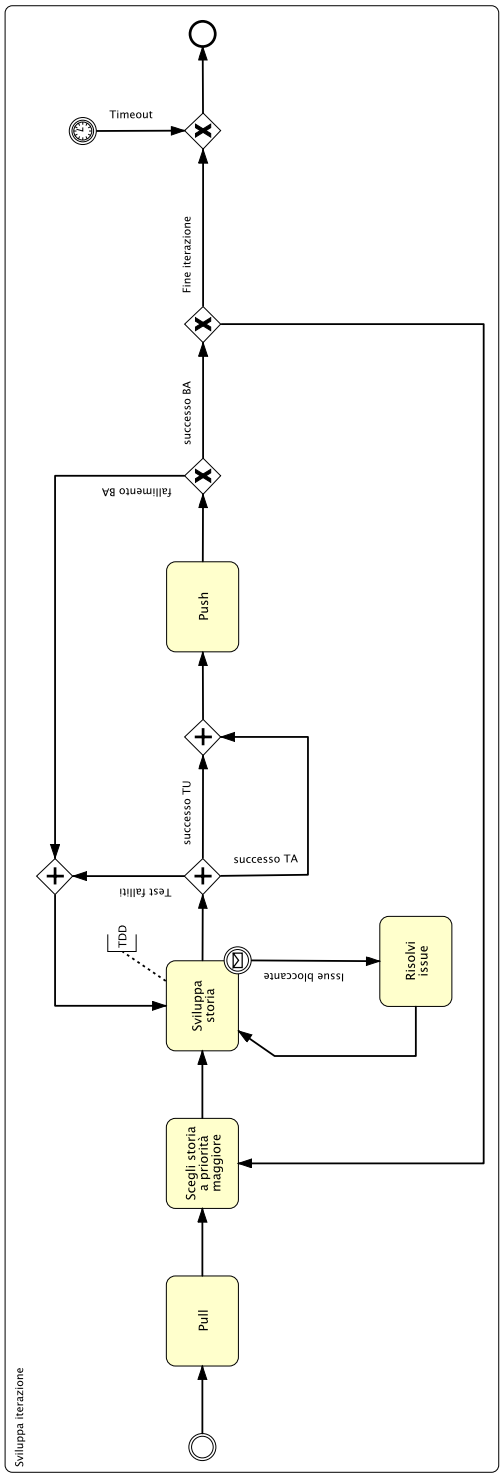


Figura 4.8: Diagramma dell'attività sviluppo iterazione

Appendice A

Effettuare una presentazione

Il presente capitolo ha lo scopo di introdurre il lettore alla preparazione e all'esecuzione di presentazioni riguardo prodotti, idee e soluzioni. L'argomento è stato studiato al fine di effettuare con successo una presentazione all'intera azienda al termine dello stage per illustrare il lavoro svolto.

Per approfondimenti sull'argomento, consultare [57, 58, 15].

A.1 Capacità di comunicazione

Per presentare[1] in modo convincente, efficace e coinvolgente qualcosa (informazioni, prodotti, soluzioni) non basta essere degli esperti in materia, servono delle competenze specifiche. *Ogni buona comunicazione è contenuti più presentazione.*

La capacità di comunicare efficacemente, è un'abilità che può e dovrebbe essere sviluppata: per "vendere" idee, comunicare con il management, colleghi, clienti, e in generale chiunque, servono abilità che spaziano dalla persuasione, al *marketing*, fino al parlare in pubblico.

> "*Presentation skills are worthy of extreme obsessive study.*"
> –Tom Peters

> "*Presentation is the 'Killer Skill' we take into the real world. It's almost an unfair advantage.*"
> –The McKinsey Mind

Chi esegue una presentazione deve avere *rispetto* per sè e per il pubblico. Troppo spesso invece gli *speaker* dimostrano un'insensibilità totale verso

[1] Per estensione, quindi, anche insegnare.

l'*audience* esponendo gli argomenti in modo confuso, incomprensibile e noioso, subissando gli ascoltatori con aridi e lunghi elenchi puntati nello sforzo di fare bella figura e dimostrare la propria "conoscenza".

> "*The deepest human need is the need to be appreciated.*"
> –William James

E' essenziale riconoscere al pubblico il rispetto che merita il suo tempo e la sua attenzione, veicolando le informazioni in modo conciso, chiaro e interessante. Per avere un'idea di cosa significhi una buona presentazione:

- Vedere su *YouTube*[2] le presentazioni di: Seth Godin, Tom Peters, Guy Kawasaki, Steve Jobs e Dick Hardt.
- Guardare le presentazioni su *TED.com*[3] di, ad esempio: Rives, Hans Rosling, Barnett Thomas, Lawrence Lessig e Ken Robinson.
- Se si ha occasione, assistere a qualche talk di *Damian Conway*[4].

A.2 Preliminari

Troppo spesso si comincia subito a preparare l'esposizione senza alcuna idea di partenza di dove si andrà a parare. Conoscere l'obiettivo di una presentazione è indispensabile per esprimere con efficacia il messaggio principale e per preparare un discorso con un suo filo logico.

E' bene anche informarsi sul pubblico[5] e sapere quanto tempo si ha a disposizione, così da da poter preparare una presentazione adatta al contesto e veicolare il messaggio nel modo più incisivo possibile.

