AlphaSix

Norme di Progetto

Informazioni sul documento

Nome Documento	NormeDiProgetto v2.0.0.pdf
Versione	2.0.0
Data di Creazione	26 novembre 2018
Data ultima modifica	06 marzo 2019
Stato	Approvato
Redazione	Ciprian Voinea
Verifica	Laura Cameran Matteo Marchiori
Approvazione	Samuele Gardin
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
2.0.0	Approvazione per il rilascio	Responsabile	Samuele Gardin	2019-03-06
1.4.0	Verifica finale	Verificatore	Timoty Granziero	2019-03-04
1.3.5	Aggiunto §2.2.7	Amministratore	Nicola Carlesso	2019-03-04
1.3.4	Aggiunti §4.4.4.2, §4.4.5.4.1 e §3.1.6.3	Amministratore	Ciprian Voinea	2019-03-03
1.3.3	Completato §2.2.5	Amministratore	Matteo Marchiori	2019-03-01
1.3.2	Aggiunta immagine in §4.4.5.4.2 e completato paragrafo	Amministratore	Laura Cameran	2019-03-01
1.3.1	Aggiunta sezione §4.4.5.4 e sottosezioni	Amministratore	Laura Cameran	2019-02-28
1.3.0	Verifica	Verificatore	Laura Cameran	2019-02-27
1.2.5	Completato §2.2.4.4	Amministratore	Ciprian Voinea	2019-02-27
1.2.4	Completato §2.2.4.2	Amministratore	Ciprian Voinea	2019-02-25
1.2.3	Completato §2.2.4.3	Amministratore	Matteo Marchiori	2019-02-22
1.2.2	Completato §2.2.4.1	Amministratore	Matteo Marchiori	2019-02-22
1.2.1	Aggiunto §4.3	Amministratore	Ciprian Voinea	2019-02-20
1.2.0	Verifica	Verificatore	Samuele Gardin	2019-02-18
1.1.5	Completati paragrafi §4.4.6.5, §4.4.6.3 e §4.4.7	Amministratore	Nicola Carlesso	2019-02-16
1.1.4	Completati paragrafi §4.4.6.1, §4.4.6.2, §4.4.6.4	Amministratore	Laura Cameran	2019-02-16
1.1.3	Completato paragrafo §2.1.3	Amministratore	Laura Cameran	2019-02-11
1.1.2	Aggiunto §4.2.1.2	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 §4.4.6, aggiornato §2.2.8.1.2 e §4.4.7	Analista	Laura Cameran	2019-02-01
1.0.2	Aggiunto scheletro per §2.2.4.1, §2.2.4.3, §2.2.4.4, §2.2.5 e §2.2.6	Amministratore	Timoty Granziero	2019-01-29
1.0.1	Aggiornato lo stile redazionale	Amministratore	Timoty Granziero	2019-01-27
1.0.0	Approvazione per il rilascio	Responsabile	Nicola Carlesso	2019-01-10

Versione	Descrizione	Ruolo	Nominativo	Data
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
0.3.1	Aggiornamento §3.1	Verificatore	Laura Cameran	2019-01-05
0.3.0	Verifica documento	Verificatore	Ciprian Voinea	2018-12-29
0.2.6	Aggiunto §4.4	Verificatore	Nicola Carlesso	2018-12-10
0.2.5	Aggiunti §4.4.6 e §4.4.5	Verificatore	Nicola Carlesso	2018-12-08
0.2.4	Aggiunto §4.1.5	Amministratore	Timoty Granziero	2018-12-06
0.2.3	Aggiunto §2.2.4.5	Analista	Laura Cameran	2018-12-05
0.2.2	Aggiunto §2.2	Amministratore	Timoty Granziero	2018-12-04
0.2.1	Completata §2.2.4	Analista	Laura Cameran	2018-12-04
0.2.0	Verifica	Verificatore	Samuele Gardin	2018-12-04
0.1.6	Aggiunto §4.1.1	Amministratore	Timoty Granziero	2018-12-03
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 §4.4.5.1	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 §2.2.8.1	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	Redattore	Timoty Granziero	2018-11-22

Indice

1 Introduzione **2**
1.1 Glossario e documenti esterni . . . 2
1.2 Premessa . . . 2
1.3 Scopo del documento . . . 2
1.4 Scopo del prodotto . . . 2
1.5 Riferimenti . . . 2
1.5.1 Normativi . . . 2
1.5.2 Informativi . . . 3

