



UNIVERSITÀ
DEGLI STUDI
FIRENZE

Scuola di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea in Informatica

Tesi di Laurea

METODI AGILI PER LO SVILUPPO DEL SOFTWARE
AGILE METHODS FOR SOFTWARE DEVELOPMENT

GABRIELE BANI

Relatore: *B. Venneri*

Anno Accademico 2012-2013

A Osvaldo, Flora
e a tutta la mia famiglia

CONTENTS

Ringraziamenti	5
Introduzione	7
1 METODOLOGIE AGILI	9
1.1 INTRODUZIONE	9
1.2 MANIFESTO PER LO SVILUPPO AGILE DI SOFTWARE	10
1.2.1 GLI INDIVIDUI E LE INTERAZIONI PIÙ CHE I PROCESSI E GLI STRUMENTI	11
1.2.2 IL SOFTWARE FUNZIONANTE PIÙ CHE LA DOCUMENTAZIONE ESAUSTIVA	11
1.2.3 LA COLLABORAZIONE COL CLIENTE PIÙ CHE LA NEGOZIAZIONE DEI CONTRATTI	11
1.2.4 RISPONDERE AL CAMBIAMENTO PIÙ CHE SEGUIRE UN PIANO	12
1.3 PRINCÌPI DEL MANIFESTO	12
1.4 TIPI DI METODOLOGIE AGILI	15
1.4.1 EXTREME PROGRAMMING	15
1.4.2 SCRUM	23
1.4.3 DYNAMIC SYSTEMS DEVELOPMENT METHOD	25
1.4.4 CRYSTAL	27
1.4.5 LEAN SOFTWARE DEVELOPMENT	28
1.5 CONCLUSIONI	30
2 TEST-DRIVEN DEVELOPMENT : "UNIT TEST"	31
2.1 INTRODUZIONE	31
2.1.1 TEST	31
2.2 SVILUPPO DEL TEST-DRIVEN DEVELOPMENT	33
2.2.1 A CHI È RIVOLTO IL TEST-DRIVEN DEVELOPMENT	35
2.2.2 STRATEGIE DEL TEST-DRIVEN DEVELOPMENT	35
2.2.3 VANTAGGI DEL TEST-DRIVEN DEVELOPMENT	39
2.3 JUNIT	40
2.3.1 ASSERT	41
2.3.2 ERROR & FAILURE	42
2.3.3 ALTRI METODI	42
2.3.4 TEST DOUBLE	43
2.4 ESEMPI DI USO DI JUNIT	45
2.4.1 ESEMPIO: CONTO CORRENTE	45
2.4.2 ESEMPIO: MONETE	48
2.5 IMPATTO DEL TEST-DRIVEN DEVELOPMENT SULLA PRODUTTIVITÀ E QUALITÀ DEL SOFTWARE	51

2.6	CONSIDERAZIONI	52
2.6.1	COSA TESTARE E COSA NON TESTARE	52
2.6.2	COME RICONOSCERE LA BONTÀ DI UN TEST	53
2.6.3	QUANDO ELIMINARE UNA CLASSE DI TEST	54
3	REFACTORING	55
3.1	INTRODUZIONE	55
3.2	VANTAGGI	56
3.2.1	MIGLIORAMENTO DEL DESIGN	56
3.2.2	CODICE PIÙ LEGGIBILE	56
3.2.3	SI "SCOPRONO" ERRORI IN ANTICIPO	57
3.2.4	SI PROGRAMMA PIÙ VELOCEMENTE	57
3.3	"CODE SMELL"	57
3.3.1	DUPLICATED CODE	58
3.3.2	LONG METHOD	58
3.3.3	LARGE CLASS	59
3.3.4	LONG PARAMETER LIST	59
3.3.5	SWITCH	59
3.3.6	LAZY CLASS	60
3.4	QUANDO ESEGUIRE IL REFACTORING	60
3.4.1	"LA REGOLA DEL TRE"	61
3.4.2	QUANDO SI AGGIUNGONO NUOVE FUNZIONALITÀ AL SOFTWARE	61
3.4.3	QUANDO C'È LA NECESSITÀ DI CORREGGERE AL- CUNI ERRORI	61
3.4.4	QUANDO C'È LA NECESSITÀ DI REVISIONARE IL CODICE	61
3.5	QUANDO NON ESEGUIRE IL REFACTORING	62
3.6	TIPI DI REFACTORING	62
3.6.1	REFACTORING "EXTRACT"	62
3.6.2	REFACTORING "PUSH/PULL"	66
3.6.3	REFACTORING COMPLESSI	69
3.7	ESEMPIO DI REFACTORING	73
3.7.1	MODELLO MVC (Model View Controller)	73
3.7.2	PATTERN OBSERVER	74
3.7.3	ESEMPIO	74
4	CONCLUSIONI	85
5	BIBLIOGRAFIA	89

RINGRAZIAMENTI

Ringrazio tutta la mia famiglia che mi è sempre stata vicina in questi anni di studio nonostante le enormi spese da sostenere.

Ringrazio soprattutto i miei nonni, Osvaldo e Flora, che mi hanno trasmesso sempre forza, coraggio e felicità anche nei momenti meno brillanti della loro splendida vita.

Ringrazio la prof.ssa Battistina Venneri per il suo prezioso aiuto nello svolgimento della tesi.

INTRODUZIONE

Il termine "metodologia Agile" fu coniato, per la prima volta, attorno all'anno 2001 quando gli esponenti più significativi nell'area della programmazione, decisero di riunirsi e di formalizzare una serie di principi innovativi, che avrebbero rivoluzionato le pratiche di sviluppo del software. Lo scopo di questa nuova prospettiva era quello di abbattere i tempi, quindi i costi, dello sviluppo aumentando la velocità di realizzazione e ponendo una forte attenzione alle qualità di flessibilità e riusabilità del codice prodotto. La metodologia Agile si è così affermata nel corso degli ultimi anni, in modo pervasivo, portando in tutti gli ambienti di sviluppo software principi e pratiche di programmazione innovativi, insieme con un nuovo stile di organizzazione del lavoro di team centrato su una forte collaborazione fra gli sviluppatori ed una continua interazione con il cliente.

L'oggetto di questa tesi è un approfondimento degli aspetti più importanti di questo nuovo approccio.

Nel primo capitolo vengono presentati i fondamenti dei metodi Agili, discutendo le caratteristiche principali delle più famose metodologie Agili.

Nel secondo capitolo, viene illustrato il Test-Driven Development, una ben nota pratica di sviluppo del codice facente parte della metodologia "eXtreme Programming": lo sviluppo di alcuni esempi permette di mettere in luce come si applica tale tecnica ed i numerosi vantaggi che essa comporta.

Nel terzo capitolo viene analizzato il "Refactoring", un metodo Agile per lo sviluppo del codice che ha come obiettivo quello di migliorare la struttura interna del progetto software senza alterarne le funzionalità. Attraverso alcuni esempi, cerchiamo di definire non solo i principali tipi di refactoring ma soprattutto il ruolo della loro applicazione nello sviluppo del software.

Infine, nell'ultimo capitolo, sono presentate alcune considerazioni critiche sulla metodologia Agile. In particolare, discutiamo quelli che sono, a nostro parere, alcuni limiti di tale approccio e, nel contempo, indichiamo come possibili sviluppi futuri quegli aspetti che sono indicati dalla stessa comunità Agile come i "punti deboli" che necessitano di un ulteriore approfondimento.

METODOLOGIE AGILI

1.1 INTRODUZIONE

Con il termine *metodologie Agili* si intende indicare un gruppo di metodi, principi di progettazione e tecniche di lavoro che possono essere utilizzati dagli sviluppatori per creare progetti software in brevi cicli temporali, privilegiando la comunicazione fra sviluppatori e cliente e riducendo al massimo i tempi di sviluppo e consegna del prodotto finale. Ogni metodologia Agile ruota attorno a tre concetti fondamentali:

- **Trasparenza.** In base alla dimensione del progetto software che si vuole realizzare, occorre che il team di sviluppo software sia composto da un numero di sviluppatori proporzionale alla dimensione del progetto. Soprattutto in team di sviluppo di notevoli dimensioni, un requisito fondamentale delle varie metodologie Agili è la conoscenza, da parte di tutti i componenti del team, delle mansioni assegnate ad ogni altro collega sviluppatore. In altre parole, dunque, occorre ci sia molta "trasparenza" in merito ai compiti assegnati all'interno del team di sviluppo;
- **Iteratività.** Un altro concetto fondamentale delle metodologie Agili è l'iteratività. Al termine di ogni iterazione, infatti, è possibile inviare l'ultima "release" del progetto software al cliente in modo da conoscere immediatamente le sue valutazioni. In base al "feedback" ricevuto dal team di sviluppo, sarà possibile modificare immediatamente il software in modo da assecondare il pensiero del cliente.
- **Auto-organizzazione.** Ogni team di sviluppo che utilizza metodologie Agili, inoltre, ha la facoltà di organizzare autonomamente il proprio lavoro. I vari sviluppatori, infatti, possono decidere autonomamente come suddividere le varie mansioni all'interno del team, senza bisogno di alcun intervento "esterno".

1.2 MANIFESTO PER LO SVILUPPO AGILE DI SOFTWARE

Per garantire la piena soddisfazione del cliente ed abbattere i costi dello sviluppo software, i più importanti ricercatori a livello mondiale si radunarono, attorno all'anno 2001, in una località sciistica dello Utah e decisero di creare l'associazione denominata "Agile Alliance". Il loro obiettivo era quello di delineare una serie di valori e principi che avrebbero permesso ai vari team di sviluppo di lavorare molto più rapidamente. Per questo motivo, nei mesi successivi, pubblicarono il seguente manifesto come riportato anche dal sito ufficiale[6]:

Manifesto per lo sviluppo Agile di software

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

Gli individui e le interazioni più che i processi e gli strumenti

Il software funzionante più che la documentazione esaustiva

La collaborazione col cliente più che la negoziazione dei contratti

Rispondere al cambiamento più che seguire un piano

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

Firmatari:

Kent Beck	James Grenning	Robert C. Martin	Mike Beedle
Jim Highsmith	Steve Mellor	Arie van Bennekum	Andrew Hunt
Ken Schwaber	Alistair Cockburn	Ron Jeffries	Jeff Sutherland
Ward Cunningham	Jon Kern	Dave Thomas	Martin Fowler
Brian Marick			

Andiamo ora ad analizzare i quattro punti centrali del manifesto.

1.2.1 GLI INDIVIDUI E LE INTERAZIONI PIÙ CHE I PROCESSI E GLI STRUMENTI

Per essere un ottimo programmatore non occorre solo il talento, ma bisogna saper anche lavorare bene, comunicare ed interagire con gli altri membri del team di sviluppo. L'interazione tra i vari componenti del team di sviluppo, infatti, è fondamentale ed anche più importante del puro talento individuale. Un team di sviluppo composto da persone che comunicano ed interagiscono tra di loro ha maggiori probabilità di successo rispetto ad un team di sviluppo composto da "superstar" che non riescono ad interagire l'uno con l'altro.

1.2.2 IL SOFTWARE FUNZIONANTE PIÙ CHE LA DOCUMENTAZIONE ESAUSTIVA

La stesura della documentazione relativa ai metodi ed alle classi presenti all'interno del codice ha un ruolo fondamentale per il corretto sviluppo di un progetto software. Tale documentazione, infatti, permette di comprendere al meglio la logica e la struttura di un sistema.

È sempre una buona idea scrivere la documentazione all'interno del codice, ma deve essere il più sintetica possibile. Avere troppa documentazione, infatti, è uno svantaggio per il team di sviluppo perché richiede molto tempo sia per essere realizzata che per essere mantenuta in sincronia con il codice.

1.2.3 LA COLLABORAZIONE COL CLIENTE PIÙ CHE LA NEGOZIAZIONE DEI CONTRATTI

Un altro aspetto centrale del manifesto è il rapporto tra il cliente ed il programmatore riguardo sia alla stesura dei contratti che alla loro collaborazione. Se il cliente chiede al programmatore di creare un progetto software in un determinato periodo di tempo, ad un prezzo fisso e con le funzionalità descritte in breve sul contratto, allora tale software fallirà inevitabilmente. Secondo i membri dell'Agile Alliance, infatti, il cliente deve collaborare attivamente con i membri del team di sviluppo poiché occorre comprendere bene quali siano le esigenze della società committente tramite frequenti e regolari valutazioni delle ultime "release" del software. L'interazione tra il cliente ed il team di sviluppo è

molto più importante della stesura del contratto. I termini e le specifiche dei vari contratti, infatti, diventano inutili molto prima che il software sia completato.

1.2.4 *RISPONDERE AL CAMBIAMENTO PIÙ CHE SEGUIRE UN PIANO*

All'interno dell'azienda committente si possono verificare continui cambiamenti di natura tecnologica e di business. Il progetto software che il team di sviluppo sta realizzando, dunque, deve adeguarsi il più fedelmente possibile alla nuova realtà. Il mancato adattamento o meno di tale software alle nuove esigenze aziendali decreta spesso il suo successo o fallimento. Lo sviluppo di un software non può essere interamente progettato per un lungo periodo di tempo poiché l'ambiente aziendale è destinato a cambiare ed i clienti possono alterare i requisiti del software anche in seguito al corretto funzionamento del sistema. Una buona strategia di pianificazione per lo sviluppo di un progetto software consiste nel creare piani di sviluppo dettagliati per le prossime due settimane, piani di sviluppo approssimativi per i successivi tre mesi e piani di sviluppo estremamente "grezzi" per periodi di tempo oltre i 3 mesi. In questo modo, il team di sviluppo potrà conoscere esattamente le esigenze software necessarie nelle prossime due settimane e, contemporaneamente, conoscere anche i requisiti necessari per lo sviluppo del software nel prossimo trimestre.

1.3 *PRINCÌPI DEL MANIFESTO*

All'interno del manifesto è possibile riconoscere 12 elementi chiave (o semplicemente "principi") che permettono di distinguere le tecniche Agili dalle comuni tecniche di progettazione. Procediamo ora alla definizione di tali elementi chiave, come riporta anche il sito ufficiale del manifesto per lo sviluppo Agile[6], e ad una breve descrizione:

- **" La nostra massima priorità è soddisfare il cliente, rilasciando software di valore, fin da subito e in maniera continua".** È possibile individuare una serie di pratiche che permettono di aiutare le aziende nella creazione di prodotti di alta qualità. Tali pratiche apportano un significativo miglioramento nella qualità del software che si sta sviluppando. Una di esse consiste nel possedere una forte correlazione tra la qualità e la consegna anticipata di un software parzialmente funzionante. Un articolo del MIT Sloan Management Review[7], infatti, sostiene che minore è il numero di funzionalità contenute nel software consegnato inizialmente al cliente, maggiore sarà la qualità del software finale. Un altro elemento fondamentale risiede nella correlazione tra la qualità finale del software prodotto ed il numero di domande che sono state effettuate tra il cliente ed il

programmatore. Maggiori saranno le domande poste, maggiore sarà la qualità del software finale. Naturalmente, dopo la consegna finale del software, occorre fornire al cliente anche opportuni aggiornamenti software. In tal senso, il cliente può decidere di fermare lo sviluppo e far partire la produzione dei prodotti utilizzando l'ultima "release" del progetto software, oppure può attendere ulteriori aggiornamenti poiché le funzionalità richieste non sono perfettamente funzionanti.

- **"Accogliamo i cambiamenti nei requisiti, anche a stadi avanzati dello sviluppo. I processi Agili sfruttano il cambiamento a favore del vantaggio competitivo del cliente"**. Un ulteriore punto chiave del manifesto è l'atteggiamento dei membri del team di sviluppo. Le possibili modifiche apportate ai requisiti del software che si sta realizzando, infatti, devono essere considerate positivamente perché permettono al team di comprendere meglio le esigenze del mercato nei confronti del software che si sta realizzando. Grazie ad una struttura flessibile ai cambiamenti, infatti, è possibile modificare i requisiti software facendo in modo che l'impatto di tali modifiche sul sistema sia minimo.
- **"Consegniamo frequentemente software funzionante, con cadenza variabile da un paio di settimane a un paio di mesi, preferendo i periodi brevi"**. Come sostenuto anche da Robert Martin[10], ogni "release" del progetto software deve essere consegnata entro le prime due settimane iniziali e, all'incirca, ogni due settimane successive. Tali periodi di tempo variano da progetto a progetto, ma è preferibile consegnare il software ogni due settimane piuttosto che, ad esempio, ogni due mesi.
- **"Committenti e sviluppatori devono lavorare insieme quotidianamente per tutta la durata del progetto"**. Affinchè un progetto possa essere considerato Agile occorre una significativa interazione tra clienti e programmatori. Il progetto software, infatti, deve essere continuamente "guidato" sia dal cliente che dai membri del team di sviluppo.
- **"Fondiamo i progetti su individui motivati. Diamo loro l'ambiente e il supporto di cui hanno bisogno e confidiamo nella loro capacità di portare il lavoro a termine"**. In un progetto Agile le persone sono considerate il più importante fattore di successo. Tutti gli altri fattori, come ad esempio l'ambiente e la gestione delle risorse, ... sono considerati di secondo piano e devono essere opportunamente modificati se provocano effetti negativi sugli individui.
- **"Una conversazione faccia a faccia è il modo più efficiente ed efficace per comunicare sia con il team sia all'interno del team"**. Il metodo più efficace per comunicare, sia all'interno che all'esterno del team di sviluppo, è la comunicazione verbale. In questo modo, infatti, vengono semplificate

notevolmente le mansioni dei componenti del team di sviluppo perché non vengono richieste particolari specifiche scritte da parte del cliente o di altri membri del team.

- **"Il software funzionante è il principale metro di misura del progresso".** I progetti software creati tramite metodologie Agili vengono valutati in base al software funzionante che incontra le esigenze del cliente.
- **"I processi Agili promuovono uno sviluppo sostenibile. Gli sponsor, gli sviluppatori e gli utenti dovrebbero essere in grado di mantenere indefinitamente un ritmo costante".** I componenti del team di sviluppo devono lavorare ad un ritmo costante in modo da mantenere i più alti standard di qualità per tutta la durata del progetto. La velocità con cui i componenti del team di sviluppo hanno iniziato a sviluppare il software non deve essere nè maggiore nè minore rispetto alla velocità finale.
- **"La continua attenzione all'eccellenza tecnica ed alla buona progettazione esaltano l'agilità".** Per ottenere una buona e costante velocità di programmazione occorre mantenere il più pulito e robusto possibile il software che si sta sviluppando. In questo modo, tutti i membri del team di sviluppo produrranno solo codice di alta qualità e non sarà necessaria nessuna operazione di "ripulitura" (o refactoring) del codice.
- **"La semplicità - l'arte di massimizzare la quantità di lavoro non svolto - è essenziale".** I team di sviluppo che utilizzano metodologie Agili devono seguire sempre il percorso più semplice e consistente con il proprio obiettivo per realizzare il progetto software. Non occorre, infatti, anticipare i problemi che si possono verificare in futuro, bensì occorre lavorare il più semplice possibile mantenendo sempre alta la qualità del progetto software.
- **"Le architetture, i requisiti e le progettazioni migliori emergono da team che si auto-organizzano".** Come già sostenuto precedentemente, ogni team di sviluppo è dotato di una "auto-organizzazione" delle mansioni. In altre parole, le mansioni all'interno del team non vengono stabilite da persone esterne alla squadra, bensì dai membri stessi del team. All'interno di tali team di sviluppo software, inoltre, riveste un ruolo fondamentale anche l'interoperabilità degli sviluppatori. Ad ogni membro del team, infatti, deve essere permesso di intervenire su qualunque aspetto del progetto, anche se la relativa mansione non è stata assegnata a lui. Così facendo, dunque, il team di sviluppo condivide le responsabilità del progetto software tra tutti i suoi componenti.

- **"Ad intervalli regolari il team riflette su come diventare più efficace, dopodiché regola ed adatta il proprio comportamento di conseguenza".** Ogni team di sviluppo gestisce opportunamente la propria organizzazione, le proprie regole e le proprie convenzioni. Ad ogni cambiamento dell'ambiente software circostante, dunque, potrà corrispondere un cambiamento nella corrispondente organizzazione, nelle regole o nelle convenzioni stipulate.

1.4 TIPI DI METODOLOGIE AGILI

Alcune fra le più note metodologie Agili hanno incluso i principi sopra esposti in una vera e propria struttura organizzativa per il team, enfatizzando l'uno o l'altro di questi principi. Nel seguito, ricordiamo brevemente alcune di queste metodologie; la prima metodologia esaminata è eXtreme Programming che è, notoriamente, anche quella più diffusa negli ambienti di sviluppo software avanzati.

1.4.1 *EXTREME PROGRAMMING*

Extreme Programming (XP) è una metodologia di sviluppo software che ha lo scopo di migliorare la qualità del software anche in presenza di mutevoli esigenze del committente.

1.4.1.1 *Valori*

Come sostenuto anche da Kent Beck[11], per poter sviluppare correttamente un progetto software, i componenti del team di sviluppo devono seguire i seguenti valori:

- Comunicazione;
- Semplicità;
- Feedback;
- Coraggio;
- Rispetto.