> "*Focus on your purpose, your message and your* audience *and you won't go far wrong. Once you are clear on those, the details of tools and delivery will become apparent.*"
> –Rowan Manahan

A.3 Scopo

Una buona presentazione ha l'obiettivo di comunicare qualcosa in modo interessante e coinvolgente (e possibilmente che sia anche utile), non di annoiare a morte il pubblico.

[2] http://www.youtube.com/
[3] http://www.ted.com
[4] http://damian.conway.org/
[5] Chi è? Quali sono i suoi interessi, lo stato d'animo, i bisogni e le aspettative?

Un'ottima e memorabile presentazione non è un arido elenco di fatti, ma un'*esperienza* che coinvolge sia intellettualmente che emotivamente. Ogni *slide* (se ce ne sono, vedi sezione A.4.4) deve essere come come una composizione poetica e la presentazione come una rappresentazione teatrale i cui scopi sono:

- Informare
- Educare
- Intrattenere

Un *talk* non è l'occasione adatta per essere esaustivi. Non senza sfinire e frustrare il pubblico. Tutti i dati, tabelle, diagrammi e informazioni aggiuntive si possono distribuire in separata sede attraverso un *medium* diverso[6]. Lo scopo di una presentazione, come mezzo di comunicazione, è quello di *ispirare*, *entusiasmare* e *spingere all'azione*, non di essere esaustiva. Altrimenti si chiamerebbe seminario.

Per catturare veramente l'attenzione e rendere un'esposizione memorabile si dice che bisogna catturare sia il cuore che la mente, comunicando una *visione* che faccia appello anche alla parte emozionale della nostra mente. Serve un *senso di scopo*. Naturalmente il contenuto è importante, ma sono l'entusiasmo, la passione, l'emozione e il senso di una *missione* che fanno la differenza durante una presentazione. Sono principi essenziali per *motivare* sia all'ascolto che all'azione sulla base delle informazioni presentate.

Infatti, nell'applicare alla mia situazione i concetti riassunti in questa appendice, ho preparato una presentazione per illustrare a grandi linee[7] il lavoro svolto, riservando i dettagli per questa tesi.

A.4 Preparazione

Il modo migliore di preparare una presentazione è quello di farlo in modo *analogico*: **con carta e penna, lontani dal PC e da *Powerpoint***. Il protagonista deve essere l'esposizione chiara, logica e ad effetto di alcuni punti chiave. Il mettersi subito al computer distrae facilmente dal fatto che anche una presentazione ad effetto ma senza una struttura e una sostanza alla base è inutile.

[6] Ad esempio preparando un articolo o una dispensa da distribuire o pubblicare su internet.

[7] Raccogliendo il *feedback* sulla stessa al termine, è emerso che ho comunque ecceduto nei dettagli.

Per prima cosa quindi, occorre stendere l'ossatura della presentazione raccogliendo materiale, facendo schemi e quant'altro. Alla fine deve essere creata, come minimo, un'*outline* o una *mappa mentale*[8] che costituisca l'anima del talk. Può essere utile cercare di esporre a voce senza avere una struttura fissata e annotare le idee migliori che vengono in mente. Inoltre, se si decide di utilizzare delle *slide* è bene preparare qualche schizzo durante questa fase.

Un passo importante che molti non considerano è che un pezzo scritto prima di raggiungere la sua forma definitiva viene modificato molte volte. Lo stesso vale per i discorsi: per renderli concisi ma evocativi e memorabili è necessario scriverli e perfezionarli esercitandosi molto nell'esposizione. Uno dei migliori "presentatori" contemporanei, *Damian Conway*[9], raccomanda un *minimo di 10 ore di preparazione per ogni ora di presentazione* (20 se l'argomento è relativamente ostico).

Inoltre, tenere presente che il modo migliore di illustrare certi concetti è attraverso delle *dimostrazioni*. Se rilevanti e possibili, inserirne il più possibile.

A.4.1 Idee chiave

Per guidare la preparazione e stabilirne l'*idea chiave* che si vuole comunicare con la presentazione e su cui ci si vuole focalizzare (l'obiettivo), è opportuno creare una breve descrizione (una singola frase) che ne catturi l'essenza. L'idea centrale costituisce il tema attorno a cui ruota l'intera presentazione. Una volta identificato, raccogliere citazioni, aneddoti, fatti a supporto e organizzarli per costruire una *storia* attorno ad esso. L'intera esposizione che si andrà a delineare sarà quindi costituita da dettagli a supporto di tale tema centrale.

E' utile renderlo esplicito all'inizio della presentazione in modo che il pubblico possa vedere il quadro generale prima di osservare i dettagli, che potranno così essere inseriti nel contesto giusto.

In generale è bene seguire la *regola del tre*, la quale dice che non bisogna inserire in una presentazione più di 3 o 4 idee chiave. La ragione è che la memoria a breve termine nella quale vengono inserite dagli spettatori, è in grado di tenerne in memoria solo quel numero. Una sola idea chiave è troppo poco per poter fare una esposizione memorabile, e più di 4 sono difficili da ricordare e tendono a rendere complicato seguire il discorso.

[8] http://en.wikipedia.org/wiki/Mind_map
[9] http://damian.conway.org/

A.4.2 Benefici

Nel presentare qualcosa, è molto importante focalizzarsi sui suoi *benefici* per chi ascolta. A nessuno importa del prodotto in sè, quanto dei vantaggi che può portare. Troppo spesso si presenta un prodotto descrivendo con dovizia di particolari le tecnologie impiegate, ma quello che interessa prima di tutto è quali bisogni soddisfa.