2 Processi primari **5**
2.1 Processo di fornitura . . . 5
2.1.1 Scopo . . . 5
2.1.2 Ricerca delle tecnologie . . . 5
2.1.3 Normazione . . . 5
2.1.4 Studio di fattibilità . . . 5
2.1.5 Preparazione in vista della revisione . . . 5
2.2 Processo di sviluppo . . . 6
2.2.1 Scopo . . . 6
2.2.2 Analisi dei requisiti . . . 6
2.2.2.1 Denominazione dei requisiti . . . 6
2.2.2.2 Metriche sui requisiti . . . 7
2.2.2.2.1 MPR005 Requisiti obbligatori non soddisfatti . . . 7
2.2.2.2.2 MPR006 Requisiti desiderabili non soddisfatti . . . 7
2.2.2.2.3 MPR007 Requisiti opzionali non soddisfatti . . . 7
2.2.2.3 Casi d'uso . . . 7
2.2.2.3.1 Denominazione del codice identificativo . . . 8
2.2.3 Progettazione . . . 8
2.2.3.1 Obiettivi . . . 8
2.2.4 Diagrammi UML . . . 9
2.2.4.1 Diagrammi delle classi . . . 9
2.2.4.2 Diagrammi dei package . . . 10
2.2.4.3 Diagrammi di attività . . . 10
2.2.4.4 Diagrammi di sequenza . . . 11
2.2.4.5 Diagrammi dei casi d'uso . . . 12
2.2.5 Qualità della progettazione . . . 13
2.2.5.1 Qualità dei diagrammi delle classi . . . 13
2.2.5.2 Qualità dei diagrammi dei package . . . 13
2.2.5.3 Qualità dei diagrammi di attività . . . 13
2.2.5.4 Qualità dei diagrammi di sequenza . . . 13
2.2.6 Codifica . . . 14
2.2.6.1 Scopo . . . 14
2.2.6.2 Intestazione . . . 14
2.2.6.3 Formattazione . . . 14
2.2.6.4 Convenzioni di nominazione . . . 14
2.2.6.5 Indentazione . . . 15
2.2.6.6 Stringhe . . . 15
2.2.6.7 Lunghezza massima delle righe . . . 15
2.2.6.8 Lunghezza massima di metodi e funzioni . . . 15
2.2.6.9 Parametri delle funzioni . . . 15

2.2.6.10 Complessità ciclomatica . . . 16
2.2.6.11 Ereditarietà . . . 16
2.2.6.12 Commenti . . . 16
2.2.6.13 Documentazione dei metodi . . . 16
2.2.7 Metriche per la codifica . . . 17
2.2.7.1 MPS005 File senza intestazione . . . 17
2.2.7.2 MPS006 Righe non formattate . . . 17
2.2.7.3 MPS007 Nomi di variabili, metodi e classi non normati . . . 17
2.2.7.4 MPS008 Righe non indentate . . . 18
2.2.7.5 MPS009 Stringhe non normate . . . 18
2.2.7.6 MPS010 Righe troppo lunghe . . . 18
2.2.7.7 MPS011 Metodi troppo lunghi . . . 18
2.2.7.8 MPS012 Metodi con troppi parametri . . . 18
2.2.7.9 MPS013 Metodi con troppa complessità ciclomatica . . . 18
2.2.7.10 MPS014 Classi che implementano classi concrete . . . 18
2.2.7.11 MPS015 Commenti non normati . . . 18
2.2.7.12 MPS016 Metodi non documentati . . . 18
2.2.8 Strumenti di base . . . 19
2.2.8.1 Ambiente di sviluppo . . . 19
2.2.8.1.1 Sistema operativo . . . 19
2.2.8.1.2 IDE . . . 19

3 Processi organizzativi 20
3.1 Gestione del progetto . . . 20
3.1.1 Scopo . . . 20
3.1.2 Pianificazione della qualità . . . 20
3.1.2.1 Classificazione dei processi . . . 20
3.1.2.2 Classificazione degli obiettivi . . . 20
3.1.2.3 Classificazione delle metriche . . . 21
3.1.3 Pianificazione di attività . . . 21
3.1.3.1 Scopo . . . 21
3.1.3.2 Obiettivo . . . 22
3.1.3.3 Ordine di esecuzione . . . 22
3.1.3.4 Metriche di pianificazione . . . 23
3.1.3.4.1 MPR001 Varianza della pianificazione . . . 23
3.1.3.4.2 MPR002 Varianza dei costi . . . 23
3.1.3.5 Classificazione dei rischi . . . 23
3.1.3.5.1 MPR008 Rischi non previsti avvenuti . . . 25
3.1.3.6 Ruoli di progetto . . . 25
3.1.4 Monitoraggio del progetto . . . 25
3.1.4.1 Monitoraggio dell'esecuzione dei processi . . . 25
3.1.4.2 Procedure di comunicazione . . . 25
3.1.4.3 Gestione dei problemi emersi . . . 26
3.1.4.4 Monitoraggio delle ore di orologio . . . 26
3.1.5 Chiusura dei processi . . . 26
3.1.5.1 Archiviazione dei prodotti . . . 26
3.1.5.2 Archiviazione delle misurazioni . . . 26
3.1.6 Strumenti organizzativi . . . 27
3.1.6.1 Slack . . . 27
3.1.6.2 GitHub . . . 27
3.1.6.3 Toggl . . . 28
3.2 Formazione . . . 28

