Indice

3 – Metodi agili 2
3.1 – Tecniche di sviluppo agile 4
3.1.1 – Storie utente (o scenari) 5
3.1.2 – Refactoring 6
3.1.3 – Sviluppo con test iniziali 7
3.1.4 – Programmazione a coppie 8
3.2 – Gestione agile della progettazione (metodo Scrum) 9

3 – Metodi agili

I **metodi agili** sono particolarmente indicati per sviluppare applicazioni nelle quali i requisiti del sistema cambiano rapidamente durante il processo di sviluppo. Sono stati **ideati per consentire** una **consegna rapida** del software ai clienti, i quali possono quindi proporre requisiti nuovi o modificati da includere in successive iterazioni del sistema.

L'**obbiettivo** è quello di ridurre la burocrazia nei processi di sviluppo, evitando quel lavoro di dubbia validità a lungo termine ed eliminando la documentazione che probabilmente non sarà mai utilizzata.

La **filosofia** che sta alla base dei metodi agili è illustrata nel seguente **manifesto** (Link):

Tutti i metodi agili suggeriscono che il software deve essere sviluppato e consegnato in modo incrementale.

Esistono cinque **principi** cardine su cui si fondano i **metodi agili**:

- **Coinvolgimento del cliente**. I clienti devono essere coinvolti in tutto il processo di sviluppo. Il loro ruolo è quello di fornire nuovi requisiti del sistema.
- **Accettare i cambiamenti**. Progettare il sistema in modo che si possa accogliere eventuali cambiamenti.
- **Consegna incrementale**. Il software viene sviluppato per incrementi e il cliente specifica i requisiti da includere in ogni incremento.
- **Mantenere la semplicità**. Concentrarsi sulla semplicità nel software e nel processo di sviluppo. Se possibile, lavorare per eliminare le complessità del sistema.
- **Persone, non processi**. Le capacità del team devono essere riconosciute e non sfruttate. Essi devono essere lasciati liberi di sviluppare il software a seconda dei metodi di lavoro, senza processi prescrittivi.

I metodi agili sono **utili** per lo **sviluppo** dei seguenti **sistemi**:

1. Sviluppo di **prodotti** di **piccole** o **medie dimensioni**. Tutti i prodotti software e le applicazioni oggi vengono sviluppati utilizzando un approccio agile.
2. Sviluppo **personalizzato** di **sistemi** all'interno di un'organizzazione, dove c'è un chiaro impegno da parte del cliente di essere coinvolto nel processo di sviluppo e dove ci sono pochi stakeholder e regolamenti esterni che possono influire sul software.

I metodi agili funzionano bene in queste situazioni, in quanto è possibile stabilire una comunicazione continua tra il responsabile del prodotto o il cliente e il team di sviluppo.

3.1 – Tecniche di sviluppo agile

Le idee alla base dei metodi agili furono sviluppate negli anni '90 grazie alla programmazione estrema (XP, *eXtreme Programming*) in cui si adotta un approccio a livelli "estremi" spingendo le normali pratiche, come lo sviluppo iterativo, al limite.

La programmazione estrema fu una tecnica controversa, in quanto introduceva un certo numero di pratiche agili che erano molto diverse da quelle tradizionali dell'epoca. Tuttavia, alcune metodologie sono comuni anche nello sviluppo agile:

1. Lo sviluppo incrementale è supportato attraverso piccole e frequenti release del sistema. I requisiti vengono chiamati "scenari", i quali sono utilizzati come base per decidere quale funzionalità deve essere inclusa in un incremento del sistema.
2. Il coinvolgimento dell'utente è supportato attraverso l'impegno costante del cliente nel team di sviluppo; anche i rappresentati del cliente hanno la responsabilità di definire i test di accettabilità del sistema.
3. Le persone, non il processo, sono supportate dalla programmazione in coppia, dal possesso collettivo del codice del sistema, e da un processo di sviluppo sostenibile che non richiede periodi di lavoro eccessivamente lunghi.
4. Le modifiche sono supportate da regolari release del sistema, dallo sviluppo preceduto da test, dal refactoring per evitare la degenerazione del codice, e dall'integrazione continua di nuove funzionalità.
5. Il mantenimento della semplicità è supportato dal costante refactoring che migliora la qualità del codice e dall'uso di semplici progetti che non necessariamente prevedono future modifiche del sistema.

Nella pratica reale, l'applicazione della programmazione estrema si è rilevata più difficile del previsto. Non può essere integrata con alcune pratiche di gestione e le tradizioni di molte aziende. Per questo motivo, le società che adottano i metodi agili selezionano quelle pratiche XP che sono più appropriate al loro modo di lavorare.