COMUNICAZIONE

L'ideazione e la costruzione di un sistema software richiede la definizione di una serie di requisiti di sistema che poi saranno fondamentali per la creazione effettiva del software. Spesso tali requisiti vengono comunicati tramite apposita documentazione. Se si utilizza la metodologia di eXtreme Programming, tuttavia, è opportuno adottare particolari tecniche al fine di definire rapidamente i requisiti software e, al contempo, renderne partecipi tutti i membri della squadra di sviluppo tramite, ad esempio, la creazione di alcuni diagrammi UML. È importante, dunque, che ci sia comunicazione tra i vari componenti del team di sviluppo.

SEMPLICITÀ

Un altro valore fondamentale dell'eXtreme Programming è la semplicità. Tale metodologia, infatti, incoraggia il programmatore ad iniziare lo sviluppo dalle funzionalità più semplici e, solo in seguito, aggiungere le funzionalità più complesse richieste dal committente. Occorre concentrarsi su design e codice attuali, non su quelli futuri. Non bisogna impiegare tempo e risorse per funzionalità che potrebbero anche non essere necessarie in futuro.

FEEDBACK

Un altro elemento fondamentale è il feedback. In particolare, vengono distinti i seguenti tipi di feedback:

- Feedback dal sistema;
- Feedback dal cliente;
- Feedback dal team.

Si parla di "feedback dal sistema" quando i membri di un team di sviluppo, scrivendo ed eseguendo le opportune classi di test, ottengono, minuto per minuto, i feedback sullo stato del sistema. Il "feedback dal cliente", invece, si ottiene quando il cliente inizia a valutare adeguatamente l'ultima release del progetto software. Così facendo, i membri del team di sviluppo ricevono immediati feedback sulla qualità delle funzionalità implementate e, in caso di bug o errori, possono immediatamente intervenire. Infine, il "feedback dal team". È possibile, infatti, che il cliente modifichi, ad un certo punto dello sviluppo, alcuni requisiti del progetto software. I membri del team di sviluppo dovranno determinare una stima del tempo di realizzazione di tali modifiche ed inviarla al cliente. Tale stima rappresenta il feedback che il cliente riceve.

CORAGGIO

Un altro valore fondamentale nell'eXtreme Programming è il coraggio. Può succedere, infatti, di dover eliminare alcune porzioni di codice obsoleto, ma funzionante, per cercare di ottenere un software più flessibile. Ci vuole coraggio anche nel prendersi la responsabilità, ad esempio, di portare a termine un task o nel chiedere aiuto quando si rimane bloccati per molto tempo su uno specifico problema.

RISPETTO

Infine, il rispetto. Ogni volta che un membro del team di sviluppo vuole introdurre una modifica all'interno del software, occorre eseguire tutti gli "unit test" che sono stati creati, in modo da rispettare e non rovinare il lavoro svolto dal team di sviluppo.

1.4.1.2 Linee-guida

Come sottolineato anche da Kent Beck[12], in base ai cinque valori descritti precedentemente, è possibile identificare, all'interno della metodologia di eXtreme Programming, le seguenti linee-guida che devono caratterizzare il lavoro svolto dal team:

- **Rapidi feedback.** Nel corso dello sviluppo di un progetto software, che utilizza la metodologia di eXtreme Programming, è opportuno che i vari componenti del team di sviluppo interagiscano opportunamente con il cliente. Quest'ultimo, infatti, deve poter valutare adeguatamente il progetto software al fine di aiutare il team di sviluppo nella realizzazione del programma. Al termine di ogni iterazione, dunque, è opportuno che gli sviluppatori inviino l'ultima "release" al cliente. La rispettiva valutazione del cliente deve essere inviata rapidamente al team di sviluppo in modo da permettere, ove necessario, di modificare immediatamente il progetto software secondo le direttive del cliente.
- **Assunzione di semplicità.** All'interno dello sviluppo del progetto software si possono verificare numerosi bug o malfunzionamenti, ma occorre che gli sviluppatori che lavorano al progetto trattino con semplicità tali problematiche.

- **Cambiamenti incrementali.** I membri del team di sviluppo non devono effettuare numerosi cambiamenti all'interno del codice, ma realizzare piccole modifiche incrementali;
- **Abbracciare i cambiamenti.** Un'altra linea-guida della metodologia di eXtreme Programming riguarda l'atteggiamento dei componenti del team di sviluppo. Anche se il software è già pienamente funzionante, infatti, gli sviluppatori non devono temere di effettuare alcune modifiche o introdurre alcuni cambiamenti all'interno del codice.
- **Lavoro di qualità.** La principale caratteristica del software che il team di sviluppo vuole realizzare, tramite la metodologia di eXtreme Programming, deve essere la qualità. Un lavoro di qualità, infatti, oltre a rendere felice il cliente, permette agli sviluppatori di effettuare eventuali modifiche future in un breve periodo di tempo e ad un basso costo.

1.4.1.3 Attività di base e pratiche

Extreme Programming distingue le seguenti attività di base:

- **Coding** : all'interno di un progetto software, il codice è la parte fondamentale dello sviluppo poiché permette concretamente di creare il software e, al contempo, permette ai vari sviluppatori di comprendere esattamente il sistema su cui si sta lavorando;
- **Testing** : un'altra attività fondamentale all'interno dello sviluppo software è il testing. Il corretto funzionamento del sistema, infatti, non può essere adeguatamente verificato fino all'effettiva esecuzione dell'attività di testing;
- **Listening** : durante lo sviluppo del progetto software occorre analizzare le effettive necessità del cliente e stabilire l'importanza delle varie funzionalità che occorre implementare. Per permettere un'adeguata comunicazione (o "listening") tra il cliente ed il team di sviluppo, è possibile utilizzare la pratica denominata "Planning game" che verrà descritta in seguito;
- **Designing** : per realizzare adeguatamente un progetto software, tramite la metodologia Agile di eXtreme Programming, è opportuno anche effettuare un'adeguata attività di "designing". In altre parole, occorre strutturare adeguatamente la logica del sistema che si vuole sviluppare.

Come sostenuto anche da Kent Beck[12], tali attività (coding, testing, listening e designing) possono essere strutturate ulteriormente nelle seguenti pratiche:

- **Planning game.** Come già anticipato precedentemente, la pratica di sviluppo denominata "Planning Game" permette un'adeguata attività di comunicazione tra il cliente ed il team di sviluppo. Al termine di ogni iterazione, infatti, occorre confrontare adeguatamente le priorità di business del cliente con le relative tempistiche del team di sviluppo;
- **Small releases.** Occorre rilasciare piccole e costanti "release" del software nel più breve tempo possibile in modo che il cliente possa valutare opportunamente i progressi dello sviluppo del progetto software. Così facendo, infatti, i componenti del team di sviluppo potranno analizzare il feedback rilasciato dal cliente e trarre le opportune valutazioni;
- **Metaphor.** Lo sviluppo del progetto software può essere facilitato utilizzando la pratica di sviluppo denominata "Metaphor". Come è facilmente intuibile, tale pratica prevede la descrizione del funzionamento del sistema tramite una semplice storia (o metafora).
- **Simple design.** All'interno dello sviluppo di un progetto software occorre semplificare notevolmente il design relativo al software che si sta realizzando tramite, ad esempio, l'eliminazione di eventuali duplicati oppure la creazione di classi di piccole dimensioni. Così facendo, infatti, miglioreremo sia la leggibilità che la comprensione del codice;
- **Test-Driven Development.** La fase di testing è fondamentale nello sviluppo di un progetto software. I vari componenti del team di sviluppo, infatti, devono poter testare continuamente il codice realizzato al fine di verificarne il corretto funzionamento. Tale pratica verrà adeguatamente analizzata all'interno del capitolo 2;
- **Refactoring.** Tramite la pratica di sviluppo denominata "Refactoring" è possibile "ristrutturare" il sistema, ossia modificare alcune porzioni del codice senza, tuttavia, alterare il comportamento del software. Tale pratica verrà descritta con maggiore attenzione all'interno del capitolo 3;
- **Pair Programming.** Il "Pair Programming" è la principale caratteristica della metodologia di eXtreme Programming e prevede che due sviluppatori lavorino insieme sulla stessa workstation. Tipicamente, come sostiene anche Kent Beck[12], un programmatore lavora alla tastiera, mentre l'altro si occupa dell'approccio da adottare. Il principale beneficio derivante dall'utilizzo di tale pratica consiste nell'incremento della qualità del codice a patto che, naturalmente, gli sviluppatori in questione possiedano la stessa esperienza di programmazione.
- **Collective ownership.** L'intera responsabilità del progetto software deve essere suddivisa tra tutti i componenti del team di sviluppo. Così facendo,

infatti, tutti i vari programmatori del team di sviluppo hanno la facoltà di modificare anche le porzioni di codice non implementate da loro stessi;

- **Continuos integration.** Le modifiche apportate al codice devono essere continuamente integrate. Così facendo, infatti, eviteremo il verificarsi di eventuali ritardi futuri;
- **40 hour week,** ossia una pratica di sviluppo in base alla quale i membri del team non devono lavorare più di 40 ore a settimana;
- **On-site customer.** Il cliente deve avere sempre un ruolo attivo nello sviluppo del progetto software. Egli, infatti, deve poter rispondere in modo tempestivo ed adeguato alle domande effettuate dai componenti del team di sviluppo;
- **Coding standards.** I vari membri del team di sviluppo devono seguire un preciso standard di scrittura del codice durante l'intero processo di sviluppo software. Occorre, dunque, stabilire alcune regole da rispettare per garantire il corretto funzionamento del progetto software.

1.4.1.4 *Rischi*

I principali rischi che si potrebbero verificare quando si sviluppa un progetto software sono:

- **Errate pianificazioni** : è possibile che il software su cui il team di sviluppo sta lavorando non sia pronto per la data inizialmente prevista per il suo completamento;
- **Progetto cancellato** : se si verificano numerose errate pianificazioni, allora il progetto software può essere definitivamente cancellato;
- **Sistema inutilizzabile** : dopo pochi anni dall'utilizzo di un sistema software, è possibile che i costi delle modifiche ed il tasso degli errori siano talmente alti da provocarne una sostituzione;
- **Elevato tasso di errori** : può succedere che un sistema software sia stato messo in produzione sebbene il tasso di rilevazione degli errori sia talmente alto da impedirne un efficace utilizzo;
- **Impossibilità di risolvere i problemi** : il sistema è in produzione, ma i problemi per cui era stato richiesto non sono stati risolti;
- **Cambiamenti** : il sistema potrebbe essere stato messo in produzione ed aver risolto i problemi per cui era stato espressamente richiesto dal cliente, ma purtroppo potrebbe accadere che i suddetti problemi siano stati sostituiti, negli ultimi mesi, da problemi più stringenti;

- **Funzionalità interessanti ma non richieste** : sebbene il software presenti grandi potenzialità e molte funzionalità interessanti, nessuna di queste potrebbe essere stata espressamente richiesta dal committente;
- **Rotazione del personale** : è possibile, infine, che dopo alcuni anni dall'inizio dello sviluppo di un progetto software, alcuni programmatori decidano di abbandonare lo sviluppo.

ERRATE PIANIFICAZIONI

Durante lo sviluppo di un progetto software si possono verificare alcuni erronee pianificazioni circa la consegna ultima del prodotto finale. Se si utilizza la metodologia di eXtreme Programming, tuttavia, tali pianificazioni non arrecheranno notevole danno ai membri del team di sviluppo. Le iterazioni del progetto software, infatti, vengono opportunamente pianificate dai programmatori e, ad ognuna di esse, viene assegnata una certa priorità. Inoltre, se le iterazioni sono buone, allora si cercherà sempre più di diminuirne il tempo di esecuzione.

PROGETTO CANCELLATO

Nel caso si verifichino numerose errate pianificazioni, è probabile che il progetto software venga definitivamente cancellato. La metodologia di eXtreme Programming, tuttavia, permette di definire inizialmente un insieme di funzionalità che permetta agli sviluppatori di iniziare a lavorare correttamente sul progetto software. Ogni successiva "release" del software rappresenta un piccolo incremento di tali funzionalità. Un ulteriore obiettivo dei componenti del team di sviluppo, dunque, è minimizzare il tempo che intercorre tra le varie iterazioni. Iterazione dopo iterazione, il progetto software viene ancor più raffinato e completato.

SISTEMA INUTILIZZABILE

Dopo pochi anni dall'utilizzo di un sistema software, è possibile che i costi delle modifiche ed il tasso degli errori siano talmente alti da provocarne una sostituzione. Utilizzando la metodologia di eXtreme Programming, tuttavia, è possibile mantenere sempre un alto livello nella qualità del software. I membri del team di sviluppo che utilizzano la metodologia di XP, infatti, devono provvedere a creare un insieme di test automatici che saranno eseguiti al termine di ogni cambiamento apportato al software.

ELEVATO TASSO DI ERRORI

È possibile, inoltre, che il software sia stato messo in produzione, ma il tasso di rilevazione degli errori sia talmente alto da impedirne un efficace utilizzo. Se si utilizza la metodologia di XP, tuttavia, al termine di ogni release, i membri del team di sviluppo software devono eseguire un insieme di test, funzionali e non, che permettano di verificare la corretta esecuzione del programma. I test funzionali sono definiti direttamente dal cliente e si occupano di testare le funzionalità implementate all'interno del software. Gli sviluppatori potranno, dunque, ottenere le immediate valutazioni del cliente in merito al software realizzato.

IMPOSSIBILITÀ DI RISOLVERE I PROBLEMI

Si potrebbe, inoltre, verificare la situazione in cui il software sia stato messo in produzione anche se non sono stati risolti i problemi per cui era stato espressamente richiesto dal cliente. Extreme Programming, tuttavia, prevede che il cliente partecipi attivamente allo sviluppo del progetto software e, dunque, il cliente può rendere noto il mancato risolvimento del problema quando ancora il software non è stato interamente realizzato.

CAMBIAMENTI

Quando si sviluppa un progetto software può anche succedere che il sistema sia già in produzione e che abbia risolto tutti i problemi per i quali era stato espressamente richiesto dal cliente. Nel frattempo, tuttavia, è possibile che tali problematiche fossero state sostituite da altre più importanti. Tale situazione di rischio può essere risolta se si utilizza la metodologia di eXtreme Programming; in tal caso, infatti, si rilasciano, iterazione dopo iterazione, nuove "release" che implementano nuove funzionalità del software o modificano quelle già esistenti. Eseguendo brevi iterazioni, è più semplice gestire i cambiamenti ed evitare questa situazione di rischio.

FUNZIONALITÀ INTERESSANTI MA NON RICHIESTE

Il software potrebbe avere grandi potenzialità e funzionalità molto interessanti; tuttavia, nessuna di esse potrebbe essere stata espressamente richiesta dal cliente. Ciò non sarebbe accaduto se il team di sviluppo avesse utilizzato la metodologia di XP per sviluppare il progetto software. In tal caso, infatti, sarebbero state realizzate solo le funzionalità prioritarie.

ROTAZIONE DEL PERSONALE

Dopo alcuni anni di sviluppo di un progetto software, può accadere che alcuni programmatori decidano di lasciare il team di sviluppo. Con l'utilizzo di eXtreme Programming, tuttavia, si riesce a ridimensionare tale rischio, incoraggiando il contatto umano tra i vari membri del team e riducendo, così, le eventuali sensazioni di solitudine ed insoddisfazione.

1.4.2 SCRUM

Poiché possiamo dire che il Pair Programming è la caratteristica principale della metodologia di eXtreme Programming, ecco le caratteristiche fondamentali del framework Scrum:

- **Trasparenza.** All'interno dello sviluppo di un progetto software occorre definire e, rendere visibili a tutti i componenti del team di sviluppo, un comune standard di scrittura del codice.
- **Ispezione.** Talvolta, durante lo sviluppo di un progetto software, occorre anche ispezionare gli artefatti di Scrum al fine di verificare il processo di completamento degli obiettivi prefissati. Naturalmente, tali ispezioni devono essere eseguite solo da coloro che sono in possesso di tutti i requisiti necessari e non devono essere eseguite troppo frequentemente. In caso contrario, infatti, si intralchierebbe il lavoro di sviluppo del progetto software.
- **Adattamento.** Se durante un'ispezione è stato verificato che alcuni aspetti del processo di sviluppo software non sono accettabili, allora occorre l'intervento dei membri del team di sviluppo per adattare lo sviluppo del processo software agli obiettivi prefissati.

1.4.2.1 Scrum Team

All'interno di Scrum è possibile individuare i seguenti ruoli:

- **il team di sviluppo**, che si occupa di realizzare il progetto software;
- **il product owner**, che si occupa di massimizzare il valore del progetto software

- lo **"Scrum Master"**, che ha un ruolo di tipo manageriale e si occupa, principalmente, di tenere concentrato il team di sviluppo sulla realizzazione degli obiettivi prefissati, fungendo da "cuscinetto" verso qualsiasi possibile influenza esterna.

Quando si fa riferimento a tutti e tre i ruoli allora si parla di "Scrum Team".

1.4.2.2 *Eventi*

Come sostengono anche Ken Schwaber & Jeff Sutherland[13], gli eventi, in Scrum, "sono utilizzati per creare regolarità e ridurre al minimo la necessità di effettuare riunioni non previste". Inoltre, tutti gli eventi in Scrum sono "time-box", ossia sono dotati di una durata massima. Così facendo, infatti, si assicura che un'appropriata quantità di tempo verrà trascorsa pianificando evitando, dunque, l'introduzione di sprechi nel processo di pianificazione.

Nella seguente sezione analizzeremo Sprint, ossia un insieme di tutti i principali eventi contenuti in Scrum.

SPRINT

Ogni Sprint è caratterizzato da tre elementi principali:

- una definizione di ciò che si deve realizzare;
- un progetto;
- un piano flessibile.

Se devono essere eseguiti più Sprint all'interno dello sviluppo di un progetto software, allora occorre attendere la terminazione di uno Sprint prima di eseguire gli altri. Non è possibile, infatti, eseguire più Sprint contemporaneamente. Ken Schwaber & Jeff Sutherland[13], infatti, sostengono che "un nuovo Sprint si avvia immediatamente dopo la conclusione dello Sprint precedente".

I principali eventi contenuti in uno Sprint sono:

- lo **Sprint Planning**, ossia un evento a cui possono partecipare tutti i membri dello "Scrum Team" avente l'obiettivo di preparare i componenti del team di sviluppo e comunicare la quantità di lavoro che sarà effettuata durante l'attuale sprint;
- il **Daily Scrum**, ossia un evento giornaliero avente lo scopo di pianificare ed organizzare al meglio le attività da effettuare nelle prossime 24 ore;
- il **lavoro di sviluppo**;

- la **Sprint Review**, ossia un incontro informale per raccogliere commenti e promuovere la collaborazione all'interno dello "Scrum Team";
- la **Sprint Retrospective**, ossia un evento tramite il quale lo "Scrum Team" ha la facoltà di ispezionare il lavoro effettuato fino a quel momento e pianificare il lavoro da effettuare durante il prossimo Sprint.

1.4.2.3 Artefatti

Gli artefatti, infine, sono elementi di Scrum creati con lo scopo di massimizzare la trasparenza delle informazioni ed effettuare adeguate attività di ispezione ed adattamento durante lo sviluppo di un progetto software.

I principali artefatti si distinguono in:

- **Product Backlog**, ossia un insieme di tutti i requisiti e le informazioni necessarie per il corretto funzionamento del progetto software;
- **Sprint Backlog**, ossia un insieme degli elementi che fanno parte del Product Backlog e di un piano di sviluppo avente lo scopo di aumentare il valore del progetto software.

