Norme di Progetto

SWEefty - da mettere

Informazioni sul documento

Versione	1.0.0
Redazione	Alberto Galinaro Davide Zago Elia Montecchio Francesco Parolini Giuseppe Merlino Lisa Parma Paolo Eccher
Verifica	XXX
Approvazione	YYY
Uso	Interno/esterno
Distribuzione	Prof.Tullio Vardanega Prof.Riccardo Cardin SWEefty

Descrizione

Questo documento descrive le regole, gli strumenti e le convenzioni adottate dal gruppo SWEefty durante la realizzazione del progetto etctectetc.

Diario delle modifiche

Modifica	Autore	Ruolo	Data	Versione
Scrittura dei processi organizzativi	Francesco Parolini	Amministratore	2017-12-13	0.6.0
Scrittura della sezione Verifica dei processi di supporto	Davide Zago	Amministratore	2017-12-11	0.5.0
Scrittura della sezione Strumenti dei processi di supporto	Giuseppe Merlino	Amministratore	2017-12-10	0.4.0
Scrittura della sezione Documentazione	Paolo Eccher	Amministratore	2017-12-10	0.3.0
Ampliata la sezione Fornitura	Elia Montecchio	Amministratore	2017-12-09	0.2.1
Scrittura della sezione Progettazione	Lisa Parma	Amministratore	2017-12-08	0.2.0
Scrittura della sezione Fornitura e Codifica	Alberto Gallinaro	Amministratore	2017-12-07	0.1.0
Prima stesura dello scheletro del documento	Francesco Parolini	Project manager	2017-12-04	0.0.1

Indice

1 Introduzione **5**
1.1 Scopo del documento 5
1.2 Scopo del prodotto 5
1.3 Ambiguità 5
1.4 Riferimenti 5
1.4.1 Normativi 5
1.4.2 Informatici 5

2 Processi primari **5**
2.1 Fornitura 5
2.1.1 Scopo 5
2.1.2 Attività 6
2.1.2.1 Studio di fattibilità 6
2.1.2.2 Piano di Progetto 6
2.1.2.3 Piano di Qualifica 6
2.2 Sviluppo 7
2.2.1 Scopo 7
2.2.2 Aspettative 7
2.2.3 Descrizione 7
2.2.4 Attività 7
2.2.4.1 Analisi dei requisiti 7
2.2.4.1.1 Scopo 7
2.2.4.1.2 Aspettative 8
2.2.4.1.3 Descrizione 8
2.2.4.1.4 Casi d'uso 8
2.2.4.1.5 Codice identificativo 8
2.2.4.1.6 Requisiti 8
2.2.4.1.7 Codice identificativo 8
2.2.4.1.8 UML 8
2.2.4.2 Progettazione 8
2.2.4.2.1 Scopo 8
2.2.4.2.2 Specifica Tecnica 9
2.2.4.2.3 Definizione di Prodotto 9
2.2.4.3 Codifica 10
2.2.4.3.1 Scopo 10
2.2.4.3.2 Stile di codifica 10
2.2.4.3.3 Intestazione 12
2.2.4.3.4 Versionamento 12
2.2.4.3.5 Ricorsione 13
2.2.5 Strumenti 13
2.2.5.1 StarUML 13
2.2.5.2 SWEgo 14
2.2.5.3 Atom 14

3 Processi Organizzativi . . . 16
3.1 Gestione . . . 16
3.1.1 Scopo . . . 16
3.1.2 Ruoli di progetto . . . 16
3.1.2.1 Amministratore di Progetto . . . 16
3.1.2.2 Analista . . . 16
3.1.2.3 Progettista . . . 17
3.1.2.4 Verificatore . . . 17
3.1.2.5 Programmatore . . . 17
3.1.2.6 Project Manager . . . 17
3.1.3 Procedure . . . 18
3.1.3.1 Gestione delle comunicazioni . . . 18
3.1.3.1.1 Comunicazioni interne . . . 18
3.1.3.1.2 Comunicazioni esterne . . . 18
3.1.3.2 Gestione degli incontri . . . 18
3.1.3.2.1 Incontri interni . . . 18
3.1.3.2.2 Incontri esterni . . . 19
3.1.3.3 Gestione degli strumenti di coordinamento . . . 19
3.1.3.3.1 Ticketing . . . 19
3.1.3.3.2 Struttura del worskspace di comunicazioni interne . . . 19
3.1.3.4 Gestione degli strumenti di versionamento . . . 21
3.1.3.4.1 Repository . . . 21
3.1.3.4.2 Struttura del repository . . . 21
3.1.3.4.3 Norme sui nomi dei files . . . 22
3.1.3.4.4 Estensioni ammesse . . . 22
3.1.3.4.5 Norme di branching e merging . . . 22
3.1.3.4.6 Norme su commit . . . 23
3.1.3.5 Gestione dei rischi . . . 23
3.1.4 Strumenti . . . 23
3.1.4.1 Sistema Operativo . . . 23
3.1.4.2 Slack . . . 23
3.1.4.3 Wrike . . . 24
3.1.4.4 Github . . . 24
3.1.4.5 Git . . . 24
3.1.4.6 Gitkraken . . . 24

4 Processi di Supporto . . . 24
4.1 Documentazione . . . 24
4.1.1 Scopo . . . 24
4.1.2 Procedure . . . 25
4.1.2.1 Approvazione dei documenti . . . 25
4.1.3 Template . . . 25
4.1.4 Struttura dei documenti . . . 26
4.1.4.1 Prima pagina . . . 26
4.1.4.2 Registro delle modifiche . . . 26
4.1.4.3 Indici . . . 26

4.1.4.4 Formattazione generale delle pagine . . . 27
4.1.4.4.1 Intestazione . . . 27
4.1.4.4.2 Piè di pagina . . . 27
4.1.4.5 Note a piè di pagina . . . 27
4.1.5 Versionamento . . . 27
4.1.6 Norme tipografiche . . . 28
4.1.6.1 Stile del testo . . . 28
4.1.6.2 Elenchi puntati . . . 28
4.1.6.3 Formati comuni . . . 28
4.1.6.4 Sigle . . . 29
4.1.7 Elementi grafici . . . 29
4.1.7.1 Tabelle . . . 29
4.1.7.2 Immagini . . . 30
4.1.8 Classificazione dei documenti . . . 30
4.1.8.1 Documenti informali . . . 30
4.1.8.2 Documenti formali . . . 30
4.1.8.3 Verbali . . . 30
4.1.9 Strumenti . . . 30
4.1.9.1 LaTeX . . . 30
4.1.9.2 TexStudio . . . 30
4.1.9.3 Lucidchart . . . 30
4.2 Verifica . . . 30
4.2.1 Scopo . . . 30
4.2.2 Aspettative . . . 30
4.2.3 Descrizione . . . 30
4.2.4 Attività . . . 30
4.2.4.1 Analisi . . . 30
4.2.4.1.1 Analisi Statica . . . 30
4.2.4.1.2 Analisi dinamica . . . 31
4.2.4.2 Test . . . 31
4.2.4.2.1 Test di unità . . . 31
4.2.4.2.2 Test di integrazione . . . 31
4.2.4.2.3 Test di sistema . . . 32
4.2.4.2.4 Test di regressione . . . 32
4.2.4.2.5 Test di accettazione . . . 32
4.2.5 Strumenti . . . 32
4.2.5.1 Verifica ortografica . . . 32
4.2.5.2 Analisi statica . . . 32
4.2.5.3 Analisi dinamica . . . 32
4.2.5.4 Metriche . . . 32

