AlphaSix

AlphaSix

Norme di Progetto

Informazioni sul documento	
Nome Documento	NormeDiProgetto v4.0.0.pdf
Versione	4.0.0
Data di Creazione	26 novembre 2018
Data ultima modifica	14 maggio 2019
Stato	Approvato
Redazione	Laura Cameran
Verifica	Samuele Gardin Matteo Marchiori
Approvazione	Timoty Granziero
Uso	Interno
Distribuzione	AlphaSix
Destinato a	Prof. Tullio Vardanega, Prof. Riccardo Cardin, AlphaSix
Email di riferimento	alpha.six.unipd@gmail.com

Descrizione

Questo documento riporta le regole, gli strumenti e le convenzioni adottate da AlphaSix durante la realizzazione del progetto Butterfly.

Registro delle modifiche

Versione	Descrizione	Ruolo	Nominativo	Data
4.0.0	Approvazione per il rilascio	Responsabile	Timoty Granziero	2019-05-14
3.3.0	Verifica finale	Verificatore	Ciprian Voinea	2019-05-12
3.2.2	Ampliamento e revisione dei processi di fornitura in §2.1	Amministratore	Nicola Carlesso	2019-05-08
3.2.1	Divisione processo di verifica e validazione in §4.4 e §4.5	Amministratore	Samuele Gardin	2019-05-07
3.2.0	Verifica	Verificatore	Timoty Granziero	2019-05-05
3.1.3	Aggiunta metrica per il processo di verifica in §4.4.9	Amministratore	Nicola Carlesso	2019-05-04
3.1.2	Completata sezione sulla progettazione inserendo §2.2.3.4	Amministratore	Nicola Carlesso	2019-05-01
3.1.1	Spostata sezione §2.2.4.14 in quanto pertinente alla codifica	Amministratore	Laura Cameran	2019-04-30
3.1.0	Verifica	Verificatore	Matteo Marchiori	2019-04-29
3.0.2	Modificato paragrafo §1.5 per aggiungere la numerazione ai riferimenti e per eliminare i riferimenti normativi sbagliati	Amministratore	Laura Cameran	2019-04-25
3.0.1	Aggiunta descrizione design pattern e architettura in §2.2.3.2.4, §2.2.3.2.5, §2.2.3.3.4 e §2.2.3.3.5 perché precedentemente mancanti	Amministratore	Nicola Carlesso	2019-04-24
3.0.0	Approvazione per il rilascio	Responsabile	Nicola Carlesso	2019-03-10
2.3.0	Verifica finale	Verificatore	Samuele Gardin	2019-03-06
2.2.6	Spostamento di §4.3.2 poichè inerente al processo di quality assurance. Aggiunti §4.3.3 e §4.4.4 perchè normano la verifica della progettazione e le attività svolte nel Piano di Qualifica. Aggiunto §4.4.7.5 poichè mancante normazione del tracciamento dei test	Amministratore	Laura Cameran	2019-04-04
2.2.5	Aggiunti §4.2.7, §4.3, e §4.3.1 per normare l'assicurazione di qualità mancante	Amministratore	Matteo Marchiori	2019-04-02
2.2.4	Aggiunti §4.2.3, §4.2.5, §4.2.4 e §4.2.6 per ampliamento del processo di configurazione	Amministratore	Laura Cameran	2019-04-02

Versione	Descrizione	Ruolo	Nominativo	Data
2.2.3	Arricchito §4.2.1 per delineare le attività del processo di gestione della configurazione e aggiunta di §4.2.2 per inizializzare il processo	Verificatore	Laura Cameran	2019-04-01
2.2.2	Aggiunta §4.1.3.3 per normare chi e quando approva il rilascio dei documenti; §4.1.4.3, §4.1.4.3.1 e §4.1.4.3.2 per definire il processo di controllo	Amministratore	Laura Cameran	2019-03-30
2.2.1	Verifica §4.3.2	Verificatore	Matteo Marchiori	2019-03-29
2.2.0	Verifica	Verificatore	Laura Cameran	2019-03-29
2.1.5	Spostato §2.1.9 in un posto più consono e aggiunti §2.1.6.2, §2.1.7 per raffinare i coinvolgimenti con committente e proponente	Amministratore	Laura Cameran	2019-03-27
2.1.4	Aggiunti §2.1.6 e §2.1.6.1 per ampliare processi di fornitura	Amministratore	Laura Cameran	2019-03-27
2.1.3	Spostate norme relative ai diagrammi UML nelle sezioni §2.2.2 e §2.2.3 per adiuvare la lettura del documento.	Amministratore	Matteo Marchiori	2019-03-26
2.1.2	Aggiunte metriche in §4.4.6.2, §4.4.6.3 per il code coverage, §4.5.3.2.1. §4.4.7.2.1 e §4.4.7.3.1 per normare le metriche sui test	Verificatore	Nicola Carlesso	2019-03-25
2.1.1	Aggiunta §4.1.6.2.1 con DockerHub per versionamento	Amministratore	Ciprian Voinea	2019-03-25
2.1.0	Verifica	Verificatore	Laura Cameran	2019-03-25
2.0.3	Aggiunta §4.4.7.1 per i test dinamici	Amministratore	Nicola Carlesso	2019-03-24
2.0.2	Spostata nelle *NormeDiProgetto v4.0.0*$_{\text{D}}$ gli standard di qualità §4.3.2 perché attinenti al documento	Amministratore	Nicola Carlesso	2019-03-22
2.0.1	Modifiche a §4.1.4.1.3 per normare il tracciamento delle decisioni dei verbali	Amministratore	Matteo Marchiori	2019-03-20
2.0.0	Approvazione per il rilascio	Responsabile	Samuele Gardin	2019-03-06
1.4.0	Verifica finale	Verificatore	Timoty Granziero	2019-03-04

Versione	Descrizione	Ruolo	Nominativo	Data
1.3.5	Aggiunto §2.2.4.14	Amministratore	Nicola Carlesso	2019-03-04
1.3.4	Aggiunti §4.4.5.2, §4.4.6.3.1 e §3.1.6.3	Amministratore	Ciprian Voinea	2019-03-03
1.3.3	Completato §2.2.3.3	Amministratore	Matteo Marchiori	2019-03-01
1.3.2	Aggiunta immagine in §4.4.6.3.2 e completato paragrafo	Amministratore	Laura Cameran	2019-03-01
1.3.1	Aggiunta sezione §4.4.6.3 e sottosezioni	Amministratore	Laura Cameran	2019-02-28
1.3.0	Verifica	Verificatore	Laura Cameran	2019-02-27
1.2.4	Completato §2.2.3.2.3	Amministratore	Ciprian Voinea	2019-02-27
1.2.3	Completato §2.2.3.2.1	Amministratore	Ciprian Voinea	2019-02-25
1.2.2	Completato §2.2.3.2.2	Amministratore	Matteo Marchiori	2019-02-22
1.2.1	Aggiunto §4.2	Amministratore	Ciprian Voinea	2019-02-20
1.2.0	Verifica	Verificatore	Samuele Gardin	2019-02-18
1.1.5	Completati paragrafi §4.5.3.3, §4.4.7.3 e §4.4.8	Amministratore	Nicola Carlesso	2019-02-16
1.1.4	Completati paragrafi §4.5.3.2, §4.4.7.2, §4.4.7.4	Amministratore	Laura Cameran	2019-02-16
1.1.3	Completato paragrafo §2.1.3	Amministratore	Laura Cameran	2019-02-11
1.1.2	Aggiunto §4.1.6.2	Amministratore	Nicola Carlesso	2019-02-09
1.1.1	Modifica norme verbali, aggiornamento §4.1.5.1	Amministratore	Nicola Carlesso	2019-02-09
1.1.0	Verifica	Verificatore	Matteo Marchiori	2019-02-05
1.0.5	Aggiornamento §4.1.5.1	Amministratore	Laura Cameran	2019-02-03
1.0.4	Aggiunta attualizzazione rischi a §3.1.3.5	Amministratore	Timoty Granziero	2019-02-01
1.0.3	Aggiornato scheletro di §4.4.7, aggiornato §2.2.5.1.2 e §4.4.8	Analista	Laura Cameran	2019-02-01
1.0.2	Aggiunto scheletro per §2.2.3.2.2, §2.2.3.2.3, §2.2.3.3 e §2.2.4	Amministratore	Timoty Granziero	2019-01-29
1.0.1	Aggiornato lo stile Ammazionale	Amministratore	Timoty Granziero	2019-01-27
1.0.0	Approvazione per il rilascio	Responsabile	Nicola Carlesso	2019-01-10
0.4.0	Verifica finale	Verificatore	Matteo Marchiori	2019-01-08
0.3.4	Revsionato §1.5 e completato §3.2	Verificatore	Laura Cameran	2019-01-06
0.3.3	Revisionato §2	Verificatore	Laura Cameran	2019-01-06
0.3.2	Completamento§3.1	Amministratore	Samuele Gardin	2019-01-05

Versione	Descrizione	Ruolo	Nominativo	Data
0.3.1	Aggiornamento §3.1	Verificatore	Laura Cameran	2019-01-05
0.3.0	Verifica documento	Verificatore	Ciprian Voinea	2018-12-29
0.2.5	Aggiunto §4.4	Verificatore	Nicola Carlesso	2018-12-10
0.2.4	Aggiunti §4.4.7 e §4.4.6	Verificatore	Nicola Carlesso	2018-12-08
0.2.3	Aggiunto §4.1.5	Amministratore	Timoty Granziero	2018-12-06
0.2.2	Aggiunto §2.2.2.3.2	Analista	Laura Cameran	2018-12-05
0.2.1	Aggiunto §2.2	Amministratore	Timoty Granziero	2018-12-04
0.2.0	Verifica	Verificatore	Samuele Gardin	2018-12-04
0.1.5	Aggiunto §4.1.2	Amministratore	Timoty Granziero	2018-12-02
0.1.4	Aggiunto §1.5	Amministratore	Laura Cameran	2018-12-01
0.1.3	Aggiunto §3.1.3.4, §2.2.2.2 e §4.4.6.1	Verificatore	Nicola Carlesso	2018-12-01
0.1.2	Aggiunto §3.1.6	Verificatore	Timoty Granziero	2018-11-30
0.1.1	Aggiunto §3.1.2	Verificatore	Nicola Carlesso	2018-11-30
0.1.0	Verifica	Verificatore	Timoty Granziero	2018-11-29
0.0.8	Completato §2.1	Amministratore	Laura Cameran	2018-11-28
0.0.7	Aggiunto §4.1.4.2 e §4.1.3.2.2	Verificatore	Timoty Granziero	2018-11-27
0.0.6	Aggiunto §2.2.5.1	Amministratore	Laura Cameran	2018-11-27
0.0.5	Aggiunto §4.1	Amministratore	Laura Cameran	2018-11-26
0.0.4	Aggiunto §4.1.4.1	Amministratore	Laura Cameran	2018-11-26
0.0.3	Aggiunto §1	Amministratore	Laura Cameran	2018-11-25
0.0.2	Aggiunto §3.2	Amministratore	Laura Cameran	2018-11-23
0.0.1	Creazione template	Amministratore	Timoty Granziero	2018-11-22

Indice

Elenco delle tabelle

Elenco delle figure

1 Introduzione

1.1 Glossario e documenti esterni

Al fine di rendere il documento più chiaro possibile, i termini che possono assumere un significato ambiguo o i riferimenti a documenti esterni avranno delle diciture convenzionali:

- **D**: indica che il termine si riferisce al nome di un particolare documento (ad esempio *PianoDiProgetto v4.0.0*$_D$).
- **G**: indica che il termine si riferisce ad una voce riportata nel *Glossario v3.0.0*$_D$ (ad esempio REDMINE$_G$).

1.2 Premessa

Il DOCUMENTO$_G$ che segue verrà prodotto incrementalmente al presentarsi della necessità di redigere nuove NORME$_G$.

1.3 Scopo del documento

Il presente documento ha l'obiettivo di mettere in chiaro le norme, le convenzioni e le tecnologie che adottiamo durante lo svolgimento del PROGETTO$_G$. Ognuno di noi è tenuto ad osservarlo rigorosamente, per mantenere consistenza ed omogeneità in ogni aspetto, durante tutta la durata del progetto.

1.4 Scopo del prodotto

Lo scopo del PRODOTTO$_G$ è creare un APPLICATIVO$_G$ per poter gestire i messaggi o le segnalazioni provenienti da diversi prodotti per la realizzazione di software. Queste segnalazioni passano attraverso un BROKER$_G$ che gestisce i canali a loro dedicate per poi distribuirle ad applicazioni di messaggistica.
Il software dovrà inoltre essere in grado di riconoscere il TOPIC$_G$ dei messaggi in input per poterli inviare a determinati canali a cui i destinatari dovranno iscriversi.
È anche richiesto di creare un canale specifico per gestire le particolari esigenze dell'azienda. Questo dovrà essere in grado, attraverso la lettura di particolari METADATI$_G$, di reindirizzare i messaggi ricevuti al destinatario più appropriato.

1.5 Riferimenti

1.5.1 Normativi

1.5.2 Informativi

1. ISO 8601
   (a) https://it.wikipedia.org/wiki/ISO_8601#Data_completa
   (b) https://it.wikipedia.org/wiki/ISO_8601#Orari
2. ISO/IEC 12207
   https://en.wikipedia.org/wiki/ISO/IEC_12207
3. APACHE KAFKA$_G$
   https://kafka.apache.org/documentation/
4. Descrizione dei ruoli di progetto
   https://www.math.unipd.it/~tullio/IS-1/2018/Progetto/RO.html

5. DockerG
https://docs.docker.com/

6. DockerHubG
https://docs.docker.com/docker-hub/

7. Fonte Figura 5:
https://en.wikipedia.org/wiki/Test-driven_development

8. Fonte Figura 3:
https://it.wikipedia.org/wiki/ISO/IEC_9126

9. Fonte Figura 4:
https://it.wikipedia.org/wiki/Ciclo_di_Deming

10. GanttProject
https://www.ganttproject.biz/

11. GitLabG
https://docs.gitlab.com/ee/

12. GNU Aspell
http://aspell.net/

13. Google Drive
https://www.google.com/drive/

14. LaTeX
https://www.latex-project.org/help/documentation/

15. Naming convention
https://en.wikipedia.org/wiki/Naming_convention_(programming)

16. PyCharm
https://www.jetbrains.com/pycharm/

17. Pytest
https://docs.pytest.org/en/latest/

18. Redmine
https://www.redmine.org/guide

19. Software Engineering (Ian Sommerville), 10° edizione, Pagina 648, Capitolo 22: Project management

20. Telegram
https://core.telegram.org/

21. TexStudio
https://www.texstudio.org/

22. Toggl
https://toggl.com/

23. Visual Studio Code
https://code.visualstudio.com/docs

24. Tipologie di architetture
https://www.math.unipd.it/~rcardin/sweb/2019/L03.pdf

25. Design pattern strutturali
https://www.math.unipd.it/~tullio/IS-1/2018/Dispense/E06.pdf

26. Design pattern creazionali
https://www.math.unipd.it/~tullio/IS-1/2018/Dispense/E07.pdf

27. Design pattern comportamentali
https://www.math.unipd.it/~tullio/IS-1/2018/Dispense/E07.pdf

28. Design pattern architetturali
https://www.math.unipd.it/~rcardin/sweb/2019/L02.pdf

29. Composition over inheritance
https://en.wikipedia.org/wiki/Composition_over_inheritance

30. Formato della data
https://it.wikipedia.org/wiki/Formato_della_data

31. The Twelve-Factor App$_G$
https://12factor.net/#the_twelve_factors

32. PEP 8$_G$
https://www.python.org/dev/peps/pep-0008

2 Processi primari

2.1 Processo di fornitura

2.1.1 Scopo

La sezione corrente ha lo scopo di riportare le attività principali che ci impegniamo ad attuare come FORNITORI$_G$ per la proponente Imola Informatica.
Il processo di fornitura ha lo scopo di curare e mantenere i rapporti con le figure esterne al gruppo AlphaSix. Questi perciò devono essere costanti, sensati e che sappiano dare un valore aggiunto al progetto.

2.1.2 Ricerca delle tecnologie

Ognuno di noi deve approfondire la propria conoscenza su tecnologie e FRAMEWORK$_G$ vari, in modo da discutere insieme quali sono quelli di interesse comune e di utilità per il progetto. Questo prevede in seguito l'auto-formazione di quelle scelte in comune accordo per acquisirne una buona padronanza.

2.1.3 Normazione

L'attività di normazione prevede l'incremento del documento corrente ogni qual volta lo si ritiene necessario. Essendo infatti un documento incrementale, è possibile che vengano aggiunte delle nuove regole o aggiornate quelle già presenti. Facendo un esempio: arrivati al momento di riscrittura dei casi d'uso, dopo la prima revisione, abbiamo dovuto suddividere i casi in categorie ben distinte, in modo che fosse evidente il sottosistema di riferimento di Butterfly. Di conseguenza è necessario cambiare il codice identificativo dei casi d'uso e la relativa norma nel paragrafo §2.2.2.3.

2.1.4 Strumenti forniti

Il proponente e il Prof. Tullio Vardanega ci hanno messo a disposizione diversi strumenti per la realizzazione del progetto e come raggiungere una buona qualità nei prodotti.
Il Prof. Tullio Vardanega è a disposizione per:

- Chiarire eventuali dubbi relativi alla realizzazione del progetto in termini di buone pratiche e regole per la sua realizzazione
- Mettere a disposizione materiale didattico per il corretto sviluppo delle varie fasi del progetto, e la stesura della documentazione

Il proponente Davide Zanetti di Imola Informatica ci mette a disposizione:

- Un CLUSTER$_G$ dell'azienda su cui è installato RANCHER$_G$ per installare in remoto i CONTAINER$_G$ necessari a Butterfly
- Un indirizzo Email col dominio di Imola Informatica per inviare i messaggi da Butterfly

Anche Davide Zanetti è disponibile a degli incontri via HANGOUNTS$_G$ per eventuali chiarimenti tecnici sullo sviluppo del progetto.

2.1.5 Studio di fattibilità

In quest'attività viene prodotto il documento *StudioDiFattibilità v1.0.0*$_D$ al fine di analizzare ogni CAPITOLATO$_G$ e scegliere quale contratto accettare. Nello specifico, il documento in ogni sezione contiene:

- **Descrizione generale**: breve descrizione del capitolato.
- **Obiettivo finale**: obiettivo da raggiungere.
- **Tecnologie coinvolte**: elenco delle tecnologie direttamente coinvolte esplicitate nel capitolato.
- **Valutazione conclusiva**: giudizio finale del team di sviluppo.

Tale documento è fondamentale per chiarire ai vari proponenti la scelta di un capitolato, chiarendone i punti di forza e quelli deboli secondo il nostro punto di vista. In questo modo il cliente può già osservare quali possono essere le nostre competenze e abilità.

2.1.6 Coinvolgimento dell'acquirente

Data la natura del progetto, riteniamo essenziale la comunicazione con il cliente per qualunque tipo di aggiornamento rilevante, come ad esempio una BASELINE$_G$. In particolare, avere un FEEDBACK$_G$ da parte del cliente ci è molto utile ai fini del proseguimento del progetto. Per esempio, un feedback del nostro PROOF OF CONCEPT$_G$ ci aiuta a capire se stiamo realizzando il prodotto come Imola Informatica lo aveva ideato. Per ricevere questo feedback e procedere ad uno scambio di idee e opinioni, preferiamo staibilire un colloquio. Questo colloquio può avvenire tramite:

- Un incontro di persona in un luogo fissato insieme (e.g. aula universitaria, bar, ecc.)
- Una videochiamata con una tecnologia scelta (e.g. HANGOUTS$_G$)

In alternativa al colloquio, nel caso di brevi domande volte al chiarimento di piccoli dubbi, è possibile combinare una serie di Email o messaggi Telegram.

