Università degli Studi di Padova

FACOLTÀ DI INGEGNERIA
Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea Magistrale

Diffusione delle metodologie di Agile Software Development

Risultati di una survey

22 aprile 2013

Candidato:
Andrea Donè
Matricola 607268-IF

Relatore:
Ch. Prof. Moreno Muffatto
Dip. di Innovazione Meccanica e Gestionale

Anno Accademico 2012–2013

A Elena
perché questo traguardo
sia un punto di partenza
per un futuro insieme.
E, ovviamente, alla mia famiglia.

INDICE

ELENCO DELLE FIGURE

ELENCO DELLE TABELLE

1 INTRODUZIONE

Al giorno d'oggi le applicazioni software sono parte integrante della nostra vita. Basti pensare a ciò che permette il funzionamento non solo di computer o telefoni cellulari, ma anche di automobili, elettrodomestici, giocattoli, sistemi di produzione industriale, ecc. Questa lista potrebbe continuare all'infinito. Per questo motivo la produzione di software di qualità e nel minore tempo possibile svolge un ruolo di fondamentale importanza all'interno dell'economia mondiale.

Da anni esistono metodologie a supporto della produzione software che guidano sviluppatori e responsabili, lungo tutto il processo di realizzazione di applicazioni e sistemi, dalla raccolta dei requisiti alla distribuzione al cliente (o all'utilizzatore finale). Alcune di queste metodologie promettono una grande agilità nella gestione dei processi di sviluppo software, specie nelle situazioni in cui i requisiti sono incerti o sono facilmente soggetti a cambiamento; consentono inoltre una maggiore velocità nel rilascio di software funzionante senza tralasciare la qualità del prodotto finale. Questi metodi sono conosciuti in tutto il mondo con il nome di metodologie di *Agile Software Development* (o metodologie agili) e nascono negli anni '90 per superare i limiti delle metodologie dette "pesanti" (es. modello a cascata) basate fortemente su un approccio di tipo prescrittivo. I metodi di *Agile Software Development* seguono un approccio di sviluppo di tipo iterativo e incrementale e si basano su principi quali l'auto-organizzazione del team di sviluppo, alto livello di collaborazione fra i membri e l'idea fondamentale di accogliere il cambiamento dei requisiti anche a stadi avanzati dello sviluppo software.

Risulta quindi interessante la realizzazione di uno studio sulla diffusione di questo tipo di tecniche per ottenere informazioni sulla effettiva adozione di questi metodi, e valutarne le caratteristiche comparandole ad approcci di tipo più tradizionale. Con questo scopo in mente si è pensato di realizzare, tramite la redazione di un questionario diffuso ad un campione di professionisti coinvolti nella produzione di software, un'indagine che potesse permettere di ottenere una serie di informazioni pre-

liminari sulla diffusione del fenomeno delle *Agile Software Development*. I risultati ottenuti potranno fornire un'idea preventiva che potrà rappresentare il punto di partenza per ulteriori approfondimenti sul tema delle metodologie agili per lo sviluppo software.

OBIETTIVO

L'obiettivo principale che ci si propone in questo lavoro di tesi è quindi avviare uno studio esplorativo per valutare la conoscenza e la diffusione delle metodologie di *Agile Software Development* sottoponendo un questionario ad un campione di professionisti che operano nel settore dell'*Information Technology*. Verranno analizzate varie caratteristiche relative alla tipologia di sviluppo software impiegata, assieme i livelli di adozione di alcune delle pratiche caratteristiche delle metodologie agili. Questo tipo di studio potrà fornire utili informazioni, in particolare si potrà valutare se la diffusione di questo tipo di pratiche agili hanno influenzato il modo di lavorare anche di coloro che non adottano dichiaratamente metodologie di *Agile Software Development*.

ORGANIZZAZIONE DEI CAPITOLI

Nel Capitolo 2 si introdurranno i *modelli a processo prescrittivo* (conosciuti anche come metodologie tradizionali di sviluppo software o metodologie pesanti) e di come hanno portato ordine nel campo dell'ingegneria del software, mentre nel Capitolo 3 parleremo delle metodologie di *Agile Software Development*, nate per superare le limitazioni delle rigide metodologie pesanti e per "reagire al cambiamento". Il capitolo 4 introdurrà la survey che è stata condotta tramite questionario e descriverà le domande che sono state poste agli intervistati. Nel Capitolo 5 si discuterà l'analisi effettuata sui dati ottenuti dall'indagine e verranno presentati i risultati ottenuti. Infine il Capitolo 6 riassumerà il lavoro svolto e proporrà alcune idee per possibili lavori futuri.

2 MODELLI PRESCRITTIVI DI SVILUPPO SOFTWARE

In questo capitolo verranno introdotti i modelli a processo prescrittivo [27]. Questi modelli, conosciuti anche come "modelli convenzionali" o "metodi tradizionali" di sviluppo software, trovano le loro radici nella seconda metà del 1900 quando, a causa della crescente complessità dei prodotti software, gli sviluppatori si resero conto di avere bisogno di una serie di linee guida che potessero permettere loro di organizzare in maniera controllata e razionale il processo di sviluppo software. I modelli a processo prescrittivo introducono elementi di stabilità, controllo e organizzazione in un'attività che se lasciata incontrollata può diventare caotica e difficilmente governabile.

2.1 IL MODELLO A CASCATA

Il modello a cascata (*Waterfall Model*), chiamato anche *Classic Life Cycle*, è un approccio sequenziale e lineare allo sviluppo software. Esso trova le sue origini negli anni '50, quando l'attività di sviluppo di software iniziò ad affermarsi come una vera e propria attività di produzione industriale. A quel tempo, non essendo presente nessuna metodologia di realizzazione software, gli sviluppatori si ispirarono ai processi di produzione manifatturiera e alle industrie di costruzione, per ottenere una metodologia che potesse essere applicata allo sviluppo di codice in modo ordinato e meno caotico.

Le prime tracce del modello a cascata si possono incontrare in una pubblicazione del 1956 di Herbert D. Benington [7]. In questo articolo egli descrive una struttura sequenziale formata da diverse fasi, che è stata impiegata per lo sviluppo del complesso sistema *S.A.G.E.* (*Semi-Automatic Ground Environment*) per la difesa americana.

Successivamente in un articolo del 1970, l'informatico Winston Royce descrive formalmente il modello a cascata [29]. Anche se nell'articolo Royce non cita mai la parola "cascata", tale metodologia è conosciuta

in tutto il mondo con questo nome a causa della particolare struttura delle varie attività che la compongono, dove il flusso di sviluppo scorre in maniera lineare da una fase a quella successiva. Ciò significa che l'output prodotto dal primo stadio, sarà l'input per quello che segue. L'esecuzione lineare di tutte le fasi produce in output il prodotto software finale.

Nella bibliografia si possono trovare delle varianti del modello a cascata. Alcune descrivono il suddetto modello con più fasi di sviluppo, altre con meno. Consultando il libro "*Software Engineering: a practitioner's approach*" [28] scritto dall'americano Roger Pressman, il quale si occupa di ingegneria del software, si può vedere come egli descriva un generico processo di sviluppo software con un preciso framework di attività: comunicazione, pianificazione, modellazione, costruzione e dispiegamento.

- **Comunicazione**: È la fase iniziale che avvia la sequenza di attività. Vi è un grande scambio di informazioni fra committente e sviluppatori che ha lo scopo di far emergere i requisiti che il prodotto richiesto deve soddisfare. Questi requisiti vengono riportati su carta e vanno a formare la documentazione iniziale.

- **Pianificazione**: Durante questa fase viene redatto un piano per stabilire ciò che dovrà essere fatto nelle fasi successive. Vengono descritte le operazione da svolgere, identificate le risorse necessarie (umane, monetarie, hardware, temporali, ecc.) e pianificato il lavoro da svolgere.

- **Modellazione**: Dall'analisi dei requisiti, viene progettato un modello rappresentante il sistema software che deve essere implementato. Ciò permetterà al cliente e agli sviluppatori di verificare l'effettiva comprensione dei requisiti software e di progetto.

- **Costruzione**: Seguendo le specifiche definite nelle fasi precedenti, il prodotto richiesto viene implementato dal team di sviluppo. A questo segue il collaudo per verificare che il sistema soddisfi i requisiti stabiliti dal cliente nella documentazione iniziale.

- **Dispiegamento**: In questa fase finale il software viene consegnato al committente completo della relativa documentazione di utilizzo e secondo le modalità di licenza stabilite fra il cliente e l'azienda produttrice. A questa, può seguire una fase di supporto e manutenzione da parte del fornitore del software.

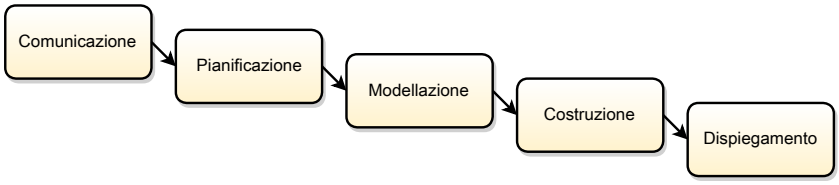


Figura 1: Modello "Waterfall"

Secondo Pressman il modello sequenziale enunciato da Royce si compone di queste cinque fasi appena descritte, eseguite semplicemente l'una subito dopo l'altra (Figura 1), con l'accortezza che una data attività venga iniziata solo dopo che la precedente è stata portata a termine. Questo perché, ad eccezione della fase di comunicazione, tutte le fasi richiedono in input ciò che è stato prodotto nella fase precedente.

Il modello waterfall si basa fortemente sul concetto di *Big Design Up Front* ovvero su una "grande progettazione a monte": se vengono rilevati difetti o imperfezioni nelle fasi iniziali di produzione del software, tali difetti saranno più facilmente eliminabili; sarà inoltre più economico in termini monetari e temporali correggere un difetto in fase di design piuttosto che in fase di implementazione, in quanto nelle ultime fasi si avrà ormai sprecato tempo e denaro nella realizzare una soluzione difettosa o poco performante. Su questo argomento infatti l'ingegnere del software Steve Mc Connell afferma "[...] rimediare a un difetto nei requisiti che rimane nascosto sino alla fase di implementazione o di manutenzione del sistema, verrà a costare da 50 a 200 volte tanto, se comparato a ciò che si sarebbe speso se il difetto fosse stato rilevato durante la raccolta dei requisiti [...]". Per questo motivo per poter utilizzare il modello *waterfall* è necessario che ogni fase sia correttamente completata in tutti i suoi aspetti prima di procedere a quella successiva.

Un altro aspetto caratterizzante il modello a cascata è il fatto che tale approccio considera di fondamentale importanza la documentazione generata dalle varie fasi[1], come affermato dallo stesso Royce. L'idea che sta alla base di questa convinzione risulta essere che nelle situazioni in cui vi siano una povera progettazione e documentazione, la perdita di un componente del team di sviluppo corrisponde alla perdita di conoscenza riguardante il lavoro da svolgere, in quanto gran parte del sapere risiede nei componenti del team. Questo può comportare gravi difficoltà a riprendere il progetto o a inserire nuovi membri. Disponendo invece di una documentazione completa ed esaustiva, risulta relativamente semplice l'aggiunta di nuovo personale o addirittura assegnare il progetto

1 Spesso infatti si sente parlare di metodologie di tipo *documentation-driven*, cioè "guidate dalla documentazione".

ad un nuovo team di sviluppo. È sufficiente che chi verrà assegnato ad un nuovo incarico, consulti i vari documenti per familiarizzare con il progetto.

Critiche al modello a cascata

Con il passare degli anni, gli stessi sostenitori del modello a cascata si resero presto conto di alcune importanti problematiche che nascevano dall'utilizzo di tale modello [17]. Nell'articolo in cui presentò tale modello, lo stesso Royce descrisse che pur essendo un approccio molto semplice, nella sua esperienza il modello a cascata non risultò adatto alla realizzazione di grandi sistemi software e che i costi di produzione utilizzando tale metodo eccedettero di gran lunga le stime inizialmente previste.

Alcune delle maggiori cause di fallimento del modello *waterfall* si possono riconoscere nei seguenti casi:

- Generalmente i progetti software reali non seguono un flusso sequenziale proposto dal modello. Nel momento in cui giungono cambiamenti ai requisiti iniziali mentre il progetto si trova già a fasi di sviluppo avanzate, gli sviluppatori si trovano in uno stato confusionale difficilmente gestibile.

- Spesso il cliente non riesce ad esprimere in modo chiaro e completo i requisiti che il prodotto da lui richiesto deve soddisfare. Molte funzionalità vengono richieste quando ormai la progettazione è stata superata ed il team di sviluppo è ormai in fase di codifica: tali modifiche sono difficilmente integrabili con il modello *waterfall*, il quale ha serie difficoltà nel gestire l'incertezza iniziale intrinseca di alcuni progetti.

- Il modello a cascata produce software funzionante solo nelle fasi finali, per cui il committente deve essere paziente e attendere la fine dello sviluppo per poter verificare il funzionamento di ciò che ha richiesto. Tuttavia egli si può rendere conto solo dopo mesi che ciò che riceve come prodotto finito non corrisponde a ciò che aveva in mente inizialmente, oppure che mancano caratteristiche e funzionalità importanti.

Al giorno d'oggi l'alto ritmo di sviluppo e il flusso costante di cambiamenti a specifiche e funzionalità, hanno fatto sì che il modello a cascata diventasse obsoleto e controproducente, e ciò ne ha causato il progressivo abbandono da parte dell'industria del software che si trovò sogget-

ta alle sue grandi limitazioni e la sua rigidità. Rimane comunque un importante riferimento teorico per comprendere il processo di naturale evoluzione di tale modello nei suoi derivati.

2.2 MODELLI A PROCESSO INCREMENTALE

Una prima evoluzione del modello a cascata si può identificare nei modelli a processo incrementale dove, al posto di considerare il prodotto finale come frutto di un'unica sequenza di attività, questi viene suddiviso in un insieme finito di incrementi più piccoli. Solitamente il primo incremento comprende le funzionalità "*core*" dell'applicazione da realizzare, le quali verranno successivamente integrate a quelle mancanti con i successivi rilasci di software.

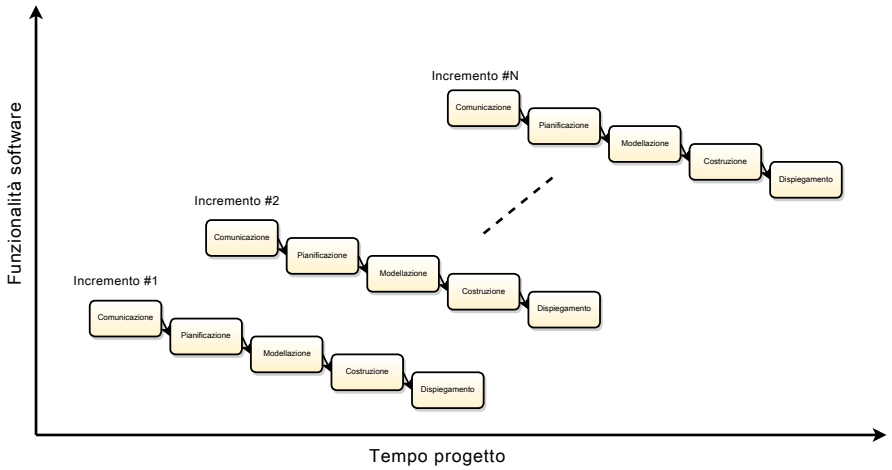


Figura 2: Modello incrementale

Osservando la Figura 2 si può vedere come ogni incremento rispecchi la sequenza lineare stabilita dal modello a cascata[2]. Al termine di ogni incremento è prodotta una versione del software funzionante, con funzionalità minimali che verranno via via migliorate e ampliate con gli incrementi successivi. Un ulteriore evoluzione rispetto all'originale modello *waterfall* risulta essere la parallelizzazione di attività: infatti nel momento in cui ci si appresta a rilasciare un dato incremento software, parte del team di sviluppo (per esempio chi analizza i requisiti) può dedicarsi alla progettazione dell'incremento successivo.

Uno dei vantaggi principali dei modelli incrementali risulta essere la possibilità di consegnare una versione minimale ma funzionante del soft-

2 Questo modello è a volte chiamato "modello *multi-waterfall*"

ware commissionato. Ciò può essere di grande aiuto nel caso in cui ci si trovi nell'impossibilità di terminare il prodotto entro la scadenza prestabilita, permettendo agli sviluppatori di fornire una versione basilare del software, per poi integrarla con gli incrementi successivi.

Risulta altrettanto importante il fatto che questo tipo di processi forniscono incrementi che possono ricevere feedback dal cliente. In questo modo eventuali dubbi, incomprensioni o requisiti non dichiarati possono essere rilevati e permettere al team di sviluppo di avviare azioni correttive, consentendo di evitare le situazioni in cui il cliente manifesta perplessità dopo mesi dall'inizio del progetto (situazioni tipiche del modello *waterfall*).

Pur portando alcuni miglioramenti, i metodi incrementali legati al modello a cascata possiedono alcuni svantaggi. Prima di poter suddividere un progetto in vari incrementi, è sempre necessario avere il quadro completo di quali requisiti l'applicazione dovrà soddisfare. Di conseguenza rimane necessaria la progettazione del software nella sua interezza e poi verificare quali caratteristiche sono separabili nei diversi incrementi. Pur suddividendo il progetto in vari componenti teoricamente indipendenti, può accadere che al momento di effettuare l'integrazione si verifichino incompatibilità causate da problematiche che sono sfuggite alla pianificazione iniziale.

Inoltre il cliente potrebbe approfittare di questa struttura ad incrementi per chiedere l'aggiunta di nuovi requisiti a cui sbadatamente non aveva pensato prima, andando a modificare la pianificazione iniziale realizzata prima di suddividere il software in incrementi. In questa situazione le varie release vanno ripianificate e possono essere necessarie risorse economiche di gran lunga superiori a quelle inizialmente stimate.

2.3 MODELLI A PROCESSO EVOLUTIVO

Il primo grande cambiamento nella mentalità di chi sviluppa sistemi informatici si può vedere con l'introduzione dei modelli a processo evolutivo come nuovi metodi di produzione software. Con gli anni i problemi relativi all'adozione delle prime metodologie di produzione software, portarono gli sviluppatori a rendersi conto che queste si basavano su ipotesi che molto spesso risultavano non soddisfatte.

Se è vero che il modello a cascata risulta la soluzione ideale nel caso di progetti di cui si conoscono a priori tutti i requisiti (perché il progetto è semplice o perché rispecchia soluzioni già sviluppate per altri clienti in passato), è anche vero che nella maggior parte dei grandi pro-

getti software tale requisito non è soddisfatto: possono essere chiare un insieme di funzionalità di base, ma i dettagli implementativi, del sistema o di alcuni componenti possono non essere ancora del tutto definiti; inoltre possono sussistere particolari scelte tecnologiche e pressioni di mercato a cui il cliente deve sottostare. In questi casi è importante che il processo utilizzato per la realizzazione di tali applicazioni lasci spazio all'evoluzione che il software può subire lungo un dato periodo di tempo.

Il caratteristico approccio incrementale e iterativo dei modelli evolutivi permette di sviluppare rapidamente delle applicazioni software che possono essere mostrate al cliente e successivamente farle evolvere in versioni sempre più complete del prodotto finale

2.3.1 Prototyping

Nei casi in cui si possiedano un set di requisiti base ma non si conoscano perfettamente altri aspetti relativi al software da realizzare, il paradigma del *prototyping* può essere il più indicato. Con questo modello, il team di sviluppo crea rapidamente un prototipo del sistema finale e lo presenta al committente. A questo punto il cliente può capire quali siano i requisiti mancanti o se sia necessario raffinare quelli già descritti; contemporaneamente gli sviluppatori possono verificare se la strada che hanno intrapreso per implementare la soluzione software è quella corretta.

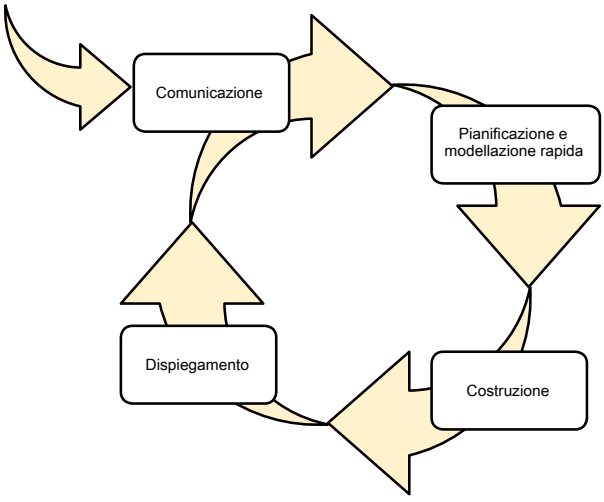


Figura 3: Prototyping

La Figura 3 rappresenta una generica esecuzione di questo paradigma. Il punto di partenza rimane in ogni caso la fase di comunicazione con

il cliente, durante la quale vengono dichiarate le prime funzionalità che si desidera includere nell'applicazione software. A questa seguono delle brevi e rapide fasi di pianificazione e modellazione che portano alla costruzione del primo prototipo che viene quindi consegnato al cliente nella fase di dispiegamento. Solitamente in questa prima iterazione, il team si focalizza sugli aspetti che interesseranno principalmente l'utente finale del software (per esempio l'interfaccia fra l'utilizzatore e l'applicazione, la visualizzazione dei dati, ecc.). Il committente può a questo punto valutare questa prima versione del prodotto software fornendo un prezioso feedback per gli sviluppatori. Entrambe le parti potranno apprendere nuovi dettagli per la successiva iterazione, mentre il modello verrà poi ri-eseguito fino a quando le varie necessità del committente saranno soddisfatte.

Esistono varie tipologie di *prototyping*, le quali sono raggruppabili in due grandi famiglie:

- *Throw-away prototyping*
- *Evolutionary prototyping*

Come suggerisce la parola, il prototipo di tipo "*Throw-away*" prevede che, al termine della prima iterazione, ciò che è stato realizzato venga gettato via. Applicare questo tipo di prototipazione significa essere consapevoli che il primo prototipo avrà gravi carenze e permetterà solamente di acquisire una maggiore conoscenza sui requisiti software che l'applicazione commissionata dal cliente dovrà avere. Con lo scopo di ottenere rapidamente un feedback dal committente, spesso viene realizzata solamente una basilare interfaccia grafica, creata in breve tempo con appositi *GUI Builder*[3]. Tale interfaccia permetterà al cliente di chiarire i suoi dubbi e raffinare i requisiti che dovrà avere il prodotto finale.

In altri casi viene effettivamente implementata una prima applicazione che svolge alcune delle funzionalità richieste, tuttavia data la rapidità con cui viene realizzata, possono venir tralasciati vari aspetti (es. complessità degli algoritmi, tempi di esecuzione) per cui si ricomincia riscrivendo l'applicativo in maniera più intelligente, realizzandone una versione riprogettata e tenendo conto della conoscenza acquisita dalla realizzazione del primo prototipo.

D'altra parte un prototipo "evolutivo" viene realizzato con lo scopo di costruire fin da subito una soluzione robusta che verrà via via raffinata man mano che si riceveranno i feedback dal committente. Le

3 I *Graphic User Interface Builder* sono dei tool informatici che permettono di costruire facilmente un "fantoccio" di applicazione con la sola interfaccia utente e priva di funzionalità.

successive iterazioni andranno ad aggiungere le funzionalità concordate con il cliente, siano quelle stabilite all'inizio o altre emerse durante la valutazione dei primi prototipi.

La natura evolutiva del paradigma di prototyping pone tuttavia alcuni problemi nella pianificazione. Le consuete tecniche di stima del progetto sono basate su un set sequenziale di attività: il numero incerto di iterazioni che sono necessarie per giungere al prodotto finito potrebbe causare difficoltà nello stabilire la corretta organizzazione delle attività[4]. Un altro problema che può nascere con questo approccio riguarda l'eccessivo tempo di sviluppo del prototipo da parte degli sviluppatori. Il punto di forza della creazione di un prototipo è proprio la velocità con cui viene creato un modello che simuli il prodotto finale, con lo scopo definire in modo il più preciso possibile i requisiti con il cliente. Se il team di sviluppo perde di vista questa peculiarità cercando di sviluppare un tipo di prototipo troppo complesso, perché mira a realizzare sin da subito qualcosa di definitivo, potrebbe impiegare una considerevole quantità di tempo per funzionalità che il cliente potrebbe ritenere inutili. Nel peggiore dei casi il prototipo potrebbe venir gettato via e i pochi requisiti raccolti potrebbero non essere sufficienti a compensare il tempo impiegato per la sua creazione.

Può infine capitare che, soprattutto in iterazioni avanzate, il cliente si trovi di fronte ad un prodotto che a suo parere risulta completo, ma sia carente di dettagli tecnici e implementativi conosciuti solo dagli sviluppatori. In questi casi il committente potrebbe spazientirsi non riuscendo a spiegarsi del perché siano necessarie ulteriori iterazioni.

2.3.2 Modello a spirale

Il modello a spirale viene proposto per la prima volta nel 1986 dall'ingegnere del software Barry Boehm nel suo articolo "*A Spiral Model of Software Development and Enhancement*" [8]. Boehm descrive come questo modello segua un approccio *risk-driven* piuttosto che uno guidato dalle specifiche o dal prototipo, sfruttando comunque i punti di forza dei precedenti modelli di sviluppo software e risolvendo alcune loro problematiche.

Nella Figura 4 si può vedere l'evoluzione di un progetto che utilizza il modello a spirale. In generale questo modello applica iterativamente le fasi del modello sequenziale a cascata, fornendo ad ogni ciclo una soluzione sempre più perfezionata e vicina alle esigenze del cliente. Man

4 Nogueira [23]

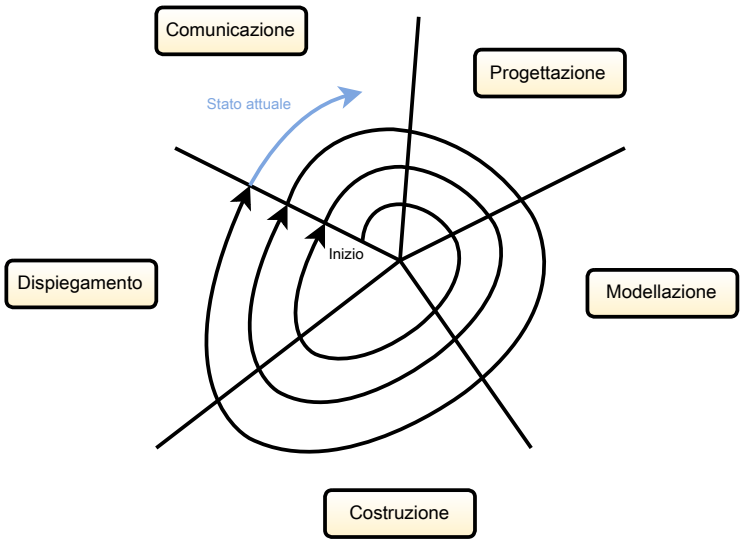


Figura 4: Modello a spirale

mano che il progetto avanza, si può vedere come il raggio della spirale aumenti costantemente, rappresentando il crescente costo cumulato che il prodotto assume. Il software, che nelle prime iterazioni può essere un modello cartaceo o un prototipo, evolve continuamente sfruttando il feedback che viene fornito alla fine di ogni iterazione, quando termina la fase di dispiegamento. Le principali *milestone*, ovvero una combinazione di prodotti funzionanti e di condizioni raggiunte lungo il percorso a spirale delle fasi precedenti, vengono annotate e documentate.

Costantemente lungo tutto il processo vi è una attenta gestione del rischio, che Boehm include intrinsecamente all'interno del modello. L'identificazione e la gestione dei maggiori tipi di rischi, sia di natura tecnica sia manageriale, permettono di mantenere il processo sotto controllo. Via via che il progetto procede, il team di sviluppo raggiunge un dettaglio tecnico sempre maggiore e migliora la visione completa del sistema, permettendo contemporaneamente la diminuzione dei rischi dovuti all'incertezza iniziale. Solitamente il modello a spirale continua la sua esecuzione ben oltre la consegna del prodotto finito al cliente e il progetto termina solamente quando il software viene definitivamente abbandonato per obsolescenza e viene sostituito con un'altra applicazione più performante.

Il modello a spirale risulta essere un approccio realistico allo sviluppo di sistemi software a larga scala, in quanto prevede sin dall'inizio che il software evolva a causa di molteplici ragioni, siano nuovi requisiti del cliente o eventuali nuove tecnologie. Combinando le fasi del modello a cascata con l'approccio iterativo del prototyping, il modello a spirale

mantiene l'approccio sistematico che contraddistingue il primo e l'aspetto evolutivo che caratterizza il secondo e, unitamente alla gestione dei rischi, riflette il processo reale di evoluzione del software.

Il modello a spirale conserva ad ogni modo alcune problematiche. Spesso risulta particolarmente difficile convincere il committente, in particolare nelle fasi di redazione del contratto, della controllabilità di questo approccio evolutivo. Inoltre questo metodo richiede una grande competenza nell'analisi dei rischi, in quanto un fattore ignorato o tralasciato può portare risultati scadenti o addirittura disastrosi.

