DStack

dstackgroup@gmail.com

Norme di Progetto

Informazioni sul documento

Nome documento	Norme di Progetto
Versione	v3.0.0
Data approvazione	2019-04-12
Responsabili	Enrico Trinco
Redattori	Harwinder Singh Alberto Schiabel
Verificatori	Federico Rispo
Stato	Approvato
Lista distribuzione	*DStack* *Prof. Tullio Vardanega* *Prof. Riccardo Cardin*
Uso	Interno

Sommario

Questo documento vuol definire le norme volte alla regolamentazione del gruppo *DStack* necessarie al processo di sviluppo del progetto Butterfly.

Diario delle Modifiche

Versione	Descrizione	Nominativo	Ruolo	Data
v3.0.0	**Approvazione per rilascio RQ**	Enrico Trinco	*Responsabile di Progetto*	2019-03-24
v2.1.0	**Verifica superata**	Federico Rispo	*Verificatore*	2019-03-24
v2.0.5	**Inserite figure 40, 41 e 42 e corretta figura 39**	Harwinder Singh	*Amministratore di Progetto*	2019-03-23
v2.0.4	**Inseriti diagrammi delle classi, di package, di sequenza e dei casi d'uso in §2.2.4.7, incremento §1.4, §3.4.1 e §3.4.2**	Harwinder Singh	*Amministratore di Progetto*	2019-03-23
v2.0.3	**Aggiunta nomeclatura diagrammi dei casi d'uso in §2.2.4.7 e aggiunta §2.2.4.9**	Harwinder Singh	*Amministratore di Progetto*	2019-03-23
v2.0.2	**Aggiunta nomeclatura diagrammi di package e di sequenza in §2.2.4.7**	Alberto Schiabel	*Amministratore di Progetto*	2019-03-22
v2.0.1	**Aggiunta nomeclatura diagrammi delle classi in §2.2.4.7**	Alberto Schiabel	*Amministratore di Progetto*	2019-03-21
v2.0.0	**Approvazione per rilascio RP**	Niccolò Vettorello	*Responsabile di Progetto*	2019-02-09
v1.1.0	**Verifica superata**	Elton Stafa	*Verificatore*	2019-02-08
v1.0.16	**Aggiunta §3.5.1.1 e §A**	Federico Rispo	*Amministratore di Progetto*	2019-02-07
v1.0.15	**Aggiunta §3.3.2.1, §3.3.2.2 e §3.3.2.**	Eleonora Signor	*Amministratore di Progetto*	2019-02-06

Versione	Descrizione	Nominativo	Ruolo	Data
v1.0.14	**Incremento §3.3.2 e §3.1.3.3**	Alberto Schiabel	*Amministratore di Progetto*	2019-02-05
v1.0.13	**Incremento §4.2**	Eleonora Signor	*Amministratore di Progetto*	2019-02-05
v1.0.12	**Incremento §3.3.5**	Federico Rispo	*Amministratore di Progetto*	2019-02-04
v1.0.11	**Incremento §2.3.2**	Eleonora Signor	*Amministratore di Progetto*	2019-02-04
v1.0.10	**Incremento §3.5**	Alberto Schiabel	*Amministratore di Progetto*	2019-02-04
v1.0.9	**Aggiunta §3.2.1**	Eleonora Signor	*Amministratore di Progetto*	2019-02-03
v1.0.8	**Aggiunta §2.2.3.2**	Federico Rispo	*Amministratore di Progetto*	2019-02-03
v1.0.7	**Aggiunta §2.1.4 e §2.1.3.2 ed incremento §2.2.3**	Eleonora Signor	*Amministratore di Progetto*	2019-02-03
v1.0.6	**Aggiunti riferimenti e citazione parti d'interesse in §1.3 e §1.4**	Alberto Schiabel	*Amministratore di Progetto*	2019-02-03
v1.0.5	**Icremento §4.3.2**	Eleonora Signor	*Amministratore di Progetto*	2019-02-03
v1.0.4	**Rivisitazione struttura indice §4**	Alberto Schiabel	*Amministratore di Progetto*	2019-02-02
v1.0.3	**Aggiunta §2.2.5 e §2.2.6**	Federico Rispo	*Amministratore di Progetto*	2019-02-02
v1.0.2	**Aggiunta §2.2**	Alberto Schiabel	*Amministratore di Progetto*	2019-02-02
v1.0.1	**Cancellazione nota §2.2.3.1 secondo esito RR**	Eleonora Signor	*Amministratore di Progetto*	2019-02-02
v1.0.0	**Approvazione per il rilascio RR**	Alberto Schiabel	*Responsabile di Progetto*	2019-01-13
v0.2.0	**Verifica superata**	Federico Rispo	*Verificatore*	2019-01-12

Versione	Descrizione	Nominativo	Ruolo	Data
v0.1.3	**Aggiunta §4.2**	Niccolò Vettorello	*Amministratore di Progetto*	2019-01-10
v0.1.2	**Consolidamento §2**	Eleonora Signor	*Amministratore di Progetto*	2019-01-08
v0.1.1	**Consolidamento §3**	Enrico Trinco	*Amministratore di Progetto*	2019-01-07
v0.1.0	**Verifica superata**	Federico Rispo	*Verificatore*	2018-11-30
v0.0.4	**Stesura §4**	Niccolò Vettorello	*Amministratore di Progetto*	2018-11-29
v0.0.3	**Stesura §2**	Eleonora Signor	*Amministratore di Progetto*	2018-11-28
v0.0.2	**Stesura §1 e §3**	Enrico Trinco	*Amministratore di Progetto*	2018-11-27
v0.0.1	**Creazione scheletro del documento**	Eleonora Signor	*Amministratore di Progetto*	2018-11-26

Indice

Elenco delle figure

Elenco delle tabelle

1 Introduzione

Lo scopo del documento è fissare tutte le regole e le procedure fondamentali per assicurare al gruppo *DStack* un modo di lavorare comune ed *efficiente*$_{\mathbf{G}}$, migliorando la coerenza e l'organicità tra i file prodotti.
Durante l'intero svolgimento del progetto tutti i membri del gruppo sono obbligati a visionare ed a sottostare alle norme ivi descritte.
Il presente documento è incompleto perché redatto secondo una filosofia incrementale: prima di iniziare un nuova attività di progetto (non ancora normata) il gruppo si accorderà su quali regole e procedure sono necessarie al suo svolgimento. In questo modo vengono decise prima le norme più urgenti, ed in un secondo momento quelle che saranno necessarie per le attività successive.
Le norme già presenti potranno subire variazioni quali aggiunte, rimozioni o modifiche che dovranno essere comunicate[1] ad ogni componente del gruppo dal *Responsabile di Progetto*.

1.1 Scopo del prodotto

Butterfly nasce dall'esigenza di uniformare e accentrare la gestione delle segnalazioni generate a partire da sistemi di terze parti, quali Redmine, GitLab e SonarQube. Questi strumenti sono parte integrante dei processi gestionali, di versionamento e di Continuous Integration dell'azienda committente. La maggior parte di essi fornisce già dei meccanismi di notifica ed inoltro delle possibili segnalazioni, sono configurabili e accessibili da dashboard molto diverse tra loro, di difficile interazione e anche con limitazione di accessibilità. Inoltre, in caso di segnalazioni di bug in ambienti di produzione è fondamentale assicurarsi che gli sviluppatori in grado di risolvere il problema siano segnalati tempestivamente, senza aspettare che loro accedano a qualche dashboard specifica. Il gruppo *DStack* si propone quindi di sviluppare una rete di soluzioni che offrano un'interfaccia condivisa, estendibile per gestire le segnalazioni relative alla pipeline di sviluppo software di *Imola Informatica S.P.A.*. Questa interfaccia deve inoltre permettere una configurazione automatica e personalizzabile di tali segnalazioni.

1.2 Glossario

All'interno del documento sono presenti termini che possono presentare significati ambigui o incongruenti a seconda del contesto. Per evitare questo tipo di problema viene allegato glossario nel file *Glossario v3.0.0*, che contiene tali termini e la loro spiegazione. Nella seguente documentazione viene indicata in corsivo e seguita da una "G" a pedice solo la prima occorrenza dei termini presenti nel glossario, per favorire maggiore chiarezza ed evitare ridondanza.

[1] Le norme relative alle comunicazioni interne sono definite in §4.1.2.1

1.3 Riferimenti normativi

1.4 Riferimenti informativi

- «The Twelve-Factor App»:
  https://12factor.net;
  - "Introduction", come riferimento per la definizione generale di cosa è un'app a microservizi;
  - "The Twelve Factors", come riferimento alla definizione delle stesse.
- ArteLatex:
  http://www.lorenzopantieri.net/LaTeX_files/ArteLaTeX.pdf;
  - Usato come riferimento per imparare la semantica e la sintassi del linguaggio *LaTeX*$_{\mathbf{G}}$.
- Confronto tra *npm*$_{\mathbf{G}}$ e *yarn*$_{\mathbf{G}}$: https://stackshare.io/stackups/npm-vs-yarn;
  - Usato per stabilire qualche package manager usare per i servizi scritti in *Node.js*$_{\mathbf{G}}$.
- Documentazione *Git*$_{\mathbf{G}}$:
  https://git-scm.com/doc;
  - "Setup and Config" con riferimento all'installazione e configurazione iniziale;
  - "Getting and Creating Projects" con riferimento ai comandi init e clone;
  - "Basic Snapshotting" con riferimento ai comandi add, diff e commit;
  - "Branching and Merging" con riferimento ai comandi branch, checkout e merge;
  - "Sharing and Updating Projects" con riferimento ai comandi fetch, pull e push;
  - "Inspection and Comparison" con riferimento ai comandi show, log e diff.
- *Bad Smalls Code*$_{\mathbf{G}}$ per il *Code refactoring*$_{\mathbf{G}}$:
  https://refactoring.guru/refactoring/smells;
  - Usato come riferimento per apprendere e comprendere l'utilità del code refactoring.
- Standard UML 2.0:
  - https://www.math.unipd.it/~tullio/IS-1/2018/Dispense/E03b.pdf;
    - Usato come riferimento i diagrammi delle classi in §2.2.4.7.
  - https://www.math.unipd.it/~tullio/IS-1/2018/Dispense/E04b.pdf;
    - Usato come riferimento i diagrammi di attività in §2.2.4.7.
  - https://www.math.unipd.it/~tullio/IS-1/2018/Dispense/E05a.pdf;
    - Usato come riferimento i diagrammi dei package in §2.2.4.7.
  - https://www.math.unipd.it/~tullio/IS-1/2018/Dispense/E04a.pdf;

    * Usato come riferimento i diagrammi di sequenza in §2.2.4.7.
- https://www.math.unipd.it/~tullio/IS-1/2018/Dispense/E05b.pdf;
    * Usato come riferimento i diagrammi dei casi d'uso in §2.2.4.7.

2 Processi primari

2.1 Fornitura

2.1.1 Scopo

Il processo di Fornitura descrive i tasks e le attività svolte dal gruppo *DStack*, che funge da fornitore. Sono incluse anche tutte le norme che il gruppo intende rispettare candidandosi all'azienda proponente *Imola Informatica S.P.A.* ed ai committenti *Prof. Tullio Vardanega* e *Prof. Riccardo Cardin* come realizzatore del prodotto Butterfly.

Le attività che verranno normate all'interno delle seguenti sottosezioni riguardano:

- analisi e redazione dello *Studio di Fattibilità*, mediante l'individuazione dei rischi e delle criticità inerenti alla richiesta d'appalto;
- definizione di un accordo contrattuale, mediante il quale vengono gestiti i rapporti con il proponente, la consegna e la manutenzione del prodotto finale;
- identificazione dei rischi, in cui il fornitore pensa di poter incorrere durante le attività di progetto.

2.1.2 Rapporti con il proponente *Imola Informatica S.P.A.*

Il gruppo si impegna ad instaurare con l'azienda *Imola Informatica S.P.A.*, in particolare con le figure di Luca Cappelletti e Davide Zanetti, un dialogo continuo[2] ed un rapporto collaborativo al fine di discutere in dettaglio i requisiti richiesti ed arrivare ad un accordo finale che sancisca il contratto.
I rapporti tra *DStack* ed *Imola Informatica S.P.A.* verteranno sui seguenti punti:

- capire e definire le necessità del proponente;
- individuare come soddisfare i bisogni e i vincoli del proponente;
- individuare vincoli di progetto collegati all'esecuzione dei processi;
- stabilire i punti chiavi riguardanti la realizzazione del prodotto;
- compiere scelte riguardanti le migliori tecnologie da impiegare per la realizzazione del prodotto;
- illustrare i progressi compiuti dal team per chiedere feedback migliorativi;
- individuare eventuali opportunità di miglioramento dell'idea di prodotto;
- valutare le misure di qualifica del prodotto.

[2] Le modalità di comunicazione sono specificate nella sezione §4.1.2.2

2.1.3 Materiale fornito

Per garantire una totale trasparenza e chiarezza durante la realizzazione del progetto, saranno consegnati all'azienda proponente *Imola Informatica S.P.A.* ed ai committenti *Prof. Tullio Vardanega* e *Prof. Riccardo Cardin* i seguenti documenti:

- ***Analisi dei Requisiti v3.0.0***: contiene l'analisi dei casi d'uso e dei requisiti diretti ed indiretti, espliciti e non, mirando ad evitare ogni tipo di ambiguità appartenente al capitolato e a definire nel dettaglio quali saranno le funzionalità offerte dal prodotto;
- ***Piano di Progetto 4.0.0***: contiene la pianificazione preventiva dei tempi e delle attività, analisi dei rischi e consuntivo al termine di ogni attività, oltre a data e costi previsti del prodotto finale;
- ***Piano di Qualifica v3.0.0***: definisce le modalità di verifica e validazione del prodotto per garantire una qualità dei processi adeguata alle aspettative del proponente;
- ***Proof of Concept*$_{\mathbf{G}}$ e *Technology Baseline*$_{\mathbf{G}}$**: nell'ambito dell'attività di progettazione architetturale, definiscono una panoramica ad alto livello dell'applicazione che il gruppo *DStack* realizzerà e delle tecnologie impiegate;
- ***Product Baseline*$_{\mathbf{G}}$**: nell'ambito dell'attività di progettazione di dettaglio, definisce l'insieme di classi, metodi, attributi e scelte implementative a livello tecnico.

Tali documenti sono un metodo di tracciamento per le attività di Analisi, Pianificazione, Verifica, Validazione e Controllo di qualità svolti durante l'intero progetto. Alla documentazione viene allegata una ***Lettera di Presentazione***.

Come da richiesta del proponente il gruppo *DStack* si impegna, entro i termini definiti dalla *Lettera di Presentazione*, a consegnare un prodotto software che soddisfi almeno tutti i requisiti obbligatori del *Capitolato d'Appalto*. Effettuata la consegna, la manutenzione del prodotto non verrà presa in carico dal gruppo *DStack*, in quanto verrà sciolto e conseguentemente cesseranno di esistere i rapporti con il proponente.

Il prodotto software finale idoneo ad accettazione verrà consegnato su un supporto CD-ROM/ DVD con allegato l'Utilità di installazione, Manuali d'uso ed eventualmente di collaudo, con tutti i sorgenti completi e l'Utilità di compilazione.

2.1.4 Studio di Fattibilità

Una volta che il gruppo *DStack* ha scelto il capitolato per il quale proporsi come fornitori, gli analisti condurranno un'ulteriore e approfondita attività di analisi dei rischi e delle criticità, quindi dovranno redigere un documento chiamato *Studio di Fattibilità* contenente le motivazioni che hanno portato il gruppo a proporsi come fornitore per il prodotto indicato nel capitolato scelto.

Per ogni capitolato presentato sarà riportato:

- **Descrizione del capitolato**: conterrà una sintesi del prodotto richiesto dal capitolato, secondo quanto compreso durante la presentazione e quanto estratto dal testo di capitolato;

- **Dominio Applicativo**: valuterà il contesto dove il prodotto software dovrà operare e le conoscenze richieste allo sviluppo;
- **Dominio Tecnologico**: conterrà la lista di tecnologie richieste dal capitolato per lo sviluppo del prodotto;
- **Analisi degli aspetti negativi e positivi**: verranno descritti i fattori positivi e le criticità che il gruppo ha riscontrato durante l'analisi preliminare svolta sfruttando le conoscenze singole di ogni membro ed effettuando ricerche web e cartacee;
- **Identificazione dei rischi**: conterrà l'analisi dei punti critici della realizzazione individuati, come la mancanza di conoscenze sufficienti o difficoltà di comprensione degli scopi del prodotto;
- **Conclusioni**: verrà riportata la scelta di accettazione o rifiuto del capitolato, tenendo conto non esclusivamente dei fattori sopra espressi; ma anche dell'interesse del gruppo verso la realizzazione del prodotto.

2.1.5 Preparazione al collaudo del prodotto

Una volta che il gruppo *DStack* abbia sviluppato un prodotto software sufficientemente stabile e completo, i membri di *DStack* si impegneranno ad instaurare un rapporto il più possibile costante con i referenti della proponente *Imola Informatica S.P.A.* per offrire loro prototipi di prodotto e raccogliere resoconti migliorativi.
Per riuscire a superare il collaudo, il prodotto finale deve superare un periodo di test, che vede lo svolgimento di:

- Test d'unità;
- Test di regressione;
- Test di integrazione;
- Test di sistema.

Nel momento in cui il software supera positivamente tutti i test sovra esposti, si ottiene la certezza di aver realizzato un prodotto affidabile, corretto e completo. Conseguentemente durante il collaudo è dimostrabile, da parte di *DStack* al committente che:

- I test definiti all'interno del *Piano di Qualifica v3.0.0* sono stati eseguiti e hanno ottenuto un esito conforme alle metriche dichiarate;
- I requisiti obbligatori definiti all'interno del *Analisi dei Requisiti v3.0.0* sono stati tutti implementati al meglio, rispettando le aspettative e quanto promesso al proponente;
- Nel caso in cui sia stato possibile implementare alcuni dei requisiti desiderabili e facoltativi, che essi siano stati implementati al meglio, andando oltre alle aspettative minime del proponente.

I test sovra esposti vengono definiti nel dettaglio in §3.5.4.

2.1.6 Collaudo del prodotto e consegna

DStack offirà all'azienda proponente *Imola Informatica S.P.A.*, e ai committenti *Prof. Tullio Vardanega* e *Prof. Riccardo Cardin* quanto segue:

- **Codice Sorgente**;
- **Documentazione di prodotto**, come indicato in §2.1.3, a cui si aggiungeranno:
  - *Glossario v3.0.0*: per eliminare l'ambiguità dei termini usati e facilitare la comprensione degli altri documenti;
  - *Manuale Utente* (da redarre): per guidare l'utente finale nel processo di installazione e utilizzo delle funzionalità del prodotto;
  - *Manuale dello Sviluppatore* (da redarre): per agevolare la contribuzione al progetto da parte di sviluppatori diversi da quelli originali.

2.1.7 Individuazione dei rischi

Ogni rischio verrà individuato e descritto da:

- Codice e nome;
- Descrizione;
- Occorrenza;
- Pericolosità;
- Modalità di rilevamento;
- Contromisure;
- Eventuale periodo di riscontro;
- Eventuale mitigazione adottata per far fronte al rischio.

Il *Codice* sarà rappresentato con:

D[Tipologia][Codice]

- **D**: carattere identificativo di rischio, valore statico di "Danger";
- **Tipologia**: individua la categoria di appartenenza del rischio:
  - **T**: rischi legati ai mezzi tecnologici;
  - **G**: rischi legati ai membri del gruppo;
  - **O**: rischi legati all'organizzazione del lavoro;
  - **R**: rischi legati ai requisiti.
- **Codice**: codice numerico progressivo che identifica univocamente il rischio.

2.2 Sviluppo

2.2.1 Scopo

Il processo di sviluppo descrive le attività e i compiti svolti durante la realizzazione del prodotto software finale.

Le attività che verranno normate all'interno delle seguenti sottosezioni riguardano:

- Analisi dei requisiti del prodotto software;
- Progettazione;
- Codifica del software.
- Strumenti impiegati durante il processo di sviluppo.

2.2.2 Obiettivi

Per implementare correttamente il processo di sviluppo, *DStack* si pone i seguenti obiettivi:

- sviluppare un prodotto software conforme alle richieste del proponente;
- superare i test di verifica definiti;
- realizzare un prodotto finale che soddisfi i test di validazione;
- definire e fissare i vincoli tecnologici;
- definire e fissare i vincoli di design del prodotto;
- individuare gli obiettivi di sviluppo.

2.2.3 Analisi dei Requisiti

L'*Analisi dei Requisiti* verrà redatta dagli *Analisti* che avranno il compito di individuare quali sono i requisiti sia diretti che indiretti, impliciti ed espliciti che il proponente richiede per il proprio prodotto.

I requisiti saranno individuati mediante un *approccio investigativo*$_{\mathbf{G}}$ utilizzando le seguenti fonti:

- ***Capitolato d'Appalto***: il requisito deriva dall'analisi del documento del *Capitolato d'Appalto* presentato da *Imola Informatica S.P.A.*;
- ***Verbali Interni***: il requisito è emerso dalle riunioni interne effettuate dagli *Analisti* del gruppo;
- ***Verbali Esterni***: il requisito è emerso dopo contatti e discussioni con i responsabili referenti lato aziendale Luca Cappelletti e Davide Zanetti;
- **Casi d'uso**: il requisito deriva dallo studio di uno o più casi d'uso.

L'Analisi dei Requisiti deve essere strutturata come di seguito esposto.

- **Descrizione**: la sezione deve contenere:

  - Descrizione generale del prodotto;
  - Descrizione delle principali componenti che interagiscono all'interno del sistema;
  - Descrizione della piattaforma d'esecuzione;
  - Trattazione dei vincoli di progettazione individuati.

- **Casi d'uso**: la sezione ha il compito di definire quali sono gli attori che interagiscono con ogni componente del sistema e le interazioni che si scatenano tra il sistema, gli attori e gli elementi esterni al sistema.
  Ogni interazione viene esplicitata formalmente mediante linguaggio naturale, come normato all'interno del suddetto documento in §2.2.3.2, corredata da relativo diagramma standardardizzato UML2.0;

- **Requisiti**: la sezione deve contenere in forma tabellare:

  - Il tracciamento dei requisiti, in cui devono essere dichiarati tutti i requisiti obbligatori, desiderabili, opzionali individuati nel sistema durante l'attività di analisi. Vengono definiti rispettando quanto normato all'interno del suddetto documento nella sezione §2.2.3.1;
  - Tracciamento fonte-requisiti, requisiti-fonte, necessari per un riferimento chiaro e formale dell'origine di ogni requisito dichiarato.

2.2.3.1 Classificazione dei requisiti

La convenzione fissata per la rappresentazione dei requisiti è la seguente:

R[Tipologia][Importanza][Codice]

- **Tipologia** è un indicatore che assume uno fra i seguenti valori:

  - **F**: indica un requisito *Funzionale*, ovvero descrive servizi o funzioni offerti dal sistema;
  - **V**: indica un requisito *di Vincolo*, ovvero descrive vincoli sui servizi offerti dal sistema;
  - **P**: indica un requisito *Prestazionale*, ovvero descrive i vincoli sulle prestazioni che bisogna soddisfare, come ad esempio il numero di informazioni che devono essere manipolate in un certo periodo di tempo;
  - **Q**: indica un requisito di *Qualità*, ovvero descrive i vincoli di qualità da realizzare quali manutenibilità, sicurezza, portabilità, disponibilità, ecc..[3]

- **Importanza** è un indicatore che assume uno fra i seguenti valori:

  - **O**: indica un requisito *Obbligatorio*, ovvero irrinunciabile per garantire il funzionamento delle attività di base del sistema;
  - **D**: indica un requisito *Desiderabile*, ovvero la cui presenza non è vincolata al funzionamento del sistema; ma se implementata darebbe una maggiore completezza. Fa parte della categoria di requisiti negoziabili tra *DStack* ed *Imola Informatica S.P.A.*;

[3] Per una descrizione dettagliata sui vincoli di qualità presi in considerazione si veda il documento *Piano di Qualifica v3.0.0*

  - **F**: indica un requisito *Facoltativo*, ovvero offre delle funzionalità aggiuntive la cui implementazione non è da ritenersi vincolante alla riuscita del progetto.

- **Codice**: è un numero progressivo espresso in forma gerarchica. Composto da:

  [IDBase](.[IDSottoCaso])*

  - **IDBase**: codice che, combinato con la *Tipologia*, identifica il requisito generale;
  - **IDSottoCaso**: codice progressivo opzionale che identifica gli eventuali sottocasi del requisito.

Un requisito deve essere sempre associato a delle informazioni aggiuntive:

- **Descrizione**: breve descrizione sintetica e non ambigua dello scopo del requisito;
- **Fonte**: indica dove si è individuato il requisito tra: *Capitolato d'Appalto*, *Verbali Interni*, *Verbali Esterni* e casi d'uso.

ID Requisito	Descrizione	Fonte
RVO2	Visualizzazione pagina di Registrazione entro 2 secondi dalla sua richiesta	Verbali Interni UC2 UC2.2

Tabella 3: Esempio di Requisito

2.2.3.2 Classificazione dei casi d'uso

La convenzione fissata per la rappresentazione dei casi d'uso è la seguente:

UC[IDBase](.[IDSottoCaso])*

L'identificativo è scomposto in:

- **UC**: acronimo di "use case";
- **IDBase**: codice che identifica il caso d'uso generico;
- **IDSottoCaso**: codice progressivo opzionale che identifica gli eventuali sottocasi del caso d'uso.

Ogni caso d'uso è composto dalle seguenti parti:

- **Identificativo**: composto dalla numerazione appena descritta sopra seguita da una stringa testuale che rappresenta la funzionalità che si vuole modellare;
- **Descrizione grafica**: utilizzando il formalismo UML 2.0[4] per creare diagrammi di casi d'uso che facilitino la comprensione del sistema in esame, degli attori coinvolti e delle azioni che posso eseguire;

[4]Lo strumento da utilizzare è specificato alla sezione §3.1.4.2.2

- **Descrizione**: contiene una breve descrizione del caso d'uso;
- **Attori**: l'attore è tutto ciò che è esterno al sistema e con il quale interagisce. Un'attore si suddivide in due categorie:
  - **Attore primario**: che interagisce direttamente con il sistema;
  - **Attore secondario**: entità esterna che interagisce con il sistema; ma con l'obiettivo di far raggiungere all'attore principale il suo scopo.
- ***Precondizione*$_G$**: descrive lo stato del sistema prima del verificarsi delle azioni espresse nel caso d'uso;
- ***Postcondizione*$_G$**: descrive lo stato del sistema una volta che si sono svolte le azioni stabilite nel caso d'uso;
- **Scenario principale**: rappresenta in modo puntale, per passi mediante elenco numerato, le azioni del caso d'uso;
- **Estensioni** (Opzionali): ognuno di essi viene eseguito solo quando si avvera una determinata condizione. Se la condizione viene verificata l'esecuzione del caso d'uso, a cui e' collegato, viene interrotta;
- **Inclusioni** (Opzionali): quando due casi d'uso sono tra loro collegati. Viene eseguito sempre e solo dopo che il caso d'uso, a cui è collegato, sia stato eseguito completamente;
- **Generalizzazioni** (Opzionali): specializzazioni di un caso d'uso. Ogni specializzazione è mutualmente accessibile.

Esempio di Caso d'uso:

UC5: Login

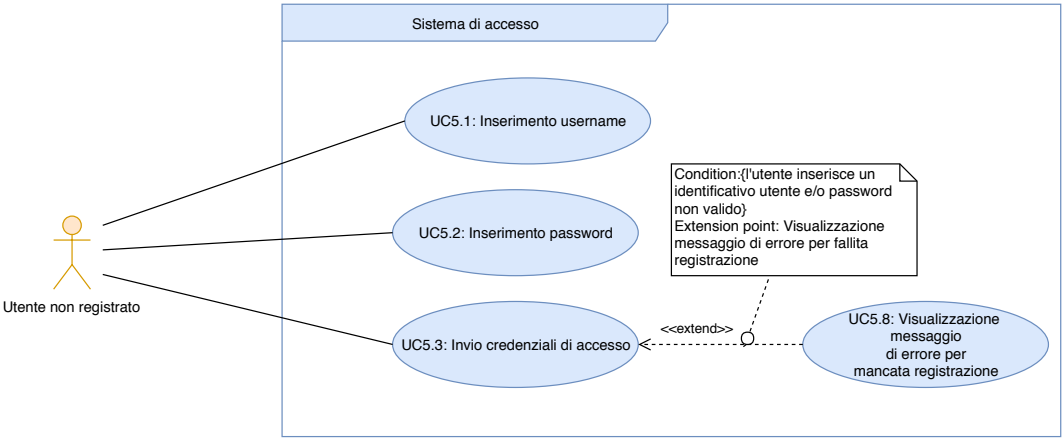


Figura 1: Esempio caso d'uso

Descrizione: l'utente non autenticato effettua il login nel sistema;
Attori: Utente non registrato;
Precondizione: l'utente non è autenticato, vuole effettuare il login;
Postcondizione: l'attore è autenticato all'interno del sistema;
Scenario principale:

1. Inserimento username (UC5.1);
2. Inserimento password (UC5.2);
3. Invio credenziali di accesso(UC5.3).

Estensioni:

1. Quando l'utente in input inserisce una password e/o un username non valido:
   (a) Non viene effettuata l'operazione di login;
   (b) Viene visualizzato un messaggio di errore di mancata registrazione utente. (UC5.8).

2.2.3.3 Tracciamento dei requisiti e casi d'uso

Sarà necessario tracciare automaticamente ogni requisito e caso d'uso individuato durante l'attività di *Analisi dei Requisiti* in modo da facilitare le attività successive. Per svolgere questo compito il gruppo ha scelto di affidarsi allo strumento *PragmaDB*$_{\mathbf{G}}$[5], un software per il tracciamento dei casi d'uso e dei requisiti che permette di esportare tabelle in LaTeX da inserire nel documento di *Analisi dei Requisiti v3.0.0*.

[5]Vedi §2.2.6, Strumenti primari

Sarà responsabilità dell'*Amministratore di Progetto* configurarlo e renderlo operativo in modo che tutti i membri del gruppo possano utilizzarlo.

2.2.3.4 Qualità dei requisiti

La specifica dei requisiti dovrà rispettare le seguenti qualità:

- **Completezza**: ogni funzionalità richiesta dal prodotto software e il suo comportamento in risposta agli input dati deve essere dettagliatamente specificato;
- **Consistenza**: non vi devono essere requisiti in contraddizione tra loro;
- **Correttezza**: tutti i requisiti specificati devono essere veramente necessari e richiesti agli utenti finali;
- **Univocità**: per evitare situazioni di ambiguità, ogni requisito deve essere identificato da un codice formale univoco;
- **Verificabilità**: deve essere possibile verificare che il sistema realizzi ogni requisito individuato;
- **Modificabilità**: la struttura dei requisiti deve poter evolvere nel tempo preservando caratteristiche come consistenza e completezza;
- **Tracciabilità**: deve essere chiara quale sia l'origine dei requisiti, e tale origine deve poter essere referenziata in futuro.

2.2.4 Progettazione

2.2.4.1 Scopo

La Progettazione viene effettuata dai *Progettisti* allo scopo di individuare la soluzione migliore adottabile che soddisfi i requisiti degli *stakeholders*$_{\mathbf{G}}$ esposti nel documento *Analisi dei Requisiti v3.0.0*.
L'attività di Progettazione è mirata quindi a realizzare l'architettura del sistema, attuata in prima istanza dal *Proof of Concept* della Technology Baseline, e approfondita e descritta nel documento tecnico allegato alla Product Baseline. La fase di progettazione permette di:

- garantire la qualità del prodotto sviluppato, seguendo il principio di **correttezza per costruzione**, affrontando i problemi con un *approccio sistematico*$_{\mathbf{G}}$;
- organizzare e suddividere i compiti di implementazione;
- ottimizzare i tempi e il numero delle risorse assegnate.

2.2.4.2 Descrizione

Prima di procedere alla realizzazione architetturale, il gruppo *DStack* dovrà definire le tecnologie da utilizzare, approfondendo e studiando gli aspetti positivi e le eventuali criticità, e producendo una bozza dimostrabile del prodotto chiamata *Proof of Concept* che rifletta quest'approfondimento. Durante le attività di progettazione il gruppo dovrà realizzare l'architettura del sistema rispettando ogni vincolo stabilito con il proponente *Imola Informatica S.P.A.*.