Chiedersi continuamente durante la preparazione: *"E allora? Quali sono i benefici?"* Bisogna cercare di osservare i concetti dal punto di vista del pubblico, in modo da poter inserire e spiegare nel modo giusto quelli più utili, senza sprecare fiato e perdere tempo in dettagli ininfluenti per gli ascoltatori.

Ad esempio, nell'illustrare dei miglioramenti infrastrutturali a dei manager, piuttosto di dire "questo nuovo strumento permetterà agli sviluppatori di diminuire il tempo di *turnaround* delle issue collegandosi direttamente all'IDE" è molto più efficace: "questo nuovo strumento consentirà di risparmiare il 15% del budget destinato ad ogni progetto".

A.4.3 Struttura

E' importante che la presentazione abbia una struttura il più possibile logica, lineare, chiara e coerente con un inizio, uno svolgimento e una fine. Esporre con efficacia qualcosa significa *raccontare una storia*, possibilmente con tutti gli elementi che possono renderla memorabile: un eroe, una minaccia e un'*escalation* di *suspense* che porti ad un climax (il messaggio chiave).

> "*Bad storytelling is beginning, muddle, end.*"
> –Philip Larkin

Gli esseri umani amano e trovano più semplice seguire ragionamenti lineari, e tale deve essere la presentazione: *una sequenza lineare, logica e concisa di dati, informazioni e aneddoti correlati.* Bisogna resistere alla tentazione di farcire il talk di dati per dimostrare la propria conoscenza. Quasi invariabilmente si è spinti a inserire *troppo* materiale, conviene quindi distillare la presentazione in modo quasi brutale eliminando tutto quanto non è essenziale. E' molto utile illustrare questa struttura all'inizio della presentazione per dare un'idea di cosa tratterà il talk e come.

> "*La semplicità è la sofisticazione ultima.*"
> –Leonardo Da Vinci

Un efficace accorgimento per rendere più incisiva e memorabile la storia è creare una contrapposizione tra un "nemico" da sconfiggere (prodotti o

idee concorrenti, problemi, ecc) e un "eroe" (la propria soluzione, idea o prodotto). Aggiunge "drammaticità" e *suspense* all'evento, consentendo di presentare l'oggetto della presentazione come l'eroe che salverà la giornata.

Il linguaggio dovrebbe essere il più semplice e diretto possibile, evitando termini gergali. Esprimere concetti anche complicati semplicemente è una grande forza, complicarli con termini oscuri e gergali no. Niente esibizionismo!

Se si vogliono rappresentare dei numeri, per renderli più incisivi è utile metterli in un contesto, rapportandoli con qualcosa che renda l'idea della loro magnitudine.

Dato che in media la soglia di attenzione del pubblico è più alta nei primi e negli ultimi 10 minuti in una presentazione di un'ora, i punti chiave della presentazione dovrebbero essere riassunti alla fine ed eventualmente anche all'inizio, in modo che vengano ricordati.

Un possibile *workflow* per la preparazione potrebbe essere:

1. Identificare i punti rilevanti per l'audience.
2. Metterli nell'ordine più interessante per il pubblico.
3. Fornire il contesto necessario (raccogliere materiale a supporto).
4. Arricchire con elementi di teatralità.
5. *Distillare ulteriormente.*

A.4.4 Slides

L'uso di slide (lucidi) è diventato ormai sinonimo di presentazione in molti campi, tanto che la stragrande maggioranza si cimenta nel realizzare questi *ausili* senza pensarci un istante. Il problema, soprattutto con slide contenenti molto materiale, è che ad un certo punto o la gente legge le slide e non segue più lo speaker (a quel punto converrebbe scrivere un semplice articolo), oppure ascolta lo speaker e non guarda nemmeno le slide (in questo caso non servono neppure). I lucidi non devono essere una fonte di distrazione, ma supportare e arricchire l'esposizione senza appesantirla.

Ripensando all'obiettivo della presentazione e al contesto, occorre essere pragmatici e riflettere sull'utilità delle slide: *il principale asset e contenuto di un talk è l'esposizione di un'argomento dal vivo e dalla voce dello speaker. Le slide sono sono un ausilio.* Se il pubblico volesse leggere una lista di punti proiettati su uno schermo andrebbe semplicemente a leggersi un libro o un articolo su internet. In breve: *prima viene il messaggio e la sua struttura, dopo (eventualmente) gli ausili visivi.*

Molti realizzano una raccolta di lucidi pieni di testo con l'intento di distribuirla poi come una dispensa, e utilizzano lo stesso documento sia per proiettarlo durante il talk sia come *handout*[10]. Se si vuole rendere disponibile del materiale di riferimento, non bisogna usare le slide per questo perché non sono il mezzo adatto.

Alcuni creano le slide per sentirsi più sicuri durante l'esposizione, ma i lucidi non possono essere un sostituto per la conoscenza dell'argomento che si presenta. O lo si conosce o non lo si conosce. Usare le slide come "stampelle" è la ragione sbagliata per propinarle e una chiara indicazione del bisogno di fare più pratica (vedi sezione A.4.5). Un argomento lo si conosce e lo si comprende veramente solo quando si è in grado di spiegarlo senza alcun ausilio.