1 Introduzione

1.1 Scopo del documento

Il seguente documento ha l'obiettivo di esplicitare le norme, le convenzioni e la strumentazione che sarà adottata dal gruppo SWEefty durante l'intero svolgimento del progetto. Con questa prospettiva deve essere visionato da tutti i membri del gruppo, i quali sono tenuti ad osservare quanto scritto per mantenere consistenza ed omogeneità in ogni aspetto del ciclo di vita del software che sarà prodotto durante il progetto.

1.2 Scopo del prodotto

Lo scopo del *prodotto*$_G$ è sviluppare dei plugin Kibana in attraverso la libreria Javascript D3.

1.3 Ambiguità

Per scongiurare ogni malinteso ed ogni ambiguità nella terminologia impiegata viene fornito il *Glossario*, il quale contiene le definizioni dei termini in corsivo con una G a pedice.

1.4 Riferimenti

1.4.1 Normativi

- **Verbale di incontro interno:**
- **Verbale di incontro esterno:**

1.4.2 Informatici

eventuali guide latex, git, però devono essere serie....

2 Processi primari

2.1 Fornitura

2.1.1 Scopo

Lo scopo di questo *processo*$_G$ è di trattare i termini e le norme, dalle più triviali alle più importanti, che tutti i componenti del gruppo SWEefty sono tenuti a rispettare per diventare fornitori dell'azienda IKS s.r.l e dei committenti Prof. Tullio Vardanega e Prof. Riccardo Cardin.

2.1.2 Attività

2.1.2.1 Studio di fattibilità

Sono state organizzate diverse riunioni al fine di permettere ai membri del team di scambiarsi opinioni sui capitolati proposti. Abbiamo valutato il capitolato secondo diversi criteri:

- **Dominio tecnologico:** E' stata presa in considerazione la conoscenza attuale delle tecnologie richieste per portare a termine i vari capitolati, in base anche a esperienze passate avute con problemi simili.
- **Individuazione rischi:** sono state analizzate le varie difficoltà che si potrebbero incontrare durante la realizzazione dei vari capitolati considerando in modo particolare la mancanza di conoscenze adeguate relative alle tecnologie necessarie per realizzare i capitolati.

2.1.2.2 Piano di Progetto

Il *Project Manager* affiancato dall' *Amministratore* dovrà stilare un piano da seguire per la realizzazione del progetto. Il documento dovrà rispettare e seguire i seguenti punti:

- **Analisi dei rischi:** si analizzano nel dettaglio i rischi che si potrebbero presentare durante lo svolgimento del progetto, capendo con quale probabilità potrebbero accadere e qual è il livello di gravità di ogni rischio.
- **Pianificazione:** si pianificano le attività da svolgere nel corso del progetto fornendo delle scadenze temporali precise.
- **Preventivo:** si stima, secondo la pianificazione, la quantità di lavoro necessaria per portare a termine ogni fase, in modo tale da riuscire a proporre un preventivo finale per il costo totale del progetto.

2.1.2.3 Piano di Qualifica

Al *Verificatore* è assegnato il compito di scegliere un metodo per la $verifica_G$ e $validazione_G$ del materiale realizzato dal team. Il documento deve contenere:

- **Metodo di verifica:** vengono stabilite le procedure di controllo sulla qualità di processo e di prodotto.
- **Metriche:** è necessario stabilire delle metriche oggettive per i documenti e i processi software.
- **Gestione della revisione:** si devono stabilire delle modalità per comunicare le anomalie e le procedure per controllare la qualità di processo;
- **Pianificazione collaudo:** è necessario definire dettagliatamente le metodologie di collaudo del progetto.
- **Resoconto attività verifica:** alla fine di ogni attività svolta si devono riportare le metriche relative e un resoconto sulla verifica effettuata.

2.2 Sviluppo

2.2.1 Scopo

Questa sezione affrontà le attività e i compiti svolti dal gruppo al fine di produrre il software richiesto dall'azienda IKS Srl.

2.2.2 Aspettative

Al fine di implementare correttamente questa attività vi sono le seguenti aspettative:

- Realizzare un prodotto software conforme alle richieste del proponente
- Realizzare un prodotto software che soddisfi i test di verifica
- Realizzare un prodotto software che soddisfi i test di validazione
- Fissare degli obiettivi di $sviluppo_G$
- Fissare eventuali vincoli tecnologici
- Fissare i vincoli di design

2.2.3 Descrizione

Il processo di sviluppo si svolge in accordo con lo standard ISO/IEC 12207.Pertanto si compone delle seguenti attività:

- Analisi dei requisiti
- Progettazione
- Codifica

2.2.4 Attività

2.2.4.1 Analisi dei requisiti

2.2.4.1.1 Scopo Lo scopo di questa analisi è quella di elencare nel modo più preciso possibile, tutti i requisiti del progetto. I requisiti sono estrapolati da diverse fonti:

- Specifica del capitolato
- Incontri tenuti direttamente con il proponente
- Casi d'uso

Dopo aver svolto questo processo viene stilato in modo formale un documento di analisi chiamato *Analisi dei Requisiti v1.0.0*. Quest'ultimo è steso dagli *Analisti* e contiene una lista dei requisiti e dei casi d'uso. In questo modo sarà possibile effettuare e definire dei test di superamento del software.

2.2.4.1.2 Aspettative L'obiettivo di questa attività è quello di definire in modo formale e dettagliato tutti i requisiti richiesti dal proponente.

2.2.4.1.3 Descrizione Nel documento inerente a quest'analisi sono specificati i requisiti analizzati. Il metodo utilizzato per l'analisi dei requisiti è quella dei casi d'uso.