3.1.1 – Storie utente (o scenari)

Le "**storie utente**" (o scenari) è uno scenario d'uso in cui potrebbe trovarsi un utente del sistema.

Dunque, il **cliente** lavora a stretto contatto con il team di sviluppo e **discute** questi **scenari con** altri membri del **team**. **Insieme sviluppano** una "carta della storia" (*story card*) che raccoglie le esigenze dell'utente. Il **team implementa** tale **scenario** in una release successiva del software.

Una volta sviluppate le *story card*, il **team** di sviluppo le **suddivide in task** e stima le risorse e gli sforzi richiesti per implementare ciascun task. Per fare questo occorre discutere con il **cliente** in modo da **perfezionare i requisiti**. Il cliente inoltre elenca in ordine di priorità le storie. L'**obbiettivo** in questa fase è identificare le funzionalità utili che possono essere implementate in due settimane circa, quando la successiva release del sistema sarà presentata al cliente.

Nel caso in cui i **requisiti variano**, le storie non implementate possono cambiare o essere scartate. Se sono richieste modifiche per un sistema che è già stato consegnato, vengono sviluppate nuove *story card* e il cliente deciderà se tali modifiche dovranno avere priorità sulle nuove funzionalità.

Le storie utente hanno un **vantaggio** ovvio, ovvero sono **molto efficaci**. Infatti, le persone trovano molto più semplice relazionarsi con queste storie, anziché con un tradizionale documento di requisiti o con i casi d'uso.

Il **problema** principale degli scenari è la **completezza**. È difficile stabilire se sono state sviluppate storie utente sufficienti a trattare tutti i requisiti essenziali di un sistema. È anche difficile stabilire se una singola storia fornisce la rappresentazione vera di un'attività. Gli utenti esperti sono così familiari con il loro lavoro che spesso omettono alcuni particolari quando lo descrivono.

3.1.2 – Refactoring

La programmazione estrema considera la progettazione per il cambiamento spesso uno sforzo inutile. Ovvero, non vale la pena impiegare il tempo per rendere più generico un programma in modo da far fronte ai cambiamenti. Spesso le modifiche previste non si avverano mai oppure vengono richieste modifiche completamente diverse.

Gli **sviluppatori** XP **suggeriscono** che il codice che si sta sviluppando debba essere **costantemente** "rifattorizzato". La "rifattorizzazione" o meglio, ***refactoring*** richiede che il team di programmazione ricerchi possibili miglioramenti del software e li implementi immediatamente.

Purtroppo, un **problema** fondamentale dello **sviluppo incrementale** è che le modifiche locali tendono a deteriorare la struttura del software. Di conseguenza, le modifiche future diventano sempre più difficili da implementare. In altre parole, lo sviluppo procede in modo da trovare dei modi per aggirare i problemi, con il risultato che il codice spesso è duplicato, parti del software vengono riutilizzate in modi inappropriati, e la struttura complessiva del software si deteriora ogni volta che viene aggiunto nuovo codice. Il **refactoring migliora la struttura e la leggibilità del software**, **evitando** così il **deterioramento strutturale** che si verifica naturalmente quando si modifica il software.

In **linea teorica**, quando il refactoring è parte del processo di sviluppo, il software dovrebbe essere sempre facile da capire e modificare quando vengono proposti nuovi requisiti. Tuttavia, nella **pratica**, non è sempre possibile eseguire il refactoring poiché spesso viene rinviato a causa dell'alta priorità dell'implementazioni di nuove funzionalità.

3.1.3 – Sviluppo con test iniziali

La differenza più grande tra lo sviluppo incrementale e lo sviluppo guidato da piani è il modo in cui il sistema viene testato. Nello **sviluppo incrementale** non c'è una specifica del sistema che può essere utilizzata da un team esterno per sviluppare i test del sistema. Questo significa che lo sviluppo incrementale ha un processo di testo molto informale.

La **programmazione estrema** ha introdotto un nuovo approccio al test dei programmi. I **test** sono **automatizzati** e sono **centrali nel processo di sviluppo**, e lo **sviluppo non** può **procedere** finché tutti i test non sono stati superati con successo.

Gli **elementi del test** nella **programmazione estrema** sono:

1. Sviluppo con test iniziali;
2. Sviluppo di test incrementale dagli scenari;
3. Coinvolgimento dell'utente nello sviluppo e nella convalida dei test;
4. Uso di strutture automatiche per i test.