> "*Non hai capito veramente qualcosa finché non sei in grado di spiegarlo a tua nonna.*"
> –Albert Einstein

Se si decide di utilizzarle il modo migliore per farlo è per sottolineare ed enfatizzare i punti importanti in modo *visuale*. L'effetto migliore si ottiene quando le slide sono poche, contengono principalmente immagini[11] e pochissime parole. Come esempio, vedere qualche presentazione (specialmente degli ultimi anni) di Steve Jobs e di Garr Reynolds[15].

La ragione per cui questo *format* funziona così bene è un fenomeno chiamato *picture superiority*: le informazioni sono richiamate alla mente più facilmente quando il testo (le eventuali parole chiave), le immagini e il parlato sono combinate in un'unica comunicazione (comunicazione *multi-modale*).

> "*Great content is a necessary condition, but not a sufficient one.*"
> –Garr Reynolds[58]

Flusso di informazioni

Se si decide di utilizzare delle slide come ausilio, bisogna controllare il flusso di informazioni in modo che il talk e i lucidi restino in sincrono.

Quando ad esempio si vogliono elencare una serie di punti, il mostrarli tutti contemporaneamente in un'unica slide mentre si parla, significa fare in modo

[10] Ovvero del materiale che di solito si consegna al pubblico al termine di una presentazione per fornire un riassunto del messaggio chiave ed eventualmente dettagli aggiuntivi.

[11] Rilevanti al contesto. L'effetto è ancora più potente quando costituiscono una *metafora* di concetti astratti per renderli più concreti e magari divertenti.

che l'*audience* li legga tutti e aspetti che vengano trattati cercando di collegare ciò che è scritto a ciò che viene detto. Questo a scapito dell'attenzione. E' più opportuno in questo caso mostrare un punto alla volta.

Allo stesso modo, se si vuole presentare un grafico, per prima cosa occorre spiegare i dati e il significato prima di mostrarlo nella slide successiva, in modo che a quel punto gli ascoltatori siano in grado di interpretarlo subito. Tra l'altro, il mostrare il grafico dopo averlo spiegato può aiutare anche a creare anticipazione e innalzare l'interesse.

Design

E' importante che le slide siano leggibili e rispettino delle buone linee guida tipografiche. Se c'è qualcosa di peggio di slide mal realizzate, sono slide mal realizzate *e* illeggibili. Ecco le linee guida[58] più importanti:

- *Semplicità e stile!*
- *Attenzione al contrasto*: colori complementari. Sfondo chiaro, testo scuro. Sfondo scuro, testo chiaro.
- *Allineare il testo a destra o a sinistra*: l'allineamento centrale è poco professionale, amatoriale e meno leggibile.
- *Andare all'essenziale*: Un titolo, qualche punto e al massimo un'immagine. Avere troppi elementi rende difficile "decodificare" la pagina ed è stressante. La cosa migliore è inserire poche parole con un font di dimensione molto grande.
- *Usare font della famiglia Serif per il corpo*: sono i font che rendono meglio su schermo.
- *Limitare l'uso di font decorativi ai titoli e* solo *se sono leggibili.*
- *Evitare di inserire il logo in ogni slide.* Non si vuole dare l'impressione di essere dei venditori.
- *Limitare l'uso di effetti di transizione*, possibilmente non usarne nessuno: distraggono e basta.

A.4.5 Pratica

Una volta costruita la presentazione: *fare pratica, pratica e ancora pratica. Preferibilmente ad alta voce.* Sperimentare spostando concetti, eliminando quelli superflui e modificando la presentazione in modo che *sia ben coesa e comunichi efficacemente il tema centrale.* Possibilmente avendo qualcuno in

carne ed ossa che ascolta e fornisca *feedback*. Inoltre fare attenzione che si rientri nei tempi.

E' molto importante praticare la presentazione, per poi poterla esporre con sicurezza e padronanza davanti al pubblico. Non occorre memorizzare tutto il talk, basta ricordarsi i punti fondamentali, gli accessori verranno con naturalezza se si conosce bene la materia e si è fatta abbastanza pratica.

> "*The only way to speak better is to speak.*"
> –Damian Conway

La pratica rende perfetti. Dopo molto allenamento, una presentazione è rifinita[12], perfezionata, e rappresentata senza apparente difficoltà, in modo naturale (vedi sezione 4.1.1). Persino Steve Jobs[57], che aveva fin dall'inizio un grande carisma, ha rifinito le proprie capacità di presentazione nel corso degli anni.

A.5 Delivery

Presentare qualcosa è più di una semplice recitazione di fatti. Si tratta di una vera e propria esibizione *comprendente elementi teatrali e di drammaticità.* Bisogna imparare a comunicare con efficacia usando non solo le parole ma anche il linguaggio del corpo, il giusto ritmo, le emozioni, la consapevolezza dell'*audience* e di come questi elementi li influenzino. Ogni presentazione è una *performace* finalizzata all'intrattenere il pubblico in modo che al termine ne esca arricchito (possibilmente di quello che si aspettava, meglio se di più).

Se possibile, arrivare prima dell'inizio del talk e preparare tutto prima che il pubblico giunga in sala. Non è molto professionale armeggiare con il proiettore e il portatile mentre l'*audience* aspetta... Prepararsi per tempo in modo da essere pronti, sicuri che tutto funzioni a dovere e (relativamente) rilassati quando arriva l'ora di iniziare.

