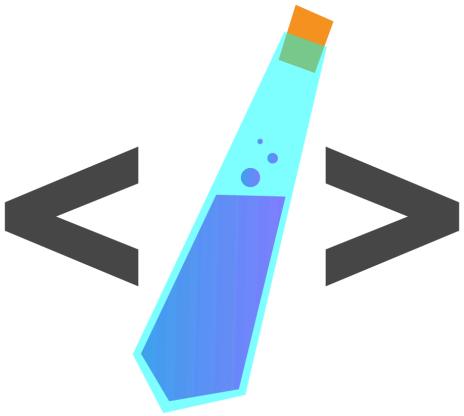

CODE ALCHEMISTS

TURNING IDEAS INTO CODE, LIKE MAGIC

Norme di progetto

Stato

Approvato

Versione

1.0.0

Distribuzione

Code Alchemists
Prof. Tullio Vardanega
Prof. Riccardo Cardin

Registro delle Modifiche

Vers.	Data	Autore	Verificatore	Descrizione
1.0.0	17/07/2025	-	N. Bolzon	Approvazione del documento
0.8.0	17/07/2025	S. Marana	N. Bolzon	Correzione di alcuni errori ortografici, migliorie nelle sezioni 3.1.5 e 3.2.2
0.7.0	16/07/2025	A. Shu	N. Bolzon	Aggiunte sezioni 3.2.4.2.13.1., 4.2.4., 4.2.5., 4.2.6., 4.2.6.1., 4.3.4. e 4.4.2.2.4., modificate 3.2.3.4. e 3.2.3.6., correzione degli errori ortografici, aggiunta una descrizione più dettagliata su NestJS nella sezione 3.2.3.4.
0.6.0	08/07/2025	R. Zangla	A. Shu	Aggiunte informazioni alle sezioni da 3.2.4.2.1 a 3.2.4.2.12 e 4.2.4.
0.5.0	23/05/2025	R. Zangla	N. Moretto	Aggiunta e compilata l'intera sezione 6. Aggiunte le sezioni 4.3.4.1 e 4.3.4.2. Eliminata la sezione 4.4.3. e spostate le sue sottosezioni dentro 4.4.2. Aggiunte informazioni alle sezioni da 3.2.2.2, 3.2.2.3, 3.2.3.4, da 4.1.2 a 4.1.11 (sottosezioni incluse, tranne 4.1.5.1), 4.2.1, 4.4.1, 4.4.2 (sottosezioni incluse), da 4.3.1 a 4.3.3, 4.5.1, 5.1.2.5, 5.1.2.6, 5.2.2, 5.2.4, da 5.3.2.1 a 5.3.2.12 (sottosezioni incluse), 5.3.1, da 5.4. a 5.5 (sottosezioni incluse)
0.4.0	09/05/2025	R. Zangla	S. Marana	Aggiunta e compilata la sezione 4.1.5.1. Aggiunte informazioni alle sezioni 2, da 3.1.1 a 3.1.5, 3.2.1, 3.2.2.1 e da 3.2.3.6 a 3.2.5 (sottosezioni incluse).
0.3.0	25/04/2025	R. Zangla	S. Marana	Ulteriore ristrutturazione del documento e aggiunte note. Modificate le sezioni 4.2 e 8.4.1.

Vers.	Data	Autore	Verificatore	Descrizione
				Aggiunte informazioni alle sezioni da 4.2.2 a 4.2.4 (sottosezioni incluse), da 5.1.2.1 a 5.1.2.4, 5.2.1, 5.2.3 e 5.2.5.
0.2.0	06/04/2025	S. Speranza	S. Marana	Ristrutturazione del documento, ulteriore stesura
0.1.0	29/03/2025	N. Bolzon	S. Marana	Inizio stesura del documento
0.0.1	28/03/2025	S. Speranza	S. Marana	Creazione template e struttura del documento

Indice

Lista delle Tabelle

1. Introduzione

1.1. Scopo del documento

Il presente documento viene redatto per descrivere il *Way of Working*[G] adottato dal Team *Code Alchemists* per lo svolgimento del progetto didattico di Ingegneria del Software. Esso stabilisce le linee guida, i processi, le metodologie e gli standard inerenti ogni attività associata al ciclo di vita del software, così da garantire coerenza, efficienza, efficacia e qualità.

1.2. Scopo del prodotto

Il prodotto[G], detto anche software, oggetto del presente progetto, consiste nella realizzazione di un sistema distribuito e scalabile per la gestione ottimale (intesa come minimizzazione dei tempi di risposta e di ottimizzazione della distribuzione delle scorte[G]) di una rete di magazzini. Ogni singolo magazzino è autonomo, così da favorire l'interoperabilità tra di essi e centralizzare le informazioni in maniera efficiente e sicura.

Il presente documento è redatto secondo lo standard *ISO/IEC 12207:1995*, che identifica tre tipologie di processi:

- **Processi primari**;
- **Processi organizzativi**;
- **Processi di supporto**.

1.3. Entità coinvolte

Il presente progetto didattico di Ingegneria del Software coinvolge le seguenti entità:

- Il proponente del progetto, *M31*, nei ruoli di:
  - *Cliente*[G]
  - *Mentore*[G]
- Il docente, Prof. Tullio Vardanega, nel ruolo di *Committente*[G]
- Il team *Code Alchemists*, nel ruolo di *Fornitore*[G]

1.4. Glossario

Le parole contrassegnate in apice con la lettera [G] sono intese con la loro definizione specificata nel documento *Glossario*[G].

1.5. Riferimenti

1.5.1. Riferimenti normativi

- **Capitolato[G] d'appalto**
  C6 - Sistema di Gestione di un Magazzino Distribuito
  https://www.math.unipd.it/~tullio/IS-1/2024/Progetto/C6.pdf
  Ultimo Accesso: 17 Luglio 2025
- **Standard ISO/IEC 12207:1995**
  https://www.math.unipd.it/~tullio/IS-1/2009/Approfondimenti/ISO_12207-1995.pdf
  Ultimo Accesso: 17 Luglio 2025

1.5.2. Riferimenti informativi

- **Glossario[G]**
  https://teamcodealchemists.github.io/glossario.html
  Ultimo Accesso: 17 Luglio 2025

- **Piano di Qualifica**[G]
  https://teamcodealchemists.github.io/docs/rtb/PdQ.pdf
  Ultimo Accesso: 17 Luglio 2025
- **Piano di Progetto**
  https://teamcodealchemists.github.io/docs/rtb/PdP.pdf
  Ultimo Accesso: 17 Luglio 2025

2. Struttura dei processi

Il processo sarà strutturato secondo le linee guida dello standard **ISO/IEC 12207:1995**, che definisce il ciclo di vita del software suddividendolo in 3 categorie principali:

- I **processi primari** riguardano le attività fondamentali di sviluppo, acquisizione e manutenzione del software;
- I **processi di supporto** forniscono strumenti e metodologie per garantire qualità, verifica[G], convalida e gestione della configurazione;
- I **processi organizzativi**, infine, includono la gestione del progetto, la formazione del personale e il miglioramento continuo, assicurando un approccio sistematico ed efficiente alla produzione software.

3. Processi primari

3.1. Fornitura

3.1.1. Scopo

Il processo di fornitura è l'insieme di attività che regolano la collaborazione tra fornitore[G] e cliente[G] per la creazione e consegna di un prodotto[G] software.

Attraverso l'analisi dei requisiti[G], la pianificazione delle operazioni e la gestione delle risorse, questo processo garantisce il rispetto dei tempi, dei costi e degli standard di qualità.

La chiave del successo è un dialogo continuo tra le parti, che permette di risolvere le difficoltà tecniche, adattare le strategie di sviluppo e assicurare che il prodotto[G] finale soddisfi le aspettative del cliente[G] (M31). L'obiettivo primario è ottimizzare l'efficienza e garantire la conformità agli accordi stabiliti.

3.1.2. Comunicazione con l'azienda proponente

Il team *Code Alchemists* ritiene essenziale mantenere un dialogo continuo con l'azienda *M31.* Tale dialogo, infatti, garantisce un allineamento efficace sulle esigenze del progetto, prevenire incomprensioni e agevolare la risoluzione di eventuali criticità. A tal fine, verranno programmati incontri su Microsoft teams, integrati da una continua comunicazione asincrona tramite email.

Le interazioni con il proponente saranno focalizzate su aspetti chiave quali definire i requisiti[G], pianificare le consegne, raccogliere di feedback, gestire le problematiche tecniche e la definizione delle priorità.

L'obiettivo principale è favorire una collaborazione strutturata, assicurando la qualità del prodotto[G] finale e la piena conformità alle aspettative del cliente[G].

3.1.3. Piano di Progetto

Il Piano di Progetto è un documento strategico che guida la pianificazione e l'esecuzione del progetto, garantendo un utilizzo ottimale delle risorse e il rispetto degli obiettivi.

Redatto dal Responsabile[G] con il supporto degli Amministratori[G], include l'analisi dei rischi, la metodologia di sviluppo basata su Scrum[G], la roadmap delle attività e la gestione dei costi. Inoltre, il piano prevede monitoraggio[G] continuo attraverso un confronto costante tra il preventivo e il consuntivo volto ad analizzare la fase attuale e migliorare le fasi future.

Questo piano facilita la comunicazione tra fornitore[G] e proponente, assicurando trasparenza e coordinazione efficace durante l'intero processo.

3.1.4. Piano di Qualifica

Il Piano di Qualifica[G] definisce le strategie per garantire la qualità del prodotto[G], stabilendo standard di sviluppo, criteri di verifica[G] e test per la conformità ai requisiti[G].

Redatto dall'Amministratore[G], include metodologie di Verifica[G] e Validazione[G], il monitoraggio[G] delle metriche e gli esiti dei test, assicurando il rispetto degli obiettivi qualitativi e il miglioramento continuo del progetto.

3.1.5. Strumenti

Di seguito sono elencati i software utilizzati nel contesto del processo di fornitura:

- **Discord**: piattaforma impiegata per lo svolgimento delle riunioni interne tra i membri del team;
- **Microsoft Teams**: piattaforma utilizzata per le riunioni con l'azienda proponente;

- **Telegram**: strumento adoperato per la comunicazione informale tra i membri del team;
- **Jira**: sistema di issue tracking adottato per la gestione e l'organizzazione delle diverse attività;
- **Typst**: sistema utilizzato per la redazione della documentazione;
- **Google Presentazioni**: strumento impiegato per la realizzazione delle presentazioni;
- **Google Calendar**: applicativo utilizzato per pianificare e monitorare gli incontri, sia interni che con l'azienda;
- **GitHub**: piattaforma utilizzata per la gestione del codice sorgente, il controllo delle versioni e la collaborazione tra i membri del team.

3.2. Sviluppo

3.2.1. Scopo

Il processo di sviluppo software è l'insieme delle attività necessarie per trasformare un'idea in un prodotto[G] funzionante, garantendo la conformità ai requisiti[G] e agli standard di qualità.

Si articola in tre fasi principali: *Analisi dei Requisiti*[G], *Progettazione* e *Codifica*.

Il processo di sviluppo ha il compito di assicurare una pianificazione strutturata e un'ottimizzazione delle risorse. Il software deve rispettare vincoli tecnologici, obiettivi di design e superare test di verifica[G] e validazione[G]. La documentazione, invece, facilita la gestione e il mantenimento del progetto nel tempo.