3.2.1 Piano di formazione . . . 28

4 Processi di supporto **29**
4.1 Documentazione . . . 29
4.1.1 Implementazione . . . 29
4.1.1.1 Template . . . 29
4.1.1.2 Ciclo di vita dei documenti . . . 29
4.1.2 Struttura . . . 29
4.1.2.1 Frontespizio . . . 29
4.1.2.2 Storico delle versioni . . . 30
4.1.2.3 Indice . . . 30
4.1.2.4 Contenuto . . . 30
4.1.3 Design . . . 30
4.1.3.1 Norme tipografiche . . . 30
4.1.3.1.1 Stile redazionale . . . 31
4.1.3.1.2 Stile del testo . . . 31
4.1.3.1.3 Elenchi puntati . . . 31
4.1.3.1.4 Altri formati testuali comuni . . . 31
4.1.3.2 Elementi grafici . . . 31
4.1.3.2.1 Figure . . . 31
4.1.3.2.2 Tabelle . . . 32
4.1.4 Produzione . . . 32
4.1.4.1 Suddivisione dei documenti . . . 32
4.1.4.1.1 Documenti interni . . . 32
4.1.4.1.2 Documenti esterni . . . 32
4.1.4.1.3 Verbali . . . 32
4.1.4.2 Strumenti di supporto . . . 32
4.1.4.2.1 LaTeX . . . 32
4.1.4.2.2 Google Drive . . . 33
4.1.4.2.3 TexStudio/Visual Studio Code . . . 33
4.1.4.2.4 GanttProject . . . 33
4.1.4.2.5 Draw.io . . . 33
4.1.4.2.6 Indice di Gulpease . . . 33
4.1.4.2.7 Controllo ortografico . . . 33
4.1.4.2.8 Glossarizzazione dei ternimi . . . 34
4.1.5 Mantenimento . . . 34
4.1.5.1 Continuous Integration . . . 34
4.1.5.2 Nomenclatura . . . 34
4.1.5.2.1 Verbali . . . 34
4.1.5.2.2 Documenti vari . . . 34
4.2 Codice sorgente . . . 34
4.2.1 Mantenimento . . . 34
4.2.1.1 Continuous Integration . . . 34
4.2.1.1.1 Jenkins . . . 34
4.2.1.2 Docker . . . 35
4.3 Configurazione . . . 35
4.3.1 Descrizione . . . 35
4.3.2 Versionamento . . . 35
4.3.3 Struttura delle Repository . . . 36
4.3.4 Norme di branching . . . 36
4.3.5 Aggiornamento della repository . . . 37
4.4 Verifica . . . 38

4.4.1 Scopo . . . 38
4.4.2 Descrizione . . . 38
4.4.3 Walkthrough e Inspection . . . 38
4.4.3.1 Walkthrough . . . 38
4.4.3.2 Inspection . . . 38
4.4.4 Metodologie di sviluppo del software . . . 39
4.4.4.1 The Twelve-Factor App . . . 39
4.4.4.2 Test Driven Development . . . 40
4.4.5 Analisi statica . . . 40
4.4.5.1 Analisi dei documenti . . . 41
4.4.5.1.1 MPD001 Indice di Gulpease . . . 41
4.4.5.1.2 MPD002 Correttezza ortografica . . . 41
4.4.5.2 Analisi dei processi . . . 41
4.4.5.3 MPR003 Aderenza agli standard . . . 42
4.4.5.3.1 MPR004 Frequenza commit nella repository . . . 42
4.4.5.3.2 MPR009 Frequenza controllo prodotti . . . 42
4.4.5.4 Analisi del software . . . 42
4.4.5.4.1 Controllo del Python coding style . . . 43
4.4.5.4.2 SonarQube . . . 43
4.4.5.4.3 MPS001 Presenza di bug . . . 43
4.4.5.4.4 MPS002 Presenza di vulnerabilità . . . 43
4.4.5.4.5 MPS003 Presenza di code smell . . . 44
4.4.5.4.6 MPS004 Duplicazione del codice . . . 44
4.4.6 Analisi dinamica . . . 44
4.4.6.1 Test di sistema . . . 44
4.4.6.2 Test d'integrazione . . . 44
4.4.6.3 Test di unità . . . 45
4.4.6.4 Test di regressione . . . 45
4.4.6.5 Test di accettazione . . . 45
4.4.7 Anomalie riscontrate . . . 45

Elenco delle tabelle

1 Metrica Indice di Gulpease 21
2 Costo orario per ruolo 23

Elenco delle figure

1 Interfaccia grafica di PyCharm 19
2 Albero dei branch da Gitkraken 37
3 Test Driven Development 40
4 Screen SonarQube 43

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 v2.0.0*$_{D}$).
- **G**: indica che il termine si riferisce ad una voce riportata nel Glossario (ad esempio Redmine$_{G}$).

1.2 Premessa

Il documento$_{G}$ che segue verrà prodotto incrementalmente al presentarsi della necessità di redigere nuove norme$_{G}$. Per questo motivo, non è da considerare al pari di un documento completo (e.g. la parte relativa alla codifica non ci sarà fino al presentarsi di tale necessità).