2.2.4.3 Qualità dell'architettura

L'architettura intesa come logica ad alto livello del prodotto deve essere di buona qualità. Ciò è possibile solo se sono presenti tutte le seguenti caratteristiche:

- **Semplicità**: l'architettura dovrà attenersi al principio *KISS*$_{\mathbf{G}}$, prediligendo le soluzioni più semplici, facilmente comprensibili a tutti e necessarie;
- **Sufficienza**: l'architettura è sufficiente ai compiti, ovvero soddisfa tutti i requisiti presenti nell'*Analisi dei Requisiti v3.0.0* ed è in grado di adattarsi al loro cambiamento;
- **Comprensibilità**: l'architettura ha uno stile. Questo deve essere riconosciuto in modo chiaro e senza ambiguità dagli stakeholders;
- **Modularità**: l'architettura è divisa in componenti ben distinte e senza sovrapposizioni di funzionalità, per garantire la separazione degli interessi (*separation of concerns*$_{\mathbf{G}}$) e per rendere più semplice l'esecuzione dei compiti;
- **Robustezza**: l'architettura dovrà rimanere operativa anche di fronte a situazioni erronee impreviste;
- **Flessibilità**: l'architettura deve permettere la sua evoluzione nel tempo (esempio cambiamento dei requisiti). Fondamentale per fare questo é la manutenzione del prodotto;
- **Riusabilità**: le singole componenti dell'architettura devono essere sviluppate in modo da poter essere riusate da altre parti del sistema;
- **Efficienza**: l'architettura raggiunge il suo obiettivo con il minor dispendio di risorse (quali cpu, memoria, tempo) possibili;
- **Affidabilità**: quando l'architettura viene usata assolve ai suo compiti;
- **Disponibilità**: quando un'architettura presenta un ridotto tempo di fermo inteso come manutenzione;
- **Sicurezza rispetto ai malfunzionamenti**: se l'architettura riscontra dei malfunzionamenti questi non devono recare danno a chi la sta usando;
- **Sicurezza rispetto alle intrusioni**: l'architettura si presenta invulnerabile rispetto ad intrusioni da parte di esterni;
- **Incapsulazione**: ogni modulo dovrà essere una *"black box"*$_{\mathbf{G}}$ rispetto all'utente esterno in modo da diminuire le dipendenze ed accrescere la mantenibilità (*Information Hiding*$_{\mathbf{G}}$);
- **Coesione**: le componenti che hanno il medesimo obiettivo vengono aggregate nello stesso modulo;
- **Basso accoppiamento**: in un architettura è necessario che componenti distinte dipendano poco o niente, le une con le altre, in modo da poterne permettere facilmente la manutentibilità e il *code refactoring*$_{\mathbf{G}}$ futuri.

2.2.4.4 Periodi di progettazione

Le attività di progetto da svolgere vengono suddivise in base agli obiettivi da raggiungere e fatte rientrare all'interno di appositi periodi temporali, come definito nel *Piano di Progetto 4.0.0*. I *Programmatori* entrano in campo durante il periodo di Progettazione Architetturale e di dettaglio.
Nella **Progettazione architetturale** devono essere definite le componenti del sistema, nello specifico i task che devono essere svolti per portare a termine positivamente il periodo:

- Individuazione delle componenti coinvolte nel sistema;
- Definizione di ruoli e di responsabilità per ogni componente;
- Definizione delle interazioni tra componenti;
- Accertamento che ogni requisito individuato nel documento *Analisi dei Requisiti v3.0.0* sia soddisfatto e viceversa;
- Creazione e documentazione, all'interno del *Piano di Qualifica v3.0.0* dei test d'integrazione, con lo scopo di verificare l'adeguato comportamento di più componenti, del sistema, integrate assieme.

Nella **Progettazione di dettaglio** viene esteso quanto svolto durante la Progettazione architetturale:

- Suddivisione delle componenti individuate in unità;
- Definizione dei ruoli che coinvolgono le unità;
- Definizione delle interazioni che coinvolgono le unità;
- Definizione dell'unità come insieme di moduli;
- Definizione dei ruoli per ogni modulo;
- Definizione degli strumenti di verifica per le unità;
-
- Definizione degli strumenti di verifica per i moduli.

2.2.4.5 *Design Pattern*$_G$

A seguito dell'attività di Analisi, i *Progettisti* avranno il compito di risolvere i problemi ricorrenti individuando Design Pattern adatti e riportandoli nel documento di *Technology Baseline*. Ogni Design Pattern dovrà essere completo di:

- diagramma che lo illustri;
- descrizione testuale del suo funzionamento;
- descrizione dell'utilità di tale Design Pattern all'interno dell'architettura realizzata.

2.2.4.6 Test

I *Progettisti* avranno inoltre il compito di definire opportuni test, che dovranno essere accompagnati da eventuali classi utili ad individuare errori e anomalie. I test progettati dovranno rispettare le convenzioni di nomenclatura specificate in §3.6.3.

2.2.4.7 Diagrammi

Con lo scopo di rendere chiare le scelte progettuali adottate e ridurre le eventuali ambiguità, *DStack* ha deciso di usare la notazione offerta da UML 2.0. Ne fanno parte:

- **Diagrammi delle attività**: assolvono al compito di illustrare graficamente il flusso di operazioni che definiscono un'attività attraverso una logica procedurale. Vengono impiegati per descrivere, ad esempio, la logica di un algoritmo;
- **Diagrammi delle classi**: assolvono al compito di illustrare graficamente una collezione di elementi di un modello. Vengono definite le classi, i tipi, i metodi, gli attributi e le relazioni che vi intercorrono, astraendosi da un linguaggio di programmazione specifico;
- **Diagrammi dei *package*$_{\mathbf{G}}$**: assolvono al compito di illustrare graficamente raggruppamenti di classi in unità, devono venire riusate assieme e condividono la medesima causa di cambiamento;
- **Diagrammi di sequenza**: assolvono al compito di illustrare graficamente tutte le iterazioni che avvengono tra oggetti/classi che devono implementare uno specifico comportamento. Le interazioni sono rappresentate per mezzo di scelte definite come una sequenza di invocazioni tra metodi;
- **Diagrammi dei casi d'uso**: assolvono al compito di rappresentare graficamente, nel dettaglio, le funzionalità del sistema offerte al proponente.

In caso di bisogno si può ricorrere all'uso di diagrammi delle componenti, di macchine a stato o di *deployment*$_{\mathbf{G}}$.

2.2.4.7.1 Diagrammi delle attività

Un diagramma delle attività permette di definire le caratteristiche dinamiche di un sistema. Risulta utile sia nel descrivere attività e concetti legati all'implementazione dei metodi, sia per descrivere livelli di astrazione più alti, ad esempio l'analisi dei requisiti.
Un diagramma descrive un processo, ovvero una sequenza di elementi azioni/attività collegate tra loro da frecce.
Il formalismo utilizzato è il seguente:

- **Nodo iniziale**: ogni diagramma deve sempre essere composto da un nodo iniziale, rappresentato da un pallino pieno. È il punto iniziale dove inizia l'esecuzione e viene generato un *token*$_{\mathbf{G}}$;

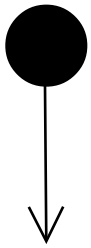


Figura 2: Nodo iniziale diagramma delle attività

- **Nodo finale di flusso**: è rappresentato da una X contenuta in un cerchio vuoto. Indica un punto in cui uno dei rami dell'esecuzione termina. Questo non indica la conclusione dell'esecuzione, che può comunque continuare su altri rami paralleli a quello terminato. Il token viene consumato;

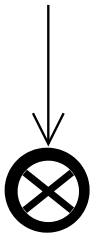


Figura 3: Nodo finale di flusso diagramma delle attività

- **Nodo finale**: ad ogni *nodo iniziale* ne deve corrispondere uno finale, è rappresentato da due cerchi concentrici, di cui il più interno pieno e quello esterno vuoto. È il punto in cui viene consumato un token;

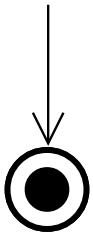


Figura 4: Nodo finale diagramma delle attività

- **Activity**: rappresenta l'attività, un rettangolo smussato che contiene la descrizione della stessa. La descrizione assume un ruolo di carattere generale, deve essere il più breve e concisa possibile, composta da parole chiavi, la prima delle quali inizia con la lettera maiuscola. In questo caso viene consumato e prodotto un token;

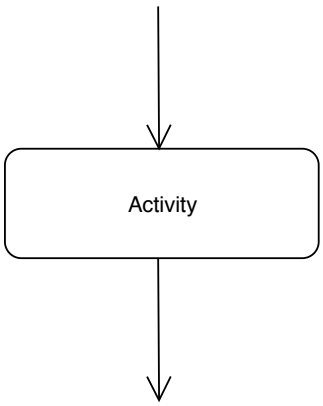


Figura 5: Activity diagramma delle attività

- **Subactivity**: rappresenta una sottoattività, un rettangolo smussato che contiene la descrizione della stessa. Viene usato quando in una singola attività andrebbero modellate più attività. Per impedire il nascere di diagrammi di dimensione troppo elevata, che comporterebbero una difficile gestione, l'attività principale, l'*Activity*, viene marcata da un piccolo tridente, in basso a destra, usato come riferimento ad una sottattività il cui diagramma viene descritto separatamente. Ogni sottoattività (*Subactivity*) è composto da un input ed un output e viene contornata da un *Activity Frame*;

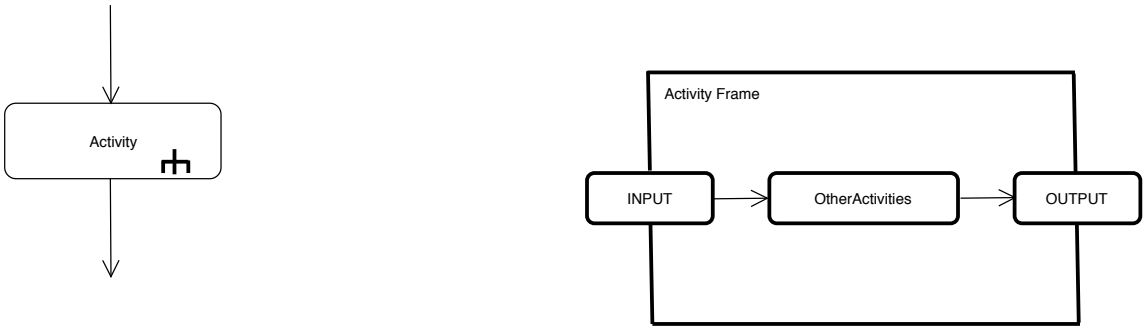


Figura 6: Subactivity diagramma delle attività

- **Fork**: è rappresentato da una sola freccia entrante e da due o più frecce uscenti, tra le quali viene posta una lunga linea orizzontale o verticale, a seconda dell'orientazione. Quest'ultima è il punto dove l'attività si parallelizza senza alcun vincolo di esecuzione temporale. Vengono generati tanti token quanti sono le frecce uscenti e ne viene consumato solo uno;
- **Join**: è rappresentato da più frecce entranti e da una freccia uscente, tra le quali viene posta una lunga linea orizzontale o verticale, a seconda dell'orientazione. Quest'ultima è il punto dove avviene la sincronizzazione tra processi paralleli. Viene generato un solo token e ne vengono consumati tanti quanti sono le frecce uscenti;

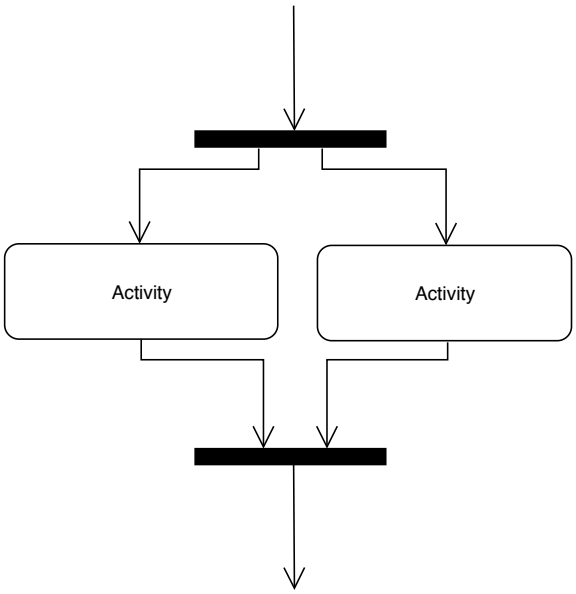


Figura 7: Fork e Join diagramma delle attività

- **Branch**: è rappresentato da un rombo con una freccia entrante e da due o più frecce uscenti. Ad ognuna di esse vengono associate delle *guardie*, indicate nel formato `[condizione guardia]`, ovvero delle condizioni che permettono ai token di proseguire per un solo ramo. Non viene generato alcun token; ma viene instradato;
- **Merge**: è il punto in cui gli i rami generati dal Branch tornano ad unirsi ed è rappresentato da un rombo con più frecce entranti ed una freccia uscente. È il punto dove viene chiusa la parte condizionale, in cui gli n rami generati dal *Branch* tornano ad unirsi. Non viene nè consumato nè prodotto alcun token, in questo caso vi è solo un passaggio;

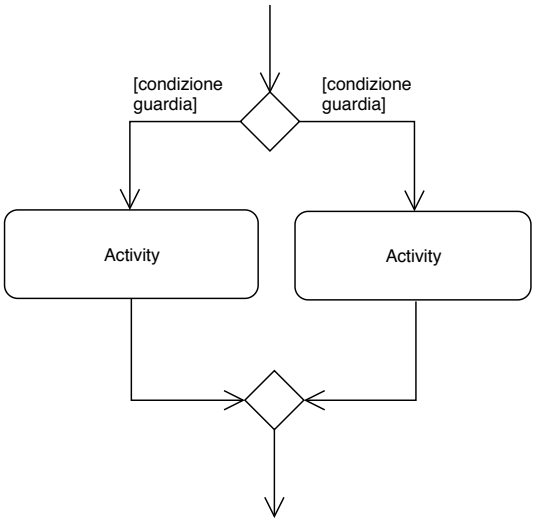


Figura 8: Branch e Merge diagramma delle attività

- **Pin**: è rappresentato da un piccolo quadratino, da dove entrano/escono frecce dalle *Activity*. Funge da indicatore del passaggio di un parametro, il cui tipo va indicato affianco seguendo il formato `tipoParametro`;

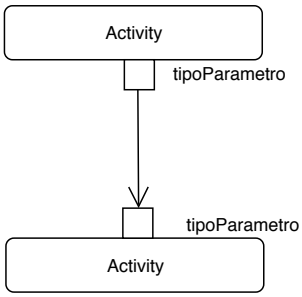


Figura 9: Pin diagramma delle attività

- **Segnali**: sono rappresentati con due figure "a incastro". La prima risulta non bloccante per l'emissione del segnale, la seconda invece risulta bloccante per la ricezione dello stesso. All'interno delle figure devono essere contenute le diciture *«signal sending»* e *«signal receipt»* rispettivamente, ognuna delle quali deve essere seguita da una descrizione breve e concisa del segnale rappresentato;

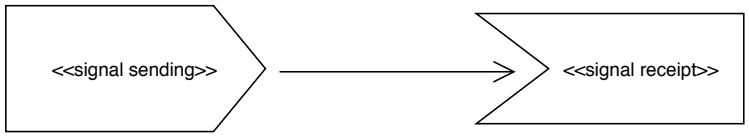


Figura 10: Segnali diagramma delle attività

- **Timeout**: è rappresentato da una clessidra. Il tempo risulta utile nel caso in cui si voglia modellare timeout o eventi ripetuti. I timeout sono composti da frecce entranti ed uscenti, invece gli eventi ripetuti vengono modellati solo con archi uscenti.
  I *timeout* devono iniziare con la parola chiave "wait", gli *eventi ripetitivi* con "every", in entrambi i casi la parola chiave vede al seguito un indicatore temporale o una descrizione in linguaggio naturale breve e concisa.

Figura 11: Timeout diagramma delle attività

2.2.4.7.2 Diagrammi delle classi

Un diagramma delle classi permette la definizione delle caratteristiche statiche di un sistema. Risulta di conseguenza utile per descrivere tutto ciò che non riguarda

l'ambiente a run-time. Ogni classe viene rappresentata con un rettangolo tripartito orizzontalmente, contenente nelle tre partizioni, partendo dall'alto:

1. **Nome della classe**: il nome assolve al ruolo di identificatore della classe. Deve essere significativo in funzione al compito svolto, scritto in inglese a lettere maiuscole e in grassetto. Nel caso di una classe *astratta* il nome deve essere presentato in formato italico, nel caso invece di un'*interfaccia* il nome deve sempre venire preceduto dalla direttiva `<<interface>>`;

2. **Attributi**: vengono presentati l'uno dopo l'altro, occupando ciascuno una riga a sé stante. Ogni attributo dichiarato deve seguire il formato:

   Visibilità nome: tipo [molteplicità ordinamento] = valori di default

   - **Visibilità**: ogni attributo dichiarato, é precedentemente, seguito dalla dichiarazione di visibilità:
     - **-**: indicatore di visibilità privata;
     - +: indicatore di visibilità pubblica;
     - #: indicatore di visibilità protetta;
     - ~ : indicatore di visibilità di package.
   - **Nome**: ogni attributo deve essere univoco e significativo, nel rispetto del formato `nomeVariabile: tipo`. Se la variabile è di tipo *costante* il nome deve essere scritto tutto in maiuscolo, nel rispetto del formato `NOMEVARIABILE: tipo`. Il tipo può essere anche definito dall'utente;
   - **Molteplicità**: nel caso di un numero multiplo di elementi, come accade in liste o array se ne può definire il numero esatto. Il formato da usare è `tipoVariaibile[molteplicità]`; * indica una molteplicità non conosciuta a priori e nel caso di un singolo elemento la sua dichiarazione può venire omessa;
   - **Ordinamento**: se vi è la necessità di far seguire all'attributo anche l'informazione sul ordinamento seguito, nel caso di collezioni in cui sia rispettato un ordine si utilizza la proprietà *orderd*; in caso contrario si fa ricorso ad*unordered*. La forma in uso per la sua dichiarazione è la seguente `tipoVariaibile[molteplicità ordinamento]`;
   - **Valori di default**: possono venire indicati anche valori di default per ogni attributo espresso.

   Nel caso di tipi non primitivi deve essere preferito l'uso di *tipi composti*, ovvero di una classe ad hoc in cui vengono definiti tutti i tipi che compongono il tipo principale e che viene collegata alla classe di partenza per mezzo di una freccia unidirezionale.
   Nel caso risulti necessario si possono usare dei *commenti*, ogni commento al suo interno può contenere un altro commento.

3. **Metodi**: i metodi hanno il compito di descrivere il comportamento, non l'implementazione di una classe. Si devono presentare uno di seguito all'altro, ognuno occupante una riga a sé stante.
   I metodi devono essere dichiarati seguendo il formato:

   Visibilità nomeMetodo (lista-paramentri-formali): return-type

- **Visibilità**: anche nel caso dei metodi, questi devono venire preceduti da un indicatore di visibilità, come descritto per gli attributi;
- **NomeMetodo**: rappresenta il nome del metodo che deve essere significativo e presentato in inglese secondo il formato uppercase;
- **Lista dei parametri formali**: può essere composta da 0 fino ad n parametri, ciascuno dei quali segue le medesime regole imposte per gli attributi. Ogni parametro presente nella lista deve venire separato da una virgola;
- **Return-type**: rappresenta il tipo di ritorno dell'operazione.

Se un metodo è *astratto* questo deve essere scritto in italico, se invece si presenta come *statico* deve avvenire la sottolineatura dello stesso.
Nella lista dei metodi possono non essere inclusi i metodi dei costruttori, i metodi getter e setter semplici, ammeno di apposita direttiva di un *Progettista*.
Anche nel caso dei metodi si prevede possano venire accompagnati da *commenti*.

Nel caso che una classe non contenga né variabili né metodi, nel diagramma appariranno le sezioni ad essi dedicati vuote.
I diagrammi sono collegati tra di loro grazie a l'uso di frecce unidirezionali, che ne indicano le **dipendenze**:

- **Dipendenze**: viene rappresentata con una freccia tratteggiata da una classe A ad una classe B. Indica che A dipende da B secondo una delle seguenti primitive, posta in capo vicino alla freccia:
  - **«call»**: A invoca un metodo di B;
  - **«create»**: A crea istanze di B;
  - **«derivate»**: A deriva da B;
  - **«instantiate»**: A è un'istanza della classe B;
  - **«permit»**: B permette ad A di accedere ai suoi campi privati;
  - **«realize»**: A è un implementazione di una specifica o di un'interfaccia definita da A;
  - **«refine»**: indica un raffinamento tra differenti livelli semantici;
  - **«substitute»**: A si può sostituire con B;
  - **«trace»**: tiene traccia dei requisiti o di come i cambiamenti di una parte di modello si colleghino ad altre;
  - **«use»**: A richiede B per la sua implementazione.

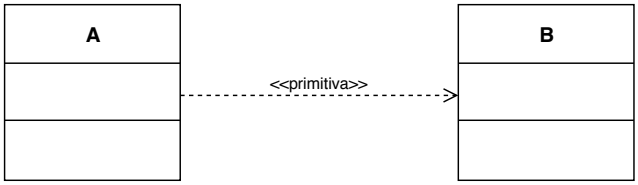


Figura 12: Relazione di dipendenza in un diagramma delle classi

- **Associazione**: viene rappresentata con una freccia semplice, da classe A a classe B. Indica che la classe A contiene dei campi dati o delle istanze di B. Le molteplicità possibili, che vengono indicate agli estremi della freccia, sono le seguenti:
  - **1**: A possiede un'istanza di B;
  - **0..1**: A possiede 0 o 1 istanze di B;
  - **0..***: A possiede 0 o più istanze di B;
  - ***: A possiede più istanze di B;
  - **n**: A può possedere n istanze di B.

Figura 13: Relazione di associazione in un diagramma delle classi

- **Aggregazione**: viene rappresentata con una freccia a diamante vuota, da classe A a classe B. Indica che la relazione "B è parte di A". Inoltre gli aggregati possono essere condivisi;

Figura 14: Relazione di aggregazione in un diagramma delle classi

- **Composizione**: viene rappresentata con una freccia a diamante piena. Si tratta di un'aggregazione ancora più forte di quanto descritto al punto sopra. Da classe A a classe B indica che:
  - Le due classi devono venire usate sempre assieme;
  - Gli aggregati possono appartenere ad un solo aggregato (aggregato con cardinalità (1,1));
  - Solo l'oggetto intero può distruggere le sue parti.

Figura 15: Relazione di composizione in un diagramma delle classi

- **Generalizzazione**:

– **Ereditarietà**: viene rappresentata con una freccia vuota continua, da classe A a classe B. Indica "che un oggetto di A è anche un oggetto di B" e rappresenta il massimo grado di dipendenza fra le classi, soprattutto se si estende un tipo concreto;

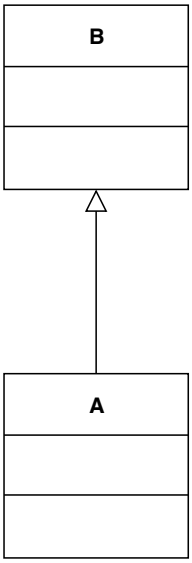


Figura 16: Relazione di generalizzazione in un diagramma delle classi

– **Subtyping**: si verifica nel caso di classi astratte o di interfacce. Nel caso in cui si verifichi un implementazione di interfacce con classi astratte/concrete questa viene rappresentata per mezzo di una freccia tratteggiata.

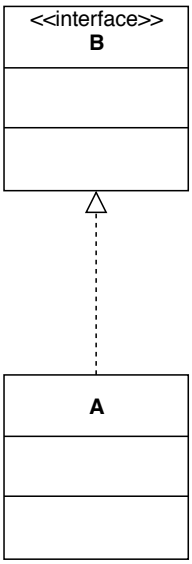


Figura 17: Relazione di implementazione in un diagramma delle classi

2.2.4.7.3 Diagrammi dei package

Un diagramma di package permette la definizione di un livello più al dettaglio dell'architettura. Ciò consiste nel raggruppamento di un numero arbitrario di elementi

UML, ovvero classi, in un unità di livello più alto.
Ogni package viene rappresentato mediante l'uso di un rettangolo con un'etichetta per il nome, con il compito di contenere i diagrammi delle classi appartenenti al package ed eventuali sotto-package.
Il nome deve essere completamente qualificato. Il formalismo in uso è il seguente:

package::package:: ... :: classe

Possono essere raggruppate solo classi, ed ogni classe appartiene ad un unico package.
Ogni elemento in un package può avere *visibilità* pubblica (+) o privata (-).
Le **dipendenze** tra i vari package sono segnalate per mezzo dell'uso di una freccia tratteggiata. Tale freccia esistente dal package A al package B, indica una dipendenza di A nei confronti di B. Sono da evitare le dipendenze cicliche.

Figura 18: Relazione di dipendenza fra packages

2.2.4.7.4 Diagrammi di sequenza

Un diagramma di sequenza permette di definire formalmente la collaborazione di un gruppo di oggetti che devono implementare collettivamente un comportamento.
In un diagramma viene descritto uno scenario composto da una determinata sequenza di azioni, presentando un senso di lettura verticale dall'alto verso il basso, come indicatore dello scorrere del tempo.
Attraverso l'uso di questi si è in grado di:

- Modellare la logica di procedure, metodi ed operazioni;
- Mostrare l'interazione tra le varie componenti coinvolte;
- Comprendere e pianificare quali sono le funzionalità di uno specifico scenario.

Ogni specifica entità deve venire rappresentata con l'uso di un rettangolo, nel quale al suo interno ne è indicato il nome, nel rispetto della forma `istanza : nomeClasse` in grassetto.
Al di sotto della sua rappresentazione deve essere collocata la sua linea di vita, mediante l'uso di una linea tratteggiata. In alcune parti, quest'ultima, viene sormontata dalla *barra di attivazione* che indica i momenti in cui l'entità è effettivamente attiva (la barra viene omessa quando l'entità è attiva sempre).
Dalla barra di attivazione si dipartiscono delle frecce, indicatori di messaggio/segnale, verso altre entità che sono state o precedentemente già istanziate, oppure verso una nuova istanza di classe per crearla.
Nel dettaglio le tipologie di frecce utilizzate sono:

- **Freccia piena per indicare un messaggio sincrono**: rappresenta la chiamata di un metodo. Sopra tale freccia deve essere specificato il metodo invocato, nel rispetto della forma `nomeMetodo(lista-parametri-formali)`. Ci si aspetta un segnale di risposta;

- **Freccia per indicare un messaggio asincrono**: rappresenta i medesimi concetti espressi al punto sopra e fa uso dello stesso formalismo, però in questo caso non ci si aspetta un segnale di risposta;
- **Freccia tratteggiata sormontata da «create»**: rappresenta la creazione di una nuova entità e termina sempre in un rettangolo che ne contiene il nome, nel rispetto della forma `istanza: nomeClasse`;
- **Freccia piena sormontata da «destroy»**: rappresenta la distruzione di un' entità, questa termia sempre con una X, nel quale muore la linea di vita della stessa;
- **Freccia tratteggiata per indicare il ritorno di un metodo chiamato**: rappresenta un segnale di risposta e sopra tale freccia deve essere indicato il tipo di ritorno, nel rispetto della forma `tipoFormato`.

Dove ritenuto necessario è possibile determinare sui diagrammi di sequenza dei frame che permettono ciclo o condizioni. I possibili sono:

- **alt**: indica dei frammenti multipli in alternativa fra loro, viene eseguito solo il frammento per il quale la condizione risulta verificata;
- **opt**: indica un singolo frammento che viene eseguito solo se la condizione specificata risulta vera;
- **par**: indica che ogni frammento presente viene eseguito in parallelo;
- **loop**: indica un ciclo, dove ogni frammento può venire eseguito più volte. La condizione di iterazione viene specificata da una guardia;
- **region**: indica una regione critica, dove il frammento può venire eseguito da un unico thread alla volta;
- **neg**: il frammento indica un'interazione non valida;
- **ref**: indica un riferimento, ovvero un'interazione definita in un altro diagramma;
- **sd**: è utilizzato per racchiudere un intero diagramma di sequenza.

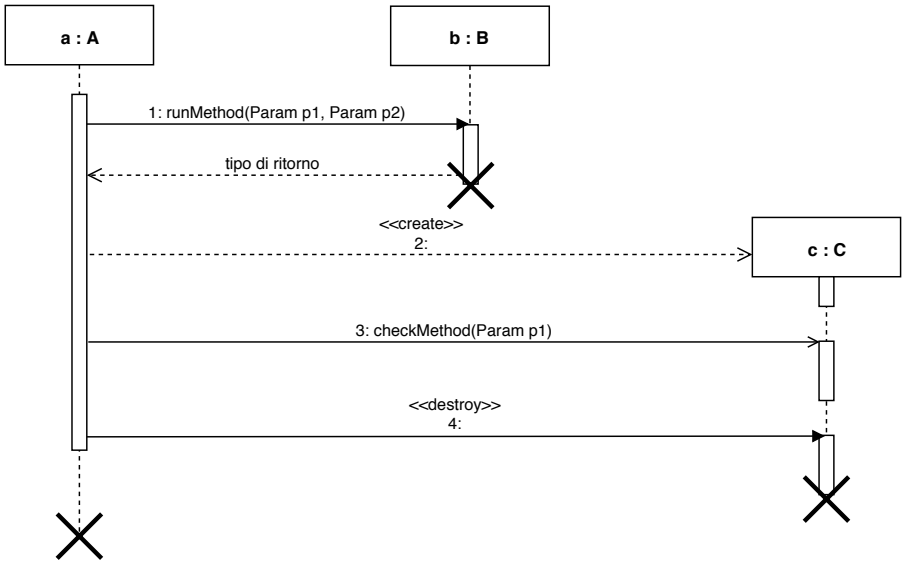


Figura 19: Diagramma di sequenza con rappresentati tutti i tipi di segnali

2.2.4.7.5 Diagrammi dei casi d'uso

Un caso d'uso ha lo scopo di descrivere le interazioni che coinvolgono il sistema, in analisi, con il resto del mondo. Nello specifico si intende un caso d'uso come un insieme di scenari, sequenze di azioni, con in comune il medesimo obiettivo per un utente.
Un diagramma deve assolvere al compito di non rappresentare alcun dettaglio implementativo e permettere la descrizione delle funzionalità coinvolte con una visione esterna al sistema, come percepita dall'utente.
Gli elementi presenti all'interno di un caso d'uso sono i seguenti:

- **Attore**: gli attori rappresentano tutto ciò che è esterno al sistema e ci interagisce. Un caso d'uso ha l'obiettivo di offrire funzionalità all'attore che può essere o persona o un altro sistema esterno. Non è concessa la possibilità di definire alcun dettaglio implementativo sui modi di interazione.
  Un attore viene disegnato con un omino stilizzato, con sotto accompagnato dal nome nel rispetto della forma `Nome attore`;

Figura 20: Attore di un caso d'uso

- **Caso d'uso**: consiste in un ovale, nel quale al suo interno deve venire inserita la descrizione dello stesso.
  La forma utilizzata consiste in una numerazione `UCx.y`, con a seguire `:` e `Breve descrizione del caso d'uso`. La descrizione deve essere concisa, ma descrittiva dello scenario. Ogni caso d'uso viene associato ad un attore e viceversa per mezzo di una linea semplice.