2.1.6.1 Negoziazione

Talvolta, in corso d'opera, è necessario negoziare gli accordi da contratto precedentemente presi con il cliente. Questo può accadere per più motivi, principalmente in momenti in cui noi fornitori realizziamo di non poter soddisfare qualche requisito nel modo in cui lo vorrebbe il nostro cliente. In seguito ad un inconveniente come questo, riteniamo necessario richiedere un colloquio con il cliente per poter esporre la nostra proposta alternativa e definire dei cambiamenti tali da assecondare i bisogni di entrambe le parti.
A seguito di un cambiamento in accordo col committente è d'obbligo riportare i cambiamenti all'interno dell'*AnalisiDeiRequisiti v3.0.0*$_D$ che risulta essere il contratto che tutela il fornitore e il committente per quanto riguarda i requisiti che il prodotto finale deve soddisfare.

2.1.6.2 Accessibilità del materiale necessario da contratto

Tutta la documentazione richiesta dall'acquirente come resoconti dei problemi riscontrati, BUG$_G$ reporting, istruzioni per l'avvio del prodotto, ecc... devono essere resi accessibili al cliente. Questo può avvenire attraverso email, Drive [1] o repository pubblica su GitHub. Oltre alla documentazione, anche il prodotto software deve essere rilasciato come specificato nel contratto.

2.1.7 Coinvolgimento del committente

Durante alcune baseline, ad esempio le revisioni, illustriamo al committente delle dimostrazioni del nostro prodotto in modo da ottenere anche da esso una valutazione utile per noi all'avanzamento del progetto.

[1] Riferirsi alla voce 13 in §1.5.2 per ulteriori informazioni su Drive

2.1.7.1 Accessibilità del materiale relativo alle revisioni

Per ogni revisione di avanzamento e accettazione rendiamo disponibile il materiale richiesto dal committente tramite:

- **Drive**: per ogni revisione rendiamo disponibile su Google Drive$_G$ l'ultima versione dei documenti relativi a Butterfly
- **GitHub**: il codice del prodotto è disponibile in una repository$_G$ pubblica. Tale repository non risulta essere il luogo dove il software viene sviluppato, ma solo il codice prodotto dopo una data revisione
- **CD-ROM**: il prodotto finale e installabile viene consegnato al committente all'interno di un dispositivo fisico come da lui richiesto, in particolare di un CD-ROM per motivi di sicurezza

2.1.8 Preparazione in vista della revisione

In questa attività prepariamo tutto il materiale necessario per il buon superamento della revisione:

- I vari documenti, i verbali e la lettera di presentazione per quanto riguarda la documentazione in ingresso
- I diagrammi UML$_G$ e il codice necessario per il superamento delle varie revisioni del progetto
- Le diapositive per l'esposizione
- Demo di dimostrazione del prodotto

2.1.8.1 Collaudo finale

In vista del collaudo finale, grazie ai vari confronti con il Prof. Tullio Vardanega e Davide Zanetti, tutti i documenti dovranno essere corretti nella forma e nei contenuti, almeno per quanto riguardano le correzioni fatte in sede di revisione. I requisiti richiesti dal capitolato ed inseriti in *AnalisiDeiRequisiti v3.0.0*$_D$dovranno essere totalmente soddisfatti, insieme anche ad eventuali richieste marginali di Davide Zanetti.
La presentazione di Butterfly attraverso una demo deve poter essere efficace mostrando la maggior parte delle funzionalità del progetto in modo chiaro e veloce, e poter far fronte ad eventuali richieste del committente.
In sede di collaudo dobbiamo mostrare i possibili miglioramenti di Butterfly, in modo da trasmettere il potenziale dell'applicazione e di una possibile futura seconda versione che la proponente potrà sviluppare.

2.1.9 Verifica post revisione

In seguito alla valutazione di ogni revisione riteniamo necessario conoscere i motivi negativi, ma anche positivi, di tale giudizio. In tal modo possiamo correggere meglio i difetti dei nostri prodotti e migliorarne i punti di forza.
Se sono presenti molteplici dubbi in rapporto alle correzioni riportate o ci sembra utile un confronto, entro la settimana successiva alla pubblicazione del giudizio della revisione, cerchiamo di fissare, se possibile, un colloquio con:

- **Proponente**: Davide Zanetti di Imola Informatica.
- **Committente**: Prof. Tullio Vardanega e Prof. Riccardo Cardin.

2.1.10 Circoscrizione dei rischi

Nell'eventualità che si possa presentare un problema con il prodotto Butterfly, può essere necessario comunicarlo a Davide Zanetti o a il Prof. Tullio Vardanega. Tale comunicazione deve presentare in modo chiaro il problema e, se possibile, allegare un'eventuale soluzione.
Ad esempio: durante il periodo di progettazione ci siamo accorti che la metodologia di inoltro dei messaggi risultava incoerente in alcuni aspetti. Questo ci ha portato a pensare ad una diversa gestione delle priorità dei progetti. Subito dopo abbiamo contattato Davide Zanetti per mostrare il problema e la possibile soluzione che è stata successivamente approvata con qualche modifica congiuntamente concordata.

2.2 Processo di sviluppo

2.2.1 Scopo

Lo sviluppo consiste nell'affrontare le attività volte a produrre il software richiesto dal proponente. Per una corretta implementazione di questo PROCESSO$_G$, è necessario:

- Fissare degli obiettivi di sviluppo
- Realizzare un prodotto che sia conforme a:
  - REQUISITI$_G$ definiti dal proponente
  - Test definiti dalle norme di QUALITÀ$_G$

Lo standard ISO/IEC 12207:1995[2] definisce il processo di sviluppo quel processo contenente tutte le attività relative al prodotto finale, quali:

- Analisi dei requisiti
- Progettazione
- Codifica
- Integrazione ed installazione

2.2.2 Analisi dei requisiti

Gli Analisti si occupano di redigere l'*AnalisiDeiRequisiti v3.0.0*$_D$, composta come segue:

- Descrizione generale del prodotto
- Modellazione concettuale del SISTEMA$_G$ tramite la definizione dei vari CASI D'USO$_G$
- Classificazione e tracciamento dei requisiti

2.2.2.1 Denominazione dei requisiti

Ogni requisito che è stato individuato durante l'analisi presenta il seguente identificativo univoco:

`R[Numero][Tipo][Priorità]`

- **R**: si riferisce a requisito.
- **Numero**: corrisponde ad un numero che cerca di seguire la struttura del documento ed è progressivo. Inizia da 1.

[2] Riferirsi alla voce 2 in §1.5.2

- **Tipo**: segnala la tipologia di requisito che può essere di:
  - **F**: funzionalità, che ha a che vedere con le funzionalità del sistema software.
  - **Q**: qualità, che riguarda tecniche ad hoc.
  - **V**: vincolo, proposto da Imola Informatica.
- **Priorità**: indica il grado di urgenza per il soddisfacimento di un requisito, come:
  - **0**: opzionale, di grado basso e solo marginalmente utile.
  - **1**: desiderabile, di medio livello quindi non strettamente necessario ma che dà valore aggiunto.
  - **2**: obbligatorio, di grado alto quindi irrinunciabile per il COMMITTENTE$_G$ e impossibile da tralasciare.

Esempio: `R2Q1` indica il secondo requisito di qualità ed è desiderabile.

2.2.2.2 Metriche sui requisiti

Dato che non è fisso il numero dei requisiti di un progetto, abbiamo scelto una serie di metriche dove il valore ottimale da raggiungere è sempre uguale, lo zero. Abbiamo anche scelto di contare il numero di requisiti non soddisfatti invece che il contrario. Lo stesso ragionamento è valido per quanto riguarda i rischi che possono verificarsi nel corso del progetto. Gli obiettivi che si vogliono raggiungere attraverso tali metriche possono essere stabiliti solo a progetto concluso.
La denominazione delle metriche è descritta in §3.1.2.3.

2.2.2.2.1 MPR005 Requisiti obbligatori non soddisfatti

Per adempiere completamente alla richiesta del cliente, ci serve individuare tutti i requisiti presenti nella sua richiesta, impliciti, espliciti, diretti e derivati. Alcuni sono imprescindibili, detti obbligatori, e il loro soddisfacimento determina la buona riuscita del progetto.
Metrica: numero dei requisiti obbligatori non soddisfatti.

2.2.2.2.2 MPR006 Requisiti desiderabili non soddisfatti

I requisiti desiderabili non sono necessari, ma offrono un valore aggiunto al progetto.
Metrica: numero dei requisiti desiderabili non soddisfatti.

2.2.2.2.3 MPR007 Requisiti opzionali non soddisfatti

Tali requisiti dovranno essere adempiuti solo nel momento in cui tutti i requisiti obbligatori saranno soddisfatti. Possono essere concordati col cliente in corso d'opera.
Metrica: numero dei requisiti opzionali non soddisfatti.

2.2.2.3 Casi d'uso

Un caso d'uso è una tecnica che identifica i requisiti funzionali descrivendo le interazioni tra il sistema di riferimento e un utente ad esso esterno.
Ogni caso d'uso che si vuol descrivere presenta:

- **Codice**: per l'identificazione.
- **Titolo**: denominazione del caso d'uso, possibilmente breve.
- **Attori primari**: tutti gli ATTORI$_G$ primari coinvolti.

- **Attori secondari**: opzionale, tutti gli attori secondari coinvolti.
- **Descrizione**: per spiegare più nel dettaglio le azioni che vengono compiute.
- **Precondizione**: per rappresentare lo stato del sistema nell'istante precedente.
- **Postcondizione**: per rappresentare lo stato del sistema l'istante successivo.
- **Scenario principale**: contenente la serie di azioni da compiere numerate nell'ordine in cui vengono compiute.
- **Estensioni**: opzionale, per azioni inerenti a scenari alternativi come d'eccezione o errore.

2.2.2.3.1 Denominazione del codice identificativo

Ogni codice presenta una forma del tipo:

`UC[Numero]-[Sottosistema]`

- **UC**: sta per "User Case", l'equivalente inglese di "caso d'uso".
- **Numero**: numero progressivo che si riferisce al caso. Se il caso d'uso è principale, è un semplice intero (e.g. 1 per il primo caso d'uso). Mentre se è un sotto caso, presenta due interi separati da un punto (e.g 1.1 per per il primo figlio del primo caso d'uso).
- **Sistema**: indica il sistema preso come riferimento per il caso d'uso, che può essere Butterfly o un suo sottosistema:
  - **PR**: Producer Redmine.
  - **PG**: Producer Gitlab.
  - **GP**: Gestore Personale.
  - **CT**: Consumer Telegram.
  - **CE**: Consumer Email.
  - **BT**: bot Telegram.
  - **SE**: server Email.

Vi possono essere al massimo tre livelli di annidamento, come nel seguente esempio:

- `UC1` identifica il caso d'uso con il numero 1
- `UC1.1` identifica il sottocaso 1 del caso d'uso 1
- `UC1.1.1` identifica il sottocaso 1 del caso d'uso 1.1

Per ogni livello i numeri partono da 1.

2.2.2.3.2 Diagrammi dei casi d'uso

Per i diagrammi dei casi d'uso utilizziamo lo strumento Draw.io$_G$ e il linguaggio UML$_G$ 2.0. Essi descrivono la visione di un utente esterno al sistema e non danno nessun dettaglio implementativo. I componenti$_G$ contenuti in questo tipo di diagrammi sono:

- Attore$_G$
- Casi d'uso
- Relazioni, che possono essere di:

- **Associazione**: è sempre presente e rappresenta una comunicazione diretta dell'attore con il caso d'uso.
- **Inclusione**: è una funzionalità comune che coinvolge più casi d'uso, in cui ogni istanza del primo esegue necessariamente il secondo. Tutte le inclusioni vengono sempre eseguite dall'utente.
- **Estensione**: aumenta le funzionalità di un caso d'uso coinvolgendone un altro. L'esecuzione di quest'ultimo interrompe il precedente, ma non è necessariamente detto che tutte le estensioni vengano eseguite.
- **Generalizzazione**: avviene tra casi d'uso o tra attori e rappresenta delle modifiche alle caratteristiche di base.

Per la specifica di un caso d'uso si fa riferimento al paragrafo §2.2.2.3.

2.2.3 Progettazione

La progettazione è l'attività svolta dai progettisti con il fine di dare una soluzione soddisfacente per tutti gli stakeholder coinvolti nel rispetto delle BEST PRACTICE$_G$ e dei design pattern scelti, attraverso la descrizione dell'architettura del software.

2.2.3.1 Obiettivi

Gli obiettivi dell'attività di progettazione sono:

- Definire l'architettura logica del prodotto.
- Organizzare le responsabilità di realizzazione.
- Garantire la qualità del prodotto finale.

2.2.3.2 Strumenti

2.2.3.2.1 Diagrammi dei package

Per i diagrammi dei package utilizziamo lo strumento DRAW.IO$_G$ e il linguaggio UML 2.0.
Vengono usati insieme ai diagrammi delle classi per raggrupparne gli elementi in un elemento di livello più alto. Rappresentiamo i package con un rettangolo che contiene altri package o classi, e un'etichetta con il nome del package. Ogni package identifica uno spazio dei nomi, che serve per dare un nome qualificato alle classi. Gli elementi di un package possono avere visibilità pubblica (+) o privata (-) verso l'esterno. Nella progettazione adottiamo il principio del common closure principle, secondo il quale elementi dello stesso package condividono la stessa causa di cambiamento. Indichiamo le dipendenze tra package con frecce tratteggiate. Rappresentiamo le dipendenze perché:

- Individuiamo subito le dipendenze circolari, evitandole
- Vediamo quali elementi dovrebbero essere più stabili (avendo più dipendenze entranti)

2.2.3.2.2 Diagrammi delle classi

Per i diagrammi delle classi utilizziamo lo strumento DRAW.IO$_G$ e il linguaggio UML 2.0.
Questi diagrammi strutturali descrivono ad alto livello i tipi di oggetto presenti nel sistema e le relazioni che intercorrono tra essi. I diagrammi delle classi sono composti dalle seguenti proprietà:

- **Attributi**: rappresentano lo stato dell'oggetto. Presentano le caratteristiche:
  - **Visibilità**: pubblica (+), privata (-) o protetta (#) rispetto alle altre classi.

  - **Nome**: nome dell'attributo.
  - **Tipo**: tipo dell'attributo (intero, stringa...).
  - **Molteplicità**: numero di occorrenze dell'attributo (opzionale).
  - **Default**: valore di default dell'attributo (opzionale).
  - **Altro**: ... (opzionali)

- **Associazioni**: rappresentano legami tra classi ed hanno un nome.

  Le rappresentiamo con linee continue orientate tra due classi.

Nella parte bassa del rettangolo delle classi rappresentiamo le operazioni fornite dalla classe, composte da:

- **Visibilità**: come per gli attributi.
- **Nome**: nome dell'operazione.
- **Lista dei parametri**: parametri necessari all'operazione. Sono formati da:
  - **Direzione**: indica se il parametro è in lettura (in), scrittura (out) o entrambi (inout). Opzionale, di default è in lettura.
  - **Nome**: nome del parametro.
  - **Tipo**: tipo del parametro.
  - **Default**: valore di default del parametro (opzionale).
- **Tipo di ritorno**: tipo ritornato dall'operazione.

Tra le classi rappresentiamo le seguenti relazioni di dipendenza:

- **Dipendenza**: rappresentata da una linea orientata tratteggiata, indica che una classe usa un'operazione fornita dalla classe puntata.
- **Associazione**: rappresentata da una linea continua semplice, indica che una classe possiede uno o più attributi (a seconda delle molteplicità indicate sulla linea) dell'altra, o viceversa.
- **Aggregazione**: rappresentata da un diamante vuoto seguito da una linea continua, indica che una classe (puntata dal diamante) possiede un'istanza di un'altra classe tra gli attributi. Tale istanza può essere condivisa.
- **Composizione**: rappresentata da un diamante pieno seguito da una linea continua, è come l'aggregazione ma in tal caso l'aggregato non è condiviso tra più istanze.
- **Generalizzazione**: rappresentata da una freccia vuota, indica che ogni classe che punta verso un'altra è anche un'istanza di essa (eredita metodi e attributi).

Usiamo inoltre i seguenti costrutti:

- **Classi astratte**: il nome è indicato in corsivo, non possono essere istanziate. Presentano almeno un'operazione astratta (con il nome in corsivo) e possono avere attributi.

  Tali classi generalizzano altre classi che devono implementare le operazioni astratte per essere istanziate, altrimenti sono anch'esse astratte e generalizzano altre classi.

- **Interfacce**: indicate da un pallino vuoto e una linea continua verso classi, presentano solamente le operazioni che vengono implementate dalle classi collegate con la linea.

  Possono essere indicate con un pallino vuoto e un semicerchio a destra per indicare che l'interfaccia viene usata da un componente in un punto qualsiasi (ritorno di un'operazione, attributo...).

2.2.3.2.3 Diagrammi di sequenza

Per i diagrammi di sequenza utilizziamo lo strumento Draw.io$_G$ e il linguaggio UML 2.0.
Usiamo questi diagrammi per descrivere come la collaborazione tramite messaggi di un gruppo di oggetti realizza un comportamento.
Essi mostrano come gli oggetti interagiscono nel tempo, leggendo il diagramma dall'alto al basso.
I principali costrutti usati sono:

- **Partecipanti**: rappresentati da un rettangolo contenente il nome dell'entità (spesso coincide col nome di una classe) e da una barra di attivazione (rettangolo che si prolunga per tutta la durata in cui l'entità è attiva), descrivono l'entità che detiene il flusso nel caso d'uso.
- **Messaggi**: rappresentano le operazioni chiamate tra le entità e i dati scambiati.

  Sono di diverse tipologie:

  - **Sincroni**: rappresentati da una freccia piena con il nome del messaggio ed eventuali parametri tra parentesi tonde, descrivono la chiamata di un messaggio in cui l'entità chiamante attende la risposta dall'entità chiamata.
  - **Asincroni**: rappresentati da una linea direzionata con il nome del messaggio ed eventuali parametri tra parentesi tonde, descrivono la chiamata di un messaggio in cui l'entità chiamante non attende la risposta dall'entità chiamata.
  - **Ritorno**: rappresentati da una linea tratteggiata direzionata, indicano un messaggio di ritorno che risponde a un precedente messaggio di chiamata.
  - **Creazione**: rappresentati da una linea tratteggiata direzionata e dalla notazione «create», indicano la creazione di una nuova entità da parte dell'entità chiamante.
  - **Distruzione**: rappresentati da una freccia piena e dalla notazione «destroy», indicano la distruzione di un'entità da parte dell'entità chiamante.

Per descrivere i cicli e le condizioni vengono utilizzati frame di interazione, rappresentati con un rettangolo che circonda i messaggi e le barre di attivazione coinvolte, con una label per indicare il tipo di frame, e da una guardia tra parentesi quadre nell'entità che ne scatena l'inizio. Vi sono i seguenti tipi di frame di interazione:

- **Alt**: più frame in alternativa, si sceglie in base alla guardia.
- **Opt**: opzionale, in base alla guardia si esegue o meno il frame.
- **Par**: parallelo, frame eseguiti in parallelo.
- **Loop**: ciclo eseguito più volte in base alla guardia.
- **Region**: regione critica eseguibile da un thread alla volta.
- **Ref**: l'interazione è definita in un altro diagramma.
- **Sd**: rachiude un diagramma di sequenza intero.

I diagrammi di sequenza sono inoltre utili per modellare la collaborazione tra entità.
In caso di controllo centralizzato vi sarà un'entità che gestisce i messaggi verso tutte le altre, mentre in caso di controllo distribuito ogni entità avrà compiti ben delineati, quindi avrà la responsabilità della chiamata dei messaggi verso le altre entità coinvolte.
Cerchiamo di usare per quanto possibile la modellazione a controllo distribuito, per non dare troppe responsabilità a una singola entità che diventerebbe il punto critico dell'architettura.