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

- ISO 8601
  - https://it.wikipedia.org/wiki/ISO_8601#Data_completa
  - https://it.wikipedia.org/wiki/ISO_8601#Orari
- ISO/IEC 12207
  https://en.wikipedia.org/wiki/ISO/IEC_12207
- Composition over inheritance
  https://en.wikipedia.org/wiki/Composition_over_inheritance
- Formato della data
  https://it.wikipedia.org/wiki/Formato_della_data

- The Twelve-Factor App$_G$
  https://12factor.net/#the_twelve_factors
- PEP 8$_G$
  https://www.python.org/dev/peps/pep-0008

1.5.2 Informativi

- Apache Kafka$_G$
  https://kafka.apache.org/documentation/
- Descrizione dei ruoli di progetto
  https://www.math.unipd.it/~tullio/IS-1/2018/Progetto/RO.html
- Docker$_G$
  https://docs.docker.com/
- Fonte Figura 3:
  https://en.wikipedia.org/wiki/Test-driven_development
- GanttProject
  https://www.ganttproject.biz/
- GitLab$_G$
  https://docs.gitlab.com/ee/
- GNU Aspell
  http://aspell.net/
- Google Drive
  https://www.google.com/drive/
- LaTeX
  https://www.latex-project.org/help/documentation/
- Naming convention
  https://en.wikipedia.org/wiki/Naming_convention_(programming)
- PyCharm
  https://www.jetbrains.com/pycharm/
- Pytest
  https://docs.pytest.org/en/latest/
- Redmine
  https://www.redmine.org/guide
- Software Engineering (Ian Sommerville), 10° edizione, Pagina 648, Capitolo 22: Project management
- Telegram
  https://core.telegram.org/
- TexStudio
  https://www.texstudio.org/
- Toggl
  https://toggl.com/

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

2 Processi primari

2.1 Processo di fornitura

2.1.1 Scopo

La sezione corrente ha lo scopo di riportare le attività principali che impegneremo ad attuare al fine di diventare FORNITORI$_G$ per la proponente Imola Informatica.

2.1.2 Ricerca delle tecnologie

Ognuno di noi approfondirà la propria conoscenza su tecnologie e FRAMEWORK$_G$ vari, in modo da discutere insieme quali sono quelle di interesse comune e di utilità per il progetto. Questo prevede poi 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, cambiare il codice identificativo di ognuno e cambiare la relativa norma nel paragrafo §2.2.2.3.

2.1.4 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.

2.1.5 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 dei package e il codice necessario per la realizzazione del PROOF OF CONCEPT$_G$
- Le diapositive per l'esposizione

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[1] 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 v2.0.0*$_D$, 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.
- **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.

[1] Riferirsi alla voce "ISO/IEC 12207" in §1.5.1

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.

2.2.3 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 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.

2.2.4 Diagrammi UML

2.2.4.1 Diagrammi delle classi

Per i diagrammi delle classi utilizziamo lo strumento Draw.io$_G$ e il linguaggio UML$_G$ 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.
- **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...).

2.2.4.2 Diagrammi dei package

Per i diagrammi dei package utilizziamo lo strumento Draw.io$_G$ e il linguaggio UML$_G$ 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)

2.2.4.3 Diagrammi di attività

Per i diagrammi di attività utilizziamo lo strumento Draw.io$_G$ e il linguaggio UML$_G$ 2.0.
Li usiamo per descrivere la logica dell'applicazione, ovvero quali attività si possono presentare e in quali sequenze vengono svolte. Rappresentano quindi la parte dinamica dei casi d'uso.
Le attività sono insiemi di azioni ordinate dei tipi:

- **Nodo iniziale**: rappresentato da un pallino pieno, è l'inizio dell'attività.
- **Fork**: rappresentato da un tratto marcato, indica l'inizio di elaborazioni parallele.
- **Join**: rappresentato da un tratto marcato, indica un punto di sincronizzazione tra elaborazioni parallele.

  Vengono indicate specifiche di join tra parentesi quadre (espressioni booleane).

- **Branch**: rappresentato da un rombo vuoto, indica che si può intraprendere solo uno dei possibili percorsi.
- **Merge**: rappresentato da un rombo vuoto, è simile alla join. Differisce per il fatto che unisce più possibili elaborazioni non parallele ma in alternativa tra esse, porvenienti da uno o più branch.
- **Pin parametro**: rappresentati da box, sono parametri che rappresentano risorse prodotte da azioni e consumate da altre azioni.
- **Nodo finale**: rappresentato da un pallino pieno circondato da un cerchio, indica il termine dell'attività con successo.

- **Nodo di fine flusso**: rappresentato da un cerchio con una croce, non termina l'attività.

  Viene usata per casi speciali (eccezione, ramo del branch...).

Ogni attività può comporre un'azione. In tal caso si tratta di una sotto attività, e viene indicata con un simbolo a forca e da un rettangolo esterno con il nome dell'attività che ne rappresenta le azioni dall'input fino all'output.
Usiamo inoltre i seguenti costrutti:

- **Swimlanes**: sono rappresentate da riquadri che suddividono la responsabilità delle azioni presenti nelle attività.

  Le azioni possono far capo a più swimlanes, per esempio l'azione di merge.

- **Segnali**: indicano eventi provenienti da processi esterni.

  Sono indicati in modo diverso a seconda dell'evento scatenante, che può essere dei seguenti tipi:

  - **Eventi esterni**: l'evento generato viene rappresentato da un rettangolo con un lato a punta uscente, mentre l'azione che attende l'evento viene rappresentata con un rettangolo con un lato a punta entrante.
  - **Eventi temporali**: viene rappresentato con una clessidra e ne viene indicato il tempo di ripetizione.

    Può trovarsi all'inizio del flusso, sostituendo il nodo iniziale, per indicare che un'attività verrà eseguita ogni volta che l'evento temporale consuma il tempo indicato.