2.2.4.1.4 Casi d'uso

2.2.4.1.5 Codice identificativo

2.2.4.1.6 Requisiti

2.2.4.1.7 Codice identificativo

2.2.4.1.8 UML I diagrammi UML devono essere realizzati usando la versione del linguaggio *v2.0*

2.2.4.2 Progettazione

2.2.4.2.1 Scopo

L'attività di Progettazione ha lo scopo di definire e descrivere la progettazione ad alto livello dell'architettura del prodotto richiesto e specificarla descrivendo la progettazione di dettaglio. Il responsabile di questa attività è il $Progettista_G$ che deve redarre la `Specifica Tecnica` e la `Definizione di Prodotto` in funzione dei requisiti delineati nell'`Analisi dei Requisiti` in modo da permettere di definire le linee guida da seguire durante l'attività di codifica. Per fare ciò deve avere una profonda conoscenza dell'intero processo di sviluppo del software. Solo in questo modo possono:

- progettare un software con le caratteristiche di qualità specificate nella fase di analisi e specifica dei requisiti;
- effettuare modifiche senza che la struttura del software già costruita debba essere messa nuovamente in discussione;
- soddisfare i requisiti di qualità fissati dal committente.

2.2.4.2.2 Specifica Tecnica

Questo documento deve descrivere la progettazione ad alto livello dell'architettura del software richiesto dal proponente. Inoltre deve provvedere alla progettazione di sistemi di integrazione. Per fare ciò si useranno i seguenti strumenti:

- Diagrammi UML_G:
  Essi forniscono una rappresentazione molto chiara e compatta dell'intera struttura dell'applicazione che si andrà ad analizzare. In particolare devono essere realizzati i seguenti diagrammi:
  - Diagrammi delle classi: illustrano una collezione di elementi dichiarativi di un modello come classi e tipi, assieme ai loro contenuti e alle loro relazioni;
  - Diagrammi dei $package_G$: raggruppamento di classi in una unità di livello più alto;
  - Diagrammi di attività: illustrano il flusso di operazioni relativo ad un'attività. Utilizzati soprattutto per descrivere la logica di un algoritmo;
  - Diagrammi di sequenza: descrivono una determinata sequenza di azioni dove tutte le scelte sono già effettuate. In pratica nel diagramma non compaiono scelte ne flussi alternativi.

  I diagrammi UML sono realizzati tramite il programma StarUML descritto nella sezione 2.2.5.1.
- $DesignPattern_G$:
  Devono essere descritti tramite descrizione e diagramma i design pattern utilizzati.
- Tracciamento delle componenti:
  Ogni requisito deve riferirsi al componente che lo soddisfa. Per fare ciò si utilizzerà SWEgo, un'applicazione web (descritta nella sezione 2.2.5.2) utile nel tracciamento Use Case - Requisiti e nella generazione automatica dei diagrammi dei Casi d'Uso.
- Test di integrazione:
  I Progettisti devono definire delle classi di verifica per verificare che i componenti del sistema funzionino nella maniera prevista.

2.2.4.2.3 Definizione di Prodotto

Questo documento deve descrivere la progettazione in dettaglio del sistema, utilizzando i seguenti strumenti:

- Diagrammi UML_G
  Devono essere realizzati i seguenti diagrammi:
  - Diagrammi delle classi;
  - Diagrammi dei $package_G$;
  - Diagrammi di attività;
  - Diagrammi di sequenza.

I diagrammi UML sono realizzati tramite il programma StarUML descritto nella sezione 2.2.5.1.

- Definizione delle classi
  Ogni $classe_G$ progettata deve essere descritta con una spiegazione sullo scopo della classe e deve specificare quale funzionalità essa modella.
- Tracciamento classi:
  Ogni requisito deve essere tracciato alle classi che lo soddisfano. Per fare ciò si utilizzerà SWEgo, un'applicazione web descritta nella sezione 2.2.5.2.
- Test di unità:
  I Progettisti devono definire i test di unità necessari per verificare che i componenti del sistema funzionino nel modo previsto.

2.2.4.3 Codifica

2.2.4.3.1 Scopo

Questa attività ha come scopo l'effettiva implementazione del prodotto software richiesto. In questa fase dunque si concretizzano attraverso la codifica le funzionalità previste dai requisiti concordati. L'attività deve rispettare quando imposto dai documenti *Definizione di prodotto* e *Piano di Qualifica*.

2.2.4.3.2 Stile di codifica

Al fine di garantire uniformità all'intera $codebase_G$, ciascun membro del gruppo è obbligato a rispettare le seguenti norme:

- **Formattazione**: è richiesto l'uso di uno spazio (" ") ove possibile per rendere il codice di facile comprensione. Di seguito alcuni esempi di buona e cattiva formattazione.

```
int a=3; // BAD
int a = 3; // GOOD
int a = 3,b = 5; // BAD
int a = 3, b = 4; // GOOD

getFoo(a,b,c,d); // BAD
getFoo(a, b, c, d); // GOOD

if(a==5){ // BAD
if (a == 5) { // GOOD
```

- **Indentazione:** sono richiesti 4 spazi (si consiglia di impostare adeguatamente il proprio IDE_G), inoltre non sono permessi spazi bianchi o tabulazioni a fine riga.
- **Lunghezza linee di codice:** una riga di codice non deve superare i 100 caratteri, in caso contrario spezzare la riga di codice andando a capo.

- **Nomi:** i nomi delle funzioni e delle variabili devono essere significativi e devono cominciare con la lettera minuscola e seguire la notazione $camelCase_G$. I nomi delle classi devono avere la prima lettera maiuscola mentre i nomi delle costanti devono essere scritti interamente in maiuscolo.

```
var foo; // BAD
var profiledata // BAD
var profileData // GOOD
```

- **Lingua:** i nomi delle variabili i metodi le funzioni e i commenti vanno scritti in inglese.
- **Lunghezza metodi e funzioni:** il corpo dei metodi e delle funzioni non deve superare le 40 linee e i due gradi di indentazione. Superare tali limiti è chiaro segno che la funzione / metodo necessita di essere spezzata. A volte però è preferibile mantenere tutta la logica su un solo metodo per facilitarne la comprensibilità.
- **Strutture di controllo:** le parentesi graffe per i costrutti che le prevedono devono essere in linea con il costrutto stesso. E' prevista l'omissione delle graffe soltanto quando il corpo della struttura è di una sola riga.

```
if (...) { // GOOD
    // CODICE
} else {
    // CODICE
}


if (...) // BAD
{
    // CODE
}
else
{
    // CODE
}
```

- **Commenti:** i commenti devono essere esaustivi e non prolissi in modo da aiutare il lettore a comprendere al meglio ciò che si sta leggendo.

  E' inoltre consentito l'utilizzo di particolari tipi di commento con lo scopo di aiutare la stesura del codice. Vi sono:

  - **TODO:** soluzione temporanee, memento di ogni tipo o zone che necessitano di un'implementazione.

```
            // TODO: this solution stinks
            // TODO: Write that query
```

– **HOW:** per segnalare che non si ha capito l'implementazione o il funzionamento di una porzione di codice e che si necessita di uno studio più approfondito.

```
// HOW: need to study this API call, I'm puzzled
```

– **FIX:** quando una particolare implementazione necessita di essere riparata o sistemata.

```
// FIX: this implementation doesn't work with IE
```

Questi commenti vanno scritti tutti in maiuscolo e seguiti da due punti (:), questa sintassi favorisce la loro individuazione da particolari $tool_G$ che ne permettono una consultazione agevolata.