2.2.3.2.4 Architettura

L'architettura può essere considerata la "strategia" che si occupa della parte più esterna del software. L'architettura si deve occupare di come si relazionano i vali elementi del software, non come questi sono formati. Anche l'architettura deve essere costruita secondo una best practise, che può cambiare in base al tipo di software che deve essere sviluppato.
Un'architettura non si inventa ma si sceglie, e tra le più utilizzate troviamo:

- **Layer**: architettura che rispecchia la struttura di un'azienda e che vede il software sviluppato in più livelli disposti uno sopra l'altro. Un livello è dipendente esclusivamente da quello sottostante e comunica con esso. Generalmente i livelli sono Presentation layer, Business layer, Persisten layer e Database layer.
- **Event-driven**: architettura utile per la creazione di software che lavorano con messaggi asincroni elaborati in modo indipendente. Quest'architettura può essere sviluppata secondo la Mediator topology dove i messaggi vengono inseriti in un "contenitore" e smistati attraverso un apposito "mediator", oppure secondo la Broker topology dove i messaggi viaggiano attraverso un BROKER$_G$ entrando ed uscendo dai vari canali.
- **Microservizi**: architettura usata per quei software destinati a possedere un incremento costante e controllato. Il software viene diviso in moduli indipendenti tra loro che comunicano verso l'esterno attraverso API REST$_G$. I moduli al loro interno sono costruiti attraverso un'architettura a Layer.

2.2.3.2.5 Design Pattern

I DESIGN PATTERN$_G$ aiutano a creare le singole parti di un software in base ai compiti che queste hanno. Dato che un design pattern non è altro che una buona pratica di sviluppo, non è scontato che la loro presenza implichi una buona struttura del software, perciò è consigliato non abusarne. I design pattern si dividono in tre tipologie approfonditi in §1.5.2 ai punti 25, 26, 27 e 28:

- **Strutturali**: aiutano ad organizzare la struttura delle classi favorendo la loro comunicazione attraverso interfacce. Questi sono l'Adapter, il Decorator il Façade e il Proxy.
- **Creazionali**: aiutano a dividere gli oggetti dalla loro costruzione e implementazione favorendo l'INFORMATION HIDING$_G$. Questi sono il Singleton, il Builder, e l'Abstract Factory.
- **Comportamentali**: applicabili nel momento in cui un componente deve eseguire una specifica funzione per cui il design pattern è stato costruito. Questi sono il Command, l'Iterator, l'Observer, lo Strategy e il Template Method.
- **Architetturali**: utili per strutturare il software con una visione ad un livello più alto degli altri design pattern, ma non alto come la visione per stabilire l'architettura software. Questi sono il Dependency Injection e i Model-View pattern.

2.2.3.3 Qualità

Per garantire una progettazione che rispetti gli standard di qualità posti nel *PianoDiQualifica v4.0.0*$_D$, seguiamo le linee guida date dalle best practice.

2.2.3.3.1 Qualità dei diagrammi dei package

Evitiamo le dipendenze circolari tra le classi osservando le dipendenze tra i package. Ne consegue che vanno evitate le dipendenze circolari tra i package.

2.2.3.3.2 Qualità dei diagrammi delle classi

Teniamo gli attributi delle classi privati per quanto possibile. Evitiamo ove possibile di usare attributi modificabili dai metodi delle classi, in quanto ostacolano la scrittura dei test ed è meno comprensibile la responsabilità sull'attributo.
Evitiamo le relazioni bidirezionali perché si crea confusione su quale classe ha la responsabilità sulla relazione.
Minimizziamo le dipendenze tra le classi in modo da rendere il diagramma più comprensibile e cerchiamo di usare le relazioni di dipendenza debole rispetto alle dipendenze forti, per favorire il riuso del codice ed evitare modifiche inutili e a cascata.

2.2.3.3.3 Qualità dei diagrammi di sequenza

Andando più nel dettaglio rispetto ai diagrammi di attività, i diagrammi di sequenza vanno scritti in modo da rispettare quanto descritto dai diagrammi di attività e delle classi.
Per modellare la logica di controllo è meglio usare i diagrammi di attività perché di più semplice lettura.

2.2.3.3.4 Qualità dell'architettura

L'architettura è una delle scelte fondamentali da fare durante il processo della progettazione, perché essa stabilirà poi la natura dell'intero software. La scelta di un'architettura ha lo scopo di:

- ridurre l'accoppiamento
- ridurre la duplicazione di codice
- semplificare lo sviluppo
- semplificare il mantenimento del software

Bisogna dunque capire quale architettura si addice di più ai propri scopi in base alle qualità e ai difetti di ognuna.
Gli aggettivi che definiscono un'architettura sono:

- **Agilità**: con che facilità e velocità riesco a modificare il software.
- **Rilasciabile**: con che facilità riesco ad aggiornare o installare il software.
- **Verificabile**: con che facilità riesco a creare ed eseguire i vari tipi di test.
- **Performance**: con che velocità riesco ad ottenere le informazioni richieste.
- **Scalabilità**: con che facilità riesco a gestire la crescita del numero di utenti.
- **Sviluppabile**: con che facilità riesco a sviluppare il software applicando tale architettura.

Analizzando le architetture in §2.2.3.3.4, le loro qualità e difetti sono:

- **Layer**:
  - **Agilità**: poca, dato che il software tende a diventare un monolite.
  - **Rilasciabile**: poca, dato che ad ogni aggiornamento tutto il software deve essere reinstallato.
  - **Verificabile**: buona, dato che ottenere il Mock$_G$ dei vari livelli non è complesso.

  - **Performance**: poca, dato che un'informazione, per essere recuperata, deve passare per i vari livelli.
  - **Scalabilità**: poca, dato che la grande granularità rende laboriosa l'aggiunta di utenti.
  - **Sviluppabile**: buona, dato che è un'architettura molto conosciuta e che rispecchia la struttura di un'azienda.

- **Event-driven**:
  - **Agilità**: buona, dato che le varie componenti sono tutte isolate.
  - **Rilasciabile**: buona, dato che si può installare un componente alla volta.
  - **Verificabile**: poco, dato i molti tipi di messaggi da generare per eseguire i test.
  - **Performance**: buona, dato che i messaggi sono asincroni.
  - **Scalabilità**: buona, dato che si posseggono canali separati che possono aggiungere facilmente.
  - **Sviluppabile**: poca, dato che coi messaggi asincroni bisogna lavorare con INTERFACCE$_G$ che non possono cambiare nel tempo.

- **Microservizi**:
  - **Agilità**: buona, dato che ogni microservizio è autonomo.
  - **Rilasciabile**: buono, perché i microservizi possono essere installati singolarmente.
  - **Verificabile**: buono, perché i microservizi lavorano in modo autonomo.
  - **Performance**: poca, dato che il numero di chiamate delle API REST può essere alto.
  - **Scalabilità**: buona, dato che i microservizi possono essere aggiunti velocemente.
  - **Sviluppabile**: buona, dato che i microservizi vengono sviluppati da gruppi di persone poco numerosi e dunque più facilmente organizzati.

2.2.3.3.5 Qualità dei design pattern

L'uso dei design pattern non deve essere abusato per evitare complessità inutile nel software. Per questo la loro applicazione deve seguire tali step per comprendere se applicarne uno o meno:

1. Comprendere lo scopo del componente
2. Analizzarne la complessità strutturale
3. Elencare i bisogni di tale componente in termini di struttura del software
4. Vedere se un design pattern strutturale è utile per semplificare la struttura del componente
5. Vedere se la creazione del componente può essere semplificata con l'uso di un design pattern creazionale
6. Vedere se la funzione del componente corrisponde alla funzionalità descritta da un design pattern comportamentale, nel caso vedere se la sua applicazione rende più flessibile il componente

2.2.3.4 Applicazione

Per effettuare una buona progettazione è utile seguire una procedura prestabilita, in modo tale da non rendere il risultato confuso o non tracciabile. È bene ricordare che all'interno della progettazione non si sviluppa codice, tale passaggio avviene durante il processo di codifica che segue il quello di progettazione.
La progettazione ha il compito di far sintesi dei requisiti, provenienti dal processo di analisi, attraverso il software applicando prima una strategia top-down e successivamente bottom-up. I passi da seguire durante la prima parte top-down sono i seguenti:

1. Analizzare nell'insieme i requisiti del prodotto finale in termini di qualità architetturali (performance, scalabilità, ecc...)
2. Individuare l'architettura che più si addice al software da realizzare
3. Rappresentare l'architettura attraverso un diagramma. Questo mostrerà solo la struttura del software ad alto livello

Ora per ogni requisito troviamo la giusta struttura seguendo una strategia bottom-up:

1. Scegliere un requisito proveniente dal processo di analisi
2. Pensare a come potrebbe essere soddisfatto attraverso il software
3. Analizzare se per tale requisito esistono delle tecnologie preesistenti da poter usare
4. Dopo aver compreso il ruolo del componente software, pensare a come questo possa costruire i suoi oggetti o come si deve comportare
5. Analizzare se un design patter può semplificare la struttura o il comportamento del componente precedentemente ideato ed applicarlo
6. Riportare la struttura del componente in UML2.0

Dopo aver preparato i diagrammi dei vari componenti, osservando l'architettura scelta, studiare come questi possono essere uniti per garantire flessibilità, manutenibilità, modularità e affidabilità.

2.2.4 Codifica

2.2.4.1 Scopo

La presente sezione ha lo scopo di normare l'attività di codifica che svolgeremo nel corso della realizzazione del progetto Butterfly, per evitare qualsiasi tipo di inconsistenza e rendere il codice il quanto più omogeneo possibile.
Il progetto sarà sviluppato utilizzando il linguaggio Python$_G$. Per le norme di stile relative a questo linguaggio, seguiremo il PEP 8[3], lo standard Python che definisce lo stile da seguire nella codifica e in attività correlate ad essa.

2.2.4.2 Intestazione

Ogni sorgente Python conterrà la seguente intestazione:

```
"""
File: <nomefile.estensione>
Data creazione: <YYYY-MM-DD>

<descrizione>

Licenza: <nome licenza>
Versione: <X.Y.Z>
Creatore: <nome cognome: email>
Autori:
<nome cognome: email>
<nome cognome: email>
....
"""
```

2.2.4.3 Formattazione

Viene utilizzato uno spazio per separare simboli e identificatori.

```
1 import math
2
3 id_utente = "12324" # Si
4 id_utente="12324"   # No
5
6 math.atan2(5, 7)    # Si
7 math.atan2(5,7)     # No
```

2.2.4.4 Convenzioni di nominazione

- Per i nomi delle classi, viene usato `PascalCase`[4]
- Per le costanti, viene usato `MACRO_CASE`
- Per denominare tutto il resto, ad esempio variabili e metodi, si usa `snake_case`

Inoltre, tutti gli identificativi devono avere dei nomi significativi, relativi al contesto e alla loro utilità.

[3] Riferirsi alla voce 32 in §1.5.2
[4] Riferirsi alla voce 15 in §1.5.2

```
uscita = True          # Si
a = True               # No

count_match = 0        # Si
b = 0                  # No

def somma(x, y):       # Si, x e y sono concessi per il chiaro contesto matematico
    return x + y
def foo(a, b):         # No
    return a + b
```

2.2.4.5 Indentazione

Per l'indentazione, come da PEP 8, vanno usano 4 spazi. L'IDE utilizzato dovrà essere configurato in modo che, premendo il tasto TAB, verranno generati 4 spazi.

2.2.4.6 Stringhe

Preferire l'uso del carattere di apice singolo (') per circondare le stringhe. Usare il doppio apice (") solo per aumentare la leggibilità in caso sia presente nella stringa un apice singolo, o per le stringhe di documentazione (circondate da tripli apici doppi).
Per ottenere stringhe formattabili con parametri, usare le f-stringhe introdotte con Python 3.6, o, in caso di migliore leggibilità, usare la funzione `format()`.

```
stringa = 'una stringa'
def foo(x):
    """Stringa di documentazione"""
    pass
f_stringa = f'{stringa} parametrizzata!'
```

2.2.4.7 Lunghezza massima delle righe

Le righe di codice devono avere una lunghezza massima di 79 caratteri. Le righe di documentazione relative ad esempio a un metodo, invece, non devono contenere più di 72 caratteri. A tal fine, vengono divisi in più righe le istruzioni che superano il limite massimo di caratteri, per migliorare la leggibilità.

```
with open('/path/to/some/file/you/want/to/read') as file_1, \
        open('/path/to/some/file/being/written', 'w') as file_2:
    file_2.write(file_1.read())
```

2.2.4.8 Lunghezza massima di metodi e funzioni

Il numero massimo di righe per ogni metodo o funzione viene fissato a 50 linee di codice, numero necessario ad occupare un'intera schermata su un comune IDE. Funzioni più lunghe di 50 righe sono quasi sicuramente sintomo di una funzione che svolge troppi compiti, che sarebbe spacchettabile in più funzioni.

2.2.4.9 Parametri delle funzioni

Per i parametri delle funzioni, usare dei nomi esplicativi se il numero dei parametri non supera i 6. Altrimenti, preferire l'uso di liste arbitrarie di parametri e l'uso di coppie chiavi-valori. Documentare, di conseguenza, i nomi dei parametri che l'operatore ** (per le coppie chiave-valore) si aspetta, e sollevare una eccezione appropriata in caso si ottenga una chiave inaspettata.

```
def cheeseshop(kind, *arguments, **keywords):
    print("-- Do you have any", kind, "?")
    print("-- I'm sorry, we're all out of", kind)
    for arg in arguments:
        print(arg)
    print("-" * 40)
    for kw in keywords:
        print(kw, ":", keywords[kw])
```

2.2.4.10 Complessità ciclomatica

Va evitato il più possibile l'annidamento di più strutture di controllo. Per questo, stare sotto i tre livelli di annidamento è una buona prassi che semplifica leggibilità e DEBUGGING$_G$ e va rispettata il più possibile.

2.2.4.11 Ereditarietà

Evitare l'ereditarietà tra classi concrete. È invece possibile che una classe derivi da un'altra se quest'ultima è un'INTERFACCIA$_G$ o una classe astratta.
Viene adottato, per ovviare all'ereditarietà tra classi concrete, il principio di "Composition over inheritance"[5].
Va limitato, ove possibile, la dipendenza tra classi concrete: dipendere quindi sulle astrazioni (interfacce e classi astratte).

2.2.4.12 Commenti

Il Programmatore è tenuto ad inserire commenti ove ritiene che questo possa migliorare la leggibilità del codice. Per contro, i commenti vanno omessi dove il codice è auto esplicatorio.
Per inserire un commento, far seguire al simbolo di commento (`#`) uno spazio, e iniziare con una lettera maiuscola (e.g. `# Cerca il match del parametro x nella lista`). In caso di commento su più righe, solo la prima riga va iniziata con carattere maiuscolo.

2.2.4.13 Documentazione dei metodi

Ogni metodo è necessario che sia il più chiaro possibile. Oltre che a non dover avere una lunghezza non troppo elevata (visibile in §2.2.4.8), un metodo deve essere anche ben documentato, indicando il tipo dei parametri e quelli delle variabili ritornate.
La documentazione di un metodo deve avere la seguente forma:

```
def complex(real=0.0, imag=0.0):
    """
    Form a complex number.

    Keyword arguments:
    real -- the real part (default 0.0)
    imag -- the imaginary part (default 0.0)

    Return arguments:
    complex_zero -- the merge of real and imag
    """
    if imag == 0.0 and real == 0.0:
    return complex_zero
    ...
```

[5] Riferirsi alla voce 29 in §1.5.2.

2.2.4.14 Metriche per la codifica

Tutti i paragrafi sulle norme di codifica precedentemente descritte devono corrispondere ad una metrica in modo tale da tenere traccia dei cambiamenti nella qualità dei prodotti contenenti codice in Python.
La denominazione delle metriche è descritta in §3.1.2.3.
Le seguenti metriche possiedono le seguenti regole:

- Il titolo del metodo sottintende prima del loro inizio "numero di/dei" (e.g. "File senza intestazione" è inteso come "numero di file senza intestazione")
- Il termine "righe" intente "righe di codice"
- I termini "metodi", "classi", "righe" o "variabili" appartengono tutti al campo semantico dei linguaggi di programmazione

Le metriche inerenti alla codifica sono riferite ad una specifica regola di codifica:

- **MPS005 File senza intestazione**: §2.2.4.2.
- **MPS006 Righe non formattate**: §2.2.4.3.
- **MPS007 Nomi di variabili, metodi e classi non normati**: §2.2.4.4.
- **MPS008 Righe non indentate**: §2.2.4.5.
- **MPS009 Stringhe non normate**: §2.2.4.6.
- **MPS010 Righe troppo lunghe**: §2.2.4.7.
- **MPS011 Metodi troppo lunghi**: §2.2.4.8.
- **MPS012 Metodi con troppi parametri**: §2.2.4.9.
- **MPS013 Metodi con troppa complessità ciclomatica**: §2.2.4.10.
- **MPS014 Classi che implementano classi concrete**: §2.2.4.11.
- **MPS015 Commenti non normati**: §2.2.4.12.
- **MPS016 Metodi non documentati**: §2.2.4.13.

2.2.4.14.1 MPS005 File senza intestazione

Ogni file deve possedere l'intestazione definita per avere subito l'idea di cosa questo contiene.
Metrica: il numero di file che non possiedono l'intestazione definita.

2.2.4.14.2 MPS006 Righe non formattate

Il codice, per essere più leggibile deve contenere delle adeguate spaziature.
Metrica: numero adeguato di spaziature non eseguite.

2.2.4.14.3 MPS007 Nomi di variabili, metodi e classi non normati

I nomi delle variabili o i titoli delle classi o dei metodi devono seguire delle specifiche regole per essere il più possibile parlanti.
Metrica: numero di nomi di variabili o di titoli di metodi o classi che non rispettano le norme stabilite.

2.2.4.14.4 MPS008 Righe non indentate

Per rendere più chiaro l'inizio e la fine di un ciclo, metodo, classe o di una condizione è obbligatorio eseguire le dovute tabulazioni.
Metrica: numero di tabulazioni non eseguite.

2.2.4.14.5 MPS009 Stringhe non normate

In Python le stringhe possono essere create in vari modi, secondo le norme noi ne scegliamo una in particolare.
Metrica: numero di volte in cui le stringhe non vengono create nel modo stabilito.

2.2.4.14.6 MPS010 Righe troppo lunghe

Perché del codice sia leggibile, le sue istruzioni non possono essere troppo lunghe. Per questo è stata stabilita una soglia massima di caratteri per istruzione.
Metrica: numero di righe di codice che superano i 79 caratteri.

2.2.4.14.7 MPS011 Metodi troppo lunghi

Un metodo trasversale per impedire che un metodo compia troppe azioni è quello di limitarne la lunghezza.
Metrica: il numero di metodi o funzioni che sono più lunghi di 50 righe di codice.

2.2.4.14.8 MPS012 Metodi con troppi parametri

Limitare il numero di parametri in un metodo è un altro modo per evitare di caricarlo di troppe mansioni.
Metrica: il numero di metodi con più di 6 parametri.

2.2.4.14.9 MPS013 Metodi con troppa complessità ciclomatica

Un metodo che possiede molte istruzioni condizionali o cicli è difficile da testare e nel peggiore dei casi lento da eseguire. Perciò è buona norma limitare l'uso di queste istruzioni.
Metrica: numero di metodi che possiedono più di 3 cicli annidati.

2.2.4.14.10 MPS014 Classi che implementano classi concrete

L'ereditarietà tra classi deve essere evitata perché la presenza di dipendenze tra più componenti ne ostacola la modifica e la verifica.
Metrica: numero di classi che ereditano un'altra classe concreta.

2.2.4.14.11 MPS015 Commenti non normati

I commenti, per essere significativi devono essere scritti seguendo le norme indicate.
Metrica: numero di commenti che seguono le norme indicate.

2.2.4.14.12 MPS016 Metodi non documentati

La documentazione dei metodi, come per i file, serve per far saltare subito all'occhio il loro compito. La loro documentazione deve seguire le norme precedentemente indicate.
Metrica: numero di metodi che non possiede una documentazione a norma.

2.2.5 Strumenti di base

2.2.5.1 Ambiente di sviluppo

Nei paragrafi successivi vengono riportate le componenti software utilizzate da ogni membro del team per lo sviluppo del progetto.