3.2.2. Analisi dei Requisiti

3.2.2.1. Scopo

L'Analisi dei Requisiti[G] è la fase preliminare dello sviluppo software, finalizzata a identificare e documentare in modo accurato le esigenze del proponente e degli utenti. Questo processo consente di definire il fine del prodotto[G], gli attori del sistema e le funzionalità chiave, fornendo ai progettisti[G] una visione chiara del problema e ai verificatori[G] una base per le attività di test.

L'analisi si concretizza in un documento strutturato che raccoglie i casi d'uso, i requisiti[G] funzionali e non funzionali, e le fonti da cui sono stati derivati. Questo documento rappresenta un riferimento essenziale per la progettazione, la pianificazione e la fase di verifica[G], garantendo che il sistema soddisfi pienamente le aspettative del cliente[G] e gli obiettivi stabiliti. Inoltre, è strettamente collegato al Piano di Qualifica[G], che consente di monitorare la conformità ai requisiti[G] attraverso test specifici.

3.2.2.2. Casi d'uso

I casi d'uso sono codificati utilizzando la seguente notazione:

UC [ID-Principale].[ID-Sottocaso]

Tale notazione fa uso dell'acronimo «UC» (cioè «Use Case», ovvero «Caso d'Uso») e di un identificativo univoco del caso d'uso, composto da:

- un ID principale che identifica il caso principale
- e, se necessario, da un ID del sottocaso.

Ogni caso d'uso è costituito da un diagramma UML e da una descrizione testuale dettagliata, utile a chiarire le funzionalità attese dal sistema. La descrizione riporta, inoltre, le informazioni previste nella tabella sottostante, fatta eccezione per i campi che, in base alla natura del caso d'uso, risultano non applicabili *(ad esempio, se non sono previste situazioni di errore, non saranno presenti scenari alternativi)*.

Campo	Descrizione
Titolo	Breve descrizione del caso d'uso.
Attori	Sono coloro che interagiscono attivamente con il Sistema e svolgono l'azione indicata dal caso d'uso.
Attori Secondari	Sono coloro che interagiscono passivamente con il Sistema.
Precondizioni	Lista di condizioni che sono **necessarie** affinché l'attore possa compiere l'azione indicata dal caso d'uso.
Postcondizioni	Lista di condizioni che si verificano successivamente alla modifica dello stato del sistema, a seguito dell'azione eseguita con successo dall'attore secondo quanto previsto dal caso d'uso.
Scenario Principale	È la sequenza di iterazioni ideale tra l'attore e il sistema, in cui tutto procede come previsto e l'obiettivo del caso d'uso viene raggiunto con successo.
Scenario Alternativo	Sono variazioni dello scenario principale che si verificano quando una delle operazioni previste non va a buon fine.
Inclusioni	Indicano che un caso d'uso utilizza un altro caso d'uso. Servono a spezzare comportamenti comuni tra più casi d'uso, inserendoli in uno separato che viene "incluso" quando serve.
Estensioni	Indicano che un caso d'uso può estendere il comportamento di un altro in situazioni particolari. Il caso d'uso base funziona da solo, ma può essere arricchito opzionalmente da quello estensore, se si verifica[G] una certa condizione.
Trigger	È l'evento iniziale che fa partire il caso d'uso. Può essere un'azione dell'utente, un evento di sistema o un cambiamento di stato che attiva il comportamento descritto nel caso d'uso.

Tabella 1: Componenti del caso d'uso e loro descrizione

3.2.2.3. Requisiti

I requisiti[G] vanno codificati in modo da facilitarne la lettura e la comprensione. Verranno suddivisi in quattro categorie principali: Requisiti[G] Funzionali, Requisiti[G] di Qualità, Requisiti[G] di Vincolo, Requisiti[G] Prestazionali.

3.2.2.3.1. Classificazione dei requisiti

- **Requisiti[G] Funzionali:** descrivono le funzionalità specifiche che il sistema deve offrire. Definiscono i comportamenti attesi in risposta a determinati input o situazioni, specificando cosa il sistema deve fare per soddisfare i bisogni degli utenti e degli stakeholder.
- **Requisiti[G] di Qualità:** detti anche non funzionali, definiscono le caratteristiche generali del sistema che ne influenzano l'efficacia, l'efficienza e l'affidabilità. Rientrano in questa categoria aspetti come la sicurezza, l'usabilità, la manutenibilità, la scalabilità[G] e l'affidabilità complessiva del sistema.
- **Requisiti[G] di Vincolo:** specificano le limitazioni imposte da fattori esterni o immutabili, che il sistema o il processo di sviluppo devono rispettare. Tali vincoli possono derivare da normative, tecnologie obbligatorie, standard industriali, vincoli temporali o economici.
- **Requisiti[G] di Prestazionali:** definiscono le aspettative in termini di prestazioni del sistema, come tempi di risposta, capacità di carico, throughput e uso delle risorse. Questi requisiti[G] sono fondamentali per garantire un'esperienza utente adeguata anche sotto carico elevato.

3.2.2.3.2. Fonti dei requisiti

Le fonti dei requisiti[G] rappresentano i documenti e le informazioni da cui vengono estratti i requisiti[G] stessi. Tra le principali fonti si annoverano il capitolato[G] d'appalto, le riunioni con il committente[G], l'analisi dello stato dell'arte e l'analisi dei casi d'uso.

Ogni requisito[G] riportato è accompagnato dall'indicazione esplicita della propria fonte di provenienza, al fine di garantirne la tracciabilità e la verificabilità[G].

3.2.2.3.3. Struttura della codifica dei requisiti

I requisiti[G] vanno codificati al fine di facilitarne la lettura, la gestione e la tracciabilità. Ogni codice è composto da un prefisso che indica la tipologia del requisito[G], seguito da un numero progressivo univoco.

I requisiti[G] funzionali sono preceduti dal prefisso «RF», i Requisiti[G] di Qualità dal prefisso «RQ», i Requisiti[G] di Vincolo dal prefisso «RV» e i Requisiti[G] Prestazionali dal prefisso «RP», dove:

- **R** sta per «**R**equisito»;
- **F** sta per «**F**unzionale»;
- **Q** sta per «**Q**ualità»;
- **V** sta per «**V**incolo»;
- **P** sta per «**P**restazionale»;

Per facilitare la lettura, la tracciabilità e la classificazione dei requisiti[G], si adotta un sistema di codifica strutturato. La codifica prevede un prefisso che identifica la tipologia e l'importanza del requisito[G], seguito da un numero progressivo. In caso di scomposizione, si aggiunge una notazione per indicare i requisiti derivati.

3.2.2.3.4. Tipologia e Importanza

I requisiti[G] si distinguano anche in base alla loro importanza o natura, secondo le seguenti convenzioni:

- **Standard**: requisiti[G] strettamente necessari al corretto funzionamento del sistema.
  - *Esempio:* RF01 → Requisito Funzionale 01.
- **Desiderabili (D)**: requisiti[G] non obbligatori, ma in grado di apportare un valore aggiunto al sistema.
  - *Esempio:* RFD04 → Requisito Funzionale Desiderabile 02.
- **Opzionali (O)**: requisiti[G] implementabili solo in presenza di tempo o risorse sufficienti.
  - *Esempio:* RFO03 → Requisito Funzionale Opzionale 02.

3.2.2.3.5. Scomposizione dei Requisiti Generali

Qualora alcuni requisiti derivanti dal capitolato[G] risultassero generici, è necessario **scomporli** in requisiti[G] più specifici, che chiariscano chi deve fare cosa e in quale modalità. Per indicare questa relazione di derivazione, verrà utilizzata la notazione **«/nr»**, dove *nr* rappresenta un numero progressivo riferito al requisito[G] secondario.

Esempio:

- RF04 → Requisito principale.
- RF04/01, RF04/02 → Requisiti[G] secondari specifici derivati dal requisito RF04.

3.2.2.3.6. Sintesi della struttura del codice

[Prefisso][Indicatore opzionale][Numero progressivo][/nr]

- **Prefisso:** RF (Funzionale), RQ (Qualità), RV (Vincolo), RP (Prestazionale).
- **Indicatore opzionale:** D (Desiderabile), O (Opzionale).
- **Numero progressivo:** numero univoco del requisito[G] all'interno della categoria.
- **/nr:** numero del requisito[G] secondario, se presente.

3.2.3. Progettazione

3.2.3.1. Scopo

La progettazione software definisce le soluzioni tecniche per soddisfare i requisiti[G] individuati, trasformandoli in un'architettura[G] strutturata e modulare[G].

Si suddivide in:

- **Progettazione Logica**, che motiva la scelta di tecnologie e framework e include il PoC[G]
- **Progettazione di dettaglio**, che specifica l'architettura[G] con diagrammi delle classi e test di unità.

Per garantire un sistema efficiente e scalabile, la progettazione segue principi di modularità[G], flessibilità e affidabilità, assicurando una struttura chiara e facilmente mantenibile prima della fase di sviluppo.

3.2.3.2. Fasi di Progettazione

3.2.3.2.1. Progettazione logica

La progettazione logica definisce la struttura di alto livello del software, traducendo i requisiti[G] in un'architettura[G] chiara e coerente. Comprende la scelta di framework, tecnologie e librerie,

validandone l'adeguatezza tramite un Proof of Concept[G] (PoC[G]). Inoltre, include diagrammi UML per rappresentare l'interazione tra i componenti.

I progettisti[G] devono garantire l'assegnazione e il dettaglio dei requisiti[G], progettare interfacce e strutture dati, definire test di integrazione e revisionare l'architettura[G] in collaborazione con il team e il cliente[G], assicurando che il sistema sia ben strutturato e facilmente mantenibile.

3.2.3.2.2. Progettazione di dettaglio

La progettazione di dettaglio suddivide il sistema in unità architetturali funzionali, garantendo una codifica efficiente e verificabile. Essa assicura che i componenti software siano ben definiti e coerenti con la progettazione logica, evitando complessità eccessive.

Durante questa fase, vengono creati diagrammi delle classi, tracciate le componenti e definiti gli unit test e integrazione, supportando un'implementazione strutturata e affidabile del prodotto[G].

3.2.3.3. Specifica tecnica

La specifica tecnica è un riferimento essenziale per lo sviluppo software, guidando le scelte architetturali e tecnologiche in modo strutturato. Essa definisce l'architettura[G] del sistema, descrivendo componenti e interfacce, e indica le tecnologie adottate, come linguaggi di programmazione e framework. Inoltre, dettaglia la gestione dei dati, le interfacce con sistemi esterni e i design pattern utilizzati, garantendo efficienza e coerenza progettuale.

Il documento include anche la pianificazione delle attività, con stime di tempi e risorse, e le procedure di testing, assicurando che il prodotto[G] finale rispetti i requisiti[G] tecnici e le aspettative del cliente[G].