Abbiamo visto in questo capitolo come gli sviluppatori software iniziarono a ideare processi che potessero guidarli durante la realizzazione di progetti, cercando di limitare il caos dovuto alle varie attività di sviluppo. Col primo modello sequenziale a cascata ci si rese conto di come un approccio di questo genere fosse raramente applicabile, data la natura mutevole del software. La naturale evoluzione di questo modello fu il modello incrementale, che permise di suddividere una soluzione in più piccoli incrementi che, una volta realizzati e integrati, costituiscono il software finale. Ulteriori miglioramenti nelle attività di sviluppo software giunsero con i modelli evolutivi che seguono un approccio di tipo iterativo e incrementale.

Nel prossimo capitolo verranno presentate le metodologie di *Agile Software Development*, le quali nacquero per rispondere alla sempre crescente mutevolezza del mercato del software e per superare le limitazioni dei modelli descritti in questo capitolo. Alcune delle idee che stanno alla base delle metodologie agili sono ispirati a principi già usati nelle prime metodologie di sviluppo software, primi fra tutti i concetti di incremento e iterazione.

3 METODOLOGIE DI AGILE SOFTWARE DEVELOPMENT

In questo capitolo verranno presentate alcune delle metodologie di *Agile Software Development*, nate alla fine degli anni '90 per superare le limitazioni delle metodologie più tradizionali derivate dal modello a cascata.

Con un approccio allo sviluppo software più incline ad "abbracciare il cambiamento" (citando Kent Beck ideatore di *Extreme Programming*) piuttosto che a controllarlo, è possibile rispondere rapidamente alle opportunità di mercato e/o alle richieste dei committenti. In molti casi il *time-to-market* rimane il requisito fondamentale per il cliente, in quanto la perdita di una finestra di mercato, può rendere il software inutile [28]. Può infatti succedere che un prodotto software soggetto a una lunga e pesante progettazione risulti obsoleto al momento del rilascio proprio a causa della rapida mutevolezza del mercato del software, sia per la pronta risposta di eventuali concorrenti, sia per la nascita di nuove tecnologie o il presentarsi di nuove opportunità. Per questo motivo questi metodi si basano su processi iterativi e incrementali (eventualmente con l'aggiunta del paradigma di *prototyping*) che possano fornire piccoli rilasci di software in brevi cicli di sviluppo. Questo permette di ottenere un rapido feedback da parte del committente sin dalle prime settimane di lavoro.

3.1 NASCITA DELLE METODOLOGIE AGILI

Le idee che stanno alla base delle metodologie agili non sono nuove, anzi risalgono alla nascita del modello a cascata. Già Royce alla fine del suo articolo del '70 sul modello a cascata, affermò che per poter realizzare un prodotto software adeguato, era necessario adottare una serie di pratiche che divennero in questi ultimi anni le fondamenta per le metodologie agili.

Nel febbraio del 2001 diciassette esperti dello sviluppo software alla

ricerca di alternative alle metodologie di sviluppo tradizionale, si incontrano in una località sciistica nelle Wasatch Mountains nello Utah, per condividere le loro opinioni sullo sviluppo software. Tutti i partecipanti erano concordi nel ritenere che le metodologie derivate dal modello a cascata non rappresentassero la soluzione più efficace in ambienti moderni, dove è importante possedere una certa "agilità" nell'affrontare la costruzione di sistemi software: nella maggior parte delle situazioni i clienti non hanno chiaro ciò che desiderano, e per questo vanno seguiti con particolare attenzione per cercare di far loro capire i requisiti intrinseci della soluzione desiderata e costruire con loro un forte legame di collaborazione. Uno dei punti principali, condiviso da tutti i partecipanti, risulta essere che tali requisiti non rimarranno immutati e subiranno indubbiamente modifiche lungo il processo di realizzazione. E' necessario quindi prevedere questo cambiamento invece di controllarlo, e sfruttarlo per permettere al cliente di ottenere una soluzione più compatibile alle nuove esigenze emerse, siano esse di tipo tecnologico, manageriale, progettuale o temporale.

Essere agili significa inoltre alleggerire i processi: se è vero che una documentazione formale e approfondita del sistema può descriverlo in ogni aspetto e permette ad un nuovo membro la completa familiarizzazione, è altrettanto vero che proprio la grande mole di tale documentazione può essere un ostacolo in situazioni turbolente, dove anche pochi giorni di ritardo fanno la differenza. Un componente del team che necessità di essere istruito o aggiornato sul progetto in corso, può necessitare di troppo tempo per essere informato sull'intero progetto.

Da queste considerazioni, i diciassette esperti concordarono i seguenti fatti:

- **Individui e le interazioni** più di processi e strumenti.
- **Software funzionante** più che documentazione esaustiva.
- **Collaborazione col committente** più che negoziazione del contratto.
- **Rispondere al cambiamento** più che seguire un piano prestabilito.

Ovvero, riconoscendo il valore delle voci a destra, considerarono più importanti le voci a sinistra. Nacque così il "*Manifesto for Agile Software Development*" che, oltre ai fatti appena descritti, racchiude una serie di dodici principi che stanno alla base del movimento agile:

I dodici principi

1. "La nostra massima priorità è soddisfare il cliente per mezzo di tempestivi e continui rilasci di software di valore."

2. "Siano benvenuti i cambiamenti nelle specifiche, anche a stadi avanzati dello sviluppo. I processi agili sfruttano il cambiamento a favore del vantaggio competitivo del cliente."
3. "Rilascia software funzionante frequentemente, da un paio di settimane a un paio di mesi, con preferenza per i periodi brevi."
4. "Manager e sviluppatori devono lavorare insieme quotidianamente durante tutto il progetto."
5. "Basa i progetti su individui motivati. Dai loro l'ambiente e il supporto di cui necessitano e confida nella loro capacità di portare il lavoro a termine."
6. "Il metodo più efficiente ed efficace di trasmettere informazione verso e all'interno di un team di sviluppo è la conversazione faccia a faccia."
7. "Il software funzionante è la misura primaria di progresso."
8. "I processi agili promuovono uno sviluppo sostenibile. Gli sponsor, gli sviluppatori e gli utenti dovrebbero essere in grado di mantenere un ritmo costante indefinitamente."
9. "L'attenzione continua per l'eccellenza tecnica e la buona progettazione esaltano l'agilità."
10. "La semplicità - l'arte di massimizzare l'ammontare di lavoro non svolto - è essenziale."
11. "Le migliori architetture, specifiche e design emergono da team auto-organizzati."
12. "A intervalli regolari il team riflette su come diventare più efficace, dopodiché mette a punto e aggiusta il suo comportamento di conseguenza."

I firmatari del "Manifesto Agile" si auto-definirono in quella occasione la *Agile Alliance* il cui scopo è diffondere le idee per nuovi e più realistici approcci alla realizzazione di prodotti software. In quegli anni tutti i membri dell'*Agile Alliance* erano coinvolti nella ricerca e nella sperimentazione sul campo di lavoro di nuove tecniche per lo sviluppo software e, rendendosi reciprocamente conto che altri esploravano la stessa strada, si convinsero a continuare la loro attività con maggior dedizione e negli anni successivi pubblicarono diversi libri su queste nuove metodologie, descrivendo le loro esperienze nell'applicare tali metodi e diffondendoli a livello mondiale.

3.2 ESEMPI DI METODOLOGIE AGILI

Nelle sezioni che seguiranno verranno descritte alcune fra le principali metodologie di *Agile Software Development*, assieme alle caratteristiche che le contraddistinguono. I proponenti delle varie metodologie seguono in genere i principi descritti dal "Manifesto Agile", enfatizzandone alcuni aspetti e tralasciandone invece degli altri.

3.2.1 Scrum

Le prime idee di *Scrum* fecero la propria comparsa nel 1986 in un articolo di Hirotaka Takeuchi e Ikujiro Nonaka i quali presentarono un nuovo approccio allo sviluppo di nuovi prodotti che prometteva rapidità e flessibilità [37]. I due autori giapponesi la definiscono come una "strategia flessibile e olistica allo sviluppo di un prodotto, dove il team di sviluppo lavora come un'unica entità per raggiungere un obiettivo comune, in opposizione all'approccio sequenziale delle metodologie tradizionali".

Il termine *Scrum* descrive la fase di mischia nel gioco del rugby, usata per riprendere il gioco dopo un'infrazione. L'analogia con la metodologia di *Agile Software Development* rappresenta il fatto, come spiega Takeuchi, che il team di sviluppo deve lavorare come una squadra che "cerca di raggiungere l'obiettivo come unità, passando la palla avanti e indietro".

Nel 1995 Jeff Sutherland e Ken Schwaber (che sei anni più tardi furono due dei diciassette sottoscriventi del "Manifesto Agile di Sviluppo Software"), presentarono per la prima volta un articolo che descriveva la metodologia *Scrum* [36]. Negli anni successivi Sutherland e Schwaber continuarono lo sviluppo di *Scrum* integrandolo con le loro esperienze lavorative e alcune *best-practise* dell'industria. Il risultato è la metodologia *Scrum* che conosciamo al giorno d'oggi.

Nel 2001 Ken Schwaber scrisse assieme a Mike Beedle (un altro firmatario del "Manifesto Agile") il libro "*Agile Software Development with Scrum*" [31] con lo scopo di descrivere in maniera completa questa metodologia.

Scrum suddivide l'intero processo di sviluppo in tre fasi principale: la fase di *Pre-game*, la fase di *Development* e infine quella di *Post-game*.

La prima fase di *Pre-game* comprende il dialogo con il cliente per comprendere le sue esigenze. Queste necessità vanno a formare il *Product Backlog*, un artefatto di *Scrum* che raccoglie tutti i requisiti che sono co-

nosciuti al momento. I requisiti possono essere enunciati sia da parte del cliente, sia dal team di sviluppatori. Ad ogni requisito viene data una priorità e vengono stabilite le stime di sviluppo. Il *Product Backlog* viene aggiornato costantemente, ogni volta che vengono riscontrati dettagli aggiuntivi oppure le priorità di sviluppo dei componenti della lista vengono modificate. In questa fase vengono definiti il team di progetto e tutte le risorse che saranno necessarie durante lo sviluppo. Vengono inoltre stabiliti standard, convenzioni, architettura e tecnologia su cui si baserà il software realizzato.

Nella fase centrale di *Development*, conosciuta anche come *Game*, il sistema viene costruito attraverso la successione di *Sprint*. Gli *Sprint* sono cicli iterativi durante ai quali le funzionalità vengono implementate valutando e aggiustando costantemente le diverse variabili di progetto (tempi di sviluppo, risorse, requisiti, qualità, ecc.) con lo scopo di raggiungere la flessibilità necessaria a rispondere ai cambiamenti imprevisti. All'inizio di ogni ciclo il team si riunisce e viene creato uno *Sprint Backlog* il quale non è altro che un sottoinsieme di funzionalità estratte dal *Product Backlog* che potranno essere realizzate nello *Sprint*. Gli *Sprint* seguono un approccio di tipo *timeboxed*, ovvero la loro durata è prestabilita (da una a quattro settimane); per questo motivo gli elementi che comporranno lo *Sprint Backlog* verranno scelti in base alla loro priorità e alle stime di sviluppo stabilite in precedenza.

Ogni giorno è prevista una brevissima riunione, chiamata *Daily Scrum meeting*, in cui gli sviluppatori si aggiornano sullo stato del progetto.

Durante i vari *Sprint*, lo sviluppo procede in maniera sequenziale secondo le consuete tecniche di sviluppo software (partendo dai requisiti dello *Sprint Backlog* si effettua l'analisi seguita da progettazione ed implementazione). Può succedere che durante queste fasi emergano nuovi requisiti espressi dal cliente, oppure causati da particolari scelte di progettazione. In questo caso al *Product Backlog* vengono aggiunti i nuovi elementi da sviluppare e di conseguenza vengono aggiornate le priorità e le stime di sviluppo. Ad ogni modo i nuovi elementi inseriti potranno venire considerati solo nel prossimo *Sprint*. Al termine di ogni *Sprint*, si ottiene un incremento funzionante del prodotto finale. Può accadere che in un'iterazione lo *Sprint Backlog* rimanga inconcluso: in questo caso gli elementi ancora da realizzare verranno inseriti nello *Sprint* successivo.

La fase di *Development* termina quando nel *Product Backlog* non vi sono più elementi da implementare.

L'ultima fase di *Post-game* determina la fine del processo. Questa fase viene raggiunta nel momento in cui vengono soddisfatte le condizione per dichiarare terminato il prodotto, ovvero che i requisiti definiti sono

stati implementati. In questa situazione non sono riscontrati ulteriori problemi da correggere e non vi sono altre funzionalità da realizzare. Tutti gli incrementi sono integrati a formare il sistema software che viene collaudato interamente. Superato il test, l'applicazione viene rilasciata al cliente assieme alla relativa documentazione.

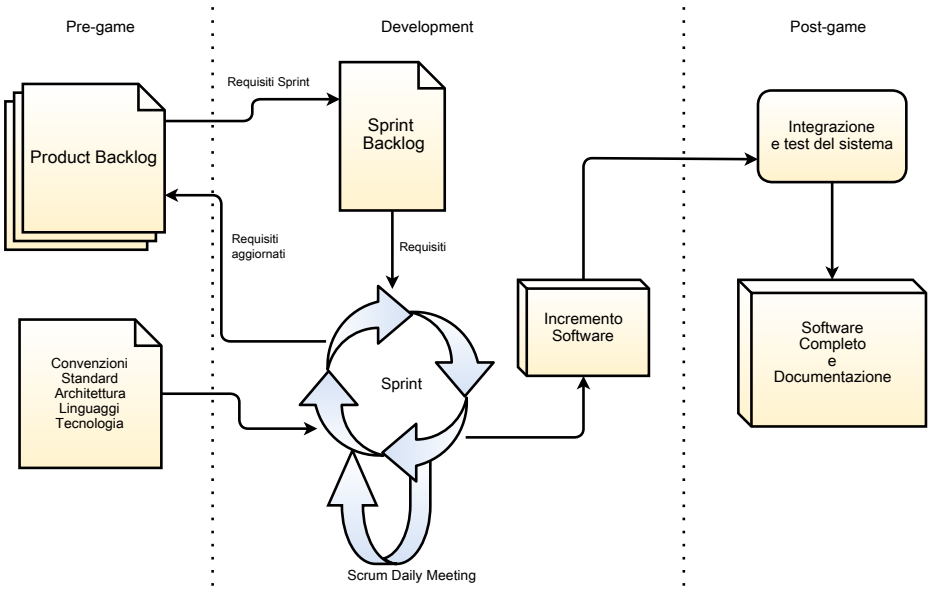


Figura 5: Ciclo di vita di Scrum

Schwaber e Beedle introducono un insieme di ruoli che verranno assegnati ai diversi partecipanti allo sviluppo del prodotto, con lo scopo di stabilire le diverse responsabilità all'interno del progetto.

- *Scrum Master* È un nuovo ruolo che *Scrum* introduce nel processo. Lo *Scrum Master* è responsabile della corretta applicazione delle regole e dei processi e che il progetto proceda come pianificato. Si occupa inoltre di rimuovere qualsiasi tipo di impedimento che ostacoli la buona riuscita del progetto, permettendo i componenti del team di rimanere il più produttivi possibile.

- *Product Owner* È il responsabile del progetto e rappresenta la "voce del cliente". Deve fare in modo che la lista *Product Backlog* sia ben visibile. Viene scelto dallo *Scrum Master*, e dal committente. Effettua le decisioni strategiche relativamente al *Product Backlog*, partecipa alle stime di impegno richiesto per lo sviluppo e traduce le funzionalità da sviluppare in elementi del *Product Backlog* (tipicamente scrivendo delle *user-story*).

- *Scrum Team* È il team di sviluppo che ha l'autorità necessaria per decidere sul da farsi, su come organizzarsi per raggiungere i ruoli

stabiliti da ogni Sprint. Deve anche comunicare allo *Scrum Master* eventuali impedimenti da rimuovere.

- *User* È l'utilizzatore finale del prodotto che viene realizzato.

La metodologia *Scrum* non suggerisce alcun tipo di metodi di sviluppo software specifici, tuttavia richiede un certo livello di conoscenza di pratiche e strumenti di management nelle varie fasi in cui *Scrum* si articola, in modo da controllare al meglio il caos causato dal cambiamento dei requisiti. Per fare ciò, utilizza un insieme di pratiche e artefatti che vengono descritti qui sotto.

- *Product Backlog* Stabilisce tutto ciò che deve essere presente nel prodotto finale, definendo il lavoro che deve essere fatto durante tutta la durata del progetto. Viene redatta una lista i cui elementi sono i requisiti (tecnici o di business) del sistema che deve essere creato o modificato. Gli elementi di tale lista possono includere funzionalità, bug da correggere, difetti, modifiche da apportare, aggiornamenti tecnologici e via dicendo e a ognuno di essi viene assegnata una priorità di sviluppo. Alla redazione del *Product Backlog* possono partecipare praticamente tutti i ruoli descritti in precedenza.

  Durante la messa in pratica di *Scrum*, il *Product Backlog* può essere liberamente modificato, aggiungendo e togliendo elementi della lista e cambiando la priorità delle entry. Il *Product Owner* si prende la responsabilità di seguire il *Product Backlog*.

- *Effort estimation* È un processo iterativo in cui le stime di sviluppo degli elementi del *Product Backlog* vengono approfondite non appena nuove informazioni sono disponibili. *Product Owner* e lo *Scrum Team* si occupano di questo compito.

- *Sprint* Lo *Scrum Team* si organizza autonomamente per produrre un nuovo incremento software funzionante in un ciclo di lavoro che dura solitamente da una a un massimo di quattro settimane.

- *Sprint planning meeting* Lo *Sprint Planning Meeting* è una riunione del team di sviluppo che si compone di due fasi. Durante la prima fase, tutte le persone coinvolte nel progetto partecipano alla riunione, la quale viene presieduta dallo Scrum Master. Vengono decisi gli obiettivi e le funzionalità del prossimo *Sprint*.

  Nella seconda fase partecipano lo *Scrum Master* assieme allo *Scrum Team* e verranno stabiliti i dettagli tecnici su come verrà implementato l'incremento durante il prossimo *Sprint*.

- *Sprint Backlog* Lo *Sprint Backlog* è il punto di partenza per ogni *Sprint*. È una lista di elementi, scelte dal *Product Backlog*, che verranno implementate nel prossimo *Sprint*. Gli elementi che comprendono lo *Sprint Backlog* vengono accuratamente scelti dallo *Scrum Master*, lo *Scrum Team* e dal *Product Owner* sulla base della loro priorità, delle stime di sviluppo e degli obiettivi stabiliti per il prossimo *Sprint*. Nel momento in cui tutti gli elementi dello *Sprint Backlog* sono terminati, l'iterazione è conclusa e si può passare alla successiva.

- *Daily Scrum Meeting* È una riunione quotidiana utilizzata per tenere traccia dei progressi dello *Scrum Team*. Serve inoltre per organizzarsi sul lavoro da svolgere. In genere si discute su ciò che ogni membro del team ha svolto dall'ultima riunione, su ciò che programma di fare prima della prossima. Vengono inoltre identificati eventuali problemi che possono impedire allo *Scrum Team* di eseguire il suo lavoro nel migliore dei modi. Lo *Scrum Master* ha il compito di rimuovere e neutralizzare tali impedimenti.

  La durata di tali riunioni è solitamente molto breve e, il più delle volte, chi vi partecipa lo fa in piedi, evitando di prendere posto ad un tavolo. Questo obbliga i partecipanti a concentrare la discussione sulle questioni più importanti e sul lavoro da svolgere.

- *Sprint Review Meeting* L'ultimo giorno dello *Sprint*, lo *Scrum Team* e lo *Scrum Master* presentano il risultato dello *Sprint*, ovvero l'incremento software, al *Product Owner* e al cliente (ed eventualmente agli utilizzatori finali, definiti come *User*) in una riunione informale. Viene valutato l'incremento e i partecipanti decidono sulle prossime attività da svolgere. Tali riunioni permettono solitamente di identificare nuove funzionalità o requisiti da inserire nel *Product Backlog* e, in casi estremi, di cambiare drasticamente il modo in cui il sistema viene costruito.

- *Burndown Chart* Viene utilizzato per valutare l'andamento del lavoro svolto e da svolgere, e per migliorare le stime future dell'impegno necessario per la produzione di nuovi incrementi. Un esempio di *Burndown chart* si può vedere in Figura 6. La linea blu rappresenta, per ogni giorno, la stima di lavoro che rimane da completare, mentre quella rossa rappresenta l'effettivo lavoro mancante.

Gli autori di *Scrum* identificano due principali tipologie di progetto alle quali *Scrum* può essere applicato: i progetti esistenti e i nuovi progetti. Nel caso di progetti esistenti può accadere che siano già definiti

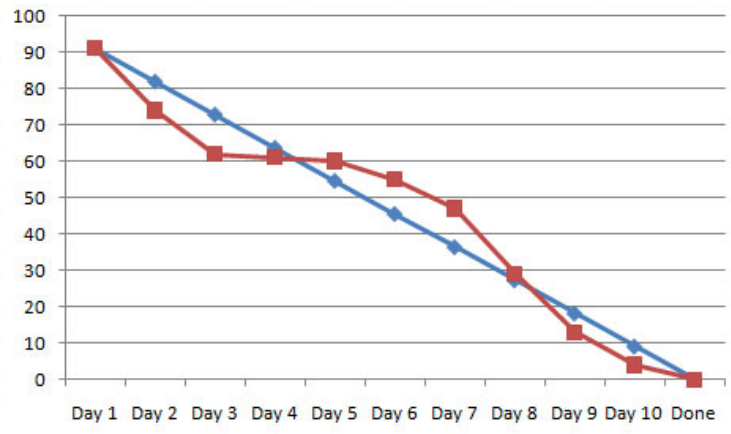


Figura 6: Esempio di Burndown chart

l'ambiente di sviluppo e la tecnologia in uso, ma il team di sviluppo non riesce a gestire i problemi dovuti al continuo cambiamento dei requisiti. In questo caso è possibile procedere, sotto la guida di uno *Scrum Master*, con la realizzazione di una qualsiasi parte di funzionalità richiesta dal committente eseguendo uno *Sprint*. In questo modo il team di sviluppo riprenderà fiducia in se stesso mentre il cliente potrà riporre fiducia nel team. Al termine di questo primo *Sprint*, *Scrum Master* e committente partecipano ad un *Sprint Review* e decidono come procedere. In caso in cui si decida di continuare con il progetto, il prossimo passo sarà uno *Sprint Planning Meeting* che stabilirà gli obiettivi e i requisiti del prossimo *Sprint*.

Nel caso invece in cui si decida di applicare *Scrum* a un nuovo progetto, Schwaber e Beedle suggeriscono di soffermarsi vari giorni alla redazione di un completo *Product Backlog*. A questo punto l'obiettivo del primo *Sprint* sarà la dimostrazione di una funzionalità chiave per il cliente. Per fare ciò, è necessario che il *Product Backlog* includa tutte le attività necessarie per raggiungere l'obiettivo, e queste prevedono l'adozione (o il tentativo di adozione) di *Scrum*. Di conseguenza il *Product Backlog*, oltre ai requisiti del sistema richiesto dal cliente, conterrà quelle attività atte a stabilire i ruoli di *Scrum* e l'adozione delle sue pratiche (questo aspetto sembra paradossale: usare *Scrum* per far adottare *Scrum* all'interno del team di sviluppo).

Schwaber consiglia che il team di sviluppo sia composto da cinque a nove persone e, nel caso in cui vi siano più componenti, suggerisce di formare team multipli. Ad ogni modo esistono casi in cui *Scrum* è stato applicato a team di sviluppo di centinaia di persone. Il segreto che ha permesso di scalare questa metodologia è applicare uno *Scrum of Scrums Meeting* [1]. Questa tecnica prevede che siano presenti team

multipli composti da massimo dieci componenti. Ogni Scrum Team partecipa al relativo *Daily Scrum Meeting* e una volta terminato sceglie un componente che parteciperà allo *Scrum of Scrums Meeting*. Data la breve durata di queste riunioni (entrambe durano massimo quindici minuti), nel giro di mezz'ora sarà in questo modo possibile ottenere informazioni sullo stato di sviluppo di un progetto di grandi dimensioni.

3.2.2 Extreme Programming

Extreme Programming (di seguito indicata con *XP*) è una metodologia agile ideata dall'ingegnere del software americano Kent Beck, che riunisce un insieme di *best practise* ben avviate e dettate dal buon senso. Egli ne descrive funzionamento e caratteristiche nel suo libro "*Extreme Programming explained: embrace change*" [4] (al quale segue una seconda edizione del 2004 [5]. Le idee che condussero Beck a ideare questa metodologia trovano le loro radici alla fine degli anni '80 e comprendono, fra gli altri, i concetti di sviluppo rapido descritti da Takeuchi e Nonaka [37] (già alla base di *Scrum*), i principi di sviluppo evolutivo descritti nel capitolo precedente e il modello a spirale di Boehm [8].

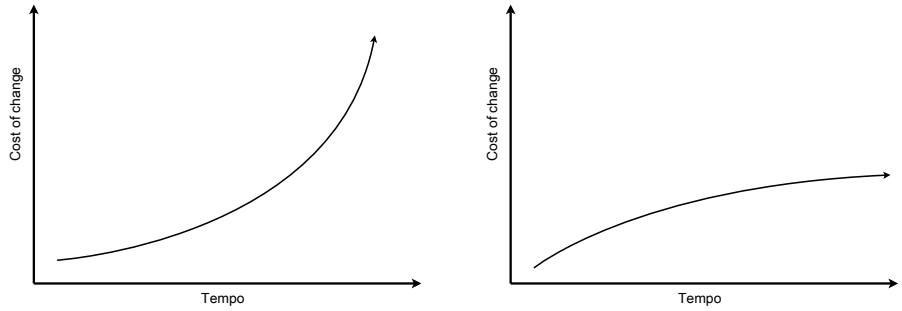
La parola *extreme* sta ad indicare il fatto che queste pratiche vengono applicate in maniera estrema durante tutto il processo di sviluppo software. Beck afferma che se revisionare il codice va bene, allora questo verrà fatto costantemente (*pair programming*). Se i test d'integrazione sono importanti, allora il team effettuerà questi test continuamente (*continuous integration*).

Ai tempi in cui Beck organizzava le sue idee su *Extreme Programming*, egli stesso ricorda di aver pensato all'insieme di pratiche alla base di questa metodologia come a delle manopole su un quadro di controllo, e di aver ipotizzato di impostarle tutte al massimo. Ciò che ne risultò fu un framework di attività che permise di ottenere un processo stabile e flessibile.

Cost of change

Una delle premesse fondamentali, che Beck identifica come "LA" premessa, riguarda il *cost of change*, il "costo del cambiamento". Abbiamo già visto nel capitolo precedente come le metodologie più tradizionali assumessero la necessità di un'accurata pianificazione iniziale, che permettesse l'identificazione di tutti i requisiti e di eventuali difetti nelle specifiche. L'ingegnere del software Steve Mc Connell afferma che il costo per correggere un difetto nell'applicazione o apportare una modifica, aumenta esponenzialmente man mano che ci si avvicina alle fasi finali dello sviluppo del software. Secondo Beck, con l'utilizzo di particolari

linguaggi di programmazione e l'adozione di pratiche adatte, queste assunzioni al giorno d'oggi non sono più valide ed è possibile fare in modo che la curva di costo assuma una forma totalmente differente, come si può osservare in Figura 7. Si può notare come la concezione tradizionale del costo del cambiamento assuma una forma esponenziale, mentre quella ipotizzata da Beck sia molto più appiattita col passare del tempo. Anche Roger Pressman afferma che quando il rilascio incrementale viene unito a integrazione continua del codice e ad altre pratiche agili, il costo del cambiamento è attenuato [28].



(a) Tradizionale **(b)** Secondo Beck

Figura 7: *Cost of change*

XP suddivide il processo di produzione software in sei fasi principali: *exploration*, *planning*, *iteration to release*, *productization*, *maintenance*, e *death*.

Nella prima fase di ***exploration***, il committente stabilisce i requisiti che il sistema software deve soddisfare scrivendo le cosiddette *user story*. Le *user story* sono una descrizione ad alto livello di ciò che il sistema deve fare e solitamente seguono un template del tipo seguente: **come <ruolo> voglio <obiettivo/desiderio>** (es. come utilizzatore voglio visualizzare tutti i dati dei prodotti; come amministratore voglio creare nuovi utenti); le *user story* servono inoltre agli sviluppatori per stimare le tempistiche di implementazione del sistema. Sempre in questa fase gli sviluppatori prendono familiarità con le caratteristiche del progetto ed esplorano le possibilità tecnologiche e le architetture software disponibili. Viene inoltre creato un prototipo del sistema. Tale fase può durare da qualche settimana a qualche mese, a seconda di quanta esperienza possiedono gli sviluppatori relativamente alla tecnologia che verrà adottata.

Terminata la fase di *exploration*, si passa alla fase di ***planning***. In questo stadio vengono definite le priorità da dare alle varie *user story* e vengono stabilite quali funzionalità comprenderà la prima distribuzione del sistema. Gli stadi finali della fase di *planning* prevedono la definizione dei

tempi necessari alla produzione della prima release, solitamente della durata di un paio di mesi.