Per indicare azioni che si ripetono su più elementi, ad esempio su una lista di elementi, usiamo le regioni di espansione. Le indichiamo con un rettangolo tratteggiato ad angoli arrotondati, e rettangoli piccoli presenti sui lati di inizio e fine regione, che rappresentano le liste di elementi processati.
Le frecce di entrata e uscita indicano in quale verso viene svolta l'attività che include la regione di espansione.

2.2.4.4 Diagrammi di sequenza

Per i diagrammi di sequenza utilizziamo lo strumento Draw.io$_G$ e il linguaggio UML$_G$ 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.

2.2.4.5 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.

2.2.5 Qualità della progettazione

Per garantire una progettazione che rispetti gli standard di qualità posti nel *PianoDiQualifica v2.0.0*$_{\text{D}}$, seguiamo le linee guida date dalle best practice.

2.2.5.1 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.

2.2.5.2 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.

2.2.5.3 Qualità dei diagrammi di attività

Poiché descrivono gli aspetti dinamici dei casi d'uso, non devono risultare eccesivamente complessi, ma descrivere quanto riportato da casi d'uso e requisiti.
Li usiamo in modo da aggiungere comprensibilità, non per toglierne.

2.2.5.4 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.

2.2.6 Codifica

2.2.6.1 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 perlopiù usando il linguaggio Python$_G$ (per una parte sarà valutato se utilizzare Ruby). Per le norme di stile relative a questo linguaggio, seguiremo il PEP 8[2], lo standard Python che definisce lo stile da seguire nella codifica e in attività correlate ad essa.

2.2.6.2 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>
....
"""
```

2.2.6.3 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
```

2.2.6.4 Convenzioni di nominazione

- Per i nomi delle classi, viene usato `PascalCase`[3]
- 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.

[2]Riferirsi alla voce "PEP 8" in §1.5.1
[3]Riferirsi alla voce "Naming convention" 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
```

2.2.6.5 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.

2.2.6.6 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!'
```

2.2.6.7 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())
```

2.2.6.8 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.

2.2.6.9 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])
```

2.2.6.10 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.

2.2.6.11 Ereditarietà

Evitare l'ereditarietà tra classi che hanno stato. È invece possibile che una classe derivi da un'altra se quest'ultima è a tutti gli effetti un'INTERFACCIA$_G$, ossia definisce solo comportamenti astratti (i.e. è composta solo da metodi astratti).
Viene adottato, per ovviare alla assente ereditarietà, il principio di "Composition over inheritance"[4].

2.2.6.12 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.

