# Agile software development

## Introduzione

Obiettivi:

- Comprendere il giusto punto di equilibrio tra documentazione e sviluppo

Spesso attività di contorno potrebbero rappresentare processi fuorvianti rispetto alla concreto sviluppo del sistema software, nonostante siano fondamentali per la comprensione dei requisiti funzionali posti. Potrebbero essere visualizzate come *pratiche burocratiche*, ossia l'insieme di elementi modellativi trattati non vengono mai sviluppati per ragioni legate alla costruzione del sistema software. A causa di questo elevato standard processuale non è riconducibile alcuna immediatezza tra la progettazione teorica e l'implementazione software, provocando una perdita della qualità.

L'impegno dovuto a garantire coesione a livello strutturale comprende un vasto insieme di azioni, le quali potrebbero provocare un'elevata perdita di tempo qualora l'analisi adottata non sia ben chiara fin dall'inizio. Da cui ne deriva l'inutilità di una pianificazione perfetta, poichè l'intero contesto è soggetto a dinamicità, soprattutto un campo che riguardi lo sviluppo e la progettazione software.

Non solo il *sistema software* dovrà essere flessibile ai cambiamenti, ma l'intero contesto sviluppato dovrà reagire prontamente a modifiche.

## Manifesto per lo sviluppo agile

Il motivo che ha portato alla creazione del **manifesto**, è dovuto al netto spreco di risorse durante fasi di sviluppo software. Il termine correlato, *Agile software development*, non deve essere considerato come un metodo di progettazione, ma rappresenta un insieme di *pratiche* guidate da *principi* e qualità sia *interne* che *esterne*. Attraverso il processo di analisi prodotto occorre valorizzare un insieme di prospettive, quali:

- Interazioni tra progettisti e sviluppatori al di sopra di processi sequenziali e strumenti tecnici
- Sviluppare e adoperare codice piuttosto che predilire una documentazione esaustiva
- Imbastire una collaborazione con il *costumer*, denigrando una *negoziazione* conflittuale
- Reagire prontamente a cambiamenti, evitando di sottostare alle linee guida originarie

Il compito del *manifesto* prevede di attribuire maggiore rilevanza all'entità poste alla sinistra dell'elenco, provando a descrivere un approccio che possa portare ad un concreto beneficio per progetti futuri, senza escludere un prossimo utilizzo di tutti gli elementi posti alla destra.

## Principi

In relazione all'introduzione precedente, sono formulati di seguito i principi su cui stabilisce il proprio approccio il *manifesto*, suddivisi in:

- La priorità principale richiede che il *costumer* sia pienamente soddisfatto del risultato ottenuto, ciò può concretizzarsi solamente se sviluppatori optino per un continuo dialogo con i clienti e fornendo sequenzialmente versioni del sistema software implementato
- Reagire prontamente a modifiche e a variazioni di requisiti funzionali, sfruttando il cambiamento per ottenere vantaggio competitivo
- Valorizzare il *lifecycle* del modello a spirale, il quale impone certe temporalità in cui richiedere confronti e discussioni
- Giornalmente sviluppatori e il *business team* devono rendersi protagonisti nella realizzazione di task scelte
- Predilire il dialogo tra i singoli, poichè permette di diffondere in maniera efficiente ed efficace tutte le informazioni ritenute importanti
- Porre particolare attenzione a tecnologie abili ed eccellenti, relative non solo all'ambiente di sviluppo utilizzato, ma rispetto anche a novità progettuali o metodi differenti di cooperazione, pur di riuscire nell'intento della richiesta
- Adottare un approccio dedicato ad *improvement*, che tendi a migliorare ad ogni passo, piuttosto che formalizzarsi sulla pianificazione dettagliata all'inizio della progettazione

## Metodi agili

Esistono molte pratiche che adottano l'insieme dei principi descritti precedentemente, alcune delle quali sono in totale contraddizione, tuttavia proprio questa discordanza dovrebbe dare vita a un processo legato al continuo miglioramento, pur di aumentare la qualità di progettazione e sviluppo. Le metodologie si suddividono in:

- *Code review*

  *Definizione informale*
  Con il termine *code review* si intende una pratica in cui una nuova porzione di codice deve essere analizzata e approvata prima di poter implementare nuove funzionalità.

  Come da definizione si adotta una revisione del codice affinchè possa essere poi incluso nella soluzione, da cui derivano un insieme di *tool* che certificano la correttezza o meno. Uno dei principali bonus di questa pratica consiste nella maggiore efficacia della soluzione illustrata, poichè è analizzata da un numero crescente di sviluppatori i quali si accertano del *code proposto*. Inoltre, adottando un approccio simile, si tende a condividere capacità e conoscenza, a causa della continua ricerca di nuove soluzioni legate a problemi che possano essere individuati conseguentemente ad azioni di *review*.
- *Test-driven design*

  *Definizione informale*
  Rappresenta uno stile di programmazione in cui tre attività si alternano, scrittura del *codice*, *test* della soluzione proposta ed infine *progettazione*.