La successiva fase di ***iterations to release*** rappresenta la parte iterativa di *XP*. La scaletta stabilita durante la fase di *planning* viene suddivisa in più iterazioni, ognuna delle quali dura solitamente da una a quattro settimane. Ad ogni iterazione, il committente sceglie via via le *user story* da implementare e si occupa inoltre di creare degli *acceptance test* che hanno lo scopo di verificare la corretta implementazione di una data *user story*. Una delle caratteristiche fondamentali di *XP* è rappresentata dal fatto che segue un approccio allo sviluppo di tipo *test-driven*, ovvero basato sui test. Ciò comporta che *acceptance test*, *unit test* e le altre tipologie di test[1], vengano implementati prima della generazione del codice software vero e proprio. Al termine di ogni iterazione tutti gli *acceptance test* vengono eseguiti per verificare la correttezza del codice. Nel caso in cui alcuni moduli non superino i test, essi verranno presi in considerazione nelle prossime iterazioni.

È possibile che durante queste ripetizioni cicliche, si presentino delle situazioni che portino a deviare dalla pianificazione iniziale. Ciò significa che è necessario effettuare dei cambiamenti (solitamente l'aggiunta o la rimozione di alcune *user story*).

Terminata l'ultima iterazione, il sistema è pronto per entrare nella fase successiva di ***productization***, durante la quale vengono effettuati ulteriori test e vengono valutate e affinate le performance del sistema. In questa fase è possibile che nuovi requisiti vengano richiesti dal committente è verrà deciso se dovranno essere inclusi nella versione corrente o meno. Tutte le idee e i suggerimenti verranno documentati e presi in considerazione nella prossima fase (fase di *manteinance*).

Al termine della fase di *productization*, la prima release del prodotto viene rilasciata al cliente e messa in produzione. Questo non significa che il prodotto è completo: ciò che è stato rilasciato, pur essendo pienamente funzionante, è un sistema che comprende il minimo di funzionalità concordate con il cliente.

La fase di ***manteinance*** è lo stato normale in cui si trova un progetto *XP*, durante la quale il sistema è in esecuzione mentre il team di sviluppo lavora per aggiungere le funzionalità mancanti. Ogni nuova release inizia con una fase di *exploration* in cui vengono prese in considerazione le *user story* ancora da implementare e vengono integrate a potenziali nuovi requisiti: potrebbe essere necessaria un'attività di *refactoring*[2] di

1 Gli *unit test* servono a verificare la correttezza di una singola unità di codice.

2 L'attività di *refactoring* consente la riscrittura di componenti software alterando la loro struttura interna ma lasciando invariato il loro comportamento esterno. Solitamente

parti di sistema attuale, potrebbe venir introdotta una nuova tecnologia nel codice esistente oppure il cliente potrebbe avere necessità di scrivere nuove *user story* per includere ulteriori funzionalità.

La fase di ***death*** avviene quando tutte le *user story* definite dal committente sono state correttamente implementate e integrate nel sistema, e il cliente non necessità di ulteriori funzionalità da sviluppare. Accertato che sistema soddisfi i requisiti di affidabilità e le performance prestabiliti, viene redatta una breve documentazione che introduca il funzionamento generale del software. Nel caso in cui in futuro il cliente avrà nuove funzionalità da integrare, questo documento sarà il punto di partenza per il team di sviluppo per richiamare alla memoria i dettagli implementativi. La fase di *death* può verificarsi anche nei casi in cui il sistema non rispetti le condizioni stabilite inizialmente, oppure diventi troppo oneroso per ulteriori sviluppi. In questo ultimo caso solitamente si predilige la realizzazione di un sistema ex-novo.

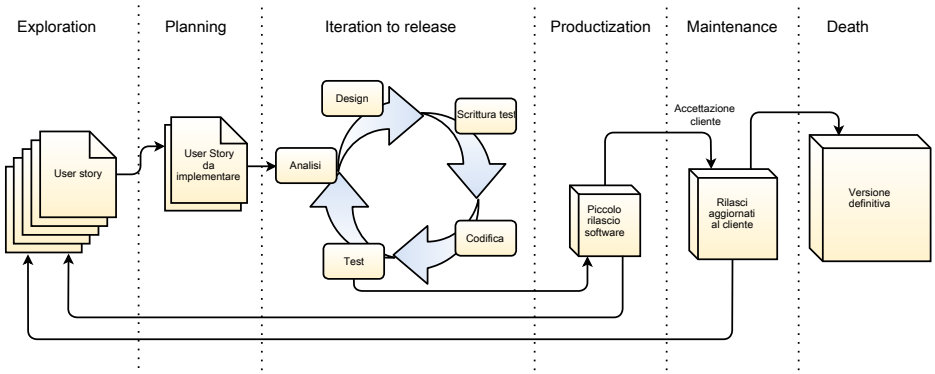


Figura 8: Ciclo di vita di XP

Come molte altre metodologie agili, anche *XP* distingue diversi ruoli all'interno del team di sviluppo.

- *Programmer* Si occupa dell'implementazione del codice e dei vari test del sistema che deve essere implementato. Beck enfatizza sul fatto che questa figura deve possedere grande capacità di comunicazione con gli altri componenti del team e deve perseguire la semplicità, cercando di non complicare inutilmente il codice.

- *Customer* Secondo Beck, la miglior figura che può assumere questo ruolo è il vero utilizzatore finale dell'applicazione che deve essere costruita. Se non fosse così, chi assume la figura del *customer* deve fare in modo di rappresentare il pensiero e le esigenze degli utenti che dovranno usare il sistema. All'interno del processo *XP*,

viene realizzata per il miglioramento di performance di esecuzione.

il *customer* scrive le user story e i test funzionali e decide quando requisiti richiesti sono correttamente soddisfatti dal sistema.

- *Tester* Dato l'approccio *test-driven* su cui *XP* si basa, Beck ha pensato di introdurre questa figura. Lo scopo del *tester* è aiutare il committente a scrivere i test funzionali partendo dalle *user story* da lui scritte. È responsabile della corretta esecuzione degli *acceptance test*, della distribuzione dei risultati e della manutenzione degli strumenti di test.

- *Tracker* Si occupa di dare feedback relativamente al processo XP. Segue le stime indicate dal team di sviluppo e ne verifica l'accuratezza, in modo da migliorare le stime future. Segue inoltre ogni iterazione dello sviluppo e stabilisce se è possibile raggiungere gli obbiettivi stabiliti con le date risorse ed entro i requisiti di tempo, o se sono necessari alcuni cambiamenti durante il processo.

- *Coach* È colui che possiede la maggiore conoscenza dei processi di *XP* e deve avere la capacità di trasmetterla agli altri componenti del team di sviluppo. È responsabile nel processo XP nella sua interezza e deve saper riconoscere quando il team sta deviando dagli obiettivi da realizzare e deve riportalo sulla giusta strada.

- *Consultant* È un membro esterno al team di sviluppo che possiede un'approfondita conoscenza tecnica necessaria per la realizzazione del progetto. Il suo scopo è guidare il team nella risoluzione di problemi di implementazione specifici.

- *Manager* Si occupa delle decisioni strategiche relative al progetto. Sono necessari un alto livello di comunicazione e un elevato feedback col team di sviluppo in modo da verificare efficacemente e tempestivamente l'attuale stato di sviluppo per determinare eventuali difficoltà o mancanze all'interno del processo *XP*.

Come già descritto all'inizio di questa sezione, le idee che stanno alla base di *XP* non sono nuove e alcune sono state formulate quasi trent'anni fa. In particolare le dodici pratiche che vengono utilizzate sono state abbandonate da tempo poiché sostituite con alcune più complesse oppure perché possedevano debolezze evidenti. Con l'appiattirsi della curva di costo del cambiamento, Beck ha reintrodotto le vecchie pratiche e le ha integrate con altre, realizzando un framework robusto dove le debolezze di una data tecnica, vengono superate dalle altre presenti nel framework.

Qui sotto descriveremo brevemente le pratiche di cui *XP* si compone.

- *Planning game* Durante la fase di *planning*, il *planning game* rappresenta il momento in cui committente e programmatori si accordano sul lavoro da svolgere: i programmatori stabiliscono l'impegno necessario per implementare le *user story* del cliente, mentre quest'ultimo stabilisce le tempistiche dei rilasci.

- *Small/short release* Durante la fase di *productization* viene costruito un semplice sistema iniziale, il quale viene rilasciato al cliente. Durante le iterazioni per la costruzione delle release successive, i requisiti verranno implementanti cercando di fare in modo di produrre una release che apporti dei piccoli incrementi e nel minor tempo possibile. Ad ogni modo tali incrementi sono decisi dal committente e solitamente rappresentano le prossime funzionalità con maggiore priorità rimaste da implementare.

- *Metaphor* Beck afferma che la metafora in questo caso rappresenta "una storia che tutti (sviluppatori, committente, manager) possono raccontare per spiegare come funziona il sistema". L'applicazione viene quindi definita con una o più metafore all'interno del team di sviluppo (es. il sistema è come una catena di montaggio; l'applicazione funziona come una scrivania virtuale). Questo serve a guidare lo sviluppo del sistema e a spiegare in maniera rapida e semplice il progetto a nuovi membri.

- *Simple Design* Durante la progettazione il codice viene mantenuto il più semplice possibile, evitando complicazioni inutili e codice extra. In questi ultimi casi, tali linee di codice vengono rimosse immediatamente. Beck aggiunge che questa pratica rappresenta l'opposto di ciò che solitamente è conosciuto come "codifica oggi progettando per il domani". Se non si è certi di come evolverà il sistema nel futuro, non ha senso includere funzionalità che potranno servire: ciò verrà fatto quando se ne presenterà la necessità.

- *Testing* Lo sviluppo de codice segue un approccio *test driven*. Gli *unit test* vengono redatti prima della produzione di codice e vengono eseguiti continuamente per verificare la correttezza del codice realizzato. Il committente (seguito dalla figura del *tester*) si incarica di scrivere i test funzionali.

- *Refactoring* Il codice viene ristrutturato in modo da eliminare eventuali ridondanze o complessità nel codice. Ciò permette di migliorare le comunicazioni all'interno del team di sviluppo, e allo stesso tempo semplifica e alleggerisce il sistema.

- *Pair programming* Durante la scrittura del codice, i programmatori lavorano a coppia utilizzando un unico computer. Mentre chi ha il controllo sul calcolatore inserisce le linee di codice, il partner controlla la correttezza delle istruzioni inserite e riflette sulla efficacia della strategia di implementazione.

- *Collective code ownership* In *XP* chiunque veda l'opportunità di aggiungere o migliorare qualsiasi porzione di codice del sistema, deve farlo. Tutti i componenti del team hanno la responsabilità della corretta implementazione del sistema. Nessuno degli sviluppatori ha una conoscenza specifica di parte del sistema, bensì tutti conoscono un po di ogni parte.

- *Continuous integration* Via via che nuove modifiche vengono apportate al sistema, vengono integrate appena possibile. In questo modo il codice viene aggiornato anche più volte al giorno. I test scritti in precedenza verificano la correttezza delle parti aggiunte e permettono di decidere se accettare o respingere le modifiche.

- *40-hours per week* I componenti del team non dovrebbero lavorare più di 40 ore a settimana. Beck sconsiglia fortemente di non rispettare questa regola per più di due settimane consecutive, in quanto sostiene la necessità per ogni componente del team di sviluppo di avere sufficiente tempo per riposare e pensare ad altro.

- *On-site customer* Il committente deve essere sempre presente e disponibile per discutere *face-to-face* con il team di sviluppo e risolvere eventuali problematiche. Deve anche avere il compito di verificare che gli sviluppatori procedano nella giusta direzione, accertandosi che gli incrementi prodotti soddisfino effettivamente le *user story* da lui definite.

- *Coding standard* Se *XP* prevede che tutti possano modificare il codice, quando se ne presenta l'opportunità (*collective code ownership*), è necessario che vengano stabilite delle regole di codifica che devono essere applicate e rispettate da tutti i componenti del team.

Beck è il primo ad affermare che le pratiche appena descritte non sono niente di unico e originale, in quanto sono già state usate (e addirittura abbandonate) in passato. Ciò che permette a *XP* di funzionare è il fatto che il particolare insieme di pratiche appena descritte, fa in modo che si supportino l'un l'altra: ognuna di esse va a coprire le debolezze delle altre, e ciò che ne risulta è un set di tecniche che porta allo stesso tempo robustezza e flessibilità al cambiamento.

Riguardo l'adozione di *XP* in un progetto software, Beck consiglia di introdurre le pratiche una alla volta (solitamente iniziando con *planning game* e l'approccio *test-driven*), ogni volta affrontando il problema principale. Una volta che il problema maggiore è stato risolto, si può affrontare quello successivo e introdurre eventualmente altre pratiche.

XP è indirizzata a team di sviluppo medio piccoli, partendo da tre persone fino a un massimo di qualche decina. La comunicazione e la collaborazione fra i componenti del team di sviluppo è un fattore di grande importanza per *XP*, di conseguenza anche l'ambiente di lavoro deve favorire queste attività. Beck consiglia che il team si posizioni in un open-space dove sia disponibile un'area comune per lo sviluppo che faciliti la pratica di *pair programming*, con eventualmente alcuni spazi personali lungo le pareti. Le situazioni in cui gli sviluppatori sono distribuiti su diverse stanze o, peggio ancora, su diversi piani o diversi edifici, sono fortemente scoraggiate e influenzano negativamente le prestazioni del processo.

3.2.3 Dynamic System Development Method

Dynamic System Development Method (successivamente identificato con *DSDM*) è una metodologia agile mantenuta dal *DSDM Consortium* [15] e descritta per la prima volta nel 1997 da Jennifer Stapleton [34].

DSDM è un processo software iterativo in cui ogni iterazione segue una versione modificata della "regola del 80-20 di Pareto": per realizzare l'80% di una applicazione software, è necessario il 20% del tempo di sviluppo che servirebbe per completare l'applicazione nella sua interezza. Ciò significa che per ogni incremento, è sufficiente una quantità di lavoro minima che permetta l'avanzamento al successivo incremento. I dettagli rimanenti potranno essere completati successivamente quando saranno disponibili più requisiti o verranno richiesti cambiamenti dal cliente.

Una delle idee principali che stanno alla base di *DMDS* è che, al posto di fissare un insieme di caratteristiche che il sistema software deve possedere e poi aggiustare budget e tempi di sviluppo, vengono prima fissati quest'ultimi e di conseguenza vengono sviluppate le funzionalità in base a questi limiti. Ciò porta ad organizzare il lavoro che viene svolto nelle varie fasi in maniera usando delle *timebox*, la cui durata viene stabilita con il cliente tenendo conto anche delle altre variabili di progetto.

DSDM si compone di cinque fasi, delle quali due sequenziali che vengono eseguite solamente la prima volta, mentre le rimanenti sono ap-

plicate ciclicamente. È possibile vedere graficamente l'interazione fra le fasi nella Figura 9.

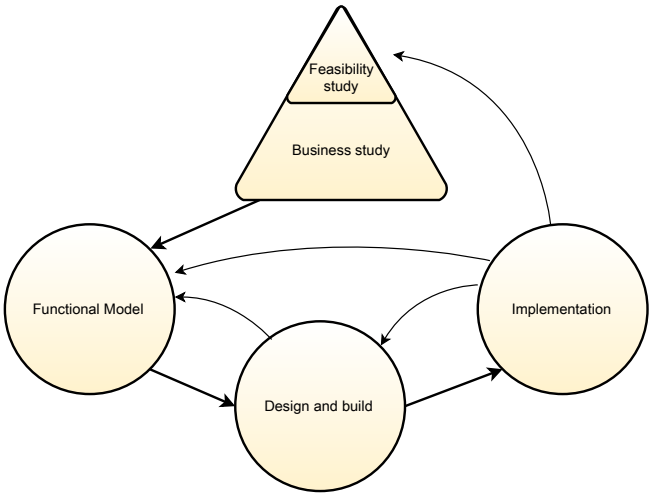


Figura 9: Ciclo di vita di *DSDM*

Nella fase iniziale di *Feasibility study*, viene stabilita la possibilità di applicare il metodo *DSDM* al progetto da realizzare. Tale decisione viene presa valutando la tipologia di progetto, la disponibilità di personale per organizzare il team di sviluppo e i dettagli tecnici per la realizzazione. Da questa fase vengono prodotti un rapporto di fattibilità e un generale piano di sviluppo. Se necessario, può essere realizzato un prototipo che permetta di comprendere meglio gli aspetti sconosciuti (es. di natura tecnologica) e decidere se procedere alla fase successiva. In generale questa fase dovrebbe durare poche settimane.

Alla fase iniziale segue la fase di *Business study* che definisce le caratteristiche tecnologiche del prodotto, l'area di business e le relative persone coinvolte da parte del committente. In questo modo si potranno riconoscere i vari utilizzatori interessati, favorendo l'interazione e lo scambio di informazioni con il cliente. Sempre in questa fase viene definita un'architettura di sistema, che può comunque essere modificata durante le successive fasi del processo. Viene infine redatto un piano di prototipazione che stabilisce la strategia di implementazione dei prototipi negli stadi successivi.

La fase definita come *Functional model* è la prima a prevedere iterazioni successive. A questo stadio vengono prodotti dei prototipi incrementali[3] che hanno lo scopo di mostrare le varie funzionalità. Testando tali prototipi il cliente può fornire il feedback necessario per ottenere requisiti aggiuntivi.

3 Solitamente il tipo di prototipazione utilizzata è del tipo evolutivo descritta nel capitolo precedente.

Parallelamente a questi prototipi, viene redatta una documentazione contenente: la lista delle funzioni implementate nell'iterazione attuale, i commenti ricevuti dal committente, che permettono di rivedere i requisiti nella successiva iterazione, e la lista delle funzionalità mancanti che verranno implementate nello stadio successivo. In questa fase viene inoltre redatto un documento relativo all'analisi del rischio, in quanto nelle fasi successive i problemi incontrati saranno molto più difficili da affrontare.

Nell'iterazione di *Design and build* il sistema viene implementato con lo scopo di ottenere un prodotto software collaudato che soddisfi almeno i requisiti minimi concordati con il cliente. Anche in questa fase rimane fondamentale l'interazione con il cliente in quanto è ancora possibile aggiungere eventuali modifiche: in questi casi può essere ripetere la fase di *Functional model*.

La fase di *Implementation* identifica il momento in cui il sistema software viene trasferito dall'ambiente di sviluppo all'ambiente di produzione, ovvero viene consegnato al cliente il quale viene inoltre istruito all'uso corretto del software. La documentazione redatta in questa fase comprende il manuale utente e una revisione dell'intero progetto che riassume i risultati raggiunti e che può comprendere spunti per eventuali miglioramenti. In base a questi risultati raggiunti il processo prende strade diverse: se tutti i requisiti emersi durante le iterazioni del modello sono soddisfatti, non è necessario ulteriore lavoro. Se invece sono evidenti alcune mancanze, in base alla loro entità il processo può reiterare in una delle fasi intermedie o, nei casi peggiori, reiniziare con la fase di *Feasibility study*.

DSDM identifica anche un insieme di ruoli per chi sarà coinvolto nel progetto. I vari ruoli sono quindici e si distribuiscono fra il team di sviluppo e il committente del software o i suoi utilizzatori. Il gran numero di incarichi porta ad avere situazioni in cui una persona svolga più ruoli all'interno del progetto; allo stesso modo può accadere che a più componenti venga assegnato lo stesso ruolo. Elenchiamo qui sotto solo i ruoli principali.

- *Developer*: è l'unico ruolo che si occupa dello sviluppo. In base all'esperienza che possiede nell'eseguire un dato compito, può acquisire lo status di *Senior Developer* e acquisire anche un certo livello di leadership. Il ruolo di *Developer* è svolto da ogni componente dello staff di sviluppo, sia egli un programmatore, un analista, un progettista, ecc.

- *Technical Coordinator*: definisce l'architettura di sistema ed è responsabile della qualità tecnica del progetto.
- *Ambassador User*: questo ruolo è il più importante fra quelli assegnati ai futuri utilizzatori del sistema software. Il suo ruolo principale è trasmettere la conoscenza della comunità di utilizzatori al team di sviluppo del progetto e specularmente informare gli utilizzatori sull'andamento dello sviluppo. In questo modo è garantito il feedback necessario per il buon svolgimento del progetto.
- *Visionary*: è la persona che, da parte del cliente, più di altri ha chiari gli obiettivi finali del sistema. Solitamente è anche colui che manifesta l'idea iniziale di sviluppare il progetto in questione. Il suo scopo è fare in modo che emergano il prima possibile tutti i requisiti dell'applicazione e che quest'ultimi vengano soddisfatti durante il processo di sviluppo.
- *Executive Sponsor*: è il ruolo da parte del committente che ha responsabilità e autorità finanziarie e, di conseguenza, il potere decisionale sulle scelte progettuali.

Durante tutta l'esecuzione del processo, *DSDM* prescrive un insieme di pratiche (che all'interno del consorzio sono conosciute come "principi") fondamentali per la buona riuscita del progetto. Si può vedere come molti di questi principi rispecchino quelli del manifesto agile, precedentemente descritti in questo capitolo, e siano condivisi da altre metodologie.

- Il cliente va coinvolto attivamente nel progetto.
- Il team DSDM deve avere un alto potere decisionale per la risoluzione dei problemi incontrati.
- È necessario focalizzare l'attenzione sul rilascio frequente di software.
- L'essenziale criterio di accettazione dei rilasci è il soddisfacimento dei requisiti di business.
- Lo sviluppo incrementale e iterativo è fondamentale per raggiungere una soluzione accurata che permetta di raffinare via via i requisiti.
- Tutti i cambiamenti effettuati durante lo sviluppo devono essere reversibili.

- I requisiti vengono stabiliti ad alto livello.
- L'attività di testing è integrata lungo tutto il ciclo di vita di *DSDM*
- È essenziale un approccio cooperativo e collaborativo fra tutti i soggetti coinvolti nel progetto.

Solitamente un progetto viene seguito da un team composto da un minimo di due (una persona per gli sviluppatori e una per il cliente) a un massimo di sei componenti. Nell'ultimo framework di *DSDM* rilasciato dal *DSDM Consortium* sono presenti delle linee guida che consentono di integrare questa metodologie con *Extreme Programming* di Kent Beck, unendo un solido modello di processo alle eccellenti pratiche dedicate alla programmazione di Beck.

3.2.4 Feature Driven Development

Le prime idee di *Feature Driven Development* furono introdotte nel 1997 da Jeff De Luca e Peter Coad i quali erano coinvolti in un grande progetto di sviluppo software a Singapore. L'eccessiva complessità di tale progetto fece presto capire a De Luca, che a quel tempo era il project manager, che non si sarebbe potuto terminare il lavoro nella data prevista usando le tradizionali strategie di sviluppo software. Con l'aiuto di Peter Coad e altri colleghi scoprirono la modellazione a tecniche di colore e il concetto di *Feature Driven Development*, raccogliendo la loro esperienza in un libro del 1999 [12]. Negli anni successivi Stephen Palmer, collaboratore di Jeff De Luca, e Mac Felsing pubblicarono una descrizione più completa di questa metodologia [26].

Feature Driven Development è un approccio agile e adattivo alla costruzione di sistemi informatici. La parola *feature*, ovvero "caratteristica", è una qualsiasi funzione che può essere descritta dal cliente secondo il template **<azione> <risultato> <da-per-di-a> <oggetto>** (es. calcola totale acquisti del cliente) e che può essere implementata indicativamente in due settimane. L'utilizzo delle *feature* corrisponde al concetto di *user story* usato in *Extreme Programming* e funge da fonte primaria di descrizione dei requisiti.

Questa metodologia non si occupa di tutto il processo di sviluppo software, bensì si focalizza sulle fasi di design e di implementazione e suddivide in cinque step sequenziali il lavoro da svolgere (Figura 10), assumendo che la raccolta dei requisiti del sistema sia già stata completata.

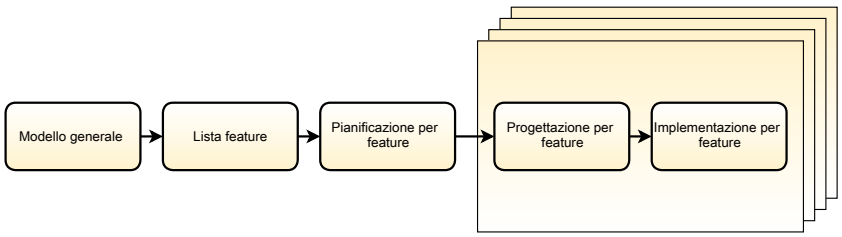


Figura 10: Ciclo di vita di *FDD*

La prima fase ha lo scopo di produrre un modello completo del sistema. I requisiti sono già stati raccolti e documentati sotto forma di casi d'uso o di specifiche funzionali. Il team del committente descrive ad alto livello il funzionamento del sistema al team degli sviluppatori e vengono identificate delle aree di dominio per il sistema da costruire. Si continua approfondendo i dettagli di ogni area di dominio e il team di sviluppo, suddiviso in piccoli gruppi, realizza un modello per ogni area di dominio. A questo punto vengono discussi e valutati i vari modelli, che successivamente vengono uniti a formare un modello generale.

Nella fase successiva viene costruita una "lista di *feature*". L'insieme dei vari modelli delle aree di dominio, uniti alla documentazione dei requisiti, permettono al team di sviluppo di estrarre quelle *feature* che rappresentano le funzioni determinanti per il cliente. Per ogni area di dominio vengono presentate le relative funzioni, le quali formano le *feature* di dominio o *feature set*. Cliente e utenti revisionano questi insiemi di *feature*, valutandone validità e completezza.

Dopo che la lista di *feature* è stata approvata, si inizia una "pianificazione tramite *feature*" in cui vengono valutate le priorità e le dipendenze dei vari insiemi di caratteristiche e viene stabilita la sequenza con cui verranno implementate.

Le fasi successive di "progettazione e implementazione tramite *feature*" si svolgono in maniera iterativa. Un limitato set di *feature* viene estratto dalla lista di caratteristiche del sistema da implementare e viene assegnato a dei gruppi ristretti del team di sviluppo. In un intervallo di tempo che va da pochi giorni a un massimo di due settimane, le *feature* prese in considerazione vengono completate; durante ogni iterazione, gli sviluppatori svolgono le consuete attività di sviluppo software (progettazione, codifica, unit test e integrazione). Al termine dell'iterazione le *feature* implementate vengono integrate nella *main build* del sistema e inizia un'altra iterazione, con l'assegnazione di nuove *feature* agli sviluppatori.

Come nelle precedenti metodologie descritte, anche in *FDD* vengono definiti alcuni ruoli che vengono assegnati alle varie figure che partecipano al progetto. Analogamente a *DSDM* un ruolo può essere interpre-

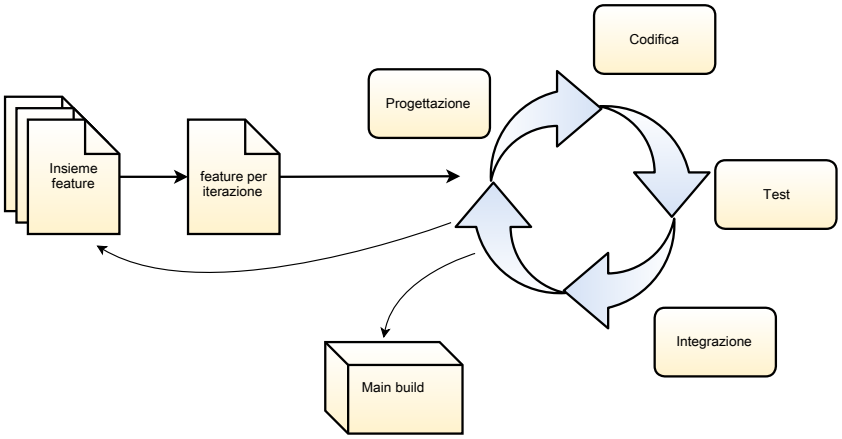


Figura 11: Iterazione di "progettazione e implementazione tramite *feature*" di *FDD*

tato da vari membri del team e, allo stesso modo, una stessa persona può interpretare più ruoli. In questa metodologie gli incarichi vengono raggruppati in tre categorie: i "ruoli chiave" sono sei e comprendono *project manager, chief architect, development manager, chief programmer, class owner* e infine *domain expert*. A questi vengono aggiunti i "ruoli di supporto" che sono *release manager, language guru, build engineer, toolsmith* e *system administrator*. Completano il quadro i "ruoli aggiuntivi" necessari, i quali sono rappresentati da *tester*, *deployer* e *tecnical writer*. Una breve descrizione dei ruoli più importanti verrà presentata qui sotto.

- **Project manager**: si occupa degli aspetti finanziari e amministrativi del progetto. Il suo compito è di seguire il team di progetto facendo in modo che ci siano le adeguate condizioni lavorative ed eliminando le eventuali distrazioni esterne. È colui che ha l'ultimo potere decisionale.

- **Chief architect**: è responsabile della progettazione del sistema nella sua interezza e ha potere decisionale sulle problematiche che emergono durante il design dell'applicazione. Se necessario, questo ruolo si può scindere in *Domain architect* e *Technical architect*.

- **Development manager**: si occupa di stabilire le attività di sviluppo giornaliere del team e risolve gli eventuali conflitti all'interno di quest'ultimo. Il suo compito è anche quello di organizzare le risorse necessarie allo svolgimento del progetto.