2.2.4.3.3 Intestazione

Ogni $file_G$ contenente codice sorgente deve avere la seguente intestazione:

```
/*

* File : nome file
* Version : versione file
* Type : tipo file
* Date : data di creazione
* Author : nome autore / i
* E - mail : email autore / i
*
* License : tipo licenza
*
* Avvertenze : lista avvertenze e limitazioni
*
* Descrizione: breve descrizione del contenuto del file
*
* Registro modifiche :
* Autore || Data || breve descrizione modifiche
*
*/
```

2.2.4.3.4 Versionamento

Il versionamento dei file segue la filosofia "Change Significance". La versione di un file è espressa secondo la notazione X.Y.Z. I valori sono ordinati per importanza decrescente, partendo da sinistra. L'aumento di un parametro comporta l'azzeramento di tutti i parametri alla sua destra. Ogni volta che si modifica un file va modificata anche la versione corrente seguendo questa logica: ATTENZIONE: HO MESSO UN SACCO DI TERMINI PARTICOLARI NEL GLOSSARIO, NON SO SE SIA NECESSARIO

- **X:** $majorrelease_G$, l'incremento di questo numero significa che sono state aggiunte funzionalità importanti
- **Y:** $minorimprovement_G$, $bugfix_G$, va incrementato in caso di implementazione di funzionalità più piccole
- **Z:** incrementato in caso di $typos_G$, errori nei commenti o piccoli bug

Quando un file raggiunge la versione 1.0.0 significa che sono state implementate le funzionalità obbligatorie e quindi si può cominciare a testare il codice.

2.2.4.3.5 Ricorsione

E' caldamente sconsigliato l'utilizzo di ricorsione, esistono però dei casi in cui la soluzione ricorsiva rende il problema triviale, in tal caso è richiesta la verifica di correttezza e di terminazione della ricorsione.

2.2.5 Strumenti

2.2.5.1 StarUML

StarUML è uno strumento di modellazione UML compatibile con gli standard UML 2.x che supporta 11 tipi di diagrammi, tra cui i diagrammi delle classi, dei package, di attività e di sequenza che ci interessano per la progettazione del nostro programma.
Sono inoltre disponibili diverse estensioni.

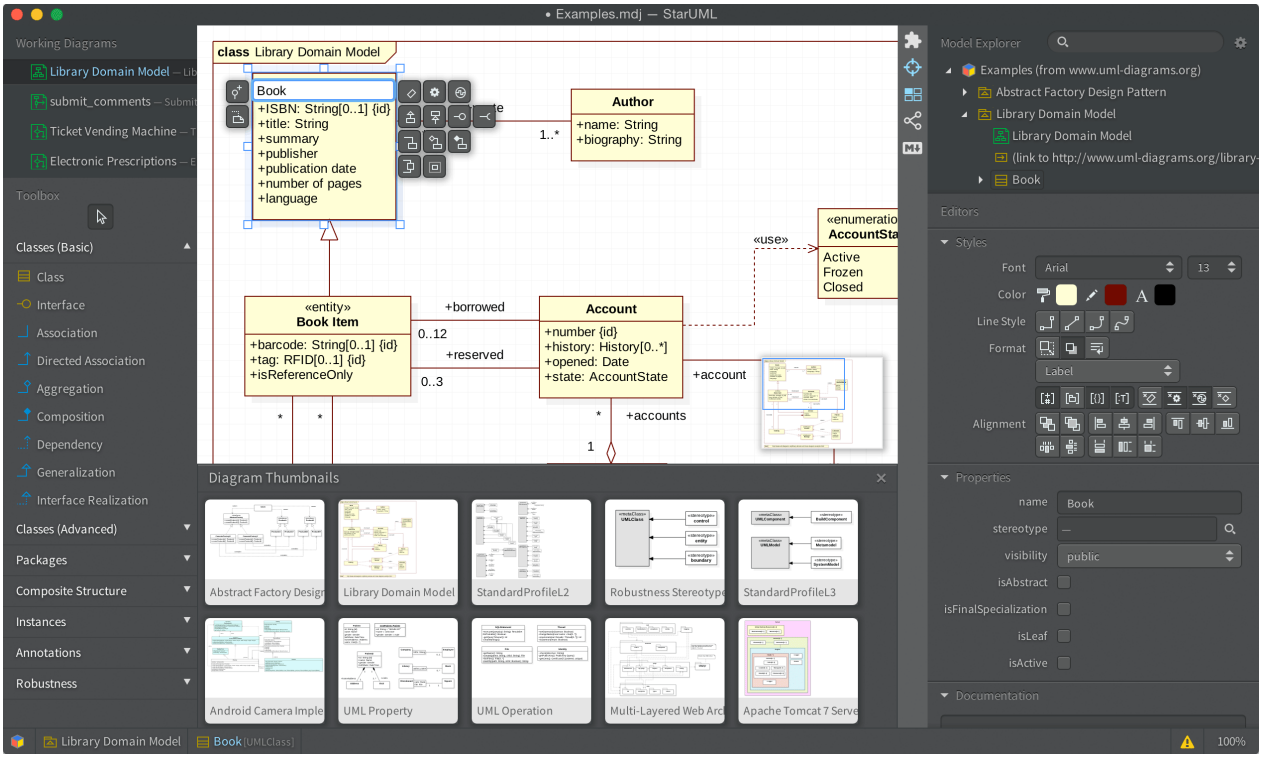


Figura 1: Diagramma UML con StarUML

2.2.5.2 SWEgo

SWEgo è l'applicativo web scelto dal gruppo per il tracciamento dei requisiti. Offre molti servizi, come:

- Tracciamento Use Case - requisiti:
  Tracciare un requisito serve a spiegare qual è l'origine di tale requisito e garantirne la necessità e la sufficienza. Il tracciamento è l'incontro tra i bisogni del proponente e dei requisiti.
- Generazione automatica dei diagrammi dei Casi d'Uso utilizzando PlantUML:
  SWEgo genera gli Use Case, dei diagrammi che esprimono un comportamento del sistema, offerto o desiderato, sulla base dei suoi risultati osservabili.
- Visualizzazione grafici che indicano la copertura dei requisiti:
  SWEgo permette di visualizzare le percentuali di copertura dei requisiti in tempo reale grazie a dei semplici grafici a torta.