3.2.3.4. Framework e tecnologie utilizzate

- **NATS**: Poiché l'architettura del sistema si basa su una comunicazione asincrona tramite messaggi, è necessario adottare un message broker. La scelta è ricaduta su NATS, in quanto si distingue per la sua elevata velocità e bassa latenza, resa possibile anche dal fatto che non garantisce l'ordinamento dei messaggi in fase di consegna. Inoltre, NATS si contraddistingue per la sua semplicità di utilizzo e per un'architettura leggera, risultando ideale per ambienti distribuiti ad alte prestazioni.
- **NestJS**: Sebbene ogni microservizio possa adottare uno stack differente, NestJS rappresenta la scelta privilegiata per la sua perfetta integrazione con NATS, che consente una comunicazione fluida e a bassa latenza tra i servizi. Basato su Node.js e potenziato da TypeScript, NestJS offre una tipizzazione forte e una struttura modulare che favorisce la scalabilità, la manutenibilità e l'evoluzione indipendente dei singoli servizi. Le sue astrazioni semplificano la realizzazione di architetture distribuite complesse, mentre il sistema di trasporto altamente estensibile supporta sia la comunicazione asincrona che sincrona tra microservizi. Questo lo rende particolarmente adatto a sistemi reattivi, dinamici ed eterogenei, dove l'interoperabilità con componenti esterni è un requisito fondamentale.
- **Docker**: Per la fase di deployment, è stata adottata la tecnologia Docker, in quanto rappresenta lo standard de facto per la containerizzazione. Docker consente di isolare le applicazioni all'interno di container leggeri e indipendenti, rendendone la distribuzione più portabile, riproducibile e coerente tra diversi ambienti.

3.2.3.5. PoC

Il Proof of Concept[G] (PoC[G]) è una fase fondamentale dello sviluppo in cui progettisti[G] e programmatori[G] valutano la validità di una soluzione prima della sua implementazione definitiva. Il suo scopo è dimostrare la fattibilità del progetto, assicurando che le tecnologie scelte siano

adeguate e compatibili con i requisiti[G] tecnici e operativi. Inoltre, offre all'azienda proponente un feedback tempestivo e concreto, minimizzando i rischi e ottimizzando le decisioni. Per il team di sviluppo, il PoC[G] rappresenta anche una fase di studio e sperimentazione delle tecnologie coinvolte, in cui vengono analizzati i limiti, le potenzialità e le integrazioni tra i diversi componenti del sistema. Ciò consente di acquisire familiarità con gli strumenti adottati, validare le scelte architetturali iniziali e definire le basi tecniche per le successive fasi progettuali.

3.2.4. Codifica

3.2.4.1. Scopo

La codifica, affidata ai programmatori[G], è una fase essenziale dello sviluppo software, finalizzata a tradurre le scelte progettuali in codice[G] sorgente funzionante.

Per garantire uniformità, leggibilità e manutenibilità, vengono seguite precise linee guida sulla formattazione, la nomenclatura e la struttura del codice[G]. Queste regole contribuiscono a semplificare la verifica[G], il debugging e l'estensione del software, assicurando un prodotto[G] finale robusto e di alta qualità.

3.2.4.2. Convenzioni di sintassi

Le norme di scrittura del codice[G] del prodotto[G] software mirano a garantire leggibilità, uniformità e manutenibilità. Di seguito sono definite le principali convenzioni da seguire.

3.2.4.2.1. Lingua

La lingua utilizzata per scrivere i commenti al codice e i nomi dei metodi, di variabili, classi e tipi è l'inglese, in modo da garantire una maggiore comprensibilità e uniformità all'interno del team e per facilitare la collaborazione con altri sviluppatori.

3.2.4.2.2. Formattazione

Ogni riga di codice[G] non deve superare gli 80 caratteri così da mantenere la leggibilità e da rendere più rapida ed efficace la revisione.

3.2.4.2.3. Indentazione

I blocchi di codice[G] innestati devono essere indentati con un livello di tabulazione.

3.2.4.2.4. Spaziatura

Per garantire un codice più chiaro e leggibile possibile, è buona norma inserire degli spazi intorno agli operatori (ad es., scrivere «x = y + z» e non «x=y+z»).

3.2.4.2.5. Importazioni

Tutte le importazioni (import) devono essere dichiarate all'inizio dello script, prima di qualsiasi altro codice. Si raccomanda di importare solo ciò che serve, evitando di usare gli import con asterisco, dove non necessari, e in generale importazioni inutili o ridondanti.

3.2.4.2.6. Commenti

Ogni costrutto significativo deve essere preceduto da un commento esplicativo, indentato di un livello di tabulazione rispetto al codice che descrive.

I commenti sono obbligatori nei punti con logica meno intuitiva e devono essere aggiornati ogni volta che viene modificato un metodo. Per descrivere un metodo, il commento a esso legato deve spiegare lo scopo della funzione stessa e i suoi argomenti.

Inoltre, nelle sezioni di codice da sviluppare e/o da completare, è buona norma scrivere all'inizio del commento «TODO» o «FIXME» per indicare che il codice necessita di ulteriori aggiunte, modifiche o correzioni.

3.2.4.2.7. Parentesi graffe

La parentesi graffa aperta deve trovarsi nella stessa riga della dichiarazione del costrutto, separata da uno spazio; invece, la parentesi graffa chiusa va messa subito dopo l'ultima istruzione del costrutto e nella sua stessa riga.

3.2.4.2.8. Metodi

I nomi dei metodi devono essere scritti utilizzando il Camel Case, convenzione per cui la prima parola[G] ha iniziale minuscola e le successive iniziali sono maiuscole. Cercare di evitare di scrivere funzioni troppo lunghe e complesse, bensì preferire scrivere metodi brevi e concisi.

3.2.4.2.9. Istruzioni

Evitare la scrittura di più istruzioni su una singola linea; in generale, infatti, ogni istruzione deve essere scritta su una riga separata, in modo da migliorare la leggibilità e la comprensione del codice[G].

3.2.4.2.10. Classi e tipi

Le classi e i tipi devono essere nominati usando il Pascal Case, convenzione per cui ogni parola[G] inizia con una lettera maiuscola.

3.2.4.2.11. Variabili e costanti

Il nome delle variabili e delle costanti deve seguire lo stile devono essere scritti utilizzando il Camel Case. Ogni variabile deve essere dichiarata all'inizio della funzione o dello script in modo tale da migliorare la leggibilità, così come per le costanti.

3.2.4.2.12. Univocità e chiarezza dei nomi assegnati

Ogni costrutto deve avere un nome chiaro, univoco e consistente, così da garantire la comprensione immediata della variabile o della funzione. Il nome scelto deve essere nè troppo generico, nè troppo lungo e dettagliato, bensì deve essere conciso e descrittivo in modo da riflettere il suo scopo principale e la sua funzionalità. Tale nomenclatura favorisce la rapida comprensione del codice, riducendo il più possibile il bisogno di commenti esplicativi aggiuntivi.

3.2.4.2.13. File

3.2.4.2.13.1. Nomenclatura dei file

Per garantire coerenza, leggibilità e chiarezza all'interno del progetto, i file devono essere nominati seguendo la convenzione strutturata nel formato:

nome.tipo.estensione

- **nome** descrive il contenuto:
  - Se è una sola parola, si usa il **PascalCase**
  - Se è composto da più parole, si usa il **camelCase**
- **tipo** indica la funzione del file.
- **estensione** definisce il formato, ad esempio .ts, .json

3.2.4.3. Tecnologie utilizzate

Visual Studio Code (o VS Code) è un potente ambiente di sviluppo (IDE) ed è stato adottato come principale strumento di sviluppo software da parte dell'intero team. Esso supporta

numerosi linguaggi di programmazione e framework. Inoltre, offre funzionalità avanzate come il completamento automatico intelligente, il debugging integrato, il controllo della versione con Git e un ampio ecosistema di estensioni che permette di personalizzare l'esperienza di sviluppo.

4. Processi di supporto

4.1. Documentazione

4.1.1. Scopo

Il processo di documentazione è essenziale per tracciare le attività di sviluppo, facilitare la manutenzione e garantire coerenza nel ciclo di vita del prodotto[G]. Seguendo regole e strutture uniformi, permette una consultazione rapida delle informazioni e supporta il lavoro asincrono, contribuendo alla storicizzazione delle decisioni prese e al miglioramento continuo del software.

4.1.2. Ciclo di vita del documento

Il ciclo di vita di un documento software è strutturato in diverse fasi, le quali garantiscono un processo chiaro e standardizzato per la sua gestione, ovvero:

- **Pianificazione**: organizzazione delle informazioni e definizione della struttura del documento, inclusi intestazioni, header e footer.
- **Redazione**: assegnazione del compito di stesura del documento a un membro del team, utilizzo di strumenti di versionamento (GitHub) per tracciare le modifiche e avvio della stesura dei contenuti.
- **Verifica[G]**: ogni sezione viene revisionata da un membro distinto dal redattore, per garantire accuratezza e qualità. Per la documentazione esterna, viene effettuata una ulteriore validazione[G] dagli enti terzi.
- **Approvazione**: il responsabile[G] di progetto analizza il documento nella sua interezza, verificando la coerenza e l'adeguatezza dei contenuti prima della pubblicazione.
- **Pubblicazione**: il documento viene reso disponibile nel repository[G] GitHub ufficiale «docs» solo dopo una verifica[G] positiva, assicurando la presenza esclusiva di documenti validati e coerenti.

Questa suddivisione garantisce un processo documentale chiaro ed efficace, facilitando la gestione e l'evoluzione dei documenti nel tempo.

4.1.3. Template Typst

Per facilitare la redazione dei documenti, sia interni che esterni, abbiamo creato dei template su Typst che ci aiutano a conferire la struttura generale al documento.

4.1.4. Documenti prodotti

I documenti prodotti durante il ciclo di vita del software apparterranno a due diverse macro categorie:

- **Documenti informali**: essi sono utilizzati per documentare le informazioni (come bozze e appunti vari) e le dinamiche interne del team durante tutto il ciclo di vita del software. Tutti i documenti informali, infatti, sono **interni**, organizzati tramite Google Drive. Essi, però, non sono soggetti nè a versionamento nè a verifica[G].
- **Documenti formali**: essi sono redatti con rigorosa cura e attenzione per documentare le attività e i progressi concreti del team durante tutto il ciclo di vita del software. Tutti i documenti formali sono oggetti a versionamento e verifica[G]. Essi sono suddivisi a loro volta in:
  - Documenti **interni**: destinato ad uso interno da parte dei membri del team (ovvero le Norme di Progetto e i verbali interni).
  - Documenti **esterni**: destinato agli enti esterni, quali l'azienda proponente e il committente[G] (ovvero Piano di Progetto, Piano di Qualifica[G], Analisi dei Requisiti[G], Glossario[G] e i verbali esterni).

4.1.5. Nomenclatura documenti

I file devono possedere dei nomi adeguati, coerentemente con gli acronimi stabiliti nella sezione **4.1.5.1**. I verbali sia interni che esterni, però, fanno eccezione e adottano il formato di nomenclatura «**YYYY_MM_DD_TIPO**» o «**YYYY_MM_DD_TIPO_x**», dove:

- **YYYY-MM-DD**: si riferisce alla data dell'incontro.
- **TIPO**: si riferisce all'acronimo corrispettivo al tipo di verbale (VI o VE).
- **x** (modificatore): indica eventuali versioni alternative del file. Le 2 opzioni disponibili sono:
  - *test*: il documento è in stato di testing tecnico o operativo, da non considerarsi ufficiale;
  - *signed*: i documenti esterni necessitano di una approvazione da parte di un ente terzo. Questo suffisso starà ad indicare che quel documento esterno è stato approvato dall'ente terzo e contiene la corrispondente firma di approvazione.

