UNIVERSITÀ DEGLI STUDI DI SALERNO

Dipartimento di Informatica

Corso di Laurea Triennale in Informatica

TESI DI LAUREA

Progettazione ed implementazione di un'applicazione web in contesti aziendali

Un caso di studio di utilizzo di Kanban e Test-Driven Development

RELATORE

Prof. Dario Di Nucci

Università degli Studi di Salerno

CANDIDATO

Gabriele Di Stefano

Matricola: 0512115667

Anno Accademico 2024-2025

Questa tesi è stata realizzata nel

"The business changes. The technology changes. The team changes. The team members change. The problem isn't change, per se, because change is going to happen; the problem, rather, is the inability to cope with change when it comes."

Kent Beck

Abstract

Le metodologie agili sono dei metodi di sviluppo software atti a favorire un lavoro in team più agevole e flessibile. Tra queste, Kanban è una metodologia che prevede la suddivisione del lavoro in attività da collocare in una bacheca, organizzandole in colonne che rappresentano le diverse fasi del processo di sviluppo. Il Test-Driven Development è una pratica che prevede la precedente scrittura di test per ogni funzionalità da implementare. L'obiettivo di questo lavoro di tesi è applicare le metodologie agili al fine di realizzare un'applicazione web, col fine ultimo di gestire le risorse aziendali assegnate a ciascun progetto, in un piccolo team di sviluppo. L'applicazione realizzata si basa su Laravel, un framework MVC scritto in PHP dedicato allo sviluppo di applicazioni web, e Livewire, un'estensione di Laravel per lo sviluppo di interfacce utente dinamiche - attraverso la creazione di componenti reattivi - che utilizza Alpine JS, un framework JavaScript leggero e minimale.

Indice

Elenco delle figure

CAPITOLO 1

Introduzione

1.1 Contesto

Le metodologie Agile rappresentano un cambio di paradigma notevole rispetto alle metodologie tradizionali: da un approccio rigido e sequenziale a uno più flessibile, iterativo e centrato sulle persone. Mentre i metodi tradizionali — come il modello a cascata — prevedono fasi rigidamente separate e un piano dettagliato fin dall'inizio, queste nuove metodologie si basano su princìpi di adattabilità, collaborazione continua e consegna incrementale di valore.

La storia delle metodologie Agile rappresenta un complesso percorso evolutivo, iniziando con i primi tentativi di ottimizzazione dei processi di sviluppo software e culminando in una filosofia organizzativa completa: partendo da Scrum, un framework precursore di Agile concepito da Sutherland e Schwaber, e dall'Extreme Programming, una metodologia sviluppata da Kent Beck negli anni Novanta, il culmine viene raggiunto con la pubblicazione del *Manifesto Agile* nel 2001. Successivamente, nel 2010, Anderson contribuì ulteriormente all'evoluzione delle metodologie Agile con la formalizzazione di Kanban, un metodo che pone l'enfasi sulla visualizzazione del flusso di lavoro, sulla limitazione del lavoro in corso e sull'ottimizzazione continua dei processi.

1.2 Motivazioni e obiettivi

La tesi si propone di **esaminare e valutare le metodologie Agile utilizzate**, prendendo come caso di studio lo sviluppo di un'applicazione realizzata durante un tirocinio. Tale progetto è stato avviato in risposta a una richiesta aziendale per ottimizzare la gestione del personale, il cui obiettivo è amministrare efficacemente le risorse aziendali, assicurando che vengano assegnate alle attività con il ruolo più adeguato. Le metodologie agile prese come caso di studio sono, rispettivamente, **Kanban**, un approccio che guida il lavoro individuale tramite la suddivisione in task più facilmente gestibili, e **Test-Driven Development (TDD)**, una metodologia di sviluppo software derivante dall'Extreme Programming che si concentra sulla realizzazione delle funzionalità del sistema attraverso la stesura preliminare dei casi di test.

1.3 Risultati

La scelta di adottare metodologie Agile si è rivelata fondamentale per **abbattere i tempi di progettazione e sviluppo richiesti**, seppur **preservando i controlli di qualità previsti** da un processo rigoroso e formale. A partire dai requisiti estratti durante la fase di progettazione, utilizzando Kanban, è stato successivamente possibile ricavare circa trenta User Stories, utilizzate direttamente per la fase implementativa. Infine, con il TDD, è stato possibile realizzare le funzionalità dell'applicativo partendo prima dalla stesura dei casi di test, raffinando successivamente attraverso l'utilizzo delle metriche di *code coverage* e *mutation score*.

1.4 Struttura della tesi

Oltre al presente capitolo, la tesi si compone di altri tre capitoli e di un'appendice. Il **Capitolo 2** fornisce una breve panoramica sulle metodologie tradizionali e Agile, mostrando vantaggi e criticità di ognuna. Il **Capitolo 3** delinea in maniera dettagliata il modus operandi con cui sono state implementate le metodologie menzionate. Il **Capitolo 4** racchiude le considerazioni finali, una valutazione dei risultati ottenuti e una sintesi degli aspetti salienti dell'esperienza descritta. Infine, l'**Appendice A** presenta una panoramica completa delle tecnologie impiegate nella realizzazione dell'applicativo.

CAPITOLO 2

Background

2.1 Ciclo di vita del software

In Ingegneria del Software, l'obiettivo principale è quello di affrontare efficientemente le varie fasi di realizzazione, gestione e manutenzione di un software, così da minimizzare il numero di problemi riscontrabili. Tutto ciò è possibile grazie a una buona progettazione preliminare, fatta attraverso l'utilizzo di un modello del ciclo di vita del software.

Tale approccio ingegneristico, opposto al metodo *code-and-fix* utilizzato sin dagli albori dell'informatica, venne teorizzato — principalmente da Benington e Hosier — tra la fine degli anni Cinquanta e l'inizio degli anni Settanta [5], e culminò con la formalizzazione del modello *Waterfall* nel 1970 [1]. Esso, infatti, provò a definire un processo lineare, suddiviso in vari passaggi: *Requirements Elicitation*, dove vengono raccolte le specifiche del sistema da realizzare; *Requirements Analysis*, durante il quale vengono analizzati i requisiti raccolti in precedenza; *Design*, che prevede la progettazione del sistema da realizzare; *Implementation*, che consiste nella codifica vera e propria; *Testing*, in cui si collauda l'implementazione del sistema; *Maintenance*, che consiste nella correzione dei problemi che si presentano dopo il rilascio.

2.1.1 Il modello a cascata

Come precedentemente specificato, il modello *Waterfall* presenta una rigida suddivisione in fasi del ciclo di vita del software, ognuna delle quali finalizzata alla produzione di un **artefatto**, o *deliverable*, utilizzato come input della fase consecutiva. Di seguito, si analizzeranno più in dettaglio le varie fasi del processo.

Elicitazione dei requisiti In questa fase si specificano i requisiti indispensabili per definire il software da sviluppare, ottenuti attraverso la ricerca, l'identificazione e la successiva raccolta fornita dagli stakeholder del progetto, ovvero da tutti i partecipanti che hanno un interesse in esso (inclusi gli sviluppatori). È qui che vengono definiti gli *scenari*, gli *attori* e gli *Use Case*.

Analisi dei requisiti A partire dai requisiti raccolti durante la fase precedente, si può dare una panoramica iniziale sul sistema che dovrà essere realizzato, col fine ultimo di eliminare ambiguità e chiarire eventuali punti non palesatisi durante la fase di elicitazione. Qui vengono modellati un *Class Diagram* ad alto livello, i *Sequence Diagram* e gli *State Chart Diagram*, finalizzati all'identificazione dei *Requisiti Funzionali* e *Non Funzionali*. Stando al modello Waterfall, il *deliverable* di questa fase è il **Requirements Analysis Document**.

Progettazione Questa fase è composta da due sotto-fasi, rispettivamente da **System Design** e **Object Design**: la prima si occupa di identificare i *System Design Goals* e *Trade-Offs* e della *decomposizione in sottosistemi*, al fine di stabilire i *flussi di accesso e di controllo* e di fissare i *pattern architetturali* che meglio si prestano alla realizzazione del sistema; la seconda, invece, è finalizzata alla selezione dei *Componenti Off-The-Shelf* e dei *Design Pattern*, alla suddivisione delle classi in *Package*, alla *specifica delle interfacce* e alla realizzazione del *Class Diagram* finale. Nel modello Waterfall, i due *deliverable* sono rispettivamente il **System Design Document** e l'**Object Design Document**.

Implementazione Questa fase consiste nella scrittura del codice relativo al software da realizzare, utilizzando le specifiche stabilite durante le fasi precedenti. Il *deliverable* risultante da questa fase è lo stesso **codice sorgente** che viene scritto.

Collaudo Una volta completata la fase di implementazione, è necessario assicurarsi che il software si comporti secondo le attese: dopo la stesura di un *Test Plan*, che definisce le fasi del processo di collaudo, e di una *Test Case Specification*, che descrive in dettaglio i casi di test, si passerà all'implementazione e all'esecuzione dei test previsti. Al termine dell'esecuzione, in caso di *fault* riscontrati, si passerà alla stesura di un **Test Incident Report**, dove saranno raccolti tutti gli errori e le relative procedure per eliminarli. In ogni caso, vi sarà poi un ultimo *deliverable*, il **Test Summary Report**, che offre un sommario dei risultati ottenuti durante questa fase. Ad ogni modo, al termine della fase, il sistema risultante sarà pronto al rilascio.

Manutenzione Questa fase segue il rilascio e la messa in produzione del sistema, e comprende tutte le operazioni volte al mantenimento del sistema, che siano di evoluzione, miglioramento o correzione di errori. Ogni modifica comporta una nuova esecuzione del processo di sviluppo a cascata, ripercorrendo dunque tutte le fasi viste in precedenza.

La figura 2.1 mostra la composizione del modello a cascata.

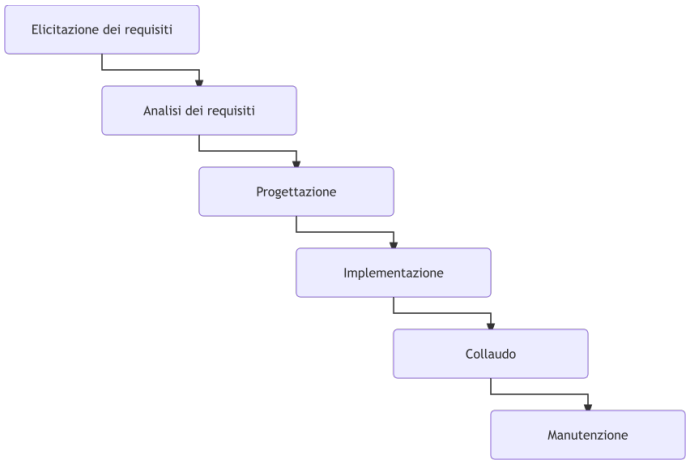

Figura 2.1: Fasi della metodologia Waterfall.

2.1.2 Le criticità

Questo rigido approccio metodico, scandito in fasi ben definite, che ben si applica a diverse discipline (basti pensare alla catena di montaggio di Henry Ford), è però problematico nell'ambito dell'Ingegneria del Software. Lo stesso Royce ha espresso perplessità sul modello, definendolo rischioso e propenso al fallimento [1].

Il problema fondamentale di questo approccio è lo scostamento che si crea tra i requisiti definiti all'inizio e ciò che vuole davvero il cliente: considerando anche la mutevolezza nel tempo delle esigenze del cliente, è difficile delineare tutte le richieste durante l'analisi dei requisiti, rendendo quindi il modello decisamente poco adatto ai cambiamenti. Questa situazione porta alla necessità di ricominciare l'intero processo dall'inizio, poiché il modello Waterfall è lineare e non consente il ritorno a fasi precedenti, e all'inevitabile ritardo nel dover sviluppare le modifiche richieste.

Un'ulteriore questione importante da considerare è che il cliente partecipa solo all'inizio e alla conclusione del processo, risultando così escluso nelle fasi di progettazione e sviluppo del software, dove invece il suo feedback sarebbe alquanto prezioso per assicurarsi che tutti gli altri stakeholder condividano la sua stessa visione sul progetto posto in essere.

Un'ulteriore sfida da non ignorare riguarda la ripartizione gerarchica delle responsabilità e la comunicazione a senso unico, che comporta una perdita di informazioni.