Lo sviluppo guidato da test è una delle più **importanti innovazioni** poiché anziché scrivere il codice e poi i test per il codice, vengono **scritti i test prima del codice**. Quindi è possibile eseguire i test mentre viene scritto il codice e scoprire eventuali problemi durante lo sviluppo.

La scrittura dei test implica la definizione di un'interfaccia e di una specifica comportamentale per le funzionalità da sviluppare. I problemi con i requisiti e le incomprensioni dell'interfaccia si riducono.

Il **problema del ritardo dei test** (*test-lag*) si manifesta quando le ambiguità e le omissioni nelle specifiche non sono chiare. Questo può accadere quando lo sviluppatore del sistema lavora più velocemente di chi esegue i test. L'implementazione è sempre più veloce del processo di test, e c'è la tendenza a tralasciare i test per poter rispettare la tempistica dello sviluppo.

L'**approccio con i test iniziali** presuppone che le storie utente (scenari) siano state sviluppate e che queste siano state suddivise in una serie di carte di task. Ogni task genera uno o più test da eseguire.

Inoltre, il **ruolo del cliente** nel processo di test consiste nell'aiutare a sviluppare i test di accettazione delle storie che devono essere implementate nella successiva release del sistema.

L'**automazione dei test** è essenziale per lo sviluppo con i test iniziali. I test vengono scritti come componenti eseguibili prima che il task sia implementato. Questi componenti devono essere autonomi, devono simulare gli input da provare e devono controllare che il risultato soddisfi la specifica degli output. Un framework automatico di test è un sistema che semplifica la scrittura di test eseguibili e propone una serie di test da eseguire.

È difficile garantire che la serie dei test sia completa poiché:

1. I programmatori preferiscono la programmazione al testing, e a volte scrivono test incompleti che non sono in grado di verificare tutti i casi insoliti che potrebbero verificarsi.
2. Alcuni test sono molto difficili da scrivere immediatamente. Per esempio, in un'interfaccia utente complessa spesso è difficili scrivere test per il codice che implementa la logica di visualizzazione e il flusso di lavoro tra le schermate.

3.1.4 – Programmazione a coppie

Un'altra pratica innovativa della programmazione estrema consiste nel fatto che i **programmatori operano in coppia** per sviluppare il software. L'idea è che le coppie vengano create dinamicamente in modo che tutti i membri del team possano lavorare in coppi con altri durante il processo di sviluppo.

I **vantaggi** sono molteplici:

1. Supporta l'**idea** della **proprietà** e della **responsabilità comune del sistema**. Quindi, il software è di proprietà dell'intero team e i singoli non sono ritenuti responsabili dei problemi riscontrati nel codice. È il team che ha la responsabilità collettiva della risoluzione di questi problemi.
2. È un processo di **revisione informale**, poiché ogni linea del codice è visionata da almeno due persone. L'ispezione e la revisione sono efficaci nello scoprire un'alta percentuale di errori del software, ma richiedono tempo per l'organizzazione e causano ritardi nel processo di sviluppo.
3. Incentiva il **refactoring** per migliorare la struttura del software. Lo sforzo richiesto produce benefici nel lungo termini.

Tuttavia, alcuni **studi formali** hanno scoperto che la produttività con la programmazione a coppie sembra essere confrontabile con quella di due persone che lavorano indipendentemente, perché le coppie discutono il software prima di svilupparlo; quindi, probabilmente hanno meno false partenze e meno necessità di rielaborare il codice. Inoltre, il numero di errori evitati dall'ispezione informale è tale che occorre meno tempo risolvere i problemi scoperti durante il processo di test.

Un eventuale **svantaggio** è nel caso di programmatori più esperti. È emerso che c'era una significativa perdita di produttività rispetto a due programmatori che lavoravano da soli. C'erano alcuni benefici per la qualità, ma questi non compensavano completamente gli overhead della programmazione a coppie.

Ciononostante, la **condivisione** delle **conoscenze** è **molto importante**, riduce i rischi di fallimento di un progetto nel caso in cui alcuni membri abbandonino il team di sviluppo. Solo questo rende valida la programmazione a coppie.

3.2 – Gestione agile della progettazione

Un approccio basato su piani richiede che un manager abbia una visione continua su qualsiasi cosa deve essere sviluppato e sullo sviluppo dei processi. Questo non accade nei metodi agili in cui i team sono organizzazioni autonome, non producono documentazione e sviluppano in cicli molto brevi. Tale modus operandi va bene per le piccole società, ma non è appropriato per le società più grandi e quindi viene visto come un **problema**.