- **Chief programmer**: è uno sviluppatore che ha alle sue spalle diversi anni di esperienza e segue gli altri programmatori lungo tutto il processo. Il suo compito è quello di guidare gli altri sviluppa-

tori nelle attività di analisi, progettazione e implementazione di nuove *feature* e di scegliere quali verranno implementate nelle iterazioni successive. Ogni *feature* viene suddivisa in classi e ognuna di queste viene assegnata ad un componente del team di sviluppo il quale diventa il *Class Owner* di quella particolare classe. Il *Chief programmer* si occupa inoltre di risolvere i problemi di natura tecnica e di redarre un rapporto settimanale riguardante i progressi del team.

- **Domain expert**: è colui che possiede la conoscenza e l'esperienza di come i diversi requisiti del sistema in via di sviluppo dovranno relazionarsi. Questa figura ha il compito di trasmettere le sue competenze agli sviluppatori e assicurarsi che possano così implementare un prodotto che soddisfi i requisiti stabiliti. Solitamente tale ruolo è ricoperto da un cliente o un suo rappresentante, il quale può essere l'utilizzatore finale. Nel caso in cui ci siano più *Domain expert*, è presente anche un *Domain manager* il quale deve fare in modo di risolvere le eventuali opinioni differenti sui requisiti del sistema.

- **Release manager**: è responsabile del progresso dello sviluppo del sistema e viene aggiornato dello stato di sviluppo, attraverso i rapporti dei *Chief programmer*. È infine incaricato di informare il *Project manager* sullo stato di tali progressi.

- **Language guru**: possiede un'approfondita conoscenza dei linguaggi di sviluppo che vengono impiegati nel progetto oppure è esperto in una particolare nuova tecnologia.

I ruoli rimanenti vengono assegnati all'interno del team di sviluppo e sostanzialmente suddividono le varie responsabilità del progetto alle diverse figure definite (es. il *toolsmith* realizza piccoli tool utili allo sviluppo del sistema per gli altri componenti del team, il *system administrator* segue la configurazione e la gestione dei server, della rete e degli ambienti di sviluppo utilizzati dal team, ecc.)

Anche questa metodologie enuncia una serie di *best practise* riconosciute dall'industria del software, le quali vengono applicate dal team di sviluppo durante le varie fasi del processo di *Feature driver development*. La pratica più importante, che è quella che caratterizza questa metodologia, è *Develop By Feature* ovvero la decomposizione del sistema da sviluppare, in piccole funzionalità che possono essere implementate in poche settimane. A questa sono collegati i *feature team*, i quali rag-

gruppano componenti del team di sviluppo che si occupano di una data caratteristica.

Una delle pratiche derivate dall'esperienza iniziale di Jeff De Luca che gettò le basi di *feature driven development* risulta essere l'utilizzo di un "codice a colori" per identificare visivamente in maniera rapida le dinamiche di interazione dei vari dominii.

La pratica di *regular build* si riferisce al fatto che deve essere sempre disponibile una versione funzionante e dimostrabile del sistema in via di sviluppo. Queste *regular build* fungono da punto di partenza per l'aggiunta di nuove funzionalità. A questa è direttamente collegata la pratica di *configuration management* che permette di tiene traccia e di identificare la cronologia di modifiche che sono state apportate a un file sorgente, funzionalità fondamentali nel momento in cui si aggiungono nuove caratteristiche ad una *main build*.

La pratica di *code inspection* viene utilizzata durante tutto il processo di sviluppo ogni qualvolta si effettua la revisione di codice sorgente e rappresenta un'eccellente meccanismo di rilevazione di bug di programmazione.

L'esperienza di utilizzo dei primi suggeritori di *Feature Driven Development*, Jeff De Luca e Peter Coad, comprende la sua adozione di questa metodologia in progetti medio/lunghi con un gran numero di componenti (essi ricordano casi di 50 e addirittura 250 persone e una durata del progetto di circa 18 mesi). Palmer e Felsing [26] affermano la compatibilità del processo sia con nuovi progetti software, sia con progetti che richiedono misure correttive ad un prodotto esistente.

Un ultimo suggerimento per l'adozione di tale metodo da parte dei suoi autori è simile a quello proposto da Beck per *Extreme Programming*, ovvero di adottare questo metodo gradualmente a piccoli pezzi e via via nella sua interezza man mano che il progetto procede.

Abbiamo visto in questo capitolo come, per superare le limitazioni di requisiti incerti, fornire rapidamente feedback e realizzare tempestivamente una soluzione software, vari esperti idearono nuovi metodi di sviluppo software che integrassero come principio l'idea del cambiamento, questo per reagire a modifiche impreviste e non per cercare di controllarle.

Nel prossimo capitolo verrà introdotta l'indagine che ha permesso di raccogliere un insieme iniziale di informazioni sulla conoscenza e la diffusione di queste nuove metodologie in un campione di intervistati

operanti nell'ambito della produzione software.

4 REALIZZAZIONE DELL'INDAGINE

In questo capitolo introdurremo l'indagine che è stata realizzata per raccogliere un insieme di informazioni sulla diffusione di pratiche e metodologie di *Agile Software Development*. Queste informazioni riguardano la consapevolezza dell'esistenza di queste metodologie agili, quali fra queste metodologie sono le più conosciute, quali le più utilizzate e in che percentuale.

Ricerche di questo tipo sono già presenti in letteratura: l'ingegnere del software canadese Scott Ambler conduce annualmente vari tipi di survey a livello internazionale riguardanti i temi di adozione, tecniche e strategie agili e presenta i risultati sul suo sito [2]. I ricercatori finlandesi Outi Salo e Pekka Abrahamsson hanno valutato l'utilità di metodologie agili percepita all'interno di un insieme di aziende di software europee [30], mentre il ricercatore americano Andrew Begel e il suo collega Nachiappan Nagappan hanno condotto uno studio esploratorio all'interno di Microsoft [6]. Nel panorama italiano è stata avviata negli ultimi giorni di marzo 2013 un indagine relativa all'utilizzo di metodologie agili nelle PMI [21].

Prendendo visione di un insieme di ricerche già concluse, si è potuto rilevare un set di tematiche interessanti relative alle metodologie agili, da includere nella nostra indagine.

4.1 PROGETTAZIONE QUESTIONARIO

Con lo scopo di raccogliere le informazioni necessarie, si è deciso di condurre l'indagine attraverso una survey, ovvero un sondaggio. Esistono varie possibilità per condurre una survey, fra cui l'intervista diretta, l'intervista telefonica, l'invio di un questionario via posta. Ormai da diversi anni sono inoltre disponibili diverse piattaforme (es. *Lime Survey*, *Survey Monkey*, *Survey Gizmo*) che permettono la redazione di questionari web, da diffondere successivamente via email o da condividere su

pagine personali. I vantaggi derivati dall'utilizzo di un questionario web sono molteplici: i dati vengono memorizzati automaticamente in formato elettronico in un database, il quale è consultabile durante tutto il periodo dell'indagine. In questo modo i ricercatori possono tenere traccia delle compilazioni ed, eventualmente, sollecitare la compilazione in caso di limitata partecipazione. Inoltre molte piattaforme rendono il database esportabile in vari formati compatibili con i maggiori software di analisi statistica (es. *IBM SPSS*, *Matlab*, *R*). Un ulteriore vantaggio del condurre un'indagine attraverso un questionario on-line riguarda la possibilità di ridurre considerevolmente l'errore umano dovuto all'inserimento manuale, all'interno di un database, di varie compilazioni cartacee. Considerando anche costi e velocità di diffusione, questo tipo di questionari supera varie limitazioni, se comparato agli approcci tradizionali; oltre a ciò la bibliografia ne conferma un più alto tasso di risposta [22].

Volendo seguire delle linee guida per la creazione della survey, il sociologo ed esperto di metodologie di indagine Robert Groves suggerisce un processo sistematico composto da varie fasi [16], del quale è presente una schematizzazione in Figura 12. Sarà questo l'approccio che seguiremo nelle prossime sezioni.

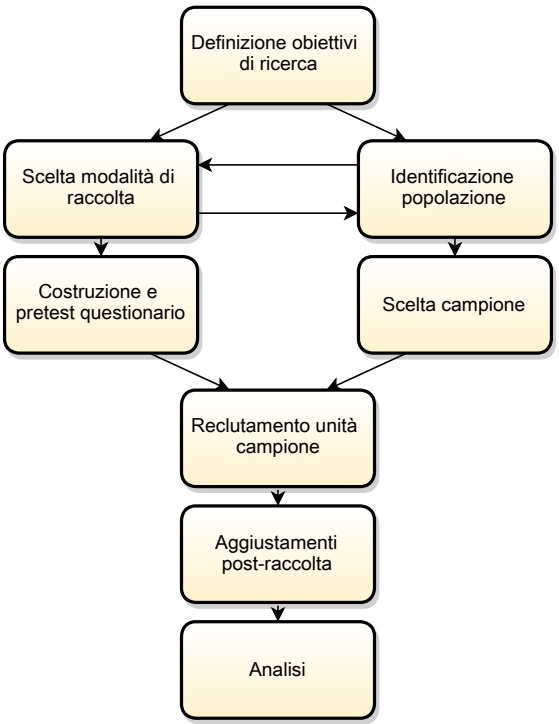


Figura 12: Processo di realizzazione di una survey

4.2 DEFINIZIONE DEGLI OBIETTIVI DI RICERCA

L'obiettivo che ci si prefigge in questo elaborato è la realizzazione di uno studio esplorativo sulla diffusione di metodologie di *Agile Software Development*. Con questo si intende la raccolta di un insieme di dati che permettano di valutare se questo tipo di approcci alla produzione software siano diffusi o meno fra gli sviluppatori e se tali metodologie hanno influenzato anche il modo di lavorare di approcci più tradizionali. Dai risultati di questo studio preliminare si potranno avanzare nuove ipotesi di ricerca, per concentrarsi su particolari aspetti di interesse specifico e approfondire la conoscenza di queste tematiche.

4.3 SCELTA DELLA MODALITÀ DI RACCOLTA

Per raggiungere il nostro scopo si è deciso di procedere con la creazione di un questionario online in quanto permette di diffondere l'indagine in maniera semplice ed economica. La scelta della piattaforma da utilizzare è ricaduta sul prodotto gratuito e open source *Lime Survey* [20]. I motivi che hanno portato a scegliere *Lime Survey* rispetto ad altre soluzioni, come ad esempio la più conosciuta *Survey Monkey* [35], riguardano i limiti imposti al numero domande inseribili e il numero di partecipanti. *Lime Survey* non impone limiti al numero di domande e di compilazioni e, nel caso in cui si scelga di installare l'applicazione in un server di proprietà personale, l'uso è totalmente gratuito. Al contrario, la versione gratuita di *Survey Monkey* limita a dieci le domande per ogni questionario e a cento le relative compilazioni. Nel caso in cui un utente desiderasse passare alla più economica versione a pagamento disponibile, il costo è di 25€ al mese per l'account base.

Considerate le necessità, si è scelto di sfruttare il servizio di hosting dedicato offerto da *Lime Survey* (*Lime Service* [19]) in quanto comprende venticinque compilazioni gratuite al mese e, all'occorrenza, con pochi euro di spesa è possibile acquistare pacchetti di compilazioni aggiuntive. In questo modo si sono potute garantire le adeguate prestazioni dell'applicazione e la continuità del servizio che avrebbero potuto mettere in difficoltà un server casalingo, senza contare il fatto che sarebbe stato preferibile avere il sistema in funzione 24h su 24h.

4.4 IDENTIFICAZIONE DELLA POPOLAZIONE E SCELTA DEL CAMPIONE

Coloro i quali possono fornirci le informazioni di nostro interesse, sono i professionisti di sviluppo software che esercitano questa attività in ambito lavorativo e formano, assieme ai colleghi, un team di sviluppo software. Ciò che ci interessa ricavare dall'indagine è la diffusione di conoscenza, uso e impressioni riguardanti pratiche e metodologie agili da parte di questi team di sviluppo, questo perché tali metodologie non si limitano all'uso individuale di una particolare pratica, bensì assumono l'esistenza di un gruppo di utilizzatori. Di conseguenza si può indicare come popolazione in esame la popolazione di team di sviluppo software.

Campionamento non-probabilistico

Dato il fatto che la nostra è una ricerca esplorativa preliminare, si è scelto di procedere con la scelta del campione con un campionamento di tipo non-probabilistico. La differenza fra le tecniche di campionamento di tipo non-probabilistico rispetto a quelle di tipo probabilistico è che nelle prime, i campioni vengono selezionati in base alla valutazione soggettiva dei ricercatori responsabili del sondaggio, piuttosto che la selezione casuale delle unità campionarie. Di conseguenza non tutte le unità campionarie della popolazione hanno una probabilità non nulla di essere scelte. La metodologia di campionamento non-probabilistico è di grande utilità e largamente usata nei casi di studi pilota, ricerche qualitative, casi studio e sviluppo di ipotesi.

I vantaggi di questo tipo di approccio sono la semplicità e la rapidità con cui lo studio viene realizzato. A questi vantaggi corrisponde tuttavia una serie di elementi negativi che non vanno tralasciati: i dati raccolti possono essere facilmente affetti da errori sistematici, il campione potrebbe non essere rappresentativo per la popolazione in esame, in quanto è solitamente soggetto da bias di selezione, ovvero a una distorsione dovuta al suo metodo di selezione. Ad ogni modo i dati ottenuti con questo tipo di approccio potranno essere sfruttati per la progettazione di ricerche più precise e raffinate, dando allo stesso tempo un'idea generale della diffusione attuale delle metodologie in esame, all'interno dell'ambiente analizzato.

Esistono vari tipi di campionamento non-probabilistico: alcuni esempi sono il "campionamento per quote", il "campionamento di convenienza", il "campionamento ragionato" e il "campionamento a valanga" [38]. Per la scelta delle nostre unità campionarie da includere nell'indagine si è scelto un campionamento di convenienza e ragionato.

Il "campionamento di convenienza" è una metodologia che viene usata spesso in studi esploratori quando il ricercatore è interessato ad ottenere informazioni che approssimano la verità in maniera facile, veloce ed economica. In questo caso il ricercatore seleziona le unità del campione in maniera che siano le più semplici a cui avere accesso. La sua naturale estensione è il "campionamento ragionato", in cui il ricercatore seleziona le unità campionarie in base al suo giudizio basandosi su esperienza e conoscenza personale.

Nel nostro caso si è scelto di inviare il questionario a un insieme di persone che si sapeva essere coinvolte nell'attività di sviluppo software a livello professionale. Parte di queste persone sono ex-studenti del corso di Ingegneria Informatica Triennale e Magistrale dell'Università degli Studi di Padova, altri sono colleghi di lavoro e altri ancora conoscenze che si occupano di realizzazione di applicazioni in azienda. Ulteriori nominativi provengono da fonti le quali conoscono figure corrispondenti al profilo del professionista in ambito di sviluppo software.

Riguardo la numerosità del campione, una ricerca esplorativa di questo tipo che ha lo scopo di acquisire maggiore comprensione di una data situazione e fornire indicazioni per indagini future, prevede che generalmente il dato di ricerca sia trattato in modo qualitativo. In questo caso la numerosità suggerita per lo studio varia fra le 50 e 60 compilazioni [13].

4.5 COSTRUZIONE E TEST DEL QUESTIONARIO

In questa sezione verranno mostrate le domande che hanno costituito la survey. La piattaforma *Lime Survey* fornisce un set di tool per la creazione automatica di domande e le relative risposte. Sono disponibili svariate tipologie di risposta, fra cui risposte a scelta multipla, risposte a scelta singola, array di domande, domande aperte, ecc. È inoltre disponibile la possibilità di mostrare o nascondere particolari domande o gruppi di domande sulla base di condizioni da configurare valutando le risposte date alle domande precedenti (es. se viene risposto "NO" alla domanda 3, le domande 5 e 6 non devono essere visualizzate). Ciò permette di organizzare in maniera ottimale il flusso di compilazione senza introdurre ambiguità per il compilatore meno attento.

Creato l'account per la survey su "Lime Survey", è stato impostato il nome che avrebbe assunto l'indagine per i futuri intervistati: molto semplicemente si è scelto "Indagine su Agile Software Development". A questo punto si è configurato il messaggio di benvenuto e si è proceduto con l'inserimento delle varie domande precedentemente studiate e

organizzate, le quali verranno introdotte di seguito.

È stato identificato un totale di 21 domande, suddivise in 7 gruppi di pertinenza: "conoscenze", "progetto", "pratiche di sviluppo software", "valutazioni", "precedenti esperienze con metodologie agili", "informazioni personali" e "considerazioni personali". Ogni gruppo verrà presentato qui di seguito.

4.5.1 Conoscenze

Il primo gruppo di domande mira a stabilire se le l'intervistato conosce le metodologie di *Agile Software Development*, quali metodologie conosce e quale metodologia agile adotta per la gestione del processo di sviluppo software assieme al suo team di sviluppo.

1. *"Conosce le metodologie di Agile Software Development?"*

   Questa prima domanda serve a stabilire se il compilatore conosce o ha sentito parlare di metodologie di *Agile Software Development*. Le risposte possibili sono palesemente *"SI"* oppure *"NO"*.

2. *"Quali metodologie di Agile Software Development conosce?"*
   - *"Scrum"*
   - *"Extreme Programming"*
   - *"Feature Driven Development"*
   - *"Crystal Family"*
   - *"Dynamic System Development Methods"*
   - *"Adaptive Software Development"*
   - *"Lean Software Development"*
   - *"Altro"*

   La seconda domanda è soggetta a condizione, in quanto viene visualizzata solo nel caso in cui l'intervistato risponda positivamente alla prima domanda. Questo quesito richiede di indicare quali siano le metodologie conosciute. In questo caso il compilatore può scegliere più risposte fra quelle proposte, in quanto è possibile che chi risponde possa conoscere più metodologie. È inoltre presente la scelta *"Altro"*, la quale permette di inserire metodologie non presenti nella lista suggerita. La scelta di quali metodologie suggerire è stata effettuata sulla base di altre ricerche già condotte [6].

3. *"Quale metodologia di Agile Software Development sta attualmente applicando nel suo team di sviluppo?"*

- *"Attualmente non applico alcuna metodologia di Agile Software Development"*
- *"Scrum"*
- *"Extreme Programming"*
- *"Feature Driven Development"*
- *"Crystal Family"*
- *"Dynamic System Development Methods"*
- *"Adaptive Software Development"*
- *"Lean Software Development"*
- *"Altro"*

Anche questa domanda viene visualizzata solo nel caso in cui l'intervistato risponda positivamente alla domanda numero uno. Le opzioni di scelta prevedono di indicare solo una delle metodologie suggerite e, anche in questo caso, è prevista la possibilità di specificare la scelta *"Altro"* e di inserire manualmente una metodologia non presente in lista.

4.5.2 Progetto

Il secondo gruppo di domande serve ad identificare un insieme di informazioni riguardanti il progetto su cui il team di sviluppo sta lavorando e sono state presentate a tutti i compilatori dell'indagine.

4. *"Da quante persone è composto il team di progetto nel quale sta lavorando?"*
   - *"Meno di 4"*
   - *"Da 4 a 9"*
   - *"Da 10 a 24"*
   - *"Da 25 a 50"*
   - *"Più di 50"*

   La prima domanda di questo gruppo serve a identificare la dimensione del team di sviluppo del compilatore in uno degli intervalli proposti.

5. *"Qual'è il livello di esperienza nella produzione software all'interno del team?"*
   - *"La maggioranza dei componenti del team ha meno di 3 anni di esperienza."*

- *"La maggioranza dei componenti del team ha dai 3 ai 6 anni di esperienza."*
- *"La maggioranza dei componenti del team ha più di 6 anni di esperienza."*
- *"I livelli di esperienza sono molto vari."*

Lo scopo di questa domanda è misurare approssimativamente il livello di esperienza del team di sviluppo. I livelli di esperienza suggeriti sono in linea con quelli proposti da Salo e Abrahamsson [30] con la differenza che è stata aggiunta un ulteriore opzione per offrire un'opportunità di risposta nel caso in cui quelle precedenti non descrivessero la situazione e identificano team di sviluppo particolarmente eterogenei.

6. *"Qual'è la durata prevista del progetto a cui sta lavorando?"*
   - *"Meno di 3 mesi."*
   - *"Da 4 a 6 mesi."*
   - *"Da 7 a 12 mesi."*
   - *"Da 1 a 2 anni."*
   - *"Più di 2 anni."*

   Questa domanda stabilisce la durata prevista del progetto software a cui il team di sviluppo sta lavorando.

7. *"Quale tipologia di software sta sviluppando?"*
   - *"Software business-to-business a supporto del processo (software per produzione industriale, controllo e gestione di macchinari, software gestionale, ecc.)."*
   - *"Software business-to-business a supporto del prodotto (software per dispositivi, elettrodomestici, giocattoli, ecc.)."*
   - *"Software business-to-customer (es. applicazioni di produttività personale, intrattenimento, ecc.)."*
   - *"Applicazione per dispositivi mobile."*
   - *"Web site/web application per aziende."*

   La presente domanda mira ad inquadrare la tipologia di software che il team di sviluppo sta producendo.

8. *"Nell'ipotesi di un malfunzionamento del software che sta sviluppando, che tipo di perdita si può verificare?"*
   - *"Perdita trascurabile."*

- *"Perdita di informazioni."*
- *"Perdita economica."*
- *"Perdita o danneggiamento di mezzi o macchinari."*
- *"Danni ambientali."*
- *"Perdita di vite umane o gravi danni a persone."*

Questa domanda si propone di identificare il tipo di perdita che si può verificare nell'ipotesi di un malfunzionamento del sistema software sviluppato dal team di sviluppo del rispondente. Le ultime tre risposte suggerite identificano le casistiche dei cosiddetti "sistemi a sicurezza critica", conosciuti anche come *safety-critical system* o *life-critical system* [33], nei quali il malfunzionamento del sistema può causare danni ad attrezzature, danni ambientali o può minacciare vite umane. Esempi di questi casi possono essere il sistema di controllo per una industria chimica oppure sistemi software per apparecchiature mediche.

9. *"Il committente del software (o un suo rappresentante) fa parte del team di sviluppo?"*
   - *"Si."*
   - *"No, ma è facilmente reperibile per spiegazioni o chiarimenti."*
   - *"No e risulta difficile mettersi in contatto con lui."*
   - *"Non so."*
   - *"Non applicabile."*

   Con questa domanda si cerca di individuare se il committente, o chi per lui ne fa le veci, è parte integrante del team di sviluppo software, come fortemente suggerito da gran parte delle metodologie di *Agile Software Development*.

10. *"Scelga la risposta che descrive meglio il suo caso: «I vari componenti del mio team di sviluppo si trovano: »"*
    - *"nella stessa stanza."*
    - *"distribuiti nello stesso edificio."*
    - *"distribuiti in più sedi."*
    - *"distribuiti in più Paesi."*

    L'ultima domanda del gruppo relativo alle informazioni sul progetto, riguarda la distribuzione geografica del team di sviluppo.

4.5.3 Pratiche di sviluppo software

In questo gruppo stato chiesto ai rispondenti di fornire la misura di adozione di un insieme di pratiche di sviluppo software

11. *"Indichi in quale misura adotta le seguenti pratiche di sviluppo software nel progetto al quale sta lavorando."*
    - *"Co-location dei componenti del team"*
    - *"Integrazione continua del codice"*
    - *"Ritmo di lavoro sostenibile (max 40h a settimana)"*
    - *"Piccole e frequenti release di software funzionante"*
    - *"Coinvolgimento attivo del cliente"*
    - *"Autogestione del team riguardo il lavoro da svolgere"*
    - *"Riunioni giornaliere del team"*
    - *"Comunicazione face-to-face"*
    - *"Pair programming"*

    Per ogni voce di questa domanda era possibile fornire una fra le seguenti risposte: 1) *Sistematicamente*; 2) *Spesso*; 3) *A volte*; 4) *Mai*; 5) *Non applicabile*; 6) *Non conosco*.

    La pratiche suggerite in questa domanda sono *best-practise* considerate agili, molte delle quali sono integrate in metodologie agili come *Extreme Programming* e *Scrum*. Questa domanda è stata posta a tutti i partecipanti all'indagine e ha lo scopo di verificare l'adozione di pratiche definite agili anche in ambienti in cui non vi è una completa conoscenza o coscienza di utilizzo di questi metodi.

12. *"Indichi in quale misura adotta queste altre pratiche di sviluppo software nel progetto al quale sta lavorando."*

    - *"Team coding standard"*
    - *"System metaphore"*
    - *"User story"*
    - *"Collective code ownership"*
    - *"Test-driven development"*
    - *"Acceptance test basati su user story"*
    - *"Product backlog"*

- *"Burndown chart"*

Per ogni voce di questa domanda era possibile fornire una fra le seguenti risposte: 1) *Sistematicamente*; 2) *Spesso*; 3) *A volte*; 4) *Mai*; 5) *Non applicabile*; 6) *Non conosco*.

Anche l'insieme di pratiche di questa domanda, come quelle relative alla domanda precedente, sono considerate pratiche agili. Essendo queste più specifiche, si è deciso di separarle dalle precedenti e di proporle solo a coloro i quali avessero risposto positivamente alla prima domanda del questionario (*"Conosce le metodologie agili?"*).

4.5.4 Valutazioni

Questo gruppo di domande raccoglie un insieme di affermazioni generali sulla produzione software alle quali ai rispondenti è stato chiesto se fossero d'accordo o meno e in quale misura.

13. *"Dia una valutazione alle seguenti affermazioni:"*
    - *"E' diffusa una certa diffidenza fra gli sviluppatori riguardo all'accogliere la modifica dei requisiti in stadi avanzati dello sviluppo software."*
    - *"E' fondamentale che il team di sviluppo abbia la capacità di autoorganizzarsi."*
    - *"Il cambiamento di requisiti, anche a stadi di sviluppo avanzati, va tutto a vantaggio competitivo del committente."*
    - *"Modificare o aggiungere requisiti a stati avanzati di sviluppo di un progetto software aumenta esponenzialmente il costo del cambiamento (in termini di sforzi, tempi, economici)."*

    Per ogni voce di questa domanda era possibile fornire una fra le seguenti risposte: 1) *Completamente d'accordo*; 2) *D'accordo*; 3) *Indifferente*; 4) *In disaccordo*; 5) *Completamente in disaccordo*; 6) *Non so*. Questa domanda è stata posta a tutti i rispondenti e raccoglie affermazioni relative ad alcuni aspetti cruciali della filosofia agile.

14. *"Dia una valutazione a queste affermazioni sulle metodologie di Agile Software Development:"*
    - *"Le metodologie di Agile Software Development funzionano bene per me e il mio team."*
    - *"Coordinamento e collaborazione del team risultano notevolmente migliorate utilizzando metodologie agili."*

- *"Gran parte dei progetti che hanno utilizzato metodologie agili sono progetti di successo."*
- *"Le metodologie di Agile Software Development sono ideali per progetti dei quali si hanno requisiti iniziali incompleti o superficiali."*
- *"Vari progetti che hanno adottato tecniche di Agile Software Development hanno dato risultati deludenti."*
- *"Sviluppare software con un approccio di tipo test-driven porta ad un prodotto di maggior qualità."*
- *"Le metodologie di Agile Software Development richiedono troppe riunioni."*
- *"Le metodologie di Agile Software Development sono difficilmente scalabili a team di sviluppo numerosi (>10 persone)."*
- *"Le metodologie di Agile Software Development non sono adatte allo sviluppo di software di tipo safety-critical."*
- *"I progetti che impiegano metodologie di Agile Software Development soffrono di una scarsa progettazione."*

Per ogni voce di questa domanda era possibile fornire una fra le seguenti risposte: 1) *Completamente d'accordo*; 2) *D'accordo*; 3) *Indifferente*; 4) *In disaccordo*; 5) *Completamente in disaccordo*; 6) *Non so*.

Questa seconda domanda del gruppo *"Valutazioni"* è stata posta solamente ai rispondenti che avessero risposto positivamente alla prima domanda del questionario (*"Conosce le metodologie agili?"*), in quanto richiedeva di valutare affermazioni relative alle metodologie agili. Parte di queste affermazioni è ispirata a quelle usate da Andrew Begel nella sua ricerca a Microsoft [6].

4.5.5 Precedenti esperienze con metodologie agili

Questo gruppo è composto da un unica domanda e ha lo scopo di verificare alcuni dei motivi per cui un compilatore che abbia usato metodologie agili in esperienze passate, non ne faccia utilizzo al momento.

15. *"Nel caso in cui abbia sperimentato metodologie di Agile Software Development in passato, può spiegare perché ora non ne fa più utilizzo?"*
    - *"Non ho mai sperimentato metodologie agili in passato."*
    - *"Ho cambiato team di sviluppo e il nuovo team non usa metodologie agili."*

- *"Vengono adottate pratiche agili senza seguire una particolare metodologia."*
- *"Sono stati effettuati tentativi di introdurre tecniche agili nel team di sviluppo ma sono falliti."*
- *"Difficoltà nel coordinare team agili con altri team."*
- *"I progetti che adottano tecniche agili soffrono di progettazione scadente."*
- *"Difficoltà nell'accettare il cambiamento dei requisiti lungo tutto lo sviluppo del progetto software."*
- *"Il fallimento o le performance scadenti in alcuni progetti hanno portato a considerare le tecniche agili come non valide per l'azienda."*
- *"Altro."*

Al compilatore è stata data la possibilità di scegliere più risposte e di suggerire, usando l'opzione "Altro", ulteriori motivazioni. Anche in questo caso, parte delle risposte è ispirata alla ricerca di Begel [6].