Inoltre, i tempi necessari per il rilascio di un'applicazione si estendono considerevolmente a causa della lunga durata del processo nella sua interezza: per un progetto di medie dimensioni, si può parlare anche di uno o più anni.

Infine, un'ultima problematica, irrisolvibile in quanto intrinseca al processo, è l'enfasi eccessiva posta sulla documentazione e sulla progettazione UML a discapito della fase di implementazione. Si presuppone, infatti, che una solida progettazione vincoli la codifica alla documentazione precedentemente redatta, trascurando tuttavia eventuali problemi o necessità che potrebbero emergere unicamente durante le fasi di implementazione e collaudo.

2.1.3 Modelli più avanzati basati sul modello a cascata

Queste problematiche erano note sin dagli albori del modello a cascata, tanto che Royce propose alcune migliorie al processo (figura 2.2), quali l'integrazione di fasi **V&V** (**Verification** & **Validation**), che si assicurano che il sistema sia costruito rispettando le specifiche richieste e che questo risponda adeguatamente alle esigenze del cliente. Grazie all'aggiunta di questi ulteriori passaggi, e della possibilità di retroazione, il modello diventa iterativo, in quanto è ora possibile tornare anche alle fasi precedenti in caso di necessità. Lo stesso Royce ritenne necessario affrontare più volte il processo, in modo incrementale, in modo da chiarire man mano alcuni aspetti che, durante le iterazioni precedenti, apparivano ancora poco chiari. Tuttavia, seppur vi sia un primo importante accenno sulla necessità di rendere il ciclo di vita iterativo, tale modello presenta ancora le stesse criticità trattate in precedenza.

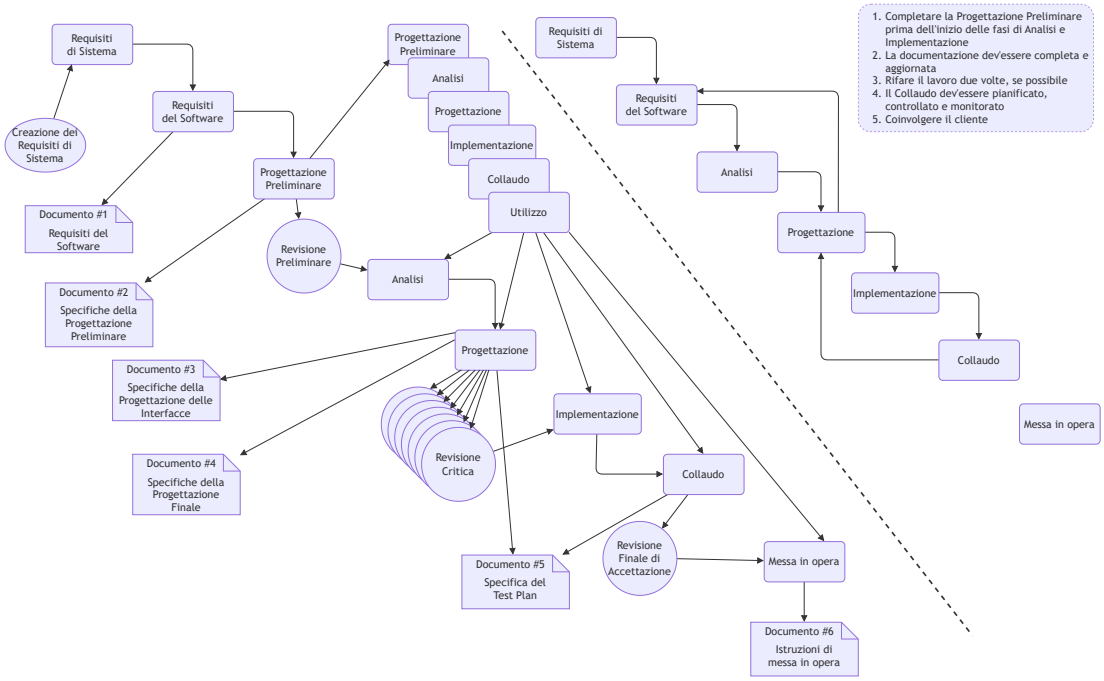

Figura 2.2: Modello finale proposto da Royce. Figura adattata da [1].

Modello a V Il modello a V è stato originariamente sviluppato come modello di sviluppo di sistemi a opera di una società statale tedesca, l'*Industrieanlagen-Betriebsgesellschaft* (*IABG*), su commissione del Ministero Federale della Difesa. È stato successivamente ripreso, più in generale, come modello di sviluppo del software. [6] Il processo è molto simile al modello a cascata, difatti ne riprende le fasi, con una differenza fondamentale: ad ogni fase di collaudo (unità, integrazione, sistema e accettazione) viene fatta corrispondere una relativa fase di progettazione (design o analisi dei requisiti), che sarà poi oggetto di processi **V&V** in merito al *deliverable* generato. In tal modo, in caso di fallimento di uno dei processi durante una specifica fase di collaudo, il processo può riprendere dalla precedente fase di progettazione associata, risultando dunque iterativo per sua stessa natura. La Figura 2.3 mostra una rappresentazione del modello a V.

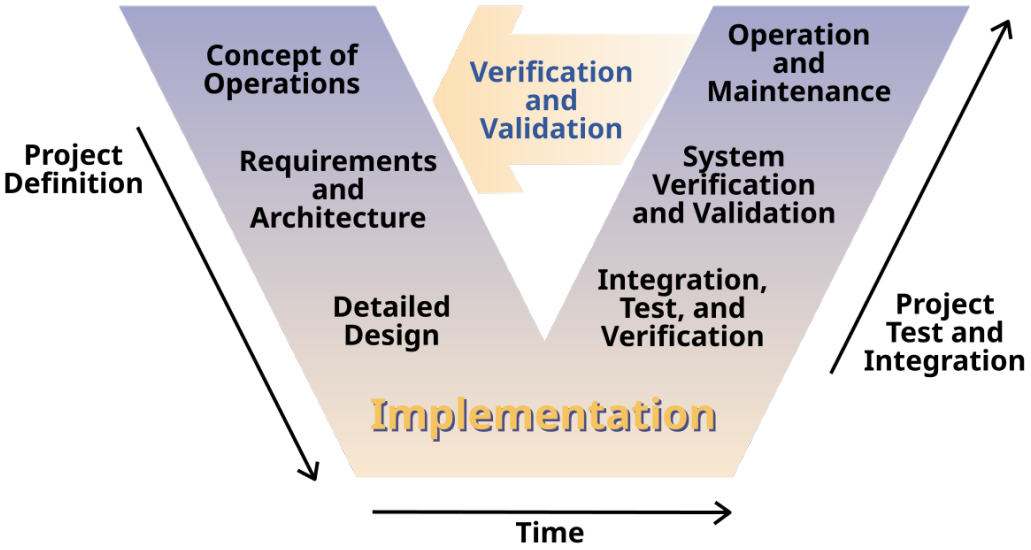

Figura 2.3: Fasi del modello a V: a ogni fase di collaudo è associata una di progettazione. [2]

Modello a spirale Descritto per la prima volta da Boehm nel 1986 [7], esso funge da archetipo per le successive metodologie che si sono susseguite, per via della presenza di un processo iterativo e incrementale del software. Viene considerato un *meta-modello*, più che un modello a sé stante, di tipo *risk-driven*, poiché il suo scopo principale è quello di integrare una qualsiasi altra metodologia di sviluppo in modo iterativo, attraverso una preventiva fase di analisi dei rischi. Il processo si compone delle stesse fasi presenti nel modello a cascata — visibile nella figura 2.4 — ma ripetute più volte, e con durate iniziali più ristrette, ma che accrescono nel tempo: in questo modo, i processi lavorativi si occupano di costruire in modo incrementale il software, partendo con una prima fase di prototipazione, in cui si costruisce una versione preliminare del progetto, per poi raffinarla durante le iterazioni successive. In realtà, questo modello prende spunto proprio dai dubbi sollevati da Royce riguardo ai rischi derivanti dal *Waterfall*, cercando di adottare un approccio allo sviluppo di tipo evolutivo, più che di pianificazione preventiva.

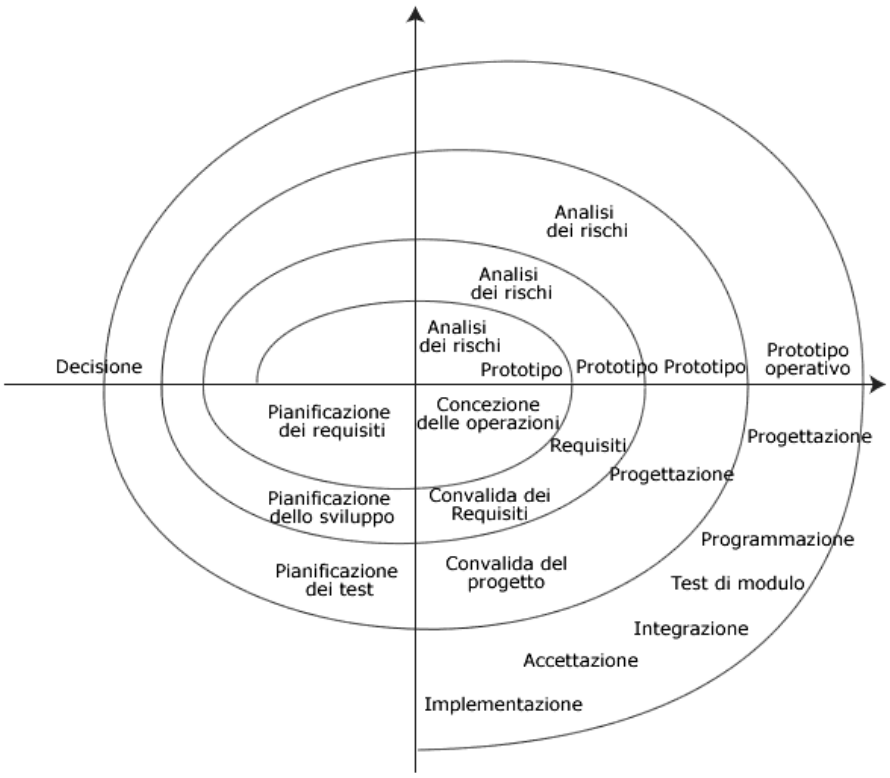

Figura 2.4: Esempio di modello a spirale con 4 iterazioni: ogni iterazione ambisce a raffinare il prodotto della precedente in modo incrementale. [3]

2.2 Metodologie di sviluppo Agile

Essendo sorta l'esigenza di nuovi modelli di sviluppo del software che rispondessero meglio ai cambiamenti e che minimizzassero i costi e i tempi richiesti, erano dunque necessarie delle metodologie che adottassero un paradigma ben diverso da un semplice approccio lineare e rigidamente suddiviso in fasi: è così che nacquero le metodologie Agile, che si prefiggono gli obiettivi di minimizzare il rischio e consegnare artefatti frequentemente e in tempi brevi. Pur non essendo direttamente derivanti dal Modello a Spirale, le metodologie Agile si basano sugli stessi concetti fondamentali di sviluppo incrementale e iterativo.

I precursori di tale dottrina sono identificati rispettivamente nell'**Extreme Programming**, introdotto da *Kent Beck*, e nello **Scrum**, sviluppato da *Ken Schwaber* e *Jeff Sutherland*. I quattro capisaldi di queste metodologie sono espressi nel Manifesto Agile, pubblicato nel 2001 [8], e danno maggiore importanza a:

- **gli individui e le interazioni**, piuttosto che i processi e gli strumenti;
- **il software funzionante**, rispetto a una documentazione esaustiva;
- **la collaborazione con il cliente**, più che la negoziazione dei contratti;
- **la reattività al cambiamento**, anziché l'adesione a un piano prestabilito.

Nonostante sia impossibile eliminare del tutto la pianificazione temporale, queste metodologie offrono un approccio significativamente più flessibile e adattabile. Ciò consente una risposta più efficace alle reali esigenze del cliente, rispetto alla necessità di sviluppare in anticipo scenari complessi che potrebbero risultare superflui o inadeguati nelle fasi più avanzate dello sviluppo. Ci sono vari modelli che provengono dalla filosofia Agile; il procedimento consiste nel dividere il progetto in diverse *iterazioni* — ovvero dei sotto-progetti, ciascuno con scadenze temporali piuttosto brevi. Il sistema sarà il risultato dell'unione degli artefatti prodotti in ogni iterazione.