A.5.1 Apertura

E' normale essere un pò nervosi prima di cominciare un'esposizione. Le cose importanti sono fare molta pratica e iniziare la presentazione con entusiasmo e sicurezza. Ad esempio, per rompere il ghiaccio si può:

- Porre una domanda.

[12] E' la cura nei dettagli che distingue un prodotto *professionale* da uno amatoriale, qualunque esso sia.

- Sfidare l'audience.
- Citare una persona famosa.
- Raccontare una storia divertente.
- Raccontare un fatto interessante.
- Proporre una visione grandiosa (e poi proporre come farla avverare)
- Proporre una visione spaventosa (e poi parlare di come *non* farla realizzare)

In altre parole basta cominciare il talk con qualcosa di interessante che catturi l'attenzione.

A.5.2 Talk (show)

Di seguito qualche suggerimento su come condurre al meglio la *performace*:

- *Muoversi e guardare le persone negli occhi.* Usare il linguaggio del corpo per sottolineare certi punti è utile, ma senza esagerare: non è una prestazione sportiva!
- *Non recitare i punti a memoria o peggio leggere le slide.* Bisogna avere rispetto per il tempo degli altri e dato che catturare l'attenzione non è affatto semplice, una volta che la si ha va gestita correttamente. Si deve conoscere bene ciò che si presenta, e questo significa essere in grado di illustrarlo senza che siano necessari ausili esterni.
- *Usare un tono di voce chiaro, parlare lentamente e variare spesso il ritmo.* Bisogna parlare in modo da essere sentiti, chiaramente e ancora una volta portando rispetto verso il pubblico evitando di annoiarlo a morte con un tono di voce costante e noioso (soporifero). Spesso i novizi durante i talk parlano troppo velocemente: rallentare deliberatamente.
- *Evitare i "filler"*, ovvero cose come: "cioè, uhm, eh" ecc. Denotano insicurezza.
- *Raccontare una storia.* Anche se inventata, è utile per intrattenere e mantenere viva l'attenzione. Ovviamente che sia in qualche modo rilevante all'argomento oggetto della presentazione. Può essere utile anche mischiarla con delle *metafore* utili ad illustrare certi concetti e relazioni. Inutile dire che l'effetto è migliore se la storia è divertente.

- *Usare la* retorica[13] *per essere* incisivi e memorabili. Praticare ad esempio con: contrasto, lista dei tre e soprattutto la *pausa*. Fare una pausa prima di esporre un punto importante crea anticipazione e drammaticità: un momento teatrale che mantiene l'attenzione.

- *Se ci si dimentica qualcosa, andare avanti con la presentazione senza scusarsi o fermarsi per cercare di ricordare.* Se l'esposizione veicola il *messaggio principale*, allora è ok.

- *Emozionare*. E' uno dei modi principali e più efficaci per mantenere l'attenzione e far arrivare i concetti a destinazione. Con un pò di pratica diventa facile visto che qualsiasi argomento può essere trattato dal punto di vista *umano*.

- E' utile anche raccontare aneddoti (soprattutto se personali), umorismo e storie. *Un tocco personale può aiutare molto a rendere una presentazione autentica e interessante*.

- *ENTUSIASMO!* L'entusiasmo è contagioso. La presentazione deve essere colma di *passione*, deve prendere vita e catturare l'attenzione del pubblico facendo perdere al tempo il suo significato.

- *Creare una sensazione di* intimità *con il pubblico*: dare l'impressione di essere lì per loro e di conoscerli trattandoli come amici. L'atmosfera dovrebbe essere rilassata e amichevole.

- Eventualmente *coinvolgere il pubblico* trasformando la presentazione in una discussione interattiva. Invitare a fare commenti e portare contributi personali. Chiaramente è possibile solo per chi conosce veramente quello di cui sta parlando ed è in grado di gestire una discussione, diversamente avrà paura di essere colto in fallo. Comunque tutto dipende dal pubblico e dalla natura dell'argomento: non sempre l'oggetto del talk si presta ad una discussione interattiva e a volte l'*audience* preferisce non essere chiamata in causa.

> "*Humans want to connect. They are built to want to belong. A great presentation is a fire to gather around and share an experience.*"
> –Chris Brogan

Per quanto riguarda le domande, è opportuno avere una *policy* per dove, come e quando accettarle e condividerla con il pubblico fin dall'inizio. Se non si conosce qualche risposta si può reagire in modo diverso a seconda della situazione:

[13] http://it.wikipedia.org/wiki/Retorica

- "Non conosco la risposta mi dispiace."
- Redirigere la domanda a chi può avere la risposta.
- "Ora non so risponderti, non appena conoscerò la risposta la scriverò sul mio blog."

A.5.3 Chiusura

E' importante chiudere la presentazione con un sommario che ricapitola i punti principali e **un'ultima, chiara ripetizione del messaggio chiave**. Il pubblico deve avere la sensazione di aver ottenuto qualcosa dalla presentazione e deve avere delle informazioni *pronte all'uso*.

Conclusioni

Gli obiettivi obbligatori sono stati pienamente raggiunti: l'infrastruttura e i processi sono stati analizzati a fondo, e sono state elaborati e proposti miglioramenti in entrambe le aree.

L'installazione e la messa in opera di nuovi strumenti in larga parte non è stata effettuata, ad eccezione del passaggio all'uso di *repository* di riferimento con *Mercurial*. Di conseguenza la realizzazione di script e automazioni per l'integrazione degli strumenti proposti nell'infrastruttura pre-esistente non si è resa necessaria.