1.4.3 DYNAMIC SYSTEMS DEVELOPMENT METHOD

La metodologia Agile denominata Dynamic Systems Development Method (DSDM), invece, è particolarmente utile quando, in un breve lasso di tempo, si vogliono sviluppare sistemi software in cui i requisiti possano essere modificati durante lo sviluppo del progetto software. In DSDM, infatti, le fasi di analisi, progettazione e sviluppo possono sovrapporsi durante la creazione del software. Come sostiene anche Jennifer Stapleton[16], ecco alcuni concetti fondamentali di tale metodologia:

- **Attivo coinvolgimento degli utenti**: il cliente deve essere attivamente coinvolto nello sviluppo del progetto software;
- **Potere decisionale al team di sviluppo**: il team di sviluppo deve poter prendere rapide ed efficaci decisioni all'interno dello sviluppo del progetto software;
- **Frequenti "release"**: i componenti del team di sviluppo devono rilasciare regolari e frequenti aggiornamenti software (chiamati anche "release");
- **Soddisfare le esigenze aziendali**: l'obiettivo principale del team di sviluppo è soddisfare le esigenze aziendali, anche a scapito della perfezione tecnica;

- **Sviluppo iterativo guidato dai feedback:** lo sviluppo software di tipo iterativo permette al team di sviluppo di conoscere frequentemente le valutazioni dell'utente riguardo alle funzionalità implementate nelle ultime "release" del software;
- **Le modifiche devono essere reversibili:** tutti gli oggetti creati all'interno del codice devono essere in uno stato noto in ogni momento dello sviluppo software. Se, infatti, una modifica non produce gli effetti sperati, allora deve essere sempre possibile effettuare il "backtracking";
- **I requisiti sono definiti ad alto livello:** i requisiti essenziali per un adeguato ed efficace sviluppo software devono essere definiti nella fase iniziale del processo. Il cliente, tuttavia, deve poter modificare opportunamente alcuni particolari durante lo sviluppo del software;
- **Test integrati:** ad ogni iterazione devono essere eseguite le classi di test in modo da garantire sempre la validità e la correttezza del codice;
- **Collaborazione e cooperazione:** la collaborazione e la cooperazione tra tutte le parti interessate alla creazione del progetto software sono fondamentali per il raggiungimento degli obiettivi aziendali;

1.4.4.1 Processo di Sviluppo

Il framework DSDM si compone delle seguenti fasi:

- **The pre-project:** in questa fase si provvede ad identificare il progetto che permetterà di realizzare il software richiesto dal cliente;
- **Ciclo di vita del progetto:** questa fase comprende, invece, la gestione del ciclo di vita del progetto software. Essa si suddivide nelle seguenti cinque sottofasi:
 - **Studio di fattibilità:** in base al tipo di progetto, all'organizzazione ed alle persone coinvolte si valuta l'idoneità ad utilizzare o meno il framework DSDM;
 - **Studio del business:** coinvolgendo un notevole gruppo di esperti si analizzano le caratteristiche essenziali del business e della tecnologia da utilizzare;
 - **Modello funzionale:** i requisiti che sono stati identificati nelle fasi precedenti vengono convertiti in un modello funzionale sviluppando un prototipo funzionante ed un relativo modello. Il prototipo è fondamentale poiché consente il coinvolgimento dell'utente durante il progetto;

- **Progettazione e costruzione:** L'obiettivo principale di questa iterazione è integrare i componenti funzionali, identificati precedentemente, in un sistema che soddisfi le esigenze del cliente;
 - **Implementazione:** il sistema e la relativa documentazione vengono consegnati al cliente che deve approvare o meno il software in relazione all'attuazione delle linee guida fissate precedentemente. In seguito, occorre procedere all'implementazione effettiva del software ed alla verifica degli obiettivi fissati precedentemente.
- **The post-project:** tale fase garantisce che il sistema funzioni in modo efficace ed efficiente attraverso varie operazioni di manutenzione, miglioramento e correzione di eventuali errori.

1.4.4 CRYSTAL

Il principale obiettivo delle metodologie Crystal, create da Alistair Cockburn[17], è il potenziamento del lavoro effettuato dai componenti del team di sviluppo. Esistono, tuttavia, varie tipologie di progetti software e, dunque, occorre adottare differenti strategie per potenziare effettivamente il lavoro svolto dal team di sviluppo.

Nella sezione seguente esamineremo la più importante metodologia della famiglia Crystal, ovvero Crystal Clear.

1.4.5.1 Crystal Clear

Crystal Clear è una metodologia Agile, facente parte della famiglia Crystal, che può essere utilizzata da tutti i team di sviluppo software composti da un minimo di due persone fino ad massimo di otto. I primi passi della metodologia Crystal Clear prevedono l'analisi dei punti di forza e di debolezza del progetto software in modo da massimizzare i vantaggi e minimizzare gli svantaggi del progetto software. Ecco le principali proprietà della metodologia Crystal Clear:

1. **Frequenti "release" del software :** in base alla lunghezza del progetto, occorre rilasciare frequentemente (da un minimo di sette giorni fino ad un massimo di tre mesi tra ogni "release") le iterazioni del progetto software. Così facendo, infatti, il cliente sarà in grado di individuare gli eventuali difetti o malfunzionamenti nella fase cruciale dello sviluppo software;
2. **Riflessioni sulle possibili correzioni apportabili al progetto software :** bisettimanalmente occorre eseguire alcune riunioni tra i vari membri del

team di sviluppo per riflettere sulle idee che possano migliorare lo sviluppo del progetto software;

3. **Comunicazione osmotica:** i componenti del team di sviluppo devono raccogliere tramite "osmosi" le informazioni sullo sviluppo del progetto software. In altre parole, occorre effettuare opportune riunioni per discutere e confrontarsi sullo sviluppo del progetto software;
4. **Fiducia:** occorre parlare liberamente all'interno del team di sviluppo quando non si è d'accordo su alcune decisioni. Solo in questo modo, infatti, il team di sviluppo potrà scoprire le debolezze del progetto software ed agire conseguentemente. La presenza della fiducia all'interno di ogni persona del team di sviluppo è positivamente correlata con la prestazione del software che si sta realizzando;
5. **Focus:** occorre che i componenti del team di sviluppo conoscano la materia su cui stanno lavorando e, soprattutto, abbiano il tempo e la necessaria "tranquillità" per sviluppare il progetto software.
6. **Utilizzo degli esperti in materia:** occorre che il team di sviluppo collabori attivamente con una persona di competenza nell'area del progetto che si sta realizzando per risolvere i possibili problemi che possono sorgere nello sviluppo del software;
7. **Test automatici ed operazioni di integrazione del codice:** il team di sviluppo software deve effettuare frequenti e regolari operazioni di integrazione del codice ed attività di testing. Se si apportano modifiche al progetto software e vengono rilevati alcuni errori o malfunzionamenti, allora occorre rilevare immediatamente le eventuali cause ed effettuare opportune operazioni di "backtracking".

Crystal Clear, in particolare, richiede obbligatoriamente che siano soddisfatte le prime tre proprietà, mentre le altre quattro possono essere utilizzate dai migliori team di sviluppo software per rendere più efficace ed efficiente il progetto software che si sta realizzando.

1.4.5 LEAN SOFTWARE DEVELOPMENT

Il Lean Software Development è una metodologia Agile che trae le sue origini dalla "produzione snella" (o "lean production"), ossia una filosofia industriale che mira a minimizzare gli sprechi fino ad annullarli. Tom e Mary Poppendieck[18] individuano i seguenti principi del Lean Software Development:

- **Eliminare gli sprechi.** Durante lo sviluppo di un progetto software occorre eliminare tutti gli elementi che non aggiungono valore al progetto software che si vuole realizzare;
- **Amplificare l'apprendimento.** All'interno di un ambiente di sviluppo software occorre massimizzare l'apprendimento del progetto software che si sta sviluppando. In particolare, il processo di apprendimento può essere accelerato tramite l'utilizzo di cicli di brevi iterazioni;
- **Decidere il più tardi possibile.** I risultati migliori di un processo di sviluppo software possono essere raggiunti ritardando le decisioni da prendere all'interno del team di sviluppo software. Così facendo, infatti, è possibile prendere le decisioni, inerenti allo sviluppo del progetto software, sulla base di fatti concreti e non in base a determinate previsioni;
- **Sviluppare il più velocemente possibile.** La velocità di programmazione è fondamentale all'interno dello sviluppo di un progetto software. Una rapida consegna dell'ultima release del software, infatti, permette di ricevere immediatamente le valutazioni del cliente e, conseguentemente, permette anche di implementare le modifiche richieste dal cliente nelle fasi cruciali dello sviluppo software;
- **Dare potere al team di sviluppo.** I manager aziendali potranno adeguatamente illustrare le azioni da adottare nello sviluppo del processo software solo se avranno adeguatamente considerato il parere degli sviluppatori software;
- **Integrità nella costruzione.** La differenza fondamentale tra un prodotto software di scarsa qualità ed uno di alta qualità risiede principalmente nell'integrità del prodotto. In particolare, all'interno di un prodotto software occorre distinguere tra:
 - **integrità esterna (o integrità "percepita")** : quando il prodotto software raggiunge un equilibrio tra le funzionalità, l'usabilità, l'affidabilità e l'economicità tale da soddisfare pienamente il cliente;
 - **integrità interna (o integrità "concettuale")** : quando le varie componenti del sistema funzionano bene congiuntamente, mantenendo sempre il dovuto equilibrio tra la flessibilità, la manutenzione, l'efficienza e la reattività del software.
- **"See the whole".** I difetti nel progetto software tendono ad accumularsi durante il processo di sviluppo. Per questo motivo, occorre trovare ed eliminare immediatamente le cause di tali difetti attraverso la "decomposizione" dei problemi in piccoli sotto-problemi e la standardizzazione dei diversi stadi di sviluppo.

1.5 CONCLUSIONI

In questo capitolo abbiamo esaminato le più note metodologie Agili di sviluppo software, ormai largamente diffuse. Extreme Programming, ad esempio, è la metodologia più famosa; sono stati condotti diversi esperimenti (per maggiori informazioni è possibile consultare i seguenti articoli: <http://www.agilemodeling.com/essays/proof.htm> e <http://www.ambysoft.com/surveys/agileFebruary2008.html>) che hanno confermato come l'adozione di questa tecnica abbia portato un vero abbattimento dei tempi di sviluppo ed un miglioramento nella qualità del codice prodotto. Scrum, invece, è un framework da molti rivisitato e sviluppato in questi ultimi anni. Andrea Tomasini[19], ossia il principale esponente italiano in materia di Scrum, sostiene che Scrum permetta, "attraverso varie ispezioni consecutive, uno sviluppo iterativo ed incrementale del prodotto che aiuta ad adattare il prodotto alle nuove richieste e scoperte, facilitando il lavoro di ricerca, la velocità con la quale ci si può adattare ai cambiamenti e riducendo anche i costi di gestione". Crystal è la metodologia più utilizzata per migliorare l'interazione e la collaborazione all'interno del team di sviluppo. Dynamic Systems Development Method (DSDM) è particolarmente utile quando, in un breve lasso di tempo, si vogliono sviluppare sistemi software in cui i requisiti possano essere modificati durante lo sviluppo del progetto software. Infine, abbiamo presentato il Lean Software Development, ossia la metodologia Agile che mira a minimizzare gli sprechi fino ad annullarli.

Nei prossimi due capitoli analizziamo le due "pratiche" della programmazione Agile denominate Test Driven Development e Refactoring. Queste due pratiche si sono ormai affermate come tecniche efficaci di produzione del codice anche in team di sviluppo che non adottano una precisa metodologia Agile come quelle viste in questo capitolo.

TEST-DRIVEN DEVELOPMENT : "UNIT TEST"

2.1 INTRODUZIONE

Il Test Driven Development (TDD) è probabilmente la tecnica Agile di sviluppo del codice che ha avuto maggiore diffusione negli ultimi anni. Tale tecnica trae le sue origini dalla metodologia di eXtreme Programming e si basa sull'utilizzo di una o più classi di test (o "unit test") per sviluppare adeguatamente il progetto software. Occorre, tuttavia, sottolineare che il Test Driven Development non è una tecnica di test, bensì uno stile di sviluppo e progettazione del codice. Essa, infatti, prevede che, inizialmente, venga sviluppato il codice di test relativo alla funzionalità che si vuole implementare e, solo in seguito, il relativo codice applicativo. Il Test Driven Development ruota attorno a due concetti fondamentali:

- o TEST FIRST PROGRAMMING : le classi di test devono essere create antecedentemente al codice di produzione;
- o REFACTORING: dopo la creazione delle classi di test e del codice di produzione è opportuno effettuare il "refactoring".

2.1.1 TEST

Un test è un insieme di operazioni effettuate durante la fase di sviluppo di un progetto software con l'obiettivo di rilevare gli eventuali malfunzionamenti del software prima dell'ultima "release" del prodotto. Quando si parla di test, tuttavia, occorre distinguere tra test funzionale ed "unit test". Per "unit test" s'intende il test dal punto di vista del programmatore, ossia la classe che permette di verificare se, il codice unitario (singolo metodo o classe), che è stato appena prodotto, risponde adeguatamente alle richieste. Quando si parla di test funzionali, invece, ci si riferisce ai test dal punto di vista dell'utente finale. In altre parole, i test in cui è necessario introdurre un input ed il programma si occupa di restituire l'output, senza però mostrare i calcoli che ha effettuato per ottenere tale risultato.

Per poter meglio distinguere gli "unit test" dai test funzionali, Michael Feathers[21] sostiene che:

"Unit tests run fast. If they don't run fast, they aren't unit tests"

Le cinque regole fondamentali per avere una buona classe di "unit test", secondo Robert Martin[2], sono:

- o **FAST**: gli unit test, come sostenuto anche da Michael Feathers[21], devono essere veloci. In caso contrario, infatti, si tende ad eseguire le classi di test con una minore frequenza rispetto alle attese. Gli eventuali bug o malfunzionamenti, in tal modo, non verranno rilevati nella fase iniziale dello sviluppo, dove potrebbero essere risolti con facilità, ma nelle fasi finali dove ogni modifica risulta più critica.
- o **INDEPENDENT**: le varie classi di unit testing, inoltre, non devono dipendere l'una dall'altra, bensì devono essere indipendenti. Se così non fosse, infatti, i difetti relativi al fallimento di un test sarebbero facilmente rilevabili, ma la relativa diagnosi sarebbe difficile da compiere.
- o **REPEATABLE**: gli unit test devono essere ripetibili (e ripetuti) con cicli regolari e continui in qualsiasi "ambiente di sviluppo".
- o **SELF-VALIDATING**: le varie classi di unit testing devono avere un'uscita booleana. Se così non fosse, infatti, si dovrebbe leggere un file di log per capire il fallimento o meno di un test. Utilizzando l'uscita booleana della classe di test, eviteremo di confrontare manualmente i due file di testo.
- o **TIMELY**: gli unit test devono essere scritti in modo tempestivo, antecedentemente al codice di produzione.

Lo "unit test" è la garanzia di affidabilità e fiducia nel codice da parte del programmatore. Come sostenuto anche da Sandro Pedrazzini[3], tuttavia, la fase di testing viene spesso tralasciata dai componenti di un team di sviluppo a causa sia delle tempistiche a cui devono attenersi, ma anche a causa della loro eccessiva sicurezza nel codice e della loro scarsa professionalità. Sebbene, infatti, al giorno d'oggi le tempistiche siano sempre più strette, i programmatori dovrebbero evitare di trascurare l'attività di test. In caso contrario, infatti, diminuirebbe la stabilità del codice e gli errori comparirebbero "a cascata". È difficile, se non impossibile, infatti, produrre immediatamente codice senza errori, soprattutto se si evita di eseguire radicalmente l'attività di test. I principali benefici che si ottengono dall'utilizzo delle classi di test sono i seguenti:

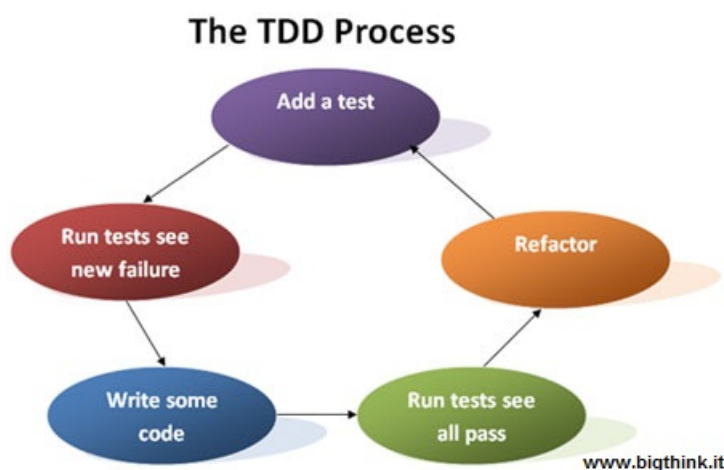
- Accorciamento dei cicli di sviluppo del codice: viene diminuito il tempo che intercorre tra i vari cambiamenti che vengono apportati al codice, senza che il software si trovi mai in uno stato inconsistente.
- Aumenta la fiducia del programmatore nel software prodotto.
- Aumenta la predisposizione del programmatore al cambiamento del codice, anche se funzionante.

- Facilita il refactoring del codice.
- Aumenta la produttività: se gli unit test vengono eseguiti automaticamente dal sistema, infatti, si aumenta la produttività del programmatore. Il test automatico, infatti, richiede un costo iniziale che, però, verrà ammortizzato ogniqualvolta il test viene eseguito automaticamente dal programma.

2.2 SVILUPPO DEL TEST-DRIVEN DEVELOPMENT

La tecnica del Test-Driven Development è basata sul fatto che il test di un modulo di codice, per esempio un metodo, venga scritto prima che sia scritto il codice che si vuole testare. Quindi il test è destinato a fallire sicuramente la prima volta (inevitabilmente, se il metodo che si sta testando non esiste ancora). Ma sarà proprio il primo fallimento, insieme con i successivi successi, a determinare il design del codice di produzione. Dunque, l'atto di scrivere il test diventa più *un momento di design del codice che una pura tecnica di verifica*. Possiamo sintetizzare il processo test-driven nelle seguenti fasi:

1. Scrivere un test;
2. Eseguire il test e verificare che fallisca;
3. Scrivere il codice di produzione;
4. Eseguire nuovamente il test;
5. Eseguire il "refactoring" del codice.



In particolare, la fase iniziale consiste nel creare il test che dovrà fallire al momento dell'esecuzione a causa della mancata scrittura del codice di produzione.

Naturalmente il test dovrà essere creato in base alle specifiche ed ai requisiti del progetto software. Quest'ultimo aspetto può sembrare banale, ma è di fondamentale importanza. Dopo aver proceduto alla realizzazione della classe di test, occorre verificare che effettivamente lo "unit test" fallisca. Se ciò accade, allora il test non passa sempre e, dunque, si può procedere alla concreta realizzazione del codice. Altrimenti, se il test non fallisce, allora c'è un problema all'interno della classe di test ed occorre analizzare adeguatamente il codice al fine di trovare l'errore che ne causa il mancato fallimento.

Supponendo che il test abbia correttamente fallito al momento dell'esecuzione, è possibile scrivere il codice di produzione, ossia il codice da testare con il suddetto "unit test". Il codice di produzione deve essere creato con le sole funzionalità che permettano al test di "passare"; nessuna funzionalità aggiuntiva deve essere inserita nella classe di test.

Dopo aver sviluppato adeguatamente il codice, è possibile eseguire nuovamente lo "unit test". Se il test passa, allora il codice prodotto è correttamente funzionante ed è possibile eseguire l'attività di "refactoring". Dopo aver verificato la correttezza del codice, infatti, occorre modificare opportunamente il codice per migliorarne la qualità. Tale procedimento, denominato "refactoring", che ha l'obiettivo di migliorare la struttura interna del software, senza però alterarne le funzionalità, sarà esaminare nel prossimo capitolo.

Infine, occorre ripetere il test e verificare la correttezza del software anche dopo aver effettuato l'attività di "refactoring".

È possibile, dunque, distinguere le seguenti tre regole fondamentali del Test-Driven Development, descritte da Robert Martin[2]:

1. Non scrivere codice di produzione se prima non hai scritto una classe di test che fallisca;
2. Non scrivere più di una classe di test che fallisca;
3. Non puoi scrivere più codice di produzione di quello sufficiente a far sì che il test, che precedentemente ha fallito, possa funzionare.

Analizzando le varie fasi che compongono il Test-Driven Development e le regole sopra citate, possiamo allora capire il seguente motto, più volte citato nella comunità Agile e dallo stesso Robert Martin[2]:

"Test code is just important as production code"

2.2.1 A CHI È RIVOLTO IL TEST-DRIVEN DEVELOPMENT

A volte, quando si sviluppa un progetto software, l'obiettivo di un team di sviluppo è solo scrivere codice funzionante, senza preoccuparsi di eventuali modifiche che il team potrebbe apportare successivamente al software. Il Test-Driven Development non è rivolto a tali persone, ma piuttosto a coloro che vogliono incrementare sempre di più la qualità del software e che non si fermano al primo successo. Il principale obiettivo del Test-Driven Development, infatti, è quello di migliorare sia la qualità che i tempi di realizzazione del codice prodotto, aiutando anche i componenti del team di sviluppo a risolvere eventuali dubbi che si sono manifestati all'interno del progetto software. L'introduzione di questa tecnica di sviluppo software permette di scrivere codice con un minor numero di bug, avere un design più "pulito" ed aumentare la fiducia che ogni programmatore nutre nei propri confronti. Spesso, infatti, succede che alcuni componenti di un team di sviluppo si trovino in difficoltà nello sviluppare adeguatamente un progetto software per cause di varia natura come, ad esempio, l'inesperienza nello sviluppare codici di notevoli dimensioni. Se ciò accade, è possibile che tali programmatori diventino ancora più scettici delle proprie qualità e si demoralizzino ancora di più. Una possibile soluzione consiste nell'utilizzare la pratica del Test-Driven Development. Attraverso l'utilizzo degli "unit test", infatti, diventa più semplice e divertente sviluppare un buon progetto software.

2.2.2 STRATEGIE DEL TEST-DRIVEN DEVELOPMENT

All'interno del Test-Driven Development è possibile distinguere tre differenti strategie di sviluppo software relative alle classi di test:

- Implementazione "fasulla";
- Triangolazione;
- Implementazione diretta.