2.3 Scrum

Presentato nel 1995 [9], Scrum è un primo tentativo di approccio Agile al ciclo di vita del software. Più che una metodologia vera e propria, Scrum è fondamentalmente un *framework* Agile che offre diversi strumenti, da impiegare o meno a seconda delle necessità progettuali. L'utilizzo di questa metodologia comporta la suddivisione del lavoro in obiettivi che vanno completati in diverse iterazioni, con durate tra le 2 e le 4 settimane, chiamate **Sprint**. Il lavoro viene organizzato in modo orizzontale, non gerarchico, attraverso un team composto da pari che prende il nome di **Scrum Team**. [10, 11, 12] Il processo (figura 2.5) si compone di diverse fasi:

- si inizia con la raccolta e l'analisi dei requisiti, da cui vengono ricavate delle *User Stories* (o, più raramente, dei *Task*) che vengono raccolte nel **Product Backlog**;
- si analizzano e si rivedono i vari elementi del Product Backlog, dopodiché si raccolgono quelli da realizzare nel prossimo Sprint. Tale processo prende il nome di **Sprint Planning**, e l'artefatto che si va a costituire è lo **Sprint Backlog**;
- si inizia lo Sprint, durante cui si tengono delle brevi riunioni giornaliere, i **Daily Scrum Meeting**, dove si discute dei progressi, dei problemi e di come si ha intenzione di proseguire nello sviluppo. Si aggiornano le metriche di progresso nello sviluppo, quali il **Burndown Chart**, il **Release Burnup Chart** e la **Velocity**.
- Al termine dello Sprint viene rilasciato un artefatto funzionale, l'**Increment**, che va ad aggiungersi ai risultati dei precedenti Sprint. Successivamente, vengono svolte rispettivamente due riunioni, la **Sprint Review**, finalizzata a ispezionare il risultato dello Sprint per vedere cos'è andato bene e cosa no, e la **Sprint Retrospective**, utile a migliorare la pianificazione degli Sprint futuri sulla base dei feedback raccolti durante la review.

Tale processo è continuo e si arresta soltanto al raggiungimento degli obiettivi prefissati nella **Definition of Done**, ovvero un documento che attesta i criteri necessari affinché il progetto possa ritenersi concluso con successo.

Il processo, inoltre, è governato da diverse figure:

- gli **Stakeholder**, come in ogni altra metodologia, sono tutti gli individui che hanno interesse nella riuscita del progetto;
- il **Product Owner** è il responsabile dell'intero progetto, funge da tramite tra gli Stakeholder e i restanti membri dello Scrum Team;
- lo **Scrum Master** è il facilitatore dello Scrum Team, ha il compito di insegnare le pratiche Scrum, gestire lo Sprint Backlog e facilitare la comunicazione tra sviluppatori e Product Owner. Non è una figura equiparabile ad un Project Manager, in quanto non ha le stesse responsabilità, poiché in Scrum il Team viene incoraggiato ad auto-gestirsi;
- gli **Sviluppatori** sono tutti gli altri individui coinvolti direttamente nello sviluppo del progetto.

Scrum si basa molto su tutte le pratiche organizzative e comunicative brevi e periodiche, che infatti prendono il nome di *riti* o *cerimonie*.

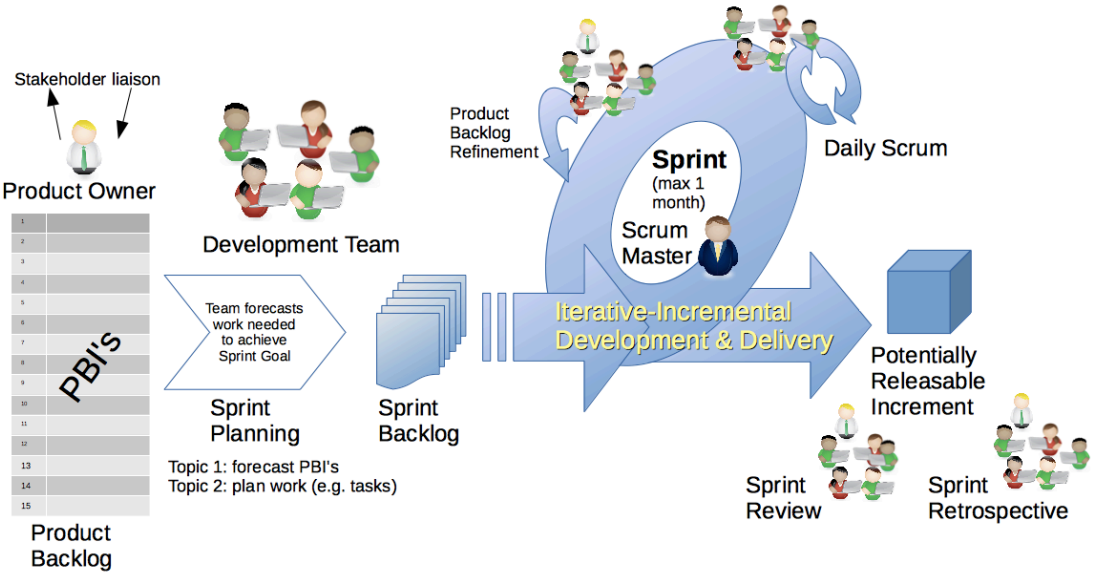

Figura 2.5: Il processo Scrum. [4]

2.3.1 I vantaggi

È interessante sottolineare come, durante ogni fase del processo, siano coinvolti tutti i componenti del team. Difatti, Scrum adotta — così come l'Extreme Programming — la filosofia della responsabilità condivisa: ogni membro è direttamente responsabile della riuscita del progetto. Ciò assume una valenza non indifferente, in quanto incoraggia ciascuna figura a partecipare attivamente alle attività del progetto, senza così delegare le proprie responsabilità a qualcun altro. È chiaro che alcune figure avranno obiettivi più specifici — si pensi, ad esempio, al Product Owner, il cui obiettivo è quello di massimizzare il valore del prodotto — ma non bisogna dimenticare che sono tutte intrinsecamente legate al successo del progetto.

Il lavoro di squadra assume, dunque, un'importanza fondamentale: come gli ingranaggi in un complesso meccanismo, il corretto funzionamento dell'intero sistema è garantito dal contributo di ogni piccolo componente, a prescindere dalle sue dimensioni effettive.

La comunicazione, poi, rappresenta il fulcro che rende possibile tutto ciò: ogni membro è fortemente incoraggiato — sia obbligatoriamente attraverso le *cerimonie*, ma anche intrinsecamente come motore di crescita personale e professionale — a comunicare con il resto del team, che sia per problematiche riscontrate, o anche solo per sottolineare i progressi ottenuti. Ciò funge da collante per l'intero Scrum Team, facilitando la risoluzione di conflitti e mantenendo un clima generale più trasparente, sereno e distensivo. [10]

In ultima analisi, è facile notare come Scrum si prefigga l'obiettivo di ridurre al minimo il Social Debt, cercando di eliminare, per quanto possibile, eventuali fallimenti del progetto derivanti da attriti all'interno del gruppo di lavoro.

2.3.2 Le criticità

Bisogna sottolineare, tuttavia, che **Scrum non rappresenta una panacea**: la sua corretta applicazione è possibile soltanto in presenza di figure con esperienza, ed è impensabile ritenere che sia applicabile da individui senza una conoscenza approfondita di tale metodologia.

Martin Fowler e Ron Jeffries, due firmatari del Manifesto Agile, hanno inoltre criticato Scrum, colpevole di aver posto troppa enfasi sulle *cerimonie*, in netto contrasto coi princìpi dello stesso manifesto. [13, 14]

La dottrina di Scrum è estremamente concisa, ma non bisogna dimenticarsi che non si tratta di una mera metodologia, ma di un *framework* completo: è compito del team scegliere esclusivamente gli strumenti che si rivelano utili al conseguimento dei propri obiettivi aziendali; altrimenti, si ricadrebbe nell'eccessiva formalizzazione del processo, ritornando a essere pericolosamente simile ai tradizionali modelli di sviluppo, con tutti i rischi che ciò comporta.

2.4 Extreme Programming

L'Extreme Programming (o *XP*), precursore delle metodologie Agile ideato nel 1999, è una metodologia che consiste nel portare all'*estremo* le buone pratiche di sviluppo software. Kent Beck ha sviluppato e raffinato tale metodologia nel periodo che va dal 1996 al 1999, durante il quale era impegnato nello sviluppo del progetto *Chrysler Comprehensive Compensation* in qualità di project leader. [15]

L'XP si concentra sull'analisi delle quattro variabili fondamentali di ciascun progetto software: costo, tempo, qualità e portata. Esso attribuisce particolare rilevanza alla portata, in quanto spesso rappresenta l'unico parametro su cui è possibile esercitare un controllo significativo. Questo approccio condivide con Scrum numerosi princìpi, tra cui l'aspirazione alla semplicità, la comunicazione aperta e trasparente, l'importanza delle modifiche incrementali e l'accettazione della natura mutevole delle esigenze. Da tali princìpi, l'XP stabilisce le seguenti pratiche:

- **Planning Game**, ovvero un metodo per pianificare il lavoro a breve e lungo termine, in accordo tra manager e sviluppatori;
- **Brevi cicli di rilascio**, cioè rilasciare in modo incrementale le modifiche al progetto, il più velocemente possibile;
- **Metafora**, in quanto è fondamentale saper descrivere il sistema in parole più semplici. In questo modo, si dimostra di aver meglio afferrato l'ambito del progetto;
- **Semplicità di progetto**, poiché altrimenti un sistema complesso diventa difficile da manutenere nel tempo;
- **Testing**, in modo da collaudare preventivamente sia le funzionalità da implementare (*Unit Testing*) che le specifiche imposte dal cliente (*Functional Testing*), in qualsiasi momento;
- **Refactoring**, poiché semplificare e rendere più leggibile il codice riduce la complessità del sistema — a condizione però che ciò avvenga solo quando necessario, altrimenti rallenterebbe inutilmente lo sviluppo;

- **Programmazione a coppie**, ovvero la scrittura del codice dev'essere effettuata per coppie, non per singoli individui;
- **Proprietà collettiva**, così da affidare una responsabilità condivisa del codice a tutti gli sviluppatori;
- **Integrazione continua**, il che consente di identificare e di eliminare possibili problemi sul nascere, poiché lo si noterebbe dal fallimento dei test dopo la modifica di una funzionalità;
- **Settimana di quaranta ore**, per mantenere il team di sviluppo riposato e meno propenso ai dissidi;
- **Cliente sul posto**, in quanto ciò permette di avere un riscontro immediato sulle funzionalità in corso d'opera, e di modificarle senza ritardi in caso di diverse esigenze;
- **Standard di codifica**, affinché tutti gli sviluppatori possano interpretare diverse porzioni del codice in modo agevole.

Queste dodici pratiche sintetizzano la dottrina dei processi Agile: la buona riuscita di un progetto dipende principalmente dalla semplicità nella manutenzione e nella sua continua evoluzione, oltre che dalla semplificazione dei processi aziendali, valorizzando la comunicazione tra i vari componenti del progetto.

Oltre agli sviluppatori e ai manager, l'XP introduce due ulteriori ruoli — ricopribili entrambi anche da una sola figura — ovvero il *coach*, responsabile della comunicazione tra management e team di sviluppo, e il *tracker*, responsabile della raccolta e della misurazione delle metriche del progetto.

2.4.1 I vantaggi

Questa metodologia presenta il notevole vantaggio di essere prontamente adottabile anche da piccoli team di sviluppo agli esordi, assicurando risultati rapidi grazie alle buone pratiche che promuove. La documentazione necessaria è davvero esigua, poiché la maggior parte dei requisiti viene estratta dal cliente situato in loco, a cui vengono poste domande e dubbi sulla progettazione. Anche qui ritroviamo l'importanza di lavorare in modo cooperativo, affidando una responsabilità condivisa a tutto il team di sviluppo e incoraggiando una comunicazione aperta e trasparente. Da menzionare, poi, è l'enfasi che viene posta sull'analisi delle metriche di progetto, che fungono da indicatore dello stato di salute dell'intero processo. Infine, lo sviluppo guidato dai test e il refactoring costante consentono di mantenere un'elevata qualità del codice scritto, a patto che ciò venga fatto con un criterio adeguato (ad esempio, attraverso la scelta di metriche significative).