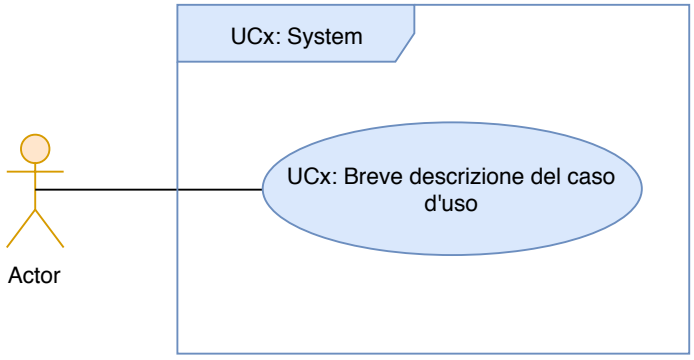


Figura 21: Rappresentazione di un caso d'uso

Sono, invece elementi opzionali, contenuti all'interno di un diagramma in base alle funzionalità che devono essere rappresentate:

- **Inclusione**: se esiste un'inclusione tra un caso d'uso A e un caso d'uso B, significa che ogni istanza di A deve eseguire anche B. B è perciò incondizionatamente incluso nell'esecuzione di A, tuttavia B non è a conoscenza di essere incluso in A, dando in questo modo esclusiva responsabilità di esecuzione ad A. Tale strategia evita la ripetizione e aumenta il riutilizzo di una medesima struttura, ma va usata solo nei casi che rispettano le condizioni appena dichiarate.
  Un' inclusione viene rappresentata con una freccia tratteggiata, che collega i casi d'uso coinvolti, in direzione del caso d'uso incluso. Viene posta al di sopra della stessa la direttiva `<<include>>`;

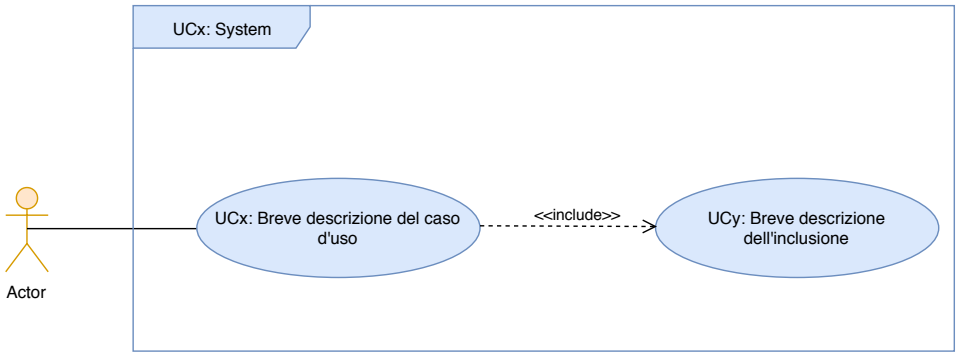


Figura 22: Rappresentazione di un caso d'uso con inclusione

- **Estensione**: se esiste un'estensione tra un caso d'uso A e un caso d'uso B, significa che ogni istanza di A esegue B in modo incondizionato, tuttavia l'esecuzione di B ha come conseguenza l'interruzione di A e la responsabilità di esecuzione è esclusiva di solo chi estende, ovvero B.
  Un'estensione viene rappresentata con una freccia tratteggiata, che collega i casi d'uso coinvolti, in direzione del caso d'uso che viene esteso. Viene posta al di sopra della stessa la direttiva `<<extend>>` e devono essere indicate, in un quadrato con un angolo piegato, le condizioni per il verificarsi dell'estensione e il nome della stessa, collegato con una linea tratteggiata alla freccia;

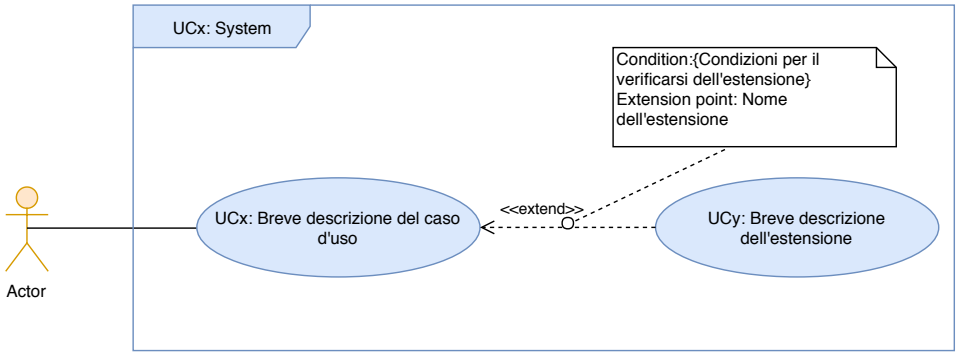


Figura 23: Rappresentazione di un caso d'uso con estensione

- **Generalizzazione**: la generalizzazione può coinvolgere sia attori che casi d'uso. Nel primo caso se "Utente" è generalizzazione di "Utente autenticato" significa che Utente autenticato condivide almeno le medesime funzionalità di Utente. Nel caso, invece, dei casi d'uso, i figli possono aggiungere funzionalità rispetto ai padri e modificarne il comportamento facendo ereditare ai figli tutte le funzionalità del padre che il figlio non ha esplicitamente ridefinito.
  Le generalizzazioni sia di attori che di casi d'uso vengono rappresentate con una freccia continua vuota dal elemento figlio all'elemento padre.

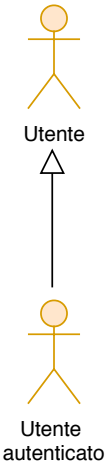


Figura 24: Generalizzazione di attori

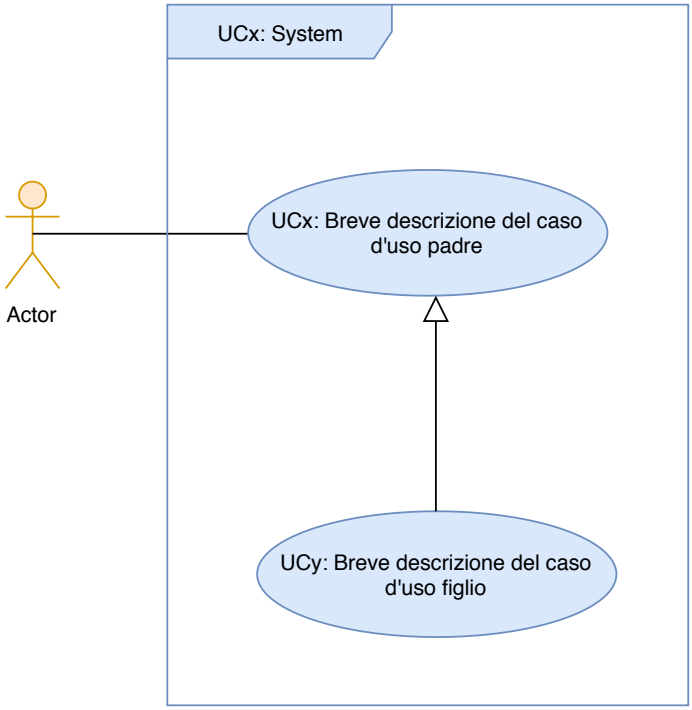


Figura 25: Rappresentazione di una con generalizzazione di casi d'uso

Un caso d'uso per essere completo, oltre al diagramma deve essere sempre accompagnato da una descrizione testuale dello stesso.
Come il formalismo spiegato in questa sotto sezione è utilizzato, è spiegato nel dettaglio in §2.2.3.2.

2.2.4.8 The Twelve-Factor App

Il *Capitolato d'Appalto* richiede che lo sviluppo rispetti i 12 fattori esposti nel documento *The Twelve-Factor App*$_{\mathbf{G}}$, che permettono di avere una buona architettura basata sui *microservizi*$_{\mathbf{G}}$. Tali punti sono espressi nel dettaglio nell'appendice A.

2.2.4.9 Qualità della progettazione

Per il fornitore vige l'obbligo di garantire un prodotto software che soddisfi tutti i requisiti individuati durante l'attività di Analisi. Per questo la progettazione ha come obiettivo il perseguimento della propria qualità nel rispetto delle visioni di intrinsichezza, relatività e misurabilità. Per riuscire ad ottenere quanto appena descritto è vincolante il rispetto dei seguenti punti:

- Piena chiarezza delle funzionalità e dei requisiti da soddisfare prima di iniziare qualsiasi attività di progettazione. Ogni attività deve essere coerente con quanto descritto all'interno del *Analisi dei Requisiti v3.0.0.* e non possono venire individuate nuove funzionalità da realizzare in quanto non è di pertinenza dell'attività in esame;
- Deve venire effettuata una ripartizione corretta delle risorse necessarie e un'attività di implementazione il più realistica possibile, sulla base di tempo/persona

disponibile, con scadenze prestabilite e task di lavoro il più piccoli possibili. Tutto ciò per facilitare il tracciamento dei problemi che possono insorgere durante il periodo di Codifica;

- Garantire correttezza per costruzione. Una volta presentata la progettazione al committente questa si deve dimostrare il più corretta possibile, per poter attualizzare a pieno il modello di sviluppo incrementale. Come dichiarato all'interno del *Piano di Progetto 4.0.0*;
- Pieno rispetto degli standard di riferimento con l'uso normato all'interno del suddetto documento;
- Ogni componente del sistema deve essere definito modularmente e nel rispetto della singola responsabilità, in modo da mantenere ad un livello accettabile la dipendenza tra le varie componenti;
- Devono venir garantire tutte le condizioni minime necessarie di tempo e strumenti utili a produrre, testare, installare e mettere in funzione il prodotto software; in mancanza delle quali la buona qualità della progettazione non può venire garantita.

2.2.5 Codifica

L'attività di codifica permette di trasporre in codice l'architettura ad alto livello pensata dai *Progettisti* rendendola eseguibile dai calcolatori. La codifica viene svolta dai *Programmatori* che, seguendo fedelmente quanto descritto dall'attività di Progettazione, realizzano il prodotto software finale attraverso il processo di programmazione.

2.2.5.1 Scopo

L'obiettivo di questa attività è l'effettiva realizzazione del prodotto richiesto. Il prodotto software da realizzare deve essere conforme con quanto contrattato con il proponente *Imola Informatica S.P.A.*.

2.2.5.2 Descrizione

La codifica è strettamente vincolata a quanto indicato nell'allegato tecnico Product Baseline, che verrà redatto nella successiva Revisione di Qualifica. In particolare, la struttura del codice sorgente dovrà sottostare ai vincoli qualitativi definiti nel documento *Piano di Qualifica v3.0.0*, al fine di garantire un prodotto facilmente manutentibile e di elevata qualità.

2.2.5.3 Convenzioni linguistiche

Qualsiasi commento, nome di file, classe o metodo dovrà essere redatto in lingua inglese. Questa scelta è motivata dalle seguenti considerazioni:

- dare la possibilità di leggere il codice sorgente in maniera agevole anche a chi non è madrelingua italiano;
- alcuni strumenti di sviluppo sono in grado di individuare errori ortografici nel codice, ma l'unica lingua supportata solitamente è l'inglese;
- molte parole chiave utilizzate dai linguaggi di programmazione si rifanno alla lingua inglese. Aggiungere commenti nella stessa lingua sembra quindi più spontaneo e naturale.

2.2.5.4 Linguaggi scelti

Come da *Capitolato d'Appalto* del progetto Butterfly, al gruppo fornitore è concessa la libertà di scegliere il linguaggio di programmazione da utilizzare. Viene però consigliato l'utilizzo di almeno un linguaggio tra *Java*$_{\mathbf{G}}$ (versioni 8 o superiori), *Python*$_{\mathbf{G}}$, *Node.js*$_{\mathbf{G}}$. Il gruppo ha adottato la versione *LTS*$_{\mathbf{G}}$ più recente di Java, Java 11, per lo sviluppo dei microservizi che prevedono un'interazione con il broker Kafka. Questa scelta è stata motivata dal fatto che Kafka sia stato nativamente sviluppato e ottimizzato per la *JVM*$_{\mathbf{G}}$, il che ha naturalmente permesso lo sviluppo di una community più ampia e una documentazione meglio strutturata. Esistono comunuque *client*$_{\mathbf{G}}$ sviluppati da terze parti per supportare altri linguaggi, tra cui Node.js e Python; tuttavia, il client ufficiale Java (org.apache.kafka:kafka-clients) è il più documentato. Per quanto concerne il Gestore Personale, invece, che espone solamente un'interfaccia *HTTP*$_{\mathbf{G}}$ *REST*$_{\mathbf{G}}$ e quindi non si preoccupa dell'interazione con il *Broker*$_{\mathbf{G}}$, il gruppo *DStack* ha optato per *Node.js*$_{\mathbf{G}}$, a cui è aggiunto il supporto ai tipi grazie alla scelta di usare *TypeScript*$_{\mathbf{G}}$. *Node.js* è stato scelto principalmente per i seguenti motivi:

- il supporto a JSON è nativo ed è estremamente più intuitivo utilizzare JSON in Node.js che in Java;
- *npm* e *yarn*, i *package manager*$_{\mathbf{G}}$ JavaScript più utilizzati, sono risultati essere molto più semplici e immediati nell'utilzzo rispetto a *Maven*$_{\mathbf{G}}$ di Java;
- è un framework fortemente richiesto a livello lavorativo, che ha suscitato l'interesse dei membri del team già in fase di scelta del capitolato;
- combinando le ultime versioni di JavaScript (ES6, ES7) con TypeScript, è possibile scrivere codice sufficientemente intuitivo per gli sviluppatori abituati a scrivere in Java, ma estremamente meno verboso;
- i tempi di traspilazione da *TypeScript* a *JavaScript* sono generalmente molto minori rispetto a quelli richiesti per compilare moduli Java equivalenti.

2.2.5.5 Convenzioni per la documentazione

- **TODO**: è possibile iniziare i commenti con la stringa maiuscola **TODO** seguita dal carattere **:** (due punti) per indicare codice di cui verrà sicuramente effettuto un *refactoring*$_{\mathbf{G}}$ in futuro. Questo è ammissibile solamente nei casi in cui vi sia l'impossibilità temporanea di trovare una soluzione più appropriata al problema riscontrato;
- **Intestazione**: Tutti i file sorgenti consegnati, file di configurazione esclusi, devono iniziare con la seguente intestazione, adeguatamente contenuta in un commento di blocco:

```
@project:   Butterfly$
@author:    Cognome e nome dell'autore originale
@module:    Nome del package
@fileName:  Nome del file
@created:   YYYY-MM-DD

------------------------------------------------------------------------------------------
Copyright 2019 DStack Group.
Licensed under the MIT License. See License.txt in the project root for license information.
------------------------------------------------------------------------------------------

@description:
Descrizione (eventualmente su molteplici righe) del contenuto di questo file.
```

2.2.5.6 Convenzioni di nomenclatura e formattazione dei files

In generale, deve valere la regola che i nomi dei file siano brevi ma descrittivi, privi di spazi e di caratteri speciali, ad eccezione del trattino (-) e dell'underscore (_). L'estensione del file va inoltre sempre esplicitata, e sempre in minuscolo. Se il nome del file contiene una sigla, ad esempio "JSON" o "URL", essa deve essere riportata in maiuscolo. A seconda del linguaggio adottato, le convenzioni possono presentare alcune differenze, riepilogate di seguito:

- **TypeScript**: è il linguaggio di programmazione utilizzato per la realizzazione del Gestore Personale. L'estensione del file è `.ts`. Tutti i nomi di file Typescript devono essere scritti in *lowerCamelCase*$_{\mathbf{G}}$ o in *PascalCase*$_{\mathbf{G}}$. Valgono le seguenti regole:
  - se il file contiene la definizione di una classe o di un'interfaccia, la prima lettera del nome del file deve essere scritta maiuscolo;
  - altrimenti, se il file contiene la definizione di funzioni o l'esportazione di moduli, la prima lettera del nome del file deve essere minuscola.
- **Java**: è il linguaggio di programmazione utilizzato per la realizzazione dei *Producer*$_{\mathbf{G}}$ e dei *Consumer*$_{\mathbf{G}}$. L'estensione del file è `.java`. Tutti i nomi di file Java devono essere scritti in *PascalCase*$_{\mathbf{G}}$, con la prima lettera del nome del file sempre maiuscola.
- **XML**: è il formato di serializzazione testuale usato per le configurazioni e la dichiarazione delle dipendenze dei servizi scritti in Java. L'estensione del file è `.xml`. Tutti i nomi di file XML devono essere scritti in *lowercase*$_{\mathbf{G}}$.
- **JSON**: è il formato di serializzazione testuale utilizzato soprattutto per le configurazioni delle applicazioni eseguite in *Node.js*$_{\mathbf{G}}$. L'estensione del file è `.json`. Tutti i nomi di file JSON devono essere scritti in *lowercase*$_{\mathbf{G}}$, eventualmente con più parole separate da punti.
- **YAML**: è il formato di serializzazione testuale utilizzato per descrivere i microservizi e i legami esistenti tra essi. I file di configurazione di *Docker Compose*$_{\mathbf{G}}$ usano questo formato. L'estensione del file è `.yml`. Tutti i nomi di file YAML devono essere scritti in *kebab-case*$_{\mathbf{G}}$. In particolare, per i file di configurazione di Docker Compose, il nome del file deve essere il seguente:

  docker-compose[.ID].yml

dove [.ID] rappresenta un identificativo opzionale che indica una versione modificata del file *docker-compose.yml* presente in quella particolare directory. Può infatti essere utile fornire configurazioni diverse tra ambiente di sviluppo (contraddistinto dal file di configurazione *docker-compose.dev.yml*) e ambiente di produzione (*docker-compose.yml*). Per un esempio di file *docker-compose.yml*, si veda Listing 1.

```yaml
version: '3.5'

services:
  postgres:
    image: "postgres:11-alpine"
    restart: always
    ports:
      - "5432:5432"
    volumes:
      - ./user-manager-database/init.sql:/docker-entrypoint-initdb.d/init.sql
      - pg_data:/var/lib/postgresql/data/
    env_file: ./user-manager-database/.user-manager-database.env
    networks:
      - user-manager-network

  user-manager:
    build:
      context: ./user-manager-rest-api
    restart: always
    ports:
      - "5000:5000"
    depends_on:
      - postgres
    env_file: ./user-manager-rest-api/.user-manager-rest-api.env
    networks:
      - user-manager-network
      - middleware-dispatcher-network

volumes:
  pg_data:

networks:
  user-manager-network:
    name: user-manager-network
  middleware-dispatcher-network:
    external:
      name: middleware-dispatcher-network
```

Listing 1: Esempio di file docker-compose.yml

- **ENV**: è il formato di serializzazione testuale utilizzato per elencare le variabili d'ambiente da rendere accessibili ad un servizio, solitamente istanziato da uno strumento come *Docker*$_{\mathbf{G}}$ o *Docker Compose*$_{\mathbf{G}}$. L'estensione del file è `.env`. Tutti i nomi di file ENV devono essere scritti in *kebab-case* e, per convenzione, devono iniziare con un punto (`.`).

- **SQL**: è il linguaggio di programmazione utilizzato per la definizione del database del Gestore Personale e per la scrittura di interrogazioni da eseguire sullo stesso. D'ora in avanti, per **SQL** si intenderà la variante del linguaggio

```
AVRO_SCHEMA_REGISTRY_URL=http://schema-registry:8081
KAFKA_MAX_BLOCK_MS_CONFIG=1000
KAFKA_CLIENT_ID_CONFIG=middleware-dispatcher
KAFKA_POLL_DURATION_MS=2000

# This is used as prefix for the topic that should be sent to
# the consumer
MESSAGE_TOPIC_PREFIX=contact-

USER_MANAGER_URL=http://user-manager:5000/search/events/users
USER_MANAGER_REQUEST_TIMEOUT_MS=1000
```

Figura 26: Esempio di file ENV

standard adottata dal database **Postgres**. L'estensione del file è `.sql`. Tutti i nomi di file SQL che contengono la definizione del database, viste, e query (eventualmente parametrizzate) devono essere scritti in *lowerCamelCase*$_{\mathbf{G}}$. Nel caso in cui sia necessario alterare la struttura del database o creare dei file di *seed*, tali file devono essere prefissati da una stringa in formato testuale che ne indichi la data e l'ora di creazione:

YYYY-MM-DD_hh:mm:ss.fileName

dove **YYYY** indica l'anno (ad esempio "2019"), **MM** indica il mese (ad esempio "02" per indicare "Febbraio"), **DD** indica il giorno (ad esempio "25"), **hh** indica l'ora, **mm** i minuti, **ss** i secondi e **fileName** il nome del file in *lowerCamelCase*$_{\mathbf{G}}$.

2.2.5.7 Larghezza delle linee di codice

Per trovare il giusto compromesso tra la leggibilità del codice e la compattezza dei file, si è fissato a **120** il numero di caratteri stampabili in una riga di codice. È però obbligatorio scrivere una singola espressione per riga. I moderni IDE offrono la possibilità di vedere una barra laterale in demarcazione sulla colonna massima fissata, in questo caso, in corrispondenza al 121esimo carattere.

2.2.5.8 Indentazione e parentesi

In caso di costrutti condizionali molto brevi, è consigliabile usare costrutti inline (tramite operatore ternario), a patto che questo non pregiudichi la leggibilità del codice. La decisione dell'indentazione dei file è stata presa sulla base della convenzione delle community online di TypeScript e Java.

2.2.5.9 Dimensione massima file

I file TypeScript e Java non devono mai superare le **200** linee di codice, al fine di imporre quanto più possibile l'aggregazione e il riutilizzo di metodi di utilità esterni alla classe corrente. Dal calcolo della lunghezza massima sono escluse le righe dedicate all'importazioni dei moduli necessari al corretto funzionamento del codice contenuto nei file.

2.2.5.10 Convenzioni per la scrittura di codice in Java

- Sono proibiti gli import generici, ovvero che presentano il carattere * (asterisco);

- le importazioni di package devono essere poste tutte in cima al file sorgente;
- l'indentazione deve essere composta da 4 spazi per ogni livello;
- le parentesi tonde, quadre o graffe devono sempre essere separate da una parola chiave o da un operatore da un carattere di spazio;
- le parentesi tonde, quadre o graffe devono sempre essere separate da una parola chiave o da un operatore da un carattere di spazio;
- limitare l'uso della parola chiave `var` ai casi in cui non ci sia ambiguità sul tipo del valore dichiarato;
- preferire la definizione di metodi statici che ritornino un'interfaccia funzionale alla definizione in linea di funzioni lambda;
- è proibita la dichiarazione di loop il cui corpo sia vuoto;
- è proibita la dichiarazione di più variabili in una sola volta;
- i costrutti `switch case` devono sempre includere un caso di `default`;
- nel caso di costrutti `try catch`, è obbligatorio catturare sia l'eccezione più precisa possibile, sia la più generica `Exception`;
- è obbligatorio usare le parentesi graffe nel caso di istruzioni di `if else`, di cicli `while` e di istruzioni analoghe;

Per agevolare gli sviluppatori nel rispettare le convenzioni scelte dal gruppo *DStack*, è stato definito un file **checkstyle.xml**, interpretabile dallo strumento di *analisi statica*$_{\mathbf{G}}$ *checkstyle*.

```
<?xml version="1.0"?>
<!DOCTYPE module PUBLIC "-//Puppy Crawl//DTD Check Configuration 1.2//EN"
        "http://www.puppycrawl.com/dtds/configuration_1_2.dtd">
<module name="Checker">
    <property name="charset" value="UTF-8"/>
    <property name="severity" value="error"/>

    <module name="TreeWalker">
        <module name="BooleanExpressionComplexity"/>
        <module name="CyclomaticComplexity"/>
        <module name="LineLength">
            <property name="max" value="120"/>
            <property name="ignorePattern" value="^package.*|^import.*"/>
        </module>
        <module name="MethodLength"/>
        <module name="EmptyCatchBlock">
            <property name="exceptionVariableName" value="expected"/>
        </module>
        <module name="AvoidStarImport"/>
        <module name="IllegalImport">
            <property name="illegalPkgs" value="java.io"/>
        </module>
        <module name="NeedBraces"/>
        <module name="WhitespaceAround">
            <property name="allowEmptyConstructors" value="true"/>
            <property name="allowEmptyLambdas" value="true"/>
            <property name="allowEmptyMethods" value="false"/>
            <property name="allowEmptyTypes" value="false"/>
            <property name="allowEmptyLoops" value="false"/>
        </module>
        <module name="OneStatementPerLine"/>
        <module name="MultipleVariableDeclarations"/>
        <module name="MissingSwitchDefault"/>

        <module name="PackageName">
            <property name="format" value="^[a-z]+(\.[a-z][a-z0-9]*)*$"/>
        </module>
        <module name="TypeName">
        </module>
        <module name="MemberName">
            <property name="format" value="^[a-z][a-z0-9][a-zA-Z0-9]*$"/>
        </module>
        <module name="ParameterName">
            <property name="format" value="^[a-z]([a-z0-9][a-zA-Z0-9]*)?$"/>
        </module>
        <module name="CatchParameterName">
            <property name="format" value="^[a-z]([a-z0-9][a-zA-Z0-9]*)?$"/>
        </module>
        <module name="LocalVariableName">
            <property name="tokens" value="VARIABLE_DEF"/>
            <property name="format" value="^[a-z]([a-z0-9][a-zA-Z0-9]*)?$"/>
        </module>
        <module name="ClassTypeParameterName">
            <property name="format" value="(^[A-Z][0-9]?)$|([A-Z][a-zA-Z0-9]*[T]$)"/>
        </module>
        <module name="MethodTypeParameterName">
            <property name="format" value="(^[A-Z][0-9]?)$|([A-Z][a-zA-Z0-9]*[T]$)"/>
        </module>
        <module name="InterfaceTypeParameterName">
            <property name="format" value="(^[A-Z][0-9]?)$|([A-Z][a-zA-Z0-9]*[T]$)"/>
        </module>
        <module name="AbbreviationAsWordInName">
            <property name="tokens" value="VARIABLE_DEF,CLASS_DEF"/>
            <property name="ignoreStatic" value="false"/>
            <property name="allowedAbbreviationLength" value="0"/>
            <property name="allowedAbbreviations" value="XML,URL,JSON"/>
        </module>
        <module name="OverloadMethodsDeclarationOrder"/>
        <module name="VariableDeclarationUsageDistance"/>
        <module name="MethodParamPad"/>
        <module name="MethodName">
            <property name="format" value="^[a-z][a-z0-9][a-zA-Z0-9_]*$"/>
        </module>
        <module name="CommentsIndentation"/>
    </module>
    <module name="FileLength">
        <property name="max" value="200"/>
    </module>
    <module name="FileTabCharacter">
        <property name="eachLine" value="true"/>
        <property name="fileExtensions" value="java"/>
    </module>
    <module name="Header">
        <property name="headerFile" value="./header"/>
        <property name="fileExtensions" value="java"/>
    </module>
</module>
```

2.2.5.11 Convenzioni per la scrittura di codice in TypeScript

Le seguenti convenzioni sono fortemente ispirate alle linee guida definite da Microsoft, azienda che ha avviato lo sviluppo di TypeScript e che ne gestisce i rilasci.

- Ci può essere una sola funzione per riga;
- le importazioni di moduli devono essere poste tutte in cima al file sorgente;
- l'indentazione deve essere composta da 2 spazi per ogni livello;
- le parentesi tonde, quadre o graffe devono sempre essere separate da una parola chiave o da un operatore da un carattere di spazio;
- ogni istruzione deve terminare con un carattere di `;` (punto e virgola);
- le dichiarazioni di tipo devono terminare con un carattere `;` (punto e virgola);
- le dichiarazioni di interfaccia, di classe e di enum non devono terminare con il carattere `;` (punto e virgola);
- ogni valore di tipi enumerabile e ogni campo delle interfacce deve essere adeguatamente commentato;
- nel caso di assegnazione di valori agli oggetti, assegnare i valori ad identificatori di variabili chiamati con lo stesso nome della chiave dell'oggetto a cui saranno assegnati;
- nella gestione del flusso asincrono dei dati, è preferibile l'approccio `async / await` rispetto all'utilizzo di `Promise`;
- per indicare liste di dati, è preferibile la forma `T[]` rispetto a `Array<T>`;
- nessun metodo pubblico delle classi deve essere marcato dalla parola chiave `public`, che è la visibilità di default;
- tutti gli argomenti devono presentare la marcatura di tipo;
- nel caso di un'espressione lambda di un unico valore, il cui tipo sia implicitamente inferito, è vietato l'utilizzo delle parentesi tonde;
- qualora fosse necessario invocare una funzione con più di 2 argomenti, deve essere utilizzato un oggetto il cui tipo deve essere adeguatamente definito;
- ove possibile, preferire la definizione di costanti alla definizione di variabili;
- preferire operazioni immutabili ad operazioni mutabili;
- ove possibile, preferire la creazione di metodi statici componibili a metodi che mutano uno stato;
- se una libreria di terze parti è sprovvista della propria definizione dei tipo, è obbligatorio crearne una, per evitare che il compilatore inferisca implicitamente il tipo `any`;
- è proibito l'utilizzo esplicito dell'istruzione `console.log`, in favore del debugger integrato nell'editor scelto e di un logger più avanzato;

- è proibito l'utilizzo, sia implicito che esplicito, del tipo generico `any`;
- è proibito l'utilizzo del metodo `require()` per importare moduli nello scope corrente, usare invece la dichiarazione di `import`;
- è proibito l'utilizzo della parola chiave `var`;
- è proibito l'utilizzo della parola chiave `eval`;
- è proibito l'utilizzo della funzione `setTimeout()`;
- è proibito l'utilizzo della concatenazione tra stringhe mediante l'operatore di somma, in favore della templatizzazione di stringhe.[6]

TypeScript$_{\mathbf{G}}$, pur migliorando sensibilmente *JavaScript*$_{\mathbf{G}}$, ha ereditato da quest'ultimo la possibilità di permettere convenzioni alternative agli sviluppatori, pur garantendo la corretta compilazione del prodotto. Ad esempio, è possibile scrivere programmi TypeScript corretti senza usare i `;` (punti e virgola) alla fine dell'istruzione, o separare la definizione dei campi delle interfacce tramite `,` (virgole) o `;` (punti e virgola). Questo è il motivo per cui la lista appena elencata è così lunga.

```
{
  "defaultSeverity": "error",
  "extends": [
    "tslint:recommended"
  ],
  "rules": {
    "no-console": false,
    "interface-name": false,
    "member-ordering": false,
    "ordered-imports": false,
    "quotemark": [true, "single"],
    "member-access": [true, "no-public"],
    "arrow-parens": [true, "ban-single-arg-parens"]
  },
  "rulesDirectory": [],
  "linterOptions": {
    "exclude": [
      "config/**/*.js",
      "node_modules/**/*.ts",
      "coverage/lcov-report/*.js"
    ]
  }
}
```

Listing 3: tslint.json

Per agevolare gli sviluppatori nel rispettare le convenzioni scelte dal gruppo *DStack*, è stato definito un file **tslint.json**, interpretabile dallo strumento di *analisi statica*$_{\mathbf{G}}$ *tslint* e riportato in Listing 3.

[6] Per un approfondimento sulle stringhe template, vedasi https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals

2.2.5.12 Convenzioni per la scrittura di codice in SQL

Le seguenti convenzioni sono atte a rendere il codice che definisce le tabelle, le viste, e le query quanto più leggibili possibile. Il gruppo *DStack* preferisce quindi una miglior chiarezza del codice, caratteristica che ne migliora l'estendibilità e la manutenibilità, rispetto alla compattezza dello stesso. Le convenzioni individuate sono le seguenti:

- Tutte le parole riservate del linguaggio **SQL**, come ad esempio `SELECT`, `FROM`, `AS` e `JOIN` devono essere scritte in maiuscolo;
- tutte le aperture di parentesi devono essere precedute da un singolo spazio;
- tutti i vincoli, come ad esempio quelli di univocità, di chiave primaria o di chiave esterna, devono essere definiti tramite istruzione `CONSTRAINT` e devono possedere un nome univoco;
  - I vincoli di **chiave primaria** dovranno rispettare la seguente nomenclatura:

    `$TABLE_pkey`, dove $TABLE indica il nome della tabella corrente.

  - I vincoli di **chiave esterna** dovranno rispettare la seguente nomenclatura:

    `$TABLE_fkey`, dove $TABLE indica il nome della tabella corrente.

  - I vincoli di **univocità** dovranno rispettare la seguente nomenclatura:

    `$VAL_[$VAL_]*unique`, dove $VAL indica ognuna delle colonne coinvolte nel vincolo.

- nessuna istruzione deve superare le 100 righe;
- costrutti come `UNIQUE`, `DEFAULT` e `KEY` devono trovarsi ad un livello di indentazione aggiuntivo rispetto alla precedente riga;
- le funzioni di utilità o aggregazione che differiscano dallo standard SQL, offerte dal database scelto, devono essere scritte in minuscolo;
- tutti i nomi di tabelle, viste temporanee, tipi personalizzati e colonne devono essere scritte in minuscolo, e possono contenere il carattere underscore (_) al posto degli spazi;
- ogni colonna coinvolta in un'istruzione di `SELECT`, `JOIN`, `WHERE` deve essere riportata su tante righe allineate quante sono le colonne citate;
- ogni colonna coinvolta in un'aggregazione come `GROUP BY` o `SUM`, deve essere riportata su tante righe allineate quante sono le colonne citate;
- nel caso di concatenazioni logiche (`AND`, `OR`, `NOT`) nelle condizioni di `JOIN` o di `WHERE`, ognuna di esse deve essere riportata in una nuova riga;
- le indentazioni devono essere a 4 spazi per ogni livello;

```
CREATE SEQUENCE public.x_service_event_type_id_seq;
CREATE TABLE public.x_service_event_type (
    x_service_event_type_id BIGINT NOT NULL
        DEFAULT nextval ('public.x_service_event_type_id_seq'),
    service_id BIGINT NOT NULL,
    event_type_id BIGINT NOT NULL,
    CONSTRAINT x_service_event_type_pkey
        PRIMARY KEY (x_service_event_type_id),
    CONSTRAINT x_service_id_event_type_id_unique
        UNIQUE (service_id, event_type_id),
    CONSTRAINT x_service_event_type_service_fkey
        FOREIGN KEY (service_id)
        REFERENCES public.service (service_id),
    CONSTRAINT x_service_event_type_event_type_fkey
        FOREIGN KEY (event_type_id)
        REFERENCES public.event_type (event_type_id)
);
```

Figura 27: Esempio di definizione di tabella SQL che rispetta la convenzioni decise.