4.5.6 Informazioni personali

Questo gruppo identifica alcune informazione legate al compilatore del questionario. Per motivi di riservatezza e per garantire l'anonimato, non si è tenuta traccia dell'azienda per cui il compilatore lavora, tuttavia si è fatto in modo di avere informazioni generali sulla provincia e la dimensione di tale azienda. La raccolta di questo tipo di informazioni non è strettamente legata alle metodologie agili, ma consente di ottenere alcune informazioni demografiche relative al campione intervistato.

16. *"La ringraziamo per la gentile collaborazione. Le chiediamo infine di terminare l'indagine fornendo alcune informazioni personali ricordandole ancora una volta che l'intera indagine è anonima."*

    Questa non è propriamente una domanda, ma serve a ricordare al compilatore l'anonimato dell'indagine, per favorire l'immissione delle informazioni più riservate.

17. *"Quale ruolo ricopre all'interno dell'azienda?"*

    - *"Sviluppatore"*
    - *"Project Manager"*
    - *"Analista"*

- *"Consulente"*
- *"Altro"*

La domanda propone alcune figure lavorative e aggiunge la possibilità di suggerirne delle altre nel caso in cui le presenti non soddisfino i rispondenti.

18. *"Quanti anni di esperienza possiede nella produzione software?"*
    - *"Meno di 3 anni"*
    - *"Dai 3 ai 6 anni"*
    - *"Più di 6 anni"*

    Stabilisce gli anni di esperienza del compilatore suggerendo gli stessi intervalli utilizzati nella domanda relativa all'esperienza del team di sviluppo.

19. *"Quanti dipendenti possiede l'azienda per cui lavora?"*
    - *"Meno di 10"*
    - *"Da 10 a 49"*
    - *"Da 50 a 249"*
    - *"Da 250 in su"*

    Identifica la dimensione dell'azienda per cui il compilatore lavora. Gli intervalli si allineano ai limiti stabiliti dalla Commissione Europea [32].

20. *"Sede di lavoro: indicare la provincia"*

    Una piccola area di testo consente di inserire la provincia della sede di lavoro.

4.5.7 Considerazioni personali

L'ultimo gruppo si compone di un unica domanda aperta (facoltativa) che chiede all'intervistato di esporre qualsiasi considerazione relativa alle metodologie di *Agile Software Development*.

21. *"Esponga qualsiasi considerazione o osservazione relativa alle metodologie di Agile Software Development."*

La compilazione di questa ultima domanda si presenta come una grande area di testo dove è possibile inserire liberamente pensieri personali o esperienze aggiuntive riguardanti le metodologie di *Agile Software Development*.

Il questionario preliminare è stato sottoposto a cinque soggetti test, ai quali è stato chiesto di valutare la chiarezza delle domande presentate, in modo da prevedere una loro eventuale riformulazione con lo scopo di evitare ogni possibile ambiguità. Inoltre è stato loro chiesto di misurare il tempo impiegato per terminare l'indagine per poi riportarlo in sede di compilazione ufficiale e fornire, agli intervistati, un'indicazione sul tempo necessario per rispondere a tutte le domande.

Nell'Appendice A si può prendere visione di come sono state presentate agli intervistati le schermate relative al questionario on-line.

4.6 RECLUTAMENTO DELLE UNITÀ DEL CAMPIONE

Un totale di 147 potenziali rispondenti è stato contattati nei primi giorni di dicembre 2012 tramite un messaggio diretto, sfruttando e-mail personali e di lavoro. Nei casi in cui questi indirizzi non fossero disponibili, sono stati utilizzati i social network *LinkedIn* e *Facebook*. Il messaggio di reclutamento introduceva le motivazioni dell'indagine assieme a tempo e modalità di compilazione: in particolare si precisava che il sondaggio richiedeva la compilazione di un solo componente del team di sviluppo e si mettevano a conoscenza gli intervistati riguardo al periodo di apertura della survey. Inoltre, per favorire la compilazione, veniva assicurato l'anonimato delle risposte, come suggerito da Oppenheim [25]. In cambio della collaborazione venivano offerti, per chi manifestasse interesse, i dati ottenuti al termine dell'indagine ed, eventualmente, un incontro per la loro esposizione.

Durante il periodo di compilazione, si è saltuariamente effettuato l'accesso alla piattaforma *Lime Survey* per prendere visione dei dati raccolti, e per verificare il tasso di risposta. A due settimane dal reclutamento, è stato inviato un reminder per ricordare agli invitati l'indagine in corso e sollecitare alla compilazione del questionario coloro i quali non lo avessero ancora fatto.

4.7 AGGIUSTAMENTI POST–RACCOLTA

Al termine del periodo di compilazione (dicembre 2012 - gennaio 2013) l'indagine è stata chiusa e, utilizzando le funzionalità fornite dalla piattaforma *Lime Survey*, i dati sono stati esportati in un formato di file che fosse compatibile con i maggiori software di analisi statistica, realiz-

zandone inoltre delle copie di backup. Nel nostro caso è stato utilizzato il software *IBM SPSS v20* per la lettura e l'analisi del database di dati.

Durante questa fase post-indagine si è resa necessaria la modifica del dataset per rendere omogenee alcune delle variabili considerate, in particolare si sono dovute editare quelle opzioni dove veniva data la possibilità al compilatore di rispondere con "Altro". È stata infine apportata una modifica alle risposte relative alla provincia della sede di lavoro (domanda 20 del gruppo *Informazioni personali*) alcune compilazioni presentavano la sigla, mentre altre la provincia per intero (es "VE" e "Venezia"). Si è deciso di uniformare la notazione usando la sigla delle varie province.

4.8 ANALISI DEI RISULTATI

Terminata la fase di aggiustamento dei dati per rendere omogenee alcune variabili, è stato utilizzato il software *IBM SPSS* per l'analisi dei dati raccolti e per la formulazione delle considerazioni. Questa analisi verrà trattata specificatamente nel Capitolo 5.

In questo capitolo abbiamo descritto il processo di costruzione dell'indagine indagine per raccogliere un insieme di informazioni preliminari sulla diffusione di pratiche e metodologie di *Agile Software Development*. Visto il tipo di studio esplorativo che si intendeva condurre, si è scelto un campionamento di tipo non-probabilistico per la scelta del campione. Questo tipo di campionamento ha dei vantaggi quali la facilità con cui viene condotta l'indagine e la velocità di raccolta dei dati, tuttavia questa scelta porta alla selezione di un campione non rappresentativo per la popolazione; di conseguenza le informazioni estratte avranno validità solo per il campione considerato. Nel capitolo che seguirà passeremo all'analisi dei dati raccolti da questa survey e avanzeremo alcune considerazioni su questi.

5 RISULTATI

Abbiamo visto nel Capitolo 3 come le metodologie di *Agile Software Development* abbiano permesso di superare le limitazioni dei processi di sviluppo software derivanti dal modello a cascata e come possano permettere di gestire progetti con requisiti software incerti oppure che possano cambiare durante il progetto. Ma tali tecniche sono effettivamente diffuse nel nostro territorio? Vengono realmente adottate nei team di sviluppo oppure ci si affida ancora a metodi più tradizionali?

Nella prossima sezione verranno analizzati i dati raccolti dalle compilazioni del questionario, descritto nel Capitolo 4, che è stato sottoposto agli intervistati. Date le modalità con cui l'indagine è stata costruita, non sarà possibile effettuare generalizzazioni sui dati, in quanto il campione scelto non è rappresentativo per la popolazione in esame. Sarà tuttavia possibile procedere con una descrizione delle informazioni raccolte limitatamente al campione analizzato che permetterà di ottenere una maggiore comprensione sui temi delle metodologie di *Agile Software Development*, utile per eventuali studi più approfonditi su questo argomento.

5.1 ANALISI DEI DATI

Nel periodo compreso fra dicembre 2012 e gennaio 2013 il questionario "Indagine su Agile Software Development", introdotto nel Capitolo 4, è stato sottoposto ad un totale di 147 persone coinvolte nell'attività di produzione di software a livello professionale. Queste persone sono state scelte fra studenti ed ex-studenti dell'Università degli Studi di Padova e una lista di professionisti gentilmente fornita da alcune fonti. Tutti i potenziali compilatori sono stati contattati direttamente tramite indirizzo email personale, email di lavoro o tramite contatto su social network[1]. Dopo alcune settimane dalla prima richiesta di compilazio-

1 Linked-In, Facebook

ne, è stato inviato un *reminder* per incrementare il tasso di risposta, che risultava inizialmente contenuto.

Al termine della fase di compilazione, il dataset contenente le variabili e i dati relativi all'indagine, è stato esportato con gli opportuni tool messi a disposizione da *LimeSurvey*, in modo da ottenere un formato leggibile dai principali pacchetti software di statistica. Per supportare e facilitare l'attività di analisi, è stato impiegato il software *IBM SPSS Statistics* nella versione 20. Tale applicazione ha permesso di ottenere in modo semplice e intuitivo i dati relativi alle frequenze delle risposte. In modo altrettanto facile è stato possibile concentrare l'analisi su determinati sottogruppi di dati, in base alla configurazione di opportune condizioni sulle variabili. Infine la creazione dei grafici che verranno presentati è stata pressoché immediata e sono disponibili varie possibilità di personalizzazione.

Del campione di 147 persone cui è stato inviato il questionario, 64 persone hanno partecipato all'indagine sulla diffusione delle metodologie di *Agile Software Development*, permettendo in questo modo di ottenere un buon tasso di risposta del 43,5% [24] e raggiungendo la quota di compilazioni suggerita di 50-60 persone [13]. Riguardo ai non rispondenti, si può assumere che questi intervistati non conoscessero o non fossero coinvolti in metodologie di *Agile Software Development*. Ad ogni modo trarremo le nostre conclusioni sulle sole risposte raccolte.

All'interno del questionario il totale di ventuno domande è stato suddiviso in sette gruppi di pertinenza: "conoscenze", "progetto", "pratiche di sviluppo software", "valutazioni", "precedenti esperienze con metodologie agili", "informazioni personali" e "considerazioni personali". Nelle sezioni che seguiranno, analizzeremo le risposte secondo la sequenza di gruppi definiti.

5.1.1 Conoscenze

La prima domanda del questionario poneva un semplice quesito: chiedeva agli intervistati se conoscessero o avessero sentito parlare di metodologie di Agile Software Development. La Tabella 1 ci mostra i risultati di questa domanda. Di 64 rispondenti, 31 (48,4%) affermano di conoscere le metodologie agili.

A coloro i quali hanno risposto positivamente è stato quindi chiesto quali metodologie agili conoscessero, fornendo loro un elenco e la possibilità di suggerire ulteriori metodologie non presenti in lista. Il grafico in Figura 13 presenta i risultati di questa seconda domanda, dove si può chiaramente notare come *Scrum* ed *Extreme Programming* siano più famo-

	Frequenza	Percentuale
Sì	31	48,4
No	33	51,6
Totale	64	100,0

Tabella 1: "Conosce le metodologie di Agile Software Development?"

se con rispettivamente 23 e 27 persone che dichiarano di conoscere tali metodologie. A seguire troviamo *Lean Software Development* con 7, *Feature Driven Development* con 6 e *Adaptive Software Development* con 5. Solo 2 persone conoscono *Dynamic System Development Methods*. Un intervistato ha scelto di aggiungere *Kanban* alla lista di metodologie di *Agile Software Development* conosciute inizialmente proposta.

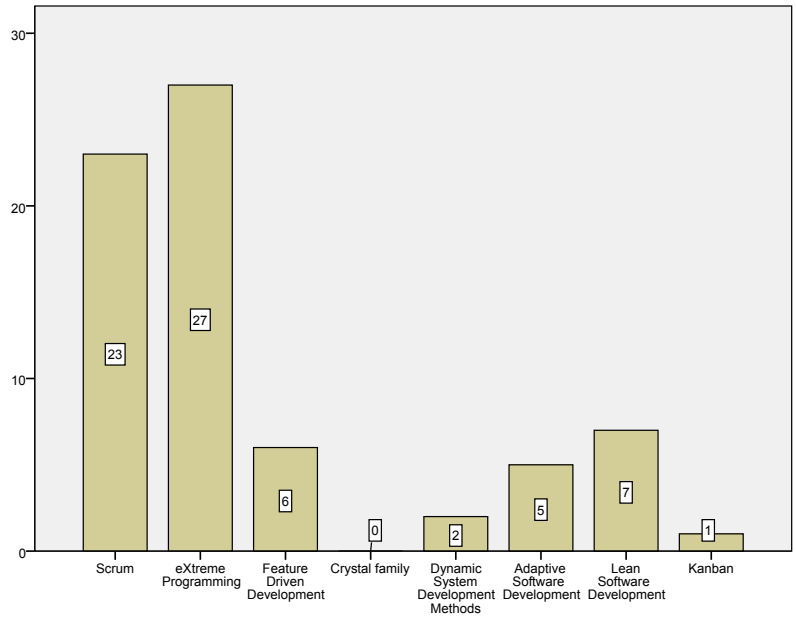


Figura 13: Metodologie agili conosciute

Dopo aver stabilito il numero di persone che conoscono le metodologie agili, si è proseguito chiedendo quali metodologie venissero effettivamente applicate all'interno dei team di sviluppo degli intervistati, al momento dell'indagine. Questa domanda è stata posta solo a coloro i quali avessero affermato di conoscere le metodologie agili in quanto, chi non le conosce, non può dire di applicarle. I risultati sono mostrati nel grafico presente in Figura 14. Circa il 45% dei compilatori che conoscono le metodologie di *Agile Software Development* dichiara di non applicare attualmente alcuna metodologia agile per lo sviluppo software.

Se ora andiamo a considerare quali metodologie vengano effettiva-

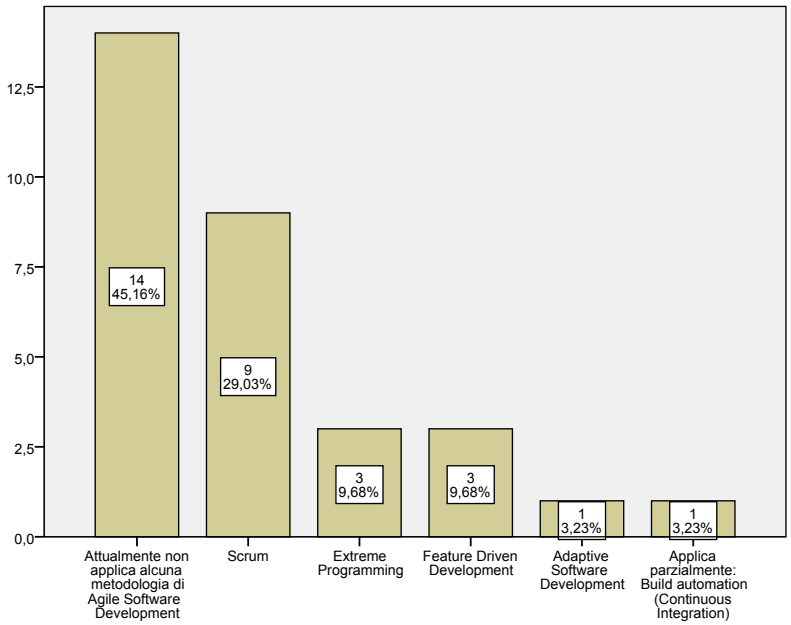


Figura 14: Metodologie agili in uso

mente impiegate, troviamo al primo posto *Scrum* con 9 utilizzatori, seguita da *Extreme Programming* e *Feature Driven Development* con entrambe 3 utilizzatori. In coda troviamo *Adaptive Software Development* con un utilizzatore. Un compilatore ha scelto di inserire nella lista la voce *Build Automation* come metodologia agile impiegata al momento dal suo team di sviluppo. È necessario precisare che più che una metodologia, l'attività di *Build Automation* è considerata un prerequisito per la pratica di *Continuous Integration*[2] ovvero di integrazione continua del codice.

Considerando anche coloro i quali affermano di non conoscere le metodologie agili, ne risulta che del totale di 64 compilatori, il 26,6% (17 individui) segue e adotta attualmente una metodologia agile di sviluppo software assieme ai componenti del relativo team di sviluppo. La più alta percentuale di *Scrum* riflette le parole di Ken Schwaber, uno dei suoi creatori: «[...] e nel 2009, l'86% dei metodi agili di sviluppo era basato su *Scrum*. Questo significa che *Scrum* è superiore? No, significa che *Scrum* è semplice, ben spiegato, e per le persone è facile da comprendere [...]» [18]. Anche il ricercatore Andrew Begel, nella sua ricerca sull'adozione di metodi di *Agile Software Development* in Microsoft, riscontra una maggiore adozione di *Scrum* [6] rispetto alle altre metodologie agili.

2 Per ulteriori informazioni si rimanda a [11][14]

5.1.2 Progetto

In questa sezione verranno presentati i dati relativi al progetto software che il team di sviluppo dell'intervistato seguiva al momento dell'indagine. Questi dati comprendono per esempio il numero di componenti del team, la durata prevista del progetto e l'esperienza del team di sviluppo.

		Usa Agile		
		No	Si	Totale
Qual'è la durata prevista del progetto a cui sta lavorando?	Meno 3 mesi.	11	2	13
		23,4%	11,8%	20,3%
	Da 4 a 6 mesi.	10	7	17
		21,3%	41,2%	26,6%
	Da 7 a 12 mesi.	10	6	16
		21,3%	35,3%	25,0%
	Da 1 a 2 anni.	8	2	10
		17,0%	11,8%	15,6%
	Più di 2 anni.	8	0	8
		17,0%	0,0%	12,5%
Totale		47	17	64
		100,0%	100,0%	100,0%

Tabella 2: Durata del progetto

La Tabella 2 mostra le risposte degli intervistati in relazione alla durata del progetto. Come possiamo vedere, la distribuzione delle risposte di chi dichiara di non seguire alcun approccio agile è sostanzialmente omogenea in tutte le gli intervalli di progetto, aggirandosi attorno al 20% in ogni intervallo, con un picco del 23,5% nel caso di progetti di durata più bassa ("Meno di 3 mesi"). La situazione è leggermente differente se consideriamo ora i rispondenti che seguono un approccio agile: le percentuali risultano più eterogenee con una concentrazione maggiore sulle fasce rispettivamente di "da 4 a 6 mesi" (con il 41,2% dei rispondenti) e "da 7 a 12 mesi" (con il 35,3%). Gli intervalli "Meno di 3 mesi" e "Da 1 a 2 anni" collezionano entrambi l'11,8% mentre non sono stati dichiarati progetti con durata stimata maggiore di 2 anni per team che adottino metodologie agili.

Se consideriamo le percentuali cumulate dei primi tre intervalli, possiamo notare come l'88,3% dei progetti agili sia di durata inferiore ai 12 mesi, mentre tale percentuale risulta essere pari al 66% nel caso dei progetti non agili, pur conservando una maggiore percentuale di progetti estremamente brevi ("Meno di 3 mesi"). Ciò riflette come generalmente le metodologie agili vengono più facilmente impiegate in progetti di

breve durata in quanto, dati i brevi cicli di iterazione, riescono a fornire software funzionante nel giro di poche settimane.

		Usa Agile		
		No	Si	Totale
Da quante persone è composto il team di progetto nel quale sta lavorando?	Meno di 4 persone.	16 34,0%	2 11,8%	18 28,1%
	Da 4 a 9 persone.	25 53,2%	12 70,6%	37 57,8%
	Da 10 a 24 persone.	3 6,4%	3 17,6%	6 9,4%
	Da 25 a 50 persone.	3 6,4%	0 0,0%	3 4,7%
Totale		47 100,0%	17 100,0%	64 100,0%

Tabella 3: Numero componenti del team di sviluppo

Considerando ora la Tabella 3, possiamo vedere la distribuzione del numero dei componenti del team di sviluppo secondo le fasce presentate.

Nessuno dei team agili ha un numero di componenti del team di sviluppo compreso fra 25 e 50 persone, mentre nessuno dei 64 rispondenti ha riportato progetti con team di sviluppo più grandi di 50 persone, scelta che era in origine prevista nella domanda del questionario, ma che non è stata riportata nella Tabella 3 in quanto avrebbe mostrato valori nulli. Considerando i team non agili vediamo un 34% di team con meno di 4 persone e a seguire un 53,2% con un numero di persone nella fascia "Da 4 a 9". Osservando gli stessi intervalli relativi ai team agili, notiamo una minore percentuale, pari a 11,8%, di team con meno di 4 persone con un conseguente aumento della percentuale relativa all'intervallo "Da 4 a 9" che raggiunge il 70,6%. In entrambi i casi, considerando la somma delle percentuali, più dell'80% dei rispondenti (87,2% per i team non agili 82,4% per i team agili) dichiara un team di sviluppo di dimensione inferiore alle 10 persone. Nel 100% dei team agili il numero di componenti non supera le 24 persone, mentre per i team non agili vi è una percentuale di 6,4% relativa a team nella fascia "Da 25 a 50". Si può osservare che, specialmente per i team agili, questi dati si riferiscono a team di sviluppo di grandezza medio/piccola e riflettono i dati indicati da Boehm e Turner riguardanti la dimensione dei team di sviluppo di metodologie agili [10] [9].

Proseguendo con l'analisi delle informazioni riguardanti il progetto, prendiamo in considerazione ora i dati della Tabella 4 la quale rappresenta i livelli di esperienza dei vari team di sviluppo. Si può notare come

le percentuali delle varie fasce nei due gruppi siano molto simili, in particolare nella risposta che indica l'esperienza del team di sviluppo compresa nell'intervallo "Da 3 a 6 anni", con 35,3% e 34% rispettivamente per team agili e non.

Ad ogni modo le percentuali non sono particolarmente discostanti da portare a pensare che siano presenti grandi differenze e, in entrambi i gruppi (agile e non agile), circa il 60% degli intervistati dichiara team di sviluppo con più di 3 anni di esperienza nella produzione di software (rispettivamente 58,8% e 63,8%). Si può notare che vi è una lieve differenza nelle altre fasce: nell'intervallo che identifica il team con in generale meno di 3 anni di esperienza la percentuale nei team agili risulta leggermente più alta (23,5% contro 19,1% dei team non agili); ciò corrisponde ad una maggiore percentuale per i team non agili nella fascia con più di 6 anni di esperienza (29,8% contro 23,5%).

		Usa Agile		
		No	Si	Totale
Qual'è il livello di esperienza nella produzione software all'interno del team?	La maggioranza dei componenti del team ha meno di 3 anni di esperienza.	9 19,1%	4 23,5%	13 20,3%
	La maggioranza dei componenti del team ha dai 3 ai 6 anni di esperienza.	16 34,0%	6 35,3%	22 34,4%
	La maggioranza dei componenti del team ha più di 6 anni di esperienza.	14 29,8%	4 23,5%	18 28,1%
	I livelli di esperienza sono molto vari.	8 17,0%	3 17,6%	11 17,2%
Totale		47 100,0%	17 100,0%	64 100,0%

Tabella 4: Livello di esperienza nel team di sviluppo

Si precisa che la voce indicante che "I livelli di esperienza del team sono molto vari" è stata intenzionalmente inserita come scelta per consentire agli intervistati un'ulteriore opzione, nel caso in cui non si riconoscessero nelle precedenti risposte. In questa fascia, che rappresenta dei team di sviluppo con livelli di esperienza fortemente eterogenei, si riconoscono circa il 17% degli intervistati di entrambe le categorie.

Pur essendo le percentuali delle varie fasce molto simili, ciò che ci si aspettava era un maggiore livello di esperienza associato a team agili, come solitamente risaputo per le metodologie agili, le quali richiedono personale con una certa esperienza, solitamente di tipo senior.

Andremo ora a considerare la Tabella 5, la quale illustra la tipologia di software che i vari team stavano sviluppando al momento dell'indagine.

Le varie tipologie suggerite riguardavano: 1) software B2B a supporto del processo (es. software gestionale, software di controllo e gestione macchinari, ecc.); 2) software B2B a supporto del prodotto (es. software per dispositivi, elettrodomestici, giocattoli, ecc.); 3) software B2C(es. applicazioni di produttività personale, intrattenimento, videogiochi, ecc.); 4) applicazioni per dispositivi mobile e 5) web-site/web-application per aziende.

		Usa Agile		
		No	Si	Totale
Quale tipologia di software sta sviluppando?	Software B2B a supporto del processo(per produzione industriale, controllo/gestione di macchinari, sw gestionale, ecc)	32 68,1%	8 47,1%	40 62,5%
	Software B2B a supporto del prodotto (software per dispositivi, elettrodomestici, giocattoli, ecc.).	1 2,1%	0 0,0%	1 1,6%
	Software B2C (es. applicazioni di produttività personale, intrattenimento, ecc.).	2 4,3%	5 29,4%	7 10,9%
	Applicazione per dispositivi mobile.	1 2,1%	0 0,0%	1 1,6%
	Web site/web application per aziende.	11 23,4%	4 23,5%	15 23,4%
Totale		47 100,0%	17 100,0%	64 100,0%

Tabella 5: Tipologia di software sviluppato

Valutando il totale delle compilazioni, possiamo vedere che le tipologie di software si distribuiscono principalmente su tre delle cinque tipologie suggerite: il 62,5% dei team dei rispondenti si occupa di software B2B a supporto del processo, seguito da un 23,4% che sviluppa prodotti di tipo *web-based*. A seguire troviamo il 10,9% che si occupa di software B2C mentre le due tipologie rimanenti (software B2B a supporto del prodotto e applicazioni per dispositivi mobile) appaiono con una percentuale molto ridotta pari al 1,6% dei compilatori per entrambi. Si nota che in queste due ultime tipologie sono state indicate da due soli team, appartenenti entrambi al gruppo che non usa metodi agili.

Riguardo alle tre tipologie con maggiore percentuale di sviluppo, si può vedere una sostanziale differenza fra i team agili e non, nei casi di software B2B a supporto del processo, dove i team non agili occupano il 68,1% contro il 47,1% dei team agili, e la tipologia B2C, dove i team agili rappresentano il 29,4% contro un molto più ridotto 4,3%.

Rimane pressoché invariata la percentuale di team che sviluppano prodotti *web-based*.

Il prossimo aspetto che verrà considerato riguarda la tipologia di perdita che si può verificare in caso di un malfunzionamento del software che i team sviluppando al momento dell'indagine. L'obiettivo di questa domanda era inoltre valutare la presenza di progetti di tipo *safety-critical* (sistemi a sicurezza critica) nell'indagine [33]. Nell'ambito di questi progetti entrano in gioco fattori importanti quali il rischio di danneggiamento di mezzi e/o macchinari, danni ambientali o gravi danni a persone e, nei casi peggiori, la perdita di vite umane.

		Usa Agile		
		No	Si	Totale
Nell'ipotesi di un malfunzionamento del software che sta sviluppando, che tipo di perdita si può verificare?	Perdita trascurabile.	4	0	4
		8,5%	0,0%	6,2%
	Perdita di informazioni.	17	7	24
		36,2%	41,2%	37,5%
	Perdita economica.	25	10	35
		53,2%	58,8%	54,7%
	Perdita o danneggiamento di mezzi o macchinari.	1	0	1
		2,1%	0,0%	1,6%
Totale		47	17	64
		100,0%	100,0%	100,0%

Tabella 6: Perdita in caso di malfunzionamento

Nella Tabella 6 vediamo i risultati relativi a tutti i progetti. Si può notare sin da subito l'assenza, sia fra i team agili che in quelli non agili, di progetti che possano comportare le perdite più dannose (danni ambientali e danni a persone) mentre è presente, seppur in minima percentuale, un progetto dove un malfunzionamento può causare danneggiamento di macchinari, gestito con un progetto che non applica metodologie agili.

Il 54,7% degli intervistati dichiara di seguire progetti software che possono subire una perdita economica nel caso di errori, seguiti da un 37,5% dove la perdita risulterebbe essere di sole informazioni. Infine abbiamo il 6,2% dei progetti che ricade nella fascia di perdita trascurabile e il rimanente 1,6% che rappresenta l'unico progetto di tipo *safety-critical* dove malfunzionamento può causare danni a mezzi.

Se ora prendiamo in esame i dati relativi ai team che impiegano metodologie agili, notiamo che i progetti sviluppati ricadono in due sole tipologie: il 58,8% presenta una possibile perdita economica mentre il rimanete 41,2% una perdita di informazioni, indicando una tipologia di progetti con perdita medio/bassa.

La prossima questione che verrà analizzata riguarda l'appartenenza o meno del committente del software (o di un suo rappresentante) al team di sviluppo. La Tabella 7 ci mostra che solo il 20,3% degli intervistati afferma che il cliente fa parte del team di sviluppo mentre una grande percentuale, pari al 67,2%, dichiara che il committente del software non fa parte del team di sviluppo, tuttavia risulta facilmente rintracciabile in caso di chiarimenti richiesti per la prosecuzione del progetto. Il 9,4% dei compilatori riporta che il cliente non è presente nel team di sviluppo e inoltre ci sono difficoltà nel mettersi in contatto per spiegazione.