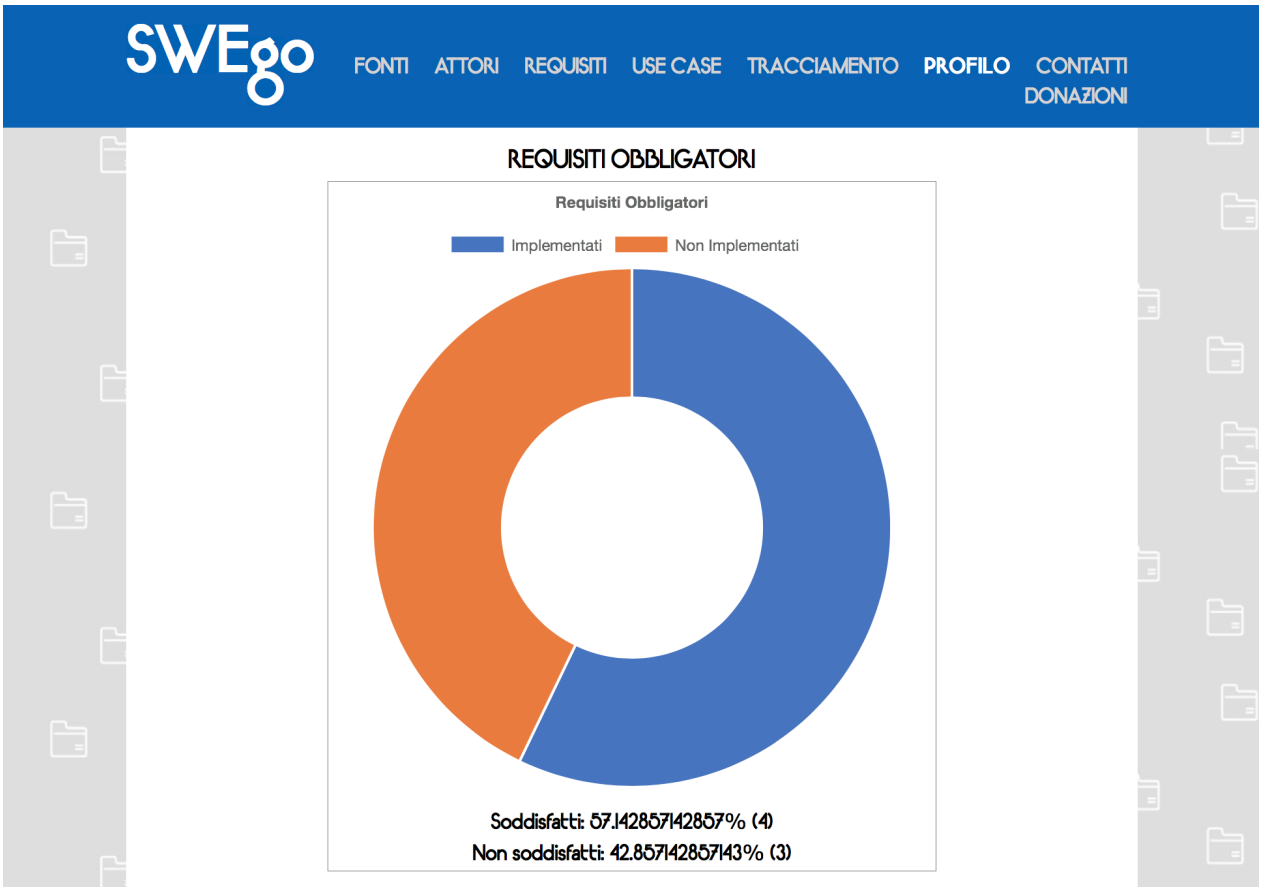


Figura 2: Copertura requisiti con Swego

2.2.5.3 Atom

Atom è un ide_G di ultima generazione, sviluppato da GitHub. E' un editor potente, $crossplatform_G$ e si integra perfettamente con tutte le tecnologie e strumenti che abbiamo deciso di utilizzare.

Figura 3: Use Case con Swego

Offre inoltre un vastissimo numero di *plugin*$_G$ grazie al grande supporto della comunità open source che lo sviluppa.

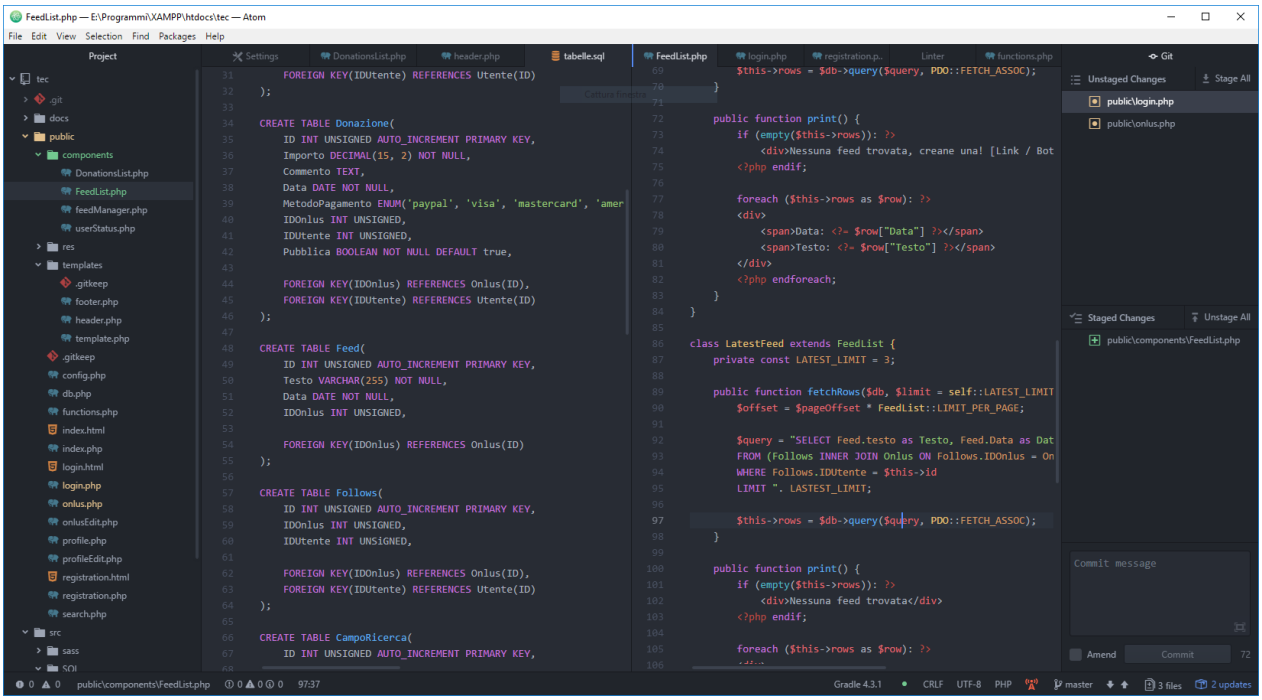


Figura 4: Interfaccia grafica di Atom

3 Processi Organizzativi

3.1 Gestione

3.1.1 Scopo

Questo *processo*$_G$ ha come scopo la redazione del documento *Piano di Progetto* e disciplinare la coordinazione del gruppo in merito a:

- Ruoli di progetto
- Gestione delle comunicazioni
- Gestione degli incontri
- Sistema di ticketing
- Sistema di versionamento

3.1.2 Ruoli di progetto

I ruoli di progetto rappresentano le figure professionali che lavoreranno al progetto. Ogni membro deve ricoprire ciascun ruolo almeno una volta. Come ciò è pianificato e verificato è specificato nel documento *Piano di Progetto v1.0.0*. I ruoli sono:

3.1.2.1 Amministratore di Progetto

L'*Amministratore di Progetto* è la figura professionale che si deve occupare di gestire l'ambiente di lavoro del gruppo. Nello specifico deve

- Ricercare nuovi strumenti che migliorino la produttività del gruppo
- Imparare ad utilizzare adeguatamente gli strumenti utilizzati
- Se necessario spiegare agli altri componenti del gruppo come utilizzare correttamente tali strumenti
- Occuparsi del controllo di versione del prodotto
- Occuparsi della configurazione del prodotto

è presente per tutta la durata del progetto.

3.1.2.2 Analista

L'*Analista* è la figura professionale che si occupa dell'analisi dei requisiti del prodotto e del dominio applicativo. Deve

- Analizzare i requisiti del prodotto. Per fare ciò dovrà parlare in prima persona con il committente
- Analizzare i requisiti di dominio

- Redigere il documento *Studio di Fattibilità*
- Redigere il documento *Analisi dei Requisiti*

è presente solamente nella fase iniziale del progetto.

3.1.2.3 Progettista

Il*Progettista* è la figura professionale che si occupa della progettazione del prodotto. Deve

- Comprendere a fondo i requisiti nel documento *Analisi dei Requisiti*
- Progettare un'*architettura*$_G$ per il prodotto
- Scegliere le tecnologie che si utilizzeranno per realizzare il prodotto, in modo tale che essere permettano di soddisfare i requisiti
- Redigere il documento *Specifica Tecnica*
- Redigere il documento *Definizione di prodotto* ATTEZIONE: questi due documenti esistono ancora?

è presente nella fase di progettazione del prodotto software e *può* seguire il progetto fino al termine.

3.1.2.4 Verificatore

Il *Verificatore* è la figura professionale che si occupa dell'attività di verifica. Deve

- Verificare che ciascuna attività svolta sia conforme alle norme stabilite nel progetto
- Redigere il documento *Piano di Qualifica*

è presente per l'intera durata del progetto.

3.1.2.5 Programmatore

Il *Programmatore* è la figura professionale che si occupa della codifica del prodotto. Deve

- Scrivere il codice del prodotto software che rispetti le decisioni del *Progettista*
- Redigere il documento *Manuale Utente*

è presente durante la fase di codifica del prodotto.

3.1.2.6 Project Manager

Il *Project Manager* è la figura professionale funge da punto di riferimento per il gruppo, il fornitore e il *committente*$_G$ in merito al progetto che amministra. Deve

- Organizzare incontri interni ed esterni
- Pianificare le attività svolte dal gruppo, suddividendole in compiti
- Individuare per ciascun compito un membro del gruppo per svolgerlo

- Analizzare, monitorare e gestire i rischi

è presente per l'intera durata del progetto.

3.1.3 Procedure

3.1.3.1 Gestione delle comunicazioni

3.1.3.1.1 Comunicazioni interne

Le comunicazioni interne sono gestite con la piattaforma $Slack_G$. Essa rende possibile creare un $workspace_G$ comune al gruppo suddiviso in più *canali di comunicazione*$_G$, ad ognuno dei quali sono associati uno scopo ed un sottoinsieme dei membri del gruppo. *Slack* permette anche a due componenti del gruppo di mandarsi messaggi privati. Il maggiore vantaggio rispetto all'utilizzo di un'applicazione di messaggistica come $Telegram_G$ o $Whatsapp_G$ sta nel poter creare uno spazio di comunicazione condiviso che permetta di gestire più categorie di comunicazioni in un singolo *workspace*.

3.1.3.1.2 Comunicazioni esterne

La gestione delle comunicazioni esterne è compito del *Project Manager*. Egli deve utilizzare l'indirizzo di posta elettronica:

sweeftyteam@gmail.com

Il *Project manager* se lo ritiene opportuno può informare i membri del gruppo delle eventuali comunicazioni esterne tramite il canale Slack *#email*.

3.1.3.2 Gestione degli incontri

3.1.3.2.1 Incontri interni

Gli incontri interni del gruppo devono essere organizzati dal *Project Manager*. Egli deve:

1. Proporre tramite il canale Slack *incontri* una data, un'ora, un luogo e una motivazione per organizzare l'incontro. La data in cui ciò avviene deve essere almeno a due giorni di distanza dalla data proposta per incontrarsi. Quest'ultima regola può essere ignorata solamente nel caso in cui fosse strettamente necessaria una riunione tempestiva.
2. Raccogliere tramite il Bot_G Slack $Simple\ Poll_G$ le risposte di partecipazione o assenza dei membri. Esse devono essere date entro e non oltre 24 ore di tempo dall'apertura del sondaggio.
3. Se almeno cinque membri avranno dato conferma di presenza il *Project Manager* conferma che l'incontro si svoltge con i parametri definiti nel punto 1. Se le conferme di presenza sono tre o quattro starà al *Project Manager* decidere fra:
    - Confermare l'incontro con solo coloro che hanno confermato al presenza

- Tornare al punto 1 cambiando i parametri data e/o ora ed eventualmente luogo

Tale decisione deve prendere in considereazione i benefici che porterebbe organizzare un incontro con tre o quattro membri e gli svantaggi che si avrebbero posticipando l'incontro. Se le conferme sono meno di tre sarà necessario tornare al punto 1 cambiando i parametri data e/o ora ed eventualmente luogo

4. Nel caso in cui l'incontro venga confermato, stilare in un file nella cartella della repository `Docs/Verbali` l'ordine del giorno. Il file deve avere come nome la data in cui si terrà l'incontro, scritta nel formato indicato in **??** dove sta sta data?. Nello stesso file dovrà essere verbalizzato ciò di cui si è discusso durante l'incontro nelle modalità descritte alla fine di questo paragrafo

Tutti i membri del team possono chiedere di organizzare un incontro al *Project Manager*, specificando una motivazione. Se quest'ultima è ritenuta sufficiente per organizzare un incontro si procede ad avviare la procedura sopra descritta.
Inoltre, ad ogni incontro, un membro del gruppo viene scelto dal *Project Manager* per stilare il verbale. Egli deve inserire nello stesso file creato dal *Project Manager* contenente l'ordine del giorno un riassunto scritto di ciò di cui si è discusso e ciò che è stato deciso.