- ci deve essere uno spazio tra i due termini delle operazioni di uguaglianza, disuguaglianza o inclusione;
- ogni subquery o vista temporanea deve rispettare l'indentazione a livelli precedentemente descritta;
- l'uso di alias per referenziare tabelle coinvolte in una query è preferibile, ma è a discrezione dello sviluppatore, che dovrà comunque garantire la leggibilità del codice scritto;
- nel caso in cui una query coinvolga più tabelle, per ogni colonna citate nell'istruzione di `SELECT` dev'essere esplicitata la tabella di riferimento.

Per un esempio di definizione di tabella SQL che rispetti le convenzioni appena elencate, si veda la figura 27. Per un esempio di definizione di query SQL che rispetti le convenzioni appena elencate, si veda la figura 28.

2.2.5.13 Nomi di classi, metodi, variabili, tipi enumerabili e tabelle

I nomi devono essere quanto più possibile espliciti nel loro intento, ma anche sufficientemente brevi. In questo caso, la decisione sulla brevità dei nomi è lasciato al buon senso degli sviluppatori.

Le convenzioni sulla formattazione dei nomi variano leggermente a seconda del linguaggio considerato, e si basano sulle convenzioni comunemente adottate dalle community di tali linguaggi, al fine di agevolare chi andrà a visionare il codice sorgente. Sono presentate di seguito:

- **TypeScript** e **Java**: classi, interfacce e i nomi dei tipi enumerabili in *PascalCase*$_{\mathbf{G}}$, metodi e variabili in *lowerCamelCase*$_{\mathbf{G}}$, valori dei tipi enumerabili in *UPPER_SNAKE_CASE*$_{\mathbf{G}}$;
- **SQL**: tabelle, colonne e nomi dei tipi enumerabili in *lower_snake_case*$_{\mathbf{G}}$, *UPPER_SNAKE_CASE*$_{\mathbf{G}}$ per le viste temporanee e valori dei tipi enumerabili;
- **ENV**: *UPPER_SNAKE_CASE*$_{\mathbf{G}}$ per la definizione di variabili d'ambiente.

```
KEYWORDS AS (
    SELECT k.keyword,
        u.user_id,
        u.priority
    FROM public.keywords k
    JOIN USERS_BY_PROJECT_URL_MATCH u
        ON u.user_id = k.user_Id
),
KEYWORDS_GROUPED AS (
    SELECT k.keyword,
        array_agg(
            json_build_object(
                'user_id', k.user_id,
                'priority', k.priority
            )
        ) AS user_arr
    FROM KEYWORDS k
    GROUP BY k.keyword
),
```

Figura 28: Esempio parziale di query SQL che rispetta la convenzioni decise.

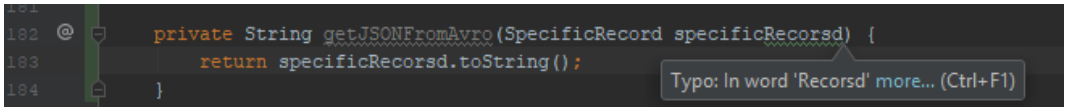


Figura 29: Esempio di individuazione di errore ortografico da parte di IntelliJ IDEA

```
CREATE TYPE public.producer_service AS ENUM (
  'REDMINE',
  'GITLAB',
  'SONARQUBE'
);
```

Figura 30: Esempio di definizione di tipo enumerabile in SQL

Le abbreviazioni dei nomi sono consentite solo in caso di sigle o unità di tempo, per ridurre il rischio di suscitare ambiguità. Le sigle dovranno inoltre essere rappresentate interamente in maiuscolo, a meno che non si trovino all'inizio di una dichiarazione di variabile o metodo. Ad esempio, nomi come `getJSONController` e `pollDurationMs` sono accettabili, mentre `magicN` e `getFirstV` non lo sono. Sono vietati nomi che presentano errori ortografici al loro interno; tale problema non si dovrebbe presentare poiché gli IDE selezionati dispongono di strumenti in grado di evidenziare identificatori di variabili con errori.

Sono vietate le dichiarazioni di variabili globali.

2.2.6 Strumenti primari

Ogni strumento elencato di seguito è gratuito, e, nella maggior parte dei casi, open source. Qualora lo strumento da adoperare non fosse open source, ciò sarà specificato nella descrizione.

2.2.6.1 Editors

- ***IntelliJ IDEA*$_{\mathbf{G}}$**: *IDE*$_{\mathbf{G}}$ per lo sviluppo dei servizi in *Java*$_{\mathbf{G}}$ e per la configurazione dei plugin e delle dipendenze di *Maven*$_{\mathbf{G}}$. Il gruppo ha deciso di adoperare la versione Community 2018.1 (gratuita ma non *open source*$_{\mathbf{G}}$) poiché, rispetto al più famoso IDE open source *Eclipse*$_{\mathbf{G}}$:

```
CREATE TABLE public.x_user_project (
  user_id BIGINT NOT NULL,
  project_id BIGINT NOT NULL,
  CONSTRAINT x_user_project_pkey PRIMARY KEY (user_id, project_id),
  CONSTRAINT x_user_project_user_fkey
    FOREIGN KEY (user_id)
    REFERENCES public.user (user_id),
  CONSTRAINT x_user_project_project_fkey
    FOREIGN KEY (project_id)
    REFERENCES public.project (project_id)
);
```

Figura 31: Esempio di definizione di tabella in SQL

```
KEYWORDS_GROUPED AS (
    SELECT k.keyword,
        array_agg(
            json_build_object(
                'user_id', k.user_id,
                'priority', k.priority
            )
        ) AS user_arr
    FROM KEYWORDS k
    GROUP BY k.keyword
),
```

Figura 32: Esempio di definizione di tabella temporanea in SQL

```
export default class HealthManager implements HealthMetrics {
  metrics(): Metrics {
    const uptime = os.uptime();
    const platform = os.platform().toString();
    const freeMemory = os.freemem();

    return {
      freeMemory,
      platform,
      uptime,
    };
  }
}
```

Figura 33: Esempio di definizione di classe in TypeScript

```
export interface Metrics {
  uptime: number;
  platform: string;
  freeMemory: number;
}
```

Figura 34: Esempio di definizione di interfaccia in TypeScript

```
export enum ThirdPartyContactService {
  TELEGRAM = 'TELEGRAM',
  SLACK = 'SLACK',
  EMAIL = 'EMAIL',
}
```

Figura 35: Esempio di definizione di tipo enumerabile in TypeScript

- offre una migliore indicizzazione dei file di progetto, garantendo una ricerca più performante all'interno dei file di lavoro;
- dispone di un'interfaccia grafica più intuitiva e, a nostro parere, più funzionale;
- come Eclipse, offre nativamente un'integrazione con sistemi di controllo della versione come *Git*$_{\mathbf{G}}$ o con build systems quali *Maven*$_{\mathbf{G}}$ e *Gradle*$_{\mathbf{G}}$;
- come per Eclipse, esistono molteplici plugin sviluppati dalla community, che consentono un'integrazione con strumenti quali, ad esempio, Checkstyle e Dockerfile.

- ***Visual Studio Code***$_{\mathbf{G}}$: *Editor*$_{\mathbf{G}}$ per lo sviluppo del Gestore Personale, per la scrittura di file SQL e per redirigere tutti gli script di configurazione (file Bash, JSON e YAML), ad eccezione di quelli XML, che sono relativi a Java e quindi dovranno essere modificati da IntelliJ IDEA. Negli ultimi 2 anni Visual Studio Code di Microsoft è diventato *de facto* l'editor preferito dalla community di sviluppatori *Node.js*$_{\mathbf{G}}$. La sua integrazione nativa con TypeScript e Git, le performance della sua indicizzazione dei file e la sua ricca rete di plugin hanno comportato la scelta di questo editor. I fattori principali che hanno comportato la scelta di questo editor sono:
  - la sua elevata popolarità, garantita anche dai frequenti aggiornamenti rilasciati dal colosso Microsoft e da un'affezionata community;
  - la sua integrazione nativa con TypeScript e Git;
  - le performance nell'indicizzazione dei file e i potenti strumenti di ricerca;
  - la sua ricca rete di plugin (sia ufficiali, sia sviluppati dalla community);
  - il fatto che molti membri del team abbiano già adoperato questo editor per progetti personali o universitari.

2.2.6.2 Test Frameworks

- ***JUnit***$_{\mathbf{G}}$: JUnit è il più diffuso framework *open source*$_{\mathbf{G}}$ per scrivere test d'unità in Java. Esso offre molte utilità e annotazioni Java che consentono di definire asserzioni sul codice in maniera pratica e concisa.
- ***Jest***$_{\mathbf{G}}$: È una suite di test *open source*$_{\mathbf{G}}$ per JavaScript, sviluppata da Facebook.

2.2.6.2.1 HTTP Frameworks

- ***Koa*$_{\mathbf{G}}$**: Microframework HTTP per *Node.js*$_{\mathbf{G}}$ creato dagli stessi sviluppatori di *ExpressJS*$_{\mathbf{G}}$, che ad oggi risulta essere il framework HTTP più usato in *Node.js*$_{\mathbf{G}}$. Dopo un primo confronto tra i principali framework per realizzare *API*$_{\mathbf{G}}$ *REST*$_{\mathbf{G}}$ in *Node.js*$_{\mathbf{G}}$, la scelta è ricaduta su *Koa*$_{\mathbf{G}}$, perché:
  - offre nativamente un'interfaccia asincrona basata sulle *Promise*$_{\mathbf{G}}$ per la gestione del controllo di flusso;
  - delega le funzioni marginali a plugin ufficiali o sviluppati dalla community, piuttosto che includerle di default nel *package*$_{\mathbf{G}}$ principale;
  - in media, a parità di funzionalità esposte da un'applicazione HTTP e di carico, ha risultati migliori nei benchmark;
  - Koa è stato sviluppato a partire dalle consapevolezze e dalle conoscenze derivate dalla scrittura del framework ExpressJS, avvenuta in anni precedenti: la sua architettura è stata infatti pensata per migliorare o eliminare alcuni difetti di ExpressJS.

- ***Spark*$_{\mathbf{G}}$**: Microframework HTTP che supporta i linguaggi Java e Kotlin, il cui nome non deve essere confuso con l'omonimo Apache Spark, che adempie a scopi totalmente diversi e non trova utilizzo nel progetto Butterfly. Rispetto a framework più famosi come *Spring Boot*$_{\mathbf{G}}$, Spark è stato scelto perché la sua sintassi è particolarmente simile a quella di framework HTTP per *Node.js*$_{\mathbf{G}}$ come *ExpressJS*$_{\mathbf{G}}$ e *KoaJS*$_{\mathbf{G}}$ (con cui parte del team ha già avuto modo di lavorare in passato), il che consente di abbattere i tempi richiesti per imparare ad usare questa libreria. Inoltre, al contrario di Spring Boot, Spark non adotta delle convenzioni particolari che impediscono una chiara comprensione del codice.

2.2.6.2.2 Database

- ***Postgres*$_{\mathbf{G}}$**: Database noto per essere "il più avanzato database relazionale open source" e per essere stato tra i primi *RDBMS*$_{\mathbf{G}}$ a supportare nativamente il tipo *JSON*$_{\mathbf{G}}$. Ai fini della scelta del database più adatto agli scopi del Gestore Personale, unico servizio di Butterfly che necessita di una persistenza permanente dei dati, non sono stati considerati database a pagamento come *IBM DB2*$_{\mathbf{G}}$ e *Oracle Database*$_{\mathbf{G}}$. In un primo momento, si è valutata la possibilità di utilizzare database non relazionali, come ad esempio *MongoDB*$_{\mathbf{G}}$, che offrono come vantaggio la possibilità di definire schemi che possono evolvere facilmente nel tempo. Quest'opzione è stata scartata, poiché l'unica entità che si prevede possa fortemente evolvere nel tempo è quella relativa al record di una segnalazione, di cui comunque non è richiesto un salvataggio a database. Le altre entità, come utenti, progetti, parole chiave a cui un utente è interessato in un determinato progetto, associazione tra un utente e uno o più sistemi di contatto, sono intuitivamente esprimibili in forma relazionale. Inoltre, sistemi relazionali come *Postgres* offrono la possibilità di definire indici sulle colonne coinvolte in operazioni condizionali, come ad esempio quelle di `WHERE` e `JOIN`, permettendo di evitare scansioni complete della tabella e garantendo performance eccellenti anche in caso di milioni di record salvati. Infine, in questo caso è preferibile adottare il paradigma relazionale anche perché il team di *DStack*

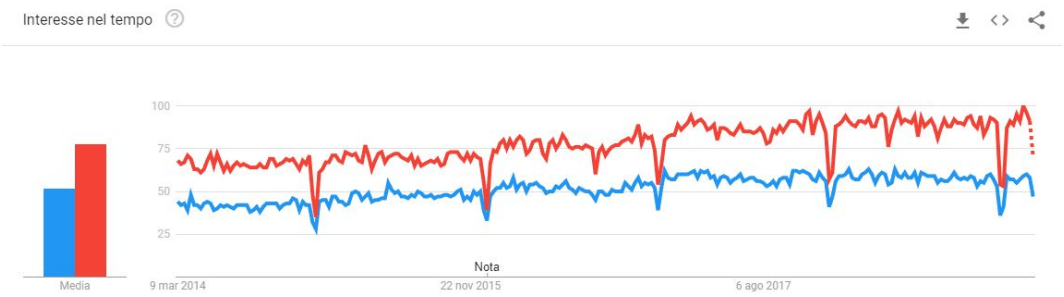


Figura 36: Confronto tra l'interesse ricevuto da Postgres (in rosso) e l'interesse ricevuto da MongoDB (in blu) negli ultimi 5 anni. Fonte: Google Trends

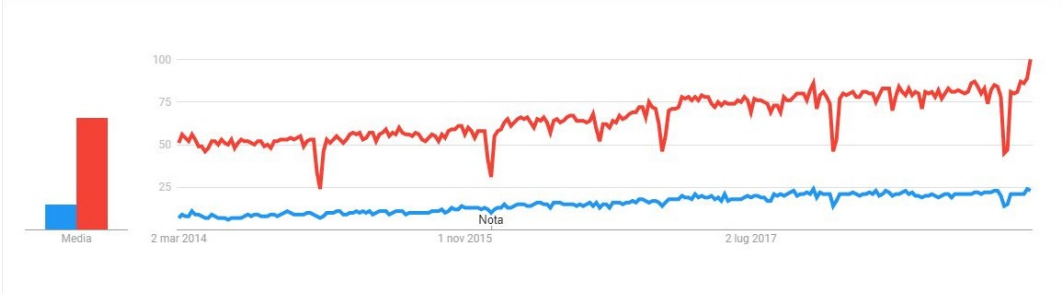


Figura 37: Confronto tra l'interesse ricevuto da Postgres (in rosso) e l'interesse ricevuto da MariaDB (in blu) negli ultimi 5 anni. Fonte: Google Trends

ritiene che Butterfly richieda già numerose nuove tecnologie da studiare.

Rispetto a *MariaDB*$_{\mathbf{G}}$ (database che ogni membro del team ha avuto modo di conoscere durante il corso di Basi di Dati), che ha introdotto alcune funzionalità come *Common Table Expressions*$_{\mathbf{G}}$, *Window Functions*$_{\mathbf{G}}$ e il supporto a *JSON*$_{\mathbf{G}}$ solo di recente, Postgres supporta tali funzionalità da più anni, con conseguente maggior documentazione online e una community che è più probabilmente in grado di rispondere ai dubbi degli sviluppatori su tali argomenti. In generale, tuttavia, si può dire che la scelta di utilizzare *Postgres* per il progetto Butterfly rispetto a *MariaDB* sia principalmente attribuibile al maggior interesse che questo database ha ricevuto negli ultimi 5 anni, e al fatto che aziende con milioni di utenti come *Instagram*, *Spotify* e *Netflix* lo usino.

2.2.6.2.3 Package Manager

- ***Maven***$_{\mathbf{G}}$: rispetto all'altro famoso package manager per Java, *Gradle*$_{\mathbf{G}}$, Maven è già stato utilizzato da buona parte del gruppo DStack durante il corso opzionale di Tecnologie open source, e quindi ne ha già acquisito una modesta confidenza, sufficiente a ridurre i tempi necessari per apprendere le tecnologie utili al progetto da realizzare. Rispetto a Gradle, il cui sviluppo è stato molto più recente e la cui adozione da parte della community Java sembra stia avvenendo piuttosto lentamente, il web pullula di tutorial e riferimenti a Maven, soprattutto per quanto concerne la configurazioni dei task che devono essere

svolti durante le varie fasi di build. L'adozione di Maven è stata quindi una scelta naturale e unanime.

- ***yarn*$_{\mathbf{G}}$**: si tratta di un package manager alternativo ad *npm*$_{\mathbf{G}}$, package manager di default per *Node.js*$_{\mathbf{G}}$. I vantaggi di yarn su npm sono:
  - supporta il download parallelo delle dipendenze di terze parti, il che consente di evitare alcuni colli di bottiglia presenti in npm e generalmente rende più veloce la fase di installazione;
  - sfrutta meglio la cache locale dei moduli;
  - offre un sistema per il lock delle versioni più efficace rispetto a quello di npm;
  - non richiede di imparare nuove convenzioni: le dipendenze installate sono listate in un file *package.json* esattamente come se fossero installate da npm;
  - è sviluppato e mantenuto da Facebook, e i nuovi rilasci sono frequenti.

2.2.6.2.4 Strumenti usati per svolgere tracciamento

- ***PragmaDB*$_{\mathbf{G}}$**:software di tracciamento dei requisiti sviluppato come progetto di Ingegneria del Software dal gruppo Pragma (AA 2014/2015). Le funzionalità offerte dall'applicativo sono:
  - Tracciamento dei requisiti;
  - Tracciamento dei casi d'uso;
  - Tracciatori degli attori del sistema;
  - Tracciamento delle fonti;
  - Tracciamento dei termini di glossario
  - Tracciamento dei packages;
  - Tracciamento delle classi;
  - Tracciamento delle metriche;
  - Tracciamento dei test.

3 Processi di supporto

3.1 Documentazione

3.1.1 Scopo

In questa sezione verranno descritte le regole e gli standard che il gruppo *DStack* s'impegnerà a seguire durante le fasi di scrittura della documentazione interna ed esterna.
Il processo di documentazione è strutturato nel seguente modo:

- Suddivisione dei documenti;
- Struttura dei documenti;
- Produzione;
- Mantenimento.

3.1.2 Suddivisione dei documenti

I documenti redatti dal gruppo saranno divisi in due categorie:

- **Informali**: tutti i documenti che sono in fase di redazione, validazione oppure che devono ancora essere stati approvati dal *Responsabile di Progetto*. Saranno da ritenersi informali anche i documenti non versionati redatti per utilità interna al gruppo;
- **Formali**: tutti i documenti *versionati*$_{\mathbf{G}}$ approvati dal *Responsabile di progetto* e pronti per essere distribuiti. Ogni volta che un documento formale viene modificato, la nuova versione è da considerarsi informale fino alla sua approvazione. In ogni istante possono esistere più documenti formali dello stesso tipo con versioni differenti, tuttavia si fa sempre riferimento a quello con la versione maggiore e le restanti serviranno solo come archivio storico.

I documenti formali saranno suddivisi in due ulteriori categorie:

- **Interni**: documenti che hanno un utilizzo interno al gruppo *DStack*;
- **Esterni**: documenti che verranno condivisi con la Proponente e i Committenti.

Tra i documenti interni rientrano:

- ***Verbale Interno***: documento che contiene tutte gli argomenti affrontati e le scelte prese durante l'incontro interno;
- ***Norme di Progetto***: documento nel quale verranno redatte le norme e gli standard che il gruppo *DStack* adotterà durante le attività di sviluppo del software;
- ***Studio di Fattibilità***: documento nel quale verranno indicati i punti di forza e quelli deboli che il gruppo *DStack* ha riscontrato nei diversi capitolati seguiti da una motivazione sul perché il capitolato è stato tenuto in considerazione oppure è stato scartato.

Tra i documenti esterni rientrano:

- ***Verbale Esterno***: documento che contiene tutti gli argomenti affrontati durante l'incontro esterno;
- ***Analisi dei Requisiti***: documento nel quale verranno descritti i requisiti del progetto. Conterrà i diagrammi d'interazione dell'utente e l'analisi dei casi d'uso relativi al prodotto;
- ***Piano di Progetto***: documento nel quale verrà descritto come il gruppo *DStack* gestirà le risorse umane e di tempo;
- ***Piano di Qualifica***: documento nel quale verranno descritti gli standard che il gruppo *DStack* seguirà per soddisfare i requisiti di qualità del progetto;
- ***Glossario***: documento nel quale verranno specificati i termini che possono indurre il lettore ad ambiguità oppure che possono essere di difficile comprensione.

3.1.2.1 Nomenclatura dei documenti

I documenti interni ed esterni, esclusi i verbali[7], adotteranno la seguente nomenclatura:

NomeDelDocumento_vX.V.Z

- **NomeDelDocumento**: rappresenta il nome del documento al quale si fa riferimento. Nel caso il nome dovesse contenere più parole, non si fa uso di spazi e la lettera iniziale di ogni parola deve essere scritta in maiuscolo;
- **vX.V.Z**: rappresenta l'ultima versione del documento[8], è separata dal nome tramite l'uso del carattere *underscore* "_".

Formato: durante il suo ciclo di vita il file verrà salvato nel formato *.tex*$_{\mathbf{G}}$. Solamente quando il file passerà dallo stato "Verificato" ad "Approvato" verrà prodotto il file in formato *pdf*$_{\mathbf{G}}$.

3.1.3 Struttura dei documenti

3.1.3.1 Template

Ogni documento dovrà essere creato utilizzando il template LaTeX presente nel repository privato all'indirizzo `https://github.com/jkomyno/swe-DStack/tree/master/template`. Utilizzare un template univoco per tutti i documenti permette di centralizzare le scelte stilistiche in pochi file. In questo modo ogni modifica ricade automaticamente su tutti i documenti evitando di dover effettuare manualmente gli stessi cambiamenti del codice sorgente per ognuno di essi.
In caso di necessità ogni membro del gruppo potrà richiedere all'*Amministratore* delle modifiche al template. Per ogni cambiamento apportato a quest'ultimo si dovranno informare tutti i membri del gruppo.

[7] vedi §3.1.3.3
[8] Il formato utilizzato per tracciare le versioni è descritto in §3.1.3.6

3.1.3.2 Struttura documenti formali

Ogni documento formale prodotto dovrà avere una pagina di apertura composta da:

- **Logo del gruppo**: sotto il quale deve comparire l'email per contattare il gruppo;
- **Titolo del documento**: contiene, appunto, il titolo del documento;
- **Informazioni sul documento**: contenente i seguenti elementi
  - Nome del documento, ripete il titolo del documento;
  - Versione del documento (omesso per i verbali);
  - Data approvazione, il giorno in cui il documento è stato approvato;
  - Responsabili, chi ha approvato la versione formale del documento;
  - Redattori, quali membri del gruppo hanno contribuito alla stesura del documento;
  - Verificatori, quali membri del gruppo hanno verificato il documento;
  - Stato, ossia lo stato corrente del documento;
  - Lista distribuzione, lista di persone alle quali deve essere pervenuto il documento;
  - Uso, specifica se il documento ha visibilità interna o esterna.
- **Sommario**: breve descrizione degli argomenti trattati nel documento.

La pagina seguente dovrà contenere:

- **Diario delle modifiche**: ossia una tabella che riporti le varie modifiche apportate al documento. La tabella è ordinata in modo decrescente dove la prima riga contiene le informazioni riguardanti le ultime modifiche. Ogni riga contiene le seguenti informazioni:
  - **Versione**[9]: il numero di versione del documento in cui è stata applicata la modifica;
  - **Descrizione**: breve descrizione dei cambiamenti apportati. Per indicare un capitolo o una sezione aggiunta o modificata viene usato un riferimento numerico preceduto dal simbolo §;
  - **Nominativo**: chi ha eseguito la modifica specificata nel campo precedente;
  - **Ruolo**: ossia il ruolo della persona che ha eseguito la modifica;
  - **Data**[10]: quando la modifica è stata effettuata.

  La terza pagina sarà composta da tre **indici** formati da:
  - Elenco dei capitoli e delle sezioni;
  - Elenco delle immagini presenti, se necessario;

[9] Il formato utilizzato è specificato in §3.1.3.6
[10] Il formato utilizzato è specificato in §3.1.3.4.3

- Elenco delle tabelle presenti, se necessario (escludendo il diario delle modifiche).

Subito dopo si estende il corpo del documento. In ogni pagina dovrà comparire:

- **Intestazione**: contenente a sinistra il logo del gruppo mentre a destra il nome della sezione corrente del documento;
- **Contenuto**: avente il contenuto effettivo della pagina;
- **Piè di pagina**: contenente il nome del documento con la relativa versione a sinistra, mentre a destra il numero di pagina effettiva del documento. Inoltre verranno aggiunte le fonti ed i riferimenti nel caso dovessero essere presenti sulla medesima pagina delle citazioni o riferimenti ad altri documenti o al documento stesso.

3.1.3.3 Struttura verbale di riunione

Il verbale di riunione è un documento ufficiale che raccoglie le principali informazioni relative ad una specifica riunione (interna o esterna), ed espone i punti salienti degli argomenti trattati durante l'incontro.
La nomenclatura dei verbali è la seguente:

Verbale[I/E]-[#incontro]-AAAA_MM_GG

dove:

- **I**: indica un verbale relativo ad una riunione interna;
- **E**: indica un verbale relativo ad una riunione esterna;
- **#incontro**: indica il numero della riunione alla quale il verbale fa riferimento;
- **AAAA_MM_GG**: la data, espressa nel formato descritto in §3.1.3.4.3.

Un verbale di riunione sarà così strutturato:

- **Verbale [Tipo] [Data]**: questa dicitura sarà inclusa nel frontespizio, e riporterà il **[Tipo]** del verbale (**I**nterno o **E**sterno), e la **[Data]** nella quale si è tenuta la riunione;
- **Informazioni sul documento**: un elenco delle informazioni principali sulla riunione:
  - **Luogo della riunione**: indica in maniera dettagliata il luogo nel quale la riunione si è tenuta, ed è espressa nel formato *Indirizzo NumeroCivico, CAP Città (Provincia)*;
  - **Ora di inizio**[11];
  - **Ora di fine**[12];
  - **Segretario**: indicato secondo il formato *Nome Cognome*, come stabilito in §3.1.3.4.3;

[11] Formato definito in §3.1.3.4.3
[12] Formato definito in §3.1.3.4.3

  - **Partecipanti**: ordinati alfabeticamente per nome.

- **Ordine del giorno**: una lista puntata che esprime gli argomenti trattati durante la riunione;
- **Resoconto della Riunione**: ad ogni elemento appartenente alla lista dell'ordine del giorno viene associata una descrizione con informazioni aggiuntive sulla discussione: le possibili scelte vagliate, la decisione presa e le motivazioni che l'hanno indotta;
- **Tracciamento delle decisioni**: riassunto delle decisioni prese dal gruppo nel corso di un incontro interno o esterno. Ogni decisione è rappresentata da un *Codice identificativo* espresso nella forma:

  V[Tipo]-[Data]-[Numero]

  - **V**: valore statico che indica una decisione a verbale;
  - **Tipo**: rappresentato con **I** se il verbale fa riferimento ad una riunione interna o **E** esterna;
  - **Data**: indica la data di verbale, espressa nella forma AAAA-MM-GG, descritto in §3.1.3.4.3;
  - **Numero**: codice numerico progressivo della decisione.

  Seguito da una *Descrizione* che traduce in linguaggio naturale il significato del codice appena espresso.

3.1.3.4 Norme tipografiche

Al fine di evitare incongruenze tra i vari file, di seguito si specificano le norme riguardanti l'ortografia, la tipografia e l'assunzione di uno stile uniforme in tutti i documenti.

3.1.3.4.1 Stile del testo

- **Grassetto**: da utilizzare per i titoli, sottotitoli, il nome degli oggetti negli elenchi puntati (seguiti da una descrizione), intestazioni delle tabelle e le parole ritenute importanti dal *Redattore*;
- **Corsivo**: da utilizzare per il nome del proponente, nomi di altri documenti, ruoli, termini specifici, termini del glossario, *path*$_{\mathbf{G}}$ (sia locali che remoti), didascalie delle immagini e le citazioni;
- **Maiuscolo**: da utilizzare solo per gli acronimi;
- **Colore del testo**: per tutti i documenti verrà utilizzato il colore nero. Solo per i link a siti internet verrà utilizzata una colorazione blu.

3.1.3.4.2 Composizione del testo

- **Elenchi puntati**: l'oggetto dell'elenco sarà in grassetto ed avrà la prima lettera maiuscola. L'oggetto può essere accompagnato da una descrizione, in questo caso si utilizzano i due punti ":" dopo l'oggetto. Ogni paragrafo deve finire con un punto e virgola ";" tranne l'ultimo che deve finire con un carattere punto ".". In caso di due livelli verranno utilizzati per il primo livello una pallina nera "•" mentre per il secondo livello il simbolo di negazione "-";
- **Note a piè pagina**: ogni nota a piè pagina dovrà iniziare con un riferimento numerico in apice seguito dalla prima lettera maiuscola della prima parola del paragrafo. Ogni nota a piè pagina dovrà terminare con un punto.
- **Punteggiatura**: a sinistra del simbolo di punteggiatura dovrà esserci un carattere alfanumerico e a seguire uno spazio;
- **Parentesi**: verranno utilizzate le parentesi tonde per descrivere un esempio. Devono iniziare e finire con un carattere alfanumerico;
- **Doppie virgolette**: verranno utilizzate le doppie virgolette per specificare l'uso di simboli.

3.1.3.4.3 Formati ricorrenti

- **Data**: scritta con il formato AAAA-MM-GG dove AAAA rappresenta l'anno, MM il mese e GG il giorno. Nel caso in cui il mese e/o il giorno dovessero essere di una singola cifra si deve anteporre uno zero ad essa;
- **Ora**: scritta con il formato HH:MM dove HH rappresenta le ore mentre MM i minuti. Nel caso in cui le ore e/o i minuti dovessero essere di una singola cifra si deve anteporre uno zero ad essa;
- **Path**: per gli indirizzo web completi e indirizzi mail si utilizzerà il seguente comando `indirizzo`;
- **Ruoli di progetto**: i vari ruoli di progetto andranno formattati utilizzando la prima lettera maiuscola per ogni parola che non sia di preposizione (es: Responsabile di Progetto). I ruoli potranno essere contratti tralasciando "di Progetto" quando il contesto renderà non ambigua la contrazione;
- **Nomi propri**: i nomi propri di persona verranno scritti riportando prima il nome e subito dopo il cognome. Per i nomi propri più utilizzati è stato creato anche un comando LaTeX personalizzato: `\ncognome` (prima lettera nome e cognome completo) che visualizzerà il nome seguito dal cognome nel formato corretto;
- **Nome del gruppo**: il nome del gruppo *DStack* verrà scritto con la prime due lettere maiuscole. É stato creato anche un comando LaTeX personalizzato: `\authorName` che visualizzerà il nome del gruppo;
- **Nome del proponente**: ci si riferirà al proponente con "*Imola Informatica S.P.A*" oppure con "proponente". É stato creato un comando LaTeX personalizzato: `\proposerName` che visualizzerà il nome del proponente;

- **Nome del progetto**: ci si riferirà al progetto con "**Butterfly**" in grassetto;
- **Nomi dei documenti**: i nomi dei documenti di progetto andranno formattati utilizzando la prima lettera maiuscola per ogni parola che non sia di preposizione (es: Norme di Progetto);
- **Citazioni a documenti interni**: devono essere scritte usando il *corsivo* seguiti da un apice;
- **Riferimenti a documenti esterni**: il testo di riferimento deve essere inserito tra doppie parentesi angolari "« »" seguiti da un apice;
- **Riferimenti al software LaTeX**: verrà utilizzato il comando `\LaTeX`;
- **Riferimenti a codice di programmazione**: verrà utilizzato il carattere `monospace`.

3.1.3.5 Elementi grafici

- **Immagini**: devono essere in formato $PDF_{\mathbf{G}}$, $PNG_{\mathbf{G}}$ o $JPG_{\mathbf{G}}$, devono possedere una didascalia sotto di essa e devono avere un'interlinea di distanza sia sopra che sotto. L'immagine può affiancare il testo solo se la dimensione lo permette;
- **Tabelle**: verranno eventualmente utilizzate delle tabelle con lo scopo di rendere più chiare e leggibili alcune parti. Verrà utilizzata una scala di grigio tra una riga e l'altra per rendere più semplice la lettura della tabella. Inoltre ogni tabella deve essere fornita di didascalia in cui dovrà necessariamente comparire un nome identificativo utile per la sua tracciabilità all'interno dl testo.