4.1.5.1. Acronimi per la documentazione

Per facilitare la fruizione dei vari documenti, sono state assegnate delle sigle alle varie tipologie di documento. Gli acronimi adottati sono:

- **VI**: Verbale Interno;
- **VE**: Verbale Esterno;
- **Gls**: Glossario[G]
- **PdQ**: Piano di Qualifica[G]
- **PdP**: Piano di Progetto;
- **AdR**: Analisi dei Requisiti[G]
- **NdP**: Norme di Progetto;
- **DdB**: Diario di Bordo.

4.1.6. Struttura documento

Tutti i documenti formali sono basati su un template e seguono una struttura fissa che deve essere rispettata. Tale struttura prevede:

- **Intestazione**: la prima pagina del documento, che contiene il nostro logo, il nome del documento, lo stato e la versione del documento e la distribuzione (ovvero i destinatari);
- **Registro delle modifiche**: una tabella che tiene traccia delle modifiche significative del documento durante il suo ciclo di vita. Ogni voce della tabella presenta vari dati:
  - **Versione** del documento;
  - **Data** di modifica;
  - **Autore** della versione;
  - **Verificatore[G]** della versione;
  - **Descrizione** delle modifiche apportate.
- **Indice dei contenuti**, il quale viene creato automaticamente invocando l'apposita funzione contenuta nel template del documento.

4.1.7. Convenzioni di scrittura

All'interno dei documenti formali adottiamo delle convenzioni sullo stile del testo, le quali riguardano:

- **Grassetto**: utilizzato per titoli e per le keyword all'interno del contenuto;
- *Corsivo*: utilizzato per evidenziare le entità coinvolte;
- Sottolineatura: utilizzato per i link.

4.1.8. Strumenti per stesura

Per la stesura della documentazione è stato adottato il linguaggio Typst, un linguaggio di markup versatile e relativamente semplice da apprendere. Durante le fasi di redazione, è stato

utilizzato Visual Studio Code come ambiente di sviluppo, grazie al supporto di numerose estensioni, tra cui quella dedicata a Typst, che consente la compilazione dei file con estensione ".typ". In particolare, l'estensione Tinymist Typst ha facilitato il processo di scrittura permettendo la visualizzazione in anteprima del documento in formato ".pdf" direttamente all'interno dell'IDE, con aggiornamenti applicati in tempo reale.

4.1.9. Struttura della documentazione

La nostra repository[G] *docs* contiene al suo interno la cartella «documents», la quale contiene tutti i documenti redatti, ed è a sua volta è suddivisa in varie sottocartelle, tra cui:

- **«candidatura/»**: raccoglie tutti i documenti redatti in fase di candidatura;
- **«glossario/»**: contiene il file .typ del glossario[G]
- **«presentazioni/»**: raccoglie tutti i file in formato .pdf dei vari Diari di Bordo redatti;
- **«rtb/»**: contiene al suo interno in formato .typ i vari documenti informali e formali (interni ed esterni), come ad esempio l'AdR oppure i vari verbali;
- **«src/»**: contiene al suo interno i template che vengono importati in altri file .typ.

4.1.9.1. Elementi grafici

Le immagini e vari elementi grafici che compaiono all'interno dei documenti, come ad esempio il nostro logo, vengono raccolte all'interno delle cartelle denominate «assets».

4.1.10. Metriche

I documenti sono sottoposti a dei controlli di qualità, tra cui:

- **Parole[G] del Glossario[G] non segnate**: Verifica il numero di termini presenti nella documentazione che compaiono anche nel Glossario[G], ma che non sono opportunamente marcati con l'apice "G".
- **Indice Gulpease**: Calcola un punteggio compreso tra 0 e 100, basato sull'indice di leggibilità Gulpease, al fine di valutare la chiarezza e l'accessibilità del testo redatto.
- **Ordinamento del Glossario[G]**: Controlla che i termini presenti nel Glossario[G] siano disposti in ordine alfabetico, garantendo coerenza e facilità di consultazione.

4.1.11. Manutenzione

La manutenzione della documentazione è fondamentale per aggiornare il documento con informazioni nuove e/o più accurate. Il processo di manutenzione prevede le seguenti fasi:

- **Identificazione della modifica**: viene controllato il documento e viene individuata una possibile modifica;
- **Valutazione dell'impatto**: viene valutato l'impatto migliorativo che può avere la modifica rispetto alla versione originale del documento;

Tale discussione coinvolge tutto il team se necessario;

- **Aggiornamento della documentazione**: una volta constatato che l'impatto migliorativo è positivo, viene applicata la modifica e il documento viene aggiornato confermando l'accuratezza delle informazioni e l'allineamento delle modifiche con quanto già presente;
- **Push della modifica**: viene creata una pull request nella repository[G] su GitHub per la nuova versione del documento da parte di chi ha operato la modifica;
- **Verifica e validazione**: a supporto del processo di verifica[G], viene fatta partire un'azione automatizzata che esegue diversi script di controllo del testo, le cui metriche si possono trovare alla sezione **4.1.10**. Se tale operazione ha esito positivo, se necessario, il documento viene posto ad ulteriore validazione[G] con l'ente esterno a cui è destinato il documento. Se sia la verifica[G] che la eventuale validazione[G] hanno avuto esito positivo, le modifiche vengono

pubblicate attraverso l'accettazione della pull request da parte del revisore. Se una delle due operazioni non ha esito positivo, la pull request viene respinta e l'autore delle modifiche è incaricato di apportare le correzioni necessarie e di ripetere i passaggi precedentemente descritti.

4.1.11.1. Procedure di revisione

Il processo di revisione è un'attività fondamentale per verificare la conformità del lavoro svolto agli standard di qualità e ai requisiti[G] fondamentali. Questa fase prevede diverse operazioni: la revisione del codice[G], l'esecuzione dei test di accettazione e la risoluzione di eventuali discrepanze.

Il materiale da sottoporre a revisione viene definito e assegnato ai verificatori tramite Jira.

4.1.11.1.1. Revisione tecnica

La revisione tecnica serve a garantire che il software sia corretto, sicuro e conforme agli standard. Il codice[G] viene analizzato allo scopo di individuare errori, migliorare le prestazioni e assicurare la manutenibilità del sistema. Inoltre, si effettuano test per verificare la stabilità e il rispetto dei requisiti[G] tecnici.

4.1.11.1.2. Revisione editoriale

La revisione editoriale ha lo scopo di migliorare la documentazione, rendendola chiara, coerente e precisa. Viene verificata la correttezza di grammatica, sintassi, uniformità stilistica e coerenza terminologica. I documenti vengono verificati per garantirne una struttura comprensibile e un linguaggio efficace.

4.1.11.1.3. Approvazione

- L'autore del documento, o della modifica, segnala tramite Jira al Verificatore[G] la necessità di una revisione del contenuto;
- Il Verificatore[G], ricevuta la notifica, esamina il documento e lo sposta nella fase «Revisione» all'interno di Jira;
- Al termine dell'attività di revisione, la relativa task viene contrassegnata come completata e il documento può essere sottoposto al processo di approvazione finale;
- Il Responsabile[G] procede con l'approvazione del documento chiudendo la relativa task su Jira e completando così il processo di gestione.
- Il registro delle modifiche viene aggiornato, riportando il numero di versione e contrassegnando lo stato del documento come «Approvato»;
- Nel caso di Verbali Esterni, viene generato il file in formato ".pdf" e inviato al destinatario designato, il quale approva il documento mediante firma e lo restituisce, autorizzandone così il caricamento nella repository[G] come documento approvato;
- Infine, il Responsabile[G] esegue la release finale del documento, unendo il branch[G] di lavoro al main. Una volta completata la release sul ramo principale, è necessario avviare manualmente l'azione di aggiornamento dei collegamenti del sito contenente la documentazione, al fine di rendere disponibile pubblicamente il documento approvato in formato ".pdf".

4.2. Gestione della Configurazione

4.2.1. Scopo

La Gestione della Configurazione descrive il versionamento dei vari documenti e la struttura della repository[G]. Lo scopo di questo processo è garantire che ogni modifica sia tracciabile, controllata e approvata prima di essere integrata nel prodotto[G] finale.

4.2.2. Versionamento

Il sistema di versionamento dei documenti seguirà la seguente semantica:

MAJOR.MINOR.PATCH

dove:

– **MAJOR**: il valore viene incrementato **solamente** alla pubblicazione finale, dopo approvazione interna e, per i verbali esterni, dopo anche l'approvazione esterna;

– **MINOR**: il valore viene incrementato **solamente** quando viene completato il processo di modifica e di verifica[G]

– **PATCH**: il valore viene incrementato **solamente** per modifiche di entità minore, quali correzioni ortografiche, di sintassi…

4.2.3. Struttura repository

La repository[G] GitHub configurata ad Organizzazione è strutturato in diverse repository[G] in base alla finalità d'uso delle stesse. All'interno di esse è possibile trovare varie directories descritte dai file README.md presenti, tra cui:

- **docs**: repository[G] contenente tutti i file .typ con la documentazione. In essa sono presenti delle Github Action che permettono il supporto alla verifica[G] e l'auto compilazione dei file per essere caricati su Github Pages come .pdf;
- **teamcodealchemists.github.io**: repository[G] dove si trova il sito web con tutta la documentazione. All'interno di essa sono state definite delle GitHub Action che permettono di tenere sincronizzato il glossario[G] nel sito con quello in formato .pdf della documentazione e di aggiornare automaticamente i link alla documentazione.
- **poc**: repository[G] contenente il PoC[G] del progetto, che include le tecnologie e le architetture[G] scelte. Essa è stata creata per testare le tecnologie e le architetture[G], così da poterle validare prima della loro implementazione definitiva nel progetto.

4.2.3.1. Organizzazione Repository

4.2.3.1.1. Modello di branching

È stato adottato il modello di branching **Git Flow** per la gestione delle repository[G] di progetto.

Ciascuna tipologia di ramo risponde a uno scopo specifico all'interno del ciclo di vita del software, consentendo una gestione ordinata e controllata del processo di sviluppo.

Ogni membro è a conoscenza delle *best practice* per la creazione, l'unione e la cancellazione dei rami, così da promuovere una collaborazione efficace e strutturata all'interno del team.

4.2.3.1.2. Nomenclatura dei branch

È stata raccomandata, ove possibile, l'associazione dei branch[G] relativi a nuove funzionalità (ovvero i branch[G] *feature*) alle corrispondenti issue tracciate su **Jira**, adottando una convenzione di nomenclatura coerente *(ad esempio: **feature/Sprint[G]−55-nome-feature**)*.

4.2.3.1.3. Branch principali

La struttura è basata sui rami principali *main* e *develop*.

Il ramo *main* è destinato alle versioni stabili del progetto, verificate, testate e pronte per essere rilasciate. Ogni commit[G] su *main* dovrebbe quindi rappresentare una versione stabile e testata del software e/o del documento in questione.

Il ramo *develop*, invece, è destinato a raccogliere tutte le nuove funzionalità che saranno incluse nel prossimo rilascio. I commit[G] sul *develop* possono contenere codice[G] che si trova ancora in fase di sviluppo e test.