L'implementazione "fasulla" (o "fake it") prevede l'inserimento nella classe di test di una costante che, gradualmente, verrà sostituita con alcune variabili per ottenere un codice corretto e funzionante. Ecco un possibile esempio tratto dal libro di Sandro Pedrazzini[3]:

Esempio

Supponiamo di voler implementare la moltiplicazione tra monete della stessa valuta e di voler eseguire la seguente classe di test:

```
public void test(){
    Moneta m = new Moneta(10, "EURO");
    m.multiply(5);
    assertEquals(50, m.getAmount());
}
```

dove la classe Moneta ed il relativo metodo multiply() sono così definiti:

```
public class Moneta(){
    private int amount;
    private String valuta;

    public Moneta(int k, String v){
        amount = k;
        valuta = v;
    }

    public void multiply(int p){
    }

    public int get_tot(){
    }

}
```

L'esecuzione della suddetta classe di test provoca, naturalmente, un fallimento, a causa della mancata implementazione del codice di produzione. Ottenuto il fallimento, è possibile implementare effettivamente il metodo multiply() nel seguente modo:

```
public void multiply(int p){
    amount = 50;
}
```

In altre parole, si è proceduto all'inserimento, all'interno del metodo multiply(), di una costante che permetta al test di passare solo in questo caso

specifico. Procedendo nuovamente all'esecuzione dello "unit test", è possibile verificarne il mancato fallimento.

L'obiettivo, tuttavia, è quello di far passare il test anche in qualsiasi altra circostanza. Generalizzando, dunque, possiamo scrivere:

```
public void multiply(int p){  
    amount = amount * p;  
}
```

Un'altra strategia possibile è nota come "Triangolazione". L'idea di base consiste nel creare un nuovo test, o modificare quello esistente, al fine di verificare l'inesattezza o meno dell'implementazione attuale. Tale strategia viene utilizzata soprattutto quando sorgono dubbi riguardo alla realizzazione finale.

Ecco il funzionamento della strategia di "Triangolazione" considerando l'esempio precedente sulla moltiplicazione di monete:

Esempio

Supponiamo di aver implementato la seguente classe di test:

```
public void test(){  
    Moneta m = new Moneta(10, "EURO");  
    assertEquals(new Moneta (50, "EURO"), m.multiply(5));  
}
```

dove il metodo multiply è così definito:

```
public void multiply(int p){  
    return new Moneta(50, "EURO");  
}
```

L'idea di base della strategia di "Triangolazione" consiste nel creare un nuovo test o modificare quello esistente al fine di dimostrare l'inesattezza dell'implementazione attuale. Procediamo dunque alla modifica della classe di test:

```
public void test(){
    Moneta m = new Moneta(10, "EURO");
    assertEquals(new Moneta (50, "EURO"), m.multiply(5));
    assertEquals(new Moneta (20, "EURO"), m.multiply(2));
}
```

Il secondo assert serve a dimostrare che il programma non è generico, anche se può essere reso tale modificando il metodo multiply nel seguente modo:

```
public void multiply(int p){
    return new Moneta (getAmount() * p, getValuta());
}
```

con il metodo getValuta() della classe Moneta così definito:

```
public String getValuta(){
    return valuta;
}
```

Esiste, infine, un'ultima strategia: l'implementazione diretta. Rispetto alle precedenti, è molto difficile da attuare poiché richiede la perfezione del codice prodotto. Occorre, infatti, produrre codice "pulito" e perfettamente funzionante al primo "tentativo". Se il test fallisce, allora è opportuno utilizzare altre strategie d'implementazione.

Esempio

Considerando l'esempio precedente sulla moltiplicazione di monete, ecco una possibile classe di test:

```
public void test(){
    Moneta m = new Moneta(10, "EURO");
    Moneta ris = m.multiply(3);
    assertEquals(new Moneta (30, "EURO"), ris);
}
```



```
ris = m.multiply(2);
assertEquals(new Moneta (20, "EURO"), ris);
assertEquals(new Moneta (10, "EURO"), m);
}
```

con il seguente metodo multiply della classe Moneta:

```
public void multiply(int p){
return new Moneta (getAmount() * p, getValuta());
}
```

2.2.3 VANTAGGI DEL TEST-DRIVEN DEVELOPMENT

L'utilizzo del Test-Driven Development permette di incrementare la qualità del progetto software e, al contempo, di ridurre i tempi di realizzazione. Avvalendosi di questa tecnica di sviluppo software, infatti, i vari componenti di un team di sviluppo possono implementare alcune porzioni del codice di produzione e procedere immediatamente all'esecuzione delle relative classi di test. Se il risultato è quello sperato, allora è possibile procedere all'implementazione di altre funzionalità del progetto software. Altrimenti, occorre analizzare adeguatamente il codice ed individuare le possibili cause di tali errori o malfunzionamenti. Tuttavia, grazie ad un regolare e continuo sviluppo del progetto e delle relative classi di test, non sarà difficile identificare le cause del fallimento. Inoltre, ad ogni concreta realizzazione di alcune funzionalità del progetto software, sarà possibile interagire con il cliente per comprendere meglio le sue aspettative a riguardo. Il Test-Driven Development, infatti, permette di accorciare il tempo che intercorre tra lo sviluppo di alcune funzionalità e la relativa valutazione del cliente. Prima dell'avvento di questa tecnica di sviluppo software, infatti, era necessario aspettare settimane o addirittura mesi prima di poter conoscere la reazione del cliente; adesso, invece, sono sufficienti alcuni minuti per conoscere le sue valutazioni. Grazie all'utilizzo del Test-Driven Development, dunque, è possibile rilevare gli eventuali bug o malfunzionamenti nella fase iniziale dello sviluppo software e non al termine di essa, come invece accadeva precedentemente. È possibile, altresì, procedere ad un rapido sviluppo del codice ed all'eliminazione delle funzionalità extra non necessarie per la corretta esecuzione del programma. In secondo luogo, l'insieme dei test, costruito durante

lo sviluppo del sistema software, rappresenta una protezione, una misura di sicurezza, per successive modifiche. Supponiamo di volere, in seguito, aggiungere delle funzionalità o modificarne qualcuna di quelle esistenti: la ripetizione dei test ci assicura che le funzionalità preesistenti non sono state intaccate dalle nostre modifiche, che il nostro codice continua a comportarsi correttamente come prima. Questo, peraltro, rende gli sviluppatori più ad effettuare cambiamenti, meno insicuri a toccare il codice esisterete. Infine, il fatto di scrivere il test prima del relativo codice di produzione influenza in modo positivo molte scelte di progettazione. Per esempio, questa tecnica ci costringe codice facilmente tentabile, semplice e modulare: ogni metodo deve avere una singola responsabilità, se vogliamo che quella funzionalità sia testabile in modo autonomo. Analogamente, essa porta a ridurre le dipendenze fra classi concrete: supponiamo di dover scrivere, ad esempio il test di un metodo che usa oggetti di una classe che non è stata ancora sviluppata. Per eseguire il test, e per scrivere questo metodo, è sufficiente un'interfaccia con le specifiche *minime* necessarie per il metodo. Fatte queste considerazioni, emerge chiaramente l'importanza di avere a disposizione un ambiente di supporto automatico per la scrittura e l'esecuzione dei test. JUnit, integrato in Eclipse, è lo strumento più noto per la programmazione in Java e, di fatto, lo abbiamo già usato parzialmente negli esempi sopra presentati. Passiamo, perciò, a illustrare alcune caratteristiche fondamentali di JUnit nel paragrafo successivo, attraverso esempi di uso di JUnit per l'approccio Test-Driven.

2.3 JUNIT

JUnit è un framework di "unit testing" per il linguaggio di programmazione Java, ossia una struttura logica in base alla quale un software può essere progettato e realizzato e che ha come obiettivo quello di facilitarne lo sviluppo. L'utilizzo di un framework impone al programmatore una precisa metodologia di sviluppo del software. Tuttavia, il programmatore può evitare di farsi carico della scrittura del "codice di contorno" in modo da focalizzarsi sul contenuto vero e proprio del programma che sta realizzando. Junit, in particolare, è un'istanza della classe xUnit. Con il termine xUnit si intendono tutti quei framework che consentono di testare svariate entità di un sistema software come, ad esempio, funzioni e classi.

La prima implementazione della classe xUnit fu denominata SUnit. Tale framework si basava, principalmente, sul linguaggio di programmazione SmallTalk e permetteva ai programmatori di scrivere test e verificarli. Il principale problema di SUnit, tuttavia, era SmallTalk. Tale linguaggio di programmazione, infatti, obbligava i programmatori a sviluppare progetti software non troppo complicati. Solo in seguito, Erich Gamma e Kent Beck introdussero il linguaggio di programmazione Java all'interno della classe xUnit, creando così il framework

JUnit. Ecco gli scopi che, secondo Gamma & Beck[4], furono alla base della realizzazione di tale framework:

- o **Scrivere test deve essere semplice** : la scrittura e la relativa esecuzione delle classi di test deve essere più semplice e richiedere meno tempo rispetto alla verifica manuale delle operazioni inserite nelle classi di test;
- o **I test devono essere automatici** : le classi di test devono essere eseguite automaticamente dal programma in modo da verificare i risultati senza bisogno di nessun intervento da parte dell'utente;
- o **I test devono essere componibili** : tramite un framework di unit testing, come ad esempio JUnit, l'utente deve avere la possibilità di "combinare" una o più classi di "unit test";
- o **I singoli test devono essere isolati**: un requisito fondamentale delle classi di unit testing, infine, è l'indipendenza. Secondo Gerard Meszaros[5], infatti, il risultato dell'esecuzione di una classe di test non deve influenzare in alcun modo l'esecuzione delle altre classi di test.

2.3.1 ASSERT

Uno dei meccanismi fondamentali del framework JUnit è rappresentato da "assert", ossia un predicato che si occupa di confrontare un risultato ottenuto da una particolare operazione con il risultato atteso. Se coincidono, allora restituisce vero; altrimenti, restituisce falso. Esistono molte tipologie di assert, ma tutte permettono di evitare l'esplicita implementazione del confronto tra due valori all'interno del codice.

Il più importante tra i vari tipi di assert è certamente AssertTrue() poiché permette di verificare il risultato di una qualsiasi espressione booleana. Gli altri tipi di assert sono:

- assertEquals(): permette di verificare l'uguaglianza tra due oggetti, due variabili, due espressioni booleane, ...;
- assertNull(): permette di verificare che un oggetto sia nullo;
- assertNotNull(): permette di verificare che un oggetto non sia nullo;
- assertFalse(): permette di verificare che un'espressione booleana sia falsa;
- assertSame(): permette di verificare che due oggetti facciano riferimento allo stesso oggetto;
- assertNotSame(): permette di verificare che due oggetti non facciano riferimento allo stesso oggetto.

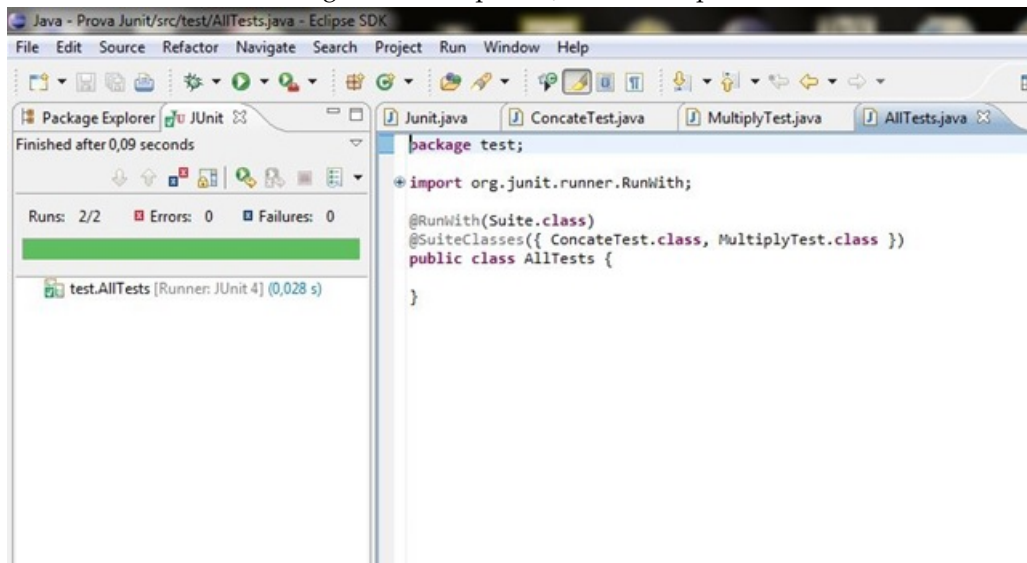
2.3.2 ERROR & FAILURE

Il framework di unit testing denominato "JUnit" prevede un'interfaccia molto semplice ed efficace. Se, infatti, il risultato dell'esecuzione di una classe di test risulta quello atteso, allora JUnit mostra una barra verde. Altrimenti, se si è verificato un problema, JUnit mostra una barra rossa. Le possibili cause possono essere essenzialmente due:

- **error** : se la causa del mancato funzionamento dello "unit test" è un errore;
- **failure**: se la causa del mancato funzionamento è un falso risultato.

I fallimenti possono essere previsti da parte del programmatore, mentre gli errori si verificano piuttosto inaspettatamente. Un failure, infatti, viene rilevato se una o più asserzioni all'interno di un test risultano false, mentre si ha un errore se si è verificata un'eccezione imprevista durante il test. Naturalmente, un errore non può essere considerato un fallimento in quanto si tratta di un'uguaglianza non verificata e perciò viene classificata come errore e non come failure.

Figure 1: Esempio di Junit su Eclipse



2.3.3 ALTRI METODI

All'interno di una classe di test, oltre ai metodi `assert()`, è possibile utilizzare anche metodi come `setUp()` e `tearDown()`. Entrambi vengono ereditati come metodi vuoti e dunque, sovrascrivendoli, è possibile creare nuove funzionalità

da utilizzare nelle relative classi di test.

Il metodo `setUp()` deve essere chiamato prima dell'esecuzione di qualsiasi metodo di testing poiché ha lo scopo di inizializzare i vari elementi in comune tra le classi di test. Così facendo, infatti, i metodi all'interno delle varie classi di test saranno più "corti", il codice sarà più sintetico e le variabili non dovranno essere più dichiarate globali. Tuttavia, ciò non significa automaticamente che, utilizzando `setUp()`, si aumenta anche la chiarezza del codice. Spesso, infatti, accade il contrario.

Il metodo `tearDown()` è particolarmente utile quando vengono utilizzate risorse esterne all'azienda; la sua esecuzione, infatti, è prevista soltanto dopo la terminazione delle varie operazioni di testing e, dunque, all'interno del metodo `tearDown()`, è possibile provvedere a rilasciare tutte le varie risorse che sono state utilizzate.

2.3.4 TEST DOUBLE

Un test double è un oggetto "fittizio" che può essere utilizzato all'interno di un progetto software per riprodurre fedelmente il comportamento di un oggetto reale. Il principale obiettivo è verificare, tramite l'esecuzione di uno o più "unit test", la corretta implementazione di alcune funzionalità presenti all'interno del codice, senza però interagire con l'esterno. Ecco quattro differenti tipi di test double, come sostenuto anche da Gerard Meszaros[5]:

- **Dummy Objects**, ossia gli oggetti che non vengono mai utilizzati e che servono solamente per "riempire" la lista dei parametri;
- **Fake Object**, ossia gli oggetti che non sono adatti a sviluppare il codice di produzione sebbene presentino implementazioni funzionanti;
- **Stubs**, ossia gli oggetti che forniscono soltanto alcune risposte precostruite alle chiamate effettuate durante l'esecuzione delle classi di test;
- **Mock Objects**, ossia gli oggetti pre-programmati aventi lo scopo di sostituire l'oggetto reale in modo che il sistema possa facilmente verificare lo stato ed il comportamento dell'unità testata.

2.3.4.1 Dummy Objects

I Dummy Objects sono oggetti "segnaposto", ossia oggetti che vengono passati come argomento ai vari metodi, ma non vengono mai realmente utilizzati. Per creare un Dummy Object occorre, innanzitutto, creare un'istanza di un oggetto

che possa essere inizializzato facilmente e senza dipendenze da altri oggetti. In seguito, occorre passare, come argomento di un metodo, l'istanza che abbiamo precedentemente creato. Tale istanza non verrà effettivamente utilizzata dal sistema in esecuzione e, dunque, non necessitiamo di nessuna implementazione per tale oggetto. In caso contrario, se il sistema utilizza la suddetta istanza, allora viene generato un errore perché si cerca di invocare un metodo inesistente. L'utilizzo dei Dummy Objects, dunque, evita al programmatore di scrivere codice irrilevante che, invece, sarebbe stato necessario se si fosse utilizzato un oggetto reale.

2.3.4.2 *Fake Objects*

I Fake Objects sono oggetti "fittizi" che sono molto più semplici rispetto all'oggetto reale che rappresentano poiché non necessitano delle varie proprietà che possiede l'oggetto reale. Un possibile esempio di una proprietà posseduta dall'oggetto reale, ma non dal Fake Object, potrebbe essere la scalabilità. Un Fake Object, infatti, deve fornire solo i servizi di cui necessita il sistema in esecuzione, facendo in modo che quest'ultimo non conosca il tipo di oggetto (reale o "fittizio") che sta utilizzando.

2.3.4.3 *Stubs*

Gli stubs sono oggetti fittizi che simulano il comportamento dell'oggetto reale corrispondente senza però fornire alcuna risposta su ciò che non è previsto dalla classe di test. L'utilizzo di uno stub, dunque, permette al programmatore di testare porzioni di codice che, altrimenti, non sarebbe stato possibile eseguire. Tuttavia, il loro utilizzo, dal punto di vista del programmatore, è piuttosto limitato poiché forniscono risposte pre-costruite alle chiamate fatte durante il test.

2.3.4.4 *Mock Objects*

Similmente agli Stubs, anche i Mock Objects sono oggetti fittizi che simulano il comportamento del corrispondente oggetto reale. Rispetto agli Stubs, tuttavia, i Mock Objects confrontano le chiamate effettivamente ricevute durante il test con le chiamate che ci si aspettava venissero effettuate e che sono state precedentemente definite mediante opportune asserzioni. Tali oggetti consentono, dunque, di verificare anche il comportamento dell'unità testata e non solo lo stato in cui si trova. È possibile distinguere due differenti tipi di Mock Objects:

- **Mock Objects strict**, che falliscono il test se le chiamate che vengono ricevute sono in un ordine differente da quello che ci si aspettava;
- **Mock Objects lenient**, che tollerano anche le chiamate non previste. Essi, infatti, tollerano o addirittura ignorano le chiamate inaspettate o perse.

2.4 ESEMPI DI USO DI JUNIT

Ecco alcuni esempi, tratti dal libro di Sandro Pedrazzini[3], relativi all'utilizzo del framework JUnit.

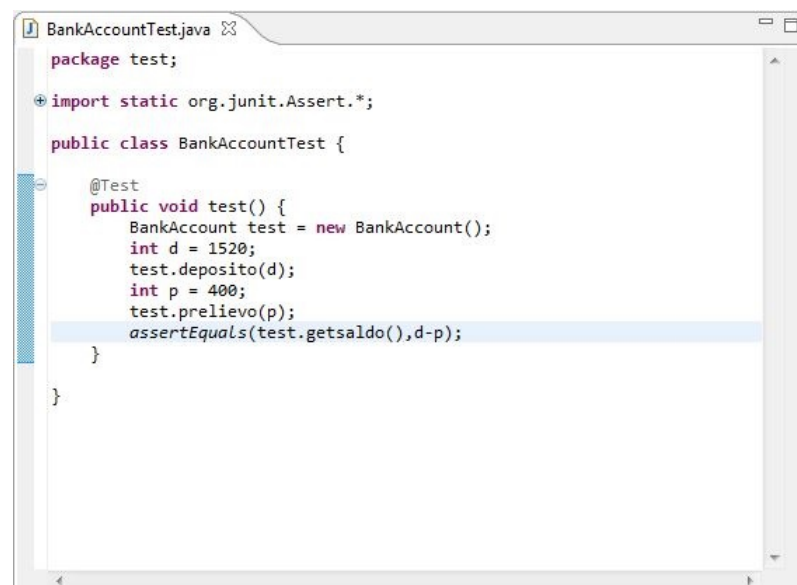
2.4.1 ESEMPIO: CONTO CORRENTE

Implementare tutte le possibili operazioni che un cliente può effettuare sul proprio conto corrente relativo ad una determinata banca X. Le principali operazioni che il cliente può eseguire sono:

- **Prelievo** : il cliente ritira soldi dal proprio conto corrente;
- **Deposito** : il cliente deposita soldi nel proprio conto corrente;
- **Saldo** : il cliente richiede il saldo del proprio conto corrente.