3.1.3.6 Codice di versione

In ogni documento formale interno, esterno esclusi i verbali saranno sempre seguiti da un codice di versione rappresentato nel seguente formato:

vX.Y.Z

Dove:

- **X**: rappresenta la versione di rilascio. Il numero verrà incrementato quando il documento è pronto per la revisione alla quale il gruppo ha deciso di sottoporre il documento;
- **Y**: rappresenta un rilascio medio. Il numero verrà incrementato ogni volta che il documento verrà approvato dal verificatore;
- **Z**: rappresenta un rilascio minore. Il numero verrà incrementato ogni volta che ci sarà una modifica del documento da parte di uno dei redattori.

Per il corretto utilizzo del formalismo si dovranno seguire le seguenti normative:

- Il nome dei documenti approvati dovranno avere il seguente formato: **NomeDelDocumento_vX.Y.Z.estensione**;
- Quando si nomina un documento all'interno di un altro, sia nel corpo che nel piè di pagina, il formato deve essere: **Nome del Documento vX.Y.Z** .

3.1.4 Produzione

3.1.4.1 Gestione termini di glossario

Dopo la fase di verifica, il verificatore dovrà utilizzare il seguente script per verificare la presenza di termini di Glossario:

parse_glossary.py

Esso inserisce in un documento di testo tutte le parole che sono da inserire successivamente nel glossario.

Per utilizzare questo script bisogna eseguire i seguenti step:

- Aprire una console di comando;
- Spostarsi nella cartella contenente lo script;
- Eseguire il seguente comando: `python3 parse_glossary.py DIRECTORY_NAME` dove "DIRECTORY_NAME" indica la cartella contenente i documenti da analizzare.

3.1.4.2 Strumenti

3.1.4.2.1 Stesura documenti

Per la stesura dei documenti è stato scelto il linguaggio di markup *LaTeX*$_{\mathbf{G}}$. La scelta è stata immediata e motivata dal fatto che, rispetto alle alternative, esso permette una gestione molto più accurata di tabelle, immagini e riferimenti, nonostante a livello intuitivo sia meno comprensibile.
L'editor scelto è *Texmaker*$_{\mathbf{G}}$, selezionato poichè software open source, multi-piattaforma e integrante molti strumenti di ausilio alla scrittura e al controllo del testo.

3.1.4.2.2 Diagrammi UML

Per la realizzazione dei diagrammi UML si è scelto di utilizzare *Draw.io*$_{\mathbf{G}}$. Si tratta di un servizio open source online per la creazione di diagrammi UML e diagrammi di flusso che presenta una grande intuitività d'utilizzo.

3.1.4.2.3 Verifica ortografica

Per la verifica ortografica dei documenti scritti in LaTeX si utilizzerà lo strumento di correzione automatica presente nell'editor Texmaker oppure lo strumento *Aspell* (versione $>$ 0.60.5). Mentre il primo fornisce avvisi di errore istantaneamente il secondo dovrà essere utilizzato dalla shell lanciando i comando direttamente sul file in formato tex:

aspell –mode=tex –lang=it check NomeFile.tex

3.2 Gestione della configurazione

3.2.1 Scopo

La documentazione e le parti versionabili del progetto verranno gestite sulla piattaforma *GitHub* con uso della tecnologia *Git*$_{\mathbf{G}}$. Per i documenti informali e le parti non versionabili del progetto, il gruppo *DStack* ha optato per l'utilizzo della piattaforma di condivisione file *Google Drive*$_{\mathbf{G}}$.

3.2.2 Comandi base

Di seguito vengono riportati i comandi basilari, corredati ognuno da una breve descrizione, fondamentali per il corretto uso di Git all'interno del progetto:

- **git clone *[indirizzo repository]***: permette la creazione di una copia locale alla repository remota del progetto specificato;
- **git pull**: la repository locale viene aggiornata, sovrascrivendola con quella remota;
- **git status**: permette la visualizzazione dello stato attuale della repository locale, vengono indicati quali sono i file modificati, tracciati e non da Git;
- **git add nomeFile**: aggiunge il file specificato alla lista dei file tracciati da Git. È consentito l'utilizzo di una wildcard "*" che permette sia l'aggiunta di nuovi file sia le modifiche a file esistenti, che altrimenti non verrebbero apportate;
- **git commit -m "Descrizione modifiche effettuate"**: eseguito solo sui file precedentemente sottoposti ad *add*. Viene effettuato il commit delle modifiche effettuate salvandole nella repository locale. La *Descrizione delle modifiche effettuate* è obbligatoria e deve contenere il nome del file aggiunto/modificato seguito da una breve descrizione della modifica apportata;
- **git rm nomeFile**: rimuove un file sia dalla repository locale che dal versionamento, tuttavia il comando è utilizzabile solo in file in cui tutte le modifiche sono già state commitate;
- **git branch**: permette la visualizzazione dei branch della repository locale. Il comando accompagnato da *-a* permette la visualizzazione dei branch locali e remoti della repository. Il branch attuale viene evidenziato da un "*" precedente al nome;
- **git checkout**: permette di cambiare branch attivo della repository locale, tutti i comandi hanno effetto esclusivamente sul branch attivo;
- **git fetch**: scarica la versione aggiornata della repository, aggiornando esclusivamente i puntatori. Non viene eseguito automaticamente alcun merge;
- **git push**: aggiorna la repository remota con quella locale;
- **git merge**: la repository locale viene unita alla repository remota scaricata con l'uso del comando *fetch*.

3.2.3 Struttura della repository

Sono state create due repository indipendenti su cui sono state sviluppate le varie parti del progetto.

3.2.3.1 Repository documenti

È stata creata una repository specializzata a contenere la documentazione del progetto. Questa è strutturata nel seguente modo:

- **Documenti Esterni**: cartella contenente le seguenti sottocartelle:
  - Analisi dei Requisiti;
  - Glossario;
  - Piano di Progetto;
  - Piano di Qualifica;
  - Manuale Utente;
  - Manuale Sviluppatore;
  - Verbali.
- **Documenti Interni**: cartella contenente le seguenti sottocartelle:
  - Verbali;
  - Norme di Progetto;
  - Studio di fattibilità;
- **Template**: cartella contenente tutti i template LaTeX che il gruppo adotterà come descritto in §3.1.3.1.

Il contenuto dei Documenti Interni ed Esterni varia a seconda della Revisione da affrontare.

3.2.3.1.1 File inclusi e uso di .gitignore

Nelle directory principali sono contenuti solo i file con estensione *.tex*, *.pdf* e *.png*. I file, invece, che vengono generati dalla compilazione sono stati inclusi all'interno di un file *.gitignore* che ne permette l'esclusione automatica tramite l'ausilio di Git.

3.2.3.2 Repository codice

Come strumento di versionamento e salvataggio dei dati il gruppo *DStack* ha scelto di utilizzare un repository su *GitHub*$_{\mathbf{G}}$. La creazione del repository è compito dell'*Amministratore di Progetto*. Lo stesso *Amministratore di Progetto* deve assicurarsi che tutti i membri del team dispongano dei permessi di lettura e scrittura nel repository.

Vi saranno due repository che ospiteranno il codice:

1. **PoC**: è la repository per il prototipo da presentare come *Proof of Concept* durante la Tecnhology Baseline;
2. **Butterfly**: è la repository per il prodotto che sarà consegnato all'utente.

Essendo Butterfly un prodotto composto di più servizi interconnessi tra loro, si è deciso di optare per un approccio che preveda una singola repository anziché molteplici diverse, suddivendo invece i moduli diversi in sottocartelle.

La struttura ad alto livello del repository per il codice è la seguente:

- **butterfly**: contiene tutti i package *Java*$_{\mathbf{G}}$ che sono a contatto con Apache Kafka, come i Producer, i Consumer e il Dispatcher;
- **kafka**: contiene la definizione del *Dockerfile*$_{\mathbf{G}}$ di *Apache Kafka*$_{\mathbf{G}}$ e eventuali file di configurazione;
- **schema-registry**: contiene la definizione del *Dockerfile*$_{\mathbf{G}}$ di *Confluent Schema Registry*$_{\mathbf{G}}$ e eventuali file di configurazione;
- **third-party-services**: contiene le definizioni dei *Dockerfile*$_{\mathbf{G}}$ per eseguire le istanze locali di *Redmine*, *GitLab*, *Sonarqube*;
- **user-manager**: contiene il database e il server che espone le REST API del modulo relativo al Gestore Personale;
- **zookeeper**: contiene la definizione del *Dockerfile*$_{\mathbf{G}}$ di *Zookeeper*$_{\mathbf{G}}$ e eventuali file di configurazione;
- **README.md**: documentazione generale dei moduli scritta in *Markdown*$_{\mathbf{G}}$;
- **docker-compose.yml**: file di configurazione *Docker Compose*$_{\mathbf{G}}$ che avvia l'intero sistema tramite processi Docker connessi tra loro.

3.2.3.2.1 /butterfly

Dettaglio del modulo *butterfly*$_{\mathbf{G}}$:

- **avro-schemas**: package che contiene le strutture dati con cui saranno rappresentati i messaggi trasmessi o ricevuti da Kafka;
- **config**: package di utilità per gestire la configurazione interna;
- **consumer**: package per raccogliere classi di utilità per tutti i Consumer;
- **email-consumer**: package per il Consumer che, ricevuto un evento, informazioni su un utente e il suo riferimento email da Kafka, gli notifichi l'evento tramite email;
- **gitlab-producer**: package per il Producer che, ricevuto un evento da GitLab, ne estrae le informazioni più importanti per inviarle a Kafka;
- **middleware-dispatcher**: package per il Middleware Dispatcher;
- **producer**: package per raccogliere classi di utilità per tutti i Producer;
- **redmine-producer**: package per il Producer che, ricevuto un evento da Redmine, ne estrae le informazioni più importanti per inviarle a Kafka;
- **slack-consumer**: package per il Consumer che, ricevuto un evento, informazioni su un utente e il suo riferimento Slack da Kafka, gli notifichi l'evento tramite Slack;
- **sonarqube-producer**: package per il Producer che, ricevuto un evento da Sonarqube, ne estrae le informazioni più importanti per inviarle a Kafka;

- **telegram-consumer**: package per il Consumer che, ricevuto un evento, informazioni su un utente e il suo riferimento *Telegram*$_{\mathbf{G}}$ da Kafka, gli notifichi l'evento tramite Telegram;
- **README.md**: documentazione generale dei moduli in */butterfly* scritta in *Markdown*$_{\mathbf{G}}$;
- **build.sh**: script per compilare tramite *Docker*$_{\mathbf{G}}$ e *Maven*$_{\mathbf{G}}$ i package Java11 in file jar eseguibili.

3.2.3.2.2 /user-manager/user-manager-rest-api

Dettaglio del modulo *user-manager/user-manager-rest-api*$_{\mathbf{G}}$:

- **openapi**: cartella contenente la definizione *OpenAPI*$_{\mathbf{G}}$ delle REST API esposte dal servizio;
- **src**: cartella contenente il codice *TypeScript*$_{\mathbf{G}}$ da traspilare in *JavaScript*$_{\mathbf{G}}$;
- **test**: cartella per raccogliere tutti i test di questo sottomodulo;
- **.dockerignore**: definisce tutti i file da non copiare in *Docker*$_{\mathbf{G}}$ durante la fase di build;
- **.editorconfig**: file che definisce le regole di spaziatura e tabulazione usato dall'estensione *Editorconfig*$_{\mathbf{G}}$ dell'IDE scelto;
- **.gitignore**: file .gitignore locale;
- **.user-manager-rest-api.env**: file di configurazione che contiene la definizione delle variabili d'ambiente per questo sottomodulo;
- **Dockerfile**: *Dockerfile*$_{\mathbf{G}}$ per il sottomodulo corrente;
- **copyStaticAssets.js**: funzione di utilità per copiare i file *SQL* dalla cartella *src* alla cartella dei file generati *dist*, poiché non è un operazione gestita automaticamente dal compilatore di *TypeScript*$_{\mathbf{G}}$;
- **jest.config.js**: configurazione per la suite utilizzata per i test, *Jest*$_{\mathbf{G}}$;
- **package.json**: contiene l'elenco di tutti i moduli e dipendenze necessarie per far funzionare il Gestore Personale;
- **tsconfig.json**: contiene la definizione delle regole di compilazione di TypeScript;
- **tsconfig.test.json**: contiene la definizione delle regole di compilazione dei test scritti in TypeScript;
- **tslint.json**: contiene la definizione delle regole di analisi statica utilizzate da *tslint*$_{\mathbf{G}}$;
- **yarn.lock**: file autogenerato durante il processo di installazione delle dipendenze che permette a tutti gli sviluppatori di avere le stesse esatte versioni delle dipendenze installate;

3.2.4 Ciclo di vita di un branch

I branch principali verranno creati attraverso l'estensione *Git Flow*$_{\mathbf{G}}$.
Dopo la creazione di una *Issue*$_{\mathbf{G}}$, la persona incaricata è responsabile della creazione di un branch di feature il cui nome richiami il titolo della issue.
Al termine del lavoro, il verificatore creerà il branch di feature dedicato alla verifica del lavoro dei redattori. A verifiche ultimate, il responsabile accederà al branch del verificatore e, in caso di approvazione, effettuerà il merge con il ramo di *development*.
I redattori sono responsabili nell'eliminare i loro branch a seguito dell'approvazione dei documenti.
Il branch *master* verrà aggiornato attraverso le release in vista delle revisioni programmate.

3.2.5 Aggiornamento della repository

Nel caso il redattore o il verificatore non l'avessero ancora fatto, devono creare il proprio branch di lavoro attraverso il comando `git flow feature start 'nome_branch'`.
Successivamente, andranno seguiti i passi riportati di seguito:

- Verificare di trovarsi nel branch nel quale si vuole lavorare attraverso il comando `git branch` (il branch nel quale si è al momento del comando è segnato da un asterisco "*" a sinistra del nome del ramo);
- Eseguire il comando `git pull origin develop` per aggiornare il proprio ramo con con quello di develop;
- Dopo aver finito il proprio lavoro, eseguire il comando `git add nome_del_file` per aggiungere un determinato file all'*area di staging*$_{\mathbf{G}}$ (per inserirli tutti basta digitare `git add .`;
- Dopo essersi accertati delle modifiche inserite, eseguire il comando `git commit -m "inserisci messaggio del commit"`, inserendo un messaggio utile e che aiuti a comprendere ciò che è stato eseguito;
- Eseguire il comando `git push` per pubblicare sul repository remoto (GitHub) le modifiche effettuate, rendendole di conseguenza visibili agli altri componenti del gruppo.

3.3 Garanzia di qualità

3.3.1 Scopo

In questa sezione sono definite le norme delle metriche e degli obiettivi di qualità utili per il perseguimento della qualità di processo e di prodotto. Inoltre vengono fissate le metriche che il gruppo *DStack* intende rispettare per garantire un livello qualitativo accettabile per la proponente. Tali elementi verranno utilizzati all'interno del documento *Piano di Qualifica v3.0.0*.

3.3.2 Controllo di qualità

Fissare delle regole di comportamento e di azioni che permettano il raggiungimento di un sistema di qualità predisposto al miglioramento continuo è fondamentale.

Di conseguenza il gruppo ha deciso che, per effettuare il controllo di qualità necessario alla valutazione oggettiva della soddisfazione dei requisiti da parte del prodotto software, sia adottata la tecnica di *Quality Assurance*$_{\mathbf{G}}$.
L'insieme di attività che ogni membro deve svolgere per garantire l'ottenimento della qualità desiderata sono:

1. Comprendere gli obiettivi di ogni task da portare a termine;
2. Evidenziare con il gruppo eventuali errori e discrepanze rilevate;
3. Effettuare una stima in valore di ogni task da svolgere;
4. Effettuare una stima in termini di dimensione e complessità di ogni task da svolgere;
5. Far uso delle competenze di ogni singolo membro del gruppo;
6. Produrre risultati concreti e quantificabili;
7. Migliorare costantemente la propria formazione e capacità facendo uso di una comunicazione costante di gruppo;
8. Usare il sistema di tiketing offerto da GitHub, in modo da tracciare ogni task;
9. Effettuare *code refactoring*$_{\mathbf{G}}$;
10. Effettuare Continuous Integration;
11. Pieno rispetto degli standard e di quanto normato all'interno del suddetto documento.

Nel dettaglio vengono, nelle prossime sezioni argomentati, i punti 8, 9, 10 della precedente lista.

3.3.2.1 Sistema di Ticketing

Il sitema di Ticketing (o anche chiamato l'Issue Tracker) di GitHub è uno strumento avanzato che di norma viene usato per tracciare l'evoluzione di una nuova idea, dalla nascita fino alla sua conclusione o per risolvere un problema.
Il gruppo ha deciso nelle modalità illustrate in §4.1.5 e §4.1.6 di farne uso in modo da ottenerne i seguenti benefici:

- L'individuazione e definizione in modo esatto di quali sono le proposte di task da eseguire, incentivandone la discussione tra i membri del team di lavoro;
- La tracciabilità dei task ancora aperti, favorendone la discussione in fase di implementazione;
- Un idea chiara di come si vuole/è organizzato il lavoro;
- Una stima, anche se ancora le issue non sono state chiuse, di come il gruppo sta lavorando, evidenziando situazioni di criticità o ritardi sul piano di lavoro.

3.3.2.2 Code refactoring

Il *code refactoring* consiste nel miglioramento della struttura del codice, senza modificarne il comportamento esterno. Viene applicata esclusivamente al fine di migliorare alcune caratteristiche non funzionali del software, in modo da aumentarne la qualità e renderne più agevole la manutenzione. La ristrutturazione del codice eseguita, permette le seguenti migliorie:

- Migliorare il design del codice;
- Aumentare il grado di comprensione del codice;
- Aumentare l'individuazione di bug/errori esistenti;
- Permettere di sviluppare codice più velocemente.

Il code refactoring viene svolto dai *Programmatori* solo una volta che il software ha passato tutti i test definiti all'interno del *Piano di Qualifica*.
Nel concreto con code refactoring, si parla di *code smell*$_{\mathbf{G}}$, che prevede la scelta di indicatori e nomi più espressivi, l'eliminazione di complessità del sorgente, la cancellazione di codice ridondante, duplicato o superfluo.
Il refactoring non è esclusivamente a carico del *Programmatore*, viene supportato in modo automatizzato con l'ide di IntellJ IDEA per il codice sviluppato in Java e con l'ide di Visual Studio per Node.js.

3.3.2.2.1 Bad Smells Code

Per *Bad Smells Code* si intendono quelle caratteristiche presenti nel codice sorgente che generalmente sono riconosciute come probabili indicatori di un difetto di programmazione. Vengono rilevate debolezze di progettazione, di design del codice che riducono la qualità del prodotto software, a prescindere se questo funzioni correttamente o meno. Esistono elenchi di numerosi smells code, il gruppo *DStack* ha deciso di tenere in considerazioni le seguenti aree:

1. **Bloaters**: sono codici, metodi e classi che tendono ad aumentare gradualmente le proprie dimensioni, fino a diventare giganteschi provocando un aumento delle difficoltà di lavoro. Tali problematiche non emergono repentinamente; ma piuttosto tendono ad accumularsi nel tempo con l'evolversi del programma. Durante l'attività di code refactoring verranno per cui presi in esame i seguenti aspetti:
    - **Metodi lunghi**: metodi che contengono molte righe di codice. Uno dei pattern applicabili per la risoluzione del problema consiste nell'estrazione dei metodi ovvero nella scomposizione del metodo originario in sottoproblemi. È anche possibile sostituire un metodo con un oggetto, rendendo le sue variabili interne attribuiti di una classe ed organizzandolo in vari sottometodi;
    - **Classi molto grandi**: classi che contengono molti attributi e metodi. I possibili pattern applicabili per la risoluzione del problema consistono nell'estrazione di una sottoclasse, metodo o interfaccia. Quest'ultima è applicata quando risulta necessario avere una lista di operazioni e comportamenti comuni tra varie classi o quando è necessario descrivere al meglio le operazioni che la classe offre;

- **Uso eccessivo delle primitive**: uso eccessivo all'interno del codice di primitive, anziché di strutture ausiliarie, generando del codice poco flessibile ed ingombrante. Un pattern applicabile consiste, quando vi è la presenza di campi dato primitivi, nella creazione di una classe adibita al raggruppamento logico degli stessi ed eventualmente anche a gestirne il comportamento. Se è presente un array di elementi, questi può essere sostituito da un oggetto;
- **Lista dei parametri troppo lunga**: metodi che accettano in input un numero di parametri troppo elevato. Questo si verifica con maggiore frequenza quando si ha una serie di algoritmi diversi tra loro posti all'interno di uno stesso metodo. Se i valori in input fanno riferimento ad una singola classe, o a classi diverse, è possibile sostituire la lista dei parametri con chiamate alla classe coinvolta, in modo da ottenere i valori desiderabili. Se, invece, i valori sono disgiunti, è possibile realizzare una classe dati, ovvero passare al metodo in refactoring esclusivamente il riferimento all'istanza della classe;
- **Data Clumps**: parti del codice che contengono gruppi di variabili identiche. Un pattern applicabile consiste nell'estrazione di una classe, in modo da incapsulare tali variabili all'interno di una struttura dati che ne permetti l'utilizzo senza che questo comporti duplicazione.

2. **Object-Oriented Abusers**: si tratta di procedure che rispettano in maniera incompleta o errata i principi della programmazione orientata agli oggetti. Durante l'attività di code refactoring verranno per cui presi in esame i seguenti aspetti:

   - **Switch statements**: quando in uno o più metodi vengono usati in abbondanza sequenze di if-then-else o del costrutto switch. Un pattern che può essere usato per isolare gli switch consiste ad esempio nella selezione del metodo di estrazione e suo relativo spostamento nella classe appropriata o con il sostituire l'istruzione condizionale con una soluzione polimorfa;
   - **Variabili temporanee**: variabili che mantengono il valore solo in circostanze ben specifiche. Spesso vengono impiegate quando devono essere svolti calcoli richiesti da un algoritmo, o quando vengono richiesti numerosi input. Un pattern applicabile consiste nell'estrazione di una classe, formata da variabili temporanee, che vengono trasformate in attributi e tutti i metodi che le utilizzano diventano a loro volta metodi della classe;
   - **Refused Bequest**: quando viene fatto uso dell'eredità tra le classi solo con il desiderio di riutilizzare il codice in una superclasse, anche quando superclasse e sottoclasse sono completamente differenti. I possibili pattern da adottare consentono nel caso in cui l'ereditarietà non ha senso e la sottoclasse in realtà non ha nulla in comune con la superclasse, di eliminare l'ereditarietà stessa.
   - **Classi alternative con differenti interfacce**: due o più classi esibiscono lo stesso comportamento, hanno tuttavia metodi con una nomeclatura differente. Un pattern possibile consiste nell'estrazione di una superclasse e nel definire la classi in questione come sottoclassi di quest'ultima. Si viene, così a creare una relazione di Generalizzazione-Specializzazione, in

cui le classi ereditate adottano la medesima nomeclatura per metodi e variabili comuni;

3. **Change Preventers**: si tratta di codici che se vengono modificati, anche in un singolo punto, provocano a cascata una miriade di cambiamenti su tutto il codice; ostacolando la modifica o l'ulteriore sviluppo del software. Durante l'attività di code refactoring verranno per cui presi in esame i seguenti aspetti:

    - **Cambiamenti sparsi**: quando cambiamenti ad una classe causano modifiche anche ad altre classi. Un pattern che può venire usato consiste nel spostare i metodi e attributi appena creati anche in tutte le altre classi che vengono coinvolte nel cambiamento;
    - **Shotgun Surgery**: questo accade quando una singola responsabilità viene suddivisa tra un gran numero di classi. Un pattern applicabile consiste nello spostare i comportamenti delle classe esistenti in una singola classe. E solo quando non c'è una classe appropriata per uno di questi, creane una nuova.
    - **Gerarchie di ereditarietà parallele**: quando l'aggiunta di nuove classi provoca l'aumento di dimensione della gerarchia causando difficoltà nell'apportarvi modifiche. Un possibile pattern consiste nel de-duplicare le gerarchie di classi parallele, innanzitutto, creando un'istanza di una gerarchia che faccia riferimento alle istanze di un'altra gerarchia e rimuovendo la gerarchia della classe indicata.

4. **Dispensables**: si tratta di qualcosa di inutile presente all'interno del codice, la cui assenza avrebbe reso il codice più pulito, efficiente e più semplice da comprendere. Durante l'attività di code refactoring verranno per cui presi in esame i seguenti aspetti:

    - **Commenti**: un metodo è pieno di commenti esplicativi. Uno dei possibili pattern permette se un commento è destinato a spiegare un'espressione complessa, di suddividere l'espressione in sottoespressioni comprensibili;
    - **Codice duplicato**: due o più frammenti di codice sono praticamente uguali. Questo accade di solito quando vi sono più programmatori che lavorano allo stesso progetto nella stesso momento, senza effettuare alcuna sincronizzazione sul lavoro svolto. Se il codice duplicato è all'interno della medesima classe, si può semplicemente estrarre il metodo; se invece i due codici non sono propriamente uguali; ma portano allo stesso risultato viene scelto l'algoritmo migliore in termini di efficienza, complessità e prestazioni;
    - **Dead Code**: variabili, parametri o attributi non più in uso, di solito perchè obsoleti. Questo accade di frequente durante la manutenzione quando c'è un cambiamento od un miglioramento di una determinata porzione di codice, tralasciando il codice vecchio. Un eventuale risoluzione della problematica consiste nel rimuovere ciò che non risulta essere necessario, o parametri di metodi non più utilizzati;
    - **Classi, metodi, parametri inutilizzati**: si verifica maggiormente quando il codice viene creato per uno scopo specifico; ma poi abbandonato. Un modo per evitare l'insorgere della problematica consiste nel eliminare

totalmente la classe che non serve, o nel caso in cui servano solo pochi metodi della stessa, effettuare l'eliminazione dei metodi e/o attributi non necessari.

5. **Couplers**: si tratta di un gruppo di codici che causano un eccessivo accoppiamento tra le classi. Durante l'attività di code refactoring verranno per cui presi in esame i seguenti aspetti:
   - **Feature Envy**: quando un metodo accede ai dati di un altro metodo senza alcun permesso. Questo si verifica con maggiore incidenza quando si effettua uno spostamento di metodi o di attributi da una classe all'altra. I metodi che richiamano gli attributi dell'altra classe possono venire spostati solo quando questi possono effettivamente appartenere a quest'ultima;
   - **Innaproriate Intimacy**: una classe usa i campi ed i metodi interni di un'altra classe. Alcuni dei pattern utilizzabili consistono nello spostamento di porzioni di una classe nella classe in cui vengono utilizzate tali parti; ma funziona solo se la prima classe non ne ha veramente bisogno, o mediante l'utilizzo di Extract Class e Hide Delegate sulla classe per rendere "ufficiali" le relazioni del codice;
   - **Middle Man**: quando una classe esegue azioni delegandole ad un'altra classe. Il pattern utilizzabile è la Rimozione Middle Man, dove i metodi deleganti sono eliminati e la classe è forzata ad eseguire l'azione desiderata in maniera diretta.

3.3.2.3 Continuous Integration

La *Continuous Integration*$_{\mathbf{G}}$ è una pratica applicata in contesti dove lo sviluppo software avviene attraverso un sistema di versioning. Consiste nell'allineamento frequente, molte volte al giorno, degli ambienti di lavoro dei vari sviluppatori verso l'ambiente di lavoro condiviso.
Questa pratica è nata e ha come scopo primario di, combattere la divergenza del codice. I concetti e le tecnologie che concorrono alla creazione della Continuous Integration permettono di far fronte a tutte quelle problematiche dovute all'aggiornamento delle dipendenze nel codice.
Risulta essere un'attività basilare, che deve essere applicata da ogni membro del gruppo, in quanto offre un metodo per affrontare i rischi legati all'incompatibilità, prima che questi abbiano conseguenze drammatiche sul lavoro svolto. Il gruppo *DStack* ha deciso di usare *TravisCI*$_{\mathbf{G}}$ come strumento di *Continuous Integration*, in quanto:

- supporta sia Java che Node.js;
- si integra perfettamente con GitHub;
- è gratuito per le repository pubbliche;
- è gratuito per le repository private nel caso in cui gli sviluppatori siano studenti, al contrario di competitors come *CircleCI*;
- supporta l'integrazione con Docker.

3.3.3 Classificazione delle metriche di qualità

Il gruppo ha deciso di assegnare ad ogni metrica un nome ed un ID in modo che possano essere distinte univocamente nel seguente modo:

M[Categoria][Numero]

dove:

- **M**: valore statico il quale indica che si tratta di una metrica;
- **Categoria**: voce che deve essere sostituita con una delle seguenti:
  - **PRD**: per le metriche riguardanti il prodotto;
  - **PRC**: per le metriche riguardanti i processi;
  - **TST**: per le metriche riguardanti i test.
- **Numero**: numero a tre cifre con il quale si identifica la metrica. Inizia dal valore 1.

3.3.4 Definizione del processo

Per ottenere qualità in un processo essa va ricercata sin dal momento della sua istanziazione. Considerando questo, il gruppo ha deciso di tenere presente i seguenti fattori durante la sua definizione:

- Gli obiettivi di un processo dovranno essere ben definiti e non in sovrapposizione con quelli degli altri;
- Un processo deve farsi carico solo di singolo obiettivo o di obiettivi fortemente in correlazione tra di loro;
- l'attuazione di un processo deve avvenire in un periodo di tempo limitato e ben definito;
- Più processi paralleli devono utilizzare le risorse umane in maniera efficiente;
- Deve essere fatta un'analisi preventiva dei rischi e delle possibili strategie per mitigarli;
- La schedulazione di un processo deve avvenire prevedendo delle *finestre di slack*$_{\mathbf{G}}$ per affrontare rischi non previsti.

3.3.5 Metriche per la qualità del processo

Per gli obiettivi di qualità del processo il gruppo userà le metriche descritte

3.3.5.1 MPRC001 Spice

3.3.5.2 MPRC002 Schedule Variance

Con *Schedule Variance*$_{\mathbf{G}}$ (SV) si intende una metrica temporale che indica se, per una certa attività in esame, si è in anticipo, in orario, o in ritardo rispetto alla schedulazione prefissata.
L' SV è cosi calcolato:

$$SV = EV - PV \tag{1}$$

Con:

- **EV (Earned Value)**: valore del lavoro effettivamente completato fino alla data odierna. Può essere indicato in Giorni o in Euro;
- **PV (Planed Value)**: valore pianificato del lavoro completato fino alla data odierna. Può essere indicato in Giorni o in Euro.

In base al valore calcolato ci si può trovare nei seguenti casi:

- **Valore $>$ 0**: certifica un anticipo rispetto alla pianificazione. Ciò comporta la necessità di migliorare le previsioni di durata nelle successive schedulazioni dei processi;
- **Valore 0**: indica il rispetto delle previsioni, quindi nessuna azione è necessaria sulla pianificazione della durata;
- **Valore $<$ 0**: indica un ritardo rispetto alla pianificazione. Ciò comporta una rischedulazione dei processi che seguono quello in esame e una pianificazione migliore della durata.

Valore ottimale per l'indice in questione: ≥ 0 giorni.
Valore accettabile per l'indice in questione: -5 giorni.

3.3.5.3 MPRC003 Budget Variance

Il *Budget Variance*$_{\mathbf{G}}$ (BV) è una metrica di costo che indica se, alla data odierna, sia stato speso di più o di meno rispetto alla previsione sul Budget.
Il BV è cosi calcolato:

$$BV = PCWS - ACWP \tag{2}$$

Con:

- **PCWS (Planed Cost of Work Scheduled)**: costo pianificato per la realizzazione delle attività fino alla data corrente;
- **ACWP (Actual Cost of Work Performed)**: costo sostenuto per la realizzazione delle attività fino alla data corrente.

Valore ottimale per l'indice in questione: 0%.
Valore accettabile per l'indice in questione: -10%.

3.4 Metriche per la qualità del prodotto

Per gli obiettivi di qualità del prodotto il gruppo userà le metriche descritte.

3.4.0.1 MPRD001 Indice Gulpease

Questo indice riporta quanto un testo prodotto in lingua italiana sia leggibile secondo la seguente formula:

$$Gulp = 89 + \frac{300 * (totale\ frasi) - 10 * (totale\ lettere)}{totale\ parole}$$

I risultati dell'indice sono compresi tra 0 e 100 dove:

- Un risultato inferiore a **80** indica una leggibilità del testo difficile per chi ha una licenza elementare;
- Un risultato inferiore a **60** indica una leggibilità del testo difficile per chi ha una licenza media;
- Un risultato inferiore a **40** indica una leggibilità del testo difficile per chi ha una licenza superiore.

3.4.0.2 MPRD002 Errori ortografici

Il controllo ortografico viene effettuato come descritto nel paragrafo *Verifica ortografica* presente nella sezione *Strumenti* di questo stesso documento.

3.4.0.3 MPRD003 Ambiguità dei requisiti

É calcolata mediante la seguente formula:

$$AR = (\frac{RA}{RT}) * 100$$

Dove

- RA indica il numero di requisiti ritenuti ambigui dai verificatori dell'*Analisi dei Requisiti*;
- RT è il numero di requisiti totali.