2.2.5.1.1 Sistema operativo

Abbiamo deciso di usare, come sistema operativo, GNU/Linux, in particolare una qualsiasi distribuzione basata su Ubuntu 18.04.

2.2.5.1.2 IDE

Gli IDE principali scelti per lo sviluppo di codice Python sono PyCharm[6] e Visual Studio Code[7]. Il primo è proprietario e supportato dal team di IntelliJ, il secondo è un progetto OPEN SOURCE$_G$ di Microsoft, che ha per questo il supporto della comunità open source.
Entrambi offrono la maggior parte dei vantaggi utili al processo di sviluppo di Butterfly, tra cui:

- Highlighting del codice per una lettura agevole
- Pylint, un LINTER$_G$ per Python per l'analisi in tempo reale della qualità del codice
- Debugging
- Supporto a una miriade di plugin esterni

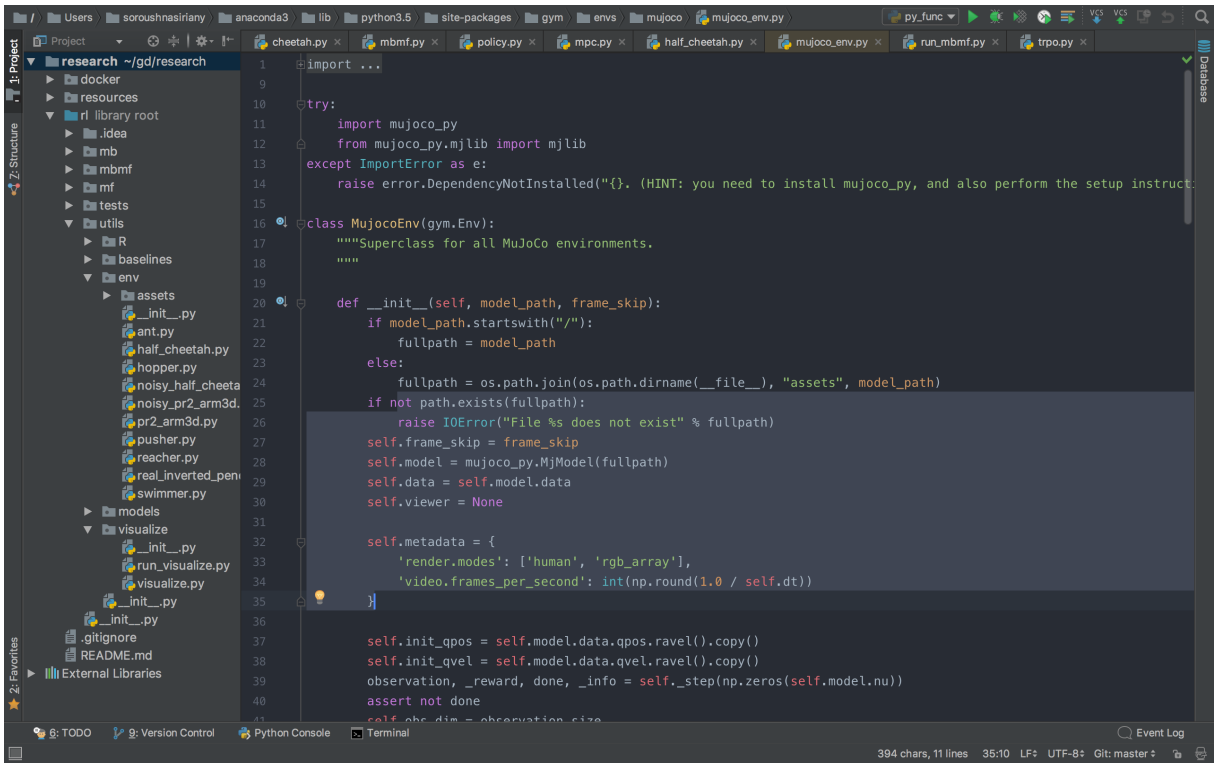

Figura 1: Interfaccia grafica di PyCharm

[6]Riferirsi alla voce 16 in §1.5.2
[7]Riferirsi alla voce 23 in §1.5.2

3 Processi organizzativi

3.1 Gestione del progetto

3.1.1 Scopo

L'attività di gestione del progetto consiste nel delineare:

- I PROCESSI$_G$ di progetto
- Le RISORSE$_G$ a essi necessarie
- I costi per la loro esecuzione
- I TASK$_G$ di risorsa umana
- La verifica delle attività di processo

3.1.2 Pianificazione della qualità

Per la ricerca della qualità nei prodotti è necessario che ne vengano misurate diverse caratteristiche associate. Per ogni caratteristica devono essere fissati un livello minimo e un livello ottimale che questa deve avere. In tale sezione, viene descritto il modo di utilizzo sia delle metriche che degli strumenti utili ad ottenere qualità nei processi, nei documenti e nei prodotti. Gli strumenti utilizzati possono essere software, descritti nelle *NormeDiProgetto v4.0.0*$_D$, o standard di qualità, descritti nel *PianoDiQualifica v4.0.0*$_D$.
Gli obiettivi che ci si pone di raggiungere sono descritti nel *PianoDiQualifica v4.0.0*$_D$.

3.1.2.1 Classificazione dei processi

I processi analizzati nel *PianoDiQualifica v4.0.0*$_D$ sono classificati nel seguente modo:

`PROC[ID] [Nome]`

- **ID**: un numero incrementale per indicare in modo univoco il processo.
- **Nome**: una breve frase per indicare la funzione del processo.

3.1.2.2 Classificazione degli obiettivi

Gli obiettivi per la qualità (Q) concordati dall'Amministratore e dal Verificatore, descritti nel *PianoDiQualifica v4.0.0*$_D$, avranno il seguente codice identificativo:

`Q[Tipo][ID] [Nome]`

- **Tipo**: indica il la tipologia dell'oggetto di cui viene valutata la qualità, e questa può essere:
  - `PR`: processi.
  - `PD`: prodotti di documentazione.
  - `PS`: prodotti software.
- **ID**: ogni tipo di obiettivo possiede una lista ordinata attraverso un numero incrementale di tre cifre.
- **Nome**: offre un'informazione più chiara dell'obiettivo attraverso una breve frase.

Ad esempio:

- QPD001 Leggibilità del testo

3.1.2.3 Classificazione delle metriche

Ogni obiettivo della qualità deve, per quanto possibile, essere collegato ad una metrica, anch'essa scelta dall'Amministratore e dal Verificatore. Questo per valutare quantitativamente il raggiungimento o meno degli obiettivi stabiliti.
Le metriche (M) verranno classificate nel seguente modo:

`M[Tipo][ID] [Nome]`

- **Tipo**: indica la tipologia di quello su cui viene applicata la metrica, può essere:
  - `PR`: processi.
  - `PD`: prodotti di documentazione.
  - `PS`: prodotti software.
- **ID**: ogni tipo di obiettivo possiede una lista ordinata attraverso un numero incrementale di tre cifre.
- **Nome**: offre un'informazione più chiara dell'obiettivo attraverso una breve frase.

Ad esempio:

- MPD001 Indice Gulpease

Quando sarà possibile, la metrica e l'obiettivo di qualità collegati fra loro avranno lo stesso Tipo e ID.
Ad esempio:

Obiettivo	Metrica	Valore desiderato
QPD001 Leggibilità del testo	MPD001 Indice Gulpease	50-60
Descrizione: quanto il testo è leggibile e comprensibile a livello sintattico lo stabilisce l'INDICE DI GULPEASE$_G$ e una verifica da parte del Verificatore.		

Tabella 1: Metrica dell'Indice di Gulpease

3.1.3 Pianificazione di attività

3.1.3.1 Scopo

Responsabile e Amministratore insieme si occupano di redigere il *PianoDiProgetto v4.0.0*$_D$ che tratta generalmente:

- **Modello di sviluppo**: per la decisione del modello di sviluppo che si intende adottare.
- **Analisi dei rischi**: in cui si identificano i vari possibili rischi e le strategie per evitarli o mitigarli.
- **Partizione del carico di lavoro**: che si occupa di delineare un preciso calendario di avanzamento e la rotazione dei ruoli.
- **Prospetto economico**: per la stima dei costi relativa alle risorse umane.

3.1.3.2 Obiettivo

L'obiettivo che ci poniamo di raggiungere tramite il *PianoDiProgetto v4.0.0*$_D$ è organizzare il lavoro con EFFICENZA$_G$ ed EFFICACIA$_G$, prevedendo anticipatamente le varie attività da svolgere e determinando dei precisi periodi di tempo da dedicare ad ognuna di esse.
Più nello specifico, il *PianoDiProgetto v4.0.0*$_D$ contiene:

- Introduzione e scopo
- Ciclo di vita con esposizione del modello di sviluppo
- Analisi dei rischi con annessa valutazione e classificazione
- PIANIFICAZIONE$_G$ delle attività in una visione temporale
- Prospetto economico del personale durante le attività
- PREVENTIVO$_G$ di ore e costi
- ORGANIGRAMMA$_G$ comprendente i componenti del team di sviluppo
- Attualizzazione dei rischi verificatisi nel corso del progetto

3.1.3.3 Ordine di esecuzione

L'insieme di azioni che indicano in quale modo procedere sono riportate di seguito, in ordine:

1. Individuazione di tutte le attività da svolgere basandosi sulle varie revisioni da affrontare
2. Riconoscimento dei vari problemi in cui è possibile incorrere nel corso del progetto, con conseguente analisi approfondita di ognuno essi. Questo comporta l'identificazione di:
   - Probabilità di verificarsi
   - Gravità della situazione
   - Strategie da seguire per l'accertamento
   - Contromisure da adottare per la risoluzione
3. Report dei problemi incorsi durante lo svolgersi delle attività del progetto, con soluzione adottata
4. Ordinamento delle attività tramite diagrammi di GANTT$_G$ creati secondo alcuni fattori importanti da tenere in considerazione, quali:
   - Dipendenze tra attività
   - Sequenzialità
   - Possibilità di parallelismo
   - Andamento del progresso
   - Margine di SLACK TIME$_G$ per consentire l'ammortizzamento di possibili rallentamenti
5. Stima delle risorse secondo la METRICA$_G$ tempo/persona e le MILESTONE$_G$ nel tempo, pianificando all'indietro
6. Assegnazione delle risorse umane alle attività secondo una precisa rotazione dei ruoli

3.1.3.4 Metriche di pianificazione

- MPR001 Varianza della pianificazione
- MPR002 Varianza dei costi

La denominazione delle metriche è descritta a §3.1.2.3.

3.1.3.4.1 MPR001 Varianza della pianificazione

Nel documento *PianoDiProgetto v4.0.0*$_D$, sono stabilite le BASELINE$_G$ e le scadenze di consegna dei vari prodotti. Nonostante il tempo di slack che ogni fase possiede, è possibile che delle date non vengano rispettate causa incidenti di vario tipo.

$$\frac{\sum_{i=1}^{n} |x_i|}{n} \qquad x_i = \text{numero di ore di variazione rispetto al preventivo per l'i-esimo ruolo.}$$

Metrica: viene indicato il numero di ore di variazione presenti nel momento della verifica rispetto al preventivo effettuato per un preciso periodo. Questo viene svolto per ogni ruolo del progetto ed infine fatta la somma delle variazioni in valore assoluto. Per misurare il numero di ore di varianza viene usato lo strumento Toggl in §3.1.6.3 per misurare le ore di lavoro effettive. I valori vengono infine confrontati col preventivo.

3.1.3.4.2 MPR002 Varianza dei costi

All'interno del *PianoDiProgetto v4.0.0*$_D$ è indicato il costo approssimativo del progetto. In corso d'opera possono presentarsi dei problemi che richiedono un'aggiunta di costo in termini di $\frac{\text{tempo}}{\text{persona}}$. Lo scopo del preventivo è infatti fare una stima non definitiva dei costi. Per verificare se il preventivo è rispettato faremo periodicamente un consuntivo di periodo.
Metrica: viene misurata la differenza conteggiata in € tra il costo finale e il preventivo. Viene seguito tale schema per la tariffa oraria dei vari ruoli del team di sviluppo:

Ruolo	Costo orario
Responsabile	€ 30
Amministratore	€ 20
Analista	€ 25
Progettista	€ 22
Programmatore	€ 15
Verificatore	€ 15

Tabella 2: Costo orario per ruolo

3.1.3.5 Classificazione dei rischi

Le tipologie in cui suddividere i rischi sono state prese dal libro Software Engineering[8] e possono essere di tipo:

- **Organizzativo**: dovuto alla gestione di persone che hanno diverse responsabilità all'interno del progetto.

[8] Riferirsi alla voce 19 in §1.5.2

- **Personale**: riguarda le conoscenze, i tempi e la FORMAZIONE$_G$ personale.
- **Requisiti**: ha a che fare con il numero di requisiti che può variare nel corso dello sviluppo del progetto.
- **Strumentale**: per l'utilizzo e la performance degli strumenti hardware.
- **Tecnologico**: per problemi riguardanti l'utilizzo e le funzionalità degli strumenti software.

A ciascun rischio viene assegnato un codice identificativo in modo da essere facilmente riconoscibile e per comprenderne le generalità (classe, probabilità e severità), per poi non doverle cercare nelle tabelle che comprendono tutti i rischi del progetto analizzati.
Questo codice è composto da:

`[Tipologia][ID]-[Gravità][Probabilità][Classe]`

Nel caso dell'attualizzazione dei rischi, vengono aggiustati i valori di gravità, probabilità, classe e al codice viene aggiunta in coda la data in cui si è verificato il problema, in modo da tenere una traccia cronologica.
Il codice in tal caso diventa:

`[Tipologia][ID]-[Gravità][Probabilità][Classe]:[Data]`

Nel caso i rischi non si siano verificati ma ne venga riconsiderata la probabilità, allora vengono posti in elenco in seguito ai rischi effettivamente riscontrati, con il vecchio codice per tipologia e id e con i valori aggiornati per gravità, probabilità e classe.
I valori che possono assumere sono:

- **Tipologia**:
  - **O**: organizzativo.
  - **P**: personale.
  - **R**: requisiti.
  - **S**: strumentale.
  - **T**: tecnologico.
- **ID**: numero progressivo di tre cifre (001 - 999).
- **Gravità**:
  - **0**: accettabile.
  - **1**: tollerabile.
  - **2**: inaccettabile.
- **Probabilità**:
  - **0**: bassa.
  - **1**: media.
  - **2**: alta.
- **Classe**: ci si riferisce ai livelli di rischio.
  - **0**: basso.
  - **1**: medio.

  - **2**: alto.

- **Data**: data in cui si è verificato il problema.

Ad esempio con P001-021 si può intuitivamente capire che si tratta del primo rischio relativo al personale, di gravità accettabile, probabilità alta e un valore di classe medio.
Invece P002-122:2019-01-13 è il secondo rischio attualizzato relativo al personale, di gravità tollerabile, probabilità alta, valore di classe alto e si è verificato in data 2019-01-13.

3.1.3.5.1 MPR008 Rischi non previsti avvenuti

La denominazione delle metriche è descritta a §3.1.2.3.
Nell'analisi dei rischi presente nel *PianoDiProgetto v4.0.0*$_{D}$, sono presenti i rischi ritenuti possibili per i quali è proposta una soluzione. Possono presentarsi anche rischi non previsti in tale analisi. Questi devono essere il meno possibili (nulli) perché la loro soluzione sarà decisa al momento causando ritardi all'interno del progetto.
Metrica: numero di rischi non previsti avvenuti nel corso dell'intero progetto.

3.1.3.6 Ruoli di progetto

I ruoli[9] adoperati per lo sviluppo del progetto sono:

- Analista
- Progettista
- Responsabile
- Amministratore
- Programmatore
- Verificatore

3.1.4 Monitoraggio del progetto

3.1.4.1 Monitoraggio dell'esecuzione dei processi

Ogni processo verrà controllato periodicamente durante tutta la sua esecuzione in modo da non intralciare il WAY OF WORKING$_{G}$, mediante misurazioni associate a metriche descritte dai processi di verifica, controllate tramite l'utilizzo di strumenti automatizzati. Le metriche adottate sono, per il processo considerato, indicatori di efficacia dei prodotti rispetto ai requisiti di funzionalità e qualità, stabiliti nel *PianoDiQualifica v4.0.0*$_{D}$ e nell'*AnalisiDeiRequisiti v3.0.0*$_{D}$.
Risultano validi indicatori per la valutazione di:

- Aderenza al way of working
- Stato di avanzamento del processo rispetto alla pianificazione
- Identificazione dei problemi
- Eventuali ripianificazioni

3.1.4.2 Procedure di comunicazione

Per la coordinazione del team e le comunicazioni con il cliente sullo stato del progetto, abbiamo stabilito le seguenti norme:

- Per le comunicazioni interne utilizzeremo gli strumenti segnalati in §3.1.6, in particolare con l'utilizzo di SLACK$_{G}$.

[9] Riferirsi alla voce 4 in §1.5.2.

- All'occorrenza, fisseremo riunioni per le questioni più importanti, concordando:
  - Data, ora e luogo
  - Un ordine del giorno da discutere
  - La persona incaricata ad appuntare il contenuto della riunione per poter poi redigere formalmente il verbale dell'incontro
- Per le comunicazioni esterne verrà utilizzata prevalentemente la comunicazione via Email e in caso di necessità si concorderà con l'azienda per riunioni via Hangouts o in un luogo prestabilito come quanto segnalato in §2.1.6.

3.1.4.3 Gestione dei problemi emersi

Problemi emersi durante l'esecuzione dei processi verranno segnalati tramite TICKET$_G$, utilizzando il sistema integrato di GITHUB$_G$ come descritto in dettaglio in §3.1.6.2. Questo ci permetterà di tracciare i problemi insorti durante le attività e di pianificare la risoluzione degli stessi.
Per segnalazioni minori, verranno spesso utilizzati dei tag commentati nei SORGENTI$_G$. Saranno nella forma:

```
[TAG]: [Descrizione]
```

I tipi di tag che utilizzeremo sono:

- **TODO**: per segnalare la presenza di un lavoro da fare o lasciato a metà.
- **FIXME**: per segnalare la presenza di una correzione da effettuare su del lavoro già svolto.
- **NOTE**: per aggiugere una nota su cui porre particolare attenzione.

3.1.4.4 Monitoraggio delle ore di orologio

È necessario distinguere le ore di orologio dalle ore di calendario. Le ore di orologio sono ore effettive di lavoro, in cui non c'è alcuna perdita di tempo. Le ore di calendario sono ore in cui non si conta solamente il lavoro effettivo ma anche tutte le distrazioni. A noi interessa monitorare le ore di orologio, e le ore rendicontate pertanto sono da considerare ore di orologio.

3.1.5 Chiusura dei processi

3.1.5.1 Archiviazione dei prodotti

I prodotti che, una volta terminate le attività e i processi per completarli, soddisfano le aspettative in termini di qualità, verranno archiviati in apposite REPOSITORY$_G$. Alla versione *2.0.0* del presente documento, saranno archiviati i file LaTeX per la generazione dei relativi PDF, i sorgenti Python per il Proof of Concept e gli eventuali file JSON$_G$.

3.1.5.2 Archiviazione delle misurazioni

Oltre ai prodotti archivieremo le metriche, definite e classificate nel presente documento, con i limiti di accettazione riportati nel *PianoDiQualifica v4.0.0*$_D$. Il calcolo delle metriche verrà effettuato su ciascun prodotto nel momento del suo inserimento nella repository, ove possibile; tali risultati o anomalie verranno opportunamente archiviati.