A.6 Discostamento orario

Il discostamento orario tra il piano di lavoro preparato all'inizio dello stage (vedi sezione 1.5) e le ore effettivamente impiegate è riportato nell'istogramma in figura A.1 e in tabella A.1.

	Docum.	**Rilev.**	**Prop.**	**Conclus.**
Piano	60	60	120	60
Effettive	71	49	122	58

Tabella A.1: Discostamento orario dal piano di lavoro

In linea di massima è stato rispettato il piano di lavoro inizialmente stabilito. Da notare che la parte di documentazione iniziale ha richiesto diverse ore in più (per lo studio di libri, articoli e presentazioni), mentre quella di rilevazione qualche ora in meno. Le rilevazioni sono state fatte principalmente attraverso interviste, che sono state effettuate quando possibile, quindi non c'è stata una netta separazione tra le prime due fasi: l'attività di documentazione è continuata anche durante la seconda fase.

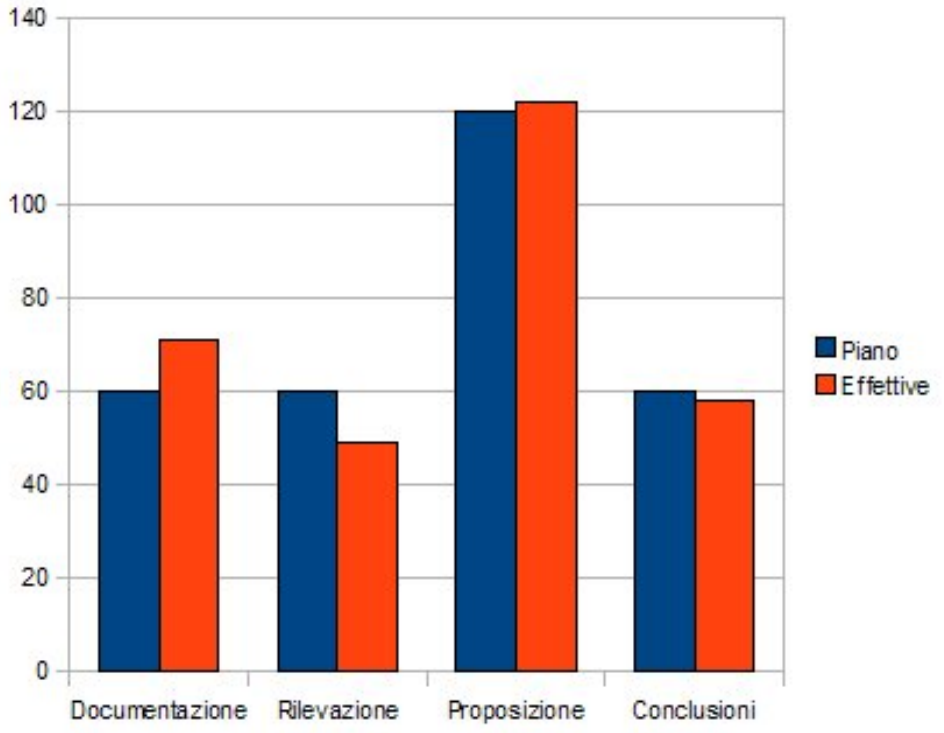


Figura A.1: Discostamento orario dal piano di lavoro

A.7 Commenti

Questo stage, e il relativo lavoro di preparazione della tesi, sono stati molto interessanti per me: coltivavo già un grande interesse sia per lo sviluppo vero e proprio, sia per quanto riguarda l'organizzazione e la pianificazione di progetti, e queste conoscenze si sono rivelate molto utili. Inoltre, le mie passioni per argomenti come il *time management*, la psicologia dell'apprendimento e la comunicazione sono state dei preziosi aiuti per portare a termine con successo questa esperienza lavorativa.

La raccolta di appunti durante lo stage, la preparazione della presentazione, la scrittura della tesi e infine la preparazione della discussione di laurea mi ha permesso di approfondire e capire veramente gli argomenti affrontati. Senz'altro molto meglio che studiandoli esclusivamente sui libri.

In definitiva è stata un'attività di forte valenza didattica, che mi ha anche permesso di osservare una realtà produttiva di alto profilo e di capire, per quanto mi ha permesso il tempo e il compito che mi era stato assegnato, la realtà lavorativa dello sviluppo del software.

Bibliografia

Libri

[1] **Practices Of An Agile Developer**, Venkat Subramaniam, 2005, ISBN: 097451408X, Pragmatic Bookshelf.

[2] **Pragmatic Thinking and Learning: Refactor Your Wetware**, Andy Hunt, 2008, ISBN: (1934356050, 9781934356050), Pragmatic Bookshelf

[3] **Test-Driven Development By Example**, Kent Beck, Addison Wesley, 2002

[4] **The RSpec Book: Behaviour Driven Development with RSpec, Cucumber, and Friends**, David Chelimsky, Dave Astels, Zach Dennis, Aslak Hellesøy, Bryan Helmkamp, Dan North, 2010, ISBN: 978-1-93435-637-1, Pragmatic Bookshelf

[5] **User Stories Applied: For Agile Software Development**, Mike Cohn, Addison Wesley, 2004, ISBN: 0-321-20568-5

[6] **SWEBOK: Software Engineering Body of Knowledge**, IEEE Computer Society, http://www.computer.org/portal/web/swebok