2.4.2 Le criticità

Seppur più veloce, l'XP non può considerarsi una metodologia completa: mancano le fasi di elicitazione e analisi dei requisiti, la cui lacuna viene parzialmente colmata dalla presenza fissa di un cliente. Realisticamente parlando, però, è chiaro che tale individuo avrà notevoli difficoltà a ricordare *tutte* le specifiche progettuali, trovandosi sommerso di richieste. L'XP cerca di arginare questa problematica attraverso la raccolta di User Stories, che però si rivelano insufficienti e imprecise rispetto a dei requisiti formalmente specificati. In ogni caso, una possibile soluzione potrebbe essere quella di integrare l'Extreme Programming all'interno di una metodologia più rigorosa come, ad esempio, Scrum oppure Kanban (che vedremo a breve).

Un altro punto critico è la necessità di aderire dogmaticamente alle 12 pratiche, poiché l'efficacia di ciascuna di esse è strettamente dipendente dalle altre (come illustrato nella Figura 2.6): infatti, le eventuali carenze di una singola pratica vengono compensate dall'applicazione congiunta delle restanti. Pertanto, l'assenza anche di una sola di queste pratiche comprometterebbe l'intero processo.

Infine, la mancanza di una progettazione iniziale rende questa metodologia ben poco funzionale. Sebbene l'XP incoraggi il refactoring continuo, senza una direzione

generale da seguire, il design del sistema cambierà ripetutamente, allungando i tempi di sviluppo. Inoltre, tale design potrebbe rivelarsi errato, senza alcun modo di verificarlo, poiché nessuna parte del processo prevede una valutazione approfondita delle scelte progettuali. [16]

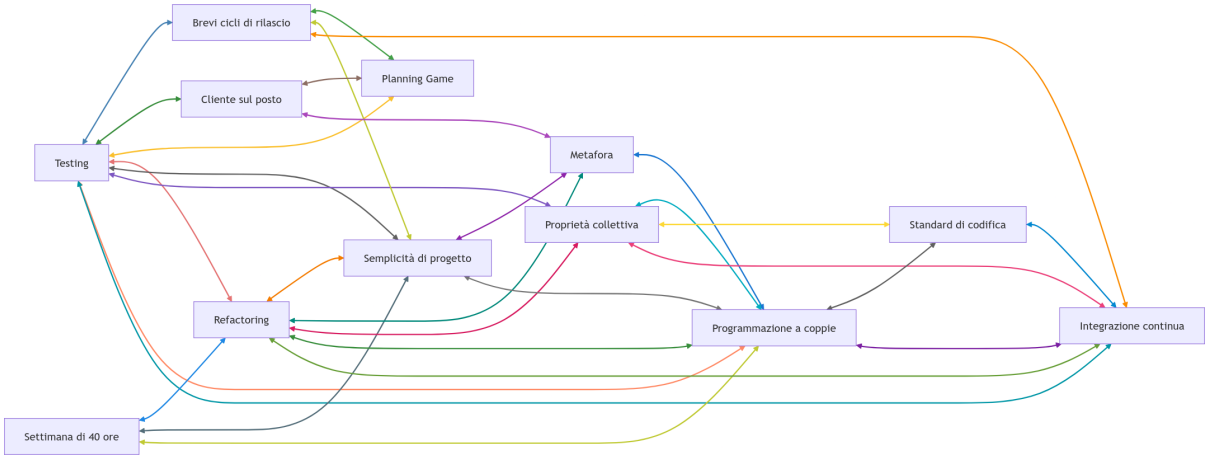

Figura 2.6: L'interdipendenza tra le 12 pratiche dell'Extreme Programming: la mancanza di una di esse comprometterebbe l'efficacia di tutte le altre.

2.5 Test-Driven Development

Ideato da Kent Beck come una delle dodici pratiche dell'*Extreme Programming* [15], il Test-Driven Development (o *TDD*) è successivamente stato ripreso come metodologia di sviluppo software a sé stante.

Il razionale dietro tale scelta è che, attraverso la stesura iniziale dei casi di test, uno sviluppatore sia in grado di definire prima **cosa** debba fare la funzionalità da implementare, per poi concentrarsi su **come** farlo soltanto in un secondo momento. Nonostante il TDD si basi principalmente sul Testing, la metodologia sfrutta significativamente anche il Refactoring, un'altra pratica dell'Extreme Programming. Dopo aver implementato una funzionalità che supera i test precedentemente definiti, viene effettuato un processo iterativo di riscrittura del codice per renderlo più efficiente rispetto alla versione iniziale. Per sintetizzare, il TDD può essere descritto come un procedimento suddiviso in tre fasi [17]:

- **Fase rossa**, che consiste nella **stesura del caso di test**, il quale dovrà verificare che la funzionalità prevista sia implementata correttamente. Inizialmente, il test dovrà fallire, in quanto la funzionalità non è stata ancora implementata;
- **Fase verde**, che prevede l'**implementazione della funzionalità** scrivendo la quantità minima di codice necessaria affinché il test passi;
- **Fase grigia** (o **Refactoring**), durante cui è necessario **revisionare la struttura del codice scritto**, al fine di migliorarne l'efficienza e la leggibilità.

Tale procedimento ha carattere iterativo, per cui si può passare più volte per una stessa fase durante la stesura di un caso di test, almeno finché l'implementazione della funzionalità non si possa ritenere completa e soddisfacente. Nella figura 2.7 è possibile vedere una rappresentazione schematica del processo.

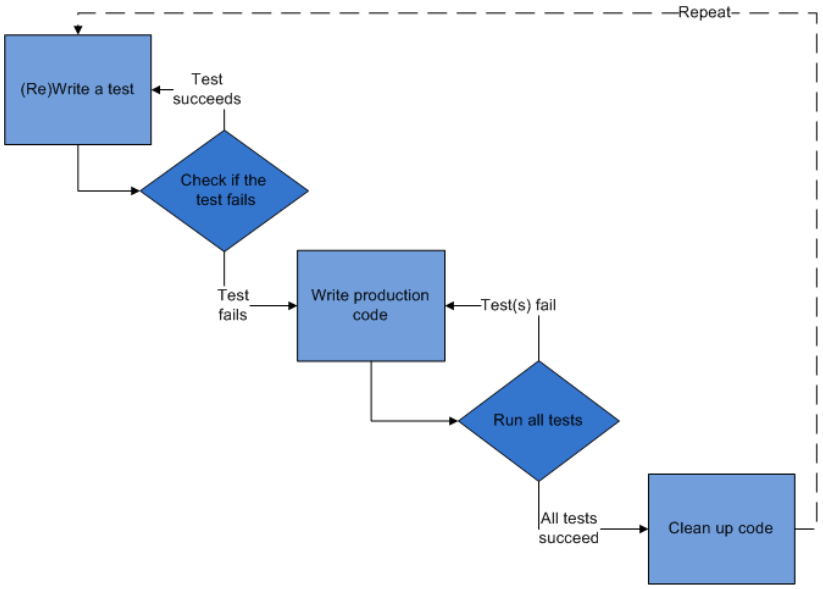

Figura 2.7: Rappresentazione del flusso di lavoro di TDD.[1] [18]

2.5.1 I vantaggi

Il TDD, così come l'XP, è facilmente applicabile anche per team molto piccoli, con l'aggiunta di essere talmente semplice da poter essere utilizzato persino per progetti individuali. Altro punto a favore è rappresentato dalla garanzia di elevata qualità del codice scritto, a patto che si segua uno specifico criterio (si pensi, anche qui, al discorso sulle metriche qualitative del software di cui tener conto).

2.5.2 Le criticità

Questa metodologia non è affatto sufficiente per progetti di medie o grandi dimensioni. Può senza dubbio rappresentare un aiuto importante alla fase implementativa, ma è necessario utilizzarla in supporto ad una metodologia in grado di gestire la progettazione ad alto livello. Infatti, in modo analogo all'XP, soffre enormemente della mancanza della fase di design, per gli stessi motivi di cui sopra.

[1] Excirial di Wikipedia in inglese, CC BY-SA 3.0 <https://creativecommons.org/licenses/by-sa/3.0>, da Wikimedia Commons

2.6 Kanban

Nata a opera di Taiichi Ohno come sviluppo del *Toyota Production System* [19], in ambito della produzione *Lean*, questa metodologia richiede la **rappresentazione del flusso di lavoro attraverso una bacheca**, sia essa fisica o digitale, **strutturata in diverse colonne**, ognuna delle quali denota uno stato del processo (ad esempio "Da Fare", "In Corso" e "Completato"). **Ogni attività è rappresentata da una scheda** (o cartellino) e viene **collocata in una delle colonne allestite**.

Si tratta di una metodologia che adotta un sistema di tipo *Pull*, poiché non si focalizza su scadenze rigide, ma permette di sviluppare una funzionalità solo quando si dispone della reale capacità di farlo. Di conseguenza, il team può concentrarsi sulla qualità del lavoro, evitando sovraccarichi e garantendo che ogni attività venga completata prima di passare alla successiva.

Nel 2010 è stata estesa da David J. Anderson [20], che l'ha riproposta come metodologia di sviluppo software: dopo aver compreso il potenziale della *Lean Production* utilizzando la **Teoria dei Vincoli** (attraverso il *Drum-Buffer-Rope*), Anderson ha scelto di adottare la metodologia Kanban, più semplice da comprendere, in un progetto di sviluppo presso Microsoft nel 2004. La figura 2.8 mostra un esempio di utilizzo della metodologia Kanban in ambito di sviluppo software.

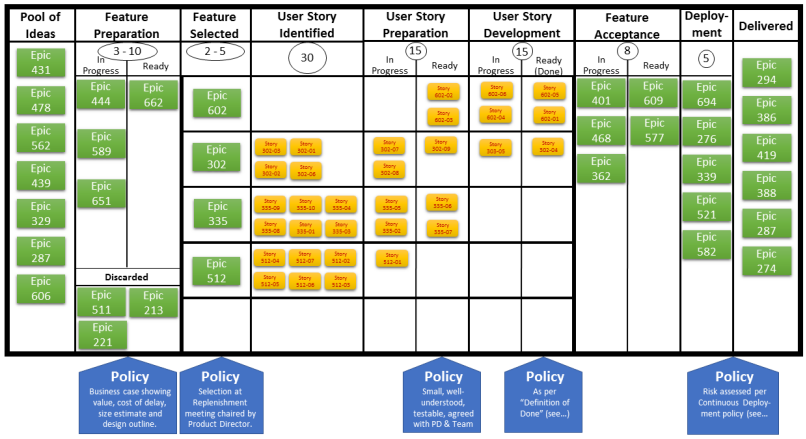

Figura 2.8: Esempio di metodologia Kanban applicata ad un progetto software.[2]

[2]Andy Carmichael, CC BY-SA 4.0 <https://creativecommons.org/licenses/by-sa/4.0>, da Wikimedia Commons

A prima vista, questo potrebbe apparire come un procedimento già contemplato dalla maggior parte delle metodologie, poiché è molto comune collocare le task su una kanban board. Tuttavia, ciò non corrisponde necessariamente a usare Kanban come metodologia di sviluppo: bisogna definire anche il modo per gestire il *Work-In-Progress* (lo vedremo a breve), cosicché il *Pull System* possa funzionare come previsto. Analizziamo ora, più in dettaglio, alcuni dei concetti fondamentali di questa metodologia.

Un **Work Item** corrisponde a ciò che abbiamo in precedenza identificato con il nome di **attività**, e **rappresenta l'unità minima di valore che andrà a percorrere l'intero flusso di lavoro** - nel nostro caso, può trattarsi di uno Use Case, una User Story, una funzionalità o un requisito, ma anche di una Issue, un bug o una Change Request.

Il **Cost of Delay** (anche chiamato **Urgency**) definisce l'urgenza con cui è necessario che un Work Item venga consegnato, prima che perda valore in modo significativo (*delay cost*). Minimizzare il *delay cost* corrisponde a un aumento del **Cost of Delay**.