3.1.6 Strumenti organizzativi

3.1.6.1 Slack

Slack è uno strumento di collaborazione nato appositamente per coordinare il lavoro tra i team, permettendo la comunicazione in tempo reale e mettendo a disposizione molte altre utilità indispensabili. Tra queste, c'è la possibilità di dividere il WORKSPACE$_G$ in vari canali specifici, marcare parole o frasi importanti con diversi stili (grassetto, corsivo, codice, barrato), fissare i messaggi importanti, aprire una discussione sotto ogni messaggio per non creare troppi messaggi sul canale, ecc ...
Il workspace usato è stato suddiviso in diversi canali in base alle esigenze del periodo di lavoro. Nel periodo relativo alla stesura della versione attuale del documento, i vari canali sono:

- # **general**: canale in cui verranno discusse tematiche generali riguardanti il progetto e le sue attività.
- # **Analisi dei Requisiti**: per discussioni inerenti alla stesura del documento *AnalisiDeiRequisiti v3.0.0*$_D$. Nello specifico verranno discussi i requisiti e i casi d'uso.
- # **Piano di Progetto**: per discussioni inerenti la pianificazione, la suddivisione del lavoro, la valutazione dei rischi, la consuntivazione e altre attività volte alla redazione del *PianoDiProgetto v4.0.0*$_D$.
- # **Piano di Qualifica**: per discussioni inerenti principalmente la qualità di processo e di prodotto, riportate nel *PianoDiQualifica v4.0.0*$_D$.
- # **Norme di Progetto**: per discussioni riguardanti le norme che il team di sviluppo adotterà e che ogni membro sarà tenuto a rispettare. Esse saranno riportate nelle Norme di Progetto.
- # **Studio di Fattibilità**: canale in cui verrà discussa la fattibilità di ogni capitolato proposto, convergenti nello *StudioDiFattibilità v1.0.0*$_D$.
- # **Manuale Sviluppatore**: canale in cui vengono discussi argomenti inerenti la stesura del *ManualeSviluppatore v2.0.0*$_D$.
- # **Manuale Utente**: canale in cui vengono discussi argomenti inerenti la stesura del *ManualeUtente v2.0.0*$_D$.
- # **code**: per discussioni inerenti alla realizzazione del codice del prodotto.
- # **git**: canale in cui un BOT$_G$ correttamente configurato manderà una notifica ogni volta che verrà effettuato un COMMIT$_G$, oppure alla chiusura o apertura di una ISSUE$_G$ nella repository principale.
- # **latex**: per discussioni riguardanti gli aspetti tecnici di LaTeX.
- # **random**: per discussioni off-topic del team.

3.1.6.2 GitHub

GitHub fornisce un ISSUE TRACKING SYSTEM$_G$ per ogni repository che ospita. Questo strumento offre più funzionalità di quanto potrebbe sembrare, poiché permette anche di trattare le issue come delle task, marcandole con una opportuna label. Infatti, questo sistema offre le seguenti funzionalità:

- Aprire una issue, attribuendone un titolo e una descrizione più dettagliata. Ogni issue è legata a un ID generato automaticamente in maniera incrementale (e.g. `#32`), al quale sarà possibile fare riferimento nei messaggi di commit.
- Creare delle milestone, e aggregare ad esse le issue.
- Assegnare delle label alle issue, per marcarne la natura. Ad esempio, una issue marcata come BUG$_G$ sarà un problema da risolvere, una issue marcata come ENHANCEMENT$_G$ sarà un miglioramento o una task. È possibile inoltre creare delle label personalizzate.
- Assegnare una issue a sé stesso o a un collaboratore.
- Supporto di MARKDOWN$_G$ esteso, che permette di formattare le pagine delle issue in modo estremamente funzionale.

3.1.6.3 Toggl

Come strumento di monitoraggio delle ore di orologio usiamo Toggl[10], un software che è disponibile come applicazione web, applicazione desktop e applicazione mobile.
Tramite questo strumento è possibile cronometrare il tempo che si spende nello svolgere una determinata task, ed è possibile organizzarsi in un team in modo che le ore dedicate a un progetto siano condivise.
Per maggiori informazioni, riferirsi alla fonte ufficiale.

3.2 Formazione

La formazione avviene tramite studio autonomo dei FRAMEWORK$_G$ menzionati da Imola Informatica durante la presentazione del capitolato e incontri interni.

3.2.1 Piano di formazione

Il piano di formazione prevede lo studio di:

- Tecnologie e metodologie per la realizzazione del Proof of Concept[11]:
  - Python 3
  - Apache Kafka
  - Redmine
  - GitLab
  - Telegram API per l'invio di messaggi su Telegram tramite un bot dedicato
  - Protocollo SMTP per l'invio di Email tramite uno script Python
  - Docker
  - Rancher
  - Flask
  - API Rest
- Possibilità d'integrazione tra le tecnologie sopra elencate

[10] Riferirsi alla voce 22 in 1.5.2 per maggiori informazioni
[11] Vedere §1.5.2 per i link alla documentazione.

4 Processi di supporto

4.1 Documentazione

4.1.1 Implementazione

4.1.1.1 Template

Prima di iniziare a redigere i documenti, abbiamo creato un TEMPLATE$_G$ per LaTeX (§4.1.4.2.1) contenente tutte le impostazioni grafiche condivise tra questi, per sfruttare il riutilizzo del codice e semplificare enormemente la manutenzione dei sorgenti.
Nello specifico, è presente un file per ognuna delle seguenti utilità:

- LAYOUT$_G$ delle pagine
- MACRO$_G$ personalizzate volte a semplificare l'utilizzo di strutture o comandi ricorrenti
- Codice per la generazione della prima pagina (struttura definita in §4.1.2.1)
- Diario delle modifiche

4.1.1.2 Ciclo di vita dei documenti

Durante il suo ciclo di vita, ogni documento potrà trovarsi in una delle seguenti fasi:

- **Redazione**: fase che inizia con la creazione del documento e dura fino alla sua ultima approvazione. Il Responsabile assegna ai REDATTORI$_G$ le varie sezioni di ogni documento da redigere, i quali aggiorneranno la versione nel diario delle modifiche come normato in §4.2.3.1.
- **Verifica**: il documento entra in questa fase nel momento in cui i Redattori hanno terminato la stesura del lavoro loro assegnato, segnalandolo al Responsabile, che a sua volta assegnerà ai Verificatori la verifica della qualità del prodotto, secondo quanto riportato nelle norme di verifica. Essi potranno approvare il documento oppure notificare il Responsabile su eventuali errori o incongruenze emerse durante la fase di verifica, che provvederà a riassegnare il lavoro.
- **Approvazione**: fase che inizia dall'accettazione del documento da parte dei Verificatori nella fase di verifica. Spetta al Responsabile l'approvazione ufficiale del documento, seguita dal rilascio di una MAJOR RELEASE$_G$.

4.1.2 Struttura

4.1.2.1 Frontespizio

La prima pagina di ogni documento, sarà caratterizzata da:

- Logo e nome del gruppo
- Titolo del documento
- Informazioni sul documento:
  - Versione documento
  - Data di creazione e ultima modifica
  - Nominativo dei Redattori
  - Nominativo dei Verificatori

  - Nominativo del Responsabile
  - Destinazione d'uso
  - Destinatari del documento
  - Contatto del gruppo

- Breve descrizione del documento

4.1.2.2 Storico delle versioni

La pagina che segue il frontespizio contiene lo storico delle versioni del documento, in cui ogni aggiunta o modifica significativa ha comportato un incremento di versione. Ogni riga contiene, a partire da sinistra:

- Il numero della versione nel formato espresso in §4.2.3.1
- Una breve descrizione delle modifiche apportate
- Il ruolo dell'autore che ha apportato la modifica
- Il nominativo dell'autore
- La data di modifica

La chiave primaria della tabella è il numero di versione ordinata in senso decrescente, in modo che la versione più vecchia sia l'ultima riga della tabella.

4.1.2.3 Indice

In ogni documento, esclusi i verbali, è presente un indice contenente tutte le sezioni, sottosezioni e paragrafi. I numeri di sezioni, sottosezioni, e paragrafi sottostanti saranno separati da un punto (e.g. 1.4.1).
Saranno eventualmente presenti un indice delle figure e un indice delle tabelle, assenti in caso non ci siano tabelle o figure nel documento.
I valori degli indici partono da 1.

4.1.2.4 Contenuto

La struttura di ogni pagina presenta:

- Intestazione con:
  - A sinistra, logo di AlphaSix
  - A destra, nome del capitolato e documento corrente
- Piè di pagina con:
  - A sinistra, nome e mail di riferimento del gruppo
  - A destra, numero della pagina corrente

4.1.3 Design

4.1.3.1 Norme tipografiche

Le norme tipografiche qui di seguito elencate sono state decise in modo che ognuno di noi concorra a mantenere una forma coerente e univoca per tutti i documenti redatti.

4.1.3.1.1 Stile redazionale

Lo stile redazionale dei vari documenti sarà prevalentemente personale, in prima persona, per rendere chiaro il soggetto delle frasi.

4.1.3.1.2 Stile del testo

- **Corsivo**: solo per i nomi dei documenti citati.
- **Maiuscolo**: la prima lettera per
  - Tutte le parole appartenenti ai nomi dei documenti tranne gli articoli
  - Nomi dei ruoli
  - Prima parola degli elenchi puntati

4.1.3.1.3 Elenchi puntati

- **Simboli di livello**: un pallino nero per il primo livello, un trattino per il secondo livello.
- **Punteggiatura**: nessuna punteggiatura alla fine di una frase, tranne nel caso in cui sia presente una descrizione. In quel caso la descrizione è preceduta dai due punti ":" e termina con un punto ".".
- **Grassetto**: solo se è presente una descrizione, allora sono in grassetto tutte le parole prima dei due punti ":".

4.1.3.1.4 Altri formati testuali comuni

- **Orari**: `HH:MM` secondo la norma ISO 8601[12] nel formato 24 ore dove:
  - `HH` indica le ore, da 00 a 23
  - `MM` i minuti, da 00 a 59
- **Date**: `YYYY-MM-DD` formato adottato in Europa dove:
  - `YYYY` l'anno
  - `MM` il mese, da 01 a 12
  - `DD` il numero del giorno, da 01 a 31
- **Nota a piè di pagina**: serve ad inserire elementi aggiuntivi, come osservazioni o riferimenti a parti interne al documento, utili alla comprensione del testo, ma se inseriti all'interno del discorso, ne interromperebbero la lettura, rendendola meno scorrevole.

4.1.3.2 Elementi grafici

4.1.3.2.1 Figure

Ogni immagine inserita nei documenti deve sempre essere centrata rispetto al foglio e adeguatamente separata dal testo. Deve inoltre essere accompagnata da una breve CAPTION$_G$ che permetta al lettore di capire esattamente che cosa sta guardando.
È presente nell'indice l'elenco delle figure che raccoglie la lista di tutte le immagini presenti.

[12] Riferirsi alla voce 1 in §1.5.2

4.1.3.2.2 Tabelle

Come per le figure, ogni tabella sarà accompagnata da una caption e sarà della dimensione del testo, o se più piccola, centrata. Tutte le tabelle saranno raccolte nell'elenco delle tabelle.
Saranno presenti due tipologie di tabelle:

- **Semplici**: tabelle standard senza uno stile particolare, in cui le celle sono separate da bordi neri (evitare, ove non risulta necessario, le righe verticali).
- **Complesse**: tabelle con un'alternanza di colori tra le righe delle celle (grigio e bianco) e senza bordi verticali. Le celle sono separate orizzontalmente da una corretta spaziatura e allineamento e verticalmente dall'alternanza dei due colori. La riga dell'HEADER$_G$ può essere bianca o di un grigio più scuro in base al contesto, con il testo che può essere in grassetto.

4.1.3.3 Approvazione

Ogni documento, nel momento in cui viene ritenuto pronto, deve essere revisionato dal Responsabile affinché ne venga controllata l'adeguatezza e approvato il rilascio.

4.1.4 Produzione

4.1.4.1 Suddivisione dei documenti

4.1.4.1.1 Documenti interni

Le *NormeDiProgetto v4.0.0*$_D$ e lo *StudioDiFattibilità v1.0.0*$_D$ sono documenti interni, consultabili solo dal team e dal committente, per motivi didattici.

4.1.4.1.2 Documenti esterni

Sono considerati esterni, invece, il *PianoDiProgetto v4.0.0*$_D$, il *PianoDiQualifica v4.0.0*$_D$, il *Glossario v4.0.0*$_D$ e l'*AnalisiDeiRequisiti v3.0.0*$_D$ che, al contrario dei precedenti, vengono consegnati al proponente.

4.1.4.1.3 Verbali

Redigiamo questi documenti successivamente alle riunioni tenute o in caso di incontri con STAKEHOLDER$_G$ esterni, per esempio con Imola Informatica. Una singola persona ha il compito di stendere la relazione relativa al verbale, presentando le seguenti sezioni:

- **Informazioni incontro**: lista delle informazioni principali riguardanti la riunione quali luogo, data, orario, ordine del giorno, ecc.
- **Argomenti**: lista dei principali argomenti trattati, con breve descrizione.
- **Tracciamento delle decisioni**: contiene il resoconto delle decisioni prese nella riunione, tracciate con un codice nel formato `yyyy-mm-dd-n`, dove `yyyy-mm-dd` è la data in cui si è tenuta la riunione e `n` è il numero della decisione presa, che parte da 1.

4.1.4.2 Strumenti di supporto

4.1.4.2.1 LaTeX

Per la stesura della documentazione abbiamo deciso di utilizzare il linguaggio di MARKUP$_G$ LaTeX perché presenta molti vantaggi, tra i quali:

- Supporta nativamente il VERSIONAMENTO$_G$, essendo un linguaggio compilato

- Supporta la modularità, rendendo più facile organizzare un documento dividendone logicamente i vari moduli
- Permette il riutilizzo del codice tramite l'uso di macro già pronte o personalizzate, oppure includendo lo stesso SORGENTE$_G$ in punti diversi
- Gestisce automaticamente indici e riferimenti

4.1.4.2.2 Google Drive

Per la condivisione di file informali, tabelle informative e diagrammi dei casi d'uso e delle classi (con draw.io), utilizziamo lo strumento di cloud Google Drive. Esso ci permette di tenere file informativi sempre aggiornati e condivisi tra tutti i membri del team, raggiungibili in qualunque momento anche da smartphone. Altre informazioni sono reperibili tramite i link informativi in §1.5.2.

4.1.4.2.3 TexStudio/Visual Studio Code

TexStudio e Visual Studio Code sono i due ambienti di sviluppo scelti per stilare la documentazione. TexStudio è un IDE$_G$ nativo per l'utilizzo di LaTeX. Visual Studio Code è un editor intelligente moderno (alla pari di un IDE$_G$) che, tramite estensioni, permette il supporto di praticamente ogni linguaggio. Entrambi permettono una rapida compilazione e un'istantanea visualizzazione dell'anteprima del PDF prodotto, oltre agli altri vantaggi che ogni IDE offre, tra cui: suggerimenti e completamenti automatici delle parole chiave, ricerca intelligente (eventualmente tramite REGEXP$_G$) controllo ortografico della lingua italiana o inglese e così via.
Più informazioni sono reperibili sui rispettivi siti ufficiali, i cui link sono presenti in §1.5.2.

4.1.4.2.4 GanttProject

GanttProject è un programma gratuito dedicato alla formazione dei diagrammi di Gantt. Permette di creare task e milestone, organizzare le task in lavoro strutturato a interruzioni, disegnare i vincoli di dipendenza tra di esse e molte altre utilità, generando automaticamente il relativo diagramma. Per maggiori informazioni, si rimanda alla fonte ufficiale (consultare §1.5.2).

4.1.4.2.5 Draw.io

Draw.io è un'applicazione web in grado di creare diagrammi UML, di Entità-Relazionale, di flusso e molto altro. Il motivo che ci ha portato a scegliere questo strumento è la sua perfetta integrazione con Google Drive$_G$, oltre al suo alto livello di intuitività. Questo permette la condivisione dei diagrammi creati tra tutti i collaboratori in ogni momento e in modo automatico. Per maggiori informazioni, visualizzare la fonte ufficiale (§1.5.2).

4.1.4.2.6 Indice di Gulpease

Si tratta di un indice di leggibilità dei documenti in lingua italiana. A differenza degli indici per le altre lingue, questo si basa sulla lunghezza delle parole in lettere e non in sillabe. In base al valore indicato si può capire che livello di istruzione deve possedere una persona per comprendere il documento.
Per automatizzare il calcolo di questo indice è stato prodotto uno SCRIPT$_G$ che viene eseguito ad ogni COMMIT$_G$ nella REPOSITORY$_G$, il quale riporta in un file i risultati ottenuti. Tale indice è descritto anche in §4.4.6.1.1.

4.1.4.2.7 Controllo ortografico

Per il controllo ortografico utilizziamo:

- In fase di redazione dei documenti, il controllo ortografico integrato degli IDE utilizzati
- In fase di verifica, lo strumento GNU Aspell, uno strumento open source per il controllo ortografico che permette tramite interfaccia interattiva di cambiare parole non riconosciute dal dizionario e scegliere tra più suggerimenti. Maggiori informazioni alla fonte ufficiale, reperibile in §1.5.2. La correttezza ortografica possiede una metrica presente in §4.4.6.1.2.

4.1.4.2.8 Glossarizzazione dei ternimi

Come spiegato in §1.1, i termini con un particolare significato vengono riportati e definiti all'interno di *Glossario v4.0.0*$_{\text{D}}$. Per automatizzare questo processo è stato creato uno script che valuta le voci all'interno di *Glossario v4.0.0*$_{\text{D}}$ e ed inserisce i riferimenti a queste voci alla prima occorrenza di ogni parola in tutti i documenti prodotti.

4.1.4.3 Controllo

I controlli devono essere stabiliti in concordanza con il processo di configurazione alla sezione §4.2.

4.1.4.3.1 Controllo sui processi

Per il controllo della qualità sui processi abbiamo scelto di seguire in ordine tre attività principali:

1. **Misurazione**: misurazione del processo al fine di accertarne la conformità con gli obiettivi.
2. **Analisi**: precisa analisi dei risultati e problemi rilevati nel processo.
3. **Rettifica**: modifiche correttive volte alla risoluzione dei problemi che contribuiranno alla nuova misurazione del prossimo ciclo di procedure.

Seguendo le attività passo per passo e ripetendo la sequenza più volte è possibile aiutarci a diminuire i costi di sviluppo in maniera considerevolmente vantaggiosa.

4.1.4.3.2 Controllo sui prodotti

Per le procedure di misurazione del prodotto effettuiamo:

- Uso delle metriche e delle procedure delineate nel documento corrente e dei test descritto nel *PianoDiQualifica v4.0.0*$_{\text{D}}$
- Confronto di tali risultati rispetto agli obiettivi prefissati [13]
- Valutazioni per il miglioramento [14]

A questo scopo utilizziamo strumenti di analisi statica e dinamica presenti nella sezione §4.4.

4.1.5 Mantenimento dei documenti

I documenti sotto l'attività di gestione delle configurazioni devono essere gestiti secondo il processo di configurazione alla sezione §4.2.

[13] Consultabili nelle appendici del *PianoDiQualifica v4.0.0*$_{\text{D}}$
[14] Consultabili anch'esse nelle appendici del *PianoDiQualifica v4.0.0*$_{\text{D}}$

4.1.5.1 Continuous Integration

Per la produzione dei documenti adotteremo la pratica della CONTINUOUS INTEGRATION$_G$. Per questa attività il principio sarà limitato a sincronizzarsi il prima possibile con la repository remota (non essendoci una vera e propria build o dei test da effettuare), sia per quanto riguarda il FETCH$_G$ che per il PUSH$_G$. Questo serve a rendere più remota possibile la probabilità di incappare nell'INTEGRATION HELL$_G$.

4.1.5.2 Nomenclatura

4.1.5.2.1 Verbali

I Verbali possono essere interni oppure esterni, nel caso in cui il team incontri gli esponenti di Imola Informatica o il committente. Il nominativo del file in cui sono formalizzati è il seguente:

- `VI_yyyy-mm-dd.pdf` per i verbali interni
- `VE_yyyy-mm-dd.pdf` per i verbali esterni

dove yyyy-mm-dd è la data in cui sono stati tenuti, nel formato descritto nel paragrafo §4.1.3.1.4. Al termine del documento è presente una tabella contenente il tracciamento delle decisioni prese durante l'incontro descritto nel verbale. Tali decisioni possiedono un identificativo definito:

`[Nominativo file].[Numero progressivo]`

Ad esempio, VE_2019-03-23.1.