4.2.3.1.4. Feature branch

I branch[G] secondari *feature* sono dedicati allo sviluppo delle singole nuove funzionalità. Essi devono essere creati a partire dal ramo *develop* e, una volta completati, deve essere effettuato il merge[G] in *develop*.

La convenzione di nomenclatura per questi branch[G] è consultabile alla sezione **4.2.4.1.2** («Nomenclatura dei branch[G]»).

4.2.4. GitHub Action

Al fine di automatizzare e garantire la qualità del processo di produzione documentale, il flusso di lavoro è stato strutturato mediante l'utilizzo di GitHub Actions, suddivise in più controlli specifici. In particolare:

- Viene effettuato un controllo sul glossario, per verificarne l'ordinamento alfabetico e garantire che tutte le voci siano coerentemente strutturate
- Un secondo controllo analizza i documenti, assicurandosi che ogni parola contenuta nel glossario sia effettivamente assegnata e utilizzata nei testi in modo corretto
- Viene calcolato l'indice di Gulpease, al fine di valutare il livello di leggibilità dei documenti prodotti

4.2.5. Best Practice su GitHub

Al fine di garantire un flusso di lavoro strutturato, tracciabile e facilmente mantenibile, sono state adottate le seguenti best practice nell'utilizzo di GitHub:

- **Pull Request verso branch di sviluppo** : ogni modifica viene integrata tramite pull request indirizzate al branch feature-develop. Questa procedura consente una revisione preventiva da parte di un verificatore, il quale ha il compito di controllare la correttezza, la qualità e la coerenza delle modifiche proposte prima della loro integrazione.
- **Commit verbosi e significativi** : si è scelto di utilizzare messaggi di commit chiari, descrittivi e coerenti con le modifiche effettuate. Questo approccio facilita la lettura della cronologia del progetto, la comprensione degli intenti delle modifiche e l'individuazione di eventuali regressioni o problemi
- **README completi e informativi** : ogni repository è corredata da un file README.MD esaustivo, che descrive in maniera dettagliata :
  - Lo scopo del progetto o della repository;
  - Le istruzioni per la configurazione e l'esecuzione;
  - Eventuali dipendenze;
  - La struttura della directory;
  - Ogni altra informazioni utile alla comprensione e al corretto utilizzo del progetto.

4.2.6. Procedure di commit e merge

Le procedure di commit vengono eseguite ogniqualvolta si completi una frase di scrittura del codice che sia stata verificata e testata in modo funzionale. In alternativa, è possibile eseguire commit intermedi in stato di pending, qualora si sia in attesa dell'intervento di un altro

membro del team, con cui si intende integrare le rispettive parti di codice all'interno della stessa funzionalità. Il merge viene effettuato solo quando si è certi che le feature sviluppata all'interno del branch dedicato sia stata completata e pienamente validata. Dopo aver condotto i necessari test di verifica, si procede con l'unione del codice: il contenuto del branch **feature/[nome-branch]** viene fuso con il branch **develop**, assicurando così l'integrazione stabile e controllata delle nuove funzionalità nel ramo di sviluppo principale

4.2.6.1. Gestione dei conflitti

Durante la fase di merge, può accadere che si verifichino dei conflitti su una o più righe di codice all'interno dei file modificati. Questo si verifica quando Git non è in grado di determinare automaticamente quale versione del contenuto mantenere, a causa di modifiche concorrenti effettuate su uno stesso frammento di codice da branch differenti. In tali casi, la risoluzione dei conflitti richiede un intervento manuale da parte dei membri del team. I colleghi incaricati dovranno analizzare il contenuto, modificare il file risolvendo i funzionante del codice. Una volta risolti tutti i conflitti, è necessario eseguire il git add dei file modificati e completare l'operazione di merge tramite git commit.

4.3. Gestione della qualità

4.3.1. Scopo

La Gestione della Qualità di progetto assicura che i processi e il prodotto[G] rispettino gli obiettivi e i requisiti[G] definiti. Il team *Code Alchemists*, dunque, lavora per comprendere e gestire le aspettative del committente[G], definendo chiaramente i requisiti[G] di qualità e documentando tutte le procedure necessarie. L'obiettivo è completare il progetto nel rispetto delle tempistiche e del budget, garantendo un prodotto[G] finale che sia pienamente in linea le aspettative.

Per mantenere elevati standard di qualità, adottiamo un approccio di miglioramento continuo, monitorando i progressi e utilizzando verifiche retrospettive per identificare opportunità di ottimizzazione. Questo metodo ci permette di affinare costantemente i nostri processi e garantire risultati sempre più performanti.

4.3.2. Piano di Qualifica

Il Piano di Qualifica[G] è un documento che definisce strategie, obiettivi e attività per garantire la qualità del prodotto[G] e del processo di sviluppo. Tali parametri vengono stabiliti in accordo ai requisiti[G] e alle aspettative del proponente e talvolta a discrezione del team sulla base delle valutazioni fatte nel corso di studi. Il suo scopo principale è assicurare che il prodotto[G] finale rispetti gli standard di qualità definiti e che eventuali problemi vengano gestiti in modo efficace. Esso include:

- **Requisiti[G] del committente[G]**: definizione chiara delle esigenze e delle aspettative.
- **Metriche di analisi**: identificazione dei parametri per valutare la qualità.
- **Controllo della qualità**: implementazione di sistemi per monitorare l'intero ciclo di vita del progetto.
- **Test e verifiche**: pianificazione e documentazione dei controlli per garantire il corretto funzionamento del prodotto[G].
- **Monitoraggio[G] dello stato**: utilizzo di strumenti per visualizzare i progressi e identificare eventuali criticità.
- **Retrospettive e miglioramenti**: discussione dei risultati, individuazione di azioni correttive e proposte di auto-miglioramento.

4.3.3. Testing

Il Piano di Qualifica[G] include gli obiettivi di qualità del processo e del prodotto[G], insieme alle metriche necessarie per valutarne l'accettabilità. Per garantire la qualità del software, vengono eseguiti diversi tipi di test, suddivisi nelle seguenti categorie:

- **Test di unità**: verifica[G] il funzionamento delle singole unità di un programma, controllando funzioni e algoritmi.
- **Test d'integrazione**: analizza il modo in cui le diverse componenti interagiscono tra loro, assicurandosi che funzionino correttamente anche in collaborazione.
- **Test di sistema**: valuta il comportamento complessivo del software, verificando la conformità ai requisiti[G] definiti.
- **Test di accettazione**: ultimo step di verifica[G], attuato dopo aver completato tutti i test precedenti e risolto eventuali problemi, per confermare il corretto funzionamento del prodotto[G].

Alla fine del processo, viene redatto un resoconto contenente i risultati delle verifiche sulla qualità del processo, del prodotto[G] e del software. Grazie alle varie analisi metriche e alla costruzione di grafici, è possibile visualizzare l'evoluzione degli indicatori nel tempo. Questi dati sono consultabili nella sezione Metriche **4.3.4**, dove sono descritte le formule e i criteri di valutazione.

4.3.4. Metriche

Le metriche costituiscono uno strumento oggettivo per valutare il livello di qualità raggiunto nel corso del progetto, sia in termini di prodotto che di processo. Permettono di misurare parametri rilevanti come copertura del codice, successo dei test, affidabilità, usabilità e leggibilità della documentazione. I valori raccolti durante lo sviluppo vengono confrontati con soglie di accettazione predefinite, al fine di valutare l'efficacia delle attività svolte. Di seguito sono descritte le principali metriche utilizzate, con il relativo criterio di calcolo.

- **Code Coverage (PSV01)**: rappresenta la percentuale di codice effettivamente eseguito durante i test. Si calcola dividendo il numero di righe di codice che sono state eseguite almeno una volta durante i test per il numero totale di righe di codice, moltiplicando poi il risultato per cento.
- **Grado di successo dei test (PSV02)**: misura la percentuale di test che sono stati superati. Si ottiene dividendo il numero di test superati per il numero totale di test eseguiti, e moltiplicando il risultato per cento.
- **Failure Density (QPA03)**: indica il numero medio di guasti rilevati nel software, in rapporto alla dimensione del codice. Si calcola dividendo il numero di guasti rilevati per il numero di migliaia di linee di codice (KLOC) del sistema.
- **Time on Task (QPU01)**: rappresenta il tempo medio necessario a un utente per completare un'operazione specifica. Questo dato viene ricavato dai test di usabilità, facendo la media dei tempi impiegati da tutti gli utenti.
- **Error Rate (QPU02)**: indica la percentuale di errori commessi dagli utenti durante l'utilizzo del sistema. Si calcola dividendo il numero di errori totali per il numero complessivo di azioni svolte dagli utenti, moltiplicando il risultato per cento.

- **Indice di Gulpease (PSD01)**: valuta la leggibilità di un testo. Il suo valore è calcolato in base al numero di lettere, parole e frasi contenute nel testo. Più alto è il valore, più il testo è ritenuto leggibile. Un valore superiore a 50 è considerato accettabile.

Tutte queste metriche vengono monitorate nel tempo e visualizzate nel Cruscotto di Valutazione (é presente nella sezione 5 del file Piano di Qualifica[G]), dove sono messi a confronto i valori reali con quelli attesi. L'analisi dei dati consente al team di individuare le aree di miglioramento e pianificare eventuali azioni correttive.

4.4. Verifica

4.4.1. Scopo

Il processo di Verifica[G] del software ha l'obiettivo di garantire che lo sviluppo sia eseguito correttamente e che il prodotto[G] finale rispetti i requisiti[G] del cliente[G]. Durante questa fase, vengono individuati eventuali difetti nelle prime fasi del ciclo di vita, riducendo così costi e tempi di correzione.

La verifica[G] assicura la conformità del prodotto[G] ai requisiti[G] stabiliti nel Piano di Qualifica[G], garantendo completezza e correttezza attraverso evidenze misurabili.

Ogni produzione viene sottoposta a controlli prima di essere pubblicata nel repository[G] GitHub, con verificatori[G] indipendenti per evitare conflitti di interesse.

Questo processo viene applicato continuamente, sia nella documentazione che nello sviluppo del software, con verifiche specifiche per garantire l'accuratezza delle attività svolte.

4.4.2. Tipi di analisi

La verifica[G] verrà effettuata in maniera efficace adottando i metodi qui di seguito elencati.

4.4.2.1. Analisi Statica

L'Analisi Statica è un processo di verifica[G] che controlla il codice[G] e la documentazione prima della loro esecuzione, garantendo conformità ai requisiti[G] e prevenendo errori. Non richiedendo l'esecuzione del software, permette di identificare problemi iniziali, migliorando la qualità complessiva.

Si concentra su aspetti sintattici e strutturali, utilizzando tecniche manuali o automatizzate per verificare convenzioni, metriche e coerenza del sistema. Può essere applicata anche ai documenti testuali, assicurando accuratezza e correttezza.

Le verifiche possono avvenire tramite metodi formali, basati su prove matematiche, o tecniche di lettura, che aiutano a rilevare incongruenze prima che si manifestino in fase di esecuzione.

Le tecniche di lettura sono due: Walkthrough ed Ispezione.

4.4.2.1.1. Walkthrough

Il Walkthrough è una tecnica di verifica[G] che consiste nell'analisi approfondita di un prodotto[G] per individuare errori o incongruenze senza una ricerca mirata su un tipo specifico di difetto. Questo metodo è particolarmente utile quando non si ha certezza sulla posizione dei problemi, permettendo un esame critico dell'intero oggetto in esame.