3.4.0.4 MPRD004 Copertura requisiti obbligatori

Questo indice permette di valutare quanti sono i requisiti obbligatori coperti dall'implementazione. La formula è la seguente:

$$CRO = (\frac{ROC}{RO}) * 100$$

Dove ROC rappresenta il numero di requisiti obbligatori coperti dall'implementazione mentre RO rappresenta il numero complessivo di requisiti obbligatori.

3.4.0.5 MPRD005 Copertura requisiti accettati

Questo indice permette di misurare quanti requisiti facoltativi e desiderabili siano effettivamente coperti dall'implementazione. La formula è la seguente:

$$CRA = (\frac{RAC}{RA}) * 100$$

Dove RAC rappresenta il numero di requisiti accettati coperti dall'implementazione, mentre RA rappresente il numero complessivo di requisiti accettati.

3.4.0.6 MPRD006 Correttezza dello scambio dei dati

Questo indice permette di misurare la qualità dello scambio dei dati, valutando la correttezza del contenuto e del formato dei dati scambiati tra i vari applicativi. La formula è la seguente:

$$CSD = (1 - (\frac{ERS}{TFI})) * 100$$

Dove ERS rappresente il numero di errori manifestati durante lo scambio di dati tra gli applicativi mentre TFI rappresenta il totale delle funzioni di interfacciamento tra i vari applicativi.

3.4.0.7 MPRD007 Copertura dei test eseguiti

Questo indice permette di valutare il numero dei test eseguiti rispetto a quelli richiesti dalla complessità e criticità del prodotto. La formula è la seguente:

$$CTE = (\frac{TES}{TTOT}) * 100$$

Dove TES rappresenta il numero di test eseguiti mentre $TOTT$ indica il numero complessivo di test da eseguire.

3.4.0.8 MPRD008 Gestione degli errori di esecuzione

Questo indice permette di valutare quante funzioni riescono a gestire correttamente le condizioni d'errore che possono manifestarsi durante la loro esecuzione. La formula è la seguente:

$$CCF = (\frac{FCG}{FCE}) * 100$$

Dove FCG rappresenta il numero totale di funzioni che gestiscono correttamente gli eventuali errori, mentre FCE rappresenta il nuemero totale di funzioni che possono causare errori durante l'esecuzione.

3.4.0.9 MPRD009 Efficacia della documentazione

Questo indice permette di misurare la capacità dell'utente di utilizzare con successo la documentazione mentre completa un task. La formula è la seguente:

$$ED = (\frac{FCSD}{FTD}) * 100$$

Dove $FCSD$ rappresenta il numero di funzioni completate con successo accedendo alla documentazione, mentre FTD è il numero di funzioni totali eseguite accedendo alla documentazione.

3.4.0.10 MPRD010 Consistenza dell'operatività

Permette di misurare la percentuale di funzionalità offerte all'utente in grado di soddisfare le sue aspettative. La formula è la seguente:

$$CO = (\frac{ISS}{IST}) * 100$$

Dove ISS rappresenta il numero di interazioni svolte che soddisfano le aspettative dell'utente, mentre IST è il numero totale di interazioni svolte.

3.4.0.11 MPRD011 Rapporto tra linee di commento e di codice

Questo indice consente di quantificare il rapporto tra linee di commento e linee di codice vero e proprio. Viene calcolato come segue:

$$RCC = (\frac{LCOM}{LC}) * 100$$

Dove $LCOM$ rappresenta il numero di linee di commento, mentre LC è il numero totale di linee di codice presenti.

3.4.0.12 MPRD012 Complessità ciclomatica

É una metrica software utilizzata per stimare la complessità di un programma analizzando il codice sorgente. Viene calcolata utilizzando il *grafo del flusso di controllo*. In questo grafo:

- I nodi rappresentano gruppi indivisibili di istruzioni;
- Un arco connette due nodi se le istruzioni di un nodo possono essere eseguite subito dopo le istruzioni dell'altro.

Sulla base di queste informazioni si evince che un valore troppo elevato è il risultato di codice troppo complesso (e quindi difficilmente testabile e manutenibile).

Esistono due modi per calcolare la complessità ciclomatica:

$$V(G) = E - N + 2P$$

dove:

- E è il numero di archi;
- N è il numero di nodi;
- P è il numero di componenti connesse.[13]

Un metodo alternativo per il calcolo di questo valore è:

$$V(G) = AC - 1$$

dove AC è il numero di aree chiuse presenti nel grafo.

[13] Per una singola unità di programma P il valore è pari a 1

3.4.0.13 MPRD013 Impatto delle modifiche

Questo indice permette di misurare l'impatto negativo sulla corretta esecuzione del software provocato dalle modifiche al codice. La formula è la seguente:

$$IM = (\frac{MCE}{MT}) * 100$$

Dove MCE rappresenta le metriche che hanno procurato un malfunzionamento del software mentre MT rappresenta le modifiche totali.

3.4.0.14 MPRD014 Technical Debt

È un valore che rappresenta la quantitá di lavoro necessari a sopperire scelte progettuali e realizzative affrettate e/o di bassa qualitá[14]. Il valore viene misurato in giorni di lavoro. Questa metrica viene calcolata automaticamente mediante l'utilizzo di $SonarCloud_{\boldsymbol{G}}$.

3.4.0.15 MPRD015 Autonomia dei test

Questo indice permette di valutare quanto i test siano indipendenti. La formula è la seguente:

$$AT = (\frac{DS}{TDP}) * 100$$

Dove DS rappresenta il numero di dipendenze che possono essere simulate ($stub_{\boldsymbol{G}}$) mentre TDP rappresenta il numero totale di dipendenze previste.

3.4.0.16 MPRD016 Facilità d'installazione

Questo indice permette di misurare la facilità con cui l'utente o l'utilizzatore installa e personalizza il prodotto nel suo ambiente senza doverla ripetere più volte. La formula è la seguente:

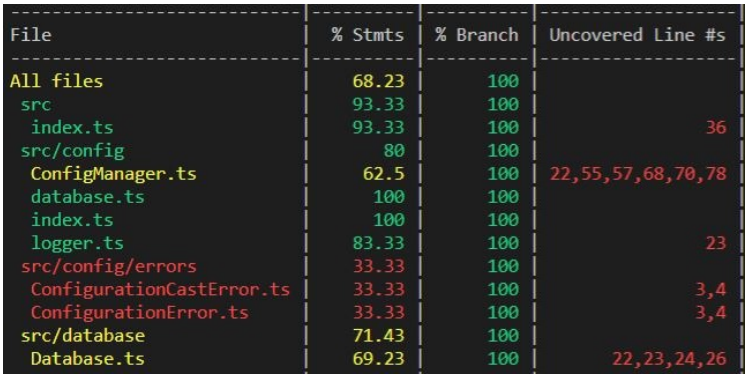
$$FINS = (1 - (\frac{INSF}{TINS})) * 100$$

Dove $INSF$ rappresenta il numero delle volte in cui l'operazione d'installazione fallisce mentre $TINS$ rappresente il numero complessivo di tentativi di installazione.

3.4.0.17 MPRD017 Copertura delle istruzioni di codice Si tratta di una metrica che evidenzia il rapporto tra le istruzioni di cui viene effettuata la verifica dinamica rispetto al numero di istruzioni di codice totali. I risultati di tale metrica sono compresi tra 0 e 100 dove:

- Un risultato inferiore a **50** indica che il grado misurato di testabilità del prodotto è insufficiente;
- Un risultato inferiore a **80** indica che, a livello di testabilità, il prodotto è discretamente maturo;

[14] https://www.martinfowler.com/bliki/TechnicalDebt.html



File	% Stmts	% Branch	Uncovered Line #s
All files	68.23	100	
src	93.33	100	
index.ts	93.33	100	36
src/config	80	100	
ConfigManager.ts	62.5	100	22,55,57,68,70,78
database.ts	100	100	
index.ts	100	100	
logger.ts	83.33	100	23
src/config/errors	33.33	100	
ConfigurationCastError.ts	33.33	100	3,4
ConfigurationError.ts	33.33	100	3,4
src/database	71.43	100	
Database.ts	69.23	100	22,23,24,26

Figura 38: Esempio di copertura delle istruzioni di codice e di *branch coverage* calcolata dal framework di testing *Jest*$_{\mathbf{G}}$.

- Un risultato inferiore a **100** indica che è stato realizzato un ottimo numero di test utili, che possono permettere di verificare eventuali regressioni future e di diminuire il rischio di errori logici, a patto che le asserzioni su cui si basano i test siano corrette.

3.4.0.18 MPRD018 Copertura dei possibili percorsi del codice

Conosciuto anche come indice di *branch coverage*, si tratta di un indicatore percentuale che indica il rapporto tra tutte le possibili deviazioni del flusso del codice (nella forma di costrutti `if else`, `operatori ternari`, `lancio di eccezioni` e altre istruzioni di questo tipo).

- L'unico valore accettabile è **100%**

Un esempio reale degli indici *MPRD018*$_{\mathbf{G}}$ e *MPRD024*$_{\mathbf{G}}$ è mostrato in figura 38.

3.4.1 MPRD019 Code smells

I code smells rappresentano un cattivo utilizzo del codice che non rispetta le convenzioni accettate dalla community. Rappresenta l'addozione di bad practies e favorisce l'introduzione di errori nel programma.
Questa metrica viene calcolata automaticamente mediante l'utilizzo di *SonarCloud*$_{\boldsymbol{G}}$
Il risultato è considerato:

- **Accettabile**: se il numero di code smells è uguale o inferiore ad 85;
- **Buono**: se il numero di code smells è uguale o inferiore a 30;
- **Ottimale**: se il numero di code smells è pari a 0;

3.4.2 MPRD020 Duplicazione codice

La duplicazione del codice rappresenta una porzione di codice che viene ripetuta più volte all'interno del codice sorgente, portando a problemi come la correzione dello stesso problema per più volte.
Questa metrica viene calcolata automaticamente mediante l'utilizzo di *SonarCloud*$_{\boldsymbol{G}}$
Il risultato ottenuto da questa metrica è considerato:

- **Accettabile**: se la percentuale di codice copiato è pari o inferiore a 5;
- **Ottimale**: se la percentuale di codice copiato è pari a 0;

3.5 Verifica

Il processo di verifica viene svolto su ogni processo in esecuzione, nel momento in cui raggiunge un livello di maturità sufficiente e successivamente a cambiamenti inerenti lo stato. Per ciascun processo viene analizzata la qualità dei prodotti ottenuti e dei processi impiegati.
All'interno del *Piano di Progetto 4.0.0* sono descritti 5 periodi, ognuno dei quali produce output distinti, imponendo per cui la necessità di avere procedure di verifica ad hoc per ogni singolo periodo.
La verifica viene svolta con l'uso delle metriche descritte all'interno del *Piano di Qualifica v3.0.0* ed i risulti conseguiti sono riportati all'interno del medesimo documento.

All'interno di questa sezione verranno descritti i metodi e gli strumenti con i quali i *Verificatori* del gruppo *DStack* eseguono il processo di *verifica*$_{\mathbf{G}}$ sulla documentazione e sul codice.

3.5.1 Verifica della documentazione

Il *Responsabile di Progetto* stabilisce l'inizio del processo di verifica pianificando la sua data di inizio, la durata ed i *Verificatori* che dovranno assolvere al task. Questi ultimi hanno il compito di effettuare un analisi accurata dei documenti assegnateli mirata al controllo della sintassi, al rispetto delle norme tipografiche stabilite all'interno del suddetto documento, al utilizzo di frasi con periodi brevi, al controllo della struttura e degli ambiti di appartenenza dei contenuti dello stesso. I *Verificatori* per adempiere in modo più agevole alle loro mansioni fanno uso di *Aspell*.

3.5.1.1 Procedura di verifica della documentazione

La procedura di verifica della documentazione, è a carico del *Verificatore* e viene svolta come illustrato nel seguente diagramma delle attività.

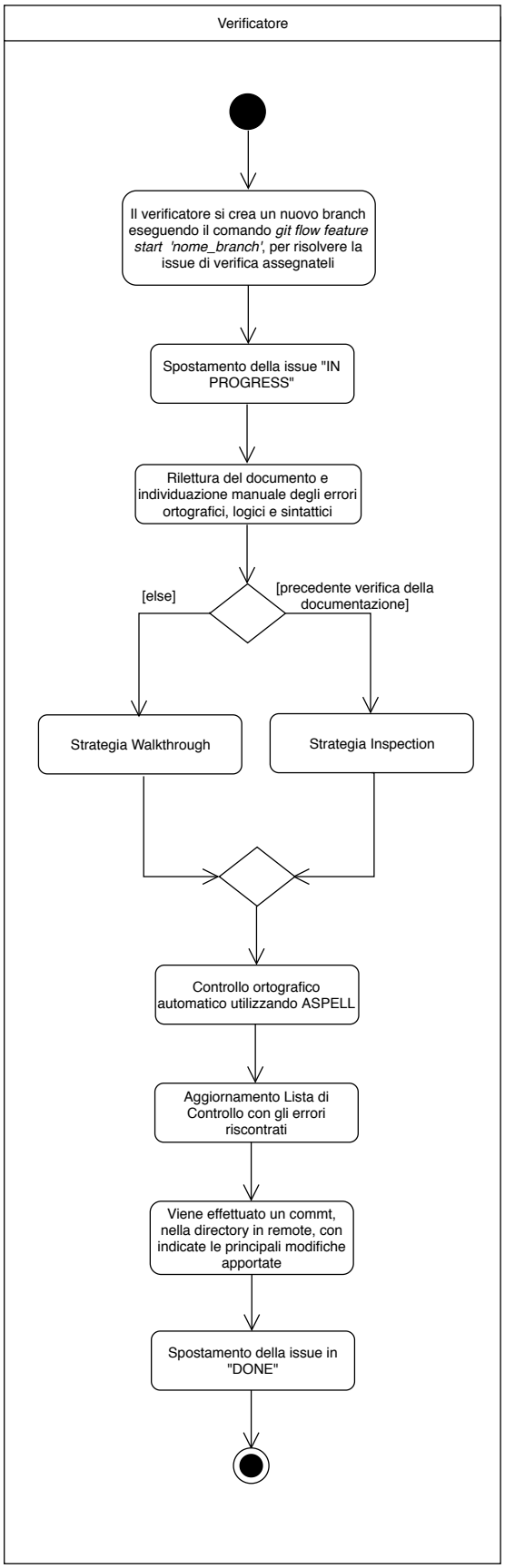


Figura 39: Diagramma delle attività di verifica della documentazione

3.5.1.2 Analisi statica

Questo tipo di analisi prevede un controllo sulla documentazione. Il *Verificatore* incaricato adotterà uno dei seguenti metodi:

- **Walkthrough**: tecnica di verifica attraverso la quale il *Verificatore* eseguirà una lettura ed un controllo completo dell'intero documento alla ricerca di eventuali errori. Verrà utilizzato questo metodo fino a quando non si conoscono gli errori che si possono incontrare;
- **Inspection**: tecnica di verifica attraverso la quale il *Verificatore* eseguirà una lettura e un controllo più mirato del documento nei punti in cui è più probabile vi siano errori.

La tecnica *Walkthrough* risulta molto onerosa a causa della ricerca a pettine. Conseguentemente deve essere usata solo nelle prime fasi del progetto o quanto non è possibile avere la lista degli errori comuni.
È la norma che con il proseguimento dell'attività di progetto, e quindi con il ripetersi del processo di verifica sulla documentazione, si venga a creare una lista degli errori comuni, denominata *Lista di Controllo*, consentendo l'utilizzo del *Inspection*.

3.5.2 Verifica del codice

Il codice verrà testato con test automatici, appositamente creati, e tramite misurazioni quantificabili definite all'interno del *Piano di Qualifica v3.0.0*.
Il codice una volta verificato deve risultare conforme ai seguenti criteri:

- Codice testabile, corretto e rispettoso di quando dichiarato dai requisiti e nello standard di codifica adottato;
- Le interfacce e i dati sono consistenti;
- Codice derivato dalla progettazione o dai requisiti;
- Corretta sequenza di operazioni implementate conformi agli eventi.

3.5.2.1 Analisi statica

L'analisi statica non è prerogativa unica della verifica dei documenti, coinvolge anche il codice prodotto. Viene effettuata automaticamente ad ogni build grazie agli strumenti *checkstyle*$_{\mathbf{G}}$ e *tslint*$_{\mathbf{G}}$ secondo quanto riportato rispettivamente in §2.2.5.10 e §2.2.5.11.

3.5.2.2 Analisi dinamica

L'analisi dinamica rappresenta una tecnica di verifica del codice prodotto, che viene testato durante la sua esecuzione. Consiste nella produzione e nell'esecuzione di una serie di test case sul codice, in modo da verificarne il corretto funzionamento dello stesso e rilevare eventuali scostamenti tra i risultati ottenuti e quanto atteso definito dall'oracolo. Ogni test è perciò decidibile.
Inoltre è fondamentale sia ripetibile, con un dato input si deve ottenere sempre il medesimo output per ogni prova effettuata.

Un test rispetta la seguente struttura:

- **Input**: dati in ingresso immessi;
- **Output**: dati in uscita immessi;
- **Oggetto di prova**: contiene una descrizione del test eseguito;
- **Ambiente di esecuzione**: viene definito l'ambiente hardware e software dove viene eseguito il test;
- **Informazioni aggiuntive**: contiene le modalità di interpretazione dei risultati ottenibili ed informazioni, eventuali, riguardanti l'esecuzione del test.

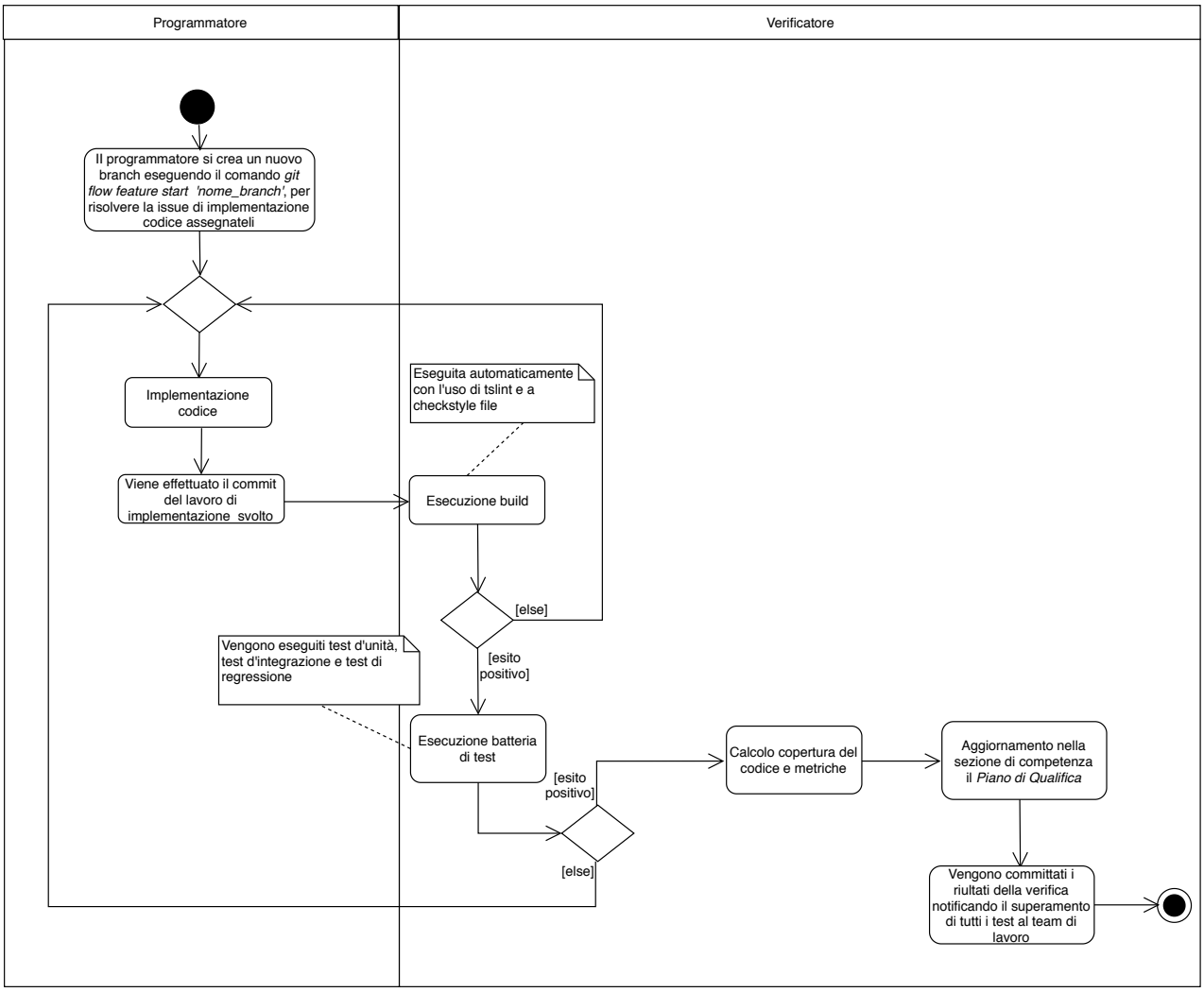


Figura 40: Diagramma delle attività di verifica del codice

3.5.3 Verifica dei requisiti

I requisiti sono anch'essi sottoposti a verifica, vengono applicate le strategie di *Walkthrough* ed *Inspection* che ne controllano la validità. I requisiti per soddisfare positivamente la verifica, devono dimostrarsi:

- Coerenti con quanto dichiarato al proponente e descritto all'interno dell'*Analisi dei Requisiti v3.0.0*;

- Dimostrarsi sempre verificabili e quantificabili;
- Essere lo specchio di quanto implementato, sotto forma di codifica, nel sistema;
- Dimostrarsi con un grado di fattibilità adeguato rispetto al tempo/risorse pianificate.

3.5.3.1 Analisi statica

I requisiti si devono presentare conformi ai seguenti criteri:

- **Verificabilità**: un requisito si deve presentare quantificabile e misurabile oggettivamente;
- **Codice univoco**: un requisito deve essere identificato da un codice, differente da requisito a requisito e rispettare quanto dichiarato in §2.2.3.1;
- **Atomicità**: un requisito deve essere indivisibile.

3.5.4 Test

Un test case su un software viene sviluppato con l'obiettivo di dimostrare che effettivamente il programma svolge effettivamente i tasks per cui è stato sviluppato. Permette di individuare tutti i possibili errori prima che il prodotto software sia reso in uso sul mercato e di valutarne la conformità rispetto all'*Analisi dei Requisiti*.

I test sviluppati sono i seguenti.

3.5.4.1 Test d'unità

I *test d'unità* sono mirati alla verifica della correttezza degli algoritmi. L'obiettivo principale di questa tipologia di test, consiste nell'isolamento della parte più piccola di software testabile, chiamata unità, in modo da stabilirne il corretto funzionamento rispetto a quanto atteso. Ogni singola unità deve essere sottoposta a test prima di procedere alla sua integrazione. Inoltre per poter eseguire un test è necessaria la scrittura di *driver*$_{\mathbf{G}}$ simulatore chiamata di un utente/sistema, e *stub*$_{\mathbf{G}}$ simulatore di un altra unità.

3.5.4.2 Test d'integrazione

I *test d'integrazione* sono mirati alla verifica della correttezza delle interfacce. Rappresentano l'estensione logica dei *test d'unità* e sono i predecessori dei *test di sistema*. Vengono eseguiti con l'aggregazione in un unico componente di due o più unità già state testate. Il concetto di base dei test d'integrazione sta nell'eseguire tali test, in modo da combinare progressivamente più parti, espandendo via via il processo al test di moduli di un gruppo con quelli di altri gruppi. Ogni test prevede vi siano almeno un paio di componenti correlate con lo scopo ultimo di sottoporre a verifica tutte le unità che compongono un processo.

3.5.4.3 Test di sistema

I *test di sistema* mirano a controllare la corretta esecuzione dell'intera applicazione. Viene effettuato un controllo sulle componenti del sistema che devono presentarsi:

- Compatibili tra di loro;
- Con proprie iterazioni e scambio di dati tra le interfacce, conformi ed al momento giusto.

I test di sistema caratterizzano la validazione del prodotto software finale, mezzo con il cui si verifica che tutti i requisiti siano soddisfatti in modo completo.

3.5.4.4 Test di regressione

I *test di regressione* mirano a verificare la qualità di nuove versioni del prodotto software. Lo scopo principale è valutare le nuove funzionalità/modifiche non ancora testate, introdotte nel codice; garantendo che quelle preesistenti abbiano mantenuto le proprie caratteristiche qualitative. Risulta, per ciò, necessario eseguire nuovamente tutti i test esistenti sul codice modificato, in modo da essere in grado di stabilire in maniera esatta se vi è stata un'alterazione del comportamento di elementi precedentemente funzionanti dovuta alle modifiche.

3.6 Validazione

Il processo di validazione consiste nell'assicurarsi che il prodotto soddisfi i requisiti imposti dal committente. È compito del *Responsabile* controllare i risultati passati dal verificatore, decidendo se:

- Accettare ed approvare il documento;
- Rigettare il documento, chiedendo un'ulteriore verifica del documento con nuove indicazioni.

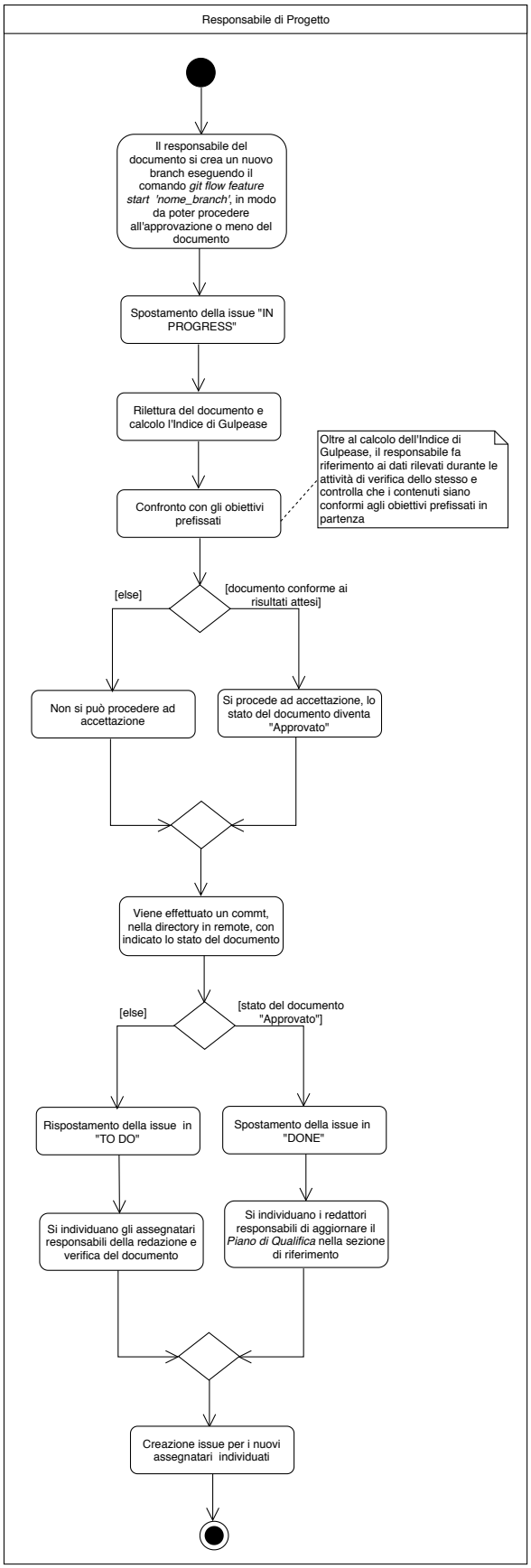


Figura 41: Diagramma delle attività di validazione della documentazione

La validazione del codice si compone di due attività fondamentali:

- **Test di validazione e di sistema**: attività di controllo interne al gruppo fornitore;
- **Collaudo**: attività di controllo eseguita in presenza di un committente, durante la *Revisione di Accettazione*. Utile a dimostrare la conformità del prodotto software secondo quanto promesso da contratto nei casi d'uso e nei requisiti espressi dal documento *Analisi dei Requisiti v3.0.0*.

I *test di sistema* sono eseguiti per mezzo di test d'integrazione, questi non hanno la funzione di verificare la correttezza del software prodotto; ma di effettuarne un controllo sulle funzionalità implementate. Questo permette di stabilire con certezza se si sta, o meno, rispettando quanto dichiarato nel contratto con il proponente.
Il tracciamento risulta un'attività fondamentale per avere conferma del il pieno soddisfacimento di tutti i requisiti. Si è ritenuto più idoneo ed efficiente svolgere un tracciamento di tipo automatizzato grazie all'ausilio di *PragmaDB*, come definito in §2.2.6, Strumenti primari;
Il processo di validazione si caratterizza con le seguenti attività:

- Pianificazione e tracciamento dei test da eseguire sul codice prodotto;
- Esecuzione dei test , vengono create situazioni di forzatura del sistema nei suoi punti critici in modo da evidenziarne gli errori;
- Aggiornamento del Piano dei Test;
- Controllo dei risultati dei test, dove viene verificato se il sistema soddisfi i requisiti.

L'esecuzione di un test di sistema e di validazione è definito in un *Piano dei Test*, incluso all'interno del documento *Piano di Qualifica*. Per ogni test eseguito devono venire specificati:

- Le entità sottoposte a validazione;
- I task e gli obiettivi della validazione;
- I verificatori che hanno eseguito il test e le risorse impiegate;
- Resoconto con identificazione dei risultati ottenuti, attesi, problematiche riscontrate durante l'esecuzione del test.

3.6.1 Test di validazione

I *test di validazione* si svolgono durante la fase di *Collaudo*$_{\mathbf{G}}$ alla presenza di committente e proponente. Se i test, appartenenti alla tipologia in esame, ottengono un risultato positivo significa che vi è garanzia che il sistema sviluppato soddisfi tutti i requisiti obbligatori, rendendo così possibile il rilascio del software. Ogni test di validazione è corredato da un requisito da testare.

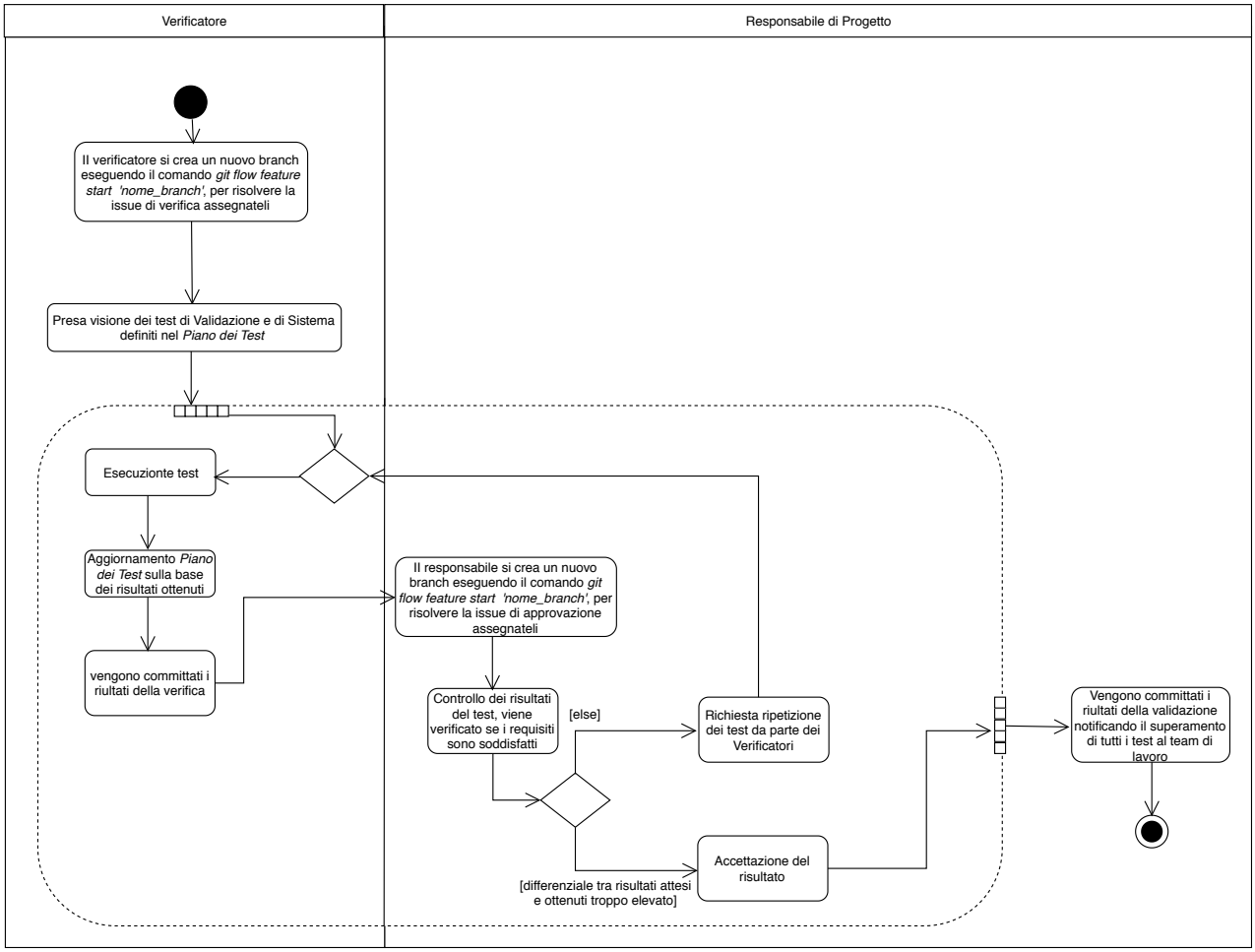


Figura 42: Diagramma delle attività di validazione del codice

3.6.2 Responsabilità dei Test

Sui test di validazione sono coinvolti tre soggetti, *Responsabile di Progetto*, *Verificatori* e *Progettisti*:

- I *Progettisti* hanno il compito di pianificare ed implementare i test. I *Progettisti* devono essere persone differenti da chi ha progettato o implementato la porzione di codice che deve essere testata. In questo modo si garantisce una natura dei test non condizionata ed imparziale rispetto al codice scritto ed obiettività nell'esecuzione degli stessi;
- I *Verificatori* hanno il compito eseguire il test sul prodotto software, tracciarne i risultati ed effettuare l'aggiornamento del *Piano dei Test*;
- Il *Responsabile di Progetto* ha il compito di controllare quanto descritto nel *Piano dei Test* e di valutarne i risultati ottenuti. Spetta al *Responsabile* decidere se:
  - Il risultato ottenuto dal test è sufficientemente buono e ne procede all'accettazione;
  - Il risultato dei test presenta un margine tra i risultati ottenuti e quelli attesi troppo elevato, per cui ne chiedere la ripetizione ai *Verificatori*, eventualmente in sua presenza.