4.1.5.2.2 Documenti vari

Saranno presenti due tipologie di file: file interni ad AlphaSix e file esterni.

- La prima categoria include moduli di LaTeX contenenti le varie sezioni e che non verranno mai esposti esternamente. Questi file verranno denominati usando la convenzione `snake_case.tex`, dove snake_case è il nome della sezione o modulo
- La seconda categoria include i file `.tex` principali che produrranno i PDF da consegnare al committente. Essi verranno denominati con la convenzione `CamelCase vX.Y.Z.pdf`, dove CamelCase sarà il nome del documento generico mentre `vX.Y.Z` sarà la versione che identifica univocamente il documento come descritto in §4.2.3.1

4.1.6 Mantenimento del codice sorgente

4.1.6.1 Continuous Integration

Anche per quanto riguarda il processo di codifica verrà adottato il principio di Continuous Integration.

4.1.6.1.1 Jenkins

Per quanto concerne il codice sorgente usiamo il tool Jenkins per tenere sotto controllo la CI pipeline, definita come una deployment pipeline priva degli ultimi stadi relativi al deploy.
Jenkins ci consente di amministrare in modo immediato tutti i passi dello sviluppo, nell'ordine:

- **SCM checkout**: sincronizzazione del codice con la repository remota.
- **Build**: build del codice su docker container.

- **Unit Test**: test di unità tramite Pytest[15].
- **Integration Test**: test di integrazione tramite Pytest.
- **System Test**: test per verificare il comportamento dell'intero sistema tramite TOX$_G$.

Configuriamo Jenkins tramite Jenkinsfile e la CI pipeline viene eseguita in automatico, in modo da avere sempre sotto controllo la correttezza del codice sulla repository.

4.1.6.2 Docker

Per rendere il sistema Butterfly completamente isolato dal sistema circostante, creiamo dei container specifici per ogni componente. Per fare ciò, utilizziamo DOCKER$_G$[16], un tool che rende questo processo estremamente semplice. Sarà presente un'immagine Docker per ognuna delle seguenti componenti, dalle quali sarà possibile costruire un container:

- Producer Redmine
- Producer GitLab
- Gestore Personale
- Consumer Email
- Consumer Telegram

Attraverso l'uso di uno specifico `docker-compose.yml` e il comando `docker-compose up` il processo di build e avvio dell'applicativo sarà semplice e immediato.

4.1.6.2.1 DockerHub

Per evitare di effettuare una build manuale dell'immagine di Docker contenente una specifica componente del software, abbiamo deciso di utilizzare DockerHub collegato alla nostra repository in GitHub. In questo modo ogni modifica effettuata nella repository, attraverso push sui branch master e develop, scatta una build automatica. Questa prevede quindi la creazione di dieci immagini salvate su cinque repository su DockerHub in un account denominato 'alphasix'. Le immagini sono classificate in questo modo nome_account/nome_repository:tag:

- alphasix/producer-gitlab:master
- alphasix/producer-gitlab:develop
- alphasix/producer-redmine:master
- alphasix/producer-redmine:develop
- alphasix/gestore-personale:master
- alphasix/gestore-personale:develop
- alphasix/consumer-telegram:master
- alphasix/consumer-telegram:develop
- alphasix/consumer-email:master
- alphasix/consumer-email:develop

Il tag rappresenta il branch dal quale è stata effettuata la build. Le immagini con tag 'master' saranno più stabili rispetto a quelle con altri tag, in quanto ancora in fase di sviluppo dato che possiedono funzionalità incomplete.

[15] Riferirsi alla voce 17 in §1.5.2
[16] Riferirsi alla voce 5 in §1.5.2.

4.2 Configurazione

4.2.1 Descrizione

Il processo di gestione delle configurazioni tratta di applicare procedure tecniche e amministrative attraverso il ciclo di vita del software per:

- Identificare, definire e fissare elementi software in un sistema
- Controllare modifiche e rilasci di questi elementi
- Registrare e fornire un resoconto del loro stato e delle richieste di modifica
- Assicurare la loro completezza, consistenza e correttezza
- Conrollare il loro immagazzinamento, gestione e consegna

Per questo motivo le attività che questo processo prevede sono:

1. Implementazione del processo
2. Identificazione della configurazione
3. Controllo della configurazione
4. Resoconto dello stato della configurazione
5. Valutazione della configurazione
6. Gestione del rilascio e consegna

Nello specifico, i sorgenti e i documenti ufficiali del gruppo vengono versionati utilizzando il sistema di controllo versione GIT$_G$, strumento scelto vista la sua ampia diffusione e la sua integrazione con GitHub. La scelta di utilizzare un client grafico oppure i comandi da terminale è lasciata al singolo membro del team.

4.2.2 Implementazione del processo

Ognuno di noi si deve attenere alle norme descritte nei prossimi paragrafi al fine di garantire l'identificazione di ogni versione dei prodotti. Ogni persona è responsabile di queste attività. L'Amministratore si occupa di controllare versioni e configurazioni del prodotto.

4.2.3 Identificazione della configurazione

Questa sezione determina le modalità con cui identifichiamo le versioni di ogni componente software e di ogni documento.

4.2.3.1 Versionamento

Tutti i documenti redatti, i file sorgente e il codice sorgente supportano il versionamento, in modo da essere univoci e rendere disponibile la possibilità di consultare versioni precedenti in qualsiasi punto del loro CICLO DI VITA$_G$. Il modello di versionamento adottato segue lo schema CHANGE SIGNIFICANCE$_G$.
La versione di un file è espressa secondo la notazione

```
X.Y.Z
```

dove:

- `X` indica il numero di versione principale. Inizia da 0 e viene incrementato ogni volta che il Responsabile approva il documento o il sorgente in esame, determinando una major release
- `Y` indica il numero di versione secondario, contatore delle fasi di verifica effettuate dal Verificatore superate positivamente. Inizia da 0. Viene azzerato ad ogni incremento della `X`
- `Z` è l'indice di modifica minore, incrementato ogni volta che viene effettuato un aggiornamento inferiore. Viene azzerato ad ogni incremento della `Y` o della `X`

4.2.4 Resoconto dello stato della configurazione e struttura delle repository

Una repository, essendo GitHub un VERSION CONTROL SYSTEM$_G$, ci serve per tenere traccia delle modifiche e di tutte le versioni delle nostre componenti. Abbiamo creato due repository principali fino a questo momento:

- **Butterfly**: contenente tutti i documenti ufficiali redatti e il relativo codice sorgente di Butterfly.
- **Butterfly-PoC**: contenente i sorgenti nello stato in cui erano per la dimostrazione del Proof of Concept e i README esplicativi.

4.2.4.1 Norme di branching

Ogni repository che usiamo avrà due BRANCH$_G$ principali:

- **master**: branch principale che viene aggiornato solamente in vista di un rilascio, quando il Responsabile approva un file o documento, che causa lo scatto del numero di versione più significativo.
- **develop**: branch usato per lo sviluppo in vista di un rilascio, senza lasciare il master branch in uno stato di incompletezza. Viene aggiornato quando una funzionalità è matura.

Sono presenti poi dei branch dedicati allo sviluppo e alla manutenzione, che verranno aperti in base alle necessità:

- **feature/nome-feature**: branch aperti in vista di uno sviluppo di una qualsiasi funzionalità (nome-feature è un placeholder) o bug fix sul develop. Una volta che la funzionalità raggiunge un adatto stato di maturità, verrà fatto il merge sul develop e cancellato il branch relativo a quella funzionalità.
- **hotfix/nome-hotfix**: branch aperti in vista di un bug non notato nel master branch che necessita di essere sistemato al più presto o in fase di correzione delle problemi segnalati dai professori a seguito di una revisione.

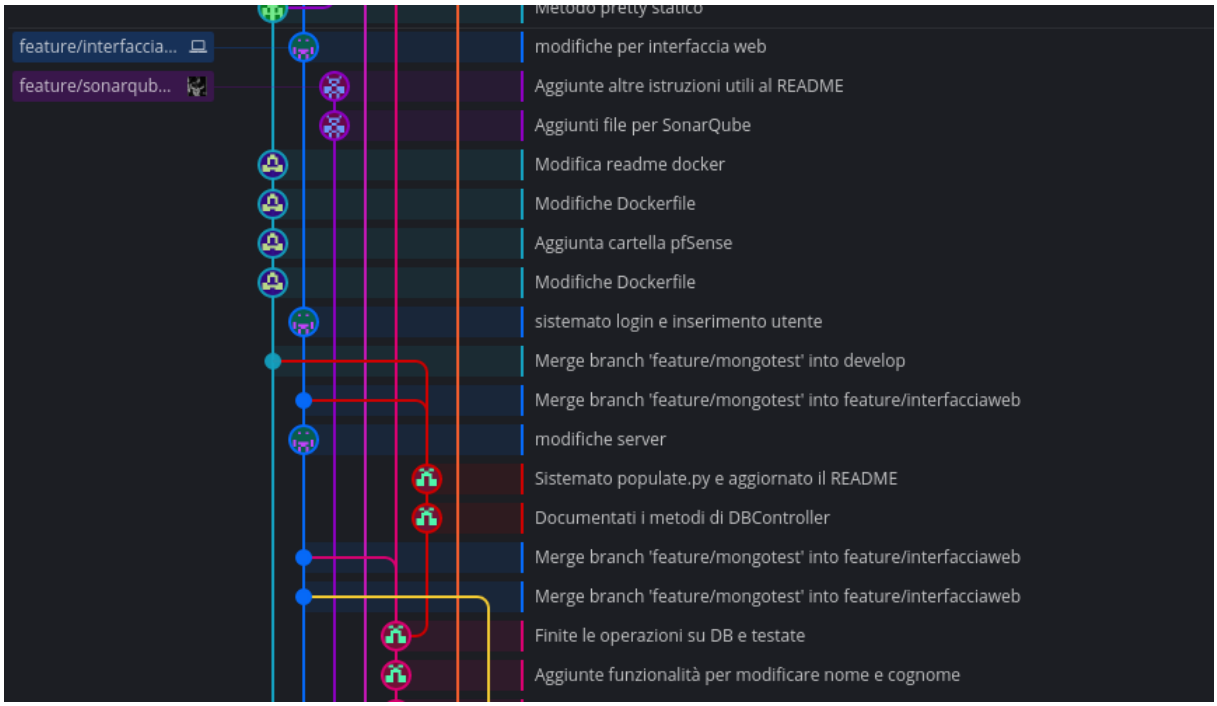

Figura 2: Visione generale dell'albero dei branch in un momento casuale da Gitkraken

4.2.5 Controllo della configurazione

Per il controllo e l'approvazione dei cambiamenti alle repository adottiamo le seguenti regole. È cura del Verificatore controllare che vengano rispettate.

4.2.5.1 Aggiornamento della repository

Per aggiornare la repository remota con i cambiamenti effettuati nella workcopy locale, la prassi da seguire è la seguente:

- Controllare di trovarsi nel branch corretto con il comando:

  ```
  $ git branch
  ```

  In caso negativo, dare il comando `git checkout nomebranch` per spostarsi nel branch corretto prima di procedere.

- Aggiornare la workcopy locale con eventuali cambiamenti presenti nella repository remota:

  ```
  $ git pull
  ```

  - In caso di conflitti, dare la sequenza di comandi che segue per risolverli senza dover effettuare un commit dedicato solo al merge:

    ```
    $ git stash
    $ git pull
    $ git stash pop
    ```

    A questo punto, si potrebbero avere dei conflitti marcati opportunamente da Git, oppure avere la copia aggiornata con le proprie modifiche applicate.

- Aggiungere i file allo "stage" relativi a una specifica modifica con il comando (è possibile usare * come wildcard per aggiungere più file contemporaneamente):

```
$ git add [nome-file1] [nome-file2] ...
```

- Fare il commit e riassumere i cambiamenti apportati:

```
$ git commit -m "Cambiamenti apportati"
```

- Effettuare il push per pubblicare le modifiche locali nella repository remota:

```
$ git push
```

4.2.6 Valutazione della configurazione

Per valutare la completezza delle funzionalità fornite da Butterfly facciamo riferimento ai test di validazione presenti nel *PianoDiQualifica v4.0.0*$_{\text{D}}$. Per quel che riguarda la progettazione e il codice ci basiamo sul feedback ottenuto dal committente tramite la Product Baseline. Grazie ad esso infatti ci interroghiamo sulla conformità della nostra progettazione software.

4.2.7 Gestione del rilascio e consegna

La consegna del prodotto finale avverrà al termine del progetto tramite un supporto fisico dato direttamente al committente.

4.3 Quality assurance

Il processo di QUALITY ASSURANCE$_{\text{G}}$ ci fornisce l'adeguata certezza che il nostro prodotto software e processi siano conformi ai loro specifici requisiti e che aderiscano ai piani stabiliti per essi.

4.3.1 Implementazione del processo

Il nostro piano, che determina come i processi e i compiti di quality assurance devono essere sviluppati e implementati, è delineato nel *PianoDiQualifica v4.0.0*$_{\text{D}}$. Esso include:

- Standard di qualità, metodologie, procedure e strumenti per eseguire attività di quality assurance
- Procedure per l'identificazione, raccolta, archiviazione e mantenimento dei risultati di qualità
- Risorse e programma per condurre le attività di quality assurance

4.3.2 Standard di qualità

Prendiamo come riferimento per gli obiettivi di qualità del prodotto determinati standard.
In questa appendice descriviamo gli standard adottati nelle loro parti più rilevanti ai fini del progetto. Nelle *NormeDiProgetto v4.0.0*$_{D}$ viene indicato in che misura questi standard saranno applicati e nel *PianoDiQualifica v4.0.0*$_{D}$ il piano che abbiamo scelto per rispettarli.

4.3.2.1 ISO/IEC 15504 (SPICE)

Lo standard ISO/IEC 15504 è stato creato per unire in un unico standard le caratteristiche principali di CMMI$_{G}$ e SPY$_{G}$; entrambi standard riguardanti la qualità di processi software. ISO/IEC 15504 è chiamato anche SPICE come acronimo di *Software Process Improvement and Capability Determination*, dando importanza al termine "Capability" inteso come la capacità di un processo di essere cognitivamente capace di raggiungere il suo scopo.
Un processo con un'alta Capability è osservato da tutti in modo disciplinato e sistematico. In caso di bassa Capability il processo viene effettuato in modo opportunistico e disorganizzato.
SPICE mette a disposizione una metrica per valutare diversi attributi per ogni processo ed assegna un valore quantificabile ad ognuno di questi in modo tale da esplicitare come poter migliorare tale processo. Ogni valutazione in questo modo può essere oggettiva, ripetibile e comparabile.
I processi vengono classificati in:

- Cliente/Fornitore
- Ingegneria
- Supporto
- Gestione
- Organizzazione

I livelli sono:

- **0 Incompleto**: il processo è caotico perché con risultati e performance incomplete.
- **1 Performato**: il processo inizia ad essere eseguito mettendo a disposizione degli input ed output.

  Attributi:

  - **Esecuzione dei processi**: indica il numero di obiettivi raggiunti.
- **2 Gestito**: le responsabilità e la gestione del processo sono definite.

  Attributi:

  - **Gestione del processo**: indica quanto sono organizzati gli obiettivi fissati.
  - **Gestione del prodotto**: indica quanto sono organizzati o gestiti i prodotti rilasciati.
- **3 Stabilito**: il processo è pronto per diventare un processo standard ed essere rilasciato.

  Attributi:

  - **Definizione del processo**: indica quanto il processo aderisce agli standard.
  - **Distribuzione del processo**: indica in che misura il processo possa essere rilasciato potendo restituire sempre lo stesso risultato.

- **4 Prevedibile**: il processo è in grado di essere sottoposto a metriche e valutazioni quantitative. Spesso i risultati sono predicibili.

  Attributi:

  - **Misurazioni del processo**: indica quanto le metriche possono essere applicate al processo.
  - **Controllo del processo**: indica quanto i risultati delle valutazioni siano predicibili.

- **5 Ottimizzante**: il processo attua miglioramenti qualitativi e quantitativi.

  Attributi:

  - **Innovazione del processo**: indica quanto i cambiamenti attuati nel processo risultino innovativi e positivi grazie ad una fase di analisi.
  - **Ottimizzazione del processo**: indica quanto la curva di miglioramento del processo sia lineare.

Ad ogni attributo viene data una valutazione assegnata in base alla percentuale di soddisfacimento dell'attributo:

- **N**: il processo non è implementato e non svolge niente di significativo (0% - 15%).
- **P**: il processo è parzialmente implementato (15% - 50%).
- **L**: il processo è largamente implementato (50% - 85%).
- **F**: il processo è completamente implementato (85% - 100%).

Attributi	**Valutazioni**			
	N	**P**	**L**	**F**
Esecuzione dei processi	[0 - 1]			
Gestione del processo Gestione del prodotto	[1 - 2]			
Definizione del processo Distribuzione del processo	[2 - 3]			
Misurazione del processo Controllo del processo	[3 - 4]			
Innovazione del processo Ottimizzazione del processo	[4 - 5]			

Tabella 3: Schema degli attributi di ISO/IEC 15504

4.3.2.2 ISO/IEC 9126:2001

ISO/IEC 9126 è uno standard inerente alla qualità del software. Esso è strutturato in modo tale che si possa migliorare l'insieme dei processi.
La sua struttura prevede tre tipi di qualità, ognuna delle quali possiede determinate caratteristiche:

- **Qualità interna**: misura la qualità del prodotto non in esecuzione, in questo caso parliamo del codice sorgente a cui si possono aggiungere diverse metriche attraverso l'analisi statica. Queste poi possono stabilire la manutenibilità e portabilità del prodotto.

  Gli attributi ad essa assegnati sono:

  - Manutenibilità
  - Portabilità

- **Qualità esterna**: misura attraverso l'analisi dinamica quanto l'esecuzione del prodotto rispetti gli obiettivi prefissati.

  Gli attributi ad essa assegnati sono:

  - Funzionalità
  - Efficienza
  - Affidabilità
  - Usabilità

- **Qualità in uso**: definisce le metriche del prodotto rilasciato ed usato dal cliente che ne misurerà la qualità.

  Gli attributi ad essa assegnati sono:

  - Efficacia
  - Produttività
  - Soddisfazione
  - Safety

4.3.2.2.1 Descrizione degli attributi della qualità interna e della qualità esterna

- **Manutenibilità**: il software nel corso delle sue revisioni deve essere facilmente modificabile.

  Nello specifico si prevede:

  - **Analizzabilità**: prevedere una lettura del codice fruibile.
  - **Modificabilità**: poter capire subito dove applicare la modifica.
  - **Stabilità**: evitare effetti indesiderati dopo le modifiche.
  - **Testabilità**: poter creare facilmente dei test su tutto il codice.

- **Portabilità**: il software dovrebbe poter essere eseguito in più ambienti.

  Nello specifico si prevede:

  - **Adattabilità**: potersi adattare automaticamente ai vari ambienti.
  - **Installabilità**: la sua fase d'installazione dovrebbe essere semplice.
  - **Conformità**: il software deve sapere coesistere con le altre applicazioni.
  - **Sostituibilità**: essere capace di sostituire un software con gli stessi scopi.

- **Funzionalità**: il software deve mettere a disposizione le funzionalità richieste in rapporto all'ambiente d'esecuzione.

  Nello specifico si prevede:

  - **Appropriatezza**: le funzionalità del software sono appropriate ai requisiti richiesti.
  - **Accuratezza**: in che misura le funzionalità aderiscono ai requisiti richiesti.
  - **Interoperabilità**: la capacità di interagire coi sistemi specificati.
  - **Security**: proteggere le informazioni da agenti esterni.

- **Efficienza**: misura della capacità di raggiungere gli obiettivi stabiliti cercando di usare meno risorse possibili, in particolare:

  - **Nel tempo**: poter dare risposte in un tempo di scadenza appropriato.
  - **Nello spazio**: utilizzo del minor numero di risorse.