3.1.3.2.2 Incontri esterni

Gli incontri esterni devono essere organizzati dal *Project Manager*. È suo compito mettersi in contatto con il committente o il proponente tramite email e determinare data, ora e luogo dell'incontro. Quando ciò è definito deve avvisare i membri del gruppo sul canale Slack *#incontri*. Anche per gli incontri esterni ciascun membro del gruppo può richiedere che venga organizzato un incontro con il committente o il proponente, specificandone la motivazione. Se essa è giudicata adeguata dal *Project Manager* egli deve procedere ad organizzare tale incontro. È anche suo compito incaricare uno dei presenti di redigere un verbale scritto.

3.1.3.3 Gestione degli strumenti di coordinamento

3.1.3.3.1 Ticketing

Per la gestione del sistema di ticketing si utilizza la piattaforma $Wrike_G$. Essa permette di definire un insieme di $tasks_G$. Ad ognuno di essi viene assegnato un nome, una breve descrizione,e un membro che sia incaricato di portarlo a termine, una data di inizio e una data entro la quale il task deve essere completato. Il *Project Manager* deve creare i tasks ed assegnare un membro a ciascuno di essi. Una volta raggiunta la data di inizio, l'incaricato di svolgerlo deve portarlo a termine entro e non oltre la data stabilita per la fine. Quando fa ciò deve impostare sulla piattaforma Wrike che tale task è stato completato. Ogni membro del gruppo è tenuto a controllare almeno una volta al giorno la piattaforma per controllare se gli sono stati assegnati nuovi compiti.

3.1.3.3.2 Struttura del worskspace di comunicazioni interne

Come detto in 3.1.3.1.1 lo strumento utilizzato per le comunicazioni interne è $Slack_G$. Il workspace del gruppo è organizzato nei seguenti canali:

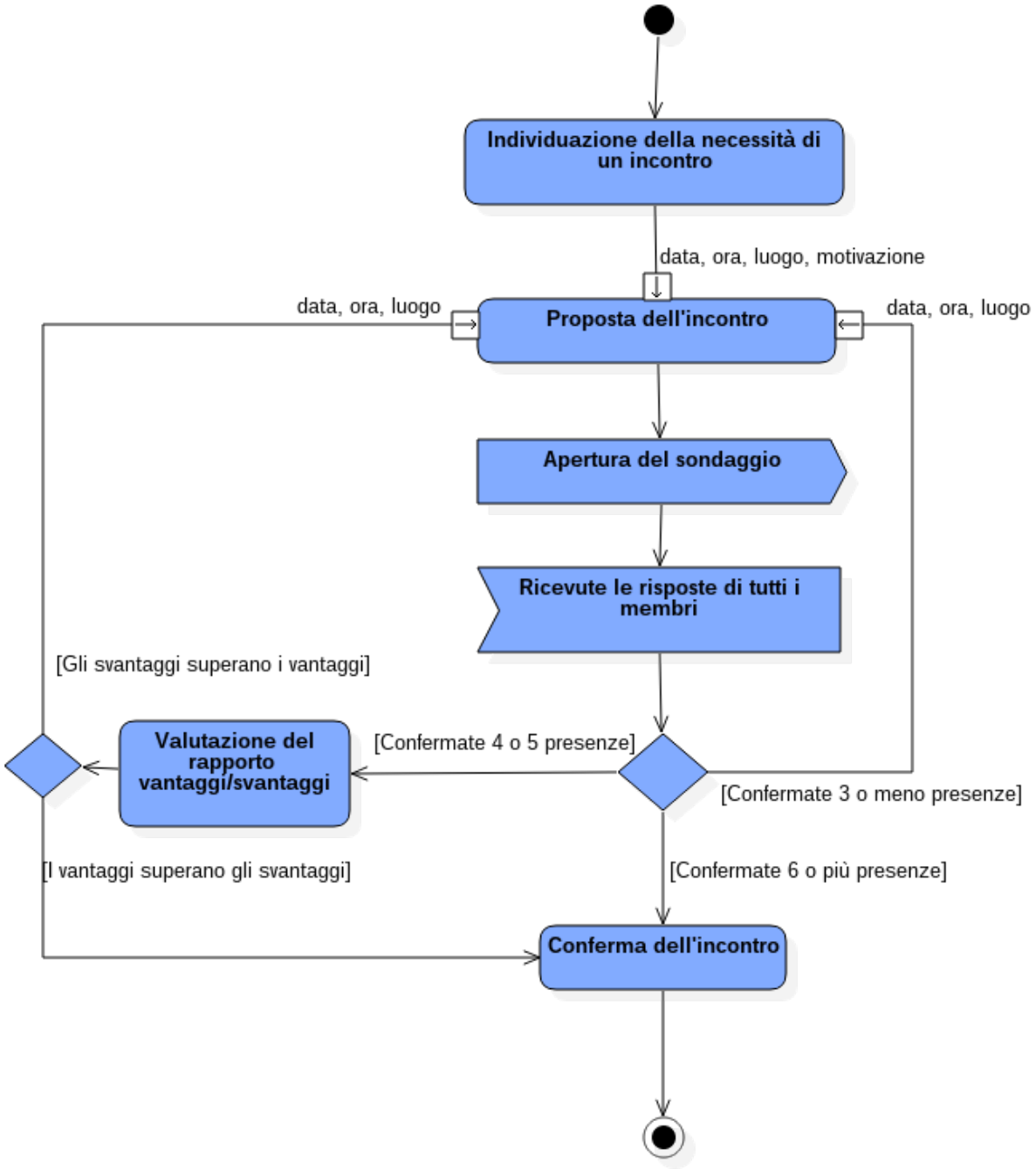


Figura 5: Diagramma che rappresenta la procedura di organizzazione di un incontro interno

- #**general** è il canale nel quale il gruppo si scambia informazioni per l'appunto di carattere generale, per esempio domande su come utilizzare una $feature_G$ di un determinato strumento piuttosto che avvisare gli altri di essere in anticipo per un incontro
- #**incontri** ha lo scopo di organizzare gli incontri interni al gruppo. Qui vengono effettuati i sondaggi di partecipazione tramite il bot *Simple Poll* e successivamente è confermato o meno l'incontro dal *Project Manager*
- #**todo** ha lo scopo di informare gli altri membri del gruppo che una certa porzione di un determinato task non è completa e potrà essere completata solo dopo che saranno state determinate delle norme nel gruppo. È compito di chi lascia il task incompleto informare gli altri tramite questo canale. I messaggi devono seguire la forma:

  [$TODO$] spiegazione del problema

- #**email** è il canale nel quale il *Project Manager* inserisce le risposte che vengono fornite da entità esterne alla casella mail del gruppo, se lo ritiene necessario o utile.