[7] **Business Process Modeling Notation (BPMN)**, Object Management Group, versione 1.2, Gennaio 2009

Presentazioni

[8] **Test-Driven Development: Ten Years Later**, Michael Feathers e Steve Freeman, 2009, InfoQ, http://www.infoq.com/presentations/tdd-ten-years-later

[9] **10 Tips For Successful Agile Transitions**, Joshua Kerievsky, 2009, InfoQ, http://www.infoq.com/presentations/10-tips-for-agile-transitions

[10] **Succeeding With Agile: A Guide To Transitioning**, Mike Cohn, 2008, InfoQ, http://www.infoq.com/presentations/Agile-Transitioning-Mike-Cohn

[11] **Adopting Agile Practices**, Amr Elssamadisy, 2009, InfoQ, http://www.infoq.com/presentations/adopting-agile-practices

[12] **Scaling Agile into the Enterprise**, Mark Ferraro, 2009, InfoQ, ScalingAgileintotheEnterprise

[13] **Agile Project Management**, Tracy Hoerschgen, http://media.govtech.net/GOVTECH_WEBSITE/EVENTS/PRESENTATION_DOCS/2009/Missouri/Hoerschgen_Agile_Project_Management.pptx

[14] **The Surprising Science Of Motivation**, Dan Pink, TED Talks, http://www.ted.com/talks/dan_pink_on_motivation.html

[15] **Presentation Zen**, Garr Reynolds, Google Talks, http://youtube.com/watch?v=DZ2vtQCESpk

Articoli

[16] **IEEE Std 610.12-1990: IEEE Standard Glossary of Software Engineering Terminology**, IEEE, 1990, http://standards.ieee.org/reading/ieee/std_public/description/se/610.12-1990_desc.html

[17] **The Risks Digest: Forum On Risks To The Public In Computers And Related Systems**, ACM Committee on Computers and Public Policy, http://catless.ncl.ac.uk/Risks

[18] **No Silver Bullets - Essence and Accidents of Software Engineering**, Fred Brooks, 1986, http://www.virtualschool.edu/mon/SoftwareEngineering/BrooksNoSilverBullet.html

[19] **CHAOS Report 1995**, The Standish Group, 1995, http://www.projectsmart.co.uk/docs/chaos-report.pdf

[20] **CHAOS Report 2009 Summary**, The Standish Group, 2009, http://www.standishgroup.com/newsroom/chaos_2009.php, http://www.statelibrary.state.pa.us/portal/server.pt/document/690719/chaos_summary_2009_pdf

[21] **Manage By Features**, http://www.agile-process.org

[22] **The Agile Manifesto**, AA.VV., http://agilemanifesto.org/

[23] **Software Engineering: An Idea Whose Time Has Come And Gone?**, Tom DeMarco, IEEE Software, Column: Viewpoints, http://www2.computer.org/cms/Computer.org/ComputingNow/homepage/2009/0709/rW_SO_Viewpoints.pdf

[24] **Design, Development, Integration: Space Shuttle Primary Flight Software System**, WA Madden, KY Rone, Communications of the ACM, 1984, http://klabs.org/DEI/Processor/shuttle/madden_rone.pdf

[25] **On the Effectiveness of Test-first Approach to Programming**, Hakan Erdogmus e Torchiano Morisio, Proceedings of the IEEE Transactions on Software Engineering, http://iit-iti.nrc-cnrc.gc.ca/publications/nrc-47445_e.html

[26] **Realizing quality improvement through test driven development: results and experiences of four industrial teams**, Nachiappan Nagappan & E. Michael Maximilien & Thirumalesh Bhat & Laurie Williams, Empirical Software Engineering, Vol. 13, No. 3, pp. 289-302, http://www.springerlink.com/index/Q91566748Q234325.pdf

[27] **About the Return on Investment of Test-Driven Development**, Matthias M. Müller e Frank Padberg, Universität Karlsruhe, Germania, http://www.ipd.uka.de/mitarbeiter/muellerm/publications/edser03.pdf

[28] **Introducing BDD**, Dan North, http://dannorth.net/introducing-bdd

[29] **A New Look At Test-Driven Development**, Dave Astels, http://blog.daveastels.com/files/BDD_Intro.pdf

[30] **Big Ball Of Mud**, Brian Foote e Joseph Yoder, Department Of Computer Science, University Of Illinois, 1999, http://www.laputan.org/mud/

[31] **Introduction to BPMN**, Stephen A. White, IBM Corporation

[32] **Process Modeling Notations and Workflow Patterns**, Stephen A. White, IBM Corporation

[33] **Managing the Work in an Agile Project**, Dan Rawsthorne, PhD

[34] **Feature Estimation**, VersionOne: Agile101, http://www.versionone.com/Resources/FeatureEstimation.asp

[35] **Where is the Work Breakdown Structure (WBS) in Agile?**, The PMI Agile Community of Practice Wiki, http://agile-pm.pbworks.com/Agile-WBS

[36] **Scrum in a nutshell**, http://www.agile42.com

[37] **Introduction to scrum: agile development and his principles**, Sprint It

[38] **Introduction to Lean Software Development**, Corey Ladas, http://shapingsoftware.com/2009/06/15/introduction-to-lean-software-development/

[39] **Lean Thinking Software**, Jackly Li, http://www.infoq.com/articles/lean-thinking-software