- **Affidabilità**: il software deve mantenere le specifiche richieste senza inconvenienti.

  Nello specifico si prevede:

  - **Maturità**: indica il livello minimo della qualità delle parti che determina poi il livello di qualità dell'intero prodotto.
  - **Robustezza**: la capacità di saper reagire agli errori.
  - **Recuperabilità**: per poter tornare alla versione precedente del software in modo semplice.

- **Usabilità**: il software deve poter essere compreso fin da subito, dunque semplice ed immediato nell'utilizzo e nella comprensione.

  Nello specifico si prevede:

  - **Comprensibilità**: il significato delle funzionalità deve essere compreso il prima possibile dall'utente.
  - **Apprendibilità**: capacità dell'utente di imparare ad utilizzare le funzionalità disponibili.
  - **Operabilità**: capacità dell'utente di usare e controllare il software.
  - **Attrattiva**: il risultato deve risultare attraente all'utente.

4.3.2.2.2 Descrizione degli attributi della qualità in uso

- **Efficacia**: misura della capacità di riuscire a raggiungere i compiti fissati. Essa si calcola in base al grado di raggiungimento degli obiettivi.
- **Produttività**: intesa come $\frac{\text{unità di prodotto realizzato}}{\text{unità di risorse utilizzate}}$.
- **Soddisfazione**: il software deve soddisfare l'utente.
- **Safety**: il software deve possedere adeguati livelli di sicurezza per il tipo di utente che ne usufruisce.

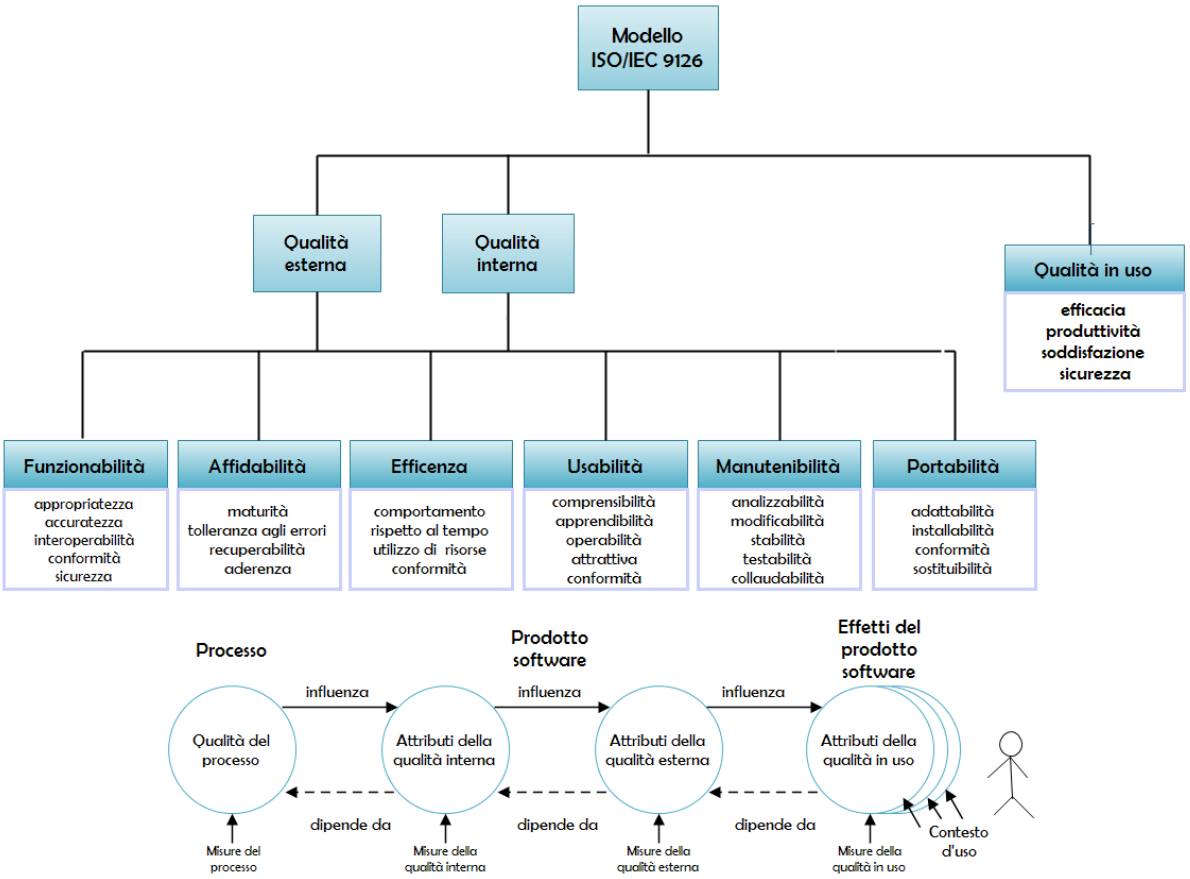

Figura 3: Schema ISO 9126 con il suo ciclo di vita[17]

4.3.2.3 Ciclo di Deming

Il Ciclo di Deming è un metodo iterativo creato per migliorare la qualità dei processi e dei prodotti software. Questo opera nell'ottica di un miglioramento continuo in termini di risultati e risorse utilizzate e al termine di ogni ciclo, l'eventuale miglioramento effettuato diventa la nuova base da cui inizia l'iterazione successiva.
Si distingue in quattro fasi, chiamate anche PDCA, che sono:

- **Plan**: pianificare i processi per ottenere i risultati attesi ed osservare dove questi possono essere migliorati.
- **Do**: eseguire il programma e i miglioramenti inseriti nella fase precedente.
- **Check**: effettuare test e controlli dell'output della fase precedente confrontandoli con gli obiettivi della fase di Plan. Dare una valutazione dei risultati per verificare se i cambiamenti attuati portano effettivamente dei miglioramenti.
- **Act**: le modifiche, se risultano essere delle migliorie vengono applicate al processo o al prodotto.

[17] Riferirsi alla fonte 8 in §1.5.2 per la fonte della figura

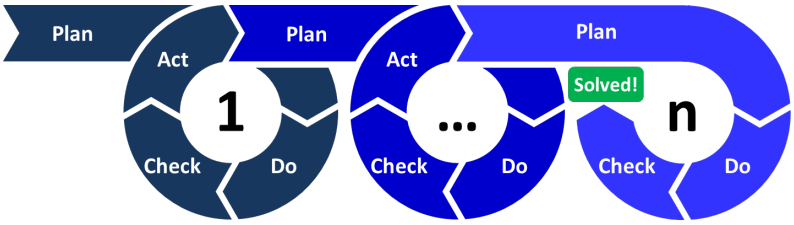

Figura 4: Ciclo di Deming[18]

4.3.2.4 ISO/IEC 90003:2004

Lo standard ISO/IEC 90003 consiste nello standard ISO 9001:2008 applicato al software. Quest'ultimo tratta di uno standard sulla qualità dei sistemi aziendali che segue il sistema il Ciclo di Deming descritto in §4.3.2.3.
L'ISO/IEC 90003:2004 si suddivide in otto capitoli che sono:

1. Scopo
2. Riferimenti normativi
3. Termini e definizioni
4. Sistema di gestione della qualità
5. Gestione delle responsabilità
6. Gestione delle risorse
7. Realizzazione del prodotto
8. Misurazione, analisi e miglioramento

Date le nostre capacità e la nostra esperienza, prendiamo in considerazione solo alcuni punti del quarto capitolo. In particolar modo quelli riguardanti la documentazione, non potendo ancora stabilire metriche sul software.

4.3.2.4.1 Capitolo 4. dell'ISO 90003:2004 requisiti di sistema e linee guida

1. **Requisiti e linee guida sull'organizzazione**:
   (a) **Stabilire la gestione del sistema di qualità (QMS)**
      - **Identificare i requisiti di cui il QMS ha bisogno**: attraverso il Piano di Qualifica osservando i processi interni ed esterni.
      - **Assicurarsi che ogni processo sia efficace**: attraverso una verifica continua su processi e prodotti.

   (b) **Documentare il proprio QMS**: riportare le relazioni tra i vari processi e come questi vengono gestiti e verificati.
   (c) **Cercare di migliorare il proprio QMS**: applicando il Ciclo di Deming.
   (d) **Mantenere la qualità del QMS**.

[18]Riferirsi alla fonte 9 in §1.5.2 per la fonte della figura

2. **Requisiti e linee guida sulla documentazione**:

   (a) **Documentare la qualità**:

   - **Pianificare la documentazione del QMS**: assicurandosi che quello che verrà scritto rispecchi gli obiettivi e la gestione dei processi.
   - **Stabilire la documentazione del QMS**: stabilendo la qualità delle diverse parti del QMS.
   - Mantenere la documentazione nel tempo.

   (b) **Controllare la qualità dei documenti**:

   - **Stabilire una procedura per controllare i documenti**: documentare la stessa procedura di controllo che dovrebbe contenere la fase di approvazione del documento, un suo versionamento e continuo aggiornamento.
   - Mantenere ed aggiornare la procedura di controllo del documento.

4.3.2.4.2 Capitolo 7. dell'ISO 90003:2004 realizzazione del prodotto

1. **Pianificazione prodotto**:

   (a) **Pianificare la realizzazione del prodotto**: i vari passaggi per la realizzazione di un prodotto, considerando il controllo della sua qualità.

   (b) Applicare la pianificazione.

   (c) **Ciclo di vita**: individuare il ciclo di vita ideale per la realizzazione dei prodotti.

   (d) **Pianificare la qualità del prodotto**: stabilire gli obiettivi di qualità dei vari prodotti.

2. **Analisi requisiti**:

   (a) **Determinare i requisiti per**:

   - Committenti.
   - Sviluppatori.

   (b) **Progettazione**:

   - Sviluppare il design del prodotto, identificando i tool o software esterni da usare.
   - Pianificare le attività di verifica indicando gli input e gli output che il software si aspetta e restituisce.
   - Valutare i rischi possibili nella fase di sviluppo.

3. **Verifica errori**

4. **Identificazione e tracciamento prodotti**

4.3.3 Attività

Nello specifico, le attività trattate dal *PianoDiQualifica v4.0.0*$_{\text{D}}$ sono:

- Qualità di processo
- Qualità di prodotto
- Resoconto delle attività di verifica
- Mitigazione delle variazioni orarie
- Valutazione per il miglioramento

4.4 Verifica

4.4.1 Scopo

Questa sezione vuole descrivere come eseguiamo la fase di verifica, lo scopo è quello di evidenziare ed eliminare la presenza di errori nell'esecuzione di un qualunque processo durante tutto lo sviluppo del prodotto.

4.4.2 Descrizione

Ogni processo e prodotto deve essere valutato in modo quantificabile attraverso metriche apposite, quando possibile, e stabilendo il risultato che si vuole raggiungere.
Come indicato dal Ciclo di Deming$_G$, nel momento in cui tale risultato sarà raggiunto, se esso non è il migliore, servirà come "base" per alzare il livello di qualità di quel processo o prodotto.
I risultati ottenuti nella fase di verifica sono riportati nel *PianoDiQualifica v4.0.0*$_D$: in questo modo, confrontandoli con gli esiti attesi, è possibile valutare un miglioramento per i vari processi e prodotti.

4.4.3 Walkthrough e Inspection

Due modi di effettuare la verifica sono attraverso Walkthrough e Inspection. Li adotteremo entrambi, ma non contemporaneamente, per i vantaggi che comporta ogni metodo.

4.4.3.1 Walkthrough

Metodo di verifica che effettua un controllo ad ampio spettro senza l'assunzione di presupposti. Dato che per mettere in atto tale metodo ci si deve dividere in gruppi dove ognuno ha un ruolo ben distinto, Walkthrough è ideale per le verifiche effettuate all'inizio dei vari periodi, dove i membri del team di sviluppo non possiedono le conoscenze adeguate per una verifica efficiente.
Walkthrough possiede delle fasi ben specifiche:

1. **Pianificazione**: viene pianificato in gruppo come effettuare la verifica dei prodotti.
2. **Lettura**: viene effettuata la lettura del prodotto.
3. **Discussione**: vengono discusse le possibili correzioni.
4. **Correzione di difetti**: si attuano i risultati ottenuti dalla fase di discussione.

4.4.3.2 Inspection

Metodo di verifica dove si esegue una lettura mirata dei prodotti, frutto di un'analisi dei risultati dei precedenti test. Questo metodo dunque, a differenza di Walkthrough, prevede l'esecuzione con dei presupposti.
Le fasi di Inspection sono:

1. **Pianificazione**: viene sempre deciso in gruppo come effettuare la pianificazione.
2. **Definizione di una lista di controllo**: dato che le parti da verificare sono specificate, queste possono essere riportate in una lista in modo da velocizzare il processo di verifica.
3. **Correzione dei difetti**: attuazione dei punti della lista di controllo.

Per la natura di Inspection, questa non può essere applicata fin dall'inizio, dunque nel momento in cui si presentano nuove tipologie di prodotti e processi da verificare viene effettuato Walkthrough; nel momento in cui il metodo di verifica è ben consolidato da tutti noi, si passa ad effettuare verifica secondo Inspection.

4.4.4 Verifica del design e conformità del codice

Il Verificatore ha il compito di verificare la progettazione. In particolare si occupa di:

- Controllare che i design pattern che abbiamo scelto vengano effettivamente rappresentati nella maniera corretta tramite i diagrammi e che rispettino gli obiettivi preposti
- Verificare che il codice sviluppato sia coerente con quanto dichiarato nella progettazione

4.4.5 Metodologie di sviluppo del software

4.4.5.1 The Twelve-Factor App

The Twelve-Factor App$_G$ è una serie di dodici regole destinate a chi vuole sviluppare software-as-a-service$_G$ (SaaS). Sono utili per verificare in corso d'opera la qualità del progetto, valutando quali punti riesce a rispettare l'applicazione.
I suoi principi sono:

1. **Codebase$_G$**: deve essere presente una sola codebase versionata da un Version Control System$_G$ (VCS) come GitLAB$_G$ da cui possono derivare diversi deploy$_G$.
2. **Dipendenze**: le librerie usate dal codice devono essere presenti nella directory della singola applicazione e non attive a livello di sistema$_G$. In questo modo l'applicazione è il meno dipendente possibile dal sistema di esecuzione.
3. **Configurazione**: i parametri di configurazione dell'applicazione devono essere completamente separati dalla sua implementazione.
4. **Backing Service**: l'applicazione non deve far distinzione tra funzionalità uguali usate in locale o remoto.
5. **Build, release, esecuzione**: bisogna separare in modo netto la fase di build, quella di deploy e quella esecuzione, usando tool$_G$ differenti e diverse repository per salvare i risultati delle varie fasi.
6. **Processi**: l'esecuzione dell'applicazione deve essere vista come l'insieme di uno o più processi che restituiscono un risultato. Questi sono di tipo stateless$_G$.
7. **Binding$_G$ delle Porte**: l'applicazione è completamente contenuta in se stessa e non fa affidamento ad un altro servizio nell'ambiente di esecuzione. Effettua invece il binding delle porte diventando un servizio per le richieste esterne.
8. **Concorrenza**: sviluppare i processi in modo tale che possano lavorare su un sistema decentralizzato.
9. **Rilasciabilità**: i processi dell'applicazione devono poter essere avviati e fermati quando se ne ha bisogno senza passaggi bruschi.
10. **Parità tra sviluppo e produzione**: deve esserci meno differenza possibile tra lo stato di sviluppo e quello di produzione. Questo si ottiene facendo un rilascio continuo del prodotto.
11. **Log**: l'applicazione dovrebbe poter offrire un sistema di login.
12. **Processi di Amministrazione**: porre attenzione a quei processi che devono essere eseguiti una tantum dagli sviluppatori ad esempio. Questi processi devono poter essere accessibili solo ad alcuni e indicati in una specifica release.

In accordo col cliente Imola Informatica, non tutti i punti di Twelve-Factor App possono essere rispettati. In particolare non è richiesto il soddisfacimento del punto 12, mentre il punto 11 è lasciato al fornitore come opzionale.

4.4.5.2 Test Driven Development

Come metodologia di sviluppo del software adottiamo il Test Driven Development (TDD) per ogni componente. Questa metodologia prevede tre fasi nello sviluppo che, ripetute ciclicamente, costituiscono il Ciclo TDD:

- **Fase rossa**: il Programmatore scrive il test relativo alla funzionalità che vuole implementare, prima di scrivere il nome del metodo o la sua implementazione. Questo porterà ad un fallimento assicurato poiché il metodo non esiste o non fa nulla.
- **Fase verde**: il Programmatore scrive il codice della funzionalità minimo indispensabile a far superare il test.
- **Fase grigia (Refactoring)**: il Programmatore riscrive il codice per far sì che il codice rispetti gli obiettivi di qualità definiti nel *PianoDiQualifica v4.0.0*$_{\text{D}}$ e al contempo superi il test.

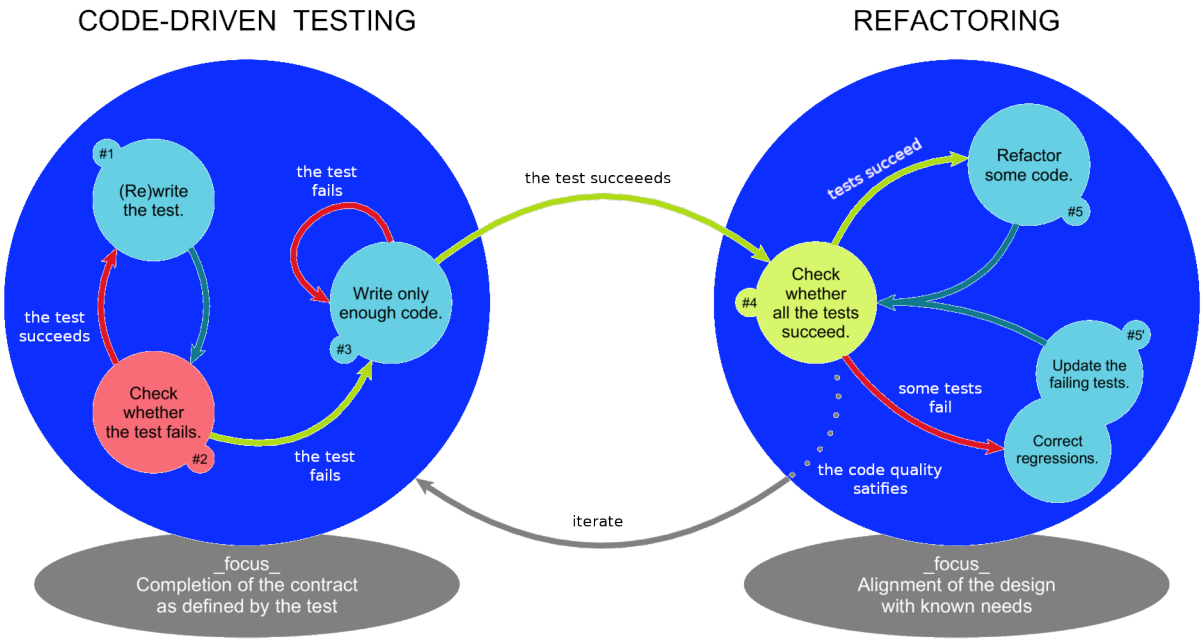

Figura 5: Test Driven Development[19]

4.4.6 Analisi statica

L'analisi statica attua una verifica non dinamica, ovvero senza eseguire codice. Può essere effettuata sui documenti o sul codice in maniera statica.

4.4.6.1 Analisi dei documenti

L'analisi statica per i documenti si limita a valutare come e con che contenuti questi vengono scritti. Per l'analisi dei documenti vengono utilizzate due metriche:

- MPD001
- MPD002

La denominazione delle metriche è descritta in §3.1.2.3.