  *Test driven design* può essere riassunto in un insieme di regole le quali prevedono un comportamento simile; innanzittutto si tende a descrivere le singole unità che indichino il processo esecutivo del sistema, per poi inizializzare la fase di testing, dove con molta

probabilità, trattandosi di una stesura iniziale, sarà carente di specifiche. Per superare l'errore si cerca di scrivere il codice più semplice possibile, che possa garantire la correttezza del test, anche se potrebbe comportare a soluzioni non adeguate. Infine, si attua *refactoring* del codice affinchè sia comprensibile e riutilizzabile; terminato, si ripete l'intero comportamento per ulteriori funzioni.

- *User stories*

  *Definizione informale*
  Indica un'illustrazione differente dei requisiti funzionali.

  Spesso sono espressioni scritte nel linguaggio del dominio che servono a catturare le aspettative dell'utente, per cui in grado di poter indirizzare lo sviluppo software. Molte *agile software development* tendono ad adottare *user stories* per rappresentare i requisiti, i quali non devono essere confusi con la progettazione modellativa posta da *use case*. Semplicemente, la propria sintassi prevede una costruzione come segue:

  *As a <role>, I want <goal> so that <benefit>*

  Ultima nota stabilisce il corretto uso del costrutto, in quanto deve essere più specifico possibile, altrimenti perderebbe di utilità, e garantire una stesura che promuova prima la descrizione dell'obiettivo per poi illustare il valore raggiungibile.

## INVEST

*User stories* rappresenta uno dei *agile software development* di maggiore uso, dove ad un primo impatto potrebbe essere di facile uso e implementazione; tuttavia, la propria complessità comincia ad emergere di pari passo all'aumento della struttura del sistema software. Per questa difficoltà crescente si adottano strumenti per comprendere la qualità e l'efficacia delle *user stories*. I criteri sono riassunti come:

- *Indipendente*, ogni user story non dovrebbe dipendere da nessun altro elemento. Ciò potrebbe accadere qualora siano descritte user stories che illustrino un livello tecnico molto specifico, attuando sottili caratteristiche.
- *Negoziabile*, le user stories non possono essere intese al pari di contratti. Anzi la loro natura è il risultato di una *negoziazione*, il quale è sottoposto in un qualsiasi momento a nuove rivalutazioni
- *Valorizzabile*, come già accennato nell'introduzione, qualsiasi user story deve portare da un obiettivo ad un risultato *valorizzabile*
- *Stimabile*, il team di sviluppo dovrebbe apprendere il livello di complessità e la totalità del lavoro che caratterizza la user story realizzata
- *Piccola*, illustrando una realtà apparentemente minuta, che contraddistingue un'unica funzionalità, essa deve essere eleborata in una sola *iterazione*. Al termine del processo esecutivo, la funzionalità elaborata è considerata conclusa. Qualora non siano di piccola dimensione, per cui ritraendo problematiche di spessore, si attua la strategia dei *pattern*, in relazione ad una particolare avversità si adotta una soluzione ricorrente, anche se essa non prevede un uso di codice semplice e non articolato

- *Testabile*, indica uno dei punti cardine dei *metodi agili*, in cui l'implementazione di una user story è conclusa solamente quando è conforme al *test di accesso*. Qualora non sia verificata la conformità allora la progettazione della funzionalità non può essere ritenuta terminata

## Agile and evolution

Come già descritto, spesso in un team di sviluppo si tende ad utilizzare la maggior parte del tempo a disposizione e delle risorse per rendere esaustiva la documentazione relativa al sistema software che dovrà essere implementato. Tuttavia tale approccio tende a soffermarsi su *questioni burocratiche*, accontando il vero sviluppo software legato ai requisiti funzionali.

Per cui lo *sviluppo* rappresenta una fase fondamentale ma non sufficiente per completare il lifecycle del sistema software; occorre promuovere un'evoluzione della soluzione attuata, affinchè sia garantito un comportamento adattivo che si adegui a possibili modifiche oppure ad aggiunte di funzionalità.

La metodica **agile-evolution** raffigura un modalità che ovvia alla costringente stesura di documentazione, focalizzandosi sulla condivisione della conoscenza acquisita durante fasi di implementazione del codice, mediante la scrittura di un *piccolo archivio*.