2.4.1.1 Svolgimento

Inizialmente occorre creare la classe di test che, inevitabilmente, dovrà fallire perché ancora non è stato creato il relativo codice di produzione.



```
BankAccountTest.java
package test;

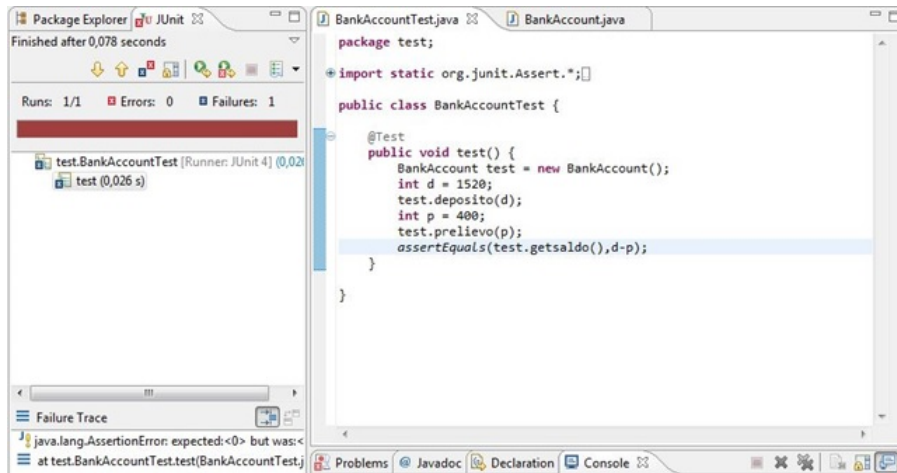
import static org.junit.Assert.*;

public class BankAccountTest {

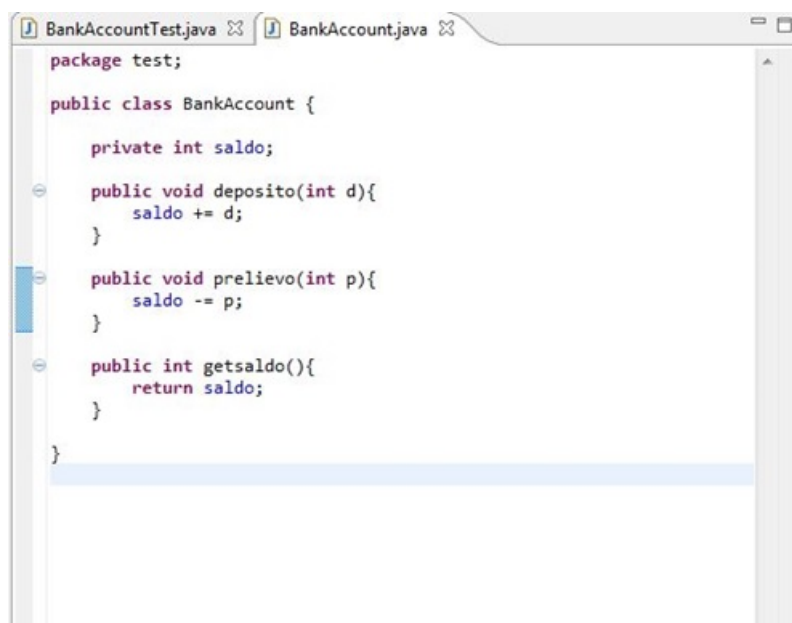
    @Test
    public void test() {
        BankAccount test = new BankAccount();
        int d = 1520;
        test.deposito(d);
        int p = 400;
        test.prelievo(p);
        assertEquals(test.getsaldo(), d-p);
    }

}
```

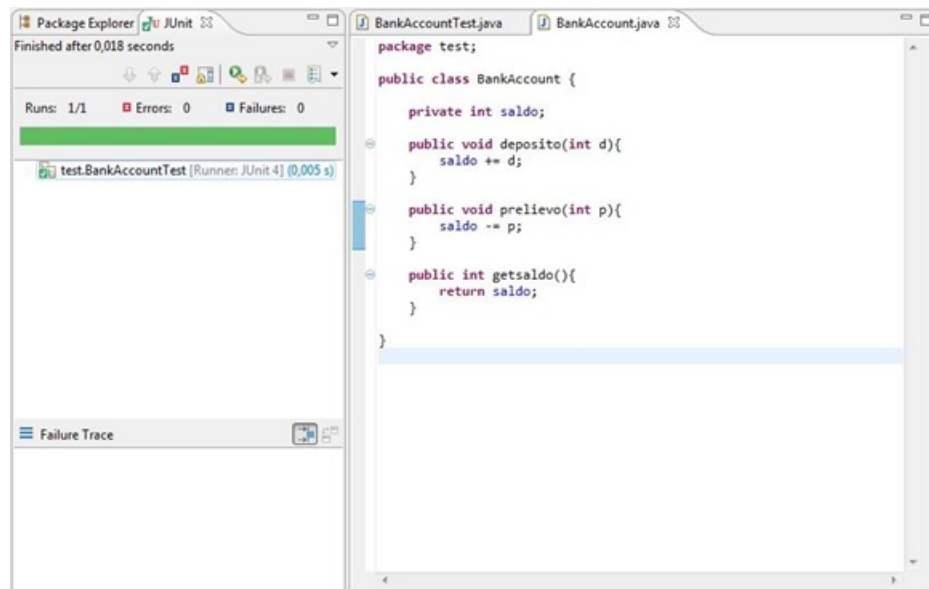
Adesso occorre verificare che il test fallisca.



Come previsto, il test ha fallito. Verificato ciò, è possibile scrivere il codice di produzione della classe BankAccount. Tale classe, però, deve contenere solo i metodi e gli attributi che permettano al test di passare; non deve contenere, infatti, nessuna funzionalità aggiuntiva. Ecco il codice:

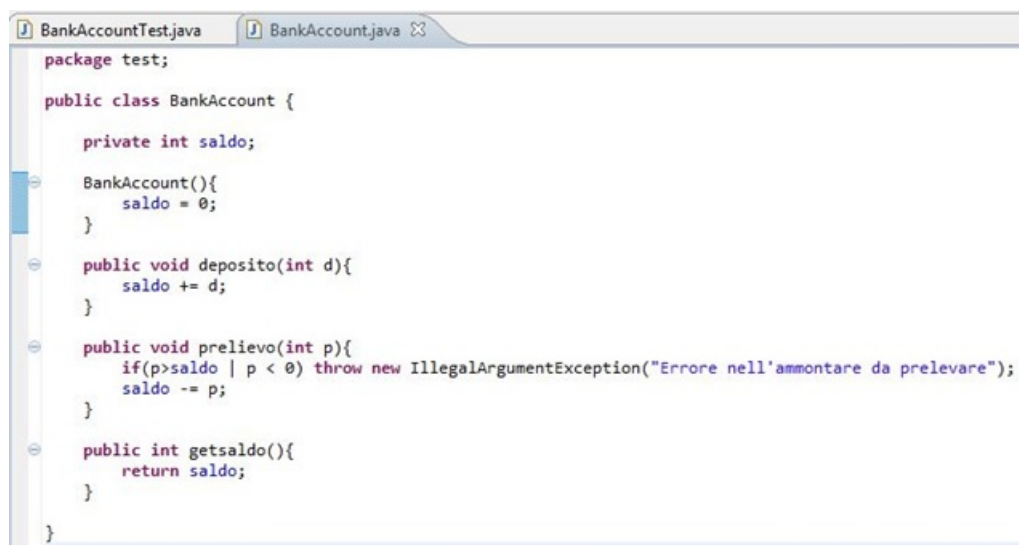


Dopo aver sviluppato il codice, occorre rieseguire il test.

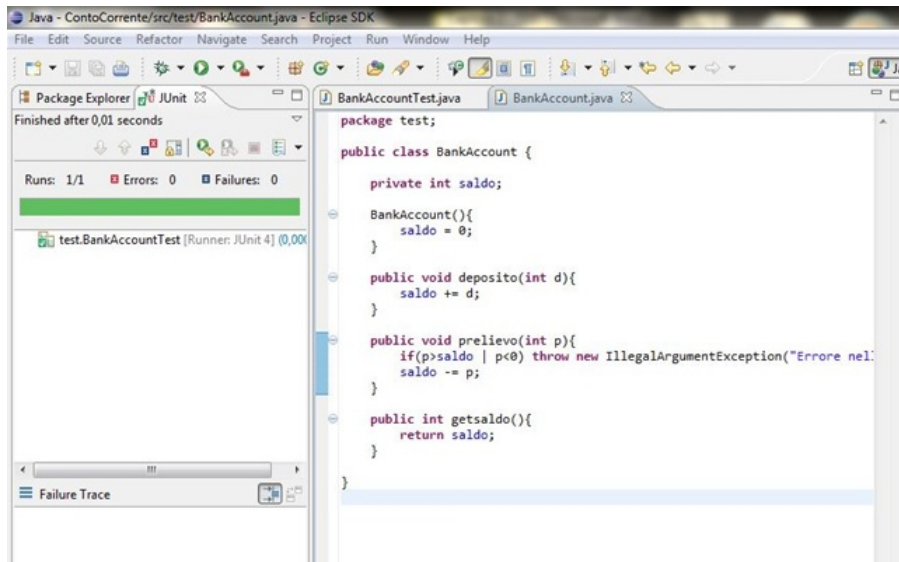


Il test passa e, quindi, il codice è funzionante.

Adesso è possibile eseguire l'attività di refactoring. Ad esempio, è possibile aggiungere il costruttore basilare del saldo, quando si crea un nuovo conto, oppure creare un nuovo controllo sulla somma che è possibile prelevare. Ecco il codice:



Adesso occorre verificare, nuovamente, che il test funzioni.



La barra è verde e non sono stati rilevati nè errori nè failure, quindi il test funziona ancora.

2.4.2 ESEMPIO: MONETE

Supponiamo, invece, di voler implementare la somma e la sottrazione tra monete. Se la somma è tra due monete dello stesso tipo, allora si tratterà di una semplice somma numerica. Altrimenti, se le due monete sono di valute diverse, il tasso di cambio potrebbe variare nel tempo e, pertanto, non deve essere effettuata immediatamente la conversione delle monete. Occorre gestire opportunamente tale somma.

2.4.2.1 Svolgimento

Inizialmente occorre creare la classe di test, denominata "MoneteTest", e le classi del relativo codice di produzione che, tuttavia, saranno completamente vuote. Inizialmente, infatti, occorre verificare che il test non passi mai. Ecco il codice della classe "MoneteTest":

```

package test;

import static org.junit.Assert.*;

public class MoneteTest {

    @Test
    public void test() {
        Moneta m10EU = new Moneta(10, "EU");
        Moneta m25EU = new Moneta(25, "EU");
        m10EU.add(m25EU);
        assertEquals(m10EU.getimporto(), 35);
        Moneta m4DOL = new Moneta(4, "DOL");
        Moneta m30YEN = new Moneta(30, "YEN");
        Salvadanaio s1 = new Salvadanaio();
        s1.add(m4DOL);
        s1.add(m30YEN);
        s1.add(m10EU);
        s1.add(m25EU);
        assertEquals(s1.gettotcurrency("EU"), 35);
    }
}

```

Nel test soprastante, si è proceduto all'implementazione della somma di due monete della stessa valuta, ovvero l'euro, ed alla gestione di monete di diverso tipo come il dollaro, lo yen e l'euro. Con gli assert, inoltre, si vuole verificare che:

- la somma delle due monete di euro sia avvenuta con successo;
- la somma degli euro all'interno del salvadanaio, contenente anche i dollari e gli yen, sia avvenuta con successo;

Dopo aver implementato opportunamente la classe di test, occorre verificarne il fallimento.

```

package test;

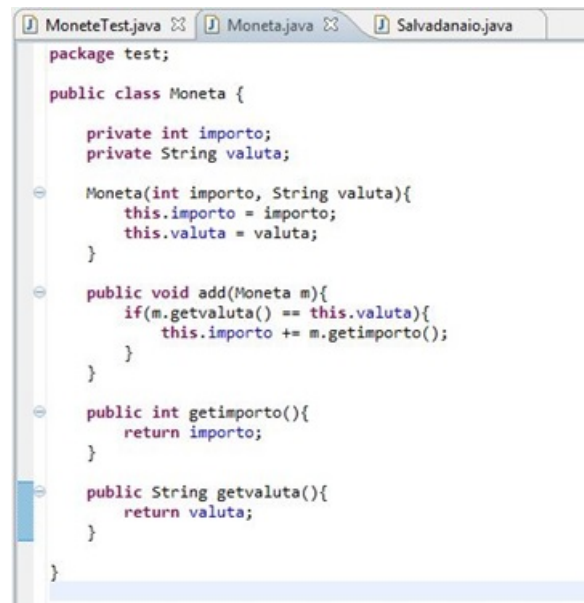
import static org.junit.Assert.*;

public class MoneteTest {

    @Test
    public void test() {
        Moneta m10EU = new Moneta(10, "EU");
        Moneta m25EU = new Moneta(25, "EU");
        m10EU.add(m25EU);
        assertEquals(m10EU.getimporto(), 35);
        Moneta m4DOL = new Moneta(4, "DOL");
        Moneta m30YEN = new Moneta(30, "YEN");
        Salvadanaio s1 = new Salvadanaio();
        s1.add(m4DOL);
        s1.add(m30YEN);
        s1.add(m10EU);
        s1.add(m25EU);
        assertEquals(s1.gettotcurrency("EU"), 35);
    }
}

```

Il test ha fallito e, dunque, ora è possibile implementare il relativo codice di produzione. La classe Moneta potrebbe essere implementata nel seguente modo:



```
package test;

public class Moneta {

    private int importo;
    private String valuta;

    Moneta(int importo, String valuta){
        this.importo = importo;
        this.valuta = valuta;
    }

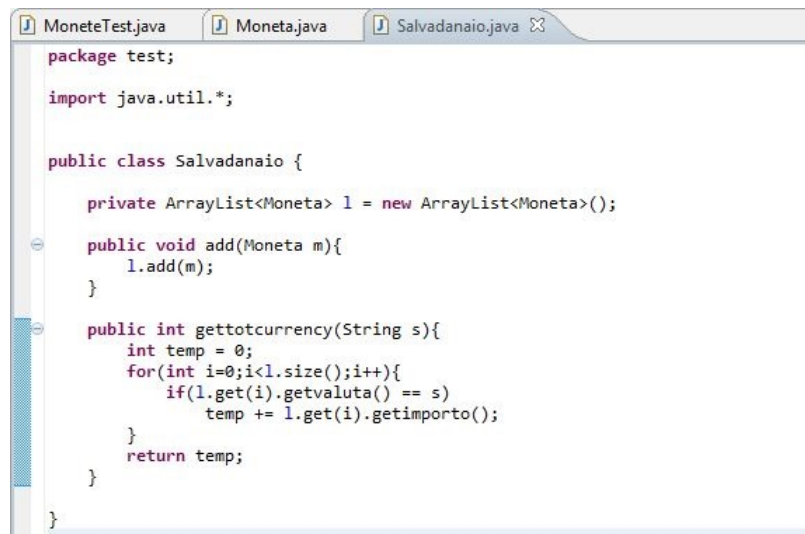
    public void add(Moneta m){
        if(m.getvaluta() == this.valuta){
            this.importo += m.getimporto();
        }
    }

    public int getimporto(){
        return importo;
    }

    public String getvaluta(){
        return valuta;
    }

}
```

La classe Salvadanaio, ovvero la classe che può contenere monete di diverso tipo, può, invece, essere implementata nel seguente modo:



```
package test;

import java.util.*;

public class Salvadanaio {

    private ArrayList<Moneta> l = new ArrayList<Moneta>();

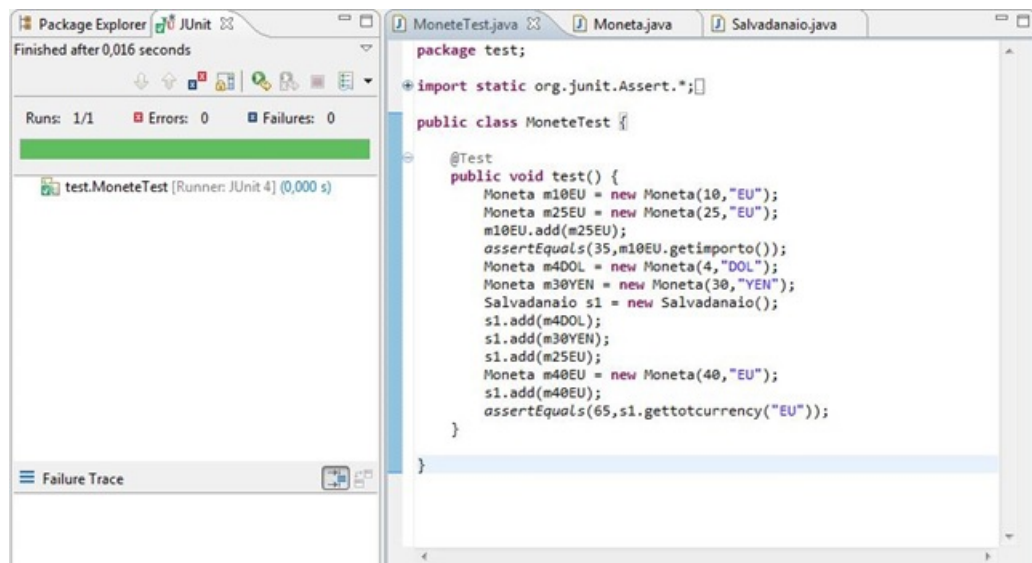
    public void add(Moneta m){
        l.add(m);
    }

    public int gettotcurrency(String s){
        int temp = 0;
        for(int i=0;i<l.size();i++){
            if(l.get(i).getvaluta() == s)
                temp += l.get(i).getimporto();
        }
        return temp;
    }

}
```

Ora è possibile rieseguire il test e verificare che passi, ovvero che la barra diventi verde e che non siano presenti nè failure nè errori.

2.5 IMPATTO DEL TEST-DRIVEN DEVELOPMENT SULLA PRODUTTIVITÀ E QUALITÀ DEL SOFTWARE



Poiché il test è stato eseguito con successo, allora è possibile eseguire l'ultima fase della pratica di Test-Driven Development, ossia il refactoring.

Il refactoring, tuttavia, non è necessario perché non sono state rilevate eventuali duplicazioni e/o migliorie che possono essere effettuate, senza aggiungere alcune funzionalità extra al software.

2.5 IMPATTO DEL TEST-DRIVEN DEVELOPMENT SULLA PRODUTTIVITÀ E QUALITÀ DEL SOFTWARE

Nonostante il Test Driven Development sia una delle più importanti pratiche di sviluppo della metodologia Agile di eXtreme Programming e sia stato adottato all'interno di molte aziende, tra cui anche Microsoft ed IBM, l'impatto di tale pratica sulla produttività e qualità del software è tuttora da stabilire. Sono state effettuate molte ricerche, sia empiriche che non empiriche, che avessero come obiettivo quello di analizzare l'impatto del TDD sulla produttività e sulla qualità del software. Yahya Rafique[8], ad esempio, ha analizzato in modo non empirico l'impatto del Test Driven Development in modo da ottenere una più profonda comprensione di tale tecnica. Il suo approccio può essere suddiviso in due fasi: nella prima fase si è proceduto alla creazione della meta-analisi, mentre nella seconda fase è stato sviluppato un modello di simulazione basato sul TDD. La meta-analisi è uno strumento di ricerca usato per combinare i dati provenienti da diversi studi utilizzando specifiche tecniche matematico-statistiche. In particolare, durante la meta-analisi, sono stati identificati cinque differenti valori che possono influenzare la qualità e la produttività del software come: l'esperienza dei programmatori, la granularità dei test, l'impegno

profuso durante lo sviluppo delle classi di test, la quantità di lavoro assegnata ai programmatori che partecipano allo sviluppo del software e, infine, i tool ed i domini utilizzati per risolvere gli eventuali problemi sorti durante l'implementazione del software. Per comprendere al meglio l'andamento di tali valori nello sviluppo del progetto software, è stato sviluppato un adeguato modello di simulazione. I risultati di tale simulazione hanno confermato o smentito alcune osservazioni effettuate durante la meta-analisi come, ad esempio, il possibile incremento della qualità del software ed il mancato aumento della produttività derivanti da un'elevata granularità delle classi di test. Non sono stati, inoltre, rilevati alcuni benefici, inerenti alla produttività ed alla qualità del software, derivati da un elevato sforzo di progettazione del programmatore o dall'utilizzo di particolari tool e domini per risolvere specifici problemi di progettazione. Un adeguato assegnamento della quantità di lavoro ai componenti del team di sviluppo può, però, incrementare la qualità del software che si sta sviluppando.

I risultati di tale ricerca, dunque, suggeriscono come l'utilizzo del TDD possa incrementare la qualità del progetto software che si sta sviluppando, ma anche causare un mancato incremento o addirittura una diminuzione della produttività.

Boby George e Laurie Williams[9], invece, hanno esaminato empiricamente l'impatto del Test Driven Development sulla produttività e sulla qualità del software. In particolare, hanno sviluppato i loro risultati comparando la pratica del TDD con la metodologia a cascata. La qualità del software prodotto utilizzando la tecnica del Test Driven Development è sembrata maggiore rispetto alla qualità di un progetto software sviluppato con una tecnica più tradizionale come la metodologia a cascata. I risultati delle ricerche effettuate, inoltre, hanno permesso di verificare anche come la produttività del TDD sia maggiore rispetto alla produttività di un software sviluppato con la metodologia a cascata. Infine, effettuando un sondaggio tra i più importanti programmatori software, è stato possibile constatare come l'80% di loro preferisca utilizzare la tecnica del Test Driven Development rispetto alla metodologia a cascata.