Si tratta di una procedura collaborativa che si articola in diverse fasi:

- **Pianificazione**: definizione delle modalità, risorse e tempistiche dell'attività.
- **Lettura**: analisi dettagliata del prodotto[G] per individuare errori e possibili miglioramenti.

- **Discussione**: confronto tra verificatore[G] e l'autore del prodotto[G] per valutare le problematiche riscontrate e proporre soluzioni.
- **Correzione dei difetti**: attuazione delle modifiche concordate per garantire la qualità del prodotto[G].

Il walkthrough permette di identificare problemi, migliorare la qualità e favorire una comprensione più approfondita del prodotto[G].

4.4.2.1.2. Ispezione

L'Ispezione è una tecnica di verifica[G] mirata che permette di identificare e rimuovere errori e difetti, basandosi su un'idea preliminare delle possibili problematiche. A differenza del walkthrough, questo metodo utilizza liste di controllo per eseguire verifiche specifiche, evitando di analizzare l'intero prodotto[G].

L'Ispezione si sviluppa attraverso le seguenti fasi:

- **Pianificazione**: definizione di modalità, risorse e tempistiche dell'attività.
- **Creazione della lista di controllo**: individuazione degli aspetti critici da verificare, con aggiornamenti progressivi in base alla frequenza di determinati errori.
- **Lettura mirata**: il verificatore[G] esamina il prodotto[G] seguendo la lista di controllo, identificando eventuali incongruenze e suggerendo miglioramenti.
- **Correzione dei difetti**: l'autore implementa le modifiche necessarie per garantire la qualità del prodotto[G].

Grazie alla sua maggiore efficienza rispetto al walkthrough, l'Ispezione consente una verifica[G] più rapida e ottimizzata, risultando talvolta automatizzabile. Sebbene non sia sempre esaustiva, soprattutto nelle fasi iniziali di un progetto, offre un metodo strutturato per individuare errori e perfezionare la qualità complessiva del prodotto[G].

4.4.2.2. Analisi dinamica

L'Analisi dinamica è una metodologia di verifica[G] che prevede l'esecuzione del software per valutarne il comportamento, le performance e la correttezza. A differenza dell'Analisi statica, che si concentra su aspetti strutturali e sintattici, l'analisi dinamica permette di identificare errori direttamente durante l'esecuzione, rilevando failure e correggendo i fault che causano comportamenti inattesi.

Questa attività si basa su test ripetibili e automatizzabili, garantendo che le correzioni applicate vengano verificate con nuove esecuzioni.

Le principali categorie di test utilizzate nell'analisi dinamica includono:
- **Test di unità**: verifica[G] il funzionamento di singole componenti.
- **Test d'integrazione**: analizza le interazioni tra i moduli.
- **Test di sistema**: valuta la conformità del software ai requisiti[G].
- **Test di accettazione**: ultimo step di verifica[G], attuato dopo aver completato tutti i test precedenti e risolto eventuali problemi, per confermare il corretto funzionamento del prodotto[G].

Per garantire un'analisi efficace, è fondamentale stabilire e regolamentare le condizioni dell'ambiente di esecuzione dei test.

4.4.2.2.1. Test di unità

I test di unità sono fondamentali per verificare il corretto funzionamento delle singole unità software, garantendo che ogni componente operi come previsto. Vengono formulati dai verificatori[G] durante la fase di progettazione di dettaglio, in conformità alle specifiche di ciascuna unità.

Durante l'esecuzione dei test, è possibile l'impiego di stub e driver, strumenti che simulano componenti non ancora disponibili, permettendo di testare le unità in isolamento. Questo approccio risulta essenziale nelle fasi iniziali dello sviluppo, quando il sistema è ancora incompleto.

I test di unità si suddividono in due categorie principali:

- **Test funzionali** (black box): verificano il corretto comportamento dell'unità in base agli input e output attesi, senza analizzare la logica interna.
- **Test strutturali** (white box): esaminano i percorsi logici interni al codice[G], verificando il corretto funzionamento delle sue istruzioni.

Per garantire un processo efficiente e sistematico, i verificatori[G] sfruttano strumenti di automazione, ottimizzando la verifica[G] e assicurando che ogni unità software rispetti le specifiche stabilite.

4.4.2.2.2. Test di integrazione

I test di integrazione verificano il corretto funzionamento delle interazioni tra le diverse unità di un sistema software, assicurandosi che i componenti già testati singolarmente collaborino in modo adeguato.

Questi test vengono definiti dai verificatori[G] nella fase di progettazione architetturale, con l'obiettivo di rilevare difetti di progettazione, errori derivanti da unit testing, incompatibilità nelle interfacce e problemi di integrazione con altre applicazioni.

Esistono due strategie principali per l'integrazione:

- **Top-down**: si parte dalle componenti con maggior numero di dipendenze d'uso e maggiore responsabilità verso l'esterno. Questo metodo permette di integrare per prime le funzionalità visibili all'utente, ma richiede l'uso di numerosi stub.
- **Bottom-up**: si parte dalle componenti con minori dipendenze e maggiore utilità interna. Questo approccio riduce l'uso di stub ma può ritardare la disponibilità delle funzionalità accessibili agli utenti.

L'integrazione viene realizzata in modo incrementale[G], ampliando progressivamente il raggio d'azione dei test per garantire la stabilità del sistema. Questo processo è essenziale per evitare incompatibilità e incongruenze tra i moduli, assicurando un funzionamento fluido e coerente.

4.4.2.2.3. Test di sistema

I test di sistema sono una fase cruciale della verifica[G] del software, mirata a valutare il comportamento complessivo dell'applicazione per garantire che soddisfi tutti i requisiti[G] definiti nella fase di analisi dei requisiti[G]. Questi test vengono eseguiti dopo il completamento dei test di unità e dei test di integrazione, ma prima del collaudo finale con il committente[G].

Considerati test funzionali (black-box), i test di sistema non richiedono la conoscenza della logica interna del software, ma si concentrano sulla validazione[G] dell'intero sistema in condizioni

realistiche, simulando il più possibile l'ambiente di produzione. Il loro obiettivo è verificare che tutte le componenti siano correttamente integrate e che il software funzioni come previsto.

Attraverso questi test, si controlla la correttezza e l'efficacia del sistema rispetto ai requisiti[G] stabiliti, assicurando che l'applicazione sia conforme alle specifiche e pronta per l'utilizzo finale.

4.4.2.2.4. Test di accettazione

I test di accettazione rappresentano l'ultima fase del processo di verifica[G] del software prima del rilascio ufficiale e dell'utilizzo in ambiente operativo. Il loro obiettivo principale è confermare che il prodotto sviluppato sia conforme ai criteri di accettazione concordati con il committente[G] o con gli stakeholder[G], garantendo la soddisfazione delle aspettative funzionali e non funzionali.

Questi test vengono eseguiti dopo i test di sistema, e costituiscono il collaudo formale del software, spesso in contesti simulati o reali che riflettono l'ambiente d'uso previsto. Possono essere condotti dal team di sviluppo insieme al cliente, oppure da figure indipendenti incaricate della qualità, come i collaudatori esterni o i rappresentanti dell'utente finale.

I test di accettazione rientrano nella categoria dei test funzionali (black-box) e si concentrano sull'uso del software dal punto di vista dell'utente. Non analizzano la struttura interna dell'applicazione, ma si focalizzano sul comportamento osservabile e sulla rispondenza ai requisiti[G] contrattuali o specificati nei documenti di accettazione.

Questa fase consente di:
- Validare che tutte le funzionalità principali siano disponibili e funzionanti come previsto.
- Verificare la rispondenza a requisiti legali, normativi o aziendali.
- Rilevare eventuali disallineamenti tra quanto implementato e quanto atteso dal committente.
- Approvare formalmente il passaggio del software alla fase di rilascio o di esercizio.

Il superamento con esito positivo dei test di accettazione rappresenta il presupposto fondamentale per la consegna del prodotto e la chiusura del progetto dal punto di vista tecnico.

4.5. Validazione

4.5.1. Scopo

La Validazione[G] del software è il processo di valutazione finalizzato a garantire che il prodotto[G] soddisfi i requisiti[G] predefiniti e le aspettative degli utenti e del committente[G]. Un esito positivo della Validazione[G] conferma che il software è allineato alle specifiche concordate e pronto per l'utilizzo.

Questo processo consente di:

- **Identificare errori** eventualmente trascurati nella fase di verifica[G].
- **Verificare il rispetto dei requisiti[G]** stabiliti nel documento di Analisi dei Requisiti[G].
- **Migliorare la qualità** e **il valore** del prodotto[G] finale.

La Validazione[G] risponde alla domanda *"Ho realizzato il giusto sistema?"*, assicurando che il software sviluppato soddisfi le esigenze del committente[G]. Si svolge alla fine dello sviluppo, rappresentando una fase fondamentale per confermare la correttezza del prodotto[G] e la sua conformità agli obiettivi stabiliti.

5. Processi Organizzativi

5.1. Gestione dei processi

5.1.1. Scopo

Lo scopo dei processi organizzativi è definire un sistema strutturato per la gestione, il monitoraggio e il miglioramento continuo dei processi aziendali, al fine di garantire coerenza operativa, efficienza organizzativa e il raggiungimento degli obiettivi strategici dell'organizzazione. Consiste nell'organizzare la progettazione, l'attuazione, il controllo e la revisione dei processi in modo sistematico e documentato.

5.1.2. Pianificazione delle risorse umane

5.1.2.1. Ruoli e Responsabilità

I ruoli all'interno del team sono suddivisi tra i seguenti:

- Responsabile[G]
- Amministratore[G]
- Analista[G]
- Progettista[G]
- Programmatore[G]
- Verificatore[G]

Le responsabilità dei ruoli qui elencati sono consultabili nel Glossario[G].

5.1.2.2. Criteri rotazione ed assegnazione ruoli

La prima assegnazione dei ruoli riflette, ove possibile, le preferenze personali dei membri del gruppo. In seguito, la rotazione dei ruoli avviene in maniera concordata con il resto del gruppo, tipicamente all'inizio di un nuovo Sprint[G], ma anche in base alle necessità del *backlog*[G].

Infatti, una gestione più flessibile dei ruoli permette il coinvolgimento di tutto il team verso l'obiettivo di terminare il prima possibile il lavoro da svolgere, non concentrandosi esclusivamente sui task specifici assegnati al proprio ruolo. In questo modo si evitano, inoltre, situazioni in cui il passaggio da un ruolo all'altro è drastico, il che porterebbe inevitabilmente ad un aumento del debito tecnico.

Viene perciò preferita una comunicazione frequente tra i membri, in modo che tutti abbiano una visione costantemente aggiornata del punto in cui ci si trova con il lavoro.

La rotazione dei ruoli avviene tipicamente all'inizio di ogni Sprint[G], ovvero ogni due settimane. Questo, però, non significa che tra uno Sprint[G] e l'altro ogni singolo membro del team debba assumere un ruolo diverso nello Sprint[G] che sta per iniziare rispetto a quello assunto durante l'ultimo Sprint[G].