Per **risolvere** il problema, fu sviluppato il **metodo agile Scrum** che offre un framework per organizzare progetti agili e fornire una visibilità esterna su ciò che sta accadendo. Gli sviluppatori di Scrum avevano chiarito che Scrum non era un metodo per la gestione dei progetti nel senso convenzionale, per questo motivo avevano deliberatamente creato una nuova terminologia, come ScrumMaster, che sostituiva termini come "project manager".

Nella seguente tabella ci sono i vari termini ed i loro significati.

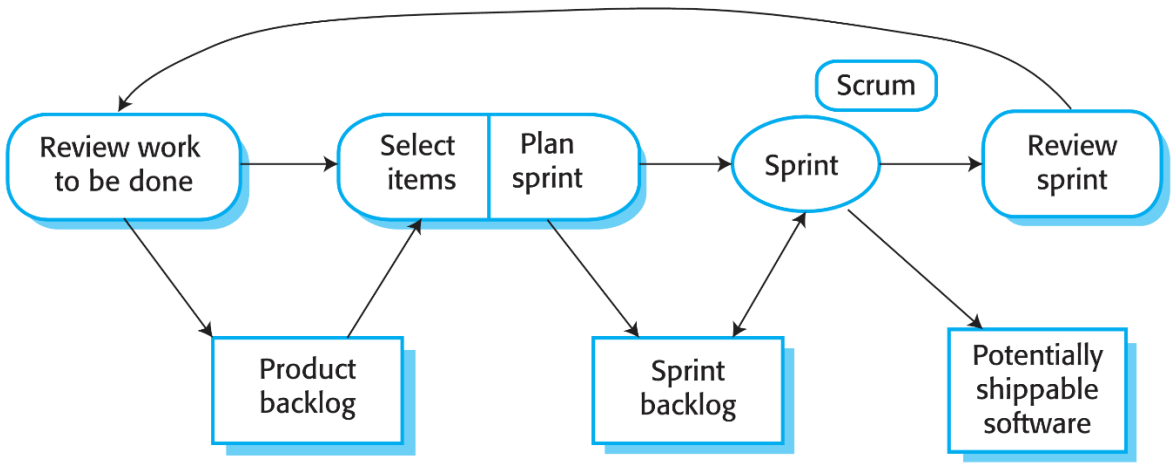
Termine di Scrum	Definizione
Team di sviluppo	Gruppo di sviluppatori software con organizzazione autonoma, che non dovrebbe avere più di 7 persone. Sono responsabili dello sviluppo del software e di altri documenti essenziali di progettazione.
Incremento di un prodotto potenzialmente rilasciabile	L'incremento del software che è consegnato da uno sprint deve essere "potenzialmente rilasciabile", ovvero deve trovarsi in uno stato finito e non occorre altro lavoro per incorporarlo nel prodotto finale.
Product backlog	Lista di elementi di cui si deve occupare il team di Scrum.
Product owner	Un individuo, o piccolo gruppo, il cui compito è identificare le caratteristiche o i requisiti del prodotto, stabilirne le priorità e rivedere continuamente il product backlog per garantire che il progetto continui a soddisfare i requisiti critici. Questa figura può essere un cliente, un product manager o un rappresentante degli stakeholder.
Scrum	Una riunione giornaliera del team che esamina l'avanzamento del lavoro e stabilisce le priorità del lavoro da svolgere in quel giorno. Teoricamente, dovrebbe essere un incontro faccia a faccia di tutti i membri del team.

ScrumMaster	Lo ScrumMaster ha la responsabilità di garantire che il processo Scrum sia seguito e di guidare il team nell'uso efficiente di Scrum. Deve anche fungere da interfaccia con il resto della società e garantire che il team non venga sviato da interferenze esterne. Nonostante possa assomigliare come figura, lo ScrumMaster **non** deve essere considerato come project manager.
Sprint	Una iterazione dello sviluppo. Gli sprint di solito durano da 2 a 4 settimane.
Velocità	Una stima della quantità di lavoro del product backlog che un team può svolgere in un singolo sprint. Conoscere la velocità di un team aiuta a stimare che cosa può essere svolto in uno sprint e fornisce la base per misurare i miglioramenti.

Lo **Scrum** è un **metodo agile** poiché segue i principi del manifesto per lo sviluppo agile; tuttavia, è stato inventato per fornire un **framework per l'organizzazione agile dei progetti**.