[19] Riferirsi alla voce 7 in §1.5.2

4.4.6.1.1 MPD001 Indice di Gulpease

Per il calcolo dell'indice di Gulpease, abbiamo creato uno script ad hoc che, preso in input un file PDF, produce in output l'indice di Gulpease. L'indice si calcola:

$$89 + \frac{300 \times n_{\text{frasi}} - 10 \times n_{\text{lettere}}}{n_{\text{parole}}}$$

Metrica: il risultato della formula è interpretato nel seguente modo

- <**80**: documento difficile da leggere per chi ha la licenza elementare.
- <**60**: documento difficile da leggere per chi ha la licenza media.
- <**40**: documento difficile da leggere per chi ha un diploma superiore.

Nel momento in cui avviene un commit all'interno di repository, in automatico si avvia uno script che analizza tutti i documenti in PDF per valutarne l'indice di Gulpease. I risultati vengono poi riportati in un apposito file di testo per verificarne la qualità ed un possibile miglioramento.

4.4.6.1.2 MPD002 Correttezza ortografica

Gli errori ortografici possono essere segnalati dallo strumento di Controllo Ortografico presente in *TexStudio* e da GNU Aspell.
Metrica: il numero di errori ortografici presenti nel documento.

4.4.6.2 Analisi dei processi

Per analizzare i processi usiamo gli standard sopra elencati. Ad ogni fase del processo, verranno valutati gli attributi richiesti secondo l'ISO 15504, in che misura questi sono stati rispettati e in che fase del Ciclo di Deming il processo si trova. Per l'analisi dei processi vengono utilizzate:

- MPR003 Aderenza agli standard
- MPR004 Frequenza commit nella repository
- MPR009 Frequenza controllo prodotti
- MPR010 Moduli codificati prima della progettazione
- MPR011 Numero di design pattern applicati
- MPR012 Moduli non testati

La denominazione delle metriche è descritta in §3.1.2.3.

4.4.6.2.1 MPR003 Aderenza agli standard

Per misurare e verificare i processi sono stati scelti alcuni specifici standard di qualità descritti nel *PianoDiQualifica v4.0.0*$_{\text{D}}$ che possono offrire una valutazione quantitativa. Gli standard scelti sono:

- **ISO/IEC 15504**: ogni processo attivato verrà classificato e valutato secondo gli attributi assegnati ai vari livelli di qualità. Per ogni attributo verrà infine indicata una percentuale di quanto il processo rispetti l'attributo, potendo infine capire nel complesso quanto quel processo riesca a superare un dato livello di maturità.
- **Ciclo di Deming**: nella fase migliorativa del processo sarà data particolare attenzione nel non iniziare una fase del Ciclo di Deming senza aver finito completamente le fasi precedenti.

Metrica: il livello di maturità è descritto in appendice A.1 del *PianoDiQualifica v4.0.0*$_{\text{D}}$.

4.4.6.2.2 MPR004 Frequenza commit nella repository

Per mantenere aggiornate le versioni dei prodotti è necessario che ognuno di noi effettui un commit ad ogni sua modifica significativa. Così facendo, in caso di errori, è possibile tornare ad una versione stabile del progetto. Misurare quanti commit sono stati effettuati inoltre serve per monitorare l'attività del team di sviluppo.

$$\frac{\sum_{i=1}^{n} x_i}{n} \qquad x_i = \text{numero di commit alla i-esima settimana del macro periodo.}$$

Metrica: numero minimo di commit da effettuare in media ogni settimana lavorativa durante un macro-periodo.

4.4.6.2.3 MPR009 Frequenza controllo prodotti

I documenti e i prodotti software hanno bisogno di una verifica frequente, commisurata in base al numero di modifiche che vengono apportate.
Chi esegue la modifica deve controllare ciò che ha fatto prima di poterla ufficializzare, mentre la verifica fatta dal Verificatore deve essere fatta solo nel momento in cui è stato raggiunto un numero significativo di modifiche, per evitare di spendere troppe risorse in questa fase.

$$\frac{\sum_{i=1}^{n} x_i}{n} \qquad x_i = \text{numero di modifiche effettuate prima della i-esima verifica.}$$

Metrica: media del numero massimo di modifiche apportate ai prodotti prima che ricevano una verifica dal parte del Verificatore.

4.4.6.2.4 MPR010 Moduli codificati prima della progettazione

La progettazione di un MODULO$_G$ è fondamentale per rendere il prodotto estensibile, mantenibile e comprensibile. Per questo è buona prassi avviare il processo di progettazione prima di quello di codifica.
La progettazione di ogni modulo all'interno dell'intero prodotto software deve essere fatta prima della sua codifica, la quale deve rispecchiare la progettazione.
Metrica: percentuale di moduli non progettati prima della loro codifica. Per fare ciò, nelle varie fasi di verifica, si contano i moduli codificati prima di essere progettati. Verso il termine del periodo di revisione si calcola la loro percentuale in base al numero totale di moduli prodotti.

4.4.6.2.5 MPR011 Numero di design pattern applicati

Dato che i design patter possono essere considerati una "buona pratica" per produrre del codice in determinati contesti, la loro presenza può aiutare ad innalzare la qualità del prodotto. Il loro uso massiccio però può portare ad un'eccessiva complessità e talvolta anche ad un mal funzionamento del prodotto se applicati erroneamente; per questo i design pattern devono essere applicati in maniera ponderata.
Metrica: numero di volte che si applica uno o più tipi di design pattern.

4.4.6.2.6 MPR012 Moduli non testati

Dobbiamo sempre testare i MODULI$_G$ per assicurarci la loro correttezza. Per fare ciò, devono essere testate le UNITÀ$_G$ che li compongono attraverso i test di unità (definiti in §4.4.7.3). Ogni modulo deve dunque essere testato.
Metrica: numero di moduli codificati e privi di test di unità.

4.4.6.3 Analisi del software

L'analisi statica del codice, insieme alle norme di codifica stabilite in §2.2.4, serve per permetterci di scrivere programmi verificabili. In particolare ci aiuta per assicurare un comportamento predicibile, per darci dei criteri di programmazione ben fondati da usare e per ragioni pragmatiche. Un potente mezzo che ci permette di eseguire analisi statica sul codice è SonarQube$_G$. Da esso, traiamo le seguenti metriche:

- MPS001 Presenza di bug
- MPS002 Presenza di vulnerabilità
- MPS003 Presenza di code smell
- MPS004 Duplicazione del codice
- MPS017 Code coverage

La denominazione delle metriche è descritta in §3.1.2.3.

4.4.6.3.1 Controllo del Python coding style

Per l'analisi statica dello stile di codifica dei sorgenti Python, utilizziamo due strumenti che segnalano se lo stile di codifica non rispetta le norme definite nel PEP 8:

- `pycodestyle`: un tool che segnala, tramite comando da terminale, se e quali righe di codice di un file non rispettano lo standard PEP 8.
- `pylama`: un linter che segnala dinamicamente se ciò che viene digitato contiene degli errori o non rispetta lo standard PEP 8.

4.4.6.3.2 SonarQube

È una piattaforma open-source per realizzare controlli sulla qualità del codice tramite analisi statica. Lo utiliazziamo perché è possibile creare dei progetti che scansionano il nostro progetto locale e ci mostrano in una dashboard le attività più significative che lo strumento rileva. Si possono rilevare bug, code smell$_G$, duplicazione e vulnerabilità del codice. Viene inoltre assegnato una valore "passato" o "non passato" denominato Quality gate, che ci consente di valutare immediatamente la qualità generale del nostro codice. In particolare, SonarQube rispetto ad altri strumenti simili, ci è molto utile perchè supporta parecchi linguaggi di programmazione, tra cui quello che noi più utilizziamo: Python.

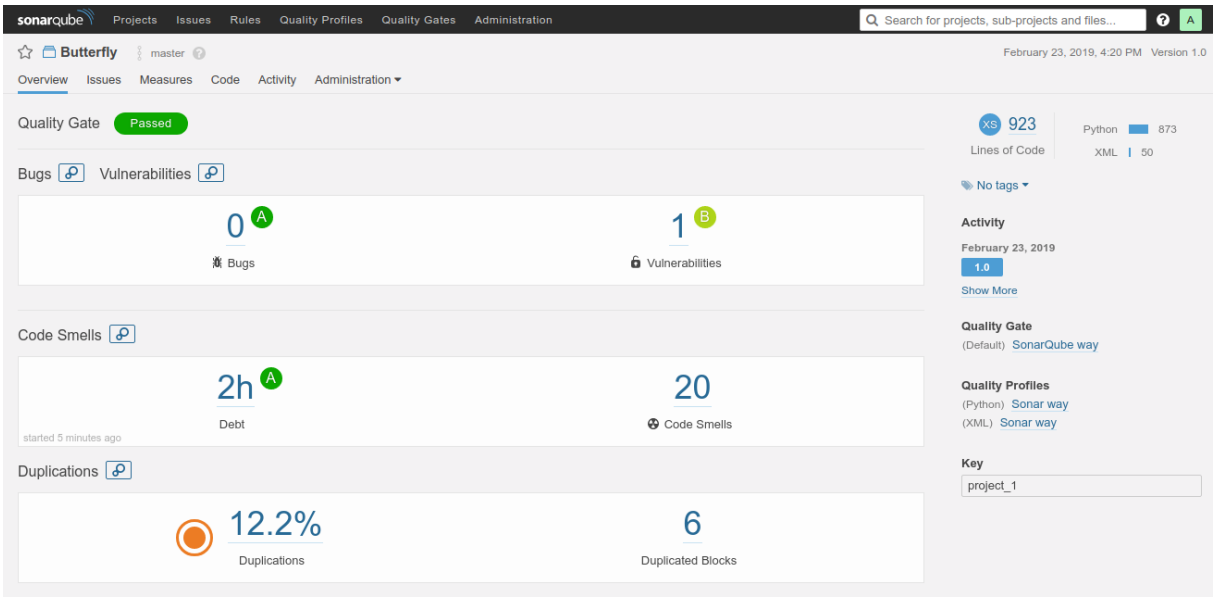

Figura 6: Esempio di immagine catturata del progetto Butterfly in SonarQube.

4.4.6.3.3 MPS001 Presenza di bug

I bug sono difetti nel codice che vogliamo assolutamente tenere sotto controllo e rimuovere se rilevati. A questo scopo, SonarQube ci dà la possibilità di conteggiare il numero di bug presenti nel codice.
Metrica: numero di bug presenti nel codice al momento della scansione.

4.4.6.3.4 MPS002 Presenza di vulnerabilità

Le vulnerabilità hanno un peso di gravità minore per noi, ma è comunque buona prassi cercare di minimizzarle il più possibile in modo da non incorrere in problemi di sicurezza. SonarQube ci dà la possibilità di visualizzare le vulnerabilità presenti nel codice.
Metrica: numero di vulnerabilità presenti nel codice al momento della scansione.

4.4.6.3.5 MPS003 Presenza di code smell

I code smell non sono errori, ma indicano delle debolezze nel codice. La maggior parte di code smell segnalati da SonarQube, riguardano spesso linee di commenti che decrementano la leggibilità del codice.
Metrica: numero di code smell presenti nel codice al momento della scansione.

4.4.6.3.6 MPS004 Duplicazione del codice

Linee di codice duplicate segnalano inefficienza all'interno del nostro codice. Del buon codice infatti non è ridondante. Se lo è, vuol dire che può essere migliorato organizzandolo in modo differente. Per questo SonarQube ci dà la possibilità di visualizzare esattamente il numero percentuale di codice duplicato all'interno di un file, sul suo totale di righe.
Metrica: percentuale di linee di codice duplicate presenti nel codice al momento della scansione.

4.4.6.3.7 MPS017 Code coverage

Per sapere se un modulo è stato correttamente testato, si verifica in percentuale quante righe di codice vengono eseguite per superare una suite di test. Più è alta questa percentuale, più è bassa la probabilità che il codice contenga dei bug.

Metrica: percentuale di CODE COVERAGE$_G$.

4.4.7 Analisi dinamica

L'analisi dinamica valuta il comportamento dei prodotti in esecuzione, verificando se restituiscono i risultati attesi e se operano nel modo stabilito. Vengono presi in esame i prodotti software e i processi.
È oppurtono ricordare che i test devono essere:

- Ripetibili
- Automatizzati tramite strumenti
- Oggettivi e non personalizzati

Quindi, perché l'analisi dinamica avvenga in modo corretto, è necessario tenere conto di alcuni importanti elementi quali:

- **Ambiente di sviluppo**: il sistema hardware e software adottato durante il test.
- **Stato iniziale**: primo stato del prodotto, antecedente all'inizio del test.
- **Input**: dati inseriti.
- **Output**: dati attesi.
- **Notifica risultato**: notifiche riguardanti il risultato ottenuto dal test (opzionale).

Per effettuare analisi dinamica è necessario sviluppare dei dei test che verifichino l'integrità dei prodotti. Questi sono classificati attraverso un identificativo univoco descritto nel prossimo paragrafo e suddivisi per tipo in base alla tipologia dell'oggetto di test.

4.4.7.1 Classificazione dei test

Ogni test viene classificato identificato univocamente da tale codice:

`T[Tipo][ID]`

- **T**: si riferisce a "Test".
- **Tipo**: la tipologia a cui il test appartiene che, seguendo il modello a V, può essere:
  - **V**: validazione.
  - **S**: sistema
  - **I**: integrazione.
  - **U**: unità.
- **ID**: numero incrementale che rispetta una struttura gerarchica.

Le tabelle che raccolgono i test di una determinata tipologia presentano i campi:

- **Codice**: comprendente il codice identificativo del test.
- **Test**: descrive cosa il test deve verificare.
- **Stato**: indica lo stato del test e può essere:
  - **NI**: non implementato.

- **I**: implementato ma non ancora avviato.
- **NS**: avviato e fallito.
- **S**: avviato e superato.

Ad esempio:

Codice	Test	Stato
TV1	Verifica la segnalazione dell'apertura di una issue in Redmine ...	NI

4.4.7.2 Test d'integrazione

I test d'integrazione verificano il residuo mettendo insieme due unità. Queste unità in particolare sono considerati indipendenti, ma realizzate al fine di essere composte per collaborare. Il test di integrazione ha tanti test quanti ne servono per accertarsi che tutti i dati scambiati attraverso ciascuna interfaccia siano conformi alla loro specifica e accertarsi che tutti i flussi di controllo previsti siano stati effettivamente provati. La strategia che i test d'integrazione adottano è una strategia di tipo incrementale: si basa nella prosecuzione per passi, aggiungendo parti fino al completamento. Più test eseguiamo, più è piccola la parte che stiamo testando.
In una buona architettura, l'integrazione può anche essere parallelizzata. In particolare, la logica dell'integrazione funzionale segue i passi:

- Selezione delle funzionalità da integrare
- Identificazione delle varie componenti che svolgono quelle funzionalità
- Ordinamento delle componenti per numero di dipendenze crescenti
- Esecuzione dell'integrazione in quel determinato ordine

Inoltre rileva problemi quali:

- Errori residui nella realizzazione dei componenti
- Modifiche delle interfacce o cambiamenti nei requisiti
- Riuso di componenti dal comportamento oscuro o inadatto
- Integrazione con altre applicazioni non ben conosciute

4.4.7.2.1 MPS019 Test di integrazione

La qualità di un prodotto si verifica attraverso la percentuale di test di integrazione superati.
Metrica: percentuale di test di integrazione superati.

4.4.7.3 Test di unità

I test di unità sono l'attività di test di singole unità software. Per unità si intende il minimo componente di un programma dotato di funzionamento autonomo, ma non è semplice individuare l'unità. Va scelta durante l'attività di progettazione e deve essere sufficientemente piccola da facilitarne la verifica e la LIABILITY$_G$.

4.4.7.3.1 MPS020 Test di unità

La qualità di un prodotto si verifica attraverso la percentuale di test di unità superati.
Metrica: percentuale di test di unità superati.

4.4.7.4 Test di regressione

I test di regressione sono una ripetizione selettiva dei test di unità, integrazione e sistema. La utilizziamo per assicurarci che una modifica fatta su un'unità per correggere un problema, non faccia danno ad altre causando regressione. Ciò avviene quando il sistema è altamente accoppiato. La regressione si riduce invece riducendo l'accoppiamento, per esempio tramite INCAPSULAMENTO$_G$.

4.4.7.5 Tracciamento dei test

Tutti i test da noi delineati presentano un tracciamento in forma tabellare con i requisiti o componenti applicativi a loro associati. Questo ci assicura una verifica più accurata e completa.

4.4.8 Anomalie riscontrate

Durante i processi di verifica è possibile riscontrare a volte alcune anomalie. Tali anomalie posso essere dovute:

- A qualche valore preso come riferimento diverso da quello prestabilito per la metrica
- Al fallimento di un test

Ciò ci è utile perché un anomalia ci indica la presenza di errori nel prodotto o nel nostro way of working. Può quindi essere per noi un buon punto di inizio per la rivisitazione dei prodotti.

4.4.9 MPR013 Percentuale soddisfacimento metriche

Al momento della consegna del prodotto al cliente è bene fare un'analisi retrospettiva del lavoro svolto, in modo da capire se il gruppo ha lavorato bene o meno, in modo da migliorarsi per i progetti futuri.
Per calcolare tale metrica è necessario usare tutte le altre metriche:

$$\sum_{i=1} \frac{x_i}{y_i} 100$$

x_i = numero di volte che il valore della i-esima metrica è maggiore o uguale al valore della soglia minima e minore o uguale al valore della soglia massima.
y_i = numero di volte in cui abbiamo misurato la i-esima metrica.
Metrica: percentuale delle volte in cui l'insieme dei valori attesi di tutte le altre metriche vengono rispettati.

4.5 Validazione

4.5.1 Scopo

Il processo di validazione è una conferma finale che accerta la conformità di un prodotto alle attese. Essa fornisce una prova oggettiva di come le specifiche del prodotto siano conformi al suo scopo e alle esigenze degli utenti.

4.5.2 Descrizione

Questo processo serve per stabilire se è opportuno o meno procedere alla fase successiva del progetto, se ciò non fosse possibile sarà necessario il ritorno ad una fase stabile del progetto per poi ripartire da lì, prendendo in considerazione i risultati della precedente verifica.

4.5.3 Analisi dinamica

L'analisi dinamica valuta il comportamento dei prodotti in esecuzione, verificando se restituiscono i risultati attesi e se operano nel modo stabilito. Vengono presi in esame i prodotti software e i processi.
È oppurtono ricordare che i test devono essere:

- Ripetibili
- Automatizzati tramite strumenti
- Oggettivi e non personalizzati

Quindi, perché l'analisi dinamica avvenga in modo corretto, è necessario tenere conto di alcuni importanti elementi quali:

- **Ambiente di sviluppo**: il sistema hardware e software adottato durante il test.
- **Stato iniziale**: primo stato del prodotto, antecedente all'inizio del test.
- **Input**: dati inseriti.
- **Output**: dati attesi.
- **Notifica risultato**: notifiche riguardanti il risultato ottenuto dal test (opzionale).

Per effettuare analisi dinamica è necessario sviluppare dei dei test che verifichino l'integrità dei prodotti. Questi sono classificati attraverso un identificativo univoco descritto nel prossimo paragrafo e suddivisi per tipo in base alla tipologia dell'oggetto di test.

4.5.3.1 Classificazione dei test

Ogni test viene classificato identificato univocamente da un codice come già riportato in 4.4.7.1.

4.5.3.2 Test di sistema

I test di sistema hanno inizio quando sono completati i test d'integrazione e sono, per definizione, a scatola chiusa, nera, ovvero non è a disposizione il codice sorgente. Ci servono per dimostrare la conformità del prodotto, in particolare controllando che tutti i requisiti siano soddisfatti.

4.5.3.2.1 MPS018 Test di sistema

La qualità di un prodotto si verifica attraverso la percentuale di test di sistema superati.
Metrica: percentuale di test di sistema superati.

4.5.3.3 Test di accettazione

Il test di accettazione viene svolto una volta completato il prodotto, immediatamente prima del rilascio, dopo aver testato l'intero sistema. Se il richiedente lo considera superato, allora il prodotto può essere approvato e conseguentemente rilasciato.