2.6 CONSIDERAZIONI

2.6.1 COSA TESTARE E COSA NON TESTARE

Sebbene spetti al programmatore decidere cosa testare e cosa non testare, è possibile porre l'accento su alcune situazioni in cui l'esecuzione degli "unit test" è di fondamentale importanza. Ecco alcuni esempi tratti da Kent Beck[1]:

- o Quando sono presenti all'interno del codice alcune istruzioni condizionali;

- o Quando sono presenti dei cicli all'interno del codice che si sta sviluppando;
- o Quando si eseguono operazioni di qualsiasi natura;
- o Quando si vuole verificare il polimorfismo.

Le classi di test, tuttavia, devono essere eseguite solo se il programmatore, che ha provveduto a creare tali classi, ha anche realizzato il relativo codice di produzione. In caso contrario, infatti, non c'è alcun motivo per cui il programmatore debba diffidare del codice altrui e, dunque, non occorre eseguire alcuna classe di test. Diverso sarebbe se l'esecuzione delle suddette classi di test avesse l'obiettivo di comprendere meglio le funzionalità implementate all'interno del codice. In tal caso, infatti, l'esecuzione degli "unit test" avrebbe uno scopo puramente illustrativo.

2.6.2 COME RICONOSCERE LA BONTÀ DI UN TEST

Un altro aspetto da considerare quando si creano nuove classi di test è l'efficacia (o bontà) delle suddette classi. Occorre, infatti, verificare se il test è "buono" oppure no. Per fare ciò, è possibile analizzare alcuni elementi fondamentali di un progetto software come:

- la fragilità del test;
- la velocità del test;
- la lunghezza del codice di testing.

Un elemento che permette di verificare la bontà o meno di un test è la lunghezza del codice della classe di test. Se vengono utilizzate molte linee di codice per creare dei nuovi oggetti che vengono utilizzati solamente in un semplice assert, allora qualcosa nel codice non torna.

Occorre, inoltre, verificare anche la fragilità dei test. A volte, infatti, succede che i test si fermino inaspettatamente durante la loro esecuzione poiché, ad esempio, una parte del software è stata influenzata da un'altra parte dello stesso. Occorre, dunque, interrompere tale "connessione" oppure far in modo di riunire le due parti.

Un altro elemento che occorre analizzare attentamente è la velocità del test, ovvero il tempo che occorre al test per terminare normalmente la propria esecuzione. La lentezza di un test, infatti, costringe il programmatore a diminuire il numero di esecuzioni dello "unit test" corrispondente o, addirittura, a non eseguirlo proprio. Così facendo, viene compromesso lo sviluppo del software che si sta producendo e la relativa classe di test. Occorre, dunque, cercare di aumentare la velocità di esecuzione del test per far incrementare il numero di esecuzioni dello stesso.

2.6.3 *QUANDO ELIMINARE UNA CLASSE DI TEST*

La possibile eliminazione di una classe di test dipende essenzialmente da due fattori:

- la fiducia che il programmatore nutre nei confronti del software che sta sviluppando;
- la "comunicazione" tra il cliente ed il software che il programmatore sta sviluppando.

Secondo Kent Beck[1], infatti, non bisogna mai eliminare una classe di test se tale eliminazione riduce la confidenza del programmatore nei confronti del software oppure se complica la "comunicazione" tra il cliente ed il software che il programmatore sta sviluppando. Potrebbe succedere, infatti, che, all'interno del progetto software, siano presenti due classi di test che effettuano le stesse operazioni, ma che per il cliente siano entrambe fondamentali perché, ad esempio, rappresentano scenari diversi. In tal caso, occorre che il programmatore mantenga entrambe le classi. In caso contrario, se l'eliminazione di una delle due classi di test non riduce la fiducia del programmatore e non complica la "comunicazione" del cliente, allora è possibile eliminare il meno utile dei due test.

REFACTORING

3.1 INTRODUZIONE

Il *refactoring* è una pratica di sviluppo del software che, secondo Martin Fowler[20], può essere definita nel modo seguente:

"Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs"

Dopo l'applicazione del refactoring, infatti, il software risulterà più comprensibile, più facilmente gestibile e meno incline alla possibilità che si verifichino nuovi bug. Il refactoring può, dunque, risultare particolarmente utile quando il software degenera al punto da diventare difficilmente gestibile ed estendibile, se non a fronte di interventi costosi in termini di tempo e risorse.

Ogni ciclo di refactoring, inoltre, dovrebbe durare poco in termini di tempo. Così facendo, infatti, diminuirebbe la possibilità che si verifichino nuovi bug all'interno del codice.

Un altro aspetto fondamentale, quando si esegue il refactoring, sono le classi di test. Avere delle buone classi di test, infatti, è fondamentale ed imprescindibile per poter pensare di effettuare il refactoring su un qualsiasi sistema, semplice o complicato che sia. Sono gli "unit test", infatti, che danno al programmatore la sicurezza di aver prodotto codice ancora funzionante, dopo aver introdotto delle modifiche. L'effetto cumulativo di piccole e semplici modifiche, unito all'esecuzione di alcune classi di test, possono migliorare drasticamente il progetto software.

Il refactoring può essere suddiviso in:

- refactoring automatico;
- refactoring manuale.

Si tratta di refactoring automatico quando, ad esempio, si modifica il nome di una variabile o di un pacchetto. La mancata automatizzazione di tale passaggio comporta un notevole carico di lavoro extra per il programmatore che è quindi costretto ad analizzare attentamente il codice e sostituire ogni occorrenza di tale variabile o di tale pacchetto con il nuovo valore. Soprattutto quando il codice è molto sviluppato, è facile commettere errori come, ad esempio, la mancata

sostituzione di un'occorrenza.

Più interessanti da analizzare sono i refactoring manuali. A differenza dei refactoring automatici, dove le operazioni che dovevano essere effettuate nel codice sono semplici e facilmente automatizzabili, i refactoring manuali sono sviluppati a discrezione del programmatore. I refactoring manuali costituiscono una parte fondamentale nello sviluppo di un software poiché permettono di diminuire l'importanza del design iniziale; è possibile, infatti, procedere "parallelamente" sia allo sviluppo del codice sia allo sviluppo del design del progetto software.

3.2 VANTAGGI

Ecco i principali vantaggi derivanti dall'utilizzo del refactoring, come sostenuto anche da Sandro Pedrazzini[3]:

- Miglioramento del design;
- Codice più leggibile;
- Si "scoprono" errori in anticipo;
- Si programma più velocemente.

3.2.1 MIGLIORAMENTO DEL DESIGN

Eseguire brevi e regolari cicli di refactoring aiuta a migliorare il design del progetto software che si sta realizzando. Così facendo, infatti, sia la struttura che la leggibilità del codice vengono mantenute ai più alti livelli qualitativi. Un altro aspetto fondamentale risiede nell'eliminazione di eventuali duplicati. Tali cancellazioni, tuttavia, non provocheranno un immediato aumento nella velocità di sviluppo del software, ma produrranno effetti a medio-lungo termine. Eliminando i duplicati, infatti, si garantisce che il codice non presenti ripetizioni inutili, evitando, dunque, di "confondere" il programmatore in eventuali future modifiche.

3.2.2 CODICE PIÙ LEGGIBILE

Non è interesse dei componenti di un team di sviluppo scrivere codice illeggibile. Se il codice è leggibile o facilmente comprensibile, sarà più semplice eseguire le correzioni auspiccate o, ad esempio, far analizzare il codice ad altri colleghi

programmatori. La leggibilità è un fattore di fondamentale importanza. È conveniente, infatti, far eseguire più cicli di refactoring al software, ma possedere codice leggibile, piuttosto che avere un software che risponde prontamente ai comandi dell'utente, ma che ha meno leggibilità.

In alcuni casi, il refactoring viene utilizzato anche per comprendere meglio porzioni sconosciute di codice. È possibile, infatti, che alcuni membri di un team di sviluppo non abbiano ben chiaro il funzionamento di una determinata porzione di codice e decidano, ad esempio, di eseguirne il refactoring per cercare di comprenderne meglio il comportamento.

3.2.3 *SI "SCOPRONO" ERRORI IN ANTICIPO*

Un ulteriore vantaggio del refactoring consiste nel rilevamento di eventuali bug presenti all'interno del codice. Utilizzando tale pratica, infatti, è possibile comprendere meglio la struttura del software che si sta realizzando e, al contempo, individuare eventuali errori o imprecisioni che possono essere sorti all'interno del codice. Alcune volte può non essere necessario eseguire il refactoring se lo scopo è solamente quello di trovare i bug presenti all'interno del codice. Succede, infatti, che tali errori o imprecisioni siano talmente evidenti e facili da risolvere da catturare immediatamente l'attenzione del programmatore.

3.2.4 *SI PROGRAMMA PIÙ VELOCEMENTE*

Il principale vantaggio derivante dall'utilizzo del refactoring, tuttavia, consiste nello sviluppare più velocemente il codice relativo al software che si sta realizzando. Secondo Sandro Pedrazzini[3], infatti, grazie alla presenza di un buon design, di un'alta qualità del software e di una buona leggibilità del codice, è possibile sviluppare codice più rapidamente. In caso contrario, è possibile progredire velocemente per un breve periodo di tempo, ma poi, inevitabilmente, lo sviluppo rallenta. Gran parte del tempo a disposizione dei componenti del team di sviluppo, infatti, sarà utilizzato per eliminare i bug presenti all'interno del codice piuttosto che introdurre nuove funzionalità.

3.3 "CODE SMELL"

Con il termine "code smell" vengono identificate una serie di caratteristiche che può avere il codice sorgente e che sono riconosciute unanimemente come difetto

di programmazione. I "code smell", tuttavia, non sono veri e propri errori di programmazione, bensì debolezze di progettazione che riducono la qualità del software prodotto. La loro identificazione è soggettiva, ossia a discrezione del programmatore. Martin Fowler & Kent Beck[20], infatti, sostengono che:

"If it stinks, change it"

Martin Fowler[20], in particolare, elenca alcuni possibili esempi di "code smell", tra cui è possibile distinguere:

- Duplicated code;
- Long method;
- Large class;
- Long parameter list;
- Switch;
- Lazy class.

3.3.1 DUPLICATED CODE

Un possibile esempio di "code smell" è il codice ridondante (o "duplicated code"). Succede spesso che, all'interno di una o più classi, siano presenti porzioni di codice identiche o semplicemente ridondanti. In particolare, se due metodi della stessa classe contengono analoghe istruzioni, allora è opportuno applicare il refactoring "Extract Method". Così facendo, infatti, viene creato un nuovo metodo contenente le suddette istruzioni ridondanti e, al contempo, vengono modificati i metodi iniziali. Le istruzioni ridondanti, infatti, saranno opportunamente sostituite dalle chiamate al nuovo metodo. Un altro esempio di codice ridondante potrebbe riguardare una classe ed alcune sue sottoclassi. In tal caso, occorre applicare i refactoring "Extract Method" e "Pull Up Field". Così facendo, infatti, verrà creato un nuovo metodo, contenente le istruzioni ridondanti, che sarà contenuto nella superclasse e non nelle varie sottoclassi.

3.3.2 LONG METHOD

Nella moderna programmazione ad oggetti vengono spesso utilizzati metodi abbastanza corti per facilitare la comprensione e l'utilizzo del software che si

sta sviluppando. A tal proposito, Martin Fowler[20] sostiene che "whenever we feel the need to comment something, we write a method instead ". Così facendo, infatti, è possibile ridurre la distanza semantica tra ciò che il metodo fa e come lo fa. È possibile, dunque, concludere che i metodi contenenti molte linee di istruzioni sono un altro segnale di "code smell".

3.3.3 *LARGE CLASS*

Si parla di "large class" quando ci si riferisce alle classi che fanno anche quello che non dovrebbero fare. Una "large class" può essere vista come un insieme di troppe variabili d'istanza che producono una notevole quantità di codice ridondante. Se all'interno del progetto software sono presenti una o più "large class", allora occorre che i componenti del team di sviluppo verifichino i possibili elementi in comune tra le suddette classi. In caso affermativo, occorre estrarre le linee di codice in comune ed inserirle nella corrispondente superclasse. Così facendo, infatti, viene notevolmente ridotta la lunghezza delle classi in questione.

3.3.4 *LONG PARAMETER LIST*

Un'altra cattiva abitudine dei programmatori riguarda il passaggio dei parametri ai metodi. Spesso, infatti, succede che ai metodi siano passati troppi parametri, molti dei quali non necessari per il corretto funzionamento del metodo. Tale abitudine si sviluppò soprattutto nel recente passato quando, ai nuovi programmatori, veniva espressamente insegnato di passare ai metodi tutti i possibili parametri. Ciò accadeva, principalmente, perché i parametri erano definiti globalmente. Attualmente, invece, si ritiene necessario ridurre al minimo indispensabile le suddette liste di parametri. Infatti, se un metodo riceve diversi parametri allora scambia troppe informazioni con l'ambiente che lo usa; in tal modo, esso risulta scarsamente riutilizzabile in contesti diversi. Dunque, un metodo deve avere una brevissima lista di parametri, meglio ancora un solo parametro: sarà il metodo stesso a ricavarsi tutte le altre informazioni.

3.3.5 *SWITCH*

Un altro aspetto da evitare, all'interno dello sviluppo di un progetto software, è l'inserimento di istruzioni condizionali come, ad esempio, l'istruzione switch. La presenza di tali istruzioni potrebbe ridurre notevolmente la qualità del software

che stiamo producendo e, nel caso di switch, introdurre anche problemi di duplicazione. All'interno del codice, infatti, potrebbe comparire più volte la stessa istruzione switch e, dunque, un cambiamento in uno dei suddetti switch, dovrebbe comportare anche il cambiamento nelle altre istruzioni. Una possibile soluzione è il polimorfismo, ossia un meccanismo attraverso il quale è possibile far riferire una qualsiasi variabile ad un determinato oggetto, purché abbiano la stessa interfaccia.

3.3.6 LAZY CLASS

La creazione di una nuova classe, all'interno di un progetto software, comporta ulteriori costi al team di sviluppo. È necessario, infatti, mantenere la suddetta classe e carpirne il comportamento in ogni momento dello sviluppo. Dunque, se una classe non è indispensabile per la creazione di un software, è conveniente eliminarla. Si parla di "lazy class" quando, ad esempio, una classe viene aggiunta inizialmente al progetto software, ma poi mai concretamente realizzata. Anche questo è un segnale di "code smell".

3.4 QUANDO ESEGUIRE IL REFACTORING

Il refactoring è particolarmente utile quando il progetto software degenera al punto da diventare difficilmente gestibile ed estendibile. È difficile, tuttavia, stabilire i requisiti che il sistema software dovrebbe possedere per essere considerato degenerare. È altresì difficile stabilire quanto tempo occorra dedicare all'esecuzione di tale attività. Il refactoring, infatti, è un processo che si svolge in piccoli passi ("little by little") e, dunque, la sua durata non può essere stabilita in un momento antecedente all'inizio dello stesso.

Martin Fowler[20], tuttavia, ha individuato alcune situazioni in cui è necessario eseguire il refactoring. Ecco alcuni esempi:

- quando si aggiungono nuove funzionalità al software;
- quando c'è la necessità di correggere alcuni bug presenti all'interno del codice;
- quando bisogna semplicemente "revisionare" il codice;
- quando si può applicare la "regola del tre".

3.4.1 "LA REGOLA DEL TRE"

Un possibile metodo per stabilire quando eseguire il refactoring può essere applicare la "regola del tre". In caso di duplicazione del codice, infatti, secondo Don Roberts[20] occorre applicare la seguente regola:

" The first time you do something, you just do it. The second time you do something similar, you win at the duplication, but you do the duplicate thing anyway. The third time you do something similar, you refactor "

3.4.2 QUANDO SI AGGIUNGONO NUOVE FUNZIONALITÀ AL SOFTWARE

La pratica del refactoring è fondamentale soprattutto quando vengono aggiunte nuove funzionalità al software che il team di sviluppo sta realizzando. In tale situazione, infatti, occorre comprendere bene quali porzioni di codice occorre preservare e, invece, quali porzioni di codice occorre modificare per poter implementare effettivamente le nuove funzionalità.

3.4.3 QUANDO C'È LA NECESSITÀ DI CORREGGERE ALCUNI ERRORI

Il refactoring è necessario anche quando il sistema rileva la presenza di alcuni bug all'interno del codice che il team di sviluppo sta realizzando. Durante lo sviluppo di un progetto software, infatti, occorre cercare di ridurre o addirittura eliminare gli eventuali bug o imprecisioni che sorgono all'interno del codice. In tal senso, il refactoring è fondamentale poiché consente al software di funzionare correttamente nonostante siano stati rilevati alcuni bug all'interno del codice.

3.4.4 QUANDO C'È LA NECESSITÀ DI REVISIONARE IL CODICE

È opportuno eseguire il refactoring anche quando i componenti del team di sviluppo hanno semplicemente la necessità di revisionare il codice sviluppato. Spesso, infatti, all'interno di un team di sviluppo, possono essere presenti alcuni programmatori che hanno minor esperienza o semplicemente maggiori difficoltà rispetto agli altri. È opportuno, dunque, cercare di aiutarli a comprendere meglio il progetto software che si sta realizzando tramite l'esecuzione di regolari attività di revisione del codice.

3.5 QUANDO NON ESEGUIRE IL REFACTORING

Non sempre è possibile effettuare il refactoring. In alcuni casi, infatti, tale pratica consente di rendere più chiaro ed efficiente il software che si sta sviluppando; in altri casi, invece, il refactoring è di difficile attuazione. Potrebbe accadere, infatti, che il codice sia talmente confuso da privilegiare una totale riscrittura del codice piuttosto che una sua "revisione".

Sebbene non esistano alcune linee-guida a cui attenersi, nel decidere se eseguire o meno il refactoring, è possibile sostenere come sia necessario riscrivere completamente il codice laddove il sistema sia non funzionante e siano presenti così tanti bug da non permettere di stabilizzare il software. Il refactoring, infatti, può essere effettuato solo se il codice lavora, in gran parte, correttamente.

Un'altra situazione in cui è opportuno evitare il refactoring è quando il team di sviluppo si avvicina alla scadenza entro la quale deve essere consegnato il software al cliente. Il tempo, infatti, non sarebbe sufficiente ad eseguire tale attività.

3.6 TIPI DI REFACTORING

Sebbene esistano molte tipologie di refactoring, è possibile distinguere le seguenti tre macro-categorie, come sostenuto anche da Sandro Pedrazzini[3]:

- Refactoring "Extract";
- Refactoring "Push/Pull";
- Refactoring complessi.

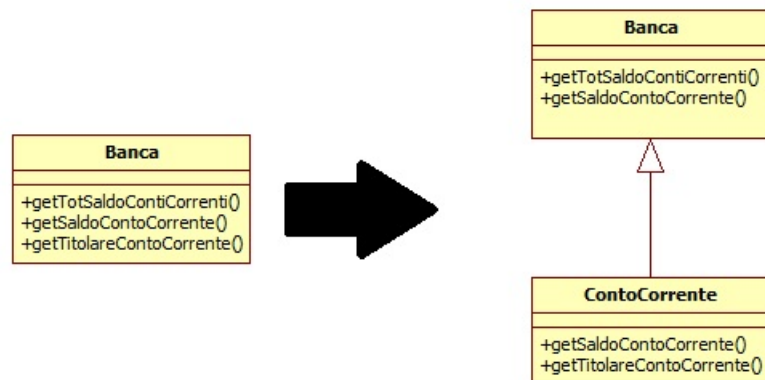
3.6.1 REFACTORING "EXTRACT"

Nella seguente categoria di refactoring risiedono tutti i metodi automatizzabili di revisione del codice che si rivelano utili quando si sviluppa un progetto software. I refactoring di tipo "extract", infatti, hanno in comune l'obiettivo di estrarre qualcosa da una struttura già preesistente.

3.6.1.1 *Extract subclass*

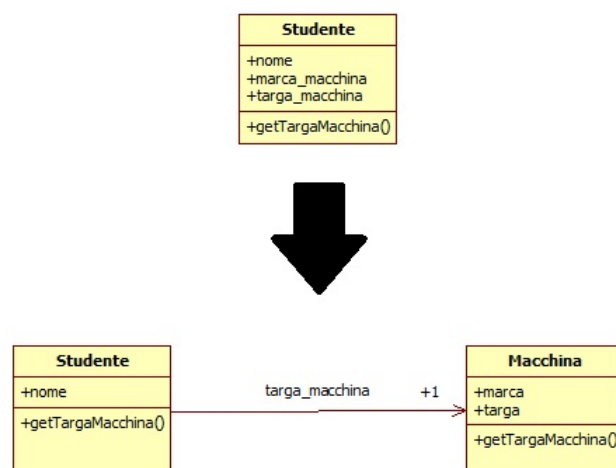
Il refactoring di tipo "extract subclass" prevede la possibilità di estrarre una sottoclasse partendo da una classe in cui sono presenti elementi utilizzati solamente in alcune istanze specifiche. Un possibile esempio di un refactoring

"extract subclass" potrebbe riguardare una banca e la relativa gestione dei conti correnti. Partendo da una classe Banca, infatti, è possibile ricavare una sotto-classe denominata "ContoCorrente" che conterrà tutti i dettagli relativi ad ogni singolo conto corrente.



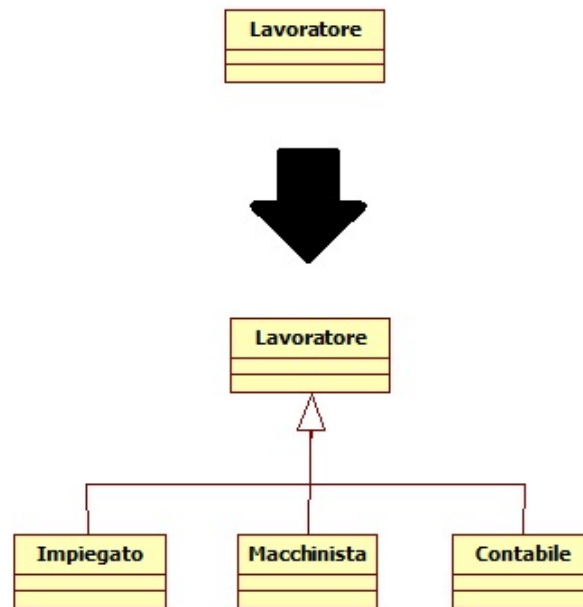
3.6.1.2 Extract class

Il refactoring di tipo "extract class", invece, provvede sia a creare una nuova classe connessa ad una classe già preesistente che a spostare eventuali attributi dalla vecchia classe nella nuova classe. Supponiamo, ad esempio, di avere una classe *Studente* con gli attributi *nome*, *marca_macchina*, *targa_macchina* e con il metodo *getTargaMacchina()*. Tramite il refactoring "extract class", è possibile estrarre, dalla classe *Studente*, una nuova classe *Macchina* che sarà "aggregata" alla classe *Studente*.



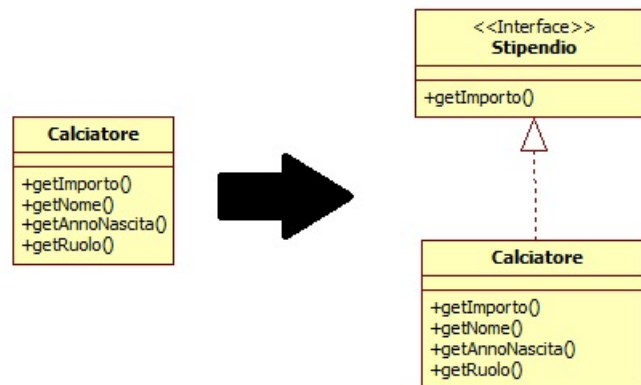
3.6.1.3 Extract hierarchy

Il refactoring "extract hierarchy" provvede a creare, all'interno di un progetto software, una gerarchia di classi in cui ogni sottoclasse rappresenta un caso particolare. Supponiamo di fare riferimento ad una classe pre-esistente denominata *Lavoratore*. Applicando il refactoring "extract hierarchy", ad esempio, è possibile ricavare le corrispondenti sottoclassi denominata *Impiegato*, *Macchinista* e *Contabile*.



3.6.1.4 Extract interface

Il refactoring "extract interface", invece, prevede la possibilità di estrarre un'interfaccia partendo da una classe pre-esistente. Questo tipo di refactoring può essere particolarmente utile quando si vuole separare l'interfaccia dall'implementazione. Supponendo di avere, all'interno di un progetto software, una classe *Calciatore*, allora è possibile ricavare, tramite il refactoring "extract interface", l'interfaccia *Stipendio*.



3.6.1.5 Extract method

Il refactoring "extract method" prevede la possibilità di estrarre un insieme di istruzioni e, conseguentemente, creare un nuovo metodo contenente tali linee di codice. Di seguito viene proposta una piccola porzione di codice in cui viene utilizzato questo tipo di refactoring.

```

void stampa(String nome, String torneo){

    System.out.println("Nome Squadra: "+nome);

    System.out.println("Torneo : "+torneo);

    stampa_risultato();

}
  
```

che diventa:

```

void stampa(String nome, String torneo){

    stampa_dettagli(nome, torneo);

    stampa_risultato();

}

void stampa_dettagli(String nome, String torneo){
  
```

```

        System.out.println("Nome Squadra: "+nome);

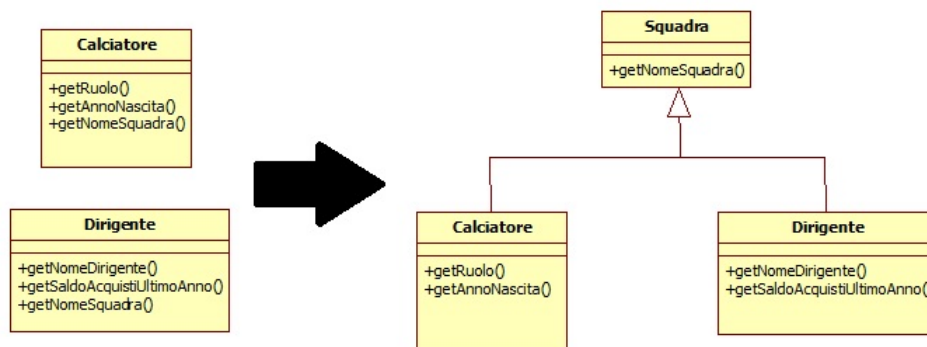
        System.out.println("Torneo : "+torneo);

    }

```

3.6.1.6 Extract superclass

Supponendo di avere due classi indipendenti aventi alcuni elementi in comune, il refactoring "extract superclass" prevede la possibilità di spostare, i suddetti elementi in comune, in una nuova superclasse. Ad esempio, date due classi indipendenti *Calciatore* e *Dirigente*, è possibile, applicando il refactoring "extract superclass" ricavare la relativa superclasse *Squadra*.



3.6.2 REFACTORING "PUSH/PULL"

Dopo aver analizzato i refactoring di tipo "extract", analizzeremo ora alcuni tipi di refactoring che utilizzano le operazioni di Push e Pull.

3.6.2.1 Pull up constructor body

Il refactoring "pull up constructor body" può essere utilizzato laddove due o più sottoclassi abbiano costruttori simili. In tal caso, infatti, è possibile riunire le linee di codice, in comune tra le varie sottoclassi, in un unico costruttore della superclasse. In seguito, naturalmente, occorre anche richiamare, all'interno delle

varie sottoclassi, il costruttore della superclasse. Ecco un piccolo esempio di codice in cui può essere applicato il refactoring "pull up constructor body":

```
public class Caramella extends Dolciumi{

    ...

    public Caramella (String marca, Integer costo){
        _marca = marca;
        _costo = costo;
    }

}
```

che diventa:

```
public class Caramella extends Dolciumi{

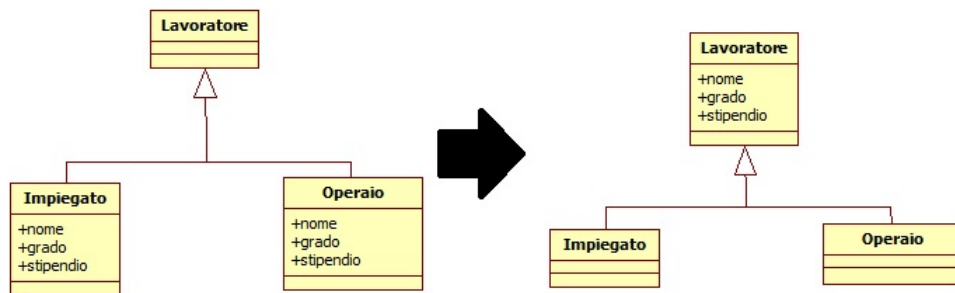
    ...

    public Caramella (String marca, Integer costo){
        super(marca, costo);
    }

}
```

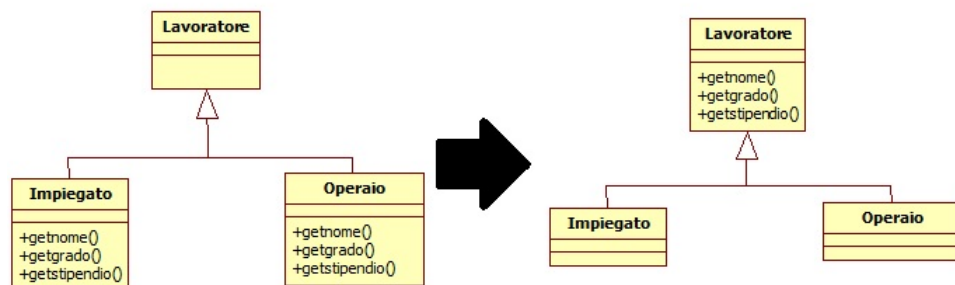
3.6.2.2 Pull up field

Il refactoring "pull up field", invece, si occupa di spostare nella superclasse uno o più attributi in comune tra tutte le varie sottoclassi. Supponiamo, ad esempio, di avere una superclasse *Lavoratore* e due sottoclassi *Impiegato* ed *Operaio*. Se le classi *Impiegato* ed *Operaio* hanno alcuni attributi in comune come, ad esempio, il nome, il grado e lo stipendio, allora è possibile applicare il refactoring "pull up field" per spostare tali attributi nella superclasse *Lavoratore*.



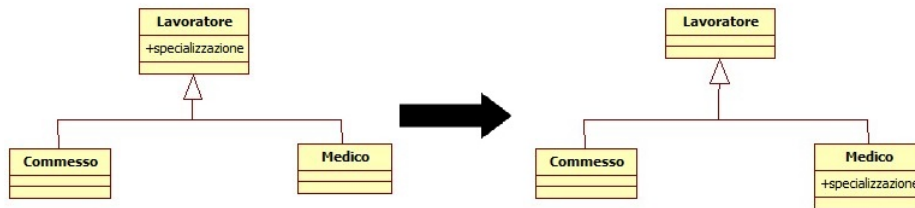
3.6.2.3 Pull up method

Similmente al precedente tipo di refactoring, il refactoring "pull up method" provvede a spostare nella superclasse uno o più metodi in comune tra tutte le varie sottoclassi. Ad esempio, supponendo sempre di avere le classi *Lavoratore*, *Impiegato* ed *Operaio*, è possibile estrarre i metodi `getnome()`, `getgrado()` e `getstipendio()` dalle due sottoclassi, per inserirli nella superclasse *Lavoratore*.



3.6.2.4 Push down field

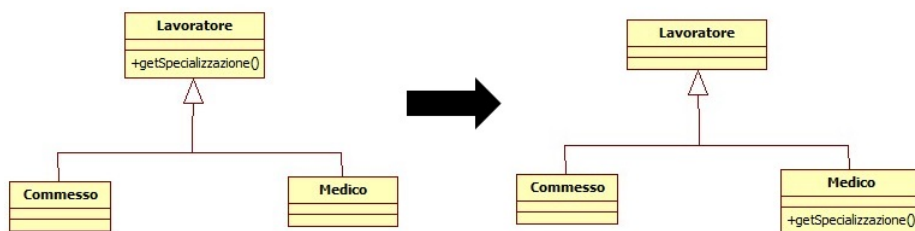
Il refactoring "push down field" si occupa di spostare, dalla superclasse in una sottoclasse, un attributo utilizzato solo nella suddetta sottoclasse. Ad esempio, supponiamo di avere una superclasse *Lavoratore*, con l'attributo *specializzazione*, e due sottoclassi *Commesso* e *Medico*. Tramite il refactoring "push down field",



è possibile spostare il campo specializzazione dalla superclasse Lavoratore nella sottoclasse Medico.

3.6.2.5 Push down method

Il refactoring "push down method", invece, si occupa di spostare un metodo dalla superclasse in una sottoclasse. Supponiamo sempre di avere le due sottoclassi Commesso e Medico e la superclasse Lavoratore. Stavolta, tuttavia, supponiamo che la superclasse Lavoratore contenga anche un metodo `getSpecializzazione()`. Tramite il refactoring "push down method", è possibile spostare il metodo `getSpecializzazione()` dalla superclasse Lavoratore nella sottoclasse Medico.



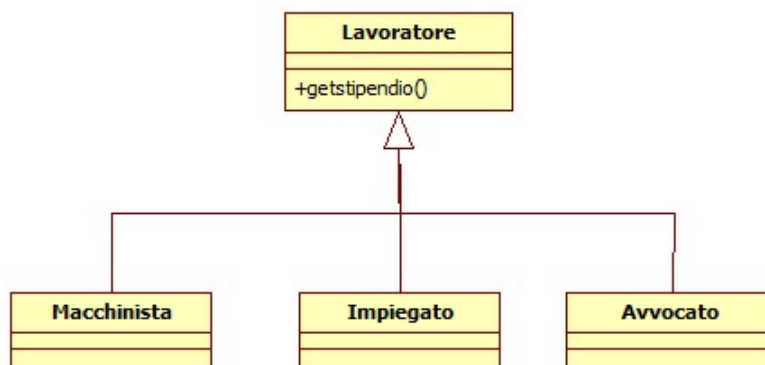
3.6.3 REFACTORING COMPLESSI

Dopo aver analizzato i refactoring di tipo "extract" e di tipo "push/pull", descriviamo ora una categoria di refactoring più complessi da utilizzare. A volte, infatti, i componenti di un team di sviluppo desiderano che il refactoring sia efficace e produca risultati anche su software con una maggiore complessità progettuale. I refactoring "complessi", tuttavia, non producono un'immediata presa visione delle modifiche da parte dei componenti del team di sviluppo. Alcuni esempi di refactoring "complessi" sono:

- replace conditional with polymorphism;
- replace inheritance with delegation;
- replace delegation with inheritance;
- separate domain from presentation.

3.6.3.1 Replace conditional with polymorphism

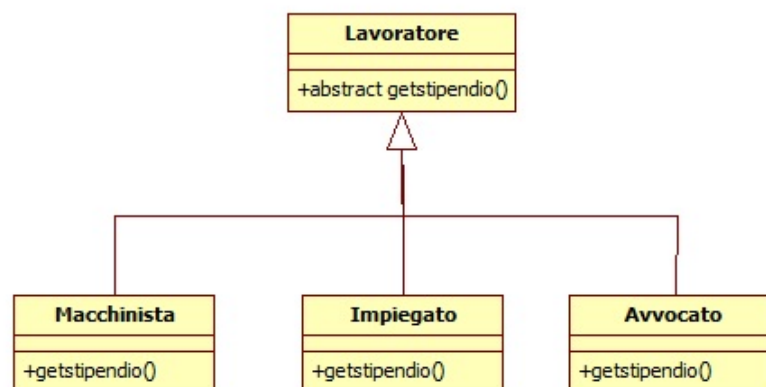
Supponiamo che, all'interno del progetto software che si sta realizzando, sia presente una classe che contiene un'espressione condizionale all'interno di un metodo e supponiamo, inoltre, che la suddetta classe abbia una propria sottoclasse. Tramite il refactoring "replace conditional with polymorphism" è possibile spostare, ogni ramo di tale espressione condizionale, all'interno di un metodo della sottoclasse in questione. In tal caso, il metodo della superclasse diverrà astratto, mentre il metodo della sottoclasse dovrà sovrascriverlo. Ad esempio, supponiamo di avere una classe *Lavoratore* con il metodo *getStipendio()* e tre sottoclassi *Macchinista*, *Operaio* e *Avvocato*.



Il metodo *getStipendio()* della classe *Lavoratore* è così definito:


```
int getStipendio(){  
    switch(qualifica){  
        case "macchinista": return 800;  
        case "impiegato": return 1800;  
        case "avvocato": return 3000;  
    } throw new RuntimeException("Errore");  
}
```

Tramite il "replace conditional with polymorphism", il metodo `getStipendio` della classe `Lavoratore` diviene astratto e, al contempo, nelle sottoclassi `Macchinista`, `Operaio` ed `Avvocato`, viene creato il nuovo metodo `getstipendio()` che sovrascrive il metodo astratto della superclasse.

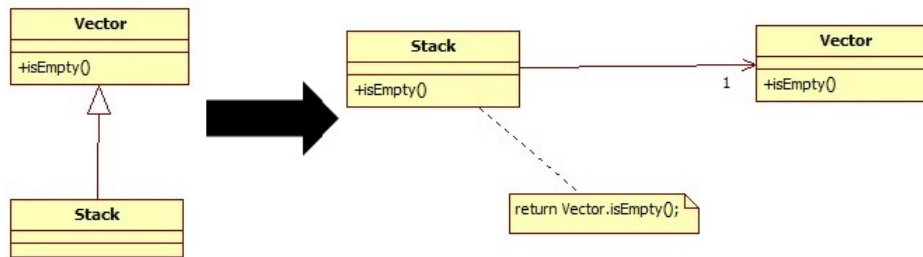


3.6.3.2 Replace inheritance with delegation

Il refactoring "replace inheritance with delegation" si occupa di sostituire la relazione di ereditarietà con la relazione di composizione. Può succedere, infatti, che, all'interno di un progetto software, sia presente una sottoclasse che riutilizzi solamente un piccolo insieme dei vari metodi ed attributi che compongono la superclasse.

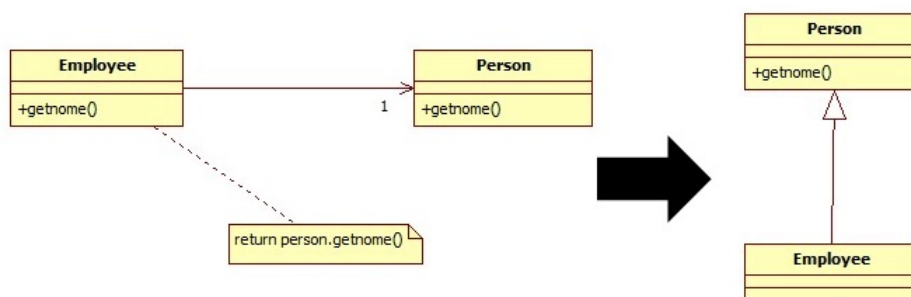
La figura sottostante mostra un esempio di applicazione di questo tipo di refactoring tratto dal libro di Martin Fowler[20]. Supponiamo, infatti, di avere

una superclasse `Vector`, con un metodo `isEmpty()`, ed una sottoclasse `Stack`. Applicando il refactoring "replace inheritance with delegation" otteniamo la trasformazione della relazione di ereditarietà nella relazione di composizione e la conseguente modifica del metodo `isEmpty()`.



3.6.3.3 Replace delegation with inheritance

Il refactoring "replace delegation with inheritance", invece, permette di effettuare l'operazione inversa rispetto al precedente tipo di refactoring. È possibile, infatti, sostituire la relazione di composizione con la relazione di ereditarietà. Ecco un esempio tratto dal libro di Martin Fowler[20]. Supponiamo di avere le classi `Employee` e `Person`, legate da una relazione di composizione. Applicando il refactoring "replace delegation with inheritance" otteniamo la seguente trasformazione:



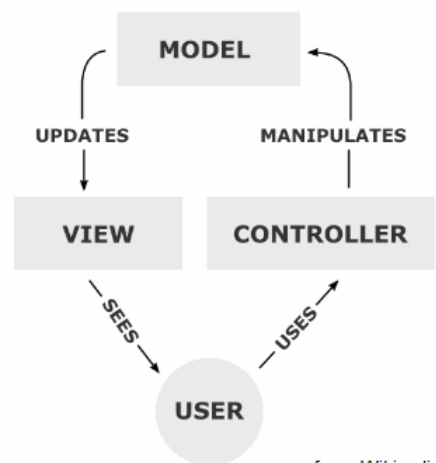
3.6.3.4 *Separate domain from presentation*

A differenza dei refactoring precedenti, il refactoring "separate domain from presentation" ha come obiettivo quello di separare gli elementi di una interfaccia grafica, relativa ad un determinato software, dalla logica del programma. Naturalmente, la complessità di tale refactoring dipende dalla complessità del programma da modificare.

3.7 ESEMPIO DI REFACTORING

Per approfondire ulteriormente il refactoring "separate domain from presentation" viene descritto un esempio di un progetto software in cui occorre derivare una struttura modulare partendo da una struttura unica iniziale. In altre parole, occorre trovare una struttura in cui il modello e la presentazione siano separati. Per fare ciò occorre, però, applicare il modello Model View Controller (MVC) e, in particolare, il pattern Observer.

3.7.1 *MODELLO MVC (Model View Controller)*



Il modello MVC (Model View Controller) è un design pattern utilizzato all'interno dello sviluppo di un progetto software per separare la logica dell'applicazione dall'interfaccia dell'utente. Il principale obiettivo del pattern Model View Controller, infatti, è separare il modello, la visualizzazione del modello (view) ed il

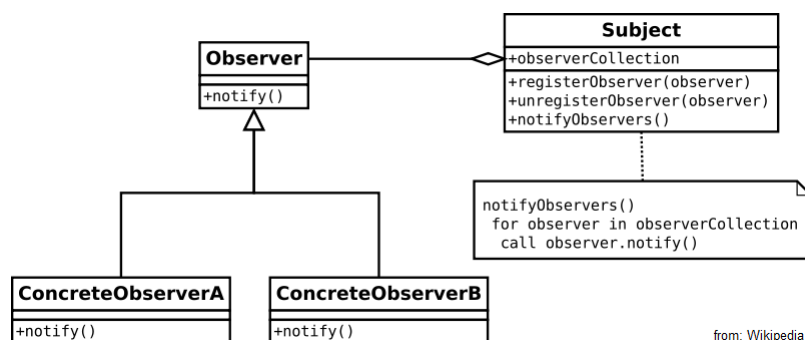
controller relativi alla stessa applicazione. Per modello si intende la parte del codice che si occupa di rappresentare i vari oggetti all'interno dell'applicazione, mentre la view è il modo attraverso il quale il modello viene rappresentato graficamente. Il controller, infine, è il meccanismo attraverso il quale vengono interpretati, ed inviati al modello, i segnali in arrivo dalle varie periferiche di input. La relazione tra view e model, ossia tra il modello e la sua visualizzazione grafica, può essere vista anche come una istanza del pattern Observer.

3.7.2 PATTERN OBSERVER

Il principale obiettivo del pattern Observer è controllare il comportamento di un determinato oggetto (Subject) avvalendosi dell'utilizzo di alcuni oggetti osservatori (Observer). All'interno di questo design pattern, dunque, è possibile distinguere due differenti categorie di oggetto:

- Subject, ossia l'oggetto che viene osservato;
- Observer, ossia un oggetto che viene utilizzato per controllare al meglio lo stato in cui si trova l'oggetto Subject.

Quando un oggetto (Subject) cambia il suo stato, tutti gli altri devono esserne informati e, pertanto, devono aggiornarsi automaticamente. L'oggetto osservato, tuttavia, non deve essere a conoscenza del tipo di Observer con cui sta comunicando.



3.7.3 ESEMPIO

Procediamo ora a descrivere l'esempio concreto, tratto dal libro di Sandro Pedrazzini[3], relativo all'applicazione del refactoring "Separate Domain from

Presentation". Supponiamo di avere un software con un'interfaccia grafica contenente tre campi di testo e supponiamo, inoltre, che i suddetti contenuti siano in relazione tra di loro. Il valore del terzo campo, infatti, corrisponde alla differenza del primo con il secondo. L'obiettivo consiste nel separare la logica del programma dalla presentazione. In altre parole, si vogliono separare le varie relazioni, che intercorrono tra i vari campi di testo, dai campi di testo stessi. Per realizzare la pratica del "refactoring" occorre:

- Modificare la vecchia interfaccia grafica in modo da separare gli osservatori dall'oggetto osservato;
- Eseguire le classi di test;
- Isolare l'accesso ai dati del modello tramite i metodi get e set;
- Eseguire le classi di test;
- All'interno del modello, occorre creare i nuovi campi tramite i metodi get e set, spostando opportunamente le varie funzioni dell'applicazione;
- Eseguire le classi di test.

Ecco il codice iniziale:

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JFrame;
import javax.swing.JTextField;

public class Esempio extends JFrame {

    private JTextField fattore1;
    private JTextField fattore2;
    private JTextField differenza;

    Esempio(){
        ...
    }

    /**
     * metodo utilizzato quando si modifica il contenuto del primo
     * campo di testo
     */
}
```

```
* e, di conseguenza, si vuole aggiornare il valore della
    differenza
*/
void start_check(){
    calcola_differenza();
}

/**
 * metodo utilizzato quando si modifica il contenuto del secondo
    campo di testo
 * e, di conseguenza, si vuole aggiornare il valore della
    differenza
 */
void end_check(){
    calcola_differenza();
}

/**
 * metodo utilizzato quando si modifica il contenuto del terzo
    campo di testo
 * e, di conseguenza, si vuole calcolare il valore del secondo
    fattore
 */
void check_differenza(){
    calcola_fattore2();
}

/**
 * Dati i due addendi, questo metodo calcola effettivamente la
    differenza
 */
void calcola_differenza(){
    int fatt1 = Integer.parseInt(fattore1.getText());
    int fatt2 = Integer.parseInt(fattore2.getText());
    differenza.setText(String.valueOf(fatt1-fatt2));
}

/**
 * Dato il primo fattore e il valore del risultato, il metodo
    calcola_fattore2()
 * provvede a calcolare il valore del secondo fattore
 */
void calcola_fattore2(){
    int fatt1 = Integer.parseInt(fattore1.getText());
    int dif = Integer.parseInt(differenza.getText());
    fattore2.setText(String.valueOf(dif-fatt1));
}
```

```
/**
 * Tramite questa classe l'utente può modificare il valore nel
 * campo
 * di testo del secondo fattore e poi premere il tasto enter della
 * tastiera
 *
 */
class EndAction implements ActionListener{
    public void actionPerformed(ActionEvent event){
        end_check();
    }
}

/**
 * Tramite questa classe l'utente può modificare il valore nel
 * campo
 * di testo del primo fattore e poi premere il tasto enter della
 * tastiera
 *
 */
class StartAction implements ActionListener{
    public void actionPerformed(ActionEvent event){
        start_check();
    }
}

/**
 * Tramite questa classe l'utente può modificare il valore nel
 * campo
 * di testo della differenza e poi premere il tasto enter della
 * tastiera
 *
 */
class SumAction implements ActionListener{
    public void actionPerformed(ActionEvent event){
        check_differenza();
    }
}
}
```

Dopo aver definito opportunamente il codice, è possibile iniziare il refactoring "Separate Domain from Presentation".

La prima fase di tale processo prevede la modifica della vecchia interfaccia grafica (GUI), in modo che gli oggetti osservati e gli oggetti osservatori siano

separati gli uni dagli altri. Occorre, dunque, creare le classi Modello e View servendosi opportunamente dei pattern Observer e Model View Controller.

Dopo aver completato la prima fase del refactoring, occorre compilare ed eseguire le classi di test al fine di verificare la correttezza e le funzionalità del software. In caso affermativo, è possibile procedere con il meccanismo di refactoring, isolando l'accesso ai dati del modello tramite i metodi get and set. In caso negativo, occorre analizzare adeguatamente il codice e trovare l'errore che causa il failure del test.

Successivamente, è possibile creare, all'interno del modello, i nuovi campi tramite l'utilizzo dei metodi get and set e spostare nel modello le funzioni dell'applicazione. Attraverso l'esecuzione di tali operazioni, dunque, abbiamo creato concretamente la classe Modello. Tale classe, infatti, precedentemente conteneva solo la lista che permetteva di gestire gli oggetti Observer. In altre parole, abbiamo proceduto allo "spostamento della logica", ossia allo spostamento dei metodi `calcola_differenza()` e `calcola_fattore2()` nella classe Modello. In seguito, naturalmente, occorre modificare opportunamente la classe View, oltre alla classe Modello. Occorre, infatti, collegare le chiamate get e set della view con quelle del modello.

Il codice risultante sarà il seguente:

```
import java.awt.*;
import java.util.*;

public class Modello {

    private ArrayList Observers = new ArrayList();
    private String fattore1, fattore2, differenza;

    Modello(){
        fattore1="0";
        fattore2="0";
        differenza="0";
    }

    /**
     * ritorna il valore del primo fattore
     */
    String getfattore1(){
        return fattore1;
    }

    /**
     * setta il valore del primo fattore
```



```

    */
    void setfattore1(String arg){
        fattore1 = arg;
        notifyObservers();
    }

    /**
     * ritorna il valore del secondo fattore
     */
    String getfattore2(){
        return fattore2;
    }

    /**
     * setta il valore del secondo fattore
     */
    void setfattore2(String arg){
        fattore2 = arg;
        notifyObservers();
    }

    /**
     * ritorna il valore della differenza
     */
    String getdifferenza(){
        return differenza;
    }

    /**
     * setta il valore della differenza
     */
    void setdifferenza(String arg){
        differenza = arg;
        notifyObservers();
    }

    /**
     * si aggiunge un elemento alla lista degli osservatori
     */
    public void addObserver(Observer o){
        Observers.add(o);
    }

    /**
     * metodo che provvede a notificare a tutti gli osservatori un
     * aggiornamento
     * dello stato dell'oggetto osservato
     */

```

```

        protected void notifyObservers(){
            Iterator i = Observers.iterator();
            while(i.hasNext()){
                Observer o = (Observer)i.next();
                o.update(this);
            }
        }

        /**
         * Dato il primo fattore e il valore della differenza, il metodo
         * calcola_fattore2()
         * provvede a calcolare il valore del secondo fattore
         */
        void calcola_fattore2(){
            int fatt = Integer.parseInt(getfattore1());
            int dif = Integer.parseInt(getdifferenza());
            setfattore2(String.valueOf(dif-fatt1));
        }

        /**
         * Dati i due fattori, questo metodo che calcola effettivamente la
         * differenza
         */
        void calcola_differenza(){
            int fatt1 = Integer.parseInt(getfattore1());
            int fatt2 = Integer.parseInt(getfattore2());
            setdifferenza(String.valueOf(fatt1-fatt2));
        }
    }

import javax.swing.JFrame;
import javax.swing.JTextField;

public class View extends JFrame implements Observer{

    private JTextField fattore1;
    private JTextField fattore2;
    private JTextField differenza;
    private Modello model;

    View(Modello m){
        model = m;
        model.addObserver(this);
        update(model);
    }
}

```

```
/**
 * restituisce il valore del secondo fattore
 */
String getfattore2(){
    return model.getfattore2();
}

/**
 * metodo che provvede a settare il valore del secondo fattore
 */
void setfattore2(String arg){
    model.setfattore2(arg);
}

/**
 * restituisce il valore del primo fattore
 */
String getfattore1(){
    return model.getfattore1();
}

/**
 * metodo che provvede a settare il valore del primo fattore
 */
void setfattore1(String arg){
    model.setfattore1(arg);
}

/**
 * restituisce il valore della differenza
 */
String getdifferenza(){
    return model.getdifferenza();
}

/**
 * metodo che provvede a settare il valore della differenza
 */
void setdifferenza(String arg){
    model.setdifferenza(arg);
}

/**
 * metodo utilizzato quando si modifica il contenuto del primo
 * campo di testo
 * e, di conseguenza, si vuole aggiornare il valore della
 * differenza
 */
```

```

void start_check(){
    setfattore1(fattore1.getText());
    calcola_differenza();
}

/**
 * metodo utilizzato quando si modifica il contenuto del secondo
 * campo di testo
 * e, di conseguenza, si vuole aggiornare il valore della
 * differenza
 */
void end_check(){
    setfattore2(fattore2.getText());
    calcola_differenza();
}

/**
 * metodo utilizzato quando si modifica il contenuto del terzo
 * campo di testo
 * e, di conseguenza, si vuole calcolare il valore del secondo
 * fattore
 */
void check_differenza(){
    setdifferenza(differenza.getText());
    calcola_fattore2();
}

/**
 * Dato il primo fattore e il valore del risultato, il metodo
 * calcola_fattore2()
 * provvede a calcolare il valore del secondo fattore
 */
void calcola_fattore2(){
    model.calcola_fattore2();
}

/**
 * Dati i due fattori, questo metodo che calcola effettivamente la
 * differenza
 */
void calcola_differenza(){
    model.calcola_differenza();
}

public void update(Modello m){
    fattore1.setText(getfattore1());
    fattore2.setText(getfattore2());
    differenza.setText(getdifferenza());
}

```

```
}  
  
}
```


CONCLUSIONI

Nel corso di questa tesi abbiamo analizzato alcune fra le più note metodologie Agili e le pratiche efficaci che queste metodologie hanno introdotto, con grande successo, nello sviluppo del software, nell'organizzazione di un team di sviluppo e del suo lavoro, nelle interazioni con il cliente.

È innegabile il fatto che l'approccio Agile abbia cambiato completamente il contesto dell'ingegneria del software, sostituendo al classico sviluppo *a cascata* procedure più snelle e veloci nella produzione del codice, maggiore attenzione alla qualità del codice e un effettivo abbattimento dei tempi di sviluppo.

Tuttavia, dopo avere spiegato, in tutta la tesi, i grandi vantaggi dei metodi Agili, in fase conclusiva ci sembra opportuno sottolinearne anche alcuni aspetti critici.

In primo luogo, le pratiche Agili richiedono un team di sviluppatori coeso e collaborativo, non sono realizzabili in un ambiente conflittuale e competitivo, o estremamente burocratico, quale si presenta spesso in molti ambienti di lavoro. In particolare, esse richiedono sviluppatori molto preparati, di alto livello, decisamente interessati non solo a ottenere un prodotto soddisfacente per il cliente, ma soprattutto un prodotto di *qualità*. Queste qualità del software sono molto chiare agli sviluppatori ma spesso incomprensibili per il cliente, per cui talvolta l'adozione della metodologia Agile può finire per creare tensioni fra gli sviluppatori e il committente.

Infine, un punto ancora poco chiaro, e certamente discutibile, dei metodi Agili è un innegabile disconoscimento della fase di *design* iniziale. A nostro parere, questa mancanza di attenzione per la fase di progettazione globale iniziale può creare qualche difficoltà nel caso di progetti di grandi dimensioni, che devono essere suddivisi fra diversi team di sviluppatori. Per questa ragione riteniamo particolarmente interessante una metodologia Agile, ancora poco diffusa, detta "Feature Driven Development", che invece enfatizza la fase iniziale di progettazione proponendone, però una nuova visione. Nella prossima sezione presentiamo alcuni cenni a questa metodologia. Infine, nell'ultima sezione, sintetizziamo possibili sviluppi futuri.

FEATURE DRIVEN DEVELOPMENT

A differenza delle altre metodologie Agili, Feature Driven Development (FDD) è una metodologia in cui vengono fornite, prima di iniziare lo sviluppo del progetto software, le opportune informazioni ed infrastrutture di cui necessitano i vari componenti del team di sviluppo per realizzare adeguatamente il progetto software. Inoltre, come sostenuto anche da Stephen Palmer e John Felsing[14], i vari responsabili del team di sviluppo hanno la facoltà di intervenire durante lo sviluppo del software se si verificano alcuni inconvenienti di programmazione. Essi, infatti, devono essere costantemente aggiornati in merito ai progressi dello sviluppo software.

Le principali fasi che possono essere distinte nella metodologia Feature Driven Development si distinguono in:

- **Sviluppare un modello generale.** Nelle fasi iniziali dello sviluppo software, occorre individuare le principali classi che faranno parte del progetto software, definendo anche i rispettivi ruoli e le varie relazioni che intercorrono tra di esse;
- **Definire una lista di funzionalità.** Dopo aver sviluppato un modello generale del software che si vuole realizzare, occorre definire una lista delle funzionalità che occorre implementare;
- **Pianificare per funzionalità.** Dopo aver sviluppato un modello generale ed aver definito una lista delle funzionalità che occorre implementare, è opportuno assegnare ai vari componenti del team di sviluppo una funzionalità da implementare;
- **Progettare per funzionalità.** Adesso occorre definire con maggiore attenzione le classi descritte nelle precedenti fasi indicando anche, ad esempio, i vari metodi ed attributi che occorre implementare;
- **Sviluppare per funzionalità.** La fase finale del Feature Driven Development riguarda l'effettiva implementazione delle varie funzionalità descritte nelle precedenti fasi di progettazione.

Dunque, nella metodologia FDD un ruolo importante è assegnato alla fase iniziale di progettazione, anche se essa è continuamente rivista e raffinata nelle varie iterazioni. Questa progettazione, guidata dalle funzionalità richieste al software, ha lo scopo di stabilire una chiara comprensione del dominio, in cui gli sviluppatori devono lavorare, e impone di produrre velocemente un modello generale del sistema (formalizzato, ad esempio, in diagrammi UML). FDD differisce dalle altre metodologie proprio nel richiedere questa fase iniziale di analisi e design: il modello risultante costituisce una base comune per gli sviluppatori e un framework concettuale per il progetto.

SVILUPPI FUTURI

Le varie metodologie Agili e le relative pratiche di sviluppo software possono essere ulteriormente migliorate al fine di aumentare la velocità e la qualità nello sviluppo del progetto software. Seguendo le prospettive future indicate da Michael Dubakov[15], vediamo in breve i principali argomenti che occorre adeguatamente analizzare per migliorare tali metodologie.

- o **Maggiore studio delle tecniche di "problem solving"**. Il problem solving, ossia l'abilità di risolvere i vari problemi sorti durante lo sviluppo di un progetto software, è un'abilità fondamentale all'interno dello sviluppo di un progetto. Attualmente, tuttavia, molti sviluppatori preferiscono studiare le varie strategie e tattiche di programmazione, piuttosto che studiare le varie tecniche di "problem solving". All'interno di un team di sviluppo, è indispensabile identificare e risolvere in modo efficace i vari problemi che si possono verificare. Poiché è una delle principali caratteristiche delle metodologie Agili è l'auto-organizzazione delle risorse all'interno di ogni team di sviluppo, allora occorre fornire anche opportuni strumenti di "problem solving" all'interno di tali team.
- o **Riduzione al minimo del ciclo dei feedback**. Dopo aver adeguatamente sviluppato alcune funzionalità, all'interno dello sviluppo di un progetto software, occorre immediatamente conoscere le relative valutazioni del cliente. L'obiettivo, dunque, è minimizzare il "ciclo" dei feedback, ossia il tempo che intercorre tra l'invio dell'ultima release al cliente e l'invio delle corrispondenti valutazioni (o "feedback") da parte di quest'ultimo. In caso contrario, infatti, il cliente potrebbe perdere alcune importanti opportunità di mercato derivanti dall'immediato utilizzo del software che si sta realizzando.
- o **Maggiore studio del dominio (" Domain knowledge ")**. Una maggiore conoscenza del dominio può aiutare notevolmente il team di sviluppo ad intraprendere la giusta "strada" per lo sviluppo del progetto software. Attualmente, tuttavia, i vari membri dei team di sviluppo non si concentrano troppo sullo studio del dominio, sottolineato solo nella metodologia FDD. Di solito, gli sviluppatori della comunità Agile non intendono conoscere esattamente l'entità che si sta sviluppando, ma soltanto ciò che interessa maggiormente il team ai fini dello sviluppo del codice. Una maggiore conoscenza del dominio diminuirebbe notevolmente la probabilità di fare scelte sbagliate e, conseguentemente, aumenterebbe la velocità di sviluppo del software.

BIBLIOGRAFIA

-
- [1] Kent Beck, Test-Driven Development: By Example, Addison-Wesley Professional, 2002
 - [2] Robert C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship, Prentice-Hall/Pearson Education, 2008
 - [3] Sandro Pedrazzini, Tecniche di progettazione agile con Java: design pattern, refactoring, test, Tecniche Nuove, 2006
 - [4] Erich Gamma & Kent Beck, Contributing to Eclipse: Principles, Patterns, and Plug-Ins, Addison-Wesley Professional, 2003
 - [5] Gerard Meszaros, xUnit Test Patterns: Refactoring Test Code, Addison Wesley, 2007
 - [6] Sito web: <http://agilemanifesto.org>
 - [7] Alan MacCormack, Product-Development Practices That Work: How Internet Companies Build Software, MIT Sloan Management Review, Winter 2001
 - [8] Yahya Rafique, TDD Dynamics: Understanding the Impact of Test-Driven Development on Software Quality and Productivity, theses and dissertations, 2011
 - [9] Bobby George & Laurie Williams, A structured experiment of test-driven development, research, Elsevier B.V., 2003
 - [10] Robert C. Martin, Agile Software Development: Principles, Patterns and Practises, Prentice Hall, 2002
 - [11] Kent Beck, Extreme Programming Explained: Embrace Change, Seconda Edizione, Addison-Wesley, 2004
 - [12] Kent Beck, Extreme Programming Explained: Embrace Change, Prima Edizione, Addison-Wesley, 1999
 - [13] Ken Schwaber & Jeff Sutherland, La guida a Scrum, sito web: <https://www.scrum.org/Portals/0/Documents/Scrum%20Guides/2013/Scrum-Guide-ITA.pdf>

- [14] Stephen R. Palmer & John M. Felsing, A Practical Guide to Feature-Driven Development, Prentice Hall, 2002
- [15] Sito web: http://readwrite.com/2007/10/16/the_future_of_software_development#awesm= osLaTW4yezqfor
- [16] Jennifer Stapleton, DSDM - Business Focused Development, Second Edition, Pearson Education, 2003
- [17] Alistair Cockburn, Crystal Clear: A Human-Powered Methodology for Small Teams, Addison-Wesley, 2004
- [18] Tom & Mary Poppendieck, Lean Software Development: An Agile Toolkit, Addison-Wesley, 2003
- [19] Sito web: <http://blog.espm.eu/2013/06/07/il-framework-scrum-un-classico-dei-nuovi-tempi/>
- [20] Martin Fowler, Kent Beck & John Brant, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999
- [21] Michael Feathers, Working Effectively with Legacy Code, Prentice Hall, 2004