Essendo un metodo agile, può essere più facilmente integrato con i metodi esistenti in una società e tutt'ora è il metodo più largamente utilizzato.

Il **processo Scrum** (o ciclo degli sprint), illustrato in figura, ad ogni iterazione genera un incremento del prodotto che può essere consegnato al cliente:



Il **punto di partenza**, nonché l'input del processo, è il **product backlog**. La versione iniziale del product backlog può essere derivata da un documento dei requisiti, da una lista delle storie utente o da un'altra descrizione del software che si sta sviluppando.

Il product backlog può essere specificato a vari livelli di dettagli; il product owner ha la responsabilità di garantire che il livello di dettagli nella specifica sia appropriato al lavoro da svolgere. Per esempio, un elemento di backlog potrebbe essere la storia completa di un utente.

Ogni ciclo di sprint ha una durata prestabilita, che solitamene è tra le 2 e 4 settimane. All'inizio di ogni ciclo, il product owner stabilisce le priorità del product backlog per definire quali sono gli elementi più importanti da sviluppare in quel ciclo. Gli **sprint** non vengono **mai prolungati** a causa del lavoro non ultimato. Gli elementi che non possono essere completati entro il tempo assegnato allo sprint vengono restituiti al product backlog.

Tutti i membri del team vengono coinvolti nella **scelta degli elementi** con priorità più alta che dovranno essere completati e successivamente viene valutato il tempo richiesto per completare tali elementi. Dopo la scelta, avviene la **creazione** di uno **sprint backlog**, ovvero il lavoro da svolgere in quello sprint. Il team sceglie chi dovrà lavorare su determinati elementi, e avvia lo sprint.

Durante lo sprint, il team si riunisce ogni giorno per esaminare l'avanzamento del lavoro e, se necessario, per ridefinire le sue priorità. Inoltre, tutti i membri del team condividono le informazioni, descrivono il lavoro svolto, illustrano i problemi che hanno incontrato e stabiliscono cosa deve essere fatto per il giorno successivo. Così facendo, ciascun membro sa che cosa sta accadendo e, in caso di problemi, si può ripianificare il lavoro a breve termine per risolverli. Ognuno partecipa a questa pianificazione a breve termine.

Le interazioni giornaliere sono coordinate tramite la **lavagna di Scrum**. Non è altro che una lavagna bianca per ufficio che riporta le informazioni e note sullo sprint backlog, sul lavoro svolto, sull'indisponibilità delle persone e via dicendo. È una risorsa condivisa per tutto il team, e chiunque può modificare o spostare gli elementi della lavagna.

Alla **fine di ogni sprint**, si esegue una riunione di verifica, che coinvolge tutti i membri. La riunione ha **due obbiettivi**:

1. Strumento per migliorare il processo; il team riesamina il modo in cui è stato svolto il lavoro e riflette su come avrebbe potuto fare meglio le cose;
2. Viene fornito l'input sul prodotto e sul suo stato per la revisione del product backlog che procede il successivo sprint.

L'obbiettivo dello **ScrumMaster** riferisce sull'avanzamento del lavoro al senior manager e prende parte alla pianificazione a lungo termine e al budget del progetto. Può essere coinvolto nell'amministrazione del progetto (programmare le ferie dello staff, collaborare con l'ufficio del personale ecc.) e nell'acquisto di componenti hardware e software.

Gli **utenti apprezzano** vari aspetti del metodo Scrum:

1. Il **prodotto** è **suddiviso in parti** gestibili e **comprensibili** alle quali gli stakeholder possono fare riferimento.
2. I **requisiti instabili non** fanno **ritardare** l'avanzamento del lavoro.
3. L'intero **team** ha una **visione su tutto** e, di conseguenza, la comunicazione e il morale dei suoi membri sono migliori.
4. I **clienti ricevono** in tempo gli **incrementi** e hanno un **feedback** su come funziona il prodotto.
5. C'è **fiducia tra clienti e sviluppatori**, quindi si genera un'atmosfera positiva. Questo perché tutti si aspettano che il progetto avrà successo.

[Attualità] Inizialmente, Scrum fu progettato per essere utilizzato da team fisicamente vicini. Tuttavia, lo sviluppo del software moderno si svolge con team fisicamente distanti, i cui membri sono distribuiti in vari luoghi del mondo. Questo consente alle società di trarre vantaggi dai costi più bassi dello staff di altre nazioni, di coinvolgere persone specializzate, di sviluppare il software 24 ore al giorno, in quanto il lavoro viene svolto in luoghi con fusi orari differenti.