3.6.3 Codice Identificativo dei Test

Ogni test viene descritto con:

- Codice identificativo;
- Descrizione;
- Stato, che può assumere uno tra i valori seguenti:
  - Implementato;
  - Non implementato;
  - Non eseguito;
  - Superato;
  - Non superato.

Ciascun test è definito con l'ausilio del codice seguente:

T[Tipo][Id]

- **T**: valore statico identificativo di "Test";
- **Tipo**: rappresenta un valore statico. I valori che può assumere sono i seguenti:
  - **U**: test d'unità
  - **I**: test d'integrazione;
  - **S**: test di sistema;
  - **R**: test di regressione;
  - **V**: test di validazione.
- **Id**: rappresenta un codice identificativo. I valori che può assumere sono i seguenti:
  - **idNumerico**: nel caso di test d'unità, di integrazione e di regressione. Viene espresso con un codice numerico progressivo che inizia da 1;
  - **idRequisito**: nel caso di test di validazione e di sistema. Composto da *[Tipologia][Importanza][Codice]* (come definito in §2.2.3.1) identifica il codice del requisito definito all'interno del documento Analisi dei Requisiti.

4 Processi organizzativi

4.1 Gestione di progetto

4.1.1 Scopo

In questa sezione sono descritte le modalità di coordinamento e di pianificazione adottate dal gruppo per quanto riguarda la comunicazione, gli incontri (con soggetti esterni, o tra i membri dello stesso gruppo), i ruoli di progetto e l'assegnazione dei compiti. Il processo di gestione è strutturato nel seguente modo:

- Comunicazione:
  - Comunicazione interna;
  - Comunicazione esterna.
- Riunioni:
  - Riunioni interne;
  - Riunioni esterne.
- Pianificazione:
  - Ruoli di progetto;
  - Assegnazione dei ruoli;
  - Ciclo di vita del ticket.
- Identificazione dei rischi.

4.1.2 Comunicazione

Durante lo svolgimento del progetto il gruppo *DStack* si troverà a comunicare su due livelli distinti: interno ed esterno. Relativamente alle comunicazioni esterne, esse avverranno con i seguenti soggetti:

- ***Proponente*$_{\mathbf{G}}$**: l'azienda *Imola Informatica S.P.A.*, rappresentata dal signor Luca Cappelletti e dal signor Davide Zanetti ;
- ***Committenti*$_{\mathbf{G}}$**: nella persona del prof. Tullio Vardanega e del prof. Riccardo Cardin;
- ***Competitor*$_{\mathbf{G}}$**: il gruppo *AlphaSix*;
- **Esperti esterni**: da consultare eventualmente previo accordo con il proponente ed i committenti.

Il gruppo si rivolgerà al competitor di persona ed in maniera informale, mentre intratterrà comunicazioni scritte, video-chiamate e riunioni con il proponente, i committenti e gli esperti.

4.1.2.1 Comunicazione interna

Il mezzo di comunicazione standard per interazioni scritte tra membri del gruppo *DStack* sarà *Slack*$_{\mathbf{G}}$, servizio di messaggistica istantanea rivolto alla collaborazione aziendale. La politica per la gestione delle discussioni sarà inoltre la seguente: per ogni compito di importanza non trascurabile che il gruppo dovrà svolgere verrà creato un canale specifico (ad esempio tutte le discussioni riguardanti un documento avverranno all'interno del canale apposito).
Oltre a questo tipo di canali, destinati a mutare nel corso del tempo, le discussioni verranno raggruppate nel seguente modo:

- #**general**: canale dedicato alle discussioni di natura generica sul progetto. Ad esso verrà aggiunto un *bot*$_{\mathbf{G}}$ per effettuare facilmente e velocemente sondaggi e votazioni;
- #**riunioni**: dedicato all'organizzazione delle riunioni, esterne o interne;
- #**documenti-esterni**: dedicato alle discussioni riguardanti i documenti esterni;
- #**documenti-interni**: dedicato alle discussioni riguardanti i documenti interni;
- #**github**: in questo canale sarà integrato un bot il cui compito sarà quello di inviare notifiche ogni qualvolta i componenti effettueranno operazioni nel repository (ad esempio commit, creazione di issue, pull request);
- #**random**: per discussioni off-topic o non strettamente collegate al progetto. Ogni membro potrà discutere di argomenti di sua preferenza.

Includendo nel corpo le seguenti parole chiave, Slack permette di notificare un particolare messaggio ad una o più persone:

- **@everyone**: il messaggio verrà notificato a tutti i componenti;
- **@channel**: il messaggio verrà notificato a tutti gli utenti iscritti a quel particolare canale;
- **@username**: inserendo l'username specifico, il messaggio verrà notificato all'utente desiderato.

In situazioni eccezionali, quali l'impossibilità di usare Slack, i membri del gruppo possono fare ricorso a servizi di comunicazione alternativi (ad esempio SMS) per non dover sospendere forzatamente il lavoro.

Esistono infine altri due tipi di comunicazioni interne:

- **Comunicazioni orali**: i membri del gruppo potranno discutere informalmente di argomenti riguardanti il progetto, per chiarire dubbi o proporre idee. Non sarà possibile peró prendere alcun tipo di decisione senza prima indire (o far indire) una riunione;
- **Comunicazioni tramite video-chiamata**: per le video-chiamate il gruppo utilizzerà *Jitsi*$_{\mathbf{G}}$, che è stato preferito ad altri servizi analoghi per alcune sue peculiarità, tra cui il fatto di essere open source e multi-piattaforma.

4.1.2.2 Comunicazione esterna

Sarà il *Responsabile di Progetto*, a nome dell'intero gruppo, a mantenere i contatti verso l'esterno attraverso un indirizzo di posta elettronica creato appositamente. L'e-mail utilizzata sarà

`dstackgroup@gmail.com`

Lo stesso *Responsabile di Progetto* dovrà notificare ad ogni membro del gruppo di eventuali corrispondenze pervenute dai committenti e proponenti applicando le norme relative alle comunicazioni interne definite in §4.1.2.1.
È stato deciso che ogni e-mail inviata si baserà sulla seguente struttura:

- **Oggetto**: l'oggetto della mail sarà preceduto dalla sigla "*[SWE-UNIPD]*", in modo tale da rendere subito chiaro l'ambito di riferimento della mail stessa, e dovrà essere espresso nella maniera più chiara e concisa possibile, per eliminare le ambiguità e favorire la comprensione dell'argomento trattato;
- **Corpo**: il tono da mantenere è formale, ci si rivolgerà dando del Voi o del Lei. Il corpo sarà sempre preceduto da una formula di apertura formale, come "*Egregio*", "*Alla cortese attenzione di Imola Informatica, nella persona di*" o "*Spettabile*", dipendentemente dal destinatario. Il contenuto dovrà inoltre essere sintetico ed esaustivo, per esporre nitidamente il problema e/o le eventuali richieste.

Altri canali di comunicazione verso il proponente verranno aperti mediante l'utilizzo del servizio di messaggistica istantanea *Telegram*$_{\mathbf{G}}$ e il servizio di video-chiamata *Google Hangouts*$_{\mathbf{G}}$, come richiesto esplicitamente nel capitolato.
I messaggi non avranno alcuna struttura predefinita e saranno decisi contestualmente poichè il tono della discussione è diretto e meno formale rispetto all'e-mail.
Le video-chiamate saranno effettuate dopo essersi opportunamente accordati su data e ora.
Per eventuali comunicazioni con soggetti terzi il mezzo e le modalità di comunicazione verranno decisi di volta in volta nella maniera piú adeguata.

4.1.3 Riunioni

Vengono definite qui le modalità di svolgimento delle riunioni, che possono essere interne od esterne.
Per ogni riunione sarà nominato un segretario, con il compito di tenere traccia delle discussioni affrontate, di far rispettare l'ordine del giorno ed infine redigere un **verbale di riunione**[15] usufruendo delle informazioni raccolte.

4.1.3.1 Riunioni interne

La partecipazione alle riunioni interne sarà permessa ai soli membri del gruppo. Alla riunione dovranno partecipare almeno quattro persone per poterne riconoscere la validità: le decisioni verranno infatti prese per maggioranza semplice. Inoltre, poichè una riunione sia ritenuta valida il *Responsabile di Progetto* dovrà portare a termine i seguenti compiti:

[15] Vedi §3.1.3.3

- Fissare la data della riunione, previa verifica della disponibilità dei membri del gruppo;
- Stabilire un ordine del giorno;
- Comunicare[16] data e ordine del giorno scelte, oppure avvertire con un anticipo adeguato (almeno un giorno) se vi sono ritardi, modifiche o cancellazioni;
- Nominare un segretario per la riunione;
- Approvare il verbale di riunione.

I partecipanti alla riunione, invece, devono:

- Avvertire preventivamente se non possono essere presenti o se prevedono ritardi;
- Presentarsi puntuali all'ora e luogo stabiliti;
- Partecipare attivamente alla discussione.

Ogni membro del gruppo avrà la possibilità di richiedere un incontro interno: tale domanda dovrà essere indirizzata solo al *Responsabile di Progetto*, il quale la visionerà e pianificherà un eventuale incontro.

4.1.3.2 Riunioni esterne

Le riunioni esterne prevedono la partecipazione di soggetti esterni, oltre ai componenti del gruppo *DStack˙*
Così come le riunioni interne, anche le riunioni esterne prevedono la nomina di un segretario che dovrà poi redigere un verbale di riunione; Data la distanza che separa Padova da Imola (dove ha sede l'azienda proponente), le riunioni esterne con il proponente avverranno principalmente tramite Hangouts, dopo essersi accordati sul giorno e sull'ora tramite Telegram o e-mail.
Se invece si presenterà la possibilità, verranno utilizzate le aule della Torre Tullio Levi Civita (ex Torre Archimede, Via Trieste 63, 35121 Padova (PD)), previa disponibilità.

4.1.4 Pianificazione

4.1.4.1 Ruoli di progetto Nel corso del progetto, i componenti del gruppo ricopriranno i seguenti ruoli:

- *Responsabile di Progetto*;
- *Amministratore*;
- *Analista*;
- *Progettista*;
- *Programmatore*;

[16] Utilizzando il canale Slack #riunioni, vedi §4.1.2.1

- *Verificatore*.

Essi corrispondono alle rispettive figure aziendali, e sarà stato stabilito un calendario che permetterà ad ogni membro di ricoprire almeno una volta ciascun ruolo per un periodo di tempo omogeneo, senza peró inficiare lo svolgimento delle attività. L'assegnazione di un ruolo comporta lo svolgimento di determinati compiti, così come previsto dal *Piano di Progetto*. Inoltre, tramite scelte oculate, si cercherà di eliminare eventuali conflitti di interesse: ad esempio, un componente non potrà redigere e poi verificare ciò che ha prodotto lui stesso.

4.1.4.1.1 Responsabile di progetto

Il *Responsabile di Progetto*, "Project Manager" in inglese, è un ruolo fondamentale, presente per tutta la durata del lavoro. Rappresenta il gruppo *DStack* presso il proponente ed i committenti, ed il suo principale compito è coordinare l'intera struttura ed armonizzare il lavoro, in maniera tale che ogni sforzo fatto sia utile al fine comune. I seguenti punti chiariscono cosa questo ruolo comporti:

- Deve occuparsi del coordinamento dei membri del gruppo e dei compiti che devono portare a termine (vedi §4.1.5);
- Deve gestire la pianificazione, intesa come attività da svolgere e scadenze da rispettare;
- È responsabile della stima dei costi e dell'analisi dei rischi;
- Deve curare le relazioni che il gruppo intratterrà con i soggetti esterni (vedi §4.1.2.2 e §4.1.3.2);
- Si occupa delle risorse umane e dell'assegnazione dei ruoli;
- Deve approvare la documentazione.

4.1.4.1.2 Amministratore

L'*Amministratore* è incaricato di gestire, controllare e curare gli strumenti che il gruppo utilizza per svolgere il proprio lavoro. È la figura che garantisce l'affidabilità e l'efficacia dei mezzi scelti dal gruppo, persegue infatti l'idea che la buona gestione dell'ambiente di lavoro (inteso come insieme di regole, strumenti e servizi) favorisca la produttività e pertanto deve:

- Amministrare infrastrutture e servizi necessari ai processi di supporto;
- Gestire il versionamento e la configurazione dei prodotti;
- Gestire la documentazione di progetto, controllando che venga corretta, verificata ed approvata, e facilitandone il reperimento;
- Correggere eventuali problemi relativi alla gestione dei processi;
- Redigere e manutenere norme e procedure che regolano il lavoro;
- Individuare strumenti utili all'automazione dei processi.

4.1.4.1.3 Analista

L'*Analista* partecipa al progetto soprattutto nei suoi stadi iniziali, in particolare nel momento di stesura dell' *Analisi dei Requisiti*. Il suo compito è estrapolare i punti chiave del problema in oggetto, comprendendone appieno tutte le sue sfaccettature. La sua figura è quindi fondamentale per la buona riuscita del lavoro, in quanto errori o mancanze nell'individuazione dei requisiti da soddisfare possono compromettere fortemente la successiva attività di progettazione. Egli:

- Studia e definisce il problema in oggetto;
- Analizza il fronte delle richieste, definisce quali sono i requisiti studiando i bisogni, siano essi impliciti o espliciti;
- Analizza il fronte applicativo, in particolare gli utenti ed i casi d'utilizzo;
- Redige lo *Studio di Fattibilità* e l'*Analisi dei Requisiti*.

4.1.4.1.4 Progettista

Il *Progettista* parte dal lavoro svolto dall' *Analista*, e ha il compito di elaborare una soluzione che soddisfi i bisogni individuati. Tale compito ha natura sintetica, poichè il suo scopo è produrre un'architettura che modelli il problema a partire da un insieme di requisiti. Chi ricopre questo ruolo:

- Applica principi noti e collaudati per produrre un'architettura che assicuri coerenza e consistenza;
- Deve produrre una soluzione sostenibile e realizzabile, che rientri nei costi stabiliti dal preventivo;
- Si adopera per la costruzione di una struttura che soddisfi tutti i requisiti e che sia aperta alla comprensione;
- Cerca di limitare il più possibile il grado di accoppiamento tra le varie componenti;
- Rivolge il suo sforzo verso la ricerca dell'efficienza, della flessibilità e della riusabilità;
- Elabora una soluzione capace di interagire in modi diversi (per forma e per numero) con l'ambiente in cui si pone, e che sia sicura rispetto ad eventuali anomalie e intrusioni esterne;
- Ricerca la massima disponibilità ed affidabilità per l'architettura proposta.

4.1.4.1.5 Programmatore

Il *Programmatore* è la figura incaricata della codifica. Egli deve implementare l'architettura prodotta dal *Progettista* in maniera che aderisca il più possibile alle specifiche, ed è responsabile della manutenzione del codice creato. Egli:

- Codifica secondo le specifiche stabilite dal *Progettista*. Il codice prodotto è documentato, versionabile ed è strutturato in maniera tale da agevolare la futura manutenzione;

- Crea le componenti che serviranno poi per la verifica e la validazione del codice;
- Scrive il *Manuale Utente* relativo al codice prodotto.

4.1.4.1.6 Verificatore

Il *Verificatore* è presente per tutta la durata del progetto. Egli si occupa di controllare che le attività svolte rispettino le norme e le attese prefissate. I compiti del *Verificatore* sono:

- Controllare ed accertarsi che l'esecuzione delle attività di processo non abbia introdotto errori;
- Redigere la parte retrospettiva del *Piano di Qualifica*, la quale descrive e chiarisce le verifiche e le prove effettuate.

4.1.5 Assegnazione dei compiti

La progressione nello svolgimento del progetto può essere vista come il completamento (in maniera sequenziale o parallelizzata) di una serie di compiti, ognuno con la sua data di scadenza, i quali producono risultati utili alla realizzazione di obiettivi posti.
Tali compiti sono determinati a volte dalla contingenza, a volte dai processi in atto, ma nella maggior parte dei casi sono dovuti da entrambi i fattori.
Per l'assegnazione dei compiti si è deciso di utilizzare il servizio di *ticketing*$_{\mathbf{G}}$ offerto da *GitHub*$_{\mathbf{G}}$, basato sul concetto di *issue*$_{\mathbf{G}}$: essa permette di assegnare un compito da svolgere entro una certa data a uno o più componenti, il tutto in maniera facile ed immediata.
La figura che si occupa della gestione dei compiti è il *Responsabile di Progetto*. Egli:

- Individua il compito da svolgere;
- Se ritiene il compito troppo complesso, lo può suddividere in più sottocompiti;
- Individua uno o più componenti del gruppo al quale assegnare il compito (o i sottocompiti);
- Apre una *issue*, assegnando il compito al soggetto/i e definendo una data entro la quale completarlo.

Di contro, i membri del gruppo:

- una volta che il compito è stato assegnato, occorre che si impegnino a svolgerlo entro la data prefissata;
- sono responsabili della chiusura della *issue*, una volta completato il lavoro.

4.1.6 Ciclo di vita del Ticket

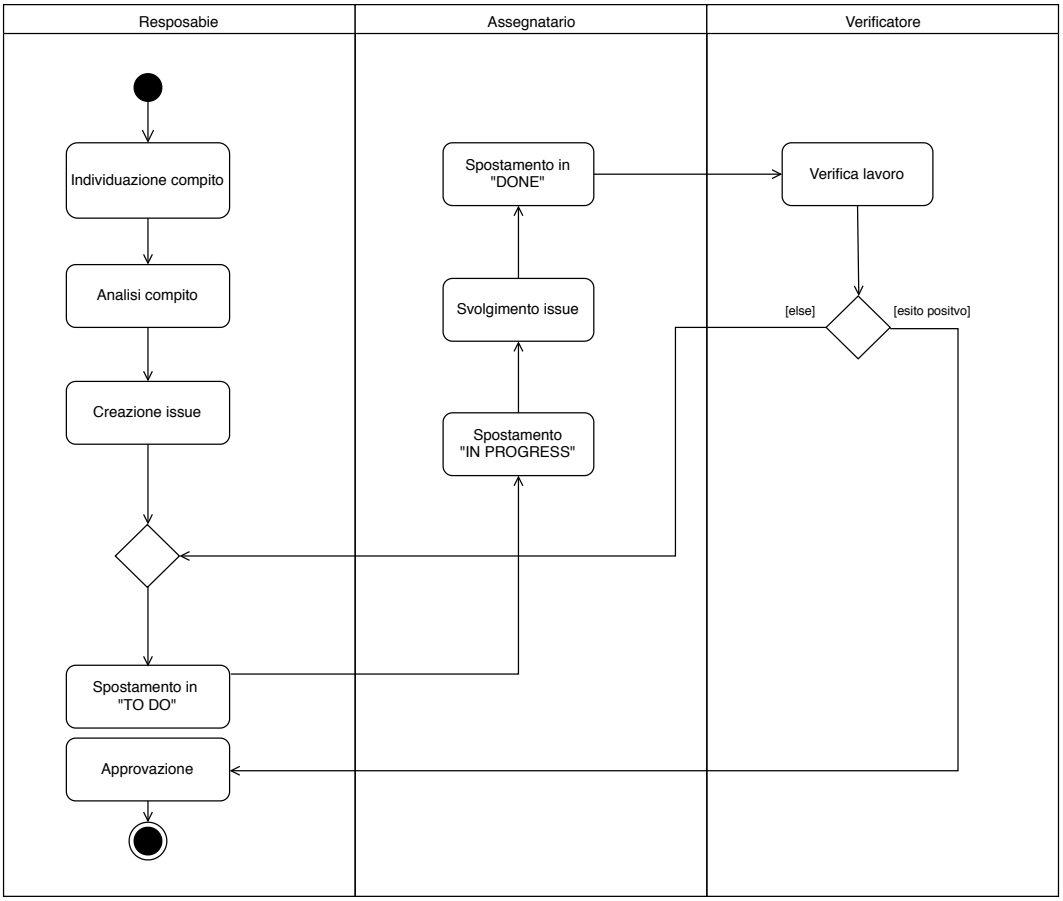


Figura 43: Rappresentazione del ciclo di vita del Ticket

Nel periodo di tempo che intercorre tra l'individuazione di un lavoro da svolgere e il completamento dello stesso entrano in gioco le figure di:

- **Responsabile**: è il *Responsabile di Progetto*, che si occupa di identificare un compito da svolgere, assegnarlo ad un componente del gruppo ed aprire la relativa issue. Una volta che il lavoro è svolto e verificato, egli lo deve approvare;
- **Assegnatario**: è il membro del gruppo incaricato allo svolgimento di tale compito;
- **Verificatore**: è la persona deputata al controllo e alla verifica del lavoro svolto dall'*Assegnatario*. Ha la facoltà di riaprire la issue qualora non lo dovesse ritenere valido.

Segue la descrizione dettagliata del ciclo di vita del ticket. Si vuole precisare che esso ha validità generale, indipendentemente dalla natura del compito da svolgere:

- **Individuazione compito**: il Responsabile rileva determinate necessità, a cui bisogna rispondere con azioni opportune. Concettualizza pertanto quanto identificato in richieste da soddisfare ed obiettivi da raggiungere, creando così un *compito*$_{\mathbf{G}}$;

- **Analisi compito**: il Responsabile stima la complessità del compito appena identificato, dividendolo in più sottocompiti se lo ritiene troppo oneroso. Individua poi uno o più Assegnatari e una data di scadenza entro la quale il lavoro dovrà essere svolto;
- **Creazione issue**: il Responsabile crea una *issue* utilizzando GitHub, impostando l'Assegnatario e la data di scadenza decisi precedentemente;
- **Spostamento issue in "TODO"**: una volta creata la issue, il Responsabile la sposta nell'insieme "TODO", indicante tutti i compiti individuati, ma il cui svolgimento non è ancora cominciato;
- **Spostamento issue in "IN PROGRESS"**: una volta che l'Assegnatario decide di iniziare il lavoro, sposta la issue nella categoria "IN PROGRESS";
- **Svolgimento issue**: in questo periodo di tempo il compito viene portato a termine dal relativo Assegnatario;
- **Spostamento issue in "DONE"**: l'Assegnatario sposta la issue nella categoria "DONE", per indicare che ha completato il suo lavoro;
- **Verifica del lavoro**: il Verificatore giudica quanto fatto dall'Assegnatario:
  - Accettando il lavoro svolto;
  - Oppure rifiutando quanto fatto, spostando nuovamente la issue in "TODO";
- **Approvazione lavoro**: il Responsabile esamina per un'ultima volta il lavoro, controllando che rispetti gli obbiettivi fissati in partenza, e lo approva.

4.2 Training e formazione

I membri del gruppo devono provvedere alla propria formazione in maniera autonoma, studiando le tecnologie usate e colmando eventuali carenze pregresse, per poter garantire una qualità di lavoro che rispetti le aspettative.
Il gruppo farà riferimento alla seguente documentazione, oltre a quella reperita per proprio conto, che dovrà essere condivisa con gli altri membri del team nel canale Slack #**resources**:

4.2.1 Creazione di documenti

- **LaTeX**: https://www.latex-project.org;
- **TexStudio**: https://www.texstudio.org/;
- **Tex StackExchange** (sito analogo a StackOverlow a tema LaTeX): https://tex.stackexchange.com;
- **Aspell**: http://aspell.net/.

4.2.2 Apache Kafka e tecnologie correlate

- **Kafka: guida ufficiale**: https://kafka.apache.org/documentation/;
- **Guida completa a Kafka**: https://www.confluent.io/wp-content/uploads/confluent-kafka-definitive-guide-complete.pdf;
- **Kafka Schema Registry**: https://docs.confluent.io/current/schema-registry/docs/index.html;
- **Apache Avro**: http://avro.apache.org/docs/1.8.2/;
- **Corso Udemy su Apache Kafka**: https://www.udemy.com/apache-kafka/;
- **Come creare un Consumer Kafka**: https://data-flair.training/blogs/kafka-consumer/;
- **Consigli sulla gestione dei Producer e Consumer Kafka**: https://blog.workwell.io/how-to-manage-your-kafka-consumers-from-the-producer-9933b88085dd.

4.2.3 Node.js e tecnologie correlate

- **Node.js: guida ufficiale**: https://nodejs.org/en/docs/;
- ***npm***$_G$: https://www.npmjs.com;
- ***TypeScript***$_G$: https://www.typescriptlang.org/;
- **Evolution of ECMAScript**: https://codeburst.io/javascript-wtf-is-es6-es8-es-2017-ecmascript-dca859e4821c;
- ***TSLint***$_G$: https://palantir.github.io/tslint/;
- ***KoaJS***$_G$: https://koajs.com/.

4.2.4 Java e tecnologie correlate

- ***Java***$_G$: https://www.java.com;
- ***Maven***$_G$: https://maven.apache.org/;
- ***Checkstyle***$_G$: https://github.com/checkstyle/checkstyle;
- ***HTTP Spark***$_G$: http://sparkjava.com/;
- **Come usare Apache Avro in Java**: https://avro.apache.org/docs/1.8.2/gettingstartedjava.html.

4.2.5 Servizi di terze parti

- ***Redmine***$_G$: https://www.redmine.org;
- ***GitLab***$_G$: https://about.gitlab.com/;
- ***Sonarqube***$_G$: https://www.sonarqube.org/;

- **Supporto webhook per Redmine**: https://github.com/suer/redmine_webhook;
- **Supporto webhook per GitLab**: https://docs.gitlab.com/ee/user/project/integrations/webhooks.html;
- **Supporto webhook per Sonarqube**: https://docs.sonarqube.org/latest/project-administration/webhooks/.

4.2.6 Docker e tecnologie correlate

- ***Docker*$_{G}$**: https://docs.docker.com/;
- ***Docker Compose*$_{G}$**: https://docs.docker.com/compose/;
- **Reti virtuali in Docker Compose**: https://docs.docker.com/network/;
- ***Kubernetes*$_{G}$**: https://kubernetes.io;
- ***Rancher*$_{G}$**: https://rancher.com/.

4.2.7 Test, Code Coverage e Continuous Integration

- ***Test di unità*$_{G}$**: http://softwaretestingfundamentals.com/unit-testing/;
- ***Test di integrazione*$_{G}$**: http://softwaretestingfundamentals.com/integration-testing/;
- ***Test d'accettazione*$_{G}$**: http://softwaretestingfundamentals.com/acceptance-testing/;
- ***Test di sistema*$_{G}$**: http://softwaretestingfundamentals.com/system-testing/;
- ***JUnit*$_{G}$**: https://junit.org/junit5/;
- ***Jest*$_{G}$**: https://jestjs.io;
- ***CodeCov*$_{G}$**: https://codecov.io/;
- ***TravisCI*$_{G}$**: https://hackernoon.com/continuous-integration-using-travis-on-github-1f7f2314b6b7.

4.3 Gestione dell'infrastruttura

4.3.1 Scopo

In questa sezione viene fornita una breve descrizione degli strumenti scelti e di come essi sono stati configurati per adempiere a tutte le attività di comunicazione e pianificazione inerenti il processo di gestione.

4.3.2 Strumenti

4.3.2.1 Di comunicazione

4.3.2.1.1 Slack

Slack[17] è uno strumento di collaborazione aziendale, utilizzato per inviare messaggi in modo istantaneo ai membri di un team di lavoro. L'applicativo permette ad un utente di iscriversi a diversi *workspace*$_{\mathbf{G}}$. Per ogni workspace di appartenenza è concessa la creazione di canali tematici nel quale sviluppare delle conversazioni specializzate. Questo procedimento evita il mischiare di diversi argomenti su un'unica chat generale, come invece avviene in *Telegram*$_{\mathbf{G}}$.
Slack offre l'integrazione con diverse applicazioni. Alcune app integrate dal gruppo sono: *GitHub* in modo da aver sempre notificato ogni azione svolta sulla repository del progetto (che sia creazione delle issue, commit, ...) da ogni membro del gruppo, favorendo così l'incremento della collaborazione ed il tracciamento dei tasks svolti da ciascuno in un momento specifico; e *Jitsi* strumento usato per le videochiamate.

4.3.2.1.2 Jitsi

Jitsi[18] è uno strumento con un set di applicazioni open source, che consentono di creare delle chiamate di gruppo personalizzate da remoto. Il core di Jitsi è composto da Jitsi Videobridge e Jitsi Meet che si occupano di gestire la videoconferenza e l'interfacciamento tra i moduli presenti all'interno dell'applicativo. Per poter effettuare una videochiamata è necessario che un utente acceda al sito `https://meet.jit.si` ed inserisca, a libera scelta, il nome della stanza che desidera creare (esempio `https://meet.jit.si/dstack`). Ai creatori della stanza è concesso, se necessario, definire una password di accesso alla videoconferenza. Per permettere ad altri utenti di accedere alla chiamata basta condividere il link della stanza e la password, se presente. Jitsi, oltre alla normale videochiamata, offre anche la possibilità di fare conferenze in *broadcasting*$_{\mathbf{G}}$, dove gli utenti invitati assistono esclusivamente, o/e di effettuare lo stream della stessa direttamente su Youtube.
Ogni partecipante alla videochiamata, in ogni caso, ha la possibilità di settare il proprio nome e avatar.
Il gruppo ha deciso di usare questo strumento di comunicazione, prediligendolo ad altri quali *Skype* e *Hangouts* a causa della sua natura free, coerente con il progetto Butterfly; e per la sua organizzazione a stanze, che concede la massima privacy senza richiedere alcun download di pacchetti software.

4.3.2.1.3 Telegram

Telegram[19] è uno strumento di messaggistica istantanea, basato su cloud, open source che il gruppo ha utilizzato durante le prime settimane successive alla sua creazione e nel caso si verifichino imprevisti che impediscano o ritardino la comunicazione attraverso *Slack*. Alcune funzionalità offerte da Telegram sono:

- Sincronizzazione istantanea del cloud, contenete i messaggi inviati dall'utente permettendo l'accesso agli stessi da diversi dispositivi contemporaneamente;
- Chat cloud, che usano una cifratura client-server, permettendo il salvataggio della conversazione cifrata sul server di Telegram. Viene concesso l'invio

[17] `https://slack.com/`
[18] `https://jitsi.org/`
[19] `https://telegram.org/`

di messaggi di testo, messaggi vocali, videomessaggi, posizione GPS attuale, coordinate GPS sulla mappa, contatti e file con una dimensione massima di 1.5 GB;

- Chat segrete, che usano una cifratura end-to-end ovvero tra i due dispositivi coinvolti nella conversazione, causa per cui la conversazione non viene salvata sul server di Telegram;
- Gruppi ospitanti fino a 200 mila membri, dove è possibili impostare amministratori con permessi selezionabili. Un gruppo può essere reso pubblico settando un username, in questo modo è data la possibilità di leggerne i messaggi senza per forza unirsi;
- Canali sono chat dove solo l'amministratore può inviare messaggi ai membri. Un canale può contenere un numero illimitato di utenti e può essere sia pubblico che privato;
- Bot sono degli account Telegram, gestiti da un programma, offrenti molteplici funzionalità con risposte immediate e automatizzate.