2.2.6.13 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 §2.2.6.8), 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
    ...
```

[4] Riferirsi alla voce "Composition over inheritance" in §1.5.1.

2.2.7 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**: §2.2.6.2.
- **MPS006 Righe non formattate**: §2.2.6.3.
- **MPS007 Nomi di variabili, metodi e classi non normati**: §2.2.6.4.
- **MPS008 Righe non indentate**: §2.2.6.5.
- **MPS009 Stringhe non normate**: §2.2.6.6.
- **MPS010 Righe troppo lunghe**: §2.2.6.7.
- **MPS011 Metodi troppo lunghi**: §2.2.6.8.
- **MPS012 Metodi con troppi parametri**: §2.2.6.9.
- **MPS013 Metodi con troppa complessità ciclomatica**: §2.2.6.10.
- **MPS014 Classi che implementano classi concrete**: §2.2.6.11.
- **MPS015 Commenti non normati**: §2.2.6.12.
- **MPS016 Metodi non documentati**: §2.2.6.13.

2.2.7.1 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.

2.2.7.2 MPS006 Righe non formattate

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

2.2.7.3 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.

2.2.7.4 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.

2.2.7.5 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.

2.2.7.6 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.

2.2.7.7 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.

2.2.7.8 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.

2.2.7.9 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.

2.2.7.10 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.

2.2.7.11 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.

2.2.7.12 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.

2.2.8 Strumenti di base

2.2.8.1 Ambiente di sviluppo

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

2.2.8.1.1 Sistema operativo

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

2.2.8.1.2 IDE

Gli IDE principali scelti per lo sviluppo di codice Python sono PyCharm[5] e Visual Studio Code[6]. 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

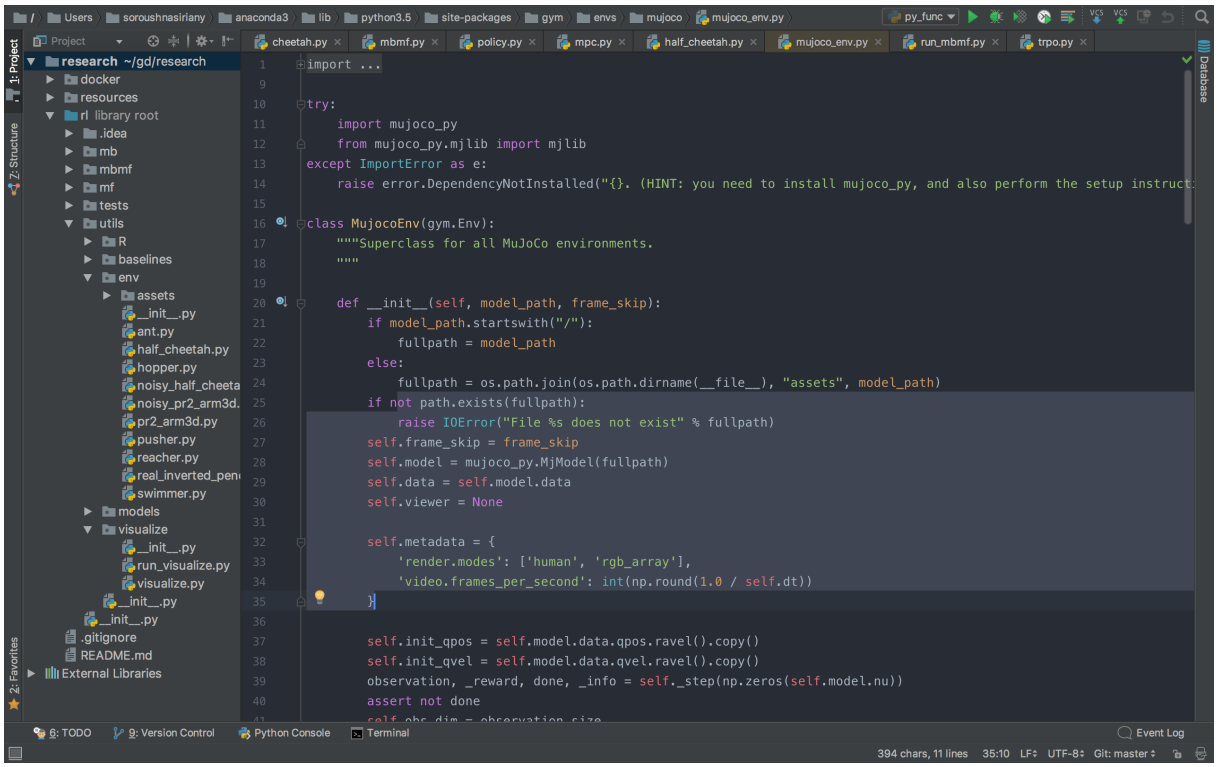

Figura 1: Interfaccia grafica di PyCharm

[5]Riferirsi alla voce "Pycharm" in §1.5.2
[6]Riferirsi alla voce "Visual Studio Code" 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 v2.0.0*$_D$, o standard di qualità, descritti nel *PianoDiQualifica v2.0.0*$_D$.
Gli obiettivi che ci si pone di raggiungere sono descritti nel *PianoDiQualifica v2.0.0*$_D$.

3.1.2.1 Classificazione dei processi

I processi analizzati nel *PianoDiQualifica v2.0.0*$_D$ 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 Piano di Qualifica, 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 Piano di Progetto 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 Piano di Progetto 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 v2.0.0*$_D$ 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 v2.0.0*$_D$, 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 v2.0.0*$_D$ 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[7] e possono essere di tipo:

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

[7] Riferirsi alla voce "Software Engineering, 10° edizione" 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 v2.0.0*$_{D}$, 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[8] 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 v2.0.0*$_{D}$ e nell'*AnalisiDeiRequisiti v2.0.0*$_{D}$.
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$_{G}$.

[8] Riferirsi alla voce "Descrizione dei ruoli di progetto" 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.