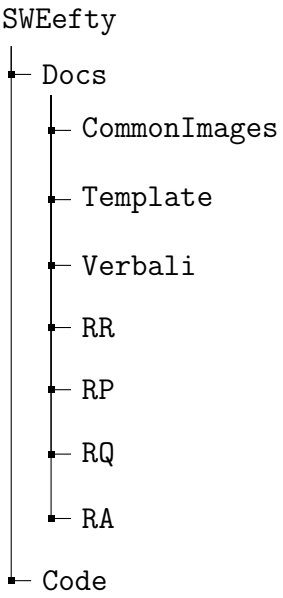
3.1.3.4 Gestione degli strumenti di versionamento

3.1.3.4.1 Repository

La piattaforma di $hosting_G$ scelta per il repository è $GitHub_G$. Si utilizza una licenza "educational", che permette di avere fino a cinque repository private condivise. Di queste solamente una è effettivamente utilizzata. È organizzata come descritto in 3.1.3.4.2

3.1.3.4.2 Struttura del repository

La struttura è la seguente



dove:

- `SWEefty` rappresenta la radice dello spazio di lavoro
- `Docs` è la directory in cui sono inseriti tutti i documenti formali
- `Code` è la directory dove è inserito il codice
- `CommonImages` contiene le immagini comuni a tutti i documenti, per esempio il logo del gruppo
- `Template` contiene i files che compongono la struttura comune a tutti i documenti che verranno redatti
- `Verbali` contiene i files dei verbali degli incontri interni. Il nome di ogni file è identificato dalla data in cui è stata fatta la riunione del gruppo nel formato descritto in **??** dove sta sto formato data?
- `RR` contiene tutti i documenti da consegnare per la Revisione dei Requisiti
- `RP` contiene tutti i documenti da consegnare per la Revisione di Progettazione
- `RQ` contiene tutti i documenti da consegnare per la Revisione di Qualifica
- `RA` contiene tutti i documenti da consegnare per la Revisione di Accettazione

3.1.3.4.3 Norme sui nomi dei files

Ciascun file deve essere denominato seguendo la codifica:

`NomeDelFile.ext`

Dove NomeDelFile rappresenta il nome del file e ciascuna parola che compone tale nome è identificata da una lettera maiuscola. Ext rappresenta l'estesione del file. Nel caso in cui dovessero essere presenti lettere accentate nel nome, l'accento deve essere ignorato e inserita semplicemente la lettera non accetata. Per esempio il file sorgente .tex del documento *Studio di Fattibilità* è denominato `StudioDiFattibilita.tex`. Ci mettiamo il controllo di versione nel nome? io non lo farei..forse

3.1.3.4.4 Estensioni ammesse

I file che si trovano effettivamente nella directory `Docs` sono solamente immagini .jpg e .png, file sorgente LaTeX .tex . Tutti i file.pdf compilati e i prodotti intermedi del programma per la compilazione di file .tex `pdflatex` .aux, .dvi, .fls, .log, .out, .toc verranno inseriti nel file `.gitignore` che rende la loro presenza trasparente a Git.

3.1.3.4.5 Norme di branching e merging

Git mette a disposizione il comando `branch` per poter creare un nuovo ramo di lavoro all'interno della repository, indipendente da quello principale. I rami possono essere a loro volta divisi in sottorami figli, così come uniti ad altri tramite il comando `merge`. Si deve creare un nuovo branch nei seguenti casi:

- Viene istanziato un nuovo documento che non sia un verbale
- Più componenti del gruppo stanno lavorando al medesimo documento ed è necessario inserire un nuovo file che abbia estensione .tex.
- Deve essere implementata nel codice un'*unit*$_G$

Tutte le modifiche ai documenti e al codice devono essere eseguite all'interno del nuovo branch. Una volta completate sarà possibile eseguire il merging del branch figlio con il padre. Questa operazione deve essere eseguita dall'*Amministratore di Sistema* che dovrà correggere eventuali conflitti.

3.1.3.4.6 Norme su commit

Ogni volta che viene effettuata una modifica significativa ad un file oppure una serie di modifiche strettamente correlate su più files, va utilizzato il comando `commit` di *git*$_G$. In esso di deve specificare un messaggio che descriva brevemente ciò che è stato fatto. I caratteri di questo messaggio non devono essere più di 80. Deve seguire la semantica

[sigla file] breve descrizione

Per esempio l'aggiunta della sezione "XYZ" al file "NormeDiProgetto.tex" comporterebbe scrivere come messaggio informativo "[NDP] Aggiunta sezione XYZ". Prima di eseguire una `commit` va aggiornato il diario delle modifiche come descritto in **??** dove sto sto diario?

3.1.3.5 Gestione dei rischi

La gestione dei rischi è compito del *Project Manager*. Egli li deve identificare, inserirli nel *Piano di Progetto*$_G$ e valutare una strategia per affrontarli. Nel caso in cui si presentassero nuovi rischi non osservati sarà sempre compito del *Project Manager* inserirli nel *Piano di Progetto*. Durante tutta la durata del *progetto*$_G$ deve inoltre monitorare tali rischi e, solo se necessario, ridefinire le strategie di progetto.

3.1.4 Strumenti

3.1.4.1 Sistema Operativo

I sistemi operativi utilizzati dai membri del gruppo sono:

- Ubuntu GNOME 17.10 x64
- Ubuntu 17.10 x64
- Windows 10 Home x64
- MacOS High Sierra x64

3.1.4.2 Slack

Slack è lo strumento adottato per gestire le comunicazioni interne al gruppo. Esso è gratuito e permette di creare in un workspace comune con più canali di comunicazione, contrassegnati da

un cancelletto. Permette anche l'integrazione con altri strumenti utilizzati, quali $Drobbox_G$, Wrike e GitHub. Queste $major\ features_G$ lo rendono preferibile ad altre applicazioni di messaggistica quali Telegram o Whatsapp, più idonee ad un uso personale.

3.1.4.3 Wrike

Wrike è lo strumento adottato per gestire il sistema di ticketing. È una piattaforma online che mette a disposizione anche una applicazione per dispositivi mobili per la gestione di tasks all'interno di un progetto. Il principale motivo per il quale risulta comodo da utilizzare è che il sistema, una volta definiti un insieme di compiti e ad essi sono stati assegnate date di inizio e di fine, permette di avere più viste degli stessi: come una lista testuale, una tabella, uno stream, una board o un $diagramma\ di\ Gantt_G$.
Wrike è uno strumento a pagamento. La licenza utilizzata dal gruppo è quella per studenti ed ha una durata di 750 giorni.

3.1.4.4 Github

Github è la piattaforma scelta per l'hosting della repository. Essa si basa sul sistema di versionamento git_G, descritto in 3.1.4.5. Ha piani di sottoscrizioni gratuiti o a pagamento. La differenza sta nella possibilità di disporre di repository private. Con l'edizione "educational" da noi utilizzata si possono ottenere gratuitamente fino a cinque repository gratuite. GitHub offre inoltre un gran numero di tools utili per lo sviluppo aggiuntivi, come la gestione tramite interfaccia web delle $issues_G$.

3.