[40] **Lean and Agile: Marriage Made in Heaven or Oxymoron?**, Dave West, http://www.infoq.com/articles/backlog-not-waste

[41] **A Five-Stage Model Of The Mental Activities Involved In Directed Skill Acquisition**, Stuart E. Dreyfus, Hubert L. Dreyfus, Febbraio 1980, Berkley University of California

[42] **From Novice to Expert: Excellence and Power in Clinical Nursing Practice**, Patricia Benner, Prentice Hall, Englewood Cliffs, NJ, commemorative edition, 2001

[43] **Wrong and right reasons to apply Kanban**, Vikas Hazrati, http://www.infoq.com/news/2009/09/reasons-for-adopting-kanban

[44] **Ruining your test automation strategy**, Robert C. Martin, http://blog.objectmentor.com/articles/2009/09/29/ruining-your-test-automation-strategy

[45] **The big list of agile practices**, Jurgen Appelo, http://www.noop.nl/2009/04/the-big-list-of-agile-practices.html

[46] **Virtual panel: the evolution of bug trackers**, InfoQ, http://www.infoq.com/articles/bug-trackers

[47] **Continuous Integration in the Cloud With Hudson**, Janice J. Heiss, http://java.sun.com/javaone/2009/articles/gen_hudson.jsp

[48] **State secrets**, David Bock, PragPub 10-2009

[49] **Opinion: Refactoring is a Necessary Waste**, Amr Elssamadisy, http://www.infoq.com/news/2007/12/refactoring-is-waste

[50] **Tools for Agility**, Kent Beck, http://www.microsoft.com/downloads/details.aspx?FamilyId=AE7E07E8-0872-47C4-B1E7-2C1DE7FACF96&displaylang=en

[51] **Fitnesse Tutorials**, Brett L. Schuchert, http://schuchert.wikispaces.com/FitNesse.Tutorials

[52] **Fitnesse User Guide**, Robert C. Martin, Micah D. Martin, Patrick Wilson-Welsh, http://fitnesse.org/.FitNesse.UserGuide

[53] **A Metric Leading to Agile**, Ron Jeffries, http://xprogramming.com/xpmag/jatrtsmetric/

[54] **Agile Stories: Agile Systems and Narrative Research** (Research Summary), Johanna Hunt, Pablo Romero, e Judith Good, PPIG Newsletter, Settembre 2006

[55] **Exploring User Stories through Mind Mapping**, Kenji Hiranabe,

[56] **Are Agile Lifecycle Management tools worth it?**, Bob Hartman, http://www.agileforall.com/2008/10/03/are-agile-lifecycle-management-tools-worth-it/

[57] **The Presentation Secrets of Steve Jobs**, Carmine Gallo, GoToMeetig, http://learn.gotomeeting.com/forms/NA-G2MC-WP-Mrk-Gallo-PesentationSecrets-S?ID=701000000005HQB

[58] **Presentation Tips**, Garr Reynolds, Handout, http://www.garrreynolds.com/Presentation/index.html

[59] **How to give a great speech, Part 1: Preparation**, Paul Sloane, http://www.lifehack.org/articles/communication/how-to-give-a-great-speech-part-1-preparation.html

[60] **How to give a great speech, Part 2: Delivery**, Paul Sloane, http://www.lifehack.org/articles/communication/how-to-give-a-great-speech-part-2-delivery.html

[61] **Presentation Masterclass - Part 1: Introduction**, Rowan Manahan, http://www.lifehack.org/articles/communication/presentation-masterclass-part-1-introduction.html

[62] **Presentation Masterclass - Part 2: 5 Key Questions When Planning Your Presentation**, Rowan Manahan, http://bit.ly/8MBlE8

[63] **Six Ways To Transform Your Presentation**, Paul Sloane, http://www.lifehack.org/articles/communication/six-ways-to-transform-your-presentation.html

[64] **4 Effective Presentation Techniques**, Raj Pooyath, http://www.lifehack.org/articles/communication/4-effective-presentation-techniques.html

[65] **LifeHack How-To: Presentation**, LifeHack, http://howto.lifehack.org/wiki/Presentation

[66] **Present Like A Rockstar**, Chris Brogan, http://www.lifehack.org/articles/communication/present-like-a-rockstar.html

[67] **My Best Presentation Tricks**, Chris Brogan, http://www.lifehack.org/articles/lifehack/my-best-presentation-tricks.html

[68] **18 Tips for Killer Presentations**, Scott H. Young, http://www.lifehack.org/articles/communication/18-tips-for-killer-presentations.html

[69] **10 Tips for More Effective PowerPoint Presentations**, Dustin Wax, http://www.lifehack.org/articles/technology/10-tips-for-more-effective-powerpoint-presentations.html

[70] **A few more Presentation How To's**, Kathy Sierra, http://headrush.typepad.com/creating_passionate_users/2006/07/a_few_more_pres.html

[71] **Let there be stoning!**, Lehr, J.H., 1985, Ground Water, v. 23, no. 2, p. 162-165, http://geology.wwu.edu/rjmitch/stoning.pdf

[72] **OSCON 2005: Presentation Aikido, Damian Conway, first half**, Brad Cavanagh, http://www.canspice.org/2005/08/01/oscon-2005-the-conway-channel-first-half/

[73] **OSCON 2005: Presentation Aikido, Damian Conway, second half**, Brad Cavanagh, http://www.canspice.org/2005/08/01/oscon-2005-presentation-aikido-damian-conway-second-half/