Per favorire il cambio dei ruoli ad ogni Sprint[G] e facilitare la transizione, infatti, si è deciso di mantenere almeno un componente di un gruppo di ruoli nello stesso ruolo come "Mentore[G]". In questo modo, i mentori[G] potranno aggiornare e supportare i nuovi arrivati nel suddetto ruolo, permettendo allo sviluppo di continuare in modo più fluido e coerente con quanto realizzato nello Sprint[G] precedente.

I criteri di rotazione dei ruoli devono inoltre consentire una suddivisione bilanciata dei ruoli tra i membri sull'intero arco del progetto, permettendo quindi una certa flessibilità sul breve periodo.

5.1.2.3. Allocazione risorse umane

Nella fase di assegnazione dei ruoli, è necessario fare riferimento alla fase di sviluppo in cui ci si trova. Infatti, in base ai progressi svolti fino a tal momento, vengono decise quali sono le figure richieste per il successivo Sprint[G].

Ciò appare evidente, ad esempio, nelle fasi iniziali di sviluppo, nelle quali sono necessari maggiormente gli Analisti[G], o nelle fasi più avanzate dove sono richieste in maggior numero figure che ricoprano il ruolo di Programmatore[G] e di Verificatore[G].

5.1.2.4. Escalation path

Durante l'intera fase di sviluppo, potrebbero capitare di venir assegnati alcuni ruoli a più membri del team per lo Sprint[G] corrente (ad esempio, il ruolo di Analista[G] viene assegnato a 4 persone durante l'attuale Sprint[G] poiché è una figura molto richiesta in quella determinata fase di sviluppo).

Al fine di coordinare meglio il flusso di lavoro, si è deciso, ad ogni Sprint[G], di scegliere tra questi membri un "sottoresponsabile[G]«. Tale sottoresponsabile[G] comunicherà periodicamente con i sottoresponsabili delle altre categorie di ruolo e con il responsabile[G] di turno per tenerli aggiornati sullo stato di avanzamento delle attività.

Inoltre, i sottoresponsabili delle categorie di ruolo sono tenuti a mantenere una comunicazione costante con i membri del loro sottogruppo, assicurandosi che ogni task sia eseguita secondo le linee guida stabilite e che eventuali dubbi vengano chiariti prontamente. Questo approccio favorisce una gestione efficiente delle attività e una rapida risoluzione delle problematiche interne. La figura del sottoresponsabile[G], infatti, viene in aiuto al responsabile[G] facendo in modo che non debba rimanere in comunicazione con tutto il gruppo, ma solamente con i sottoresponsabili del loro determinato ruolo.

5.1.2.5. Orari di disponibilità

All'interno della cartella Drive del team è stato creato il documento **«Orari disponibilità»** su Google Sheets nel quale, settimanalmente, i membri del gruppo aggiornano la loro corrispettiva riga con gli orari di disponibilità nel corso della settimana entrante. Tale documento permette di organizzare riunioni e sessioni di lavoro in base alle disponibilità di tutti, evitando sovrapposizioni, e permette l'accesso immediato a quelle informazioni.

5.1.2.6. Bilanciamento del carico di lavoro

Il carico di lavoro assegnato ad ogni membro del team viene deciso in base alla complessità e al livello di priorità di completamento delle determinate task da assegnare. Nell'assegnare le task ad ogni Sprint[G] si cerca il più possibile di non sfociare nel sovraccarico. In caso di qualsiasi imprevisti o difficoltà, ogni membro potrà comunicare tale disagio al Responsabile[G] in carica.

5.1.3. Sprint

5.1.3.1. Durata degli Sprint

Ogni Sprint[G] ha tendenzialmente durata di due settimane. Questa durata permette al team di concentrarsi su obiettivi concreti per un lasso di tempo sufficientemente ampio e di valutare i progressi in modo regolare, così da permettere un equilibrio tra flessibilità e stabilità.

5.1.3.2. Pianificazione dello Sprint

Pianificare gli Sprint[G] implica definire gli obiettivi e le attività da completare durante lo Sprint[G]. Durante questa fase, vengono identificati i requisiti[G], assegnate le responsabilità e stabilite le priorità. Una buona pianificazione è essenziale per il successo dello Sprint[G].

5.1.3.3. Review dello Sprint

La review dello Sprint[G] è una riunione che si tiene alla fine di ogni Sprint[G] secondo quanto descritto al punto **5.2.3**. Durante tale riunione, il team discute a grandi linee su ciò a cui si è lavorato e stabilisce il materiale da presentare agli stakeholder, i quali daranno dei feedback su quanto svolto così da identificare eventuali miglioramenti per il prossimo Sprint[G].

5.1.3.4. Retrospettiva dello Sprint

La retrospettiva dello Sprint[G] è una fase in cui il team analizza nel dettaglio il proprio lavoro realizzato durante lo Sprint[G]. L'obiettivo di questa fase è di identificare ciò che ha funzionato bene e quel che può essere migliorato, promuovendo un continuo miglioramento della qualità del prodotto[G] in sviluppo.

5.2. Procedure di comunicazione

5.2.1. Comunicazioni interne

Durante l'intero ciclo di sviluppo del progetto, la comunicazione tra i membri del team è facilitata attraverso i canali ufficiali Discord e Telegram, scelti per la loro efficienza e versatilità.

- *Discord* è stato selezionato per la sua praticità d'uso e per la possibilità di creare canali dedicati per ogni ruolo specifico, quali, ad esempio, analista[G], programmatore[G] e progettista[G]. Questa suddivisione assicura che ogni ruolo possa discutere direttamente delle proprie attività e delle difficoltà incontrate, facilitando il coordinamento e l'implementazione delle soluzioni.
- *Telegram* è stato scelto per la sua semplicità e immediatezza nella comunicazione. Esso consente infatti di inviare messaggi rapidi e di condividere documenti e aggiornamenti in tempo reale. In questo modo, è possibile garantire che tutte le informazioni rilevanti siano facilmente accessibili a tutti i membri del team in qualsiasi momento.

L'uso combinato di Discord e Telegram risulta dunque più che valido nel coprire tutte le esigenze comunicative nel corso delle varie fasi di sviluppo. Ricorrere ad entrambe le applicazioni, infatti, assicura che ogni membro del team possa interagire efficacemente e contribuire al raggiungimento degli obiettivi comuni.

5.2.2. Comunicazioni esterne

Durante l'intero ciclo di sviluppo del progetto, la comunicazione esterna (tra i membri del gruppo e il committente[G]/il proponente) è a carico del Responsabile[G] di turno. Essa avviene attraverso diversi canali di comunicazione, tra cui:

- **Zoom**: per comunicare in modalità sincrona con il committente[G]
- **Gmail**: per comunicare in modalità asincrona con il committente[G] e il proponente. Per comunicare con gli enti terzi, infatti, viene utilizzata l'indirizzo mail ufficiale del gruppo;
- **Microsoft Teams**: per comunicare in modalità sincrona con il proponente.

5.2.3. Riunioni interne

Le riunioni interne possono essere indette durante vari momenti strategici dello sviluppo del progetto, soprattutto tra uno Sprint[G] e l'altro.

Difatti, al termine di ogni Sprint[G], tutti i membri del team si riuniscono per discutere di quanto svolto nel corso dello Sprint[G] appena concluso. Durante tali incontri, vengono rese note eventuali difficoltà emerse durante l'esecuzione di determinate task e si affrontano possibili soluzioni alle problematiche riscontrate. Inoltre, vengono elencate le prossime task da svolgere per garantire la continuità del lavoro.

È in questa occasione che viene annunciata la rotazione dei ruoli per il prossimo Sprint[G] e la conseguente assegnazione delle varie task a ogni membro a seconda dello specifico ruolo che ricopre.

Durante l'intera fase di sviluppo, lì dove necessario, possono essere organizzate brevi riunioni interne tra più sottogruppi così da chiarire eventuali dubbi riguardo alle attività svolte nei precedenti Sprint[G].

5.2.4. Riunioni esterne

Le riunioni esterne con il proponente vengono svolte tendenzialmente due volte al mese, previo accordo con l'azienda tramite il canale di comunicazione Gmail. Esse vengono saranno richieste dal gruppo in caso di necessità di opinioni esperte e/o chiarimenti su questioni tecniche e per avere una verifica[G] il corretto progresso del progetto didattico.

Il gruppo si impegna a presenziare in presenza in maniera più assidua possibile agli incontri con il proponente. Nel caso in cui, per qualunque motivazione, non sia possibile partecipare in presenza alla riunione, si fa uso del canale di comunicazione online Microsoft Teams. Al termine di ogni riunione viene redatto un verbale esterno al fine di documentare l'incontro, il quale dovrà ricevere l'approvazione dell'azienda stessa.

5.2.5. Reportistica

A seguito di ogni riunione, la stesura del corrispondente verbale viene assegnato, se disponibile, a uno dei verificatori[G]. In questo modo, un altro verificatore[G] avrà il compito di effettuare una cross-verifica[G] del verbale redatto, assicurando così che tutte le informazioni siano corrette e che eventuali errori siano prontamente individuati e corretti.

Nel caso in cui i verificatori[G] siano già impegnati in altre attività di priorità più alta, la stesura del verbale viene assegnata a un altro membro del team. Questo approccio flessibile garantisce che la documentazione delle riunioni non subisca ritardi e che ogni riunione sia adeguatamente registrata. Il membro del team incaricato di redigere il verbale è responsabile[G] di seguire le linee guida stabilite per la stesura, assicurando che il documento finale sia più chiaro, conciso e completo possibile.

La verifica[G] dello stesso sarà assegnata a uno dei verificatori[G], che avrà cura di controllare, ed eventualmente correggere, quanto riportato nel verbale non appena gli sarà possibile.

Questo processo strutturato e collaborativo assicura che i verbali interni siano sempre accurati e tempestivi, facilitando la comunicazione e la trasparenza all'interno del team.

5.3. Gestione dell'infrastruttura

5.3.1. Scopo

Il processo di Infrastruttura è responsabile[G] della creazione, gestione e mantenimento dei componenti hardware e software necessari per supportare tutti gli altri processi. Include strumenti organizzativi che facilitano comunicazione, coordinamento e pianificazione, permettendo al team di operare in modo efficace ed efficiente.

Questo processo documenta l'infrastruttura utilizzata, definendo le modalità di implementazione e manutenzione. Inoltre, garantisce la gestione ottimale delle risorse, assicurando che l'infrastruttura sia adeguata a tutte le fasi del ciclo di vita del software.

5.3.2. Strumenti

5.3.2.1. GitHub

GitHub è la piattaforma che utilizziamo per il versionamento e l'hosting dei nostri repository[G] di [G]. Ci fornisce funzionalità avanzate per la gestione collaborativa del codice[G], tra cui pull request, issue tracking, continuous integration e automazione dei flussi di lavoro.

5.3.2.2. GitHub Pages

GitHub Pages ci permette di pubblicare facilmente una versione web del nostro repository[G]. Avere una repository web, infatti, è molto utile per la documentazione, la presentazione di progetti o la condivisione di informazioni con il team e i collaboratori esterni.

5.3.2.3. GitHub Flow

Adottiamo GitHub Flow come metodo di branching per garantire un flusso di sviluppo agile[G] e ben strutturato. Utilizziamo il modello basato su feature branches[G] e pull request per integrare le modifiche nel ramo principale in modo controllato e sicuro.