La **Definition of Workflow** (o **DoW**) rappresenta il punto di partenza di ogni progetto Kanban, poiché fornisce le specifiche per i seguenti aspetti [20, 21, 22]:

- La descrizione di un Work Item, ovvero il suo contenuto. In genere, un Work Item è composto da un identificativo, una breve sintesi, uno o più assegnatari e un'eventuale data di scadenza, ma è chiaro che si possono integrare diversi elementi aggiuntivi (si pensi alla priorità, ad esempio);

- La determinazione dei punti di inizio e fine nel flusso di lavoro. È importante notare che possono esserci diversi punti di inizio e fine, i quali delimitano lo stato dei diversi sotto-processi che fanno parte del processo di sviluppo;

- La definizione degli stati attraverso i quali transitano i Work Item dalla partenza all'arrivo (qualunque Work Item che non si trovi in uno stato di inizio o fine è considerato come *Work-in-Progress*);

- La descrizione delle politiche per controllare il Work-In-Progress (ad es. limitare il numero massimo di Work Item in una parte del flusso di lavoro, stabilire i criteri di ingresso e uscita per ogni stato, ecc.);

- Una *Service Level Expectation*, che rappresenta il tempo previsto per il completamento di un Work Item;
- Una *Service Level Capability*, che definisce il throughput del processo;
- Un *Service Level Agreement*, che stabilisce ciò che viene concordato col cliente;
- Una *Service Fitness Threshold*, che assicura un limite inferiore alla qualità dei *deliverable*, sotto cui vengono ritenuti inaccettabili dal cliente.

La bacheca (o *board*) può considerarsi la visualizzazione della *DoW*: difatti, ogni stato corrisponderà a una colonna, i Work Item saranno le schede presenti sulla bacheca, mentre i limiti imposti verranno gestiti come il numero massimo di elementi collocabili in una specifica sezione della bacheca.

Il **Commitment Point** è la sezione del flusso che, una volta attraversata da un Work Item, impone che quest'ultimo raggiunga la fine del processo. Sino a quando non viene attraversato, non vi è certezza che un Work Item debba essere effettivamente realizzato. L'attraversamento indica l'approvazione da parte del cliente che tale richiesta debba essere eseguita, e il team deve provvedere alla consegna.

Il **Delivery Point** è il punto del flusso in cui un Work Item viene considerato completo oppure consegnato al cliente.

Il **Lead Time** è il tempo medio richiesto da un Work Item per passare dal Commitment Point al Delivery Point.

2.6.1 I vantaggi

Il fulcro di Kanban verte sulla segnalazione visiva, il che rende molto più immediato comprendere lo stato di avanzamento del processo produttivo. Inoltre, il sistema Pull-based garantisce che si cominci a sviluppare una funzionalità solo quando la capacità del team lo permette: la definizione di limiti al Work-in-Progress fa sì che il Lead Time venga drasticamente diminuito, permettendo di concentrarsi sui lavori già avviati, prima di iniziarne di ulteriori. Infine, pur essendo funzionante come metodologia a sé stante, spesso Kanban viene utilizzata come metodologia di passaggio per migrare progressivamente verso altre metodologie di tipo Agile, come ad esempio Scrum — tant'è che viene spesso adottata in concomitanza con quest'ultimo, sia come metodologia separata (*Scrum con Kanban*) che come un ibrido tra le due (*Scrumban*).

2.6.2 Le criticità

È importante definire dei limiti al WiP adeguati, altrimenti potrebbero non esserci miglioramenti significativi nei tempi di sviluppo. Altrettanto essenziale, inoltre, è il continuo aggiornamento della board: in caso contrario, diventa estremamente difficile tenere traccia dello stato del processo ed estrapolare delle metriche utili al miglioramento dello stesso. Ciò comporterebbe anche un elevato rischio di stagnazione del progetto. In egual misura, è fondamentale saper scegliere il grado di dettaglio della board: una bacheca troppo semplice descriverebbe in modo troppo grossolano lo stato di avanzamento del progetto; al contrario, una troppo complessa causerebbe troppo overhead nella gestione del progetto.

2.7 Collaudo del software

Come già anticipato in precedenza, è necessario che il software venga testato. Il motivo è molto semplice: il processo di sviluppo di un software fa sì che, durante la fase di codifica, l'implementazione possa avere delle discrepanze rispetto alle specifiche definite durante la progettazione, o che la progettazione stessa sia fallace.

Non è affatto raro che un software si comporti in modo imprevisto, ed è compito di un bravo ingegnere del software fare in modo di minimizzare il numero di *fault* — ovvero porzioni di codice problematiche che potrebbero dar luogo a *failure*, cioè una deviazione del comportamento osservato da quello atteso.

Il metodo più comunemente utilizzato per cercare il maggior numero di *fault* in modo sistematico è il Testing, che mira a verificare la conformità del sistema agli output previsti in relazione a quelli osservati: nel caso in cui differiscano, saremo in grado di certificare la presenza di fault.

Il Testing non può certificare l'assenza di errori, ma può solo confermarne la presenza per alcuni input prestabiliti. Sebbene questo approccio offra un risultato parziale, risulta tuttavia molto utile per particolari casi d'interesse. L'obiettivo principale è l'identificazione dei casi che sono in grado di massimizzare la probabilità di trovare *fault* nel sistema.

Il Testing viene effettuato attraverso l'esecuzione di più *test*. Un *test* (o *Test Suite*) è un insieme di *Test Case*, il cui compito è provare molteplici dati di input in diverse parti del sistema e controllare che i risultati coincidano con quelli attesi.

2.7.1 Livelli di testing

Per poter collaudare in maniera efficace differenti parti del sistema, è necessario utilizzare diversi tipi di *test*, ognuno dei quali è indirizzato a uno specifico grado di granularità nel collaudo. [23]

Il **Test di Unità** (o **Unit Testing**) è una forma di *Testing* che prevede il **collaudo di ogni singola unità del software**. Un'unità è definita come il minimo componente del sistema da testare e, generalmente, nella programmazione orientata agli oggetti corrisponde a una classe, a un metodo o a un modulo; in alcuni casi, tuttavia, può corrispondere anche a un insieme di metodi che, combinati, soddisfano un requisito specifico. Scopo del Test di Unità è di assicurarsi che ogni componente testato funzioni correttamente, secondo la sua specifica: ciò non è sufficiente per collaudare l'intero sistema e, in ogni caso, dovrebbe essere utilizzato congiuntamente ad altri tipi di test.

Il **Test di Integrazione** (o **Integration Testing**) prevede il collaudo di più componenti software in gruppo, andando quindi a testare l'integrazione di tali componenti in sottosistemi di livello superiore. Esistono quattro principali strategie per testare gruppi di componenti:

- *Big Bang Integration*, ovvero testare l'integrazione di tutti i componenti a livello dell'intero sistema;
- *Bottom-Up Integration*, cioè partire dai sottosistemi dei livelli più bassi per poi risalire;
- *Top-Down Integration*, che adotta la strategia inversa del *Bottom-Up* e quindi parte dall'alto per poi scendere;
- *Sandwich Integration*, un ibrido delle due precedenti strategie, che parte sia dall'alto che dal basso, fino a convergere ad un livello *target*.

Il **Test di Sistema** (o **System Testing**) mira al collaudo del sistema nella sua interezza, per verificare che sia conforme ai requisiti funzionali e non funzionali. Si suddivide in diverse attività:

- *Functional Testing*, che certifica l'aderenza del sistema alle funzionalità specificate durante la fase di analisi dei requisiti;
- *Performance Testing*, una forma di collaudo che ha il compito di analizzare le prestazioni del sistema;
- *Acceptance Testing*, utilizzato dal cliente per valutare se il sistema rispecchi davvero le sue necessità o meno;
- *Installation Testing*, per verificare che il sistema funzioni correttamente nell'ambiente di utilizzo (ovvero, l'ambiente di produzione).

2.7.2 Strategie di testing

È impossibile collaudare un sistema verificandone il comportamento per ogni singolo input possibile, il che rende la ricerca esaustiva — ancora una volta — un metodo inapplicabile per casi reali. Esistono diverse strategie che consentono di verificare la validità del sistema per i casi più significativi, il che ci consente di massimizzare la probabilità di individuare dei *fault*. Tali strategie si differenziano in due categorie principali, il **Black-Box Testing** e il **White-Box Testing**. La prima si concentra nell'analizzare esclusivamente i risultati forniti dal sistema, senza analizzarne il comportamento, mentre la seconda verte proprio sulla verifica del comportamento interno di quest'ultimo. Di seguito vedremo alcune delle strategie di testing più comunemente utilizzate. [24]

Model-based Testing Una forma di *Black-Box Testing*, questa strategia prevede la costruzione di un modello formale (ad es. un automa a stati finiti), in modo da verificare l'aderenza del sistema al modello realizzato.

Equivalence Class Partitioning Questa strategia, di tipo *Black-Box*, determina i casi di test rilevanti attraverso il raggruppamento degli input secondo classi di equivalenza e la selezione di campioni rappresentativi da ciascuna classe. Per un programma che accetta numeri da 1 a 10, si possono creare tre classi di equivalenza: numeri sotto 1, tra 1 e 10, e sopra 10. Così facendo, i casi di test si riducono a 3 invece di infiniti.

Boundary Value Testing Strategia simile all'ECP, ma che si focalizza sul testare esclusivamente i casi limite tra i valori validi. Una sua variante è il Robustness Testing, che include anche i valori non validi.

Combinatorial Testing Questa strategia consiste nel riassumere tutti i potenziali casi di test mediante la scomposizione della specifica del programma in una serie di attributi verificabili in maniera sistematica, analizzandone l'interazione mediante l'esame delle combinazioni di scelte, siano esse consentite o meno. Un esempio ben noto è il *Category-Partition*, che estrae, a partire dalle specifiche, delle funzionalità testabili in modo indipendente tra loro.

Path Testing Una strategia di tipo *White-Box*, essa prevede che ogni *path* del programma testato venga eseguito almeno una volta. In breve, si realizza un grafo che descrive tutti i possibili percorsi del codice da testare, dal quale sarà poi possibile ricavare i casi di test che percorrano ognuno di essi.

Fault-based Testing Un'ulteriore strategia *White-Box* è rappresentata dal Fault-based Testing. Questa metodologia consiste nel creare versioni alternative del programma da testare, contenenti delle piccole variazioni nel codice sorgente, le quali potrebbero potenzialmente causare dei *fault*. La variante più conosciuta è il *Mutation Testing*, che crea delle varianti del programma (chiamate *mutanti*) in modo deterministico attraverso l'utilizzo di pattern (chiamati *operatori di mutazione*).

CAPITOLO 3

Utilizzo di Kanban e Test-Driven Development per lo sviluppo di un'applicazione web

3.1 Obiettivo dell'applicazione web

Il sistema organizza il personale dell'azienda, coordinando le risorse per i progetti e pianificando il carico di lavoro in accordo con le strategie a breve e lungo termine, migliorando così l'efficacia nel raggiungimento degli obiettivi. La nuova piattaforma è un'applicazione web che automatizza ampiamente la gestione. Poiché il sistema precedente richiedeva l'inserimento *manuale* di tutte le risorse, il nuovo sistema rende più semplice e automatizza tutte le procedure di pianificazione e dimensionamento.

Alle sue fondamenta, il sistema utilizza Laravel, un framework open-source per sviluppatori PHP che offre una suite di strumenti utili a una veloce prototipazione e conseguente realizzazione di un'applicazione web, assistendo i developer dalla modellazione della base di dati fino al collaudo e al rilascio.

3.2 Realizzazione

Fase preliminare L'iter progettuale è cominciato attraverso l'interazione con gli stakeholder aziendali per raccogliere informazioni generali sul sistema da realizzare. In seguito, ho costituito una prima minimale Kanban board, dove ho elencato i vari passaggi necessari per completare il progetto, organizzando il flusso di lavoro in task — ognuna delle quali associata a una specifica fase del processo di sviluppo (Raccolta Informazioni, Stesura DoW, Elicitazione Requisiti, Analisi Requisiti, Implementazione, ecc.). Infine, ho creato una Definition of Workflow, propedeutica alla realizzazione della Kanban board inerente alla gestione del progetto, al fine di poter formalizzare il flusso di lavoro per l'implementazione delle varie funzionalità che sarebbero state successivamente identificate e realizzate.