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 v2.0.0*$_D$. 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 Analisi dei Requisiti. 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 Piano di Progetto.
- # **Piano di Qualifica**: per discussioni inerenti principalmente la qualità di processo e di prodotto, riportate nel Piano di Qualifica.
- # **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 Studio di Fattibilità.
- # **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.
- # **Proof of Concept**: per discussioni inerenti alla realizzazione del Proof of Concept.

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[9], 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[10]:
  - 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
  - CherryPy
- Possibilità d'integrazione tra le tecnologie sopra elencate

[9] Riferirsi alla voce "Toggl" in 1.5.2 per maggiori informazioni
[10] 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 §4.3.2.
- **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 §4.3.2
- 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[11] 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.

[11] Riferirsi alla voce "ISO 8601" in §1.5.1

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.

4.1.4 Produzione

4.1.4.1 Suddivisione dei documenti

4.1.4.1.1 Documenti interni

Le Norme di Progetto e lo Studio di Fattibilità 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 Piano di Progetto, il Piano di Qualifica, il Glossario e l'Analisi dei Requisiti 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...
- **Agenda**: lista dei principali argomenti trattati, con breve descrizione.
- **Riepilogo**: 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.

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 §4.4.5.1.1.

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 §4.4.5.1.2.

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 v2.0.0*$_D$. Per automatizzare questo processo è stato creato uno script che valuta le voci all'interno di *Glossario v2.0.0*$_D$ e ed inserisce i riferimenti a queste voci alla prima occorrenza di ogni parola in tutti i documenti prodotti.

4.1.5 Mantenimento

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.

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 §4.3.2

4.2 Codice sorgente

4.2.1 Mantenimento

4.2.1.1 Continuous Integration

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

4.2.1.1.1 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[12].
- **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.

4.2.1.2 Docker

Per rendere il sistema Butterfly completamente isolato dal sistema circostante, creiamo dei container specifici per ogni componente. Per fare ciò, utilizziamo DOCKER$_G$[13], 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.

4.3 Configurazione

4.3.1 Descrizione

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.

4.3.2 Versionamento

Tutti i documenti redatti e i file sorgente supportano il versionamento, in modo da essere univoci e per 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

[12] Riferirsi alla voce "Pytest" in §1.5.2
[13] Riferirsi alla voce "Docker" in §1.5.2.

- `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`

4.3.3 Struttura delle Repository

Abbiamo creato due repository principali fino a questo momento:

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

4.3.4 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.

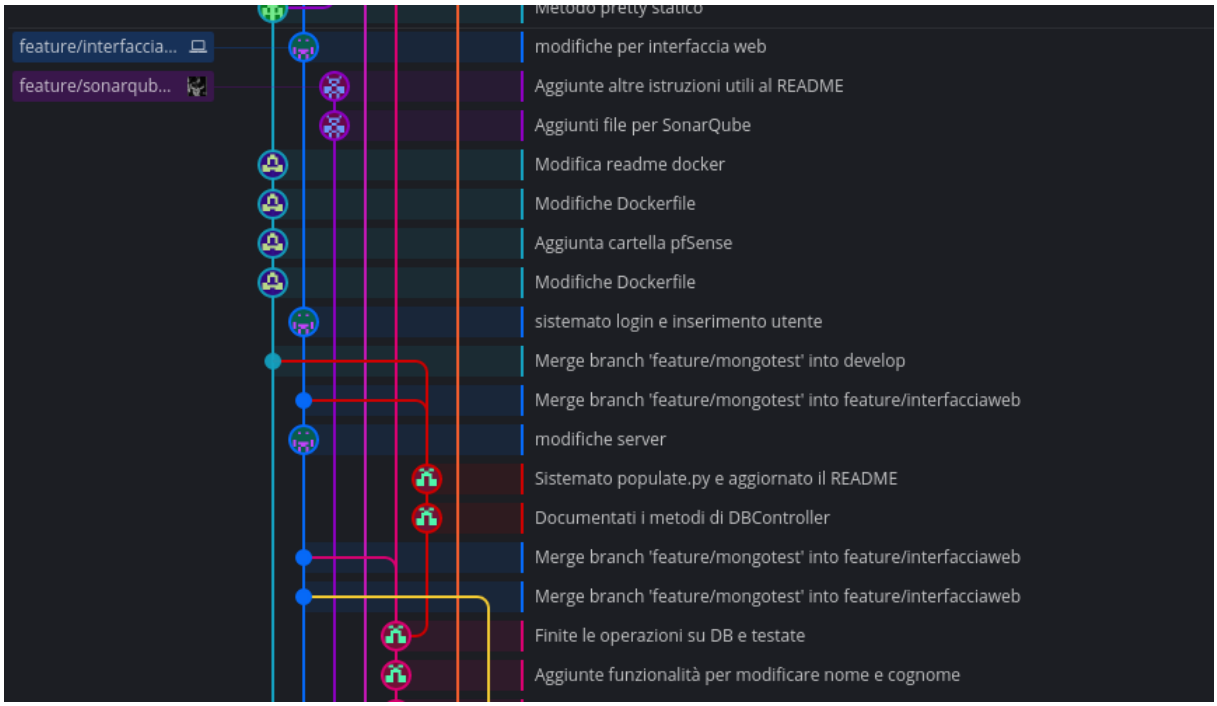

Figura 2: Visione generale dell'albero dei branch in un momento casuale da Gitkraken

4.3.5 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
```