Il gruppo fa uso durante il progetto di un gruppo privato, in cui sono coinvolti tutti i membri del team, di Bot per creare sondaggi e se necessario di chat cloud. Inoltre una chat cloud viene utilizzata per comunicare con l'azienda proponente, come definito in §4.1.2.2.

4.3.2.1.4 Hangouts

Hangouts[20] è uno strumento di messaggistica istantanea e VoIP offerto come servizio di comunicazione da Google. Offre numerose integrazioni tramite piattaforme e plugin compatibili con il browser di Google Chrome. La più comune è *Gmail*.
Hangout in Gmail permette di effettuare chiamate e videochiamate, condivisione foto esclusivamente con l'ausilio di una casella di posta Gmail, senza l'istallazione di applicazioni aggiuntive.
Il gruppo impiega questo strumento esclusivamente per trattare con l'azienda proponete, come descritto in §4.1.2.2.

4.3.2.1.5 Gmail

Gmail[21] è uno strumento open source di posta elettronica fornito da Google.
Il servizio gestisce la casella e-mail del gruppo, come descritto in §4.1.2.2. Inoltre sono integrati nella stessa, nativamente, anche altri strumenti come Google Keep, Google Tasks e Google Calandar. Quest'ultimo è utilizzato da ogni membro del gruppo, per segnare la propria disponibilità giornaliera nei confronti dello svolgimento di attività che coinvolgono il progetto.

4.3.2.2 Di pianificazione

[20] https://hangouts.google.com/
[21] https://www.google.com/intl/it/gmail/about/

4.3.2.2.1 GanttProject

Per il supporto alla pianificazione del progetto e alla realizzazione di diagrammi di Gantt all'unanimità si è deciso di utilizzare lo strumento *GanttProject*[22], software open source e multi-piattaforma che offre le seguenti funzionalità:

- Creazione di *diagrammi di Gantt*$_{\mathbf{G}}$ e di *PERT*$_{\mathbf{G}}$;
- Organizzazione dei compiti secondo la *Work Breakdown Structure*$_{\mathbf{G}}$;
- Gestione delle dipendenze tra compiti;
- Creazione di *baseline*$_{\mathbf{G}}$.

A GanttProject viene affiancato *Clockify*$_{\mathbf{G}}$.

4.3.2.2.2 Clockify

Clockify [23] è uno strumento per il time tracking. Quest'ultimo:

- Ha una grande facilità d'uso, permette di effettuare in pochi passi operazioni che altrimenti potrebbero risultare più complesse;
- Presenta la possibilità di essere usato da qualsiasi dispositivo: è infatti disponibile sotto forma di applicazione desktop e mobile, e offre anche un plug-in per browser;
- Permette di avere statistiche utili a creare un quadro generale della situazione, ed è flessibile nel riportare i dati di interesse.

Il gruppo ha valutato l'uso di foglio di *Google Sheets*$_{\mathbf{G}}$ [24] condiviso per tenere traccia delle ore di lavoro di ciascuno, tuttavia si è deciso di preferire Clockify per i maggiori automatismi offerti, ritenuti fonti di miglioramento della produttività generale.

[22] https://www.ganttproject.biz/
[23] https://clockify.me/
[24] https://www.google.com/sheets/about/

A The Twelve Factor App

Si tratta di una metodologia di sviluppo orientata alla costruzione di applicazioni *Software as a Service*$_{G}$ che può essere applicata ad ogni prodotto software, a prescindere dal linguaggio di programmazione in cui è scritto. Queste applicazioni sono caratterizzate date:

- Formati dichiarativi per l'automazione della configurazione, allo scopo di ridurre al minimo i tempi ed i costi di ingresso dei nuovi sviluppatori che aderiscono al progetto;
- Disaccoppiamento tra codice scritto e piattaforma d'esecuzione, in modo da offire la massima portabilità sui vari sistemi operativi di destinazione;
- Addattabilità dello sviluppo su piattaforme cloud più moderne;
- Riduzione al minimo delle divergenze tra ambiente di sviluppo e produzione, incentivando il *Continuous Deployment*$_{G}$ per ottenere la massima agilità;
- Scalabilità significativa senza eccessivi cambiamenti ai tool, al processo di sviluppo o architettura.

I 12 principi contenuti in *The Twelve-Factor App* sono esposti di seguito.

- **Codebase**: una sola codebase sotto controllo di versione, tanti deployment;
- **Dipendenze**: dipendenze dichiarate e isolate;
- **Configurazione**: memorizza le informazioni di configurazione nell'ambiente;
- **Backing Service**: tratta i backing service come "risorse";
- **Build, Release, Esecuzione**: separare in modo netto lo stadio di build dall'esecuzione;
- **Processi**: eseguire l'applicazione come uno o più processi privi di stato;
- **Binding delle porte**: esportare i servizi tramite binding delle porte;
- **Concorrenza**: scalare orizzontalmente attraverso il process model;
- **Rilasciabilità**: massimizzare la robustezza con avvii rapidi e chiusure graduali, senza interrompere il sistema bruscamente;
- **Parità tra Sviluppo e Produzione**: mantenere gli ambiente di sviluppo, staging e produzioni quanto più simili possibile;
- **Log**: trattare i log dei servizi come flussi di eventi aggregati;
- **Processi di Amministrazione**: eseguire i task di amministrazione e gestione come processi *una tantum*.

A.1 Codebase

Un servizio deve essere sempre sotto controllo di versione, ad esempio tramite lo strumento *Git*$_{\mathbf{G}}$. Esiste una relazione uno a uno tra *codebase*$_{\mathbf{G}}$ e servizio applicativo, ma ci possono essere più *deploy*$_{\mathbf{G}}$. Ad ogni istante possono esistere più versioni della stessa *codebase*.

Esistono alcune varianti:

- In presenza di più codebase, non si parla più di applicazione, ma di sistema distribuito;
- Più app non possono condividere la stessa codebase, è possibile soltanto con un apposito sistema di dipendenze.

A.2 Dipendenze

Le applicazioni:

- Non devono mai assumere l'esistenza implicita di librerie installate a livello di sistema;
- Devono dichiarare le dipendenze attraverso un file di manifest dedicato, sottoposto a controllo della versione;
- Devono isolare le dipendenze durante l'esecuzione per evitare che dipendenze implicite creino interferenze con il resto del sistema in esecuzione;
- Devono usare le stesse versioni delle dipendenze sia in sviluppo che in produzione;

A.3 Configurazione

Tutte le impostazioni dei servizi devono essere lette in fase di avvio dei servizi da variabili d'ambiente, che sono utili per le seguenti ragioni:

- È molto semplici cambiarle a seconda del *deploy*$_{\mathbf{G}}$, senza dover né toccare il codice sorgente, né ricompilarlo;
- Possono essere dichiarate in un file di *env*, che sono file indipendenti dal sistema operativo e dal linguaggio utilizzato;

A.4 Backing Service

I servizi di appoggio, come ad esempio database, devono essere trattati come risorse identificate da un'*URL*$_{\mathbf{G}}$ e degli eventuali parametri di configurazione, che dovranno essere letti dalle variabili di ambiente. Alle applicazioni non deve far differenza il connettersi a risorse locali o a risorse di terze parti accedibili online.

A.5 Build, Release, Esecuzione

A partire dal codice sorgente, viene effettuato il *deploy*$_{\mathbf{G}}$ in 3 fasi distinte:

- Build: converte il codice della repository in una build "eseguibile". Usando una certa versione del codice, a una specifica commit, durante questa fase vengono compilati i file binari con gli asset appropriati, includendo anche le eventuali dipendenze;

- Release: prende la build prodotta nella fase precedente e la combina con l'attuale insieme di impostazioni di configurazione del *deploy*$_{\mathbf{G}}$ specifico. La release risultante contiene sia la build che le impostazioni;
- Esecuzione: (conosciuta anche come "runtime"), vede l'applicazione in esecuzione nell'ambiente di destinazione, attraverso l'avvio di processi della release scelta.

A.6 Processi

Le applicazioni vengono eseguite nell'ambiente di esecuzione sotto forma di uno o più processi. I processi non devono possedere uno stato interno: tutti i dati che richiedono una persistenza devono essere memorizzati in appositi *Backing Services*$_{\mathbf{G}}$, come ad esempio dei database. Nel caso i file sorgenti dell'applicazione richiedano l'installazione di dipendenze o la compilazione, è necessario che tali operazioni avvengano durante la fase di build, e non a runtime.

A.7 Binding delle porte

I servizi devono essere completamente *self-contained*, ovvero "contenuti in se stessi", nel senso che non devono affidare ad altri servizi (come un webserver) durante l'esecuzione. I servizi devono permettere l'accesso ad un servizio riservando una porta che accetti richieste di protocolli come HTTP, SMTP o altri. I servizi rimarranno in ascolto su tale porta per tutte le richieste in entrata. È di particolare importanza il fatto che, abilitando il binding delle porte, si permette ad un servizio di diventare il *backing service*$_{\mathbf{G}}$ di un altro.

A.8 Concorrenza

Suddividere il carico di lavoro in più processi distinti permette di distribuire il carico in maniera concorrente molto semplicemente, favorendo la scalabilità di sistema. I processi possono essere sincroni o asincroni, a seconda della mole di lavoro da eseguire e le risorse coinvolte, e possono invocare a loro volta altri processi. I processi dovrebbero fare affidamento a sistemi di process manager o al sistema operativo, in modo da rispondere adeguatamente e in maniera automatica a crash improvvisi.

A.9 Rilasciabilità

I processi devono:

- Ambire a minimizzare i tempi di avvio. Idealmente, un processo impiega pochi secondi dal tempo di lancio al momento in cui tutto è pronto per ricevere nuove richieste. Dei tempi brevi di avvio inoltre forniscono una maggiore agilità in fase di release, il tutto a vantaggio della robustezza dell'applicazione;
- Terminare in modo tutt'altro che brusco, quindi graduale, in caso di ricezione di un segnale SIGTERM dal process manager. Per un'applicazione web la giusta terminazione di un processo viene ottenuta quando si cessa di ascoltare sulla porta dedicata al servizio, evitando quindi di ricevere altre richieste, permettendo di terminare le richieste esistenti ed infine di terminare definitivamente;

- Essere robusti nei confronti di situazioni di crash improvviso, cosa che si verificano ad esempio in caso di problemi a livello di hardware sottostante. Nonostante questa seconda evenienza si verifichi meno frequentemente di una chiusura con SIGTERM, può comunque succedere. L'approccio raccomandato, in questi casi, è l'uso di un sistema robusto di code che rimette nella lista delle richieste il lavoro che non può essere immediatamente completato. Una buona applicazione twelve-factor deve poter gestire senza problemi le terminazioni inaspettate, senza che questo generi problemi alla successiva esecuzione.

A.10 Parità tra sviluppo e produzione

Gestire ambiente di lavoro ed esecuzioni diversi provoca solo compilazioni durante la fase di *deploy*$_{\mathbf{G}}$ del servizio. Per ottenere *deploy*$_{\mathbf{G}}$ affidabili, è necessario minimizzare le differenze esistenti tr tempo, personale e strumenti. Per fare ciò si deve:

- **Rendere la differenze temporali minime**: cercare scrivere del codice da rilasciare nel giro di poche ore, se non minuti;
- **Rendere le differenze a livello di personale minime**: gli sviluppatori devono essere coinvolti anche nella fase di deploy, per permettere loro di osservare il comportamento di ciò che hanno scritto in produzione;
- **Rendere le differenze a livello di strumenti minime**: mantenere gli ambienti di lavoro i più simili possibile.

A volte in fase di sviluppo viene utilizzato un servizio meno complesso e con meno caratteristiche rispetto a quello che poi verrà utilizzato in produzione. Inoltre, molti linguaggi offrono anche delle librerie che facilitano l'accesso a questi servizi, tra cui anche degli adattatori. Lo sviluppatore twelve-factor "resiste" a questa necessità. Nulla impedisce, infatti, a qualche altra incompatibilità di uscire allo scoperto quando meno ce lo si aspetta, soprattutto se in ambiente di sviluppo funziona tutto e poi, magari, in produzione i test non vengono superati. Il costo di questa differenza può risultare abbastanza alto, soprattutto in situazioni in cui si effettua il rilascio continuo. Piuttosto è preferibile utilizzare alcuni tool di provisioning, che combinati con sistemi di ambienti virtuali permettono agli sviluppatori di riprodurre in locale delle macchine molto simili, se non identiche, a quelle in produzione. Ne risente quindi positivamente il costo di deploy.
Tutto questo, non rende gli adapter meno utili: grazie ad essi infatti il porting verso nuovi servizi, in un secondo momento, rimane un processo indolore. Nonostante questo, comunque, rimane scontato che sarebbe buona norma usare uno stesso backing service su tutti i deploy di un'applicazione.

A.11 *Log*$_{\mathbf{G}}$

Un'applicazione twelve-factor conforme al principio di *Log* non dovrebbe preoccuparsi di lavorare con il proprio output stream. Non dovrebbe lavorare o comunque gestire i vari i propri logfile. Dovrebbe, invece, fare in modo che ogni processo si occupi di scrivere il proprio stream di eventi su una interfaccia di output standardizzata.
Il log si differenzia in base alla fase dello sviluppo nel quale viene utilizzato:

- **Sviluppo in locale**: lo sviluppatore potrà visionare lo stream in modo completo direttamente da terminale, per comprendere meglio il comportamento della sua applicazione;
- **Staging/produzione**: ogni stream viene gestito dall'ambiente di esecuzione ed elaborato assieme a tutti gli altri stream dell'applicazione, e indirizzato verso una o più "destinazioni" finali per la visualizzazione ed archiviazione a lungo termine. Queste destinazioni non sono visibili o configurabili; ma vengono gestite totalmente dall'ambiente di esecuzione. Per fare ciò esistono strumenti appositi.

Lo stream inoltre può essere inviato ad un sistema di analisi ed indicizzazione di log, oppure ad un sistema di memorizzazione general-purpose. Questi sistemi hanno ottimi tool per effettuare un lavoro di analisi del comportamento dell'applicazione, ne sono un esempio:

- La ricerca di specifici eventi nel passato;
- L'utilizzo di grafici per rappresentare dei trend, ad esempio il numero di richieste per minuto;
- L'attivazione di avvisi specifici in base a regole definite dall'utente, ad esempio nel caso in cui la frequenza di eventi al minuto salga oltre una certa soglia.

A.12 Processi di amministrazione

In un'applicazione twelve-factor conforme al principio di *Processi di amministrazione* lo sviluppatore potrebbe voler l'esecuzione di alcuni task, quali:

- L'esecuzione delle modifiche (migration) del database;
- L'esecuzione di una console in modo tale da avviare del codice arbitrariamente o analizzare alcuni aspetti dell'applicazione specifici;
- L'esecuzione one-time di alcuni script specifici.

Tali processi dovrebbero essere avviati in un ambiente identico a quello in cui lavorano gli altri nel contesto dell'applicazione. Dovrebbe essere eseguita quindi su una specifica release, partendo dalla stessa codebase e dalle impostazioni di configurazione. Il codice per l'amministrazione dovrebbe inoltre essere incluso in quello dell'applicativo, in modo tale da evitare qualsiasi problema di sincronizzazione. La stessa tecnica di isolamento delle dipendenze dovrebbe poter essere usata allo stesso modo su tutti i processi. La metodologia twelve-factor favorisce tutti quei linguaggi che offrono un terminale interattivo, rendendo quindi semplice l'esecuzione di script una tantum. In un deploy locale, gli sviluppatori possono invocare questi processi speciali tramite un semplice comando diretto, mentre in un ambiente di produzione si può raggiungere lo stesso obiettivo tramite SSH o un qualsiasi altro sistema di esecuzione di comandi remoto.

B ISO/IEC 15504 (o SPICE)

L'ISO/IEC 15504, noto come SPICE (Software Process Improvement and Capability Determination) è uno standard che permette di valutare oggettivamene la qualità

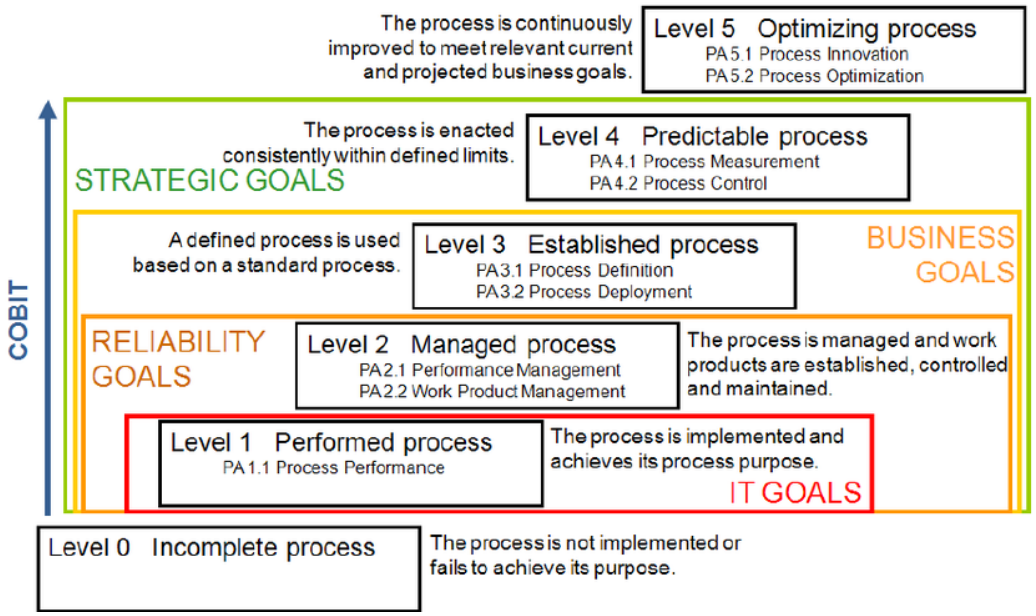


Figura 44: ISO/IEC 15504(SPICE) Fonte:www.researchgate.net

di un processo software al fine di migliorarlo fornendo delle valutazioni sui processi ripetibili, oggettive e comparabili.
Per fare ciò lo standard valuta un processo, inteso singolarmente, in termini di capacità, considerando i seguenti attributi:

- **Process Performance (PP)**: misura la quantità di obiettivi raggiunti;
- **Performance Management (PM)**: misura il livello di organizzazione usato per raggiungere gli obiettivi;
- **Work Product Management (WPM)**: misura il livello di gestione dei prodotti in termini di documentazione, controllo e verifica;
- **Process Definition (PDEF)**: misura il livello di aderenza agli standard;
- **Process Deployment (PDEP)**: misura il livello con cui il processo standard viene rilasciato e distribuito come processo definito in grado di raggiungere sempre gli stessi risultati;
- **Process Measurement (PME)**: misura il grado con cui i risultati delle valutazioni vengono utilizzati per garantire che il processo raggiunga i suoi obiettivi;
- **Process Control (PC)**: misura il livello di stabilità, capacità e predicibilità di un processo;
- **Process Innovation (PI)**: misura il livello di identificazione delle modifiche da apportare ad un processo individuate attraverso una fase di analisi delle performance e studio di soluzioni alternative;
- **Process Optimization (PO)**: misura il livello con cui i cambiamenti migliorativi da apportare al processo hanno un impatto positivo.

A loro volta ognuno degli attributi sopra elencati vengono classificati in:

- **N - Not Implemented**: 0% - 15%, il processo non ha l'attributo implementato o possiede gravi carenze in merito;
- **P - Partially Implemented**: >16% - 50%, l'attributo è parzialmente implementato con approccio sistematico, ma possiede ancora parti non prevedibili e migliorabili;
- **L - Largely Implemented**: >51% - 85%, l'attributo è ampiamente implementato con approccio sistematico, ma il risultato varia nelle diverse unità;
- **F - Fully Implemented**: >86% - 100%, l'attributo è completamente implementato con un approccio sistematico ed è attuato in ugual modo in tutte le unità.

Utilizzando la classificazione degli attributi così ottenuta, si può classificare il livello di capacità di un processo in:

- **5 Optimizing**: un processo di questo livello si presenta come ben definito e tracciato; soggetto ad una continua analisi e al miglioramento, quando necessario. Inoltre, i processi meno efficaci, subiscono modifiche ulteriori in modo da aderire meglio agli obiettivi.
  Attributi associati al livello sono: *Process Innovation* e *Process Optimization*;
- **4 Predictable**: un processo di questo livello, è attuato in modo coerente entro limiti ben definiti. È caratterizzato dalla raccolta e dal controllo di misure dettagliate, che consentono la prevedibilità del processo stesso.
  Attributi associati al livello sono: *Process Measurement* e *Process Control*;
- **3 Established**: un processo di questo livello è definito e basato su uno standard e conseguentemente controllato da principi imposti dall'Ingegneria del Software.
  L'output risultante impiega una quantità di risorse limitate e si attiene agli standard.
  Attributi associati al livello sono: *Process Definition* e *Process Deployment*;
- **2 Managed**: il processo di questo livello è gestito ed i suoi prodotti sono organizzati tramite pianificazione, controllo e correzione. L'output risultante, tracciato e controllato, raggiunge gli obiettivi fissati.
  Attributi associati al livello sono: *Performance Management* e *Work Product Management*;
- **1 Performed**: il processo di questo livello è implementato e raggiunge i propri obiettivi. Tuttavia non subisce alcun controllo costante ai fini di correzione e miglioramento, e gli output che produce sono identificabili.
  Attributi associati al livello sono: *Process Performance*;
- **0 Incomplete**: il processo di questo livello non è implementato o comunque non è in grado di raggiungere i propri obiettivi. Non viene prodotto alcun output e se accade non è significativo.
  Attributi associati al livello sono: *Nessuno*.

C Standard ISO/IEC 9126

Nello standard ISO/IEC 9126 sono riportate una serie di normative e linee guida con lo scopo di descrivere un modello della qualità del software.

La normativa è composta in quattro parti

- **9126-1**: descrive il modello della qualità del software;
- **9126-2**: descrive le metriche per le qualità esterna;
- **9126-3**: descrive le metriche per le qualità interna;
- **9126-4**: descrive le metriche per le qualità d'uso.

C.1 Modello della qualità

Viene descritto nella prima parte dello standard. Il modello è composto da sei caratteristiche generali, ognuna delle quali possiede diversi attributi.

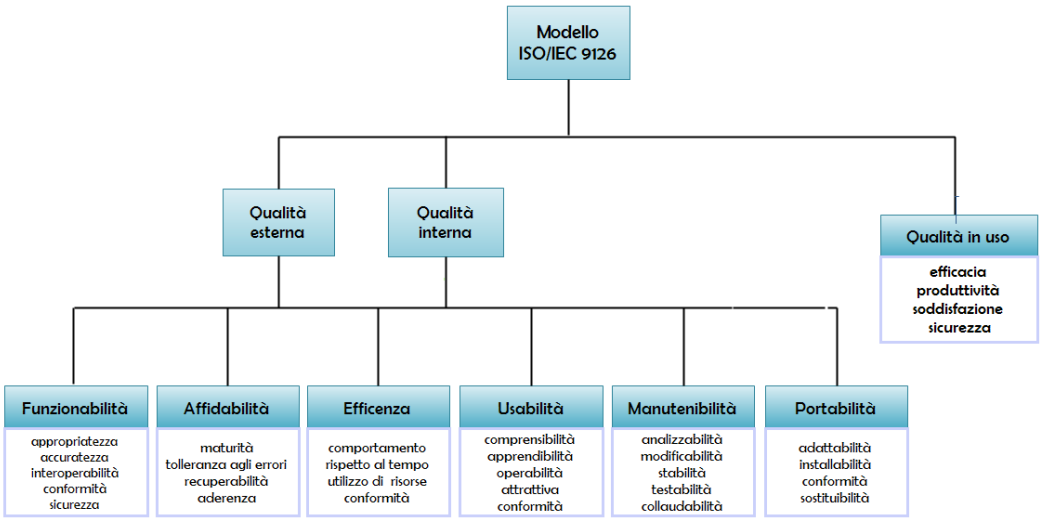


Figura 45: Modello ISO/IEC 9126

C.2 Qualità esterne

Le qualità esterne, nello specifico le loro metriche, vengono descritte nella seconda parte dello standard. Queste metriche sono utili per misurare i comportamenti del software rilevati dai test, dall'operatività e dalla sua osservazione sull'esecuzione. Sono utili ad utenti, tester e sviluppatori per misurare e valutare le caratteristiche esterne. Rientrano nelle qualità esterne le seguenti caratteristiche:

- **Funzionalità**: rappresenta la capacità del prodotto software di fornire le funzioni necessarie per operare in determinate condizioni.

  Caratteristiche:

  - **Appropriatezza**: rappresenta la capacità del software di fornire funzioni adeguate per i compiti e gli obiettivi da raggiungere;
  - **Accuratezza**: rappresenta le capacità del software di fornire un appropriato insieme di funzioni che permettano di svolgere determinati task;
  - **Interoperabilità**: rappresenta la capacità del software di interagire con uno o più sistemi;
  - **Accuratezza**: rappresenta le capacità del software di fornire i risultati con il livello di precisione richiesta;
  - **Sicurezza**: rappresenta la capacità del software di proteggere le informazioni e i dati;
  - **Conformità**: rappresenta la capacità del software di aderire a standard, convenzioni o regolamenti relativi alla funzionalità.

- **Affidabilità:** rappresenta la capacità del prodotto software di mantenere un certo livello di prestazione quando viene utilizzato condizioni di stress.

  Caratteristiche:

  - **Maturità**: rappresenta la capacità di evitare che si verifichino errori o guasti in fase di esecuzione;
  - **Tolleranza agli errori**: rappresenta la capacità di mantenere il livello di prestazioni in caso di errori nel software o violazioni delle interfacce;
  - **Recuperabilità**: rappresenta la capacità di un prodotto di ripristinare il livello appropriato di prestazioni e di recupero delle informazioni rilevanti, a seguito di malfunzionamenti;
  - **Conformità**: rappresenta la capacità del software di aderire a standard, convenzioni o regolamenti relativi all'affidabilità.

- **Usabilità**: rappresenta la capacità del prodotto software di essere comprensibile, di poter essere studiato e di poter essere attraente per l'utente sotto determinate condizioni.

  Caratteristiche:

  - **Operabilità**: rappresenta la capacità del software di poter essere utilizzato e controllato dall'utente;
  - **Apprendibilità**: rappresenta la capacità del software di essere facilmente compressibile per l'utente riguardo le sue funzionalità;
  - **Comprensibilità**: rappresenta la capacità nel permettere all'utente di capire l'uso dell'applicazione;
  - **Attrattivà**: rappresenta la capacità di un software di risultare attraente per un utente;
  - **Comprensibilità**: rappresenta la capacità di un software di risultare chiaro rispetto alle proprie funzionalità ed utilizzo;
  - **Conformità**: rappresenta la capacità del software di aderire a standard, convenzioni o regolamenti relativi all'usabilità.

- **Efficienza**: rappresenta la capacità di un prodotto software di raggiungere gli obiettivi minimizzando il più possibile il numero di risorse disponibile nel tempo necessario.

  Caratteristiche:

  - **Nel tempo**: rappresenta la capacità del software di fornire appropriati tempi di risposta ed elaborazione;
  - **Nello spazio**: rappresenta la capacità del software di utilizzare quantità e tipo di risorse in modo adeguato;
  - **Conformità**: rappresenta la capacità del software di aderire a standard, convenzioni o regolamenti relativi all'efficienza.

C.3 Qualità interne

Le metriche per la qualità interna vengono descritte nella terza parte dello standard. Queste metriche vengono applicate al software non eseguibile durante le fasi di progettazione e codifica. Solitamente sono impiegate nelle fasi di sviluppo per misurare le proprietà del prodotto. Rientrano nelle qualità interne le seguenti caratteristiche:

- **Manutenibilità**: rappresenta la capacità del software di poter essere modificato. Le modifiche includono le aggiunte, rimozioni o eventuali adattamenti del software.

  Caratteristiche:

  - **Modificabilità**: rappresenta la capacità nel poter apportare modifiche al prodotto originale;
  - **Testabilità**: rappresenta la capacità del poter effettuare test sul software appena modificato, consentendone la verifica e la validazione;
  - **Stabilità**: rappresenta la capacità del software nell'evitare la manifestazione di errori dopo una modifica;
  - **Analizzabilità**: rappresenta la capacità di poter effettuare controlli sul software ed individuare cause di errori o malfunzionamenti;
  - **Conformità**: rappresenta la capacità del software di aderire a standard, convenzioni o regolamenti relativi alla manutenibilità.

- **Portabilità**: rappresenta la capacità del software di essere trasportato da un ambiente di lavoro, inteso come hardware/sistema operativo, ad un altro.

  Caratteristiche:

  - **Adattabilità**: rappresenta la capacità del software di adattarsi a differenti ambienti operativi, senza dover applicare modifiche al software considerato;
  - **Installabilità**: rappresenta la capacità del software di essere installato in uno specifico ambiente;
  - **Sostituibilità**: rappresenta la capacità del software di essere impiegato al posto di un altro software con lo scopo di svolgerne le medesime funzioni nello stesso ambiente;
  - **Coesistenza**: rappresenta la capacità del software di coesistere con altri software indipendenti, nel medesimo ambiente condividendo risorse comuni;

  - **Conformità**: rappresenta la capacità del software di aderire a standard, convenzioni o regolamenti relativi alla portabilità.

C.4 Qualità d'uso

Le metriche per la qualità d'uso vengono descritte nella quarta parte dello standard. Le caratteristiche presenti in questa parte rappresentano il punto di vista dell'utente sulla qualità. Questo livello di qualità viene raggiunto dopo aver raggiunto gli obiettivi di qualità interna ed esterna. Rientrano nelle qualità d'uso:

- **Efficacia**: rappresenta la capacità del software di supportare un utente nel raggiungimento dei suoi obiettivi con accuratezza e completezza;
- **Produttività**: rappresenta la capacità del software di supportare un utente nello spendere una quantità di risorse adeguata all'efficienza dei risultati da raggiungere;
- **Soddisfazione**: rappresenta la capacità del software di soddisfare le richieste dell'utente;
- **Sicurezza**: rappresenta la capacità del software di possedere un livello di rischio accettabile rispetto a danni che possono manifestarsi a persone, software, apparecchiature, ambiente operativo.

D PDCA (o Ciclo di Deming)

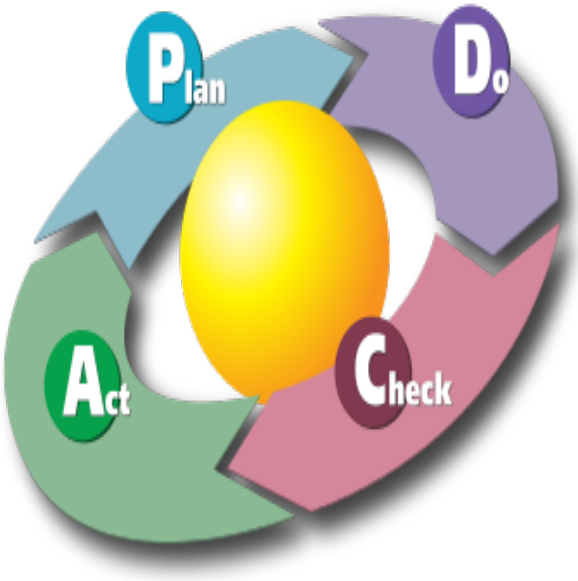


Figura 46: PDCA Fonte:Wikipedia

PDCA (Plan-Do-Check-Act), noto anche come il ciclo di Deming. È un metodo iterativo utilizzato per il controllo e il miglioramento continuo di processi e prodotti. Le quattro fasi di cui è composto sono strutturate come di seguito :

- **Plan**: fase di pianificazione di un processo corrente. Vengono definiti gli obiettivi di miglioramento e si determinano quali sono le attività da svolgere, le risorse da assegnarvi, le scadenze da rispettare ed i risultati attesi.
  Risulta utile pianificare miglioramenti di dimensione ridotta facilmente monitorabili, con output prevedibili;

- **Do**: fase di attuazione di ciò che si è pianificato durante la fase di *Plan* e raccolta di dati utili da analizzare durante le fasi successive;

- **Check**: fase di verifica e valutazione dei dati. I dati emersi durante la fase di *Do* vengono messi a confronto, con quanto previsto dalla fase di *Plan*, con lo scopo di valutarne gli esiti positivi;

- **Act**: fase di miglioramento del processo. Gli aspetti positivi vengono consolidati e le eventuali problematiche, riscontrate durante la fase di *Check*, analizzate. Inoltre, vengono pianificate le attività correttive e quelle volte a garantire un miglioramento della qualità del processo.
  Al termine di questa fase, il ciclo si chiude e tutte le attività pianificate durante quest'ultima, concorrono a creare una nuova fase di *Plan*.