Raccolta informazioni Durante questa fase ho creato una prima Kanban board orientativa (figura 3.1), per potermi meglio preparare al lavoro da affrontare. Le prime task create erano relative allo studio delle tecnologie da utilizzare, culminando con la realizzazione di un piccolo applicativo di esempio in cui ho acquisito maggiore dimestichezza con l'ecosistema Laravel. In seguito, dopo averne appreso il funzionamento, ho discusso con gli stakeholder dell'azienda per poter cominciare ad affrontare la sfida progettuale che avrebbe rappresentato il mio percorso di tirocinio.

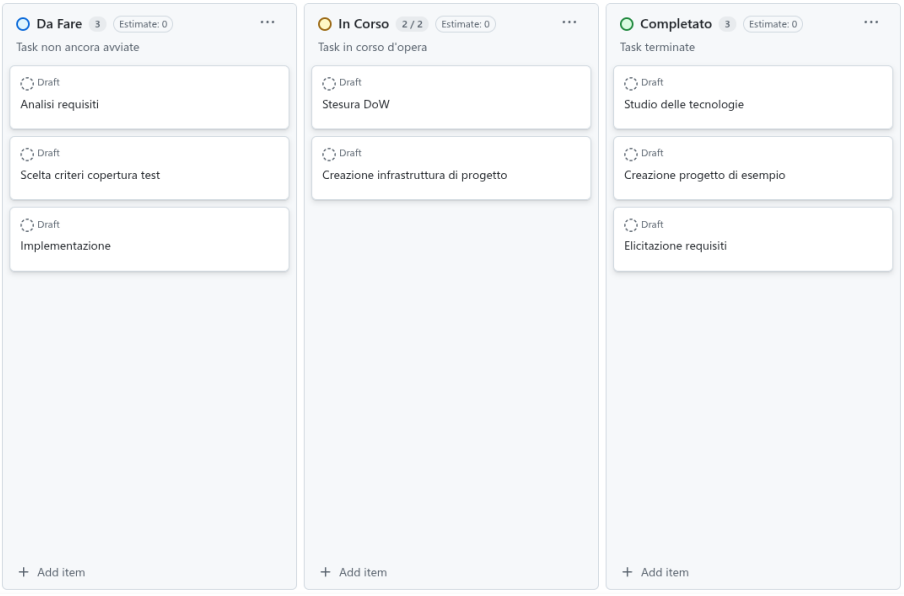

Figura 3.1: Kanban board minimale per suddividere il processo di sviluppo in varie task.

Stesura Definition of Workflow Dopo aver raccolto informazioni preliminari sul sistema da realizzare — ovvero il dominio applicativo e le informazioni sul sistema in uso dall'azienda — ho cominciato la stesura di una Definition of Workflow, in modo tale da costituire un processo formale per la Kanban board che avrei poi utilizzato durante le fasi implementative del progetto.

Una volta ultimata, ho potuto chiarire tutti i dettagli relativi al flusso di lavoro che avrebbero dovuto affrontare le varie User Stories:

- in primis, sono state create quattro colonne denominate rispettivamente *Backlog*, *To Do*, *In Progress*, *Done*;
- la colonna *To Do* ha assunto il ruolo di Commitment Point, assicurando che le User Stories che vi giungono vengano completate, mentre la colonna *Done* è stata designata come Delivery Point;
- sono stati imposti dei limiti al numero di User Stories che collocabili contemporaneamente nelle colonne *To Do* e *In Progress*, rispettivamente tre e uno, permettendomi di concentrarmi su una sola User Story alla volta e di implementare una delle tre più urgenti scelte;
- infine, ho attribuito delle semplici priorità a ciascuna User Story, cioè *Bassa*, *Media* e *Alta*. Per distinguere quale delle due Stories con la stessa priorità dovesse essere realizzata prima, posizionavo una delle due più in alto dell'altra per identificare subito quella più urgente.

La figura 3.2 rappresenta un esempio della Kanban board risultante.

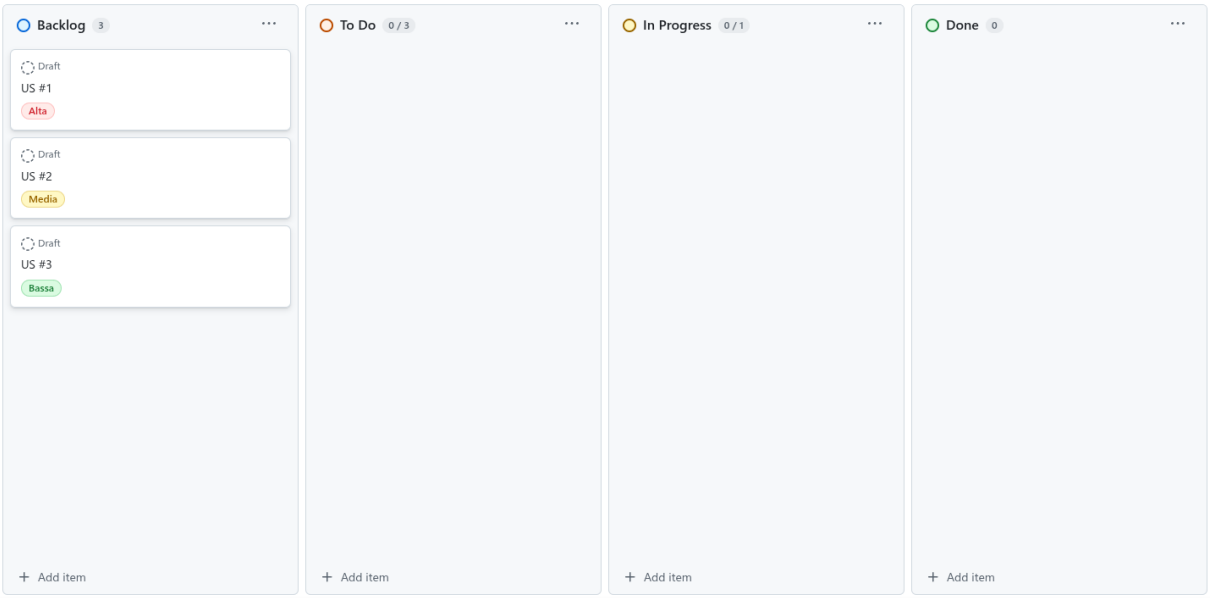

Figura 3.2: Kanban board esemplificativa, risultato della Definition of Workflow.

Elicitazione dei requisiti Questa è stata la prima vera e propria fase che ha dato inizio al progetto: ho creato un breve documento in cui ho raccolto maggiori informazioni presso gli stakeholder aziendali sul sistema da realizzare, rappresentando il tutto sotto forma di User Stories. Ho, inoltre, identificato gli attori del sistema — *Amministratore* e *Risorsa* — focalizzandomi però esclusivamente sull'*Amministratore*, secondo quanto concordato con l'azienda. Di seguito, alcune tra le più importanti User Stories:

- *Come Amministratore, voglio poter creare, modificare ed eliminare le risorse aziendali*;
- *Come Amministratore, voglio poter creare, modificare ed eliminare i progetti*;
- *Come Amministratore, voglio poter avviare un progetto non ancora avviato, in modo da poter creare il calendario per le varie risorse.*

Analisi dei requisiti Durante questa fase, ho analizzato tutte le User Stories, ricavando diversi artefatti che mi facessero maggiormente comprendere come suddividere il sistema in componenti logiche, oltre che facilitare la fase implementativa, in quanto ciò mi ha permesso di colmare diverse lacune derivanti dal fatto che una semplice User Story non è sufficiente a descrivere il comportamento di un'intera funzionalità. Prenderò come esempio soltanto una User Story relativa alla creazione del calendario di un progetto; tuttavia, bisogna considerare il fatto che tale processo è stato ripetuto per tutte le User Stories ricavate durante la fase di elicitazione. Al termine di questa fase, infine, ho ricavato il Class Diagram ad alto livello, che mi ha permesso di distinguere le entità significative del sistema (figura 3.3).

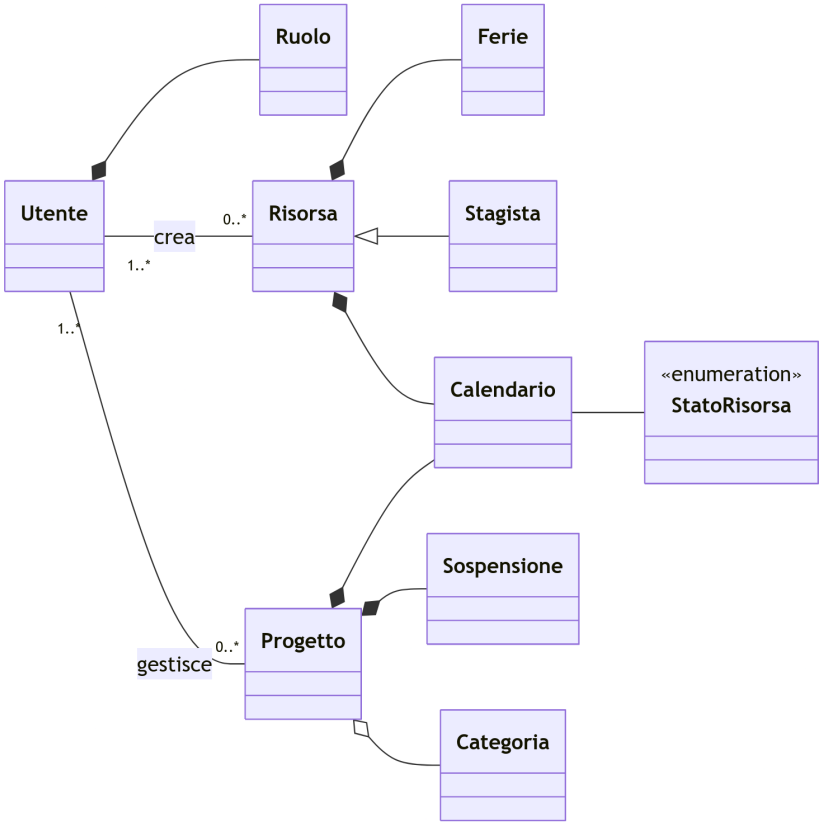

Figura 3.3: Class Diagram del sistema.

Esempio di analisi: creazione del calendario di progetto Come anticipato, la User Story presa in esame è relativa alla creazione del calendario di progetto. A partire da essa ho ricavato due diversi artefatti, ovvero il Diagramma a Stati Finiti — rappresentato in figura 3.4 — e il Sequence Diagram — in figura 3.5.

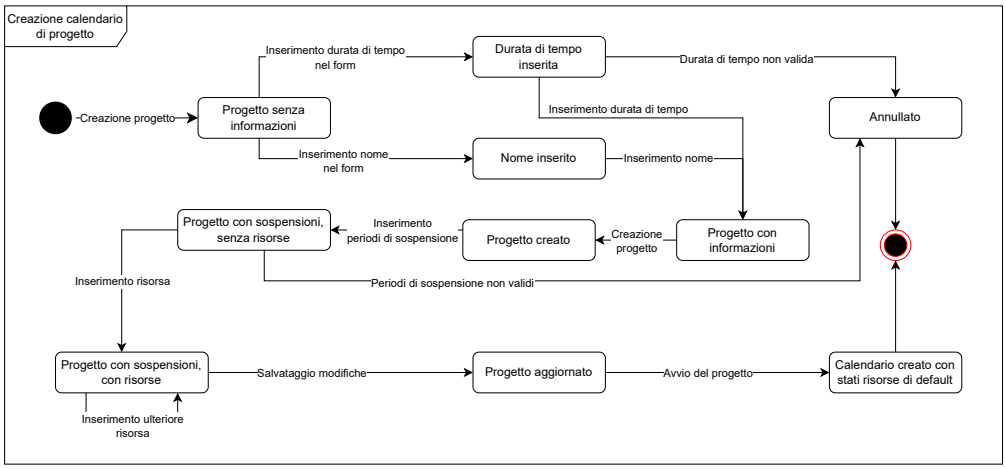

Figura 3.4: Macchina a stati finiti che descrive il processo di creazione del calendario per un nuovo progetto.