4.4 Verifica

4.4.1 Scopo

Questa sezione vuole descrivere come eseguiamo la fase di verifica, per capire se i prodotti e i processi sono conformi a quanto ci attendiamo. La fase di verifica 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.

4.4.2 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 *PianoDiQualifica v2.0.0*$_D$, 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 Piano di Qualifica: in questo modo, confrontandoli con gli esiti attesi, è possibile valutare un miglioramento per i vari processi e prodotti sempre nel Piano di Qualifica.

4.4.3 Walkthrough e Inspection

Due modi di effettuare la verifica sono attraverso Walkthrough$_G$ 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 Metodologie di sviluppo del software

4.4.4.1 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.

4.4.4.2 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 v2.0.0*$_{\text{D}}$ e al contempo superi il test.

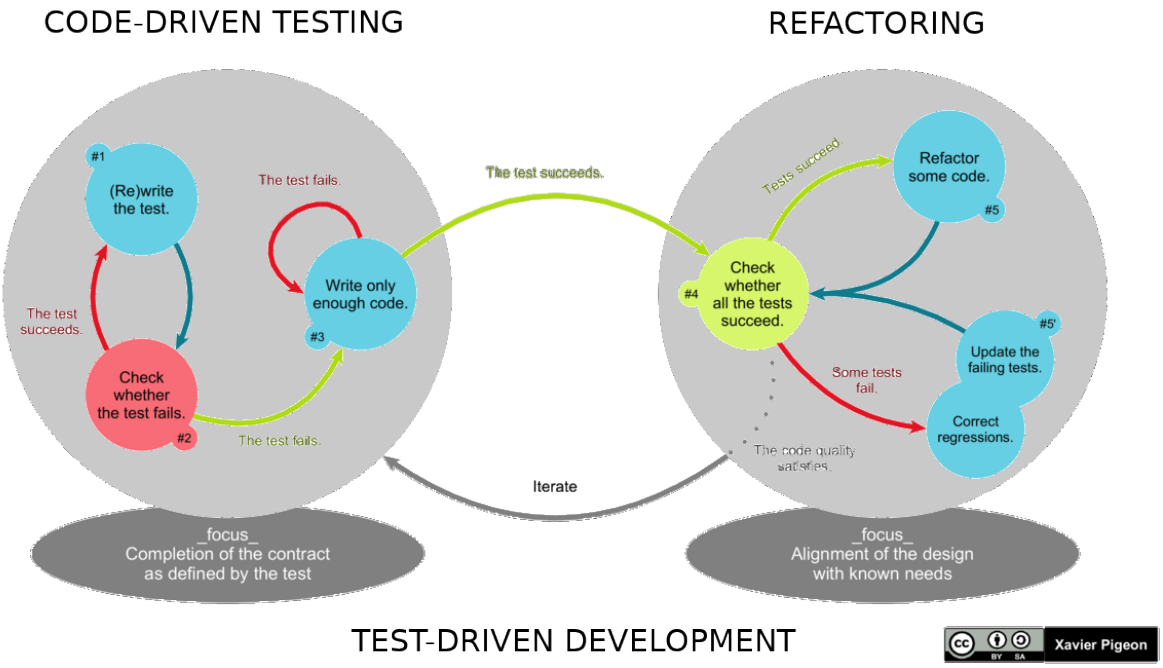

Figura 3: Test Driven Development[14]

4.4.5 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.

[14]Riferirsi alla voce "Fonte Figura 3" in §1.5.2

4.4.5.1 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.

4.4.5.1.1 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.

4.4.5.1.2 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.

4.4.5.2 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

La denominazione delle metriche è descritta in §3.1.2.3.

4.4.5.3 MPR003 Aderenza agli standard

Per misurare e verificare i processi sono stati scelti alcuni specifici standard di qualità descritti nel Piano di Qualifica 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 Piano di Qualifica.

4.4.5.3.1 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.

4.4.5.3.2 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.

4.4.5.4 Analisi del software

L'analisi statica del codice, insieme alle norme di codifica stabilite in §2.2.6, 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

La denominazione delle metriche è descritta in §3.1.2.3.

4.4.5.4.1 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.

4.4.5.4.2 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.

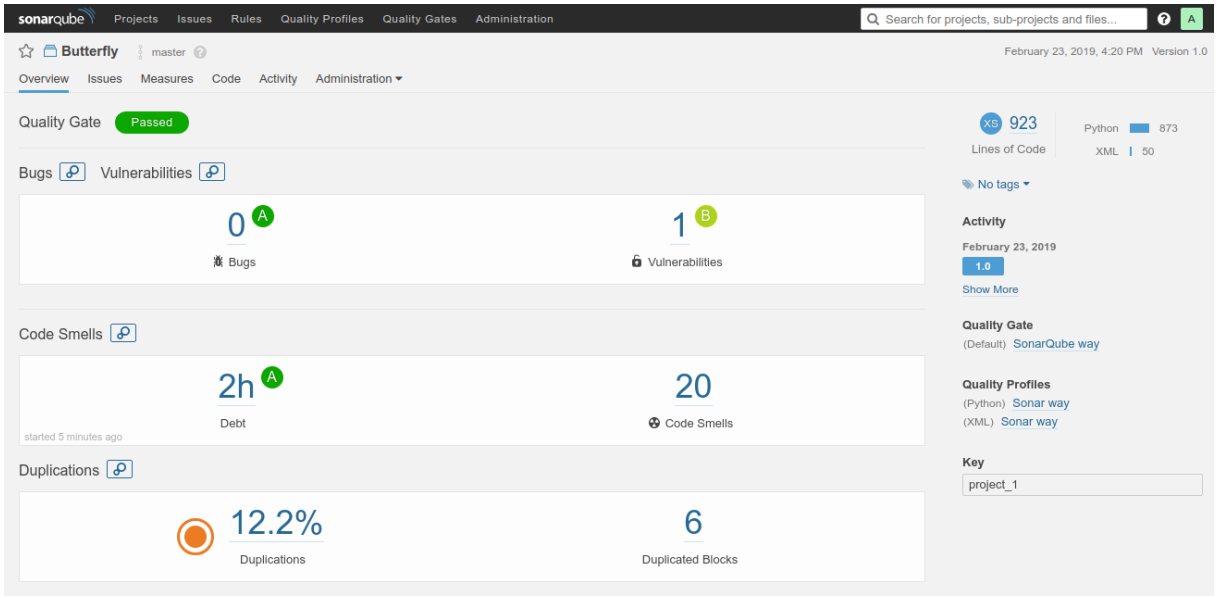

Figura 4: Esempio di immagine catturata del progetto Butterfly in SonarQube.

4.4.5.4.3 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.

4.4.5.4.4 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.

4.4.5.4.5 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.

4.4.5.4.6 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 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).

4.4.6.1 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.

4.4.6.2 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

4.4.6.3 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$_G$.

4.4.6.4 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$_G$.

4.4.6.5 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.

4.4.7 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.