In questa domanda è stata inserita la possibilità di inserire la risposta "Non applicabile" in modo da permettere agli intervistati di rispondere, nel caso in cui le scelte precedenti non fossero indicate. In questa categoria possono cadere i casi in cui non esiste un vero e proprio committente (es. un software creato per il mercato di massa o un'applicazione per dispositivi mobile). Ad ogni modo solo il 3,1% (corrispondente a 2 intervistati) effettua questa scelta.

		Usa Agile		
		No	Si	Totale
Il committente del software (o un suo rappresentante) fa parte del team di sviluppo?	Si.	12 25,5%	1 5,9%	13 20,3%
	No, ma è facilmente reperibile per spiegazioni o chiarimenti.	30 63,8%	13 76,5%	43 67,2%
	No e risulta difficile mettersi in contatto con lui.	3 6,4%	3 17,6%	6 9,4%
	Non applicabile.	2 4,3%	0 0,0%	2 3,1%
Totale		47 100,0%	17 100,0%	64 100,0%

Tabella 7: Committente appartenente al team di sviluppo

Valutando i risultati per i team agili, possiamo vedere come sia sensibilmente diminuita la percentuale di risposte che dichiarano che il cliente fa parte del team di sviluppo mentre aumenta la percentuale di intervistati che dichiarano una ottima disponibilità del cliente a chiarire dubbi (76,5%). Abbiamo infine un 17,6% di risposte che segnalano una difficile comunicazione col cliente, contro il 6,4% dei team non agili.

I dati mostrati nella Tabella 7 rappresentano una situazione inaspettata, in quanto uno dei princìpi portanti delle metodologie di *Agile Software Development*, è la stretta collaborazione fra team di sviluppo e il committente del software. Inoltre varie metodologie agili prevedono che il committente sia parte integrante del team di sviluppo.

L'ultimo quesito appartenente al gruppo delle domande relative al progetto è stato posto per stabilire la distribuzione geografica dei componenti del team di sviluppo.

		Usa Agile		
		No	Si	Totale
Scelga la risposta che descrive meglio il suo caso: "I vari componenti del mio team di sviluppo si trovano "	Componenti nella stessa stanza.	28 59,6%	10 58,8%	38 59,4%
	Componenti distribuiti nello stesso edificio.	4 8,5%	2 11,8%	6 9,4%
	Componenti distribuiti in più sedi.	14 29,8%	4 23,5%	18 28,1%
	Componenti distribuiti in più Paesi.	1 2,1%	1 5,9%	2 3,1%
Totale		47 100,0%	17 100,0%	64 100,0%

Tabella 8: Distribuzione geografica dei componenti del team di sviluppo

Se osserviamo la Tabella 8 possiamo vedere come i valori in percentuale siano molto simili, con una gran parte di quasi il 60% che afferma che i componenti del team di sviluppo si ubicano tutti nella stessa stanza.

Un interessante punto da notare riguarda il fatto che, indipendentemente dalla metodologia utilizzata, i team risultano nella maggior parte dei casi ubicati nella stessa stanza e circa il 70% di entrambi i gruppi riporta che i componenti del team di sviluppo si trovano almeno nello stesso edificio. Ciò è particolarmente importante e rappresenta uno dei principi agili (co-location) che favorisce la comunicazione diretta all'interno del team.

5.1.3 Pratiche di sviluppo software

In questo gruppo di domande è stato fornito ai partecipanti all'indagine un elenco di pratiche di sviluppo software che stanno alla base delle metodologie di *Agile Software Development* ed è stato loro chiesto di indicare in quale misura venissero adottate tali pratiche. Per ognuna delle pratiche suggerite, gli intervistati potevano indicare una fra le seguenti valutazioni: "Sistematicamente", "Spesso", "A volte", "Mai", "Non applicabile" e "Non conosco".

Le prime quattro voci indicano il livello di adozione di una data pratica mentre la quinta, ovvero "Non applicabile", è stata inserita per permettere di dare una risposta nei casi in cui le precedenti non avessero senso nel contesto del rispondente. È stata infine aggiunta l'opzione

"Non conosco" per dare all'intervistato la possibilità di rispondere nel caso in cui non conoscesse una data pratica.

La Figura 15 riporta un grafico relativo alle risposte date dai team di sviluppo che dichiarassero di non adottare metodologie di *Agile Software Development*, mentre la Figura 16 riporta i livelli di adozione delle stesse pratiche riferite però ai team agili.

Considerando le percentuali di risposta di "sistematicamente" e "spesso", ci si accorge che gran parte di queste pratiche sono largamente impiegate nella produzione software indipendentemente dal fatto che il team di sviluppo adotti o meno una metodologia di *Agile Software Development*, con valori percentuali superiori al 50%. Le differenze più evidenti fra i due gruppi riguardano la pratica di "ritmo sostenibile" che paradossalmente risulta seguita con più frequenza da parte dei team non agili e quella riguardante le "riunioni giornaliere del team di sviluppo", meno praticata nel caso dei team non agili.

La pratica di *pair programming* risulta in entrambi i gruppi la meno utilizzata; inoltre più del 25% dei team non agili dichiara di non conoscerla. Ciò può essere dovuto alle varie controversie relative al suo utilizzo: in un articolo del 2009, il ricercatore della *University of Texas* VenuGopal Balijepally e i suoi colleghi dimostrano che le performance ottenute a lavorare in coppia tipicamente non superano quelle del miglior membro nell'ipotesi che lavori individualmente [3]. Risulta strana anche la percentuale di persone che dichiarano di non conoscere la pratica di "co-location" nei team agili, essendo questa pratica condivisa e incoraggiata da molte metodologie di *Agile Software Development*. Ad ogni modo in entrambi i gruppi viene usata in circa il 70% dei casi "sistematicamente" e "spesso" e conferma i risultati presentati nella Tabella 8 riguardanti la distribuzione geografica dei componenti del team di sviluppo.

Anche nel caso di "coinvolgimento attivo del cliente" si può trovare conferma di queste percentuali, confrontando i dati della Tabella 7 relativa alla presenza del committente nel team di sviluppo (se consideriamo il cliente come "appartenente al team di sviluppo" o "facilmente reperibile").

In entrambe le tabelle gli alti livelli dichiarati di "co-location del team" favoriscono di conseguenza un'alta percentuale di "comunicazione faccia-to-face", in quanto i vari componenti del team si ubicano nella stessa stanza e possono comunicare più agevolmente. Ne segue una migliore condivisione delle informazioni relative al progetto.

Volendo includere nel livello di adozione anche coloro che dichiarano di usare le suddette pratiche "a volte", si raggiunge facilmente l'80% di adozione, sia per i team agili, sia quelli non agili. Ciò può portare a pen-

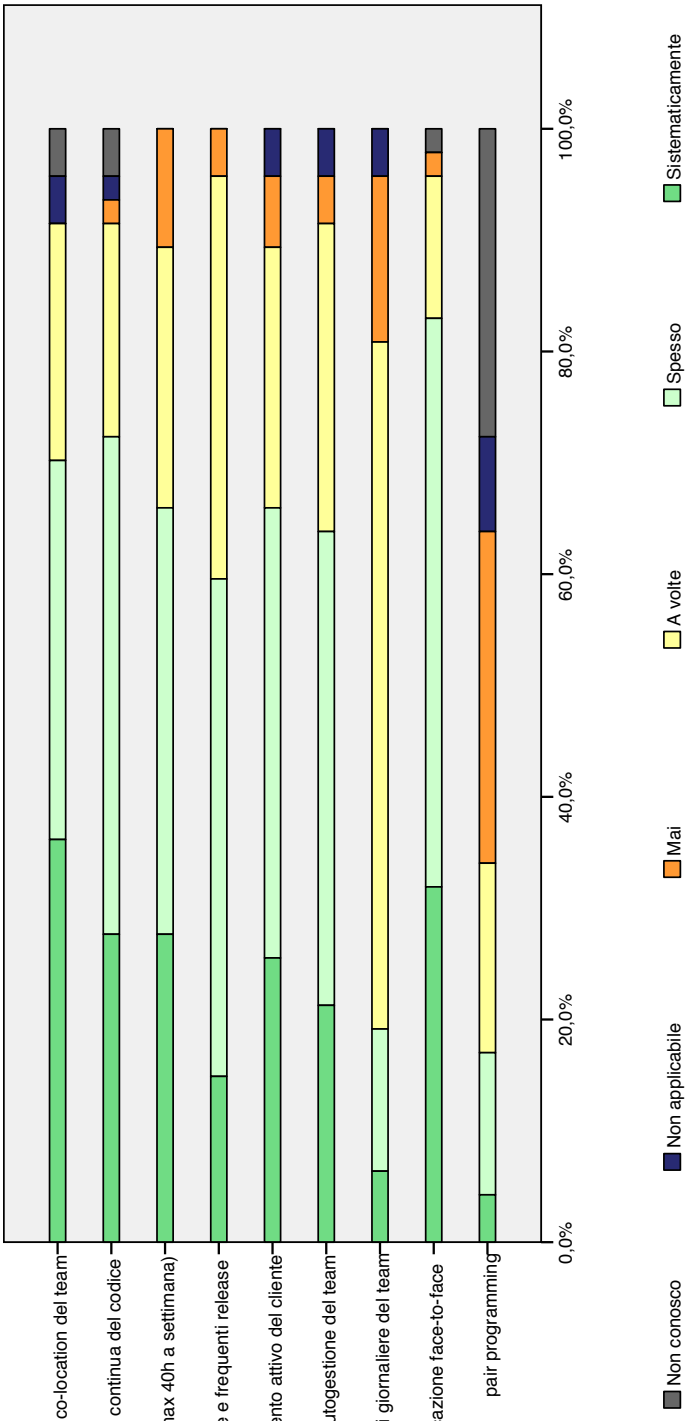


Figura 15: Pratiche di sviluppo software parte 1 (team non agili)

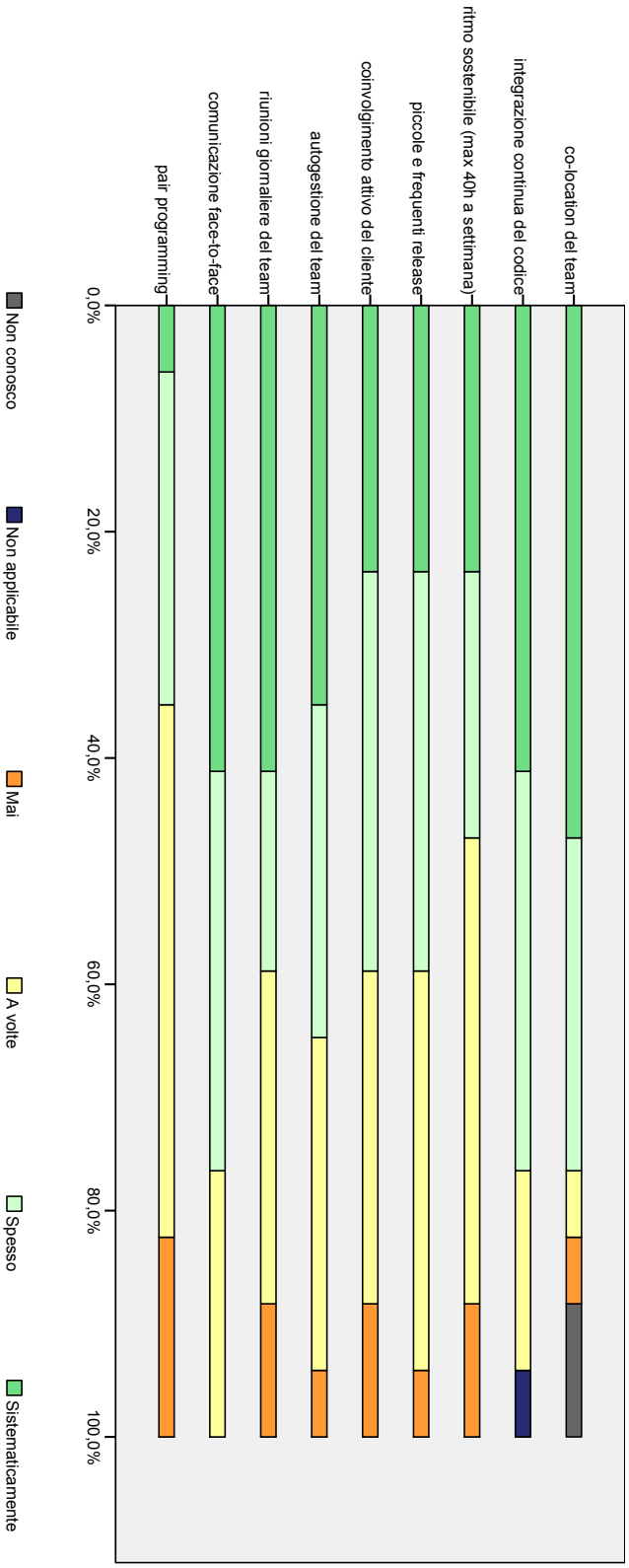


Figura 16: Pratiche di sviluppo software parte 1 (team agili)

sare che l'insieme di pratiche suggerite siano state incluse nel processo di produzione software come *best-practise*, che portano vantaggi e favoriscono questo tipo di attività in modo completamente indipendente dalla tipologia di metodologia usata.

Le pratiche presenti in Figura 17 sono più specifiche rispetto alle precedenti, e per questo è stato deciso di porre la relativa domanda solo a chi dichiarasse di avere familiarità con le metodologie di *Agile Software Development*. Questo tipo di pratiche risultano essere sostanzialmente meno diffuse e soprattutto meno conosciute, come dimostrato dalle più alte percentuali di "Mai" e "Non conosco" mostrati in figura. La pratica di *System metaphor*, risulta essere la meno adottata, con poco più del 15% di rispondenti che la utilizzano usualmente. I motivi potrebbero essere legati alle problematiche relative a questa pratica, fra cui la possibilità che i componenti del team di sviluppo non possiedano lo stesso livello di familiarità di altri, o che la metafora sia troppo debole e renda difficile rappresentare ragionevolmente il sistema [39].

In generale questo insieme di pratiche supera il 30% di adozione se consideriamo i livelli più alti di utilizzo ("sistematicamente" e "spesso"), con picchi che si avvicinano il 50% per le pratiche di *team coding standard* e di *product backlog*.

5.1.4 Valutazioni

Questa sezione di domande chiedeva agli intervistati di esprimere una valutazione su alcune affermazioni riguardanti l'attività di sviluppo software. In questo caso le opzioni di risposta disponibili per ogni domanda erano le seguenti: "Completamente d'accordo", "D'accordo", "Indifferente", "In disaccordo", "Completamente in disaccordo" e "Non so".

La Figura 18 mostra i risultati relativi ai team che non fanno uso di metodologie di *Agile Software Development*, mentre la Figura 19 illustra le risposte ottenute dai team che dichiarano di usare metodi agili.

L'80% dei rispondenti appartenenti a team non agili è d'accordo con l'affermazione che dichiara che vi è una diffusa diffidenza fra gli sviluppatori riguardo l'accogliere modifiche a stadi avanzati dello sviluppo software, contro il 70% dichiarato da chi invece fa parte di team agili. Questa affermazione rappresenta il fatto che la maggior parte degli sviluppatori temono il "cambiamento", il quale porta scompiglio e caos all'interno dei processi di sviluppo software più tradizionali.

Riguardo la capacità di auto-organizzazione del team di sviluppo, i

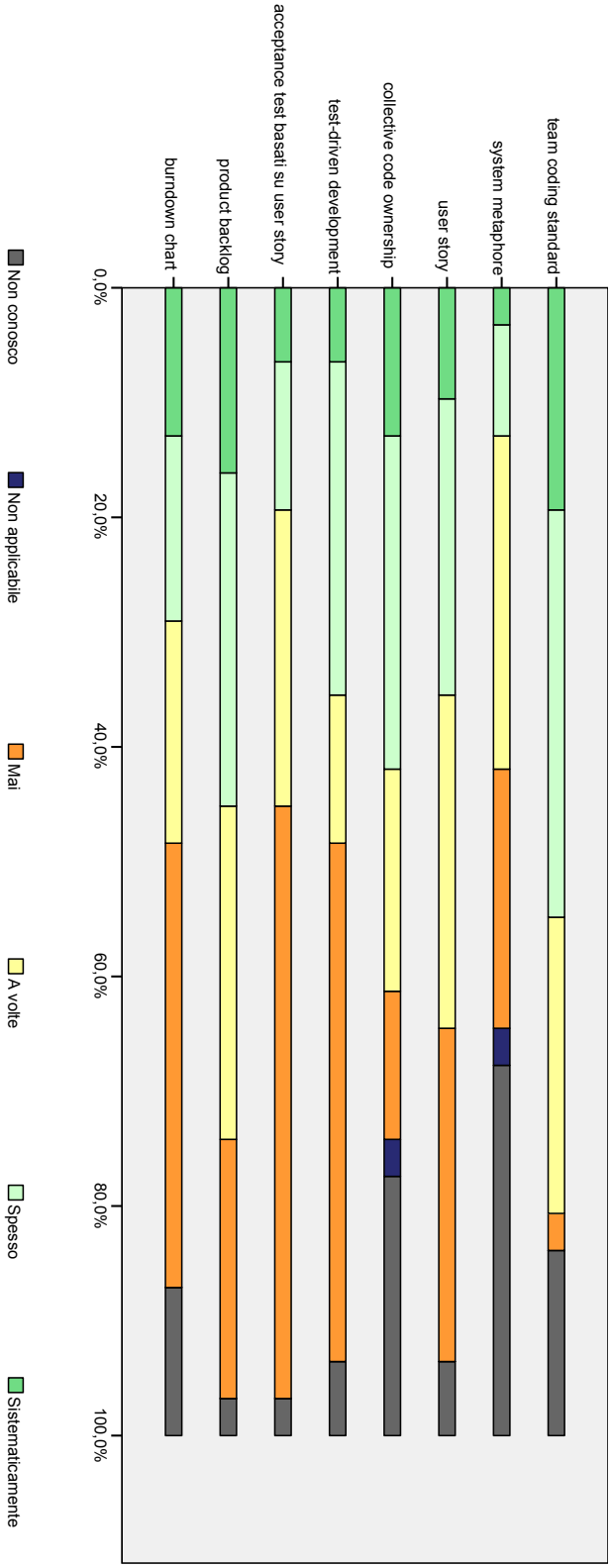


Figura 17: Pratiche di sviluppo software parte 2

due gruppi sono generalmente concordi, con quasi il 90% dei team non agili e ben il valore massimo di 100% dei team agili. In particolare più del 50% di quest'ultimi si dichiarano completamente d'accordo con questa affermazione.

Uno dei dodici principi agili afferma che il cambiamento dei requisiti anche a stadi di sviluppo avanzati, caratteristica delle metodologie di *Agile Software Development*, si trasforma in vantaggio competitivo per il committente. Questa idea è condivisa da circa il 35% dei team non agili e dal 55% dei team agili.

L'ultima affermazione di questo gruppo riguardava l'aumento esponeziale del costo del cambiamento di requisiti a stadi avanzati di sviluppo software in termini di sforzi, costi, tempi di sviluppo: è ciò che Kent Beck chiama *cost of change*. Secondo Beck con gli opportuni linguaggi e le corrette pratiche di sviluppo, è possibile appiattire la curva di "costo del cambiamento" [4]. Quasi il 90% dei rispondenti di team non agili si dichiara d'accordo con l'affermazione riportata nel questionario, mentre la percentuale per gli intervistati dei team agili che sono d'accordo con questa affermazione si riduce al 65%. Ciò significa che nel gruppo di team agili, pur conservando un'alta percentuale di intervistati che si trovano d'accordo, è presente l'idea della riduzione del "costo del cambiamento" nella produzione di software con metodologie di *Agile Software Development*. Ne è prova anche il maggiore numero di intervistati che esprimono il disaccordo, che da un 5% dei team non agili passa al 30% dei team agili.

Un secondo gruppo di affermazioni è stato sottoposto a tutti i compilatori che avessero manifestato una conoscenza delle metodologie di *Agile Software Development*. Anche in questo caso i rispondenti dovevano selezionare se si trovassero d'accordo o meno con le frasi proposte. Prendendo visione della Figura 20, più del 60% dei rispondenti sono d'accordo con l'affermare che le metodologie di *Agile Software Development* funzionano bene per l'intervistato e per il suo team di sviluppo e che la loro adozione permette di migliorare coordinamento e collaborazione del gruppo. Un picco dell'85% dei rispondenti a questa domanda è d'accordo con l'indicare le medologie di *Agile Software Development* adatte a gestire progetti con requisiti incompleti o superficiali. Per queste affermazioni non è stata riportata nessuna scelta che manifestasse disaccordo. A seguire troviamo un 70% di rispondenti che si trova d'accordo nell'affermare che l'approccio allo sviluppo *test-driven*, pratica intrinseca di *Extreme Programming* ma usata anche in altre metodologie, porta ad un prodotto di maggiore qualità.

Circa il 35% è d'accordo nell'asserire che gran parte dei progetti che

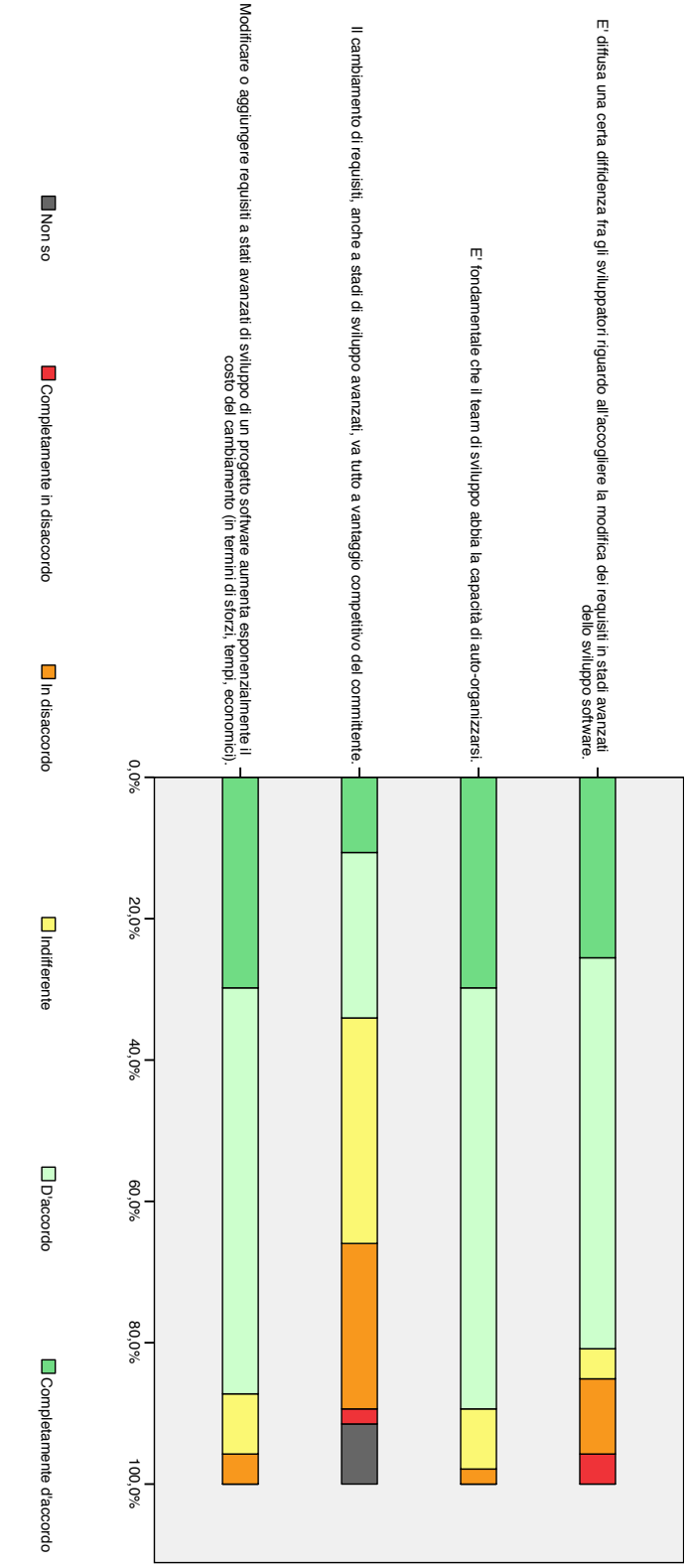


Figura 18: Valutazioni generali - team non agili

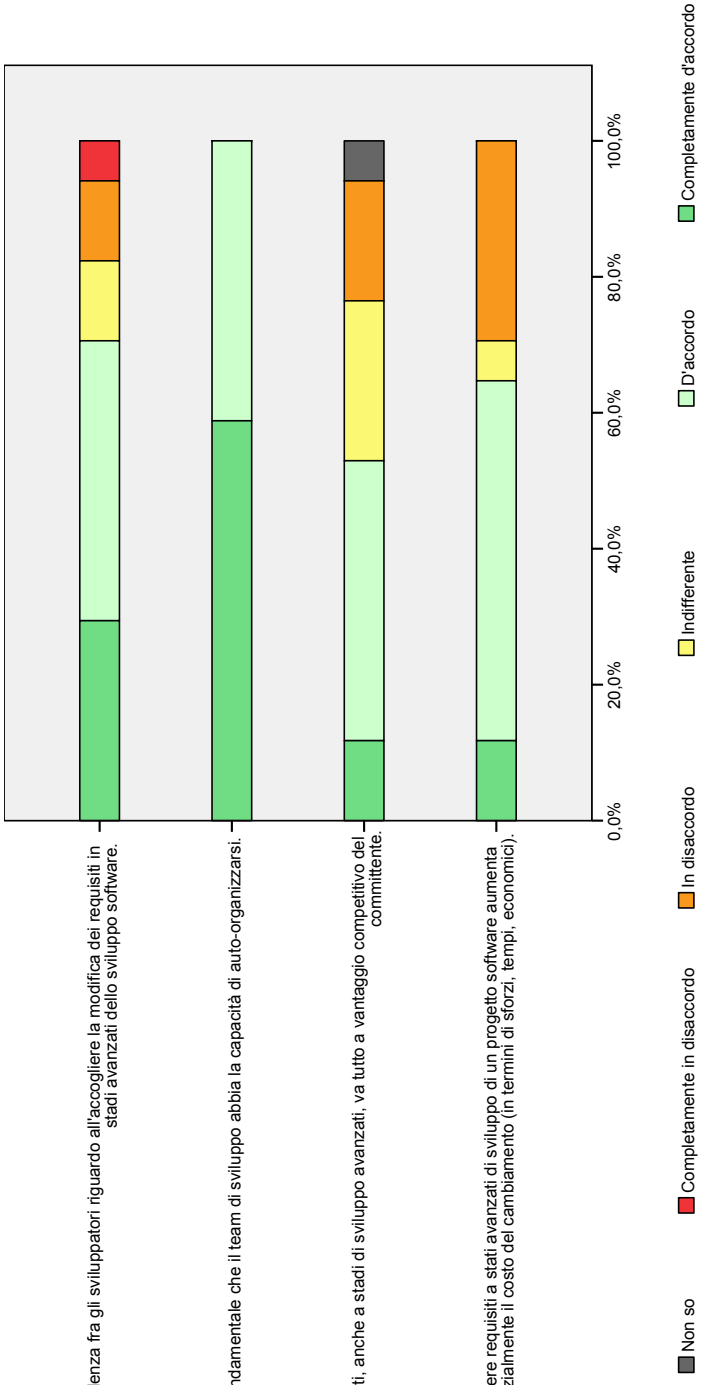


Figura 19: Valutazioni generali - team agili

hanno impiegato metodologie agili, sono stati progetti di successo, mentre il 15% dei rispondenti è in disaccordo. Questa affermazione viene confermata dalla sua speculare la quale riporta che vari progetti sviluppati con metodologie agili hanno dato risultati deludenti: il 10% di chi ha completato questa domanda si trova d'accordo, mentre circa il 45% si dichiara in disaccordo. I rimanenti si dichiarano indifferenti o non sanno rispondere.

Circa il 25% dei rispondenti a questa domanda si dichiara d'accordo col fatto che le metodologie di *Agile Software Development* richiedano troppe riunioni, tuttavia quasi il 50% dissente, mentre circa il 20% si dichiara indifferente. Per il 35% degli intervistati le metodologie agili non sono facilmente scalabili a team di sviluppo con più di 10 persone contro una percentuale del 25% di chi si trova in disaccordo. Riguardo la non compatibilità delle metodologie agili con sistemi di tipo *safety-critical*, vediamo quasi il 30% dei rispondenti che dichiara di non saper rispondere, un secondo 30% di rispondenti che si trovano in disaccordo e infine circa il 25% che si dice d'accordo.

Infine la diffusa concezione che le metodologie agili soffrano di scarsa progettazione viene smentita da un 60% di rispondenti che si dichiarano in disaccordo (con un 30% completamente in disaccordo), mentre meno del 15% si trova d'accordo; i rimanenti sono indifferenti o non sanno rispondere.

5.1.5 Precedenti esperienze con metodologie di Agile Software Development

Con questo gruppo di domande si sono volute valutare alcune considerazioni su quali fossero i motivi per cui chi avesse avuto esperienze con metodologie agili in passato, non le adottasse al momento dell'indagine. Le risposte suggerite riprendono alcuni aspetti problematici relativi alle metodologie agili riscontrati nello studio di Begel [6]. La domanda è stata posta a tutti coloro i quali avessero affermato di conoscere le metodologie agili.