5.3.2.4. Jira

Jira è il nostro sistema di issue tracking e gestione dei task. Ci consente di organizzare e coordinare il lavoro, monitorare le milestone e garantire la visibilità sull'avanzamento delle attività del team.

5.3.2.4.1. Creazione task

Quando creiamo un task su Jira, utilizziamo specifici parametri, tra cui la priorità, la descrizione dettagliata, la categoria e il collegamento ad altri task o milestone correlati.

5.3.2.4.2. Assegnazione task

L'assegnazione di un task avviene in base al suo ambito di riferimento e alla sua complessità. Inoltre, si applica la convenzione intera per cui, ad esempio, il Sottoresponsabile[G] Analista[G], che rappresenta l'intero gruppo di analisti[G], è incaricato di fare da portavoce per tutti gli analisti[G] se stanno lavorando contemporaneamente a quella determinata task.

5.3.2.5. Discord

Discord è il nostro strumento principale per le comunicazioni vocali e le riunioni interne. La suddivisione dei canali per ruoli ci permette di mantenere l'organizzazione e facilitare la discussione tra gruppi con competenze specifiche.

5.3.2.6. Telegram

Telegram viene utilizzato per comunicazioni testuali rapide, sia tra singoli membri del team che all'interno di gruppi più ampi. È utile per scambi veloci e aggiornamenti istantanei.

5.3.2.7. Typst

Typst è la piattaforma che usiamo per la stesura della documentazione ufficiale, offrendo strumenti avanzati per la formattazione e la gestione dei contenuti tecnici.

5.3.2.8. Google Drive
Google Drive funge da archivio per documenti informali e materiali destinati alla formazione individuale, permettendo un facile accesso e condivisione all'interno del team.

5.3.2.9. Google Calendar
Google Calendar ci supporta nella gestione degli appuntamenti e delle riunioni, facilitando il coordinamento e la pianificazione delle attività comuni.

5.3.2.10. Google Sheets
Utilizziamo Google Sheets per la definizione dei ruoli, la pianificazione degli orari di disponibilità e la gestione organizzativa delle attività operative del team.

5.3.2.11. Gmail
Il nostro indirizzo email Gmail è il canale ufficiale attraverso cui comunichiamo con il proponente e il committente[G], garantendo un flusso di comunicazione formale e tracciabile.

5.3.2.12. Draw.io
Draw.io è lo strumento che usiamo per creare grafici e schemi visivi, essenziali per la rappresentazione di concetti e processi operativi in modo chiaro e strutturato.

5.4. Miglioramento
Il processo di miglioramento ha il compito di definire, misurare e controllare l'evoluzione della qualità durante tutto il ciclo di vita del software.

5.4.1. Creazione del processo
Il team implementa una serie di processi basati sullo Standard ISO/IEC 12207:1997, documentandone l'applicazione e, ove possibile, istituendo strumenti di controllo per monitorare e perfezionare il processo nel tempo.

5.4.2. Valutazione del processo
Attraverso revisioni interne, revisioni congiunte e procedure di valutazione, il team analizza l'efficacia dei processi adottati. I risultati vengono archiviati nella documentazione e nei report generati da Jira, determinando le necessità di miglioramento. Per garantire continuità ed efficienza, il team programma revisioni periodiche per valutare l'adeguatezza dei processi.

5.4.3. Attuazione miglioramento del processo
Sulla base delle valutazioni effettuate, il team introduce le modifiche necessarie ai processi, aggiornando la documentazione per riflettere i miglioramenti apportati e mettendo in evidenza punti di forza e criticità.

5.5. Formazione

5.5.1. Scopo
Il processo di formazione definisce le modalità con cui i membri del gruppo acquisiscono le competenze necessarie per la produzione della documentazione e lo sviluppo del prodotto[G] software.

5.5.2. Metodo di Formazione
Ogni componente ha la libertà di scegliere il metodo di apprendimento più adatto, privilegiando un approccio learning by doing. Questo favorisce una visione più ampia sulle tecnologie utilizzate, aumentando la consapevolezza del team e migliorando le scelte implementative.

Per ridurre eventuali discrepanze di conoscenza, se necessario, vengono organizzati dei brevi incontri in cui i membri più esperti condividono le loro competenze con il resto del gruppo.

5.5.3. Materiale formativo

Il materiale formativo viene archiviato all'interno della cartella condivisa su Google Drive, aggiornata periodicamente con il contributo di tutti i membri del team.

6. Standard di qualità

6.1. ISO/IEC 12207:1995

Lo standard ISO/IEC 12207:1995 stabilisce un modello per il ciclo di vita del software, suddividendo i processi in primari, di supporto e organizzativi.

Ogni processo è articolato in attività e task specifici per garantire una gestione strutturata del progetto.

6.1.1. Processi primari

I processi primari riguardano le parti principali coinvolte nello sviluppo, operatività e manutenzione del software: acquirente, fornitore[G], sviluppatore, operatore e manutentore.

- **Acquisizione**: comprende le attività necessarie per la richiesta, il controllo e l'accettazione di un prodotto[G] o servizio[G] software.
- **Fornitura**: riguarda la gestione della fornitura del prodotto[G], dalla pianificazione alla consegna.
- **Sviluppo**: include tutte le fasi di progettazione e implementazione del software.
- **Operatività**: gestisce il funzionamento del sistema per gli utenti.
- **Manutenzione**: garantisce aggiornamenti e correzioni per mantenere il software operativo.

6.1.2. Processi di supporto

I processi di supporto affiancano i processi primari, contribuendo alla qualità del prodotto[G].

- **Documentazione**: registra le informazioni prodotte in ogni fase del progetto.
- **Gestione della Configurazione**: controlla le modifiche e la consistenza degli elementi del sistema.
- **Accertamento della qualità**: verifica[G] la conformità dei processi e dei prodotti[G].
- **Verifica[G] e validazione[G]**: assicurano che il software soddisfi i requisiti[G] e sia utilizzabile.
- **Revisioni**: valutano lo stato del progetto e la conformità agli standard.
- **Risoluzione dei problemi**: analizza e gestisce le problematiche emerse nel ciclo di vita.

6.1.3. Processi organizzativi

I processi organizzativi definiscono la struttura e migliorano i processi aziendali nel tempo.

- **Gestione organizzativa**: pianifica e controlla i processi generali.
- **Infrastruttura**: costruisce e mantiene le risorse necessarie.
- **Miglioramento**: valuta e ottimizza i processi esistenti.
- **Formazione**: garantisce l'aggiornamento delle competenze del team.

6.2. Standard ISO/IEC 9126

Lo standard ISO/IEC 9126 stabilisce un modello di qualità del software, consentendo di gestire, monitorare e migliorare ogni fase del ciclo di vita del software, garantendo un approccio efficace e sistematico. Tale modello è suddiviso in sei caratteristiche fondamentali:

- **Funzionalità**;
- **Affidabilità**;

- **Efficienza**
- **Usabilità**;
- **Manutenibilità**;
- **Portabilità**.

6.2.1. Funzionalità

La Funzionalità misura l'adeguatezza del software rispetto ai requisiti[G] da soddisfare. Un sistema funzionale garantisce che ogni componente sia adeguato alle esigenze del proponente e degli utenti.

Gli aspetti fondamentali della Funzionalità includono:

- **Adeguatezza**: verifica[G] che il software fornisca le funzioni necessarie per il suo scopo.
- **Accuratezza**: garantisce che i risultati siano corretti e conformi alle aspettative.
- **Interoperabilità**: assicura che il software possa interagire con altri sistemi specificati.
- **Conformità**: garantisce il rispetto di standard e convenzioni del settore.
- **Sicurezza**: protegge i dati degli utenti, prevenendo accessi non autorizzati.

6.2.2. Affidabilità

L'Affidabilità aiuta la capacità del sistema a funzionare correttamente mantenendo prestazioni stabili nel tempo, anche sotto carichi intensi. Un sistema affidabile deve garantire un funzionamento corretto, evitando errori e guasti che potrebbero comprometterne l'operatività.

I principali aspetti valutati nell'Affidabilità sono:

- **Maturità**: capacità di ridurre il rischio di malfunzionamenti ed errori.
- **Tolleranza agli errori**: mantenimento delle prestazioni anche in presenza di problemi.
- **Recuperabilità**: possibilità di ripristinare il corretto funzionamento dopo un guasto.
- **Aderenza**: conformità agli standard di affidabilità stabiliti per il sistema.

6.2.3. Efficienza

L'Efficienza di un software misura la capacità di fornire prestazioni ottimali in relazione alle risorse impiegate. Un sistema efficiente garantisce tempi di risposta rapidi e un utilizzo proporzionale delle risorse, senza sprechi o sovraccarichi inutili.

I principali aspetti valutati per l'Efficienza includono:

- **Comportamento temporale**: verifica[G] che il software risponda entro tempi adeguati.
- **Utilizzo delle risorse**: ottimizza il consumo di memoria, CPU e altre risorse in rapporto all'uso.
- **Conformità**: garantisce l'aderenza agli standard di efficienza stabiliti nel settore.

6.2.4. Usabilità

L'Usabilità misura la capacità di un software di essere comprensibile, facile da apprendere e intuitivo nell'utilizzo. Un sistema ben progettato deve permettere agli utenti di interagire senza difficoltà, migliorando l'esperienza complessiva.

Le principali caratteristiche dell'Usabilità includono:

- **Comprensibilità**: facilita la comprensione delle funzionalità e dell'interfaccia.
- **Apprendibilità**: consente un apprendimento rapido e intuitivo del sistema.
- **Operabilità**: garantisce un utilizzo semplice e fluido, riducendo complessità.
- **Attrattività**: rende l'interazione piacevole e soddisfacente per gli utenti.
- **Conformità**: assicura l'adesione agli standard e alle convenzioni di usabilità.

6.2.5. Manutenibilità

La Manutenibilità di un software misura la facilità con cui può essere modificato, aggiornato e testato, garantendo la sua evoluzione e adattabilità a nuovi requisiti[G] o ambienti. Un software ben progettato deve consentire interventi rapidi ed efficaci per la correzione di errori e l'implementazione di miglioramenti.

I principali aspetti valutati nella Manutenibilità sono:

- **Analizzabilità**: semplifica la diagnosi di problemi, facilitando l'ispezione del codice[G].
- **Modificabilità**: permette di apportare modifiche con minima complessità.
- **Stabilità**: assicura che il sistema mantenga un comportamento coerente dopo aggiornamenti.
- **Testabilità**: garantisce che il software sia facilmente verificabile, riducendo i tempi di debug.

6.2.6. Portabilità

La Portabilità analizza la sua capacità di essere trasferito e adattato a diversi ambienti di lavoro, garantendo compatibilità con sistemi e tecnologie differenti. Un software altamente portabile offre maggiore flessibilità e facilità d'uso in contesti variabili.

I principali aspetti valutati nella Portabilità includono:

- **Adattabilità**: consente al software di funzionare correttamente in diversi ambienti operativi.
- **Installabilità**: facilita la configurazione e l'implementazione su nuove piattaforme.
- **Coesistenza**: assicura che il software possa operare correttamente insieme ad altri sistemi.
- **Sostituibilità**: permette di rimpiazzare un prodotto[G] esistente con il nuovo software in modo semplice.
- **Conformità**: garantisce l'aderenza agli standard e alle convenzioni sulla portabilità.