La realizzazione della documentazione deve avvenire al di fuori delle fasi prestabilite per l'implementazione del codice, con l'elevata probabilità che la stesura dell'archivio non venga mai effettivamente concretizzata. Ciò non rappresenta una mancanza di estrema importanza, poichè in *agile software development* le informazioni di spessore sono veicolate tramite il dialogo, ponendo in secondo luogo la documentazione, anche se una totale assenza di uno storico relativo al lavoro svolto potrebbe provocare difficoltà qualora il progetto software sia demandato ad altri team di sviluppo.

## Extreme programming

**Extreme programming** è una metodologia agile ideata dal teorico *Beck*, personaggio che nel panorama della materia trattata capì l'importanza delle dipendenze all'interno dello sviluppo di un sistema software. Per certi aspetti è molto simile al *manifesto agile*, ma provvede a focalizzarsi su ulteriori caratterizzazioni.

*XP* tende a valorizzare attività che ovviano alla stesura di una documentazione dettagliata, ponendo maggiore importanza sulla concreta implementazione della soluzione. Per cui sono imposti all'interno del team atteggiamenti che ritraggono il dialogo tra gli sviluppatori, la scrittura del codice oppure la progettazione di soluzioni inerenti a problemi sorti.

Si fonda, come da descrizione precedente, su valori legati alla comunicazione all'interno del team di sviluppo, all'adozione di soluzioni più semplici possibili affinchè sia possibile superare l'ostacolo ed infine alla responsabilizzazione di ogni componente del gruppo, rendendolo partecipe attivamente alla progettazione del sistema software; tuttavia per rendere concreto ciò che è stato trattato, occorre attuare una serie di principi che contraddistinguono *extreme programming*, suddivisi come segue:

- *Fine scale feedback*

  *Definizione informale*
  Fine scale feedback ammette che qualsiasi decisione strutturale deve essere condivisa da tutto il team di sviluppo.

  Per cui sono adottate metodologie relative alla definizione precedente, quali:

  – *Pair programming*, la stesura delle funzionalità è affidata a due entità del gruppo di sviluppo, dove tendenzialmente avviene uno scambio di mansioni tra le due figure, come la scrittura del codice e l'analisi della soluzione proposta
  – *Test driven development*, si attuano fasi di test dello sviluppo adeguato fino a un dato momento, in cui, qualora siano verificate anomalie, occorre sviluppare la soluzione più semplice possibile, a cui, successivamente, si adattano attività di refactoring
  – *Whole team*, all'interno del team non devono esistere suddivisioni nette tra i membri, e ognuno di essi deve acquisire le competenze necessarie per cimentarsi in qualsiasi sezione dello sviluppo del software

  Concludendo, tramite questo principio, è imposta la coesione tra ogni elemento del team, in cui si tende a valorizzare la partecipazione di ognuno di essi nell'ambito decisionale e progettuale del sistema software attuato.

- *Continuos process*

  *Definizione informale*
  ...

  Nuovamente sono introdotte ulteriori pratiche che contraddistinguono *continuos process*, suddivise come:

  – *Continuos integration*, qualora siano attuate delle modifiche, occorre che siano riportate immediatamente all'interno della soluzione del *code base*, dove il termine proposto indica l'insieme di punti necessari per ottenere la soluzione software
  – *Design improvement*, la stesura di risoluzioni semplici a problemi sorti non deve accontanare l'attività di *refactoring*, poichè potrebbero rappresentare una situazione di *debito tecnico*
  – *Small releases*, adeguare lo sviluppo software affinchè sia possibile rilasciare versioni del sistema monitorabili e analizzabili, affinchè ciò si possa tradurre in un vantaggio competitivo e dichiarare se la progettazione attuata sia conforme con *user stories*, oppure semplicemente con i requisiti funzionali dell'utente finale

- *Shared understanding*

  *Definizione informale*
  ...

  Principio che comprende:

  – *Coding standard*, opportuno che siano definiti paradigmi di progettazione e sviluppo equivalenti per ogni membro del team

  - *Simple design*, processo esecutivo equivalente a *test driven design*, ossia verificate anomalie si sviluppano soluzioni di estrema semplicità, a cui si adeguano attività di *refactoring* affinchè si possa eliminare il *debito tecnico*
  - *System metaphor*, in certi contesti, potrebbe rappresentare una buona idea l'utilizzo di termini, all'interno della soluzione proposta, correlati al linguaggio del domonio, ossia rendere la tematica comprensibile anche per attori posti al di fuori del team di sviluppo, nonostante spesso è raffigurato come un principio di scarsa importanza, poichè non influisce direttamente sull'implementazione dei requisiti funzionali

- *Programmer welfare*

  *Definizione informale*
  *Programmer welfare* pone la propria centralità sul lavoro e sviluppo cooperativo, affinchè l'implementazione del sistema software sia suddivisa in *time box*, alludendo ad un progetto che debba essere svolto a step e valorizzando le interazioni tra membri del gruppo, racchiudendo l'intero contesto su cui si formula *extreme programming*.