In Figura 21 vengono presentate le frequenze di risposta. È necessario precisare che queste opzioni di scelta non sono mutualmente esclusive, in quanto al compilatore era permesso effettuare più di una risposta, in quanto potevano sussistere varie motivazioni.

Possiamo vedere chiaramente che un buon numero di persone dichiara di adottare pratiche agili senza seguire una particolare metodologia (10 casi), seguito poi da 8 intervistati che affermano di aver cambiato

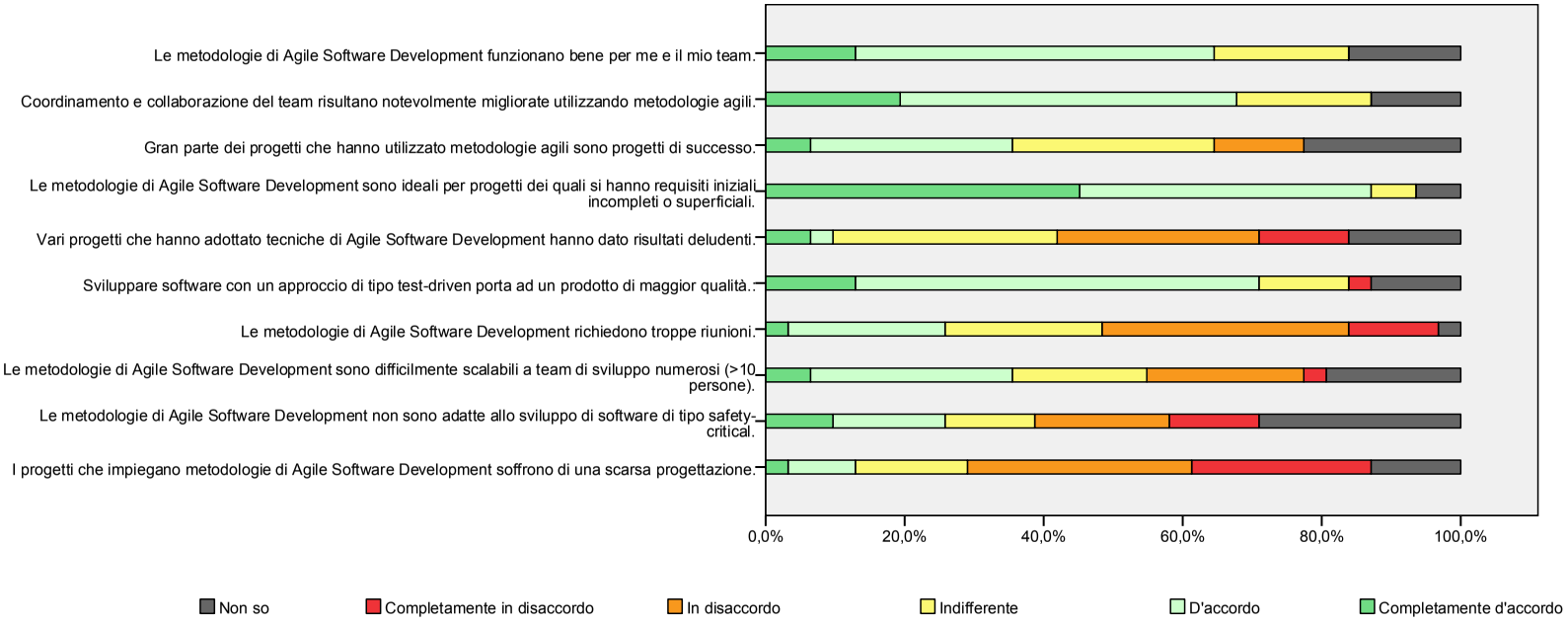


Figura 20: Valutazioni met. agili

Figura 21: "Se adottava, perché ora non usa?"

team di sviluppo e che il team attuale non utilizza metodologie di *Agile Software Development*.

Proseguendo con l'analisi, vediamo che 3 rispondenti testimoniano il tentativo di introdurre metodologie agili nel team di sviluppo, purtroppo con esito negativo. Le ultime tre opinioni selezionate con una preferenza riportano problematiche riguardanti: 1) "la difficoltà di coordinare team agili con gli altri"; 2) "la presunta progettazione scadente dei progetti che fanno uso di tecniche di *Agile Software Development*" e 3) "la difficoltà di accettare cambiamenti nei requisiti lungo tutto lo sviluppo del progetto software"

Si noti che nessuno ha scelto di compilare "fallimenti e/o performance scadenti in alcuni progetti hanno portato considerare le metodologie di *Agile Software Development* come non valide per l'azienda".

Anche in questo caso si è scelto di prevedere la possibilità di inserire ulteriori opzioni in modo che l'intervistato potesse aggiungere opinioni particolari non presenti in lista. Ad ogni modo nessun compilatore ha scelto di aggiungere ulteriori casi che hanno portato all'abbandono di metodi agili.

5.1.6 Informazioni personali

In questa parte dell'indagine sono state poste ai partecipanti domande di carattere più personale, quali la posizione lavorativa, l'esperienza nella produzione software, la dimensione dell'azienda per cui lavorano e la provincia della sede di lavoro. Queste informazioni non hanno una diretta relazione con l'utilizzo o meno di metodologie di *Agile Software Development*, ma sono state raccolte per una pura ragione descrittiva del campione preso in esame.

	Frequenza	Percentuale
Sviluppatore	39	60,9
Project Manager	10	15,6
Analista	10	15,6
Consulente	2	3,1
Team Leader	1	1,6
Solution Architect	1	1,6
Progettista	1	1,6
Totale	64	100,0

Tabella 9: Ruolo all'interno dell'azienda

Nella Tabella 9 vengono mostrati i ruoli di tutti gli intervistati all'interno delle rispettive aziende. Questa domanda prevedeva inizialmente quattro possibili risposte: "Sviluppatore", "Project Manager", "Analista" e "Consulente". Oltre a questi ruoli di default, è stata introdotta la possibilità di dichiarare il proprio ruolo, nel caso un rispondente non si riconoscesse in quelli proposti. Ciò ha comportato l'introduzione di tre nuovi ruoli da parte degli intervistati: "Team Leader", "Solution Architect" e "Progettista".

La Tabella 10 riporta gli anni di esperienza nella produzione software dell'intervistato, mentre la Tabella 11 riporta le informazione relative alla grandezza dell'azienda per cui i vari intervistati lavorano. Infine nell'ultima tabella presentata, Tabella 12, vengono indicate le varie province relative alle sedi di lavoro dei compilatori.

5.1.7 Considerazioni personali

Questo gruppo racchiude un'unica domanda aperta facoltativa dove veniva chiesto di esprimere liberamente qualunque considerazione riguardante il tema delle metodologie di *Agile Software Development*. Qui sotto vengono riportate le risposte date dagli 8 compilatori che hanno completato tale domanda.

1. «Chi le critica, lo fa perché non le conosce, o perché non ha voglia di mettersi a studiare ed imparare. Lui o il suo team.»

2. «Il cliente diretto attualmente è in una situazione in cui per il progetto in atto, ha conoscenze parziale o dedotte di come dovrà lavorare. Questo comporta durante la fase di System-Test/Certificazione frequenti interventi di sistemazione che non possono essere pianificati in quanto gestiti in emergenza. Inoltre i tempi sono talmente stretti che lo sviluppo è ancora parziale a System-Test/Certificazione iniziata. I test vengono effettuati parzialmente in collaborazione con il cliente che promuove/obbliga la messa in produzione accollandosi il rischio delle verifiche parziali. Tra una fase e l'altra del progetto i tempi di "quiete" sono sofferti da una visibilità che sfiora le 2-3 giornate rendendo impraticabile una pianificazione/organizzazione a medio termine. In tali contesti i metodi Agile sono di fatto più efficaci dei processi a cascata che divengono per il contesto sopra non efficacissimi.»

3. «Le metodologie agili hanno aiutato a costruire team di persone motivate e fortemente coinvolte. La tipologia di comunicazione

	Frequenza	Percentuale
Meno di 3 anni.	18	28,1
Dai 3 ai 6 anni.	22	34,4
Più di 6 anni.	24	37,5
Totale	64	100,0

Tabella 10: Esperienza nella produzione software dei rispondenti

	Frequenza	Percentuale
meno di 10.	6	9,4
da 10 a 49.	5	7,8
da 50 a 249.	33	51,6
da 250 in su.	20	31,3
Totale	64	100,0

Tabella 11: Numero di dipendenti dell'azienda dei rispondenti

	Frequenza	Percentuale
BT	1	1,6
MI	3	4,7
NA	1	1,6
PA	5	7,8
PD	12	18,8
RM	1	1,6
TO	2	3,1
TV	8	12,5
VE	9	14,1
VR	22	34,4
Totale	64	100,0

Tabella 12: Provincia delle sedi di lavoro dei rispondenti

prevista da questo tipo di metodologie ha consentito di abbattere le barriere di comunicazione che spesso sono presenti a causa del diverso inquadramento dei singoli all'interno dell'organizzazione (ruolo istituzionale). Ho sperimentato che attraverso le metodologie agili, si favorisce il passaggio di competenze tra i membri del team. La capacita di adattamento come atteggiamento ha permesso in molti casi di anticipare e risolvere criticità che si sarebbero manifestate solo nelle fasi finali di progetto. Ritengo che grazie all'attenzione all'eccellenza tecnica e alla semplicità della soluzione (principi agili) si possa raggiungere soluzioni tecnologiche di alta qualità. Ho riscontrato nel mio ambito di lavoro una certa difficoltà a far percepire il valore di queste metodologie al committente. sia da punto di vista dell'organizzazione di team sia dal punto di vista contrattuale (contratti agili).»

4. «Aspetto positivo: ogni membro del team si sente parte del team; non subisce la gerarchia ma è protagonista nello sviluppo e nelle scelte tecnologiche. Favorisce l'integrazione tra i vari membri del team e favorisce intercambiabilità del team, e quindi problematiche relative ad assenze possono essere superabili.»

5. «Secondo la mia esperienza di 1 anno e mezzo di lavoro nell'azienda e 3 team differenti, ho potuto vedere che di per sè le metodologie agili possono essere utili, soprattutto se vengono applicate in modo informale e sono sentite dal team. Se vengono imposte dall'alto in modo rigido rischiano di essere controproducenti, soprattutto se all'interno del team è presente un committente o un responsabile che costringe il team all'applicazione rigida. Viene cioè meno la vera autogestione del team. Inoltre vanno applicati solo aspetti che tornano utili come i daily meeting e non altri come la stime di carico che possono portare all'ansia da prestazione soprattutto per gli apprendisti. In generale comunque per quello che ho visto le metodologie agili rischiano troppo spesso di venire utilizzate come scusa per analisi mancanti, incomplete o con dei clamorosi buchi funzionali, del tipo "inizia a fare qualcosa e poi si vedrà". Se applicate in questo modo si arriva inevitabilmente ad un software scadente, con forte rischio di bug e di difficile manutenzione. Nell'attuale progetto, che si occupa di effettuare correzioni ad un progetto già in produzione, è difficile usare alcuni elementi di pianificazione a medio/lungo termine, ma troviamo molto comodi le riunioni giornaliere fatte anche in modo informa-

le, le relazioni faccia a faccia ed i rilasci abbastanza frequenti di correzioni e migliorie.»

6. «Ho personalmente caldeggiato ed introdotto negli ultimi 3 anni l'utilizzo di tecnologie di sviluppo agile nel team di sviluppo ed in particolare l'utilizzo di strumenti ALM (Continuous Integration, Issue tracking etc.). Nel nostro ambito lavorativo tuttavia questo tipo di approccio non è attuabile al 100% perché in generale i contratti con i committenti fissano a priori tempi e costi limitandone l'applicazione. Sicuramente l'utilizzo di tecnologie Agili prevede che anche il contratto col committente sia di tipo Agile.»

7. «È un approccio alla gestione dei progetti software che richiede prima di qualsiasi altra cosa un cambiamento sia nella cultura aziendale (che può promuovere queste tecniche) sia nella cultura delle singole persone. Esperienze isolate all'interno di un'azienda, viste magari come eccentricità da tollerare se portano risultati e eliminare al primo insuccesso, non rappresentano un vero cambiamento dell'azienda stessa, e possono portare, dopo una ventata di entusiasmo iniziale, a delusione e quindi ad un giudizio critico in realtà non motivato sulle metodologie agili.»

8. «Sovente questa metodologia di programmazione necessita di personale in grado di applicarlo con cognizione di causa. Condizione necessaria per non ottenere un insuccesso. A fronte della sua semplicità teorica non sempre corrisponde un altrettanto semplice fattibilità.»

Dall'analisi di queste considerazioni sono emersi diversi aspetti importanti. Vi è la conferma che negli ambienti odierni, caratterizzati da una forte incertezza è impossibile riuscire a pianificare a medio/lungo termine e ciò richiede quindi la capacità di gestire questo tipo di incertezza. Per fare ciò, vengono in aiuto le metodologie agili che tuttavia per funzionare correttamente hanno bisogno di vari prerequisiti: l'azienda, ma soprattutto il personale che dovrà adoperarle, dovranno essere in grado di gestire questo tipo di approcci, e dovranno sentirsi coinvolti e motivati nell'adottare tali metodi. Anche da parte del committente è necessaria l'adozione di principi e filosofie agili, specie nella redazione dei contratti con l'azienda sviluppatrice. Clienti che, abituati a metodi più tradizionali, stabiliscono limiti, costi e tempi rendono impensabile qualsiasi tipo di approccio agile per la realizzazione dei progetti commissionati.

Un fatto importante che viene sottolineato nelle considerazioni, risulta essere che queste metodologie innovative non devono essere viste

come scusa per analisi superficiali, mancanti o incomplete: il team che adotta questi approcci deve avere un'adeguata esperienza e competenza nell'applicarle, pena l'insuccesso.

Riguardo agli aspetti positivi, viene testimoniato che il particolare livello di comunicazione di queste metodologie favorisce il passaggio di competenze fra componenti del team di sviluppo e permette di superare facilmente situazioni di assenze di membri del team di sviluppo. Inoltre, sotto il punto di vista tecnico, viene osservato che se i principi agili di semplicità e ricerca dell'eccellenza vengono rispettati, la garanzia è un prodotto finale di alta qualità.

Un ulteriore aspetto emerso da queste considerazioni prevede che la vera utilità di approcci agili si manifesti se questi vengono applicati in modo informale, facendo uso solo di ciò che veramente serve ed è utile al team di sviluppo (vengono citati *daily meeting*, rilasci frequenti e comunicazione *face-to-face*). L'imposizione dall'alto di un'approccio piuttosto di un altro si contrappone al principio di autogestione del team e quindi impedisce di raggiungere l'agilità ricercata.

La filosofia di "usare solo ciò che serve" può essere dimostrata anche dal buon numero di rispondenti che in Figura 21, dichiara di utilizzare singole pratiche agili senza seguire una particolare metodologia. Ne sono ulteriore conferma le alte percentuali di utilizzo di pratiche agili, da parte di team che si dichiarano non agili (Figura 15): pur non adottando una precisa metodologia agile, viene utilizzato un set di semplici *best-practise* poiché è ciò che risulta sufficiente per lavorare in maniera efficiente e proficua. Questo potrebbe essere inoltre la dimostrazione del fatto per cui le pratiche di Figura 17 siano meno diffuse, pur considerando che a tale risposta hanno partecipato rispondenti che conoscono le metodologie di *Agile Software Development*: utilizzare pratiche più specifiche che richiedono un impegno maggiore da parte del team di sviluppatori, oppure metodologie che vengono imposte dall'alto da un manager o dall'azienda, potrebbero far si che manchi il coinvolgimento da parte dei singoli sviluppatori e rendere in questo modo inefficiente una metodologia agile.

In questo capitolo abbiamo discusso i dati raccolti dal questionario "Indagine su *Agile Software Development*" che è stato sottoposto ad un campione di professionisti coinvolti nella produzione di software. Date le modalità descritte nel Capitolo 4 con cui l'indagine è stata condot-

ta, non è stato possibile generalizzare le considerazioni che sono state avanzate, ma ci si è limitati ad un'analisi di tipo descrittivo per il campione in questione. Le informazione raccolte e discusse in questa sede, potranno comunque fornire un punto di partenza per ulteriori studi sull'argomento delle metodologie di *Agile Software Development*.

6 CONCLUSIONI

L'obiettivo che ci si era proposti in questo elaborato era la realizzazione di uno studio esplorativo che potesse fornire un insieme preliminare di informazioni sulla diffusione di metodologie e pratiche di *Agile Software Development* all'interno di un campione di professionisti nello sviluppo software, con lo scopo di valutare l'adozione di queste pratiche e permettesse inoltre di stabilire se queste metodologie abbiano influenzato anche il modo di lavorare di chi segue approcci più tradizionali.

Con lo scopo di raccogliere le informazioni necessarie su queste metodologie, è stata realizzata una survey utilizzando un questionario online basato sulla piattaforma open-source *Lime Survey*, la quale permette in maniera facile e veloce la creazione di sondaggi. Un aspetto critico dell'indagine costruita, riguarda la scelta del campione da analizzare: a favore di una maggior semplicità e velocità nella realizzazione dell'indagine si è scelto un campionamento di tipo non-probabilistico di convenienza e ragionato. Questo fatto impedisce di effettuare generalizzazioni o inferenze sui dati raccolti, in quanto il campione così creato non risulta rappresentativo per la popolazione di sviluppatori presa in esame. In questo caso l'analisi condotta ha validità solo per il campione considerato.

Il questionario è stato inviato ad un totale di 147 contatti, scelti in modo arbitrario, dei quali si conosceva il loro coinvolgimento nell'attività di sviluppo software. Di questi, 64 hanno fornito le risposte che hanno costituito il campione in esame. L'analisi dei dati relativi alla survey realizzata è stata supportata dall'uso del software statistico *IBM SPSS*, con il quale è stato possibile ottenere in maniera semplice, informazioni sulle frequenze delle varie risposte, suddivise per le tipologie di casi raccolti (conoscenza o meno di metodologie di *Agile Software Development*, adozione di metodologie, ecc.).

Dall'analisi dei dati raccolti emerge che nel campione di 64 rispondenti preso in esame, 31 individui (48,4% sul totale) dichiarano di conoscere

le metodologie di *Agile Software Development* (con *Extreme Programming* e *Scrum* come metodologie più conosciute) e, di questi, 17 rispondenti le usano effettivamente per la produzione software (26,6% sul totale). La metodologia più adottata risulta essere *Scrum*, come confermato da varie ricerche. Ciò che si può notare è che generalmente le singole pratiche di sviluppo software discusse nell'indagine (considerate agili) riportano alti livelli di adozione indipendentemente dal fatto che il team di sviluppo dichiari di adottare o meno una metodologia di *Agile Software Development*. Ciò porta a pensare che vi siano situazioni in cui vengano utilizzate delle pratiche di buon senso da parte degli sviluppatori, che permettano di svolgere l'attività di produzione software in maniera efficiente e proficua e che quindi siano per natura ben diffuse e adottate. Queste pratiche corrispondono a quelle che gli autori delle metodologie agili dichiarano *best-practise* e che essi integrano nei processi di sviluppo da loro ideati. In questo modo gli sviluppatori fanno uso di un framework di tecniche che vengono usate continuamente, senza però adottare una precisa metodologia agile. Ne è conferma il fatto che un buon numero di intervistati dichiara esplicitamente di adottare pratiche agili senza seguire una metodologia agile in particolare.

Alcuni fatti inaspettati riguardano casi di pratiche agili o situazioni in cui sorprendentemente chi dichiarasse di non seguire alcuna metodologia agile rappresentasse maggiormente la filosofia agile (cliente incluso nel team di sviluppo, ritmo sostenibile, alto livello di esperienza del team di sviluppo). Ad ogni modo queste discrepanze risultano essere di entità molto limitata e possono essere fortemente soggette alla numerosità del campione.

È stato inoltre rilevato che alcune pratiche agili più specifiche, risultano meno diffuse all'interno degli stessi team agili. Ciò trova riscontro nelle considerazioni finali suggerite da alcuni intervistati dove viene affermato che, per esperienza, la vera utilità degli approcci agili si manifesta con l'adozione delle pratiche che risultano utili per il team di sviluppo. Metodi imposti forzatamente dall'alto da un manager o dall'azienda potrebbero ridurre il coinvolgimento dei singoli sviluppatori e rendere poco efficiente la metodologia agile impiegata.

In un'altra considerazione degli intervistati, viene rimarcato il fatto che negli ambienti odierni, caratterizzati da veloci ritmi di produzione, alta incertezza e dove il cambiamento dei requisiti è all'ordine del giorno, non è possibile avanzare pianificazioni a medio/lungo termine. Questo rende i tradizionali approcci di tipo *plan-driven* inefficaci, e richiede l'adozione di metodi che possano accogliere e gestire questo concetto del "cambiamento", ovvero dei metodi agili. È inoltre richiesto che la filo-

sofia agile abbracci gli sviluppatori quanto i committenti: il tradizionale approccio di definire data di consegna e costo, difficilmente può funzionare in questi contesti ed è necessario che anche i contratti con i clienti siano agili.

POSSIBILI SVILUPPI FUTURI

Un interessante e possibile sviluppo futuro di questo elaborato potrebbe essere la conduzione di uno studio che permetta l'analisi della diffusione delle metodologie di *Agile Software Development* prendendo in esame un campione rappresentativo della popolazione degli sviluppatori software, in modo da poter generalizzare i risultati ottenuti. Si potrebbe per esempio realizzare uno studio che preveda come popolazione tutti i professionisti nello sviluppo software in Italia. Ciò necessita di un accurato piano di campionamento che garantisca una corretta scelta del campione rappresentativo su cui effettuare l'analisi. Questo probabilmente comporterà la raccolta di tutti i nominativi di aziende italiane operanti nell'ambito della produzione software, per poi passare a sceglierne un numero adeguato, possibilmente suddiviso per regione o addirittura per comune. Tali aziende dovranno poi essere invitate a partecipare all'indagine e fornire i nominativi delle unità campionarie considerate.

Un ulteriore punto di approfondimento potrebbe essere lo studio sul campo, cioè in azienda, dell'applicazione di una metodologia di *Agile Software Development* in particolare. L'attività potrebbe comprendere la misurazione del livello di adozione della metodologia presa in esame all'interno del progetto, per capire se vi è l'applicazione rigorosa oppure esistono alcuni gradi di adozione e quindi dimostrare che viene utilizzato ciò che realmente risulta utile al team di sviluppo. Sarebbe interessante in questo caso poter considerare aspetti non trattati in questo elaborato, come le effettive entità di costi e tempi di produzione di progetti sviluppati con metodologie agili e considerare quali tipologie di contratti vengano redatti fra committente e l'azienda sviluppatrice ("contratti agili").

Un ulteriore sviluppo potrebbe essere uno studio comparativo di due progetti simili, che però utilizzano metodologie diverse durante lo sviluppo software (due metodologie agili o una metodologia agile e una più tradizionale), per valutare come varino caratteristiche quali tempi di sviluppo, costi, comunicazione con il committente e gestione del progetto.

A APPENDICE

In questo appendice verranno mostrate le schermate relative al questionario online che è stato presentato agli intervistati. Le domande sono state suddivise in sette gruppi principali e corrispondono a sette differenti pagine della survey:

- Conoscenze
- Progetto
- Pratiche di sviluppo software
- Valutazioni
- Precedenti esperienze con metodologie agili
- Informazioni personali
- Considerazioni

La piattaforma di questionari online *Lime survey* ha permesso l'inclusione di condizioni per la presentazione delle domande. Nel nostro caso, ad un compilatore che dichiarasse di non conoscere le metodologie di *Agile Software Development*, le successive domande che presupponessero la conoscenza di tali metodologie, non sarebbero state mostrate[1].

1 In particolare per chi ha dichiarato di non conoscere le metodologie agili, sono state nascoste le domande 2, 3, 12, 14 e 15

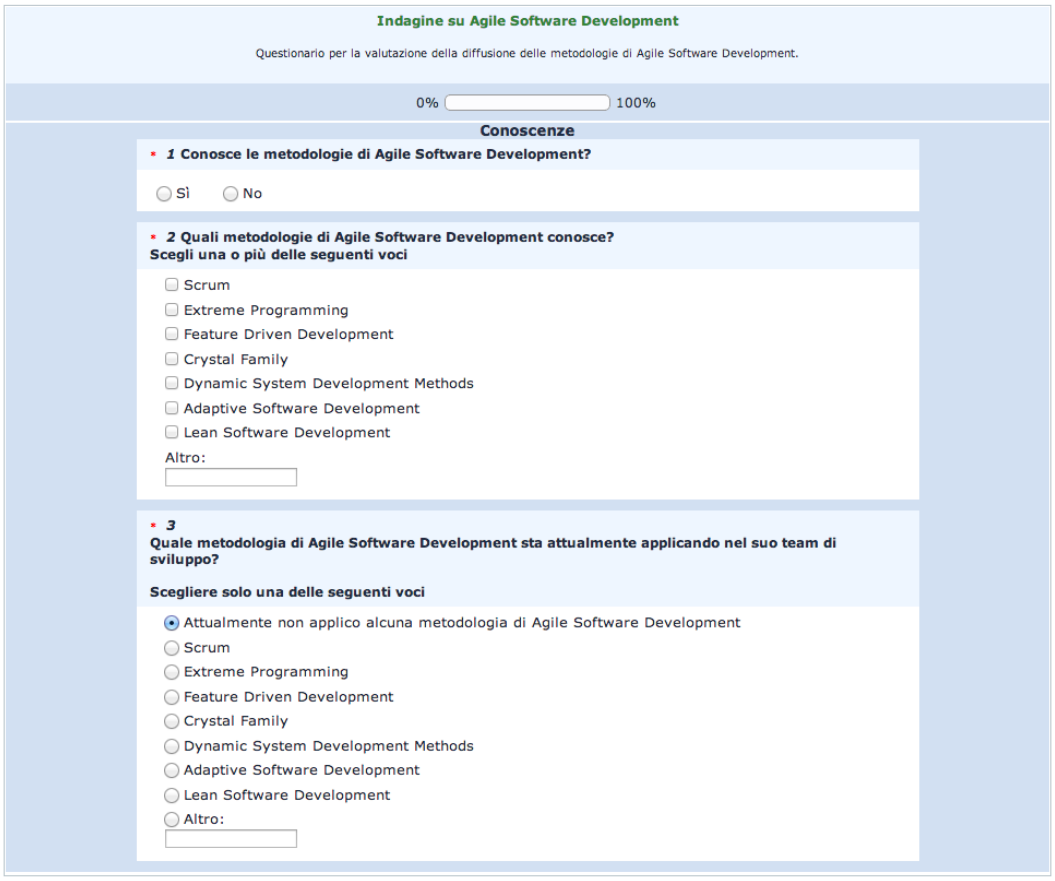
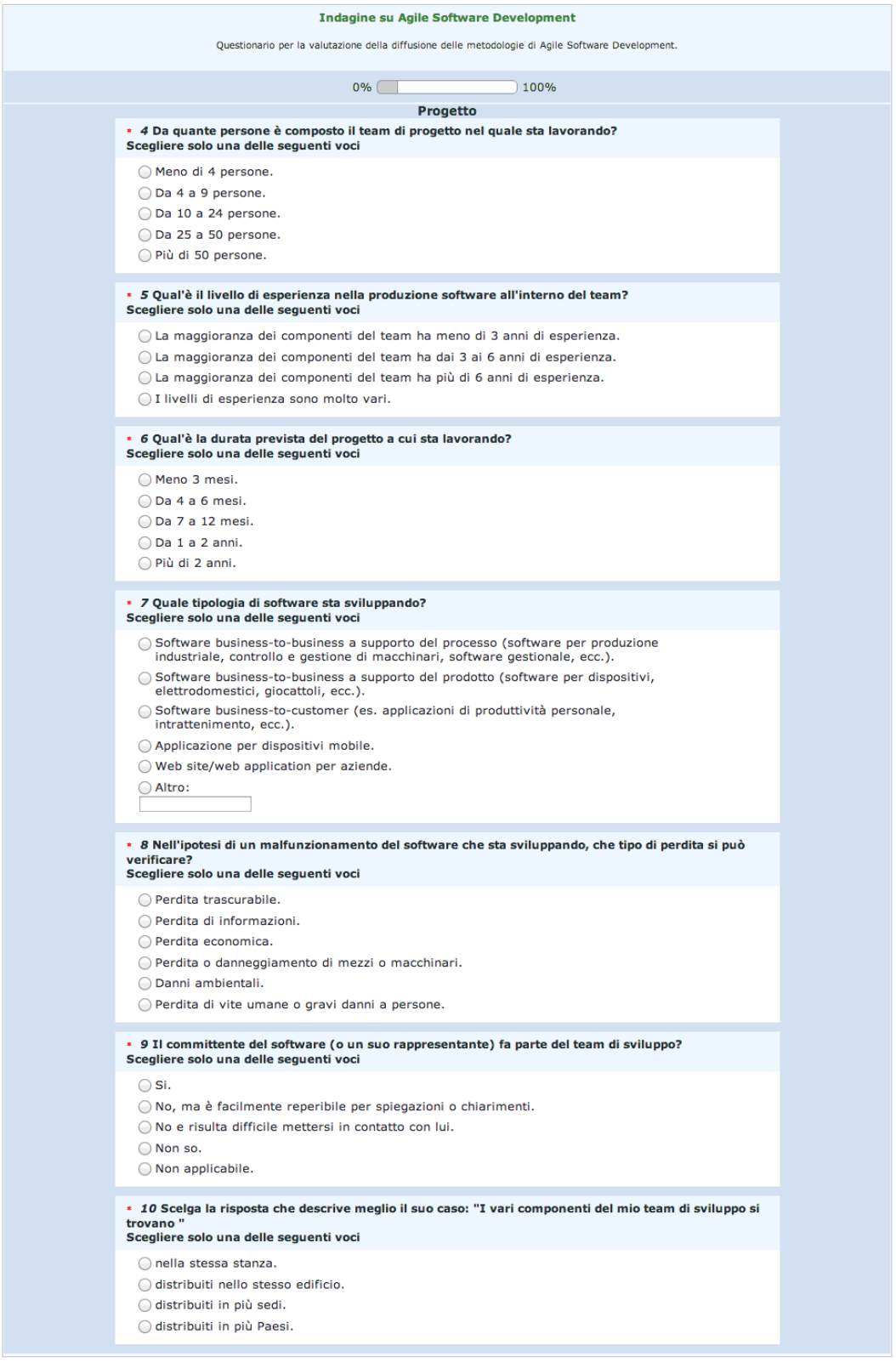


Figura 22: Questionario: gruppo "conoscenze"



Indagine su Agile Software Development

Questionario per la valutazione della diffusione delle metodologie di Agile Software Development.