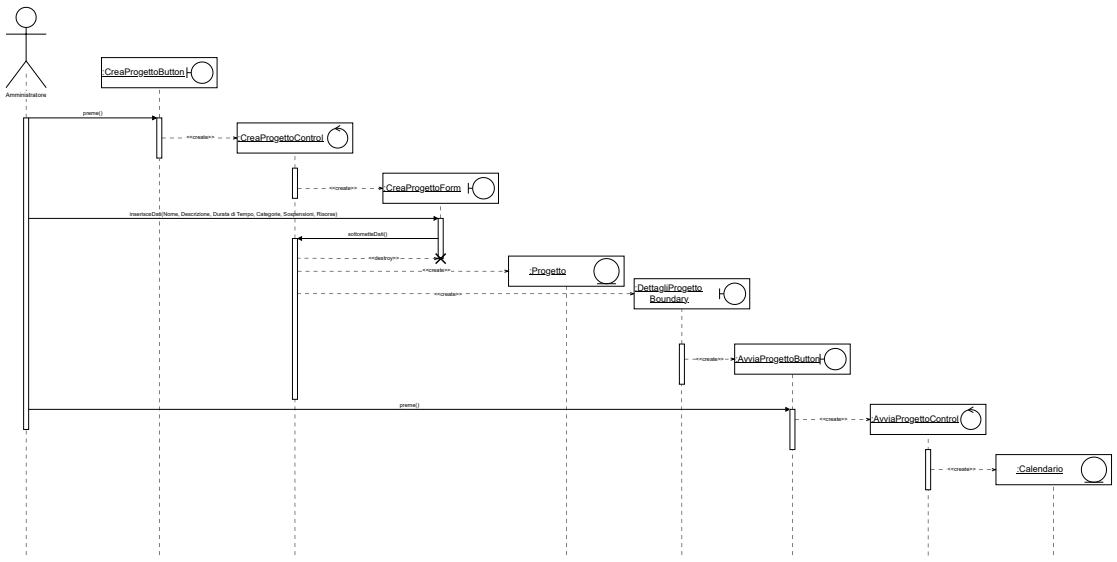

Figura 3.5: Diagramma di sequenza che descrive il processo di creazione di un nuovo progetto, con annessa creazione del calendario nell'istante di avvio dello stesso.

Implementazione Una volta terminata la fase di analisi, ho proceduto con la fase implementativa: dopo aver inizializzato un esoscheletro di progetto, offerto dalla suite di strumenti messi a disposizione da Laravel, ho utilizzato il Class Diagram per creare lo schema per il database sotto forma di *migrations* — un formato dichiarativo utilizzato da Laravel per poter alterare lo schema di un database, in genere utilizzato per creare, modificare o eliminare tabelle al suo interno. Ho finalizzato la creazione del progetto configurando **Laravel Sail** – il quale mette a disposizione uno stack Docker Compose con vari servizi predefiniti come un DBMS (in questo caso specifico, PostgreSQL), un worker per processare gli eventi emessi dall'app (tra cui anche notifiche e mail) e il servizio stesso in cui eseguire l'applicativo, rendendo il tutto accessibile in locale sull'ambiente di sviluppo – per una più veloce integrazione delle funzionalità progressivamente realizzate. In seguito, ho arricchito la suite di testing procedendo con l'installazione di Infection, al fine di introdurre il mutation testing.

Prima di procedere con l'implementazione vera e propria, ho deciso le metriche da adottare per considerare una funzionalità correttamente e completamente realizzata: esse sono la *code coverage*, la quale rappresenta la percentuale di codice delle funzionalità effettivamente eseguita dalla test suite, e la *mutation score*, ovvero la percentuale delle mutazioni rilevate dal codice testato e coperto dai casi di test (è quindi strettamente dipendente dalla *code coverage*). Ho stabilito che il criterio minimo di accettabilità per una funzionalità sia ottenere almeno l'80% del punteggio in ogni metrica. Una volta conclusa la preparazione, ho proseguito con la selezione delle User Stories da implementare in base alla loro priorità, utilizzando TDD per dedurre le funzionalità da implementare partendo prima dallo sviluppo dei casi di test relativi alla specifica User Story (il Class Diagram risultante è rappresentato in figura 3.6). Anche in questo caso, utilizzeremo come esempio la creazione del calendario di progetto.

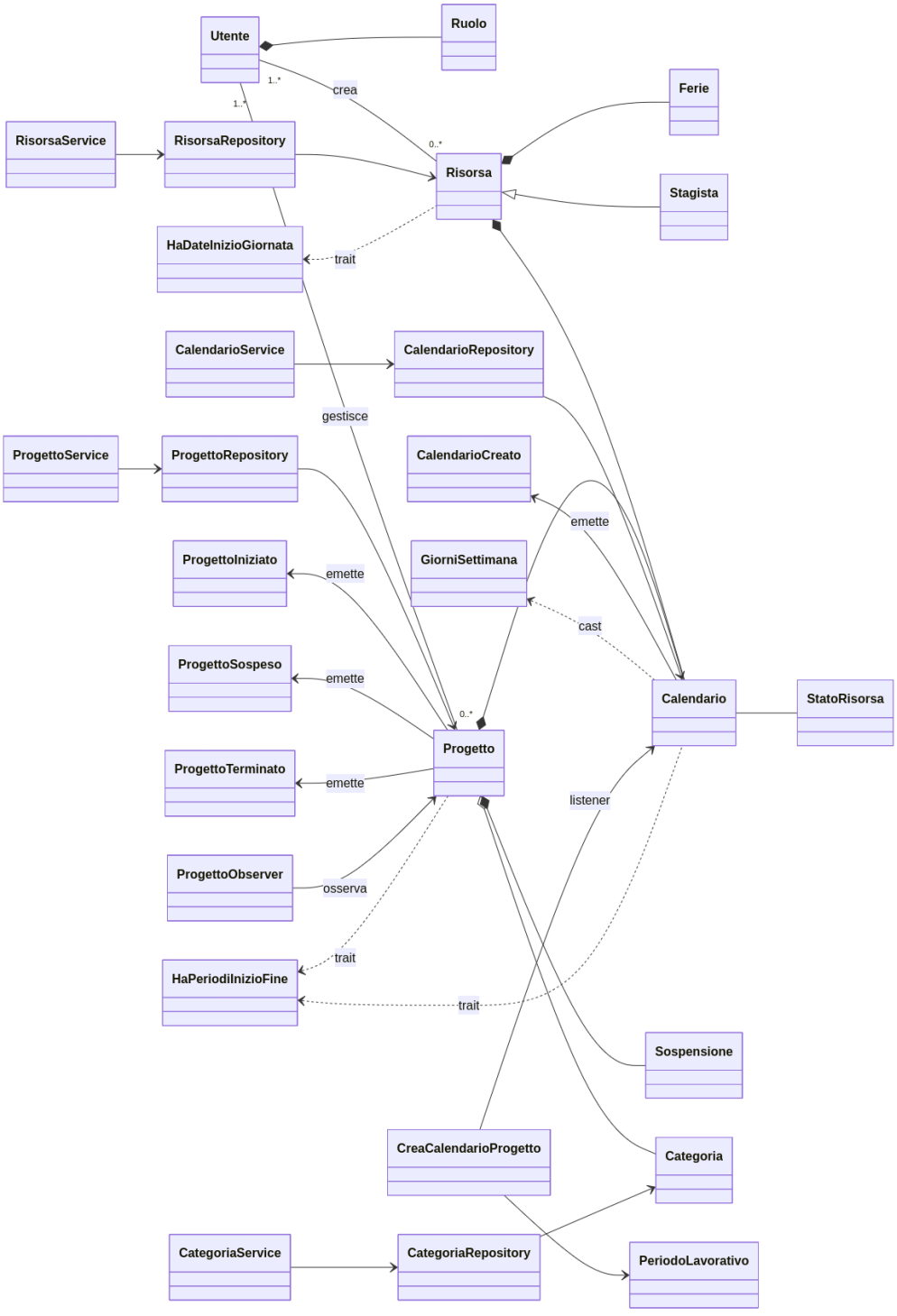

Figura 3.6: Class Diagram di basso livello.

Esempio di TDD: creazione del calendario di progetto Una volta selezionata la User Story da implementare, essa veniva in primis spostata nella colonna "In Progress" nella Kanban board, dopodiché procedevo con la scrittura dei casi di test basandomi sugli artefatti a essa correlati.

Il modus operandi è il seguente: estrapolare dei casi di test in base alle funzionalità da realizzare che erano state identificate negli artefatti; scrivere dei casi di test relativi alla funzionalità da implementare, che dovranno inizialmente fallire; procedere con l'implementazione della funzionalità scrivendo il codice minimo necessario a far passare i test; riscrivere sia il test che la funzionalità in modo più efficiente finché le metriche di code coverage e mutation score non raggiungano almeno l'80%.

Integrazione continua A conclusione di ciascuna iterazione nel corso del refactoring di ogni specifica funzionalità, Laravel Sail mi ha permesso di integrare in modo continuo il codice appena sviluppato, eseguendo automaticamente i test case. Due report (figure 3.7 e 3.8) mi fornivano le metriche di coverage da utilizzare per valutare la qualità del codice, dopo di che decidevo se iniziare una nuova iterazione del refactoring o passare alla prossima User Story.

	Lines	
Total	92.61%	351 / 379
Casts	100.00%	11 / 11
Facades	96.00%	24 / 25
Listeners	95.00%	38 / 40
Repositories	100.00%	115 / 115
Services	86.70%	163 / 188

Figura 3.7: Valori di code coverage per alcuni package del progetto.

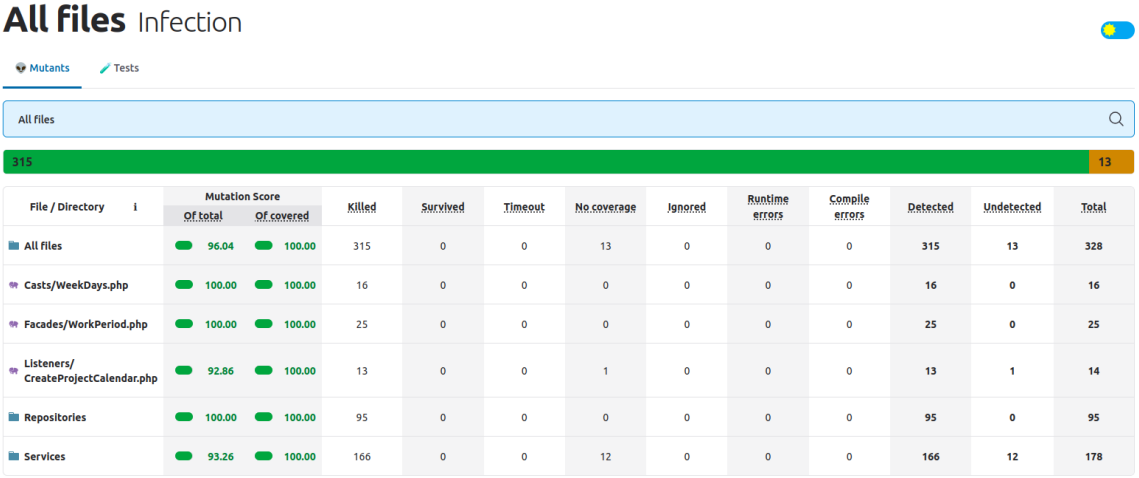

Figura 3.8: Valori di mutation score per alcuni file del progetto.

CAPITOLO 4

Conclusioni

Grazie all'approccio adottato, è stato possibile accelerare il raggiungimento della fase implementativa, riducendo lo sforzo richiesto per la predisposizione della documentazione iniziale. L'impegno richiesto dal metodo Kanban è risultato essere minimo, poiché la possibilità di visualizzare immediatamente lo stato del progetto in qualsiasi momento consentiva di individuare facilmente le componenti mancanti, quelle già completate e le aree su cui focalizzarsi, attraverso un'unica sintesi visiva. A parità di tempistiche, dunque, un approccio Agile consente il più rapido raggiungimento di un prodotto funzionante, contrapponendosi alle tradizionali metodologie di sviluppo, con tempi ben più diluiti. Da non sottovalutare, inoltre, il fatto che l'effort richiesto sia stato gestibile da un team composto da una singola persona, il che le rende applicabili anche in contesti individuali.

Nozioni apprese Nel corso dell'esperienza formativa si sono rivelate particolarmente significative le conoscenze acquisite in merito a framework, metodologie di sviluppo e tecniche di collaudo, le quali hanno contribuito ad arricchire il mio bagaglio di competenze tecnologiche e metodologiche. In primis, dal punto di vista tecnologico, Laravel e la sua suite di strumenti si sono rivelati molto utili per minimizzare i tempi di inizializzazione del progetto, eliminando la necessità di aggiungere codice boilerplate — codice prettamente inutile al dominio applicativo in sé, ma necessario affinché il progetto stesso possa funzionare correttamente — per via dell'enorme contributo che apporta nell'astrarre e nel semplificare la vita allo sviluppatore, che sarà responsabile esclusivamente della logica di business inerente all'applicativo da realizzare. L'utilizzo di Infection, poi, ha reso più robusti i casi di test scritti, implementando una strategia di mutation testing che ha consentito di valutare con precisione la qualità della test suite. Introducendo difetti nel codice, è stato possibile identificare e colmare eventuali lacune nei test, migliorando l'affidabilità complessiva del software. Per quanto riguarda le metodologie di sviluppo software, Kanban è stato essenziale per ridurre il tempo di progettazione rispetto a un modello tradizionale di gestione del ciclo di vita del software, con il beneficio di richiedere una preparazione minima per una corretta implementazione (anche se è necessaria una certa disciplina e costanza per aggiornare regolarmente lo stato della bacheca, attività che diventa più intuitiva con la pratica continua). Il Test-Driven Development, infine, ha rappresentato un'inversione del consueto processo di collaudo al termine della fase implementativa: se applicato con rigore, esso consente di astrarre concettualmente la logica di business, rafforzando un approccio dichiarativo (basato sul *cosa*) piuttosto che quello imperativo (basato sul *come*), rappresentando dunque un valido aiuto per scrivere codice più efficiente e snello.

Considerazioni finali Kanban mi ha introdotto pragmaticamente alla dottrina Agile, permettendomi di gestire autonomamente un progetto con meccanismi di ottimizzazione continua, attraverso la visualizzazione delle attività e la flessibilità degli obiettivi. Questo approccio, grazie anche all'introduzione di metriche qualitative per valutare lo stato di salute del progetto, ha incrementato significativamente la mia efficienza, rendendomi un miglior ingegnere del software. Inoltre, l'integrazione di pratiche come il Test-Driven Development e il mutation testing ha ulteriormente affinato la qualità del codice, garantendo che ogni funzionalità fosse testata in modo rigoroso. Utilizzando Laravel e Livewire, infine, ho potuto sviluppare applicazioni web scrivendo una quantità di codice molto ridotta, mantenendo così un alto standard di testabilità e manutenibilità del sistema.

APPENDICE A

Tecnologie Utilizzate

A.1 Laravel

Laravel è un framework open-source di tipo MVC (**M**odel-**V**iew-**C**ontroller, un pattern architetturale comunemente adottato per la separazione della logica di presentazione dei dati dalla logica di business) per il linguaggio PHP. Creato nel 2011 da Taylor Otwell, esso prende ispirazione da Symfony — un altro framework per PHP, da cui Laravel prende alcuni componenti — ed ha lo scopo di semplificare lo sviluppo, la distribuzione e la manutenzione di un'applicazione web: alcune feature, infatti, includono la gestione dei database e della cache, la facilitazione delle procedure di autenticazione e di autorizzazione degli utenti, il supporto per la localizzazione, un sistema di templating per le pagine web, e molto altro. Laravel, inoltre, rende molto più facile il collaudo dell'applicazione, fornendo una vasta gamma di strumenti di mock e una visualizzazione immediata delle metriche di coverage, integrando diversi framework di test come PHPUnit e Pest.

A.2 Laravel Livewire

Livewire è un package per Laravel che utilizza il framework Alpine JS per realizzare interfacce reattive. Si integra con Laravel Blade, il sistema di templating di default, e consente di creare dei componenti riutilizzabili in più viste e dinamici, il cui comportamento può essere specificato in linguaggio PHP - diversamente da altri approcci comuni, che prevedono la realizzazione di un frontend ad hoc utilizzando uno dei più popolari framework JavaScript, tra cui Vue, Angular e React.

A.3 Alpine JS

Un framework JavaScript minimale che offre interattività e reattività con un approccio dichiarativo, nato per alleggerire la complessità dei framework tradizionali. Si caratterizza per una sintassi inline e permette agli sviluppatori di aggiungere comportamenti dinamici con poche semplici direttive. Rappresenta un'inversione del paradigma dettato dai framework tradizionali perché, mentre React, Vue e Angular costruiscono interfacce attraverso componenti JavaScript, Alpine si integra direttamente nell'HTML, semplificando radicalmente l'approccio allo sviluppo web.

A.4 Tailwind CSS

Tailwind è un framework CSS *utility-first*, in quanto — differentemente da altri framework come Bootstrap — mette a disposizione un vasto elenco di classi utility da utilizzare per realizzare componenti su misura, piuttosto che offrire una serie di componenti prefabbricati. Tale scelta consente di avere maggiore flessibilità nel design delle pagine web, promuovendo un approccio più modulare e personalizzabile alla progettazione delle interfacce.

A.5 PHPUnit

È un framework di Unit Testing per il linguaggio PHP, creato da *Sebastian Bergmann* nel 2001. Si basa sull'architettura *xUnit* introdotta da *Kent Beck* ed *Erich Gamma* per testare le componenti software in modo automatico.

A.6 Infection

Infection è una libreria di Mutation Testing per PHP, basata su mutazioni introdotte nell'AST (**A**bstract **S**yntax **T**ree, una rappresentazione ad albero del codice sorgente). Supporta diversi framework di test, tra cui anche PHPUnit, e viene utilizzata come strumento da riga di comando nella cartella di progetto. In sintesi, questa libreria avvia il processo garantendo che l'esecuzione della suite di test sia priva di errori. Successivamente, esamina il codice sorgente e genera mutanti tramite operatori di mutazione. Per ciascun mutante creato, verifica se i test correlati riescono a individuare le modifiche e a fallire di conseguenza. Infine, compila i risultati ottenuti, fornendo varie metriche riguardanti la copertura dei mutanti.

A.7 Docker

Docker è una piattaforma di containerizzazione che incapsula servizi e dipendenze in unità standardizzate chiamate *container*, garantendo esecuzione uniforme su qualsiasi sistema. I container condividono il kernel dell'host, risultando più leggeri delle macchine virtuali e ottimizzando l'allocazione delle risorse. Un componente aggiuntivo, chiamato Docker Compose, semplifica la gestione di applicazioni multi-container, consentendo la configurazione e l'orchestrazione di servizi interconnessi mediante un file YAML, facilitando la definizione dichiarativa dell'architettura software a microservizi.

A.8 Laravel Sail

Un componente di Laravel che offre un'astrazione su Docker Compose, semplificando la configurazione dell'ambiente di sviluppo per applicazioni PHP. Sail fornisce un'interfaccia minimale per gestire uno stack Docker, permettendo agli sviluppatori di avviare rapidamente un ambiente di sviluppo standardizzato con servizi predefiniti come database, server web e altri servizi necessari. Sail nasconde la complessità della configurazione sottostante, rendendo l'implementazione e la gestione dell'infrastruttura di sviluppo più accessibile e immediata.

Bibliografia

[1] W. W. Royce, "Managing the development of large software systems," *Proceedings of IEEE WESCON*, 1970. (Citato alle pagine iii, 4, 7 e 8)

[2] Wikipedia contributors, "V-model — Wikipedia, the free encyclopedia," 2025. [Online]. Available: https://en.wikipedia.org/w/index.php?title=V-model&oldid=1271927212 (Citato alle pagine iii e 9)

[3] Wikipedia, "Modello a spirale — wikipedia, l'enciclopedia libera," 2023. [Online]. Available: https://it.wikipedia.org/w/index.php?title=Modello_a_spirale&oldid=135020905 (Citato alle pagine iii e 10)

[4] Wikipedia contributors, "Scrum (software development) — Wikipedia, the free encyclopedia," 2025. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Scrum_(software_development)&oldid=1286025115 (Citato alle pagine iii e 13)

[5] R. Kneuper, "Sixty years of software development life cycle models," *IEEE Annals of the History of Computing*, vol. 39, no. 3, pp. 41–54, 2017. (Citato a pagina 4)

[6] G. Müller-Ettrich, "System development with v-model and uml," in *The Unified Modeling Language*, M. Schader and A. Korthaus, Eds. Heidelberg: Physica-Verlag HD, 1998, pp. 238–249. (Citato a pagina 9)

[7] B. Boehm, "A spiral model of software development and enhancement," *SIGSOFT Softw. Eng. Notes*, vol. 11, no. 4, p. 14–24, Aug. 1986. [Online]. Available: https://doi.org/10.1145/12944.12948 (Citato a pagina 10)

[8] K. Beck *et al.*, "Manifesto for agile software development," https://agilemanifesto.org, 2001. (Citato a pagina 11)

[9] K. Schwaber, "Scrum development process," in *Business Object Design and Implementation: OOPSLA '95 Workshop Proceedings 16 October 1995, Austin, Texas*, J. Sutherland, C. Casanave, J. Miller, P. Patel, and G. Hollowell, Eds. London: Springer London, 1997, pp. 117–134. (Citato a pagina 12)

[10] J. Sutherland, *Fare il doppio in metà tempo*. Rizzoli, 2015. (Citato alle pagine 12 e 14)

[11] K. S. Rubin, *Essential Scrum: A practical guide to the most popular Agile process*. Addison-Wesley, 2012. (Citato a pagina 12)

[12] K. Schwaber and J. Sutherland, "The scrum guide," https://scrumguides.org/, 2020. (Citato a pagina 12)

[13] M. Fowler, "The state of agile software in 2018," https://martinfowler.com/articles/agile-aus-2018.html, 2018. (Citato a pagina 15)

[14] R. Jeffries, "Dark scrum," https://ronjeffries.com/articles/016-09ff/defense/, 2016. (Citato a pagina 15)

[15] K. Beck and C. Andres, *Extreme programming eXplained : embrace change*. Addison-Wesley, 1999. (Citato alle pagine 16 e 20)

[16] M. Stephens and D. Rosenberg, *Extreme Programming Refactored: The Case Against XP*, ser. Apresspod Series. Apress, 2003. (Citato a pagina 19)

[17] K. Beck, *Test-driven Development: By Example*, ser. Addison-Wesley signature series. Addison-Wesley, 2003. (Citato a pagina 20)

[18] Wikipedia, "Test driven development — wikipedia, l'enciclopedia libera," 2024. [Online]. Available: https://it.wikipedia.org/w/index.php?title=Test_driven_development&oldid=141471402 (Citato a pagina 21)

[19] T. Ohno and N. Bodek, *Toyota Production System: Beyond Large-Scale Production*. Productivity Press, 1988. (Citato a pagina 22)

[20] D. J. Anderson, *Kanban: Successful Evolutionary Change for Your Technology Business*. Blue Hole Press, 2010. (Citato alle pagine 22 e 23)

[21] D. Anderson and A. Carmichael, *Essential Kanban Condensed*, ser. [Essential Kanban]. Lean-Kanban University, 2015. (Citato a pagina 23)

[22] D. Vacanti and J. Coleman, "Kanban guides," https://kanbanguides.org/english/. (Citato a pagina 23)

[23] I. Sommerville, *Software Engineering*. Pearson Education, 2011. (Citato a pagina 27)

[24] M. Pezze and M. Young, *Software Testing and Analysis: Process, Principles and Techniques*. Wiley, 2008. (Citato a pagina 28)

Ringraziamenti

Ringrazio sentitamente tutti coloro che mi hanno sostenuto e che hanno creduto in me, dandomi la forza di raggiungere quest'importante traguardo per la mia vita.

Ringrazio tutti i professori che mi hanno accompagnato e guidato in questo percorso triennale.

Ringrazio il mio relatore, il professor Di Nucci, per la disponibilità e per l'enorme pazienza.

Un grazie speciale va a Roberta, la mia compagna, che mi ha supportato in ogni circostanza e che spero rimanga al mio fianco non solo lungo il mio percorso accademico, bensì per tutta la vita.