UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea Specialistica in Ingegneria Informatica

Tesi di Laurea Specialistica in Ingegneria Informatica

Test Driven Development nei Processi di Sviluppo Software Agile

Anno Accademico 2019

Relatore

Prof Tramontana Porfirio

Candidato
Salvatore Ambrosio
matr. N46002341

Indice

TDD

ALL(CODE):

IS FLAWED {

UNTIL PROVEN FLAWLESS;

}

Introduzione alla tesi

La stesura della tesi ha lo scopo di mostrare come le esigenze di modernizzazione e ingegnerizzazione in abito dello sviluppo software negli ultimi anni sia diventata una necessità, sia per ridurre i rischi di fallimento di progetti da parte delle Software House sia per un sensibile aumento della qualità dei prodotti software da loro forniti.

Verranno mostrate le principali differenze fra le tecniche di sviluppo tradizionali come Modelli a Cascata e i più recenti Approcci Agili cercando di mettere in evidenza vantaggi e svantaggi dell'una e dell'altra.

Il tema centrale su cui ci concentreremo saranno i Test per la verifica della Qualità e Funzionalità del software con le appropriate strutture e caratterizzazioni, e il loro utilizzo nelle Metodologie Agili come nel caso del TDD (Test Driven Development), che verrà sviluppato nel capitolo a lui dedicato.

Infine verrà mostrato lo sviluppo di un software reale attraverso l'utilizzo del TDD mettendo in luce i vantaggi di tale processo di sviluppo, creando gli appositi casi di test e il relativo codice che ci permetterà di testare automaticamente il programma scritto in seguito.

Capitolo 1: Sviluppo Software

1.1 *Introduzione allo Sviluppo Software*

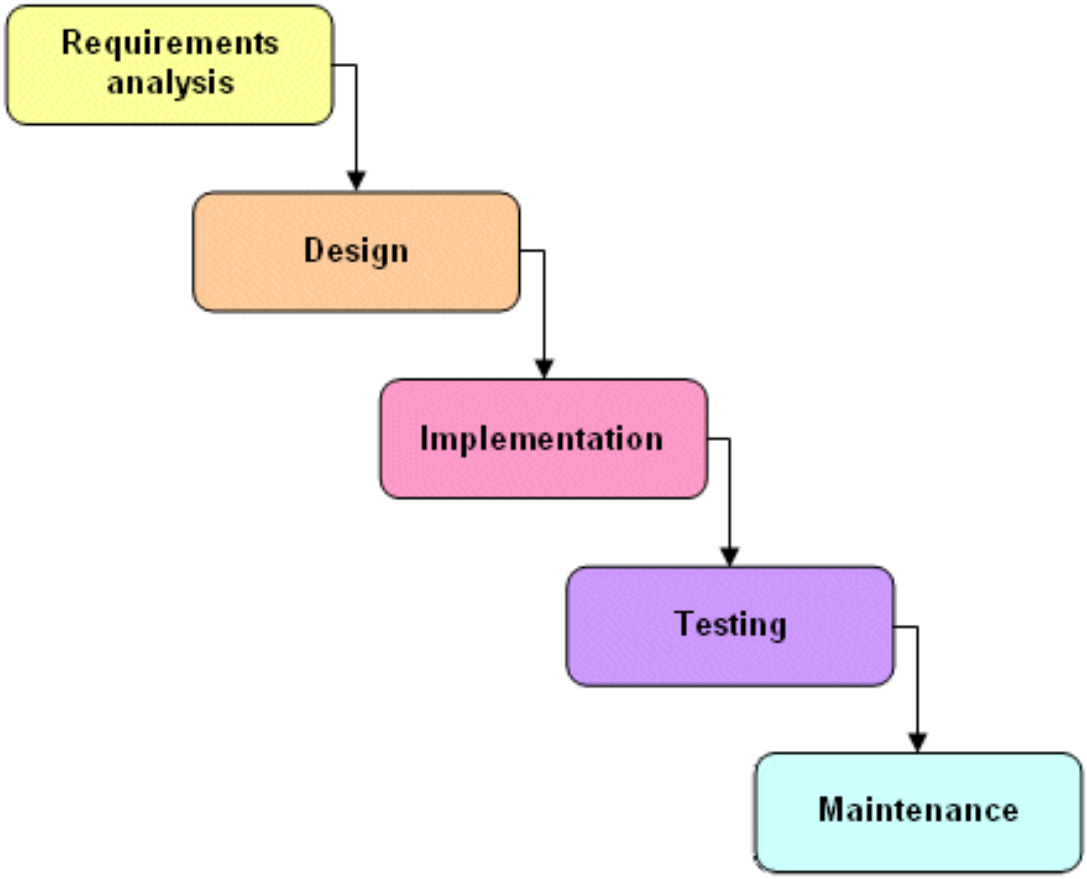
Agli albori dell'informatica, lo sviluppo di software era considerato una attività caotica e soggettiva, dove lo sviluppatore metteva in campo le proprie competenze senza seguire una metodologia di progettazione. Tutto ciò comportava che spesso il codice risultava incomprensibile ad altri sviluppatori che lavoravano allo stesso progetto. Conseguenze dirette erano percepibili nel prodotto finale che si mostrava qualitativamente basso e le comuni pratiche di debug come anche l'ampliamento futuro del software, divenivano impraticabili e quindi costosissime.

Per risolvere questi problemi sono state proposte delle Metodologie che impongono allo sviluppatore di seguire un preciso processo di sviluppo. Le prime Metodologie furono definite Pesanti a causa della grande quantità di documentazione che era prodotta nelle varie fasi di processo, ma con il passare del tempo e per potersi adeguare alle nuove esigenze del mercato, queste si sono evolute portando alla nascita le seguenti:

1. Metodologie Iterative: come il Modello a spirale, Incrementale, Prototipale…..
2. Metodologie Agili: Extreme Programming, TDD…..

1.2 Metodologie Pesanti

Il primo modello appartenente a queste Metodologie e nonché il più rappresentativo è quello a cascata (Waterfall model) strutturato nelle seguenti fasi: Analisi dei Requisiti, Progettazione, Integrazione e Test, Manutenzione.



- **L'Analisi dei Requisiti** ha lo scopo di capire cosa deve fare il software. In genere qui avviene la comunicazione con il cliente o analista, che non risulta continua nel tempo ma avviene solo una volta e sarà una delle differenze con il Manifesto Agile. Il prodotto generato dall'Analisi è il "documento di specifica dei requisiti".
- **La Progettazione (Design)** fa uso dei precedenti requisiti estrapolati, per poter creare la struttura che il sistema dovrà avere. La progettazione ha diversi livelli di dettaglio, in generale vengono utilizzati quelli di livello medio-alto senza riferimenti espliciti al codice poiché si vuole progettare la struttura e non i

componenti particolari del sistema, comprendere come comunicano tra loro attraverso le relazioni, a prescindere del linguaggio che si vuole utilizzare. Il Sistema è descritto graficamente attraverso l'uso dell'UML, che mette a disposizione diverse tipologie di diagrammi, che variano di profondità descrittiva secondo le esigenze. I diagrammi UML sono dotati sia di regole prescrittive che di regole descrittive e quindi regolati da lessico, sintassi e semantica, permettendo così una traduzione automatica in codice di base.

- **L'Implementazione** riguarda la fase di codifica del software. Questa fase può essere legata al linguaggio di programmazione utilizzato dal team di sviluppo per il progetto; ad esempio potrebbe favorire la scelta di un determinato Design Pattern piuttosto che un altro.
- **L'Integrazione e Test** è una fase delicata dello sviluppo in quanto ha lo scopo di integrare i vari componenti precedentemente realizzati e testati, al fine di creare il sistema nella sua interezza. Il sistema verrà testato attraverso adeguate release di versioni alfa e beta; la prima destinata a pochi tester perlopiù interni all'organizzazione, la seconda destinata ad un gruppo di tester esterni (selezionati dall'azienda). Ciò consente di ricevere feedback, quali eventuali bug o debolezze, ma anche un bilancio sui pregi del prodotto.
- **La Manutenzione** è una fase critica nei processi software per i seguenti motivi:
    1. **I Costi** di questa fase possono coprire anche il 50% dell'ammontare totale del software e che a sua volta può essere suddiviso nelle seguenti porzioni: 30% manutenzione correttiva, 20% manutenzione adattiva, 50% manutenzione perfettiva.
    2. **La Regressione** è un problema che non va sottovalutato, in quanto l'apportare modifiche o correzioni al codice spesso porta alla creazione di altri errori o difetti, e quindi in un continuo deterioramento della struttura originale. Altra pratica comune (certamente non giusta) è quella di

aggiornare la documentazione originale in modo non adeguato o peggio ancora quello di non farlo affatto.

Le Metodologie agili permettono di mitigare e in molti casi eliminare problemi del genere come nel TDD.

Le Metodologie Pesanti, con il passare del tempo, si sono evolute dando luogo a modelli come quello Iterativo, che fa uso delle retroazioni per poter risalire la cascata e ripetere determinate fasi, o ancora lo sviluppo Evolutivo-Prototipale che impiega prototipi (per esplorare ad esempio lo spazio dei requisiti) che possono evolvere nel software definitivo o essere "usati" per chiarire precise dinamiche e poi "gettati".

1.3 Agile Programming: Differenze tra Metodi "Classici" ed "Agili"

La metodologia agile ha cercato di alleggerire il pesante carico portato non solo da una copiosa documentazione ma anche dalla rigidità delle strutture appena spiegate. Il suo obbiettivo è quello di trovare un compromesso tra "nessun processo" ed il "troppo trattato".

I pilastri fondamentali di questa metodologia sono fondati su affermazioni ben precise, dichiarate sul Manifesto Agile mostrato qui in basso:

Individuals and interactions over processes and tools

Working software over comprehensive documentation

Customer collaboration over contract negotiation

Responding to change over following a plan

Da questi si formano i **12 Principi** di seguito elencati:

1. La soddisfazione del cliente è sempre la massima priorità e viene raggiunta attraverso una consegna rapida e precisa;
2. L'evoluzione, sotto tutti gli aspetti connessi con il progetto, viene adottata in qualsiasi fase del processo;
3. Un prodotto o servizio viene consegnato con una frequenza più alta;
4. Gli stakeholder e gli sviluppatori collaborano strettamente ogni giorno;
5. Tutti gli stakeholder e i membri del team devono rimanere motivati per ottenere risultati di progetto ottimali. I team dispongono di tutti gli strumenti e il supporto necessari per poter raggiungere gli obiettivi del progetto;
6. Le riunioni faccia a faccia sono considerate la forma di comunicazione più efficiente ed efficace per il successo del progetto;
7. Un prodotto finale funzionante è la misura finale del successo;
8. Lo sviluppo sostenibile si realizza attraverso processi agili in cui team di sviluppo e stakeholder sono in grado di mantenere un ritmo costante e continuo;
9. L'agilità è migliorata attraverso una continua attenzione all'eccellenza tecnica e alla corretta progettazione;
10. La semplicità è un elemento essenziale, in ogni fase del progetto;
11. I team auto-organizzati hanno maggiori probabilità di sviluppare le migliori idee e progetti e di soddisfare i requisiti prefissati;
12. I team effettuano cambi di comportamento per migliorare l'efficacia e l'efficienza del lavoro.

Dalla lettura del manifesto e dei 12 Principi si possono estrapolare le differenze (e non poche) che intercorrono tra le Metodologie Classiche e Agili.

Interattivo: le metodologie Agili mettono al centro dello sviluppo il cliente finale, coinvolgendolo il più possibile attraverso fasi ripetute in momenti temporali diversi. Ciò permette di modificare il progetto in corso d'opera affinando ad esempio i requisiti del

sistema, rendendo le funzionalità più accurate e quindi aumentando la soddisfazione del cliente stesso. Nelle Metodologie pesanti il cliente entra in gioco solo nella fase dei requisiti.

Incrementale: rispetto al passato, quando il software veniva rilasciato nella sua "interezza", ora vengono distribuite versioni incrementali che aggiungono altre funzionalità, le quali permettono di ricevere vari tipi di feedback e quindi di rettificare "il tiro" in corso d'opera. Questo consente di capire la possibile ubicazione dei difetti e di diminuire sensibilmente la regressione del sistema.

Semplice: Nei principi di sviluppo classici capita che lo sviluppatore tende ad implementare funzionalità non richieste, ciò può avvenire per diversi motivi, ad esempio per anticipare funzionalità future o per una cattiva comprensione-interpretazione dei requisiti stessi. Nei processi agili tutto ciò viene abolito; lo sviluppatore si concentra su un requisito ben compreso e lo implementa, permettendo di risparmiare tempo e denaro. Da notare è il fatto che spesso tali funzionalità "aggiuntive" risultano inutili anche in un futuro prossimo, in quanto o i requisiti o il dominio del sistema cambia molto velocemente.

Organizzazione: permette ai vari team di sviluppo di potersi autogestire nei modi più opportuni, consentendo di incrementare la produttività, stimolando gli sviluppatori a trovare soluzioni alternative e creando anche una certa flessibilità in casi di emergenza. Un esempio lampante di tale pratica è quella della Programmazione in coppia sostenuta dall'Extreme Programming, con lo scopo di favorire il controllo reciproco del codice prodotto e la generazione di soluzioni innovative tramite il confronto. Questo approccio permette inoltre ai due componenti una crescita reciproca e continua basata sull'osservazione dell'uno sull'altro (come ho potuto sperimentare personalmente durante il mio percorso alla Apple Developer Accademy).

Quanto detto finora, se non pesato nel modo giusto, porterebbe ad una visione distorta di questi due macro Approcci allo sviluppo del software. Da una parte il male (I metodi Pesanti) e dall'altra il bene (i metodi Agili). In realtà anche i metodi agili presentano criticità, ad esempio dato che ci si concentra principalmente sul breve termine, potrebbe

sussistere il rischio di perdere la visione a lungo termine. La ridotta documentazione potrebbe portare a problemi di comprensione e comunicazione tra gli sviluppatori, soprattutto se vengono effettuate nuove assunzioni nei team di lavoro.

In definitiva sarebbe ideale avere un giusto bilanciamento tra le due parti modulando versatilità, documentazione e controllo sui passi di sviluppo.

Capitolo 2: Il Testing nell'Ingegneria del Software

2.1 Cosa è il Testing

Il Testing è l'attività di specificare, disegnare ed eseguire test. Un test consiste nell'esecuzioni di porzioni di codice per verificarne il funzionamento.

Lo Standard IEEE 610 lo definisce nel seguente modo:

Un'attività nella quale un sistema o le sue componenti sono analizzate sotto particolari condizioni, al fine di osservarne e registrarne il comportamento, valutando i differenti aspetti".

Come brevemente accennato nel capitolo precedente il testing, insieme alla manutenzione, è una fase molto costosa all'interno di qualsiasi processo di sviluppo sia per quantità di tempo che di costi. Alcuni studi indicano che investimenti in questa fase portino a guadagni significativi e a vantaggi per l'organizzazione dal 20% fino al 70% del costo totale di produzione. Il vantaggio di investire nel testing dipende anche dalla capacità di organizzare, nelle prime fasi di sviluppo, politiche di pianificazione laddove il test è considerata un'attività che inizia solo dopo l'implementazione.

2.2 Scopi e problemi del Testing

Lo scopo del testing è quello di scoprire la presenza di malfunzionamenti definiti come un funzionamento diverso da quello previsto dalla sua specifica. I malfunzionamenti nascono dalla presenza di difetti all'interno del software generati da errori umani, ad esempio la battitura errata di un simbolo come un " + " al posto di un " - " o ancora qualche passaggio logico errato che si pensava essere corretto.

Il testing svolge due attività di controllo definite come Verifica e Validazione. Per poterle spiegare, secondo Bohem, bisogna porsi le seguenti domande.

- Stiamo realizzando correttamente il prodotto?

- Stiamo realizzando il prodotto corretto?
- **L'obiettivo della verifica** è il controllo di qualità delle attività svolte durante una fase dello sviluppo
- **L'obiettivo della validazione** è il controllo di qualità del prodotto rispetto ai requisiti del committente, e quindi viene effettuata alla fine del ciclo di sviluppo.

In molti campi dell'ingegneria il testing è semplificato dalle proprietà di continuità del dominio: ad esempio se un ponte resiste ad un carico di 1000 tonnellate allora resisterà anche a carichi inferiori. Nello sviluppo software questa proprietà non è sempre vera dato il dominio discreto, dove piccole variazioni possono portare a valori risultanti errati.
Da ciò nascono ulteriori riflessioni riguardanti i limiti del testing:

Tesi di Dijkstra

Il testing può solo dimostrare la presenza dei difetti, ma non la loro assenza

Si intuisce che per poter verificare la correttezza di un programma, bisogna effettuare un testing esaustivo (ciò dimostrerebbe l'assenza di difetti), il che è impossibile perché richiederebbe un tempo di esecuzione molto ampio, anche della durata di migliaia di anni, per poter concludersi. Ciò che è possibile fare è ridurre le risorse umane e temporali e massimizzare il numero di difetti trovati (con il minimo numero di casi di test).

2.3 Caratteristiche e Tecniche di Testing

In questa sezione trascureremo i Controlli Statici focalizzandoci su quelli Dinamici di interesse applicativo maggiore. I Controlli dinamici o test richiedono l'esecuzione di un programma in un determinato ambiente e con dati di ingresso controllati, l'insieme dei risultati dei vari test fornisce un valore tangibile della qualità del software proposta sotto varie forme come: affidabilità, usabilità, ed efficienza. I test per essere efficaci devono essere strutturati e pianificati in modo da non fallire (fallimento di un test indica che il test non ha trovato problemi e quindi il codice ha superato il test).

Il testing può essere effettuato su vari livelli:

- Livello Produttore
    - Testing di Unità
    - Testing di Integrazione
    - Testing di Sistema
- Livello Cooperativo Produttore-Utente (Privilegiato)
    - Alpha e Beta testing
- Livello Utente
    - Testing di accettazione

In genere le modalità di testing vengono eseguite nell'ordine sopra-rappresentato.

Come è rappresentato un test?

Un caso di test è una tripla costruita come segue:

<input, output, ambiente>

L'insieme dei casi di test costituisce una Test Suite. I Criteri che guidano la progettazione dei test possono essere divisi in due classi principali: Testing funzionale e Testing Strutturali. I due approcci non sono opposti tra loro ma le caratteristiche che possiedono ci consentiranno di scegliere il primo o il secondo in base alla fase di sviluppo in cui ci troviamo. Per poter valutare i risultati ottenuti utilizzeremo uno strumento di paragone, l'Oracolo, che conosce il comportamento atteso per ogni caso di prova.

2.4 Testing Strutturale (White Box)

Il white box testing tiene conto della struttura interna del programma da testare e quindi agisce direttamente sul codice. Diversamente dai black box, i white box possono fornire maggiori indicazioni sulla posizione dell'errore. I criteri su cui si basa fanno uso di analisi del flusso di controllo su appositi grafici (CFG) per determinare il fattore di copertura, cioè quante righe di codice sono state effettivamente eseguite con il mio test. I grafi di flusso sono composti da nodi, a cui sono associati comandi atomici ed archi orientati che determinano il verso del flusso.

Di seguito sono riportati i principali criteri:

- **Test sui comandi**. Riguarda la copertura, intesa come esecuzione dei blocchi del grafo, ogni nodo deve essere percorso almeno una volta.
- **Copertura delle decisioni.** Richiede che ogni arco del grafo sia attraversato almeno una volta; ciò implica che nei nodi decisionali (branch) sia stato coperto sia la decisione vera che falsa
- **Copertura delle condizioni**. Deriva dalla copertura delle decisioni ma è più specifica in quanto non solo vuole la copertura dei rami vero-falso ma anche il modo in cui sceglie la direzione in cui andare. Infatti è comune trovare branch con espressioni booleane composte e quindi è bene testare le combinazioni possibili che permettono di raggiungere sia il ramo true che false.
- **Copertura delle decisioni e delle condizioni**. Questo approccio lega i due precedenti, in quanto i test sufficienti per soddisfare un criterio secondo il primo non sono validi per l'altro e viceversa. Quindi è ragionevole applicare una strategia di composizione dei due criteri di test

Esistono software come CodeCover che permettono di valutare metriche di copertura. CodeCover è un plug-in per Eclipse, che può essere utilizzato anche in modo autonomo, e permette di misurate la copertura per esecuzioni di un'applicazione, affiancandolo a JUnit per la creazione dei casi di test.

2.5 Testing funzionale (Black Box)

Come indicato dalla dicitura "funzionale", possiamo da subito capire come questa tipologia di test si forma sui requisiti e funzionalità che il software deve rispettare e garantire. La struttura interna del componente da testare non verrà presa in considerazione e proprio da questa caratteristica che nasce il nome con cui vengono comunemente chiamati, Black-Box, indicando appunto che il programma verrà trattato come una scatola chiusa.

La selezione dei dati di ingresso (input) avviene a partire dallo studio delle funzionalità del programma. Quando le specifiche del programma sono espresse in linguaggio formale, ricavare i dati di interesse risulta un lavoro semplice spesso anche automatizzabile, nel caso opposto questo lavoro va effettuato manualmente. Quindi nella migliore delle ipotesi è possibile ricavare la definizione dell'insieme di tutti gli input del programma.

I test black box presentano molteplici criteri funzionali:

- **Statistico**. I casi di test sono selezionati in base alla distribuzione di probabilità dei dati in ingresso del programma. Il Test è quindi progettato per esercitare il programma su valori di ingresso più probabili. Questo approccio, se ben utilizzato può portare ad un risparmio economico, ma presenta lati negativi da non sottovalutare, infatti non sempre si possono ricavare le distribuzioni di probabilità dalle specifiche del software.
- **Partizionamento in classi di equivalenza.** Le classi di equivalenza sono gruppi dove gli elementi verranno trattati similarmente dal processo elaborativo. Per definizione, le classi di equivalenza sono delle partizioni, quindi la loro intersezione restituisce una classe vuota. Un semplice esempio può essere la suddivisione in stati validi o non validi delle variabili di ingresso. Questa metodologia non fa uso dell'Oracolo per via della conformazione stessa delle classi di equivalenza.
- **Valori di frontiera**. Anche in questo caso si fa uso d'una partizione in classi di equivalenza dei dati in ingresso. Come dati di ingresso si considerano i valori estremi di ogni classe di equivalenza, ma rispetto al caso precedente qui è possibile che non siano noti i risultati del programma e quindi vi è bisogno dell'oracolo.

Come già detto eseguire un test esaustivo è praticamente impossibile, quindi l'obbiettivo che ci poniamo è quello di approssimare il test ideale. I seguenti criteri sono fondamentali per capire in che modo selezionare i test da eseguire.

- **Affidabilità**. Un criterio di selezione di test C è affidabile per un programma se per ogni coppia di test selezionati da C, T1 e T2, se il test T1 ha successo allora anche T2 ha successo e viceversa.
- **Validità**. Un criterio di selezione di un test C è valido per un programma se, qualora il programma non è corretto, esiste almeno un test selezionato da C che ha successo.

Possiamo affermare che se il criterio è affidabile e valido, allora qualsiasi test T genato da C avrà successo. Sfortunatamente, queste due proprietà danno vita al test ideale-esaustivo e tale modo di procedere può essere applicato solo ai casi banali poiché non è possibile creane per i casi complessi come deducibile dal teorema di Howden.

Teorema di Howden

Non esiste un algoritmo che, dato un programma arbitrario P, generi un test ideale finito, e cioè un test definito da un criterio affidabile e valido

Vediamo i passi da effettuare per poter avere una suite di test con un apprezzato grado di bontà utilizzando il criterio delle classi di equivalenza. Il primo passo è suddivisione in gruppi dove i dati, si presume, vengano trattati allo stesso modo dal processo. In genere i dati in ingresso sono di una determinata tipologia ad esempio numeri, stringhe o booleani, e ciò ci permette di dividerli in uno dei seguenti modi:

- **Intervallo di valori:** Ad esempio una specifica ci impone che la password sia alfanumerica di lunghezza fra 5 e 10 caratteri, allora una classe valida sarà
  - **CV1 5< L <10 con L = Lunghezza**
- **Valore specificato:** potremo avere una classe di equivalenza come la seguente
  - **Cv2 A > 2000 con A = Anno**
- **Insieme discreto**: rappresenta una classe che non mostra continuità matematica come ad esempio
  - **Cv3 S = [Lunedi, Martedì, ……, Sabato, Domenica]**

- **Valore booleano:** rappresenta una classe banale o true o false

Ciascun caso di test deve comprendere il maggior numero di classi valide ancora scoperte e un caso di test per ogni classe non valida.

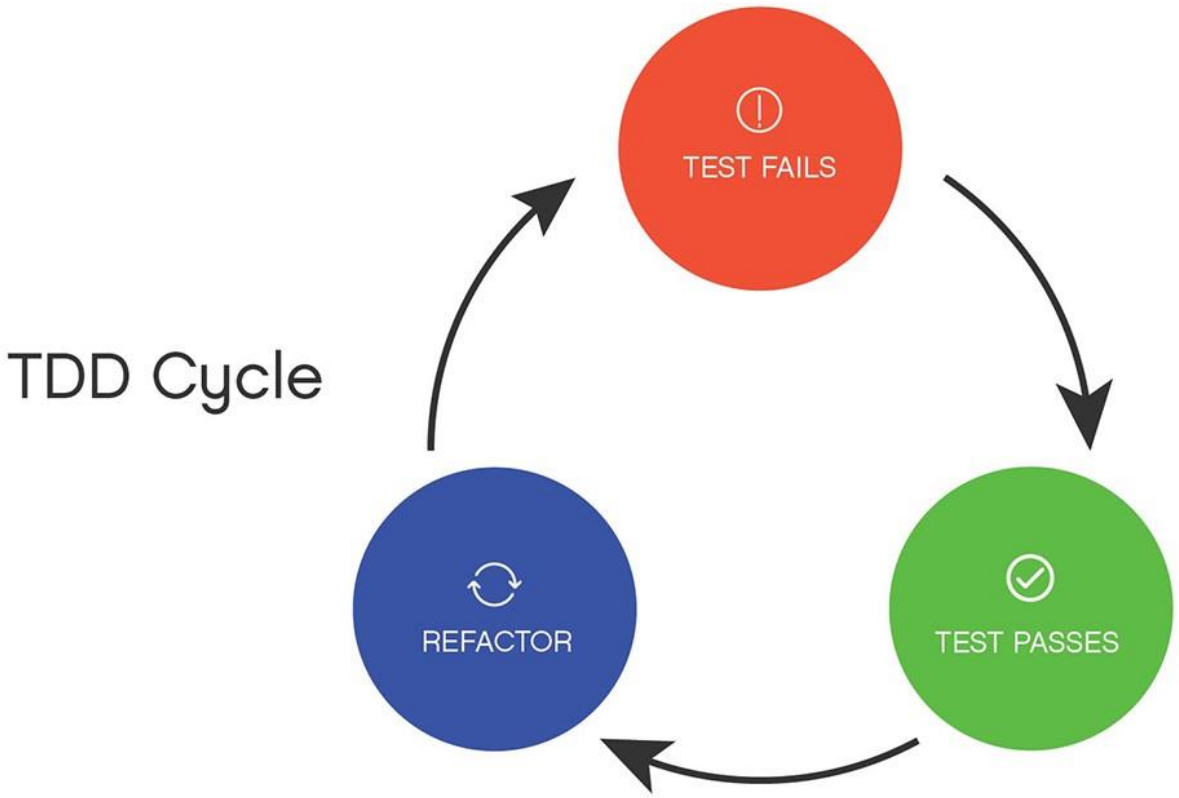
Casi di test che coprono più classi non valide, possono non permettere la comprensione di dove sia il problema.

Il solo utilizzo degli input per la determinazione delle classi di equivalenza a volte non è sufficiente, infatti potrebbero essere presenti delle condizioni, che se non rispettate, possono compromettere la validità del test: un utente potrebbe autenticarsi con l'uso di una password valida, poiché rispetta le specifiche, ma il suo nome utente non ha diritto di accesso all'area riservata. Quindi per la stesura di questi test vi è la necessita di aggiungere altri due campi, Precondizione ed Output Atteso, che permettano di capire in che condizioni possano essere eseguiti.

Capitolo 3: Test Driven Development

3.1 Introduzione al TDD

TDD sta per Test Driven Development ed è una pratica ben conosciuta nell'ambito dei metodi agili essendo un approccio iterativo dove il suo ciclo vitale evolve su tre step fondamentali: Fail – Pass – Refactor, come rappresentato nell' immagine in basso.



Il grafico proposto è semplificato per poter permettere una più naturale comprensione dei punti cardine su cui verte tale metodologia, nel prossimo paragrafo mostreremo quello completo.

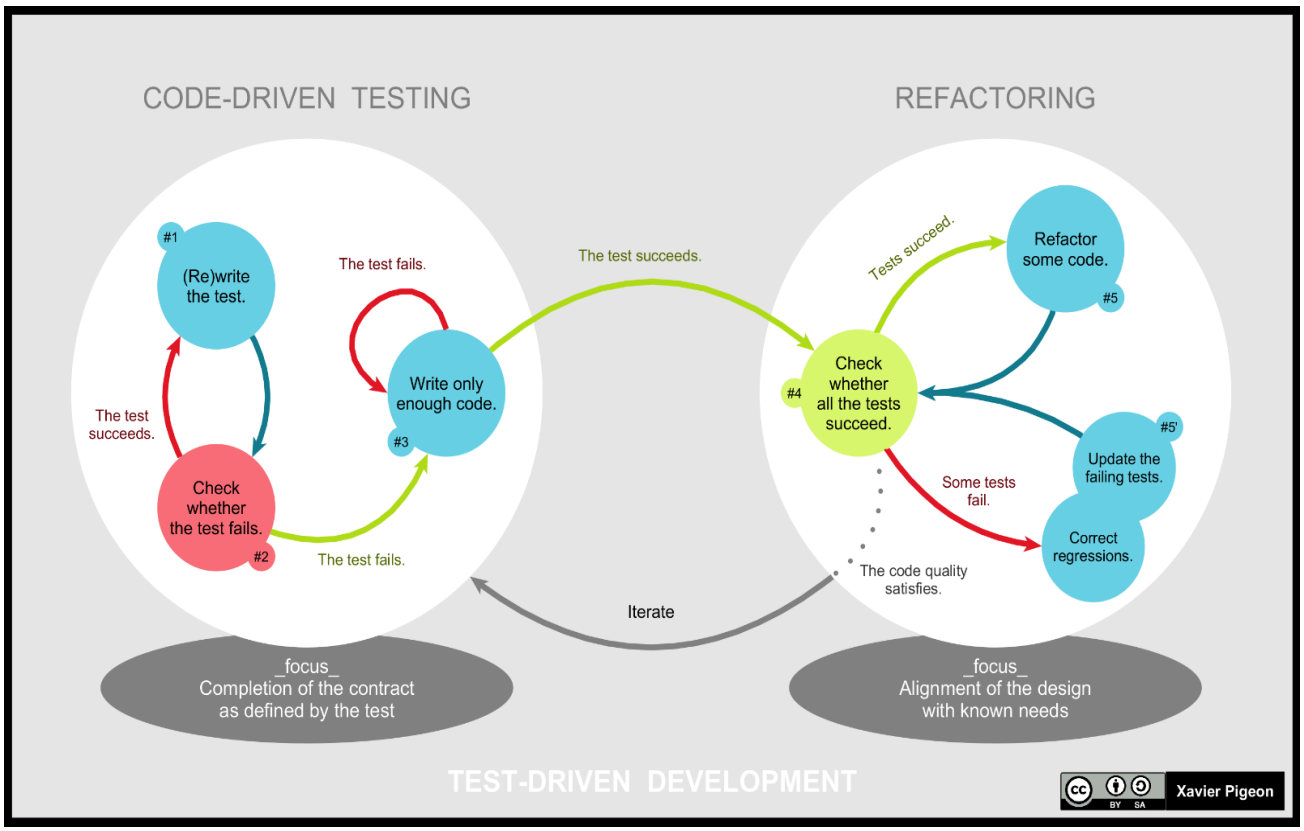
1. Il primo passo da fare è quello di scrivere un test, il più semplice possibile e che riguardi una sola feature, dopodiché va implementato il codice che cerca di coprire

tale feature. Dato che il test viene scritto prima del codice, inizialmente il test fallirà.

2. Aggiungere il codice minimo affinché venga superato il test
3. Infine controllare il codice rifattorizzandolo, cioè riscrivendolo in modo che sia comprensibile da un lettore esterno.

3.2 Il Ciclo Del TDD

Avendo ora una visione generale di come funziona il TDD possiamo introdurre lo schema completo che mostra nei dettagli i tre step prima accennati:



Partendo dal #1 nella figura sopra, la prima fase, è quella di scrittura del test minimo che fallisca. Se il test (successivamente la suite intera) passa, allora potrebbe significare che abbiamo scritto male il test e quindi va ricontrollato passando al #2; questa verifica viene anche chiamata “Test Harness” (va testato il test). Se il test fallisce allora possiamo passare

al punto #3 dove scriveremo il minimo codice affinché il test venga superato e se ciò non avviene ritorniamo al punto #3 andando avanti nella codifica.

Quando il test viene superato si passa al punto #4 dove controlliamo se la modifica apportata al codice dovuto ad un caso di test non abbia compromesso gli altri casi. Se ciò accade allora si verifica un caso di regressione e quindi dobbiamo fare in modo che tutti i test creati fino ad ora passino, modificando opportunamente il codice.

In caso di successo dell'intera test suite possiamo passare al #5 dove avviene la "pulizia" del codice. Bisogna porre attenzione sul fatto che anche dopo il refactoring bisogna ritestare l'intera suite in modo da escludere possibili errori.

3.3 Caratteristiche e Vantaggi del TDD

Mettiamo in luce quelle che sono le caratteristiche principali del TDD che lo differenziano da altri metodi di sviluppo.

Il TDD si basa sulle "**Tre leggi di zio Bob**":

1. You must write failing test before you write any production code.
2. You must not write more of a test than is sufficient to fail, or fail to compile.
3. You must not write more production code than is sufficient to make the currently failing test pass.

Le seguenti leggi sono state definite da Robert C. Martin (noto come Uncle Bob), appartenente al gruppo dei padri delle metodologie agili.

La prima caratteristica è quella di creare inizialmente dei test semplici, cioè se dobbiamo testare una funzionalità che prenda in ingresso una password di lunghezza minima 5 e lunghezza massima 10 e che sia formata da lettere ed almeno 1 numero, non andrò a creare un test che racchiuda tutta questa complessità ma inizialmente scrivo il test per la lunghezza della stringa, creo il relativo codice, poi una volta superato, scrivo il test per il formato e così via. Questa caratteristica viene definita "**line by line granularity**".

La seconda caratteristica è quella che si manifesta realmente nel prodotto finale, nello specifico come si presenta il codice; parliamo della **Testabilità**. Il codice prodotto è

componibile, focalizzato al soddisfacimento dei requisiti e mantenibile, proprietà non ovvie quando il test viene effettuato a posteriori. Lo stesso codice per la produzione del test automatico presenta una natura semplice grazie alla granularità e questo ci permette di usarlo come se fosse una vera e propria documentazione.

Le caratteristiche e il ciclo di sviluppo mostrati, portano inevitabilmente sia a di vantaggi che svantaggi. Tra i primi, abbiamo la presenza di un codice ordinato, testabile, componibile e con un livello di copertura molto alto che porta ad un debugging nella fase di pre-release quasi nullo. I vari test possono essere usati come documentazione e riusati da altri sviluppatori che lavoreranno al progetto senza problemi di incomprensione, ma soprattutto avendo a disposizione il codice delle test suites, il processo può essere automatizzato. I benefici, d'altronde, non riguardano solo il progetto ma anche gli stessi sviluppatori, infatti aumenta la fiducia nel codice scritto (così possono dormire notti tranquille invece di quelle insonni a risolvere bug), aumenta il focus su interfaccia (dettata dalla stesura a priori del test) e sul codice (che fa essenzialmente solo ciò che viene richiesto dal test). Tra gli svantaggi che presenta il TDD vi è la scrittura di molto codice dovuto alla presenza di molti casi di test e il tempo per eseguire quest'ultimo. Secondo alcuni studi diretti su team che utilizzano il TDD il tempo utilizzato per la creazione dei test e la loro esecuzione viene recuperato successivamente nelle fasi di debugging e mantenimento del software, per un guadagno anche superiore al 20%. Naturalmente il TDD deve essere utilizzato dal team intero e non solo una parte, poiché ciò potrebbe portare a contrasti tra i membri. Il fatto che alcuni membri non sappiano usare tale metodologia obbliga comunque l'azienda a dover formare tali soggetti e ciò comporta altri costi da sostenere. Un possibile rimedio è il Pair Programming, che però a sua volta obbliga ad impiegare due sviluppatori sulla stessa risorsa.

3.4 JUnit: Un Esempio Pratico del TDD

Introduzione a JUnit

Per poter sperimentare l'utilizzo del TDD abbiamo bisogno in primis di un ambiente di sviluppo, nel nostro caso Eclipse, ma potremmo optare anche per altri, come IntelliJ IDEA. Eclipse oltre a fornire le funzionalità di base per lo sviluppo software integra un framework chiamato JUnit che ci permetterà di effettuare *unit testing* in modo agevole ed ordinato.

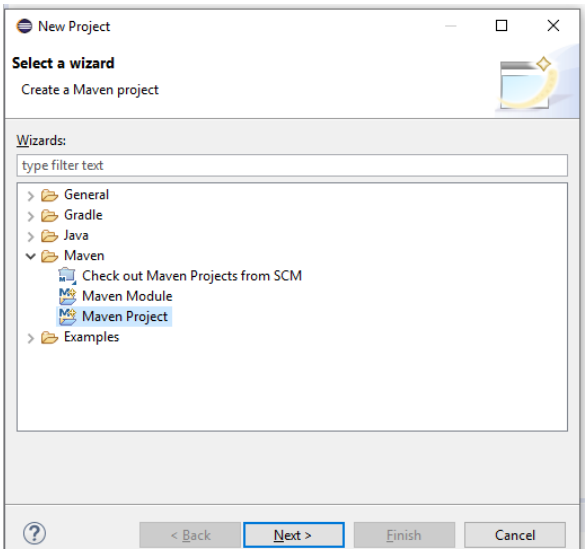
Un primo approccio al testing (che un po' tutti gli aspiranti sviluppatori fanno) è quello di scrivere una classe e poi testarla nel *Main file*. Questo modo di procedere comporta limitazioni non trascurabili, pensiamo ad esempio se volessimo automatizzare l'esecuzione di tale test, sarebbe impossibile. Altro problema riguarderebbe la leggibilità e comprensibilità del codice (un eventuale tester dovrebbe decifrare i *print* ("Tipiche frasi per testare codice") disseminati nel codice).

```java
public class Calculator {

  public int add(int a, int b) {
     return a + b;
  }

}
```

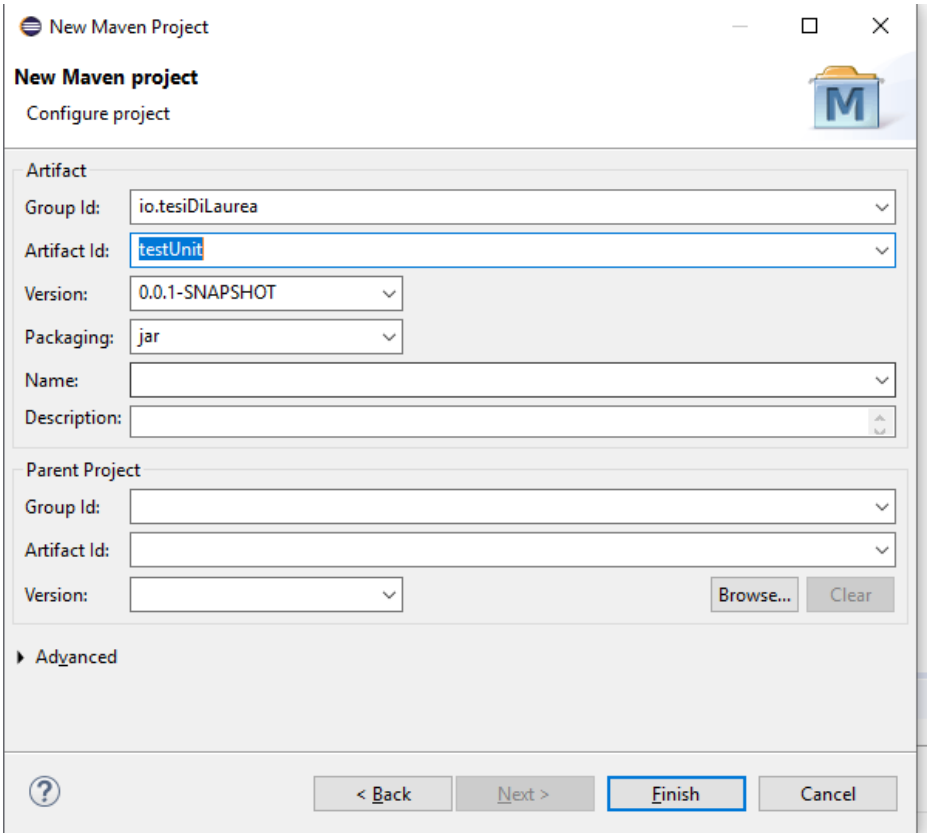
```java
Calculator calc = new Calculator();

int sum = calc.add(0, 1);

if (sum != 1) {
   System.out.println("Test failed");
}
```

Configurazione Eclipse

Per poter utilizzare JUnit bisogna creare un nuovo progetto



Inseriamo i vari dati relativi al nostro progetto come segue:



Optare per un progetto **Maven** e non uno generico, permette di organizzare in modo molto efficiente un progetto java, infatti questa tipologia implementa le seguenti funzionalità:

- Standardizzazione della struttura di un progetto.
- Test ed esportazione automatizzate.
- Gestione e download automatico delle librerie necessarie al progetto.

Fatto ciò siamo quasi pronti per iniziare, ma prima dobbiamo settare le varie dipendenze e proprietà nel file **pom.xml** come in foto.

Progetto in TDD: Requisiti Del Software

Ora siamo pronti a creare il nostro primo test; prenderemo ad esempio la seguente funzionalità: *Si consideri il seguente programma che cerca sottosequenza ordinate (non strettamente) in un vettore di numeri interi positivi inserito in input. Il programma restituisce la lunghezza della massima sottosequenza ordinata e gli elementi da cui essa e fatta. Ad esempio, se gli elementi del vettore fossero: {2, 1, 5, 5, 2, 3, 4}, allora la massima sottosequenza ordinata sarebbe {1,5,5}, di lunghezza 3. In caso di sequenze della stessa lunghezza, deve essere restituita la prima (quella più a sinistra): in questo caso {1, 5, 5} e non {2, 3, 4}. Il programma dovrà estendere le sue funzionalità anche a sequenze di caratteri mostrando comportamenti del tutto simili alle sequenze numeriche; ad esempio: {a, f, g, d, e} allora la massima sottosequenza sarà {a, f, g}.*

Letta la specifica del software possiamo creare una nostra lista “di cose da fare”, intesa come una lista di piccole funzionalità che il programma deve eseguire, creando su queste appositi test e poi pian piano eliminando ed aggiungendo righe nuove (se necessario).

Ad Esempio:

[1,2,3,6,5,1] → [1,2,3,6] [a, f, g, d, e] → [a, f, g]

[5] → [5] [a] → [a]

[ ] → [ ] [-1,4,6,10,3,2] → Eccezione

In tal modo e come se avessimo diviso il problema generale in tanti sotto-problemi più semplici da risolvere (aspetto positivo poiché il ciclo di vita del TDD è breve data la velocita di esecuzioni delle varie fasi).

Primo Test

Iniziamo dalla funzionalità che appare più banale. Creiamo una classe di test e il primo test contrassegnato da *@Test*. Questo deve testare, dato in ingresso un *ArrayList* semplice (composto da un unico valore), che la funzione restituisca lo stesso in uscita.

```
class MyUtilityTest {

    @Test
    void test_NotNull_Imput() {
        MyUtility utility = new MyUtility();
        ArrayList<Integer> sequence = new ArrayList<Integer>();
        sequence.add(1);
        ArrayList<Integer> actual  = utility.getOrderedSequenceNumber(sequence);
        ArrayList<Integer> expected = new ArrayList<Integer>();
        expected.add(1);
        assertEquals(expected, actual);
    }
```

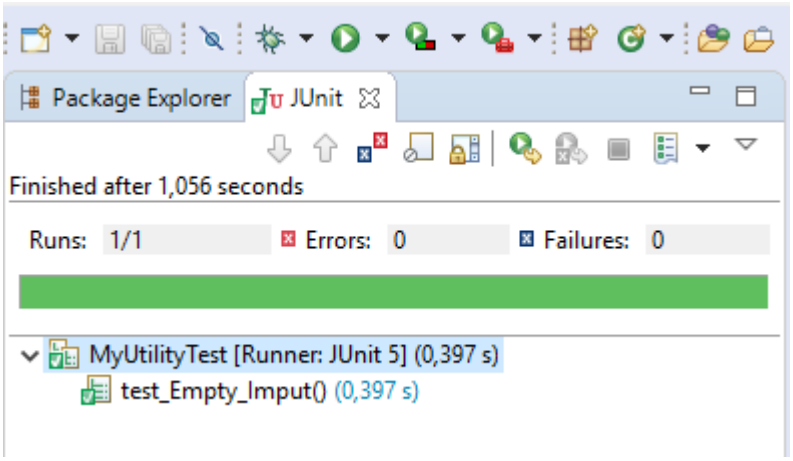
Il test ovviamente **fallisce** poiché non è presente la classe *MyUtility* e il relativo metodo *getOrderedSequenceNumber(),* quindi utilizzando gli strumenti automatici forniti da Eclipse li auto-generiamo.

Coding

Rieseguendo il test questo continua a fallire poiché il metodo richiamato è vuoto, quindi dobbiamo scrivere il *codice minimo* affiche passi il test.

```
    public ArrayList<Integer> getOrderedSequenceNumber(ArrayList<Integer> seq) {
        if(seq.size() > 1)
        {
    //      NON ANCORA IMPLEMENTATO
        }
        return seq;
    }
}
```

Avviato il test, questo viene superato come viene indicato dalla barra verde in basso.



Nel caso in cui il test fosse fallito la barra sarebbe diventata rossa.

Refactoring

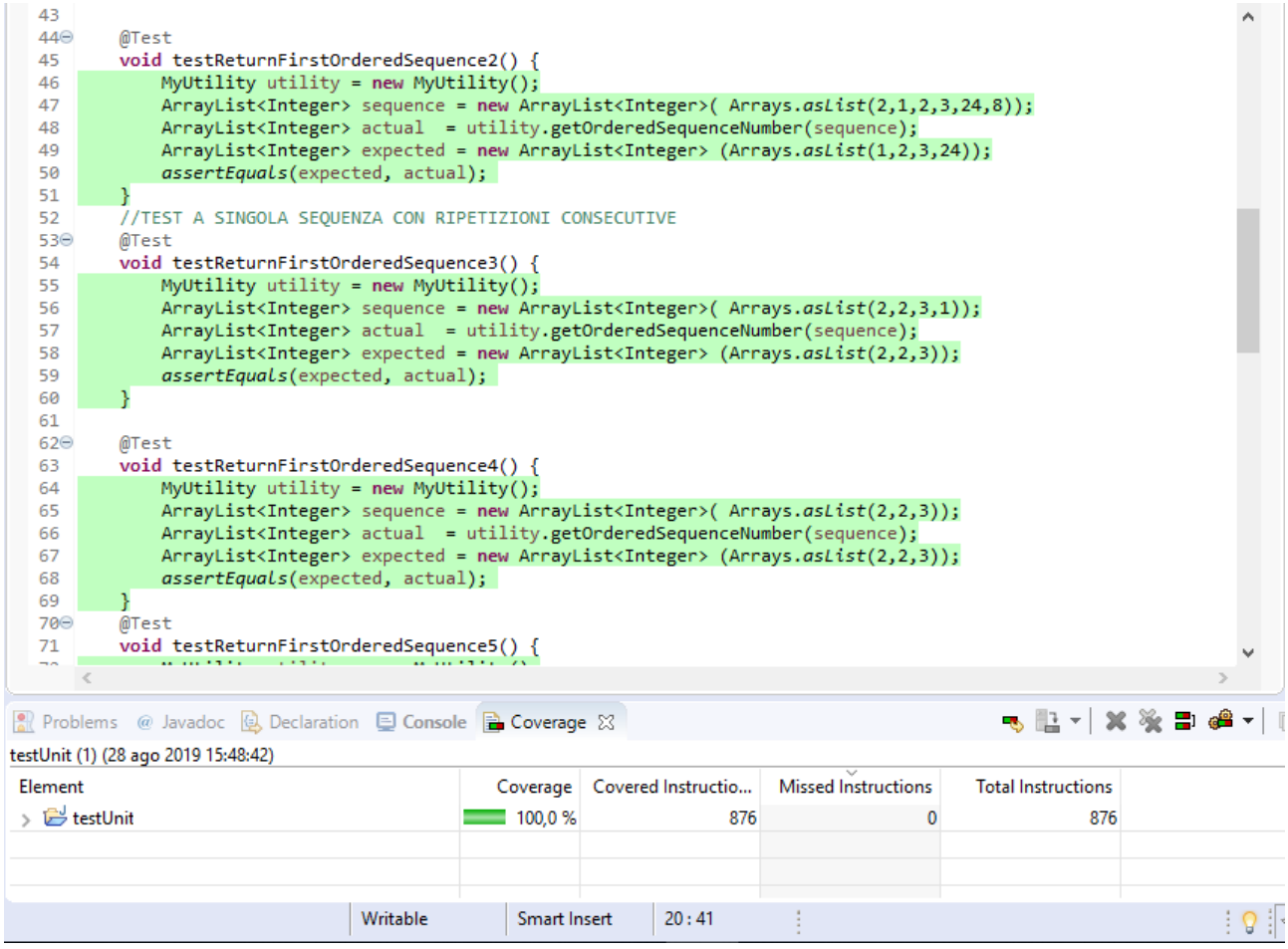
Nel nostro caso i primi cicli svolti non hanno bisogno di refactoring in quanto stiamo sviluppando le funzionalità basilari di ciò che ci è stato richiesto, quindi ci accontentiamo momentaneamente e passiamo all'implementazione del prossimo test.

La fase di refactoring verrà mostrata in modo appropriato quando implementeremo il funzionamento su caratteri.

Test successivi

Il fatto di avere tanti sotto-problemi da risolvere permette di aggiungere il codice mancante con i relativi test un po' alla volta e ciò ci consentirà non solo di avere una casistica di test più ampia, ma anche una maggiore copertura del codice.

Di seguito sono riportati alcuni casi di test nati dalla lista delle specifiche (riportarli tutti sarebbe complicato). La maggior parte dei casi di test sono particolarizzazioni di altri già eseguiti. Per questo motivo saltiamo questa porzione di cicli ripetuti per andare ad una versione del codice un po' più avanzata.



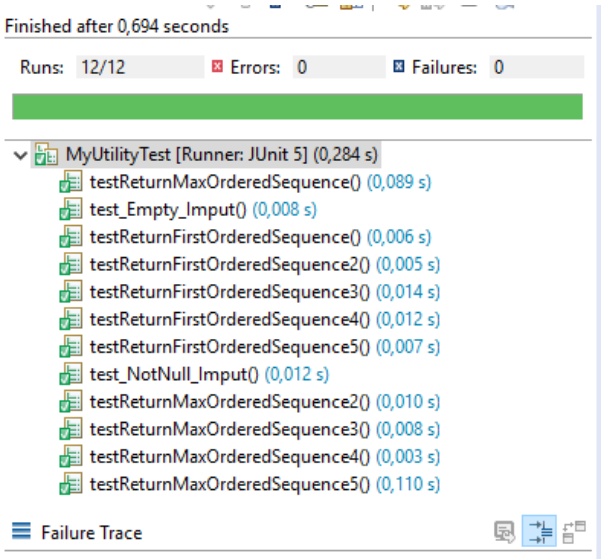
Coding

Questa è la funzione *getOrderedSequenceNumber()* generata attraverso i test sopra citati:

```
public class MyUtility {

	public ArrayList<Integer> getOrderedSequenceNumber(ArrayList<Integer> seq) {
		if(seq.size() > 1)
		{
			int lunghezza = 0;
			int indice = 0;
			int lungMax = 0;
			int indiceMax = 0;
			ArrayList<Integer> seq2 = new ArrayList<Integer>();
			for (int i = 0; i < seq.size(); i++)
			{
				if(lunghezza == 0)
				{
					lunghezza++;
					indice = i;
				}
				else
				{
					if(seq.get(i) >= seq.get(i-1))
					{
						lunghezza++;
					}
					else
					{
						lunghezza = 1;
						indice = i;
					}
				}
				if(lunghezza > lungMax)
				{
					lungMax = lunghezza;
					indiceMax = indice;
				}

			}
			System.out.print(indiceMax);
			System.out.print(lungMax);

			for (int i = indiceMax; i < lungMax+indiceMax; i++)
			{
				seq2.add(seq.get(i));
			}
			return seq2;
		}
		return seq;
	}
```

Controllo validità della Test Suite

Dopo aver apportato aggiunte al codice, ad esempio per effettuare un nuovo test, dobbiamo ritestare tutta la suite per verificare di non aver fatto fallire un test già eseguito in passato.



Refactoring

Anche in questo caso possiamo oltrepassare la fase di refactoring per via della semplicità del codice per passare al test successivo.

Test

Leggendo con attenzione la specifica fornitaci osserviamo come venga imposto che l'ingresso della funzione deve essere fatto da numeri positivi. In questo caso implementiamo un test per rilevare il lancio di un'eccezione del tipo *IllegalArgumentException*.

Per tale scopo JUnit ci fornisce un metodo chiamato *assertThrows* che controlla l'uguaglianza tra due classi di eccezioni.

```
141
142     @Test
143     @Tag("Exception")
144     void testIllegalArgument() {
145         ArrayList<Integer> sequence = new ArrayList<Integer>( Arrays.asList(-3,8,9,2,1,4,5,3,4,8,9));
146         ArrayList<Integer> sequence2 = new ArrayList<Integer>( Arrays.asList(-3,-8,-9,-2,-1,-4,-5,-3,-4,-8,-9));
147         ArrayList<Integer> sequence3 = new ArrayList<Integer>( Arrays.asList(3,8,9,-2,1,4,5,-3,-4,8,9));
148         assertAll(
149                 ()->assertThrows(IllegalArgumentException.class, ()->utility.getOrderedSequenceNumber(sequence)),
150                 ()->assertThrows(IllegalArgumentException.class, ()->utility.getOrderedSequenceNumber(sequence2)),
151                 ()->assertThrows(IllegalArgumentException.class, ()->utility.getOrderedSequenceNumber(sequence3))
152                 );
153     }
154 }
155
```

Coding

Il fallimento del test ci porta ad implementare il seguente controllo nel codice della funzione:

```
5  public class MyUtility {
6
7      public ArrayList<Integer> getOrderedSequenceNumber(ArrayList<Integer> seq) {
8          // NEGATIVE ARGUMENT CONTROLL
9          if(seq.stream().anyMatch(i-> i<0))
10         {
11             throw new IllegalArgumentException("La seguente funzione non supporta valori negativi");
12         }
13
14         if(seq.size() > 1)
15         {
16             int lunghezza = 0;
17             int indice = 0;
18             int lungMax = 0;
```

Controllo validità della Test Suite

Anche in questo caso l'esecuzione della test suite restituisce tutti test verdi.

Con Questo ultimo passaggio abbiamo completato la funzionalità richiesta.

Refactoring

Ora che abbiamo codificato tutte le funzionalità richieste possiamo pensare di applicare un primo refactoring, migliorando la leggibilità del codice e la sua usabilità sia per i futuri test che per altri sviluppatori che utilizzeranno la nostra interfaccia.

Iniziamo:

1. Come primo passo modifichiamo il nome della classe;
2. Creiamo un costruttore;
3. Il ritorno della funzione non deve più essere un array list ma un tipo NumSequence.

Le frecce indicano i cambi effettuati:

```
public class NumSequence {

	public ArrayList<Integer> seq = new ArrayList<Integer>();

	public NumSequence(int[] seq1) {

		for (int i = 0; i < seq1.length; i++) {
			System.out.print(seq1[i]);
			seq.add(seq1[i]);

		}
	}

	public NumSequence(ArrayList<Integer> seq1) {
		seq = seq1;
	}

	public NumSequence getOrderedSequenceNumber() {
		// NEGATIVE ARGUMENT CONTROLL
		if(seq.stream().anyMatch(i-> i<0))
		{
			throw new IllegalArgumentException("La seguente funzione non
supporta valori negativi");
		}

		if(seq.size() > 1)
		{
			int lunghezza = 0;
			int indice = 0;
			int lungMax = 0;
			int indiceMax = 0;
			ArrayList<Integer> seq2 = new ArrayList<Integer>();
			for (int i = 0; i < seq.size(); i++)
			{
				if(lunghezza == 0)
				{
					lunghezza++;
					indice = i;
				}
				else
				{
					if(seq.get(i) >= seq.get(i-1))
					{
						lunghezza++;
```

```
                        }
                        else
                        {
                                lunghezza = 1;
                                indice = i;
                        }
                }
                if(lunghezza > lungMax)
                {
                        lungMax = lunghezza;
                        indiceMax = indice;
                }

        }
        System.out.print(indiceMax);
        System.out.print(lungMax);

        for (int i = indiceMax; i < lungMax+indiceMax; i++)
        {
                seq2.add(seq.get(i));
        }
        NumSequence sequence = new NumSequence(seq2);
        return sequence;
    }
    return this;
  }
}
```

Aggiornamento dei Test

Rieseguendo i test vediamo che andiamo in fase Rossa, dobbiamo modificarli in modo che ritornino Verdi.

Vediamo un esempio di test ristrutturato.

```
    @Test
    void testRefactored() {
        int[] seq1 = {2,1,5,11,24,8};
        int[] seq2 = {1,5,11,24};
        NumSequence mysequence = new NumSequence (seq1);
        NumSequence actual = mysequence.getOrderedSequenceNumber();
        NumSequence expected = new NumSequence (seq2);
        assertEquals(expected.seq, actual.seq);

    }
```

Su questa linea sistemiamo tutti gli altri test.

Test

Nel test notiamo l'accesso ad una variabile interna per poter permettere il confronto, ma sarebbe più giusto rendere la variabile privata ed implementare un metodo che permetta il confronto tra due oggetti dello stesso tipo. Quindi alla lista di funzionalità da implementare aggiungiamo quella dell'equals() per i confronti.

Quindi pensiamo ad un caso di test che permetta di fare ciò:

```
@Test
void test_Equals_between_Sequence() {
    int[] seq1 = {2,1,5,11,24,8};
    assertTrue(new NumSequence(seq1).equals(new NumSequence(seq1)));

}
```

Coding

Ovviamente il test fallisce; bisogna implementare (o meglio overriddare) equals() con il codice più semplice possibile:

```
77    public boolean equals(Object obj) {
78        return true;
79    }
```

Test superato!

Refactoring

Questo modo di scrivere codice, dove non vi è logica ma un semplice valore di ritorno, viene definito Fake It (infatti la nostra funzione è finta).

Quando dobbiamo rimuovere il codice fake? Quando vi è un altro test simile ma con dati diversi e che non passa il test.

Implemento Il resto del codice e rendiamo le variabili interne private.

```java
public boolean equals(Object obj) {
    NumSequence extsequence = (NumSequence) obj;
    if (this.seq.equals(extsequence.seq))
    {
        return true;
    }
    return false;
}
```

Test superato con successo!

Riorganizzazione ed Aggiornamento dei Test

Ora possiamo sistemare il codice di test, eliminando la chiamata ad **equals()**, e lavorare sugli oggetti senza toccare le variabili interne che rendiamo **private**.

```java
25  @Test
26  void testRefactored() {
27      int[] seq1 = {2,1,5,11,24,8};
28      int[] seq2 = {1,5,11,24};
29      NumSequence mysequence = new NumSequence (seq1);
30      NumSequence actual = mysequence.getOrderedSequenceNumber();
31      NumSequence expected = new NumSequence (seq2);
32      assertEquals(expected, actual);
33
34  }
```

I test eseguiti risultano tutti verdi!

Test Esplorativo

Passiamo ora ad implementare la funzionalità che lavora su un vettore composto da caratteri. Scriviamo un primo test per capire se effettivamente posso gestire i confronti direttamente sulle lettere, questo è un test esplorativo e soprattutto personale per comprendere le potenzialità del linguaggio.

```java
49  @Test
50  void test_DimensionLetter() {
51      char a = 'a';
52      char b = 'b';
53      assertTrue(b > a);
54  }
```

Il test passa poiché il confronto viene effettuato implicitamente sul valore ASCII dei caratteri.

Test

Scriviamo un possibile test della funzionalità:

```
56    @Test
57    void test_DimensionLetter2() {
58        char[] seq1 = {'a','b','c','a'};
59        char[] seq2 = {'a','b','c'};
60        AlfaSequence mysequence = new AlfaSequence (seq1);
61        AlfaSequence actual = mysequence.getOrderedSequenceNumber();
62        AlfaSequence expected = new AlfaSequence (seq2);
63        assertEquals(expected, actual);
64    }
```

Il test non passa perché la classe non esiste ancora.

Coding

La cosa più ovvia che possiamo fare per superare il test sarebbe quello di utilizzare il potente strumento “Copia e Incolla” che ci permette di copiare la classe che lavora con i vettori di numeri e poi modificarla opportunamente per farla funzionare con i caratteri. La nuova classe si chiama *AlfaSequence* e di seguito è riportato il codice corretto:

```
public class AlfaSequence {

	private ArrayList<Character > seq = new ArrayList<Character>();

	public AlfaSequence(char[] seq1) {
		for (int i = 0; i < seq1.length; i++) {
			System.out.print(seq1[i]);
			seq.add(seq1[i]);

		}
	}

	public AlfaSequence(ArrayList<Character> seq1) {
		seq = seq1;
```

```
	}

	public AlfaSequence getOrderedSequenceNumber() {
		// NEGATIVE ARGUMENT CONTROLL
		if(seq.stream().anyMatch(i-> i<0))
		{
			throw new IllegalArgumentException("La seguente funzione non
supporta valori negativi");
		}

		if(seq.size() > 1)
		{
			int lunghezza = 0;
			int indice = 0;
			int lungMax = 0;
			int indiceMax = 0;
			ArrayList<Character> seq2 = new ArrayList<Character>();
			for (int i = 0; i < seq.size(); i++)
			{
				if(lunghezza == 0)
				{
					lunghezza++;
					indice = i;
				}
				else
				{
					if(seq.get(i) >= seq.get(i-1))
					{
						lunghezza++;
					}
					else
					{
						lunghezza = 1;
						indice = i;
					}
				}
				if(lunghezza > lungMax)
				{
					lungMax = lunghezza;
					indiceMax = indice;
				}

			}

			for (int i = indiceMax; i < lungMax+indiceMax; i++)
			{
				seq2.add(seq.get(i));
			}
			AlfaSequence sequence = new AlfaSequence(seq2);
			return sequence;
		}
		return this;
	}

	public boolean equals(Object obj) {
		AlfaSequence extsequence = (AlfaSequence) obj;
		if (this.seq.equals(extsequence.seq))
		{
```

```
                return true;
            }
            return false;
        }
}
```

Rieseguendo il test viene superato con successo.

Refactoring

È giunto il momento di riflettere su ciò che stiamo facendo.

Steve Freeman ha sottolineato che il problema con il test e il codice così com'è non è la duplicazione. Il problema è la dipendenza tra il codice e il test: non puoi cambiarne uno senza cambiare l'altro. Il nostro obiettivo è di essere in grado di scrivere un altro test che "abbia senso" per noi, senza dover cambiare il codice.

La dipendenza è il problema chiave nello sviluppo del software a tutti i livelli. Se disponi di dettagli sull'implementazione di SQL di un fornitore sparsa in tutto il codice e decidi di passare a un altro fornitore, scoprirai che il tuo codice dipende dal fornitore del database. Non è possibile modificare il database senza cambiare il codice.

Se la dipendenza è il problema, la duplicazione è il sintomo. La duplicazione molto spesso assume la forma di una logica duplicata, la stessa espressione condizionale che appare in più punti del codice, come successo tra le due classi che stiamo testando. Gli oggetti sono eccellenti per sottrarre la duplicazione della logica.

La duplicazione appare anche nei dati. Vengono introdotte Costanti simboliche per eliminare le dipendenze tra codice e numeri "magici". Dopo aver utilizzato una costante simbolica, il codice non dipende più dal numero effettivo. Si può modificare il numero senza dover toccare il codice ed un caso simile si incontra con l'uso del Fake It che viene poi risolto con la Triangolazione.

A differenza della maggior parte dei problemi, in cui l'eliminazione dei sintomi fa apparire il problema solo altrove in forma peggiore, l'eliminazione della duplicazione nei

programmi elimina la dipendenza. Eliminando la duplicazione prima di passare al test successivo, massimizziamo le nostre possibilità di essere in grado di eseguire il test successivo con una sola modifica.

Quindi il nostro compito è eliminare la duplicazione; una possibile soluzione è di creare una classe padre comune per le due classi che abbiamo e rendere più componenti possibili delle due classi uguali (comporta una generalizzazione) e a quel punto trasportarle nella classe padre.

Il refactoring può essere eseguito a passi diversi a seconda della agilità dello sviluppatore e della sicurezza nelle modifiche che sta apportando, quindi un'intera fase di refactoring può essere divisa in più fasi. In questo caso divideremo il refactoring in quattro parti, che si alterneranno con l'aggiornamento dei test e con la verifica della test suite:

1. Passaggio parametri interni dalle classi figlie a quella padre **Sequence**;
2. Generalizzazione costruttori delle classi figlie e della funzione *equals()*;
3. Generalizzazione funzione *getOrderedSequenceNumber();*
4. Eliminare le classi superflue.

Refactoring 1/4

Creiamo una classe **Sequence** e con questa estendiamo le due già presenti.

Le variabili interne sono entrambe ArrayList ma una di interi e una di caratteri.

Cosa facciamo ora? Utilizziamo i Generics per la creazione della classe padre e degli attributi interni. Faremo uso del tipo generico T e della classe Comparable<T> che ci permettono di fare due cose: la prima è poter utilizzare un tipo generico T in sostituzione dei tipi Integer e Character, la seconda è l'estensione del Comparable<T> che permette di usare la funzione *compareTo()* per poter confrontare gli oggetti passati alla classe.

```java
package testUnit;

import java.util.ArrayList;

public class Sequence <T extends Comparable<T>> {

    protected ArrayList<T>  seq = new ArrayList<T>();

}
```

Estendiamo le classi *NumSequence* ed *AlfaSequence* nel seguente modo:

```java
public class NumSequence extends Sequence<Integer> {


    public NumSequence(Integer[] seq1) {
        for (int i = 0; i < seq1.length; i++) {
            seq.add(seq1[i]);

        }
    }
```

Aggiornamento ed Esecuzione Test Suite

Questo refactoring non comporta alcun tipo di modifica al codice dei Test, infatti rieseguendoli sono tutti verdi quindi possiamo proseguire con il prossimo passo.

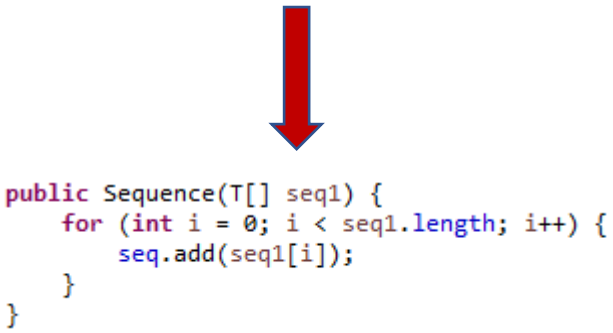
Refactoring 2/4

Generalizziamo i due costruttori, lasciandoli momentaneamente nelle due classi figlie e riportando la versione generalizzata del metodo nella classe padre.

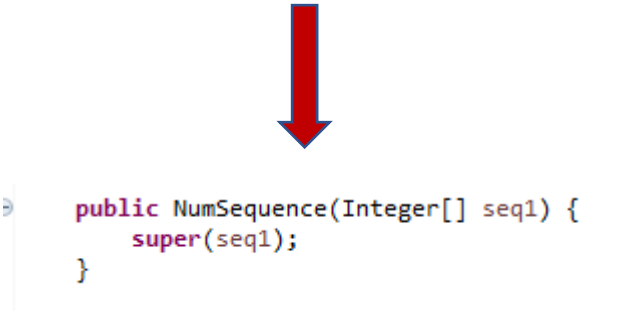
```java
    public NumSequence(Integer[] seq1) {
        for (int i = 0; i < seq1.length; i++) {
            seq.add(seq1[i]);
        }
    }
```

Classe figlia

Forma generalizzata



Classe Padre

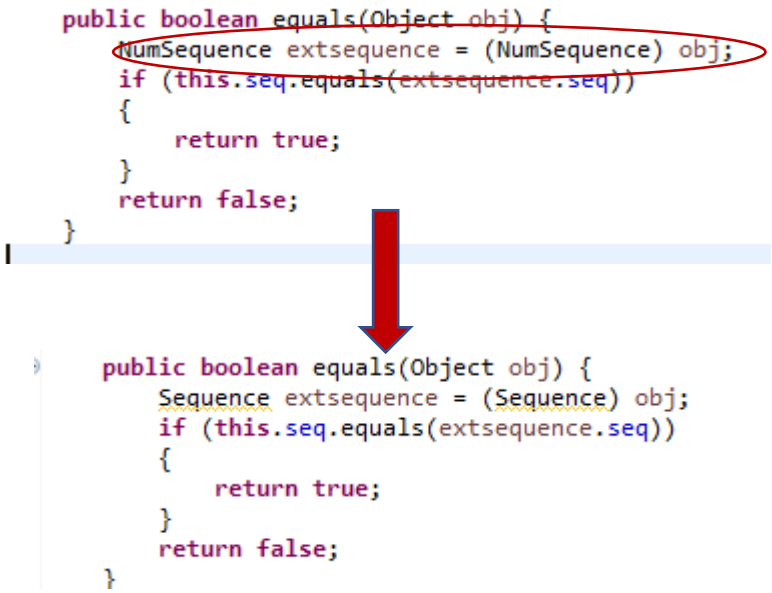


C.Figlia Rifattorizzata

I Test sono VERDI.

NB: le stesse modifiche sono apportate alla classe *AlfaSequence* dove al posto di *Integer* viene usato *Character*.

Ora tocca alla funzione *equals()* alla quale cambiamo i tipi delle variabili e dei vari cast



La funzione della classe figlia può essere eliminata e possiamo usare quella generalizzata della classe padre.

Refactoring 3/4

Ultima funzione da rifattorizzare è *getOrderedSequenceNumber()*.Dobbiamo sostituire gli operatori di confronto con la funzione *compareTo()* ed infine restituire un oggetto di tipo padre **Sequence** e non dei tipi figli.

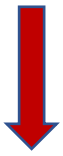
```
public Sequence<T> getOrderedSequenceNumber( ) {
		// NEGATIVE ARGUMENT CONTROLL
		if(seq.get(0).getClass().equals(Integer.class))
		{
			if(seq.stream().anyMatch(i-> (Integer)i<0))
			{
				throw new IllegalArgumentException("La seguente funzione non supporta valori negativi");
			}
		}

		if(seq.size() > 1)
		{
			int lunghezza = 0;
			int indice = 0;
			int lungMax = 0;
			int indiceMax = 0;
			ArrayList<T> seq2 = new ArrayList<T>();
			for (int i = 0; i < seq.size(); i++)
			{
				if(lunghezza == 0)
				{
					lunghezza++;
					indice = i;
				}
				else
				{
					if( (seq.get(i)).compareTo(seq.get(i-1)) >0 )
					{
						lunghezza++;
					}
					else
					{
						lunghezza = 1;
						indice = i;
					}
				}
				if(lunghezza > lungMax)
				{
					lungMax = lunghezza;
					indiceMax = indice;
```

```
                }

            }

            for (int i = indiceMax; i < lungMax+indiceMax; i++)
            {
                seq2.add(seq.get(i));
            }
            Sequence<T> sequence = new Sequence<T>(seq2);
            return sequence;
        }
        return this;
}
```

Riorganizzazione ed esecuzione Test Suite

Se eseguiamo ora i test non tutti passano ad esempio questo

```
    @Test
    void testRefactored() {
        Integer[] seq1 = {2,1,5,11,24,8};
        Integer[] seq2 = {1,5,11,24};
        NumSequence mysequence = new NumSequence (seq1);
        NumSequence actual = mysequence.getOrderedSequenceNumber();
        NumSequence expected = new NumSequence (seq2);
        assertEquals(expected, actual);

    }
```



Quindi risistemiamo i test che falliscono:

```
    @Test
    void testRefactored() {
        Integer[] seq1 = {2,1,5,11,24,8};
        Integer[] seq2 = {1,5,11,24};
        NumSequence mysequence = new NumSequence (seq1);
        Sequence<Integer> actual = mysequence.getOrderedSequenceNumber();
        NumSequence expected = new NumSequence (seq2);
        assertEquals(expected, actual);
    }
```

Refactoring 4/4

Non ci resta che eliminare le classi figlie e utilizzare anche nei test solo la classe padre utilizzando usando come parametri o array di interi o di caratteri a seconda dei test.

Riorganizzazione ed esecuzione Test Suite

Quindi eliminati i due file che contengono le classi figlie, sistemiamo i casi di test in modo da usare solo la classe padre.

```
@Test
void testRefactored() {
    Integer[] seq1 = {2,1,5,11,24,8};
    Integer[] seq2 = {1,5,11,24};
    Sequence<Integer> mysequence = new Sequence<Integer>(seq1);
    Sequence<Integer> actual = mysequence.getOrderedSequenceNumber();
    Sequence<Integer> expected = new Sequence<Integer> (seq2);
    assertEquals(expected, actual);
}
```

```
@Test
void test_RefactoredChar() {
    Character[] seq1 = {'a','b','c','a'};
    Character[] seq2 = {'a','b','c'};
    Sequence<Character> actual = new Sequence<Character>(seq1);
    Sequence<Character> expected = new Sequence<Character>(seq2);
    assertEquals(expected, actual.getOrderedSequenceNumber());
}
```

Abbiamo concluso il progetto ora potremmo integrare nuove funzionalità (se richieste), oppure includere il nostro lavoro in un progetto più ampio.

Componenti JUnit per Riorganizzare i Test

Di seguito mostreremo elementi di JUnit che permettono di avere una scrittura dei test più semplice e comprensibile, agevolando il modo di organizzare ed eseguire l'attività di testing.

- **Assert**: sono funzioni che permettono di verificare l'uguaglianza tra due membri, di cui il primo è il riferimento ed il secondo è quello ritornato dalla funzione da testare. Ci permetterà di non usare If per i controlli.
- **@Before @After**: con questi due tag indichiamo quali metodi devono essere eseguiti prima o dopo altri, dando così un ordine di esecuzione (di default JUnit non esegue i test secondo l'ordine di codifica). Ci può essere utile per allocare

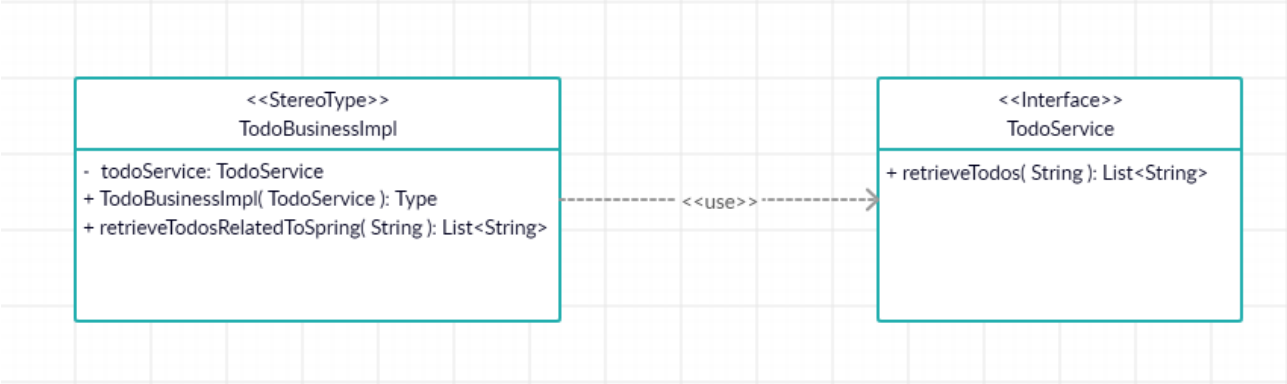
prima di ogni test una determinata risorsa, o di effettuarne un rilascio dopo averla usata, senza dover riscriverlo ogni volta nei vari test.

- **@DysplayName**: viene utilizzato per dare un "secondo nome" al metodo; sarà il nome mostrato nei test case terminati sulla parte sinistra di Eclipse.
- **@Disable**: permette di disabilitare i test che non vogliamo vengano eseguiti, quindi evita ad esempio di commentare codice.
- **@EnabledOn**: indica su quale sistema (e.s sistema operativo) i test verranno eseguiti; se non rispettata il test viene disabilitato.
- **@EnabledIf**: indica la condizione sotto la quale il test si deve svolgere; se non viene rispettata il test viene disabilitato.
- **Assume**: è un comando che permette di verificare la veridicità o falsità di un valore. Può essere usato per esecuzioni di test controllate, ad esempio se in un test abbiamo una parte di codice che controlla la connessione ad un server ed una seconda parte che deve inviare pacchetti, con assume possiamo controllare se effettivamente la condizione di connessione è rispettata, se ciò non avviene il test viene saltato.
- **AssertAll():**permette di inglobare in un unico blocco più funzioni assert; in questo modo però verrà testata come un'unica entità.
- **@Nested**: permette di organizzare i test in classi di appartenenza; ad esempio se ho più test che verificano ad esempio una funzione somma(a,b), dove un test è con a negativo ed uno con a positivo, li posso raggruppare. Ad esecuzione completata verranno eseguiti come se fossero singoli test ma mostrati all'interno di sezioni.
- **@Tag**: permette di assegnare un'etichetta con una stringa. Un suo utilizzo potrebbe essere quello di eseguire ad esempio test di un determinato Tag escludendo i restanti.

3.5 Mockito

Siamo finalmente riusciti a testare ed assolvere in modo completo la richiesta del problema, ma in software più complessi vi è sempre la presenza di dipendenze come chiamate a funzioni appartenente ad altre classi, ad esempio la chiamata a costruttori ecc. Ciò non ci deve stupire in quanto fa parte dell'accoppiamento tra le classi del nostro software. Analizziamo un esempio, immaginiamo di voler creare una app To-Do, cioè una app che mostra e permetta di gestire una lista di To-Do (lista di cose da fare). Una delle funzionalità cardine sarà appunto il mostrare all'utente i propri compiti attraverso l'inserimento del proprio username. Potremo progettare questa funzionalità attraverso due classi, la prima si connette ad un server esterno per ottenere la lista completa dei dati in base al nome utente e una seconda che filtra tali dati secondo dei parametri.

Schema:



Poiché immaginiamo che il metodo *TodoService()* venga implementato da altri sviluppatori, lo progettiamo come interfaccia.

- **retrieveTodos(String user)**: effettua una connessione di rete verso un server e restituisce una lista di Stringhe attraverso la ricerca con nome utente
- **retrieveTodosRelatedToSpring():** preleva i dati da *retrieveTodos()* e li filtra secondo un criterio.

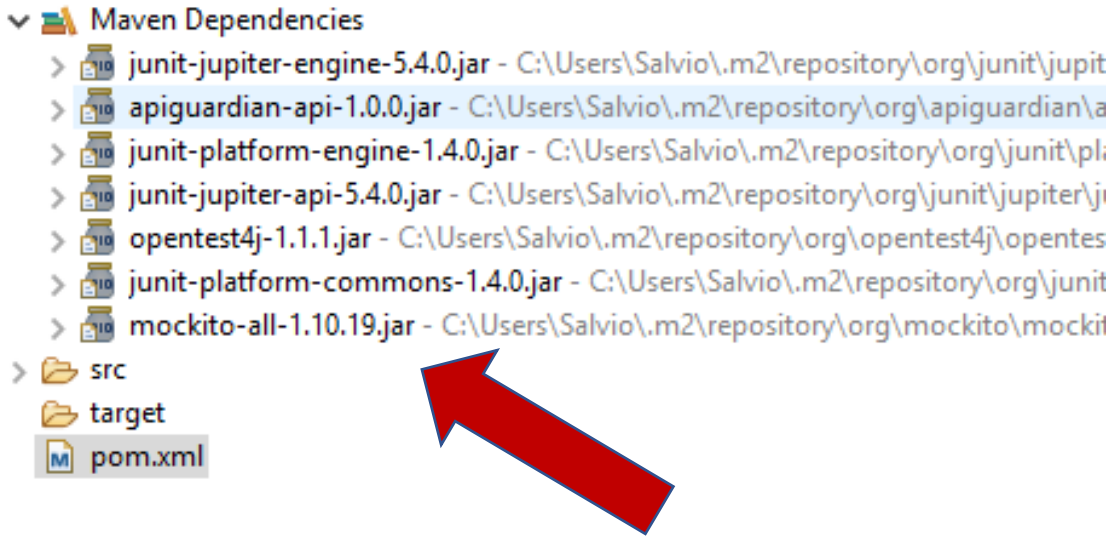
La domanda che sorge spontanea è: Come posso testare una funzione che dipende da altri metodi non ancora implementati?

La risposta viene data dagli Stub e dai Mocks; entrambi permettono di creare classi fittizie che sopperiscono alla mancanza di quelle reali e di cui avremo bisogno. Per implementare lo Stub bisogna creare una nuova classe con metodi che restituiscono direttamente i valori desiderati senza realizzare la logica. Questa "staticità" è stata superata dai mocks, che oltre a non richiedere alcuna scrittura di classi aggiuntive, permettono dinamicamente di assegnare i valori da restituire.

Per poter utilizzare i mocks useremo il framework Mockito, che va importato come dipendenza nel progetto Maven con l'aggiunta del seguente codice nel file pom.xml

```
27        <dependency>
28            <groupId>org.mockito</groupId>
29            <artifactId>mockito-all</artifactId>
30            <version>1.10.19</version>
31            <scope>test</scope>
32        </dependency>
```

Aggiorniamo il progetto e vedremo comparire il seguente file mockito-all-1.10.19.jar nelle dipendenze:



Ora siamo pronti per scrivere il nostro caso di test:

```java
package mockito.test;

import static org.junit.jupiter.api.Assertions.*;

class TodoAppMockitoTest {

    @Test
    public void usingMockito() {
        TodoService todoService = mock(TodoService.class);
        List<String> allTodos = Arrays.asList("Learn Spring MVC", "Learn Spring","Learn to Dance");
        when(todoService.retrieveTodos("Salvio")).thenReturn(allTodos);
        TodoBusinessImpl todoBusinessImpl = new TodoBusinessImpl(todoService);
        List<String> todos = todoBusinessImpl
                .retrieveTodosRelatedToSpring("Salvio");
        assertEquals(2, todos.size());
    }

}
```

Per creare un oggetto mockito utilizziamo il metodo *mock()* che riceve come parametro la classe che *TodoService.class*, cioè la classe da "mockare". Fatto ciò bisogna dire che valore restituire e sulla chiamata di quale metodo; utilizziamo la funzione ***when***(funzione del mock).***thenReturn***(valore che vogliamo ritorni), come nella foto in alto.

Di seguito riportiamo gli screen delle classi create:

```java
package todoapp.api;

import java.util.List;

public interface TodoService {
    public List<String> retrieveTodos(String user);
}
```

TodoBusinessImpl.java | testUnit/pom.xml | TodoAppMockitoTest.java | TodoService.java

```java
package todoapp.business;

import java.util.ArrayList;

public class TodoBusinessImpl {

    private TodoService todoService;

    public TodoBusinessImpl(TodoService todoService) {
        this.todoService = todoService;
    }

    public List<String> retrieveTodosRelatedToSpring(String user) {
        List<String> filteredTodos = new ArrayList<String>();
        List<String> allTodos = todoService.retrieveTodos(user);
        for (String todo : allTodos) {
            if (todo.contains("Spring")) {
                filteredTodos.add(todo);
            }
        }
        return filteredTodos;
    }

}
```

I mocks in generale assolvono anche altri problemi, come la variabilità del risultato di alcune funzioni partendo dalle stesse precondizioni. Ad esempio se ho una funzione che effettua una richiesta http, uno dei possibili risultati potrebbero essere errori dovuti allo stato della connessione. Quindi se presumiamo che la funzione non riesca a raggiungere un determinato indirizzo ip per mancanza di connessione il nostro test fallirà pur avendo scritto correttamente il nostro codice. In definitiva i mocks possono essere visti come componenti che ci permettono di astrarre tutte le dipendenze e di poter iniettare i valori desiderati nel processo.

3.6 Considerazioni finali

Prima della stesura di questa tesi non ero a conoscenza del TDD, e questo significa che è stata la mia prima esperienza nella scrittura di codice e test organizzati in tale modo. Anche

io, come molti sviluppatori alle prime armi, effettuavo il test del codice all'interno del Main file, sprecando continuamente tempo durante lo sviluppo. Non è stato semplice digerire questo nuovo modo di testare, essenzialmente poiché la forma mentis ci porta naturalmente prima a creare codice e poi il suo test, ma armato di pazienza e tanti tentativi, ci si abitua velocemente (ho impiegato circa tre giorni nel riuscire a pensare e scrivere test prima del codice). I vantaggi ricavati sono tangibili in quanto il tempo dedicato al debugging è stato pari a zero, anche perché il partire da test semplicissimi permette di costruire il tutto un pezzo alla volta e se anche ci fossero errori verrebbero allo scoperto grazie all'alternanza test-code-fix. Ciò concede anche una certa soddisfazione nel vedere che il codice scritto funziona, e i test risultano tutti verdi.

Di contro posso dire che si spende molto tempo nel test di una funzionalità, ma forse ciò è anche dato dalla non completa conoscenza degli strumenti che mettono a disposizione JUnit e Mockito e dalla non praticità nell'usarli.

Durante i vari test mi sono chiesto se ci siano casi in cui sia obbligatorio usare il TDD e casi in cui se ne possa fare a meno. Nelle applicazioni critiche è svantaggioso poiché il TDD preclude la fase di progettazione, in effetti per molti l'ultima D sta ad indicare *Design* e non *Development*. Da un lato ci obbliga a scrivere codice tascabile con unit test ma dall'altro ciò potrebbe portare a creare un design più complicato di quello che si potrebbe progettare con le tecniche classiche. In favore del TDD vi è la progettazione di API; scrivere prima il codice client (il nostro test in questo caso) costringe a pensare come vorremo usarlo e quindi ci guida alla realizzazione di un API sensata. Questo è anche vero in conseguenza al fatto che la metodologia richiede agli sviluppatori di pensare al software in termini di piccole unità che possono essere testate ed integrate in modo indipendente abbassando l'accoppiamento.

Potremmo evitarlo in ambienti dove è possibile subito runnare il codice e testarlo in live, come avviene in XCode 10, dove possiamo eseguire la nostra applicazione all'interno di simulatori o ancora meglio con Swift UI vi è la possibilità di vedere il comportamento del codice subito dopo averlo scritto senza doverlo compilare.

Un suggerimento che ritorna utile è quello di lasciare l'ultimo test in rosso prima di andare a letto, questo ci permetterà il giorno dopo di ricordare dove avevamo sospeso il lavoro e quindi di riprendere in modo più spedito.