0% 100%

Progetto

* ***4*** **Da quante persone è composto il team di progetto nel quale sta lavorando?**
Scegliere solo una delle seguenti voci

- Meno di 4 persone.
- Da 4 a 9 persone.
- Da 10 a 24 persone.
- Da 25 a 50 persone.
- Più di 50 persone.

* ***5*** **Qual'è il livello di esperienza nella produzione software all'interno del team?**
Scegliere solo una delle seguenti voci

- La maggioranza dei componenti del team ha meno di 3 anni di esperienza.
- La maggioranza dei componenti del team ha dai 3 ai 6 anni di esperienza.
- La maggioranza dei componenti del team ha più di 6 anni di esperienza.
- I livelli di esperienza sono molto vari.

* ***6*** **Qual'è la durata prevista del progetto a cui sta lavorando?**
Scegliere solo una delle seguenti voci

- Meno 3 mesi.
- Da 4 a 6 mesi.
- Da 7 a 12 mesi.
- Da 1 a 2 anni.
- Più di 2 anni.

* ***7*** **Quale tipologia di software sta sviluppando?**
Scegliere solo una delle seguenti voci

- Software business-to-business a supporto del processo (software per produzione industriale, controllo e gestione di macchinari, software gestionale, ecc.).
- Software business-to-business a supporto del prodotto (software per dispositivi, elettrodomestici, giocattoli, ecc.).
- Software business-to-customer (es. applicazioni di produttività personale, intrattenimento, ecc.).
- Applicazione per dispositivi mobile.
- Web site/web application per aziende.
- Altro:

* ***8*** **Nell'ipotesi di un malfunzionamento del software che sta sviluppando, che tipo di perdita si può verificare?**
Scegliere solo una delle seguenti voci

- Perdita trascurabile.
- Perdita di informazioni.
- Perdita economica.
- Perdita o danneggiamento di mezzi o macchinari.
- Danni ambientali.
- Perdita di vite umane o gravi danni a persone.

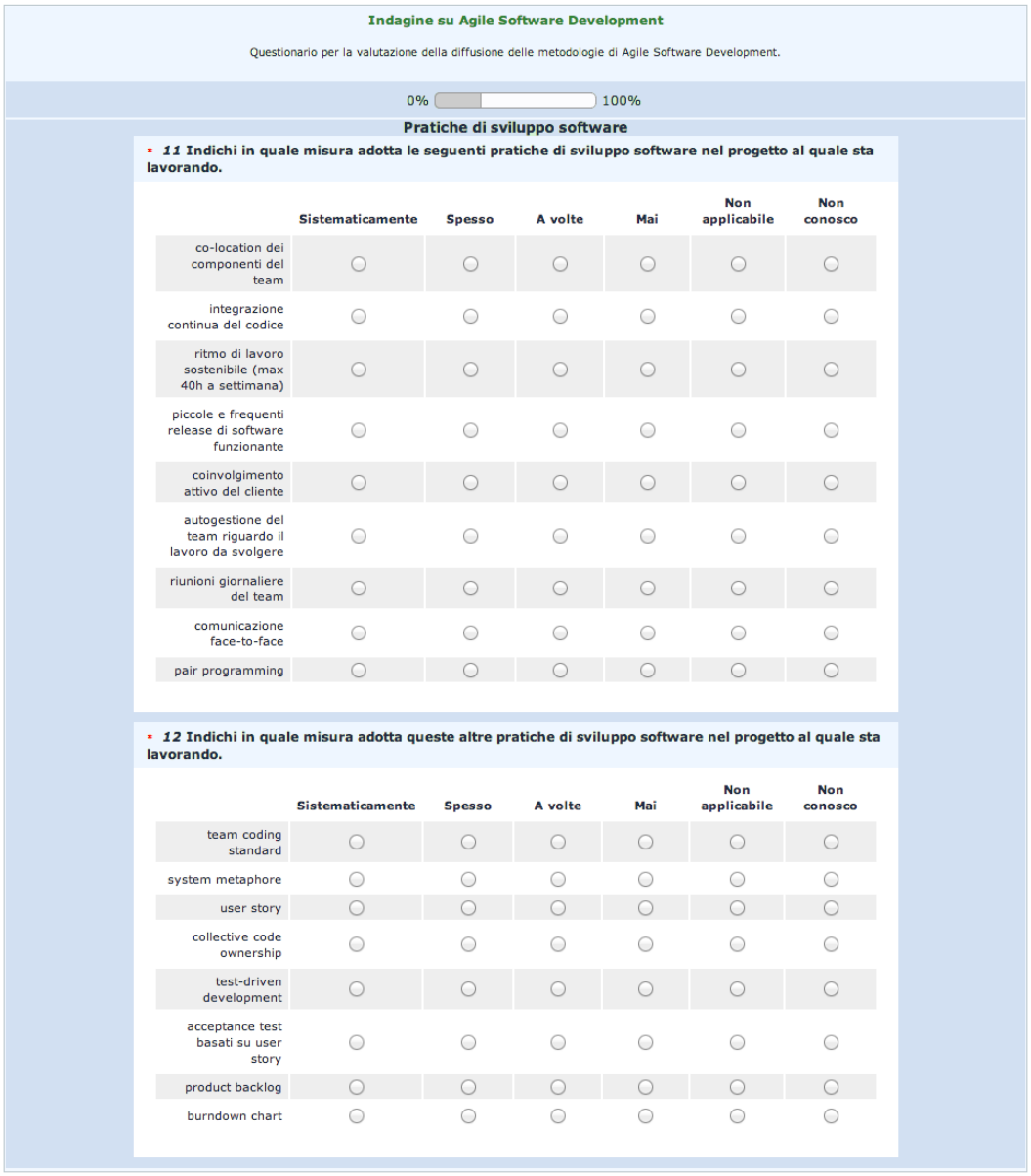
* ***9*** **Il committente del software (o un suo rappresentante) fa parte del team di sviluppo?**
Scegliere solo una delle seguenti voci

- Si.
- No, ma è facilmente reperibile per spiegazioni o chiarimenti.
- No e risulta difficile mettersi in contatto con lui.
- Non so.
- Non applicabile.

* ***10*** **Scelga la risposta che descrive meglio il suo caso: "I vari componenti del mio team di sviluppo si trovano "**
Scegliere solo una delle seguenti voci

- nella stessa stanza.
- distribuiti nello stesso edificio.
- distribuiti in più sedi.
- distribuiti in più Paesi.

Figura 23: Questionario: gruppo “progetto”



Indagine su Agile Software Development

Questionario per la valutazione della diffusione delle metodologie di Agile Software Development.

0% 100%

Pratiche di sviluppo software

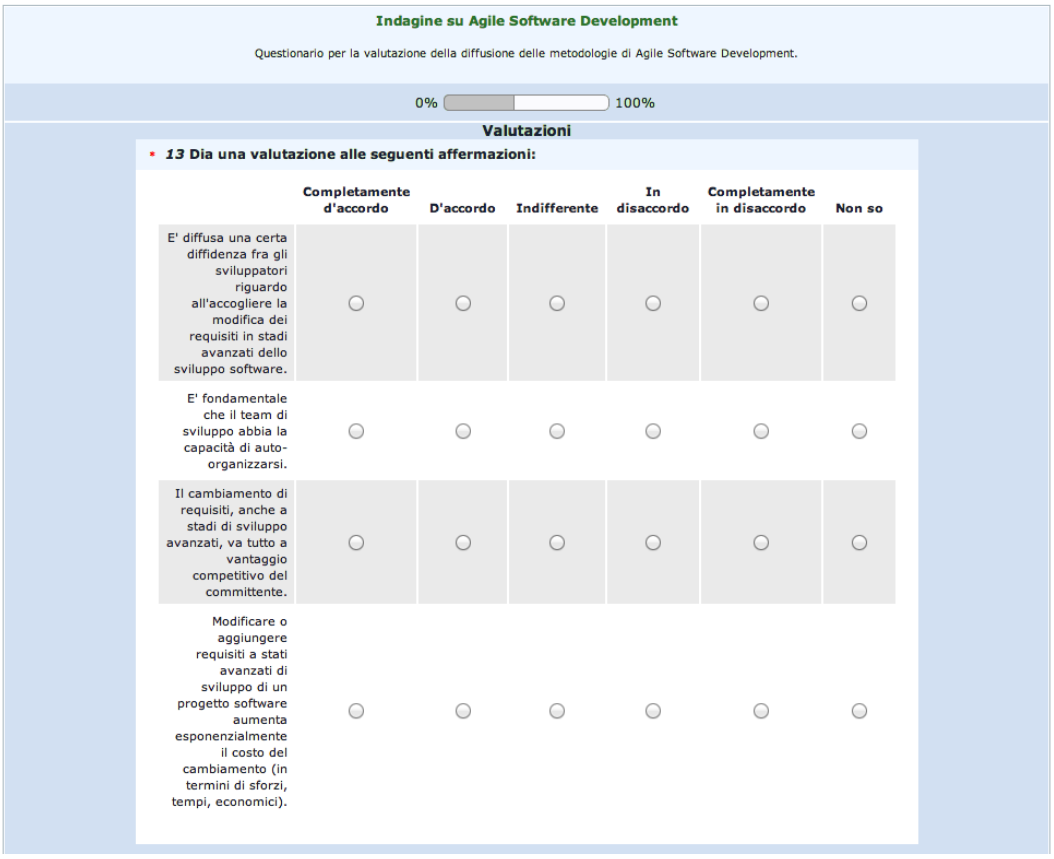
* ***11*** **Indichi in quale misura adotta le seguenti pratiche di sviluppo software nel progetto al quale sta lavorando.**

	Sistematicamente	Spesso	A volte	Mai	Non applicabile	Non conosco
co-location dei componenti del team	○	○	○	○	○	○
integrazione continua del codice	○	○	○	○	○	○
ritmo di lavoro sostenibile (max 40h a settimana)	○	○	○	○	○	○
piccole e frequenti release di software funzionante	○	○	○	○	○	○
coinvolgimento attivo del cliente	○	○	○	○	○	○
autogestione del team riguardo il lavoro da svolgere	○	○	○	○	○	○
riunioni giornaliere del team	○	○	○	○	○	○
comunicazione face-to-face	○	○	○	○	○	○
pair programming	○	○	○	○	○	○

* ***12*** **Indichi in quale misura adotta queste altre pratiche di sviluppo software nel progetto al quale sta lavorando.**

	Sistematicamente	Spesso	A volte	Mai	Non applicabile	Non conosco
team coding standard	○	○	○	○	○	○
system metaphore	○	○	○	○	○	○
user story	○	○	○	○	○	○
collective code ownership	○	○	○	○	○	○
test-driven development	○	○	○	○	○	○
acceptance test basati su user story	○	○	○	○	○	○
product backlog	○	○	○	○	○	○
burndown chart	○	○	○	○	○	○

Figura 24: Questionario: gruppo “pratiche di sviluppo software”



Indagine su Agile Software Development

Questionario per la valutazione della diffusione delle metodologie di Agile Software Development.

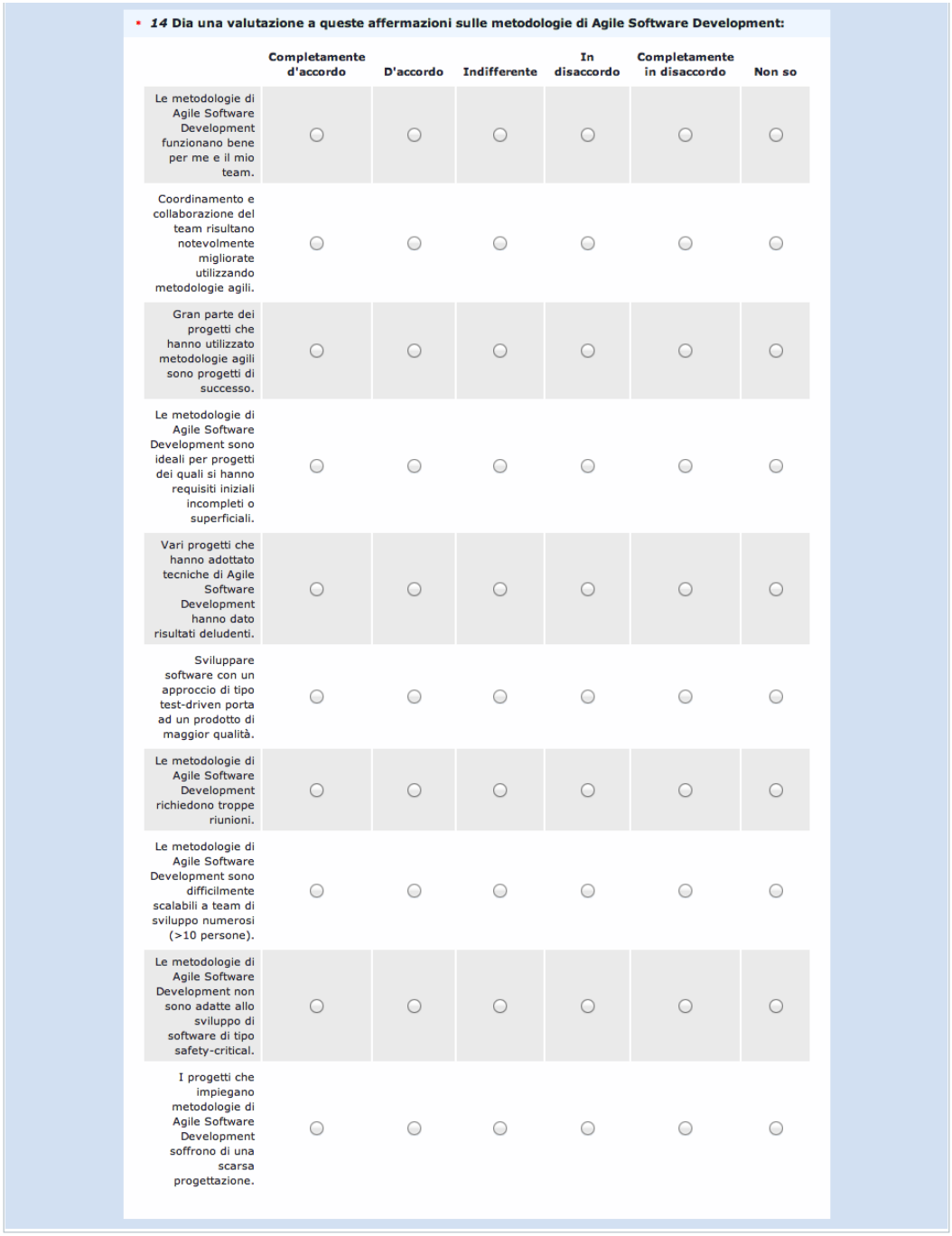
0% 100%

Valutazioni

• ***13*** **Dia una valutazione alle seguenti affermazioni:**

	Completamente d'accordo	D'accordo	Indifferente	In disaccordo	Completamente in disaccordo	Non so
E' diffusa una certa diffidenza fra gli sviluppatori riguardo all'accogliere la modifica dei requisiti in stadi avanzati dello sviluppo software.	○	○	○	○	○	○
E' fondamentale che il team di sviluppo abbia la capacità di auto-organizzarsi.	○	○	○	○	○	○
Il cambiamento di requisiti, anche a stadi di sviluppo avanzati, va tutto a vantaggio competitivo del committente.	○	○	○	○	○	○
Modificare o aggiungere requisiti a stati avanzati di sviluppo di un progetto software aumenta esponenzialmente il costo del cambiamento (in termini di sforzi, tempi, economici).	○	○	○	○	○	○

Figura 25: Questionario: gruppo "valutazioni" (A)



* *14* **Dia una valutazione a queste affermazioni sulle metodologie di Agile Software Development:**

	Completamente d'accordo	D'accordo	Indifferente	In disaccordo	Completamente in disaccordo	Non so
Le metodologie di Agile Software Development funzionano bene per me e il mio team.	○	○	○	○	○	○
Coordinamento e collaborazione del team risultano notevolmente migliorate utilizzando metodologie agili.	○	○	○	○	○	○
Gran parte dei progetti che hanno utilizzato metodologie agili sono progetti di successo.	○	○	○	○	○	○
Le metodologie di Agile Software Development sono ideali per progetti dei quali si hanno requisiti iniziali incompleti o superficiali.	○	○	○	○	○	○
Vari progetti che hanno adottato tecniche di Agile Software Development hanno dato risultati deludenti.	○	○	○	○	○	○
Sviluppare software con un approccio di tipo test-driven porta ad un prodotto di maggior qualità.	○	○	○	○	○	○
Le metodologie di Agile Software Development richiedono troppe riunioni.	○	○	○	○	○	○
Le metodologie di Agile Software Development sono difficilmente scalabili a team di sviluppo numerosi (>10 persone).	○	○	○	○	○	○
Le metodologie di Agile Software Development non sono adatte allo sviluppo di software di tipo safety-critical.	○	○	○	○	○	○
I progetti che impiegano metodologie di Agile Software Development soffrono di una scarsa progettazione.	○	○	○	○	○	○

Figura 26: Questionario: gruppo “valutazioni” (B)

Figura 27: Questionario: gruppo “precedenti esperienze”

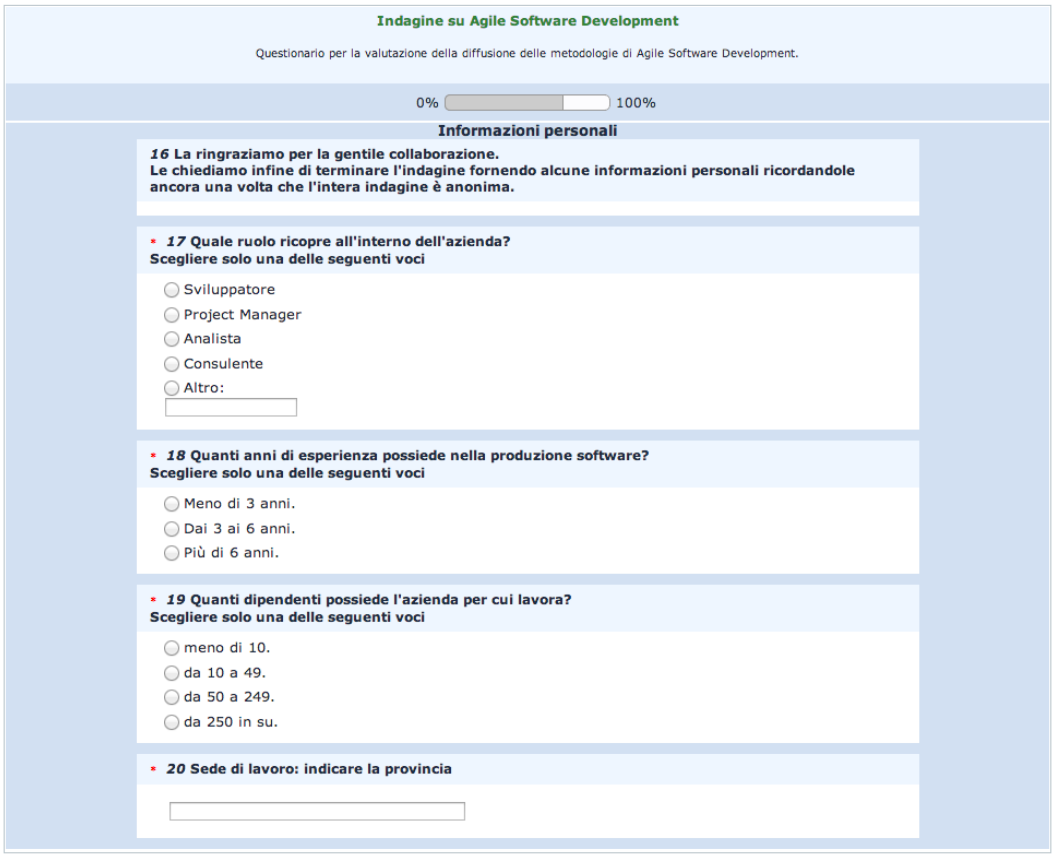


Figura 28: Questionario: gruppo “informazioni personali”

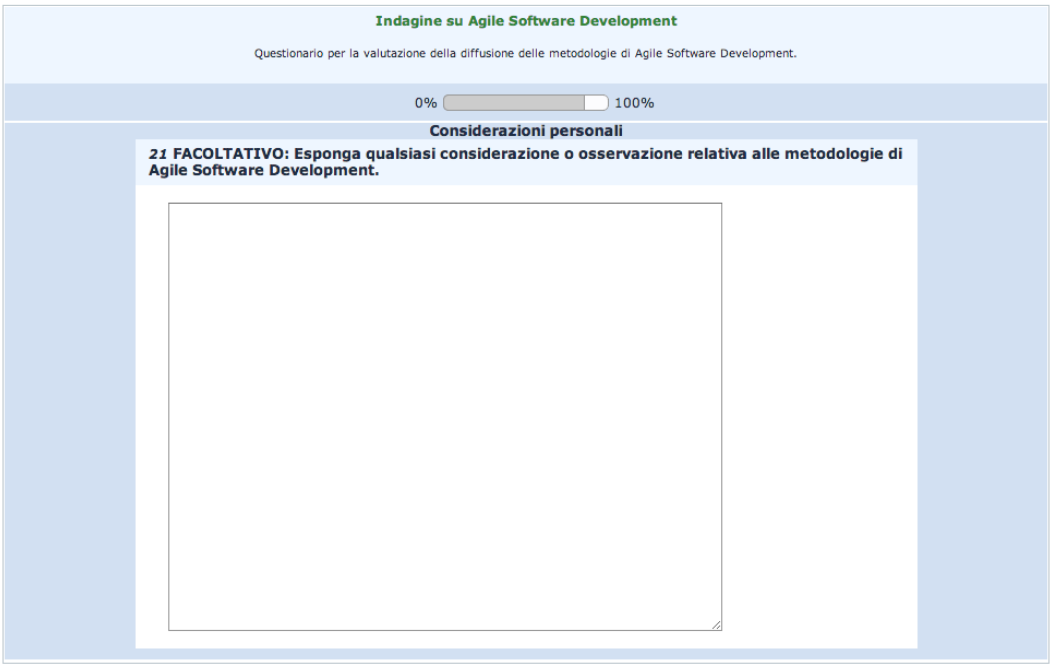


Figura 29: Questionario: gruppo “considerazioni”

BIBLIOGRAFIA

[1] *Advice on Conducting the Scrum of Scrums Meeting - Scrum Alliance*. URL: http://www.scrumalliance.org/articles/46-advice-on-conducting-the-scrum-of-scrums-meeting.

[2] Scott Ambler. *Ambysoft*. http://www.ambysoft.com/surveys/. [Ultimo accesso 4 aprile 2013].

[3] VenuGopal Balijepally et al. «The productivity paradox of pair programming». In: *MIS Quarterly* 33.1 (2009), pp. 91–118.

[4] Kent Beck. *Extreme Programming Explained: Embrace change*. Addison Wesley, 1999.

[5] Kent Beck e Cynthia Andres. *Extreme Programming Explained: Embrace change*. 2ª ed. Addison Wesley, 2004.

[6] Andrew Begel e Nachiappan Nagappan. «*Usage and Perceptions of Agile Software Development in an Industrial Context: An Exploratory Study*». In: *First International Symposium on Empirical Software Engineering and Metrics* (2007).

[7] Herbert D. Benington. «Production of large computer programs». In: *Symposium on advanced programming methods for digital computers* (1956).

[8] Barry Boehm. «A Spiral Model of Software Development and Enhancement». In: *ACM SIGSOFT Software Engineering Notes* 11.4 (1986), pp. 14–24.

[9] Barry Boehm e Richard Turner. *Balancing Agility and Discipline: A Guide for the Perplexed*. Addison Wesley, 2003.

[10] Barry Boehm e Richard Turner. «Observations on Balancing Discipline and Agility». In: *IEEE Computer Society* (2003).

[11] *Build Automation, Agile Alliance*. http://guide.agilealliance.org/guide/autobuild.html. [Ultimo accesso 10 marzo 2013].

[12] Peter Coad, Eric Lefebvre e Jeff De Luca. *Java Modeling In Color With UML: Enterprise Components and Process*. Prentice Hall, 1999.

[13] Umberto Collesei. *Marketing*. 3ª ed. Cedam, 2000.

[14] *Continuous Integration, Agile Alliance*. http://guide.agilealliance.org/guide/ci.html. [Ultimo accesso 10 marzo 2013].

[15] *Dynamic System Development Method Consortium*. http://www.dsdm.org/.

[16] Robert M. Groves et al. *Survey Methodology*. 2ª ed. Wiley, 2009.

[17] Mary Hanna. «Farewell to waterfalls?» In: *Software Magazine* 15.5 (mag. 1995), pp. 38–46.

[18] *Ken Schwaber on Scrum*. http://www.controlchaos.com/message-from-ken/. [Ultimo accesso 6 aprile 2013].

[19] *Lime Service*. http://www.limeservice.com/. [Ultimo accesso: 2 aprile 2013].

[20] *Lime Survey*. http://www.limesurvey.com/. [Ultimo accesso: 2 aprile 2013].

[21] Claudia Lopresto. *Utilizzo di metodologie agili nella PMI*. http://www.linkedin.com/groups/UTILIZZO-METODOLOGIE-AGILE-NELLE-PMI-1944601.S.225935412. [Ultimo accesso 4 aprile 2013].

[22] Peter Nardi. *Doing Survey Research: A Guide to Quantitative Research Methods*. Pearson, 2006.

[23] Juan Nogueira et al. «Surfing the edges of chaos: applications to software engineering». In: *Command and Control Research and Technology Symposium* (2000).

[24] Duncan D. Nulty. «The adequacy of response rates to online and paper surveys: what can be done?» In: *Assessment & Evaluation in Higher Education* 33.3 (2008), pp. 301–314.

[25] A. N. Oppenheim. *Questionnaire design and attitude measurement*. Continuum, 1992.

[26] Steve Palmer e Mac Felsing. *A Practical Guide to Feature-Driven Development*. Prentice Hall, 2002.

[27] Roger S. Pressman. *Principi di Ingegneria del Software*. 4th. Mc Graw Hill, 2004.

[28] Roger S. Pressman. *Software Engineering: a practitioner approach*. 7th. Mc Graw Hill, 2010.

[29] Winston W. Royce. «Managing the Development of Large Software Systems». In: *Proceedings of IEEE WESCON* (1970).

[30] Outi Salo e Pekka Abrahamsson. «Agile methods in European embedded software development organisations». In: *IET Software* (2008).

[31] Ken Schwaber e Mike Beedle. *Agile Project Management with SCRUM*. Prentice Hall, 2002.

[32] *Small and Medium size enterprises*. http://ec.europa.eu/enterprise/policies/sme/facts-figures-analysis/sme-definition/. [Ultimo accesso: 4 aprile 2013].

[33] Ian Somerville. *Ingegneria del software*. 7th. Pearson, 2005.

[34] Jennifer Stapleton. *DSDM Dynamic Systems Development Method: The Method in Practice*. Addison Wesley, 1997.

[35] *SurveyMonkey*. http://www.surveymonkey.com/. [Ultimo accesso: 2 aprile 2013].

[36] Jeff Sutherland e Ken Schwaber. «Business object design and implementation». In: *OOPSLA '95 workshop proceedings* (1995).

[37] Hirotaka Takeuchi e Ikujiro Nonaka. «The New New Product Development Game». In: *Harvard Business Review* (1986).

[38] Giovanni Visco. *Tipologie di campionamento*. http://w3.uniroma1.it/chemo/heritage/campionamento/cstart.html. [Ultimo accesso 4 aprile 2013].

[39] Bill Wake. *System Metaphor*. http://xp123.com/articles/the-system-metaphor/. [Ultimo accesso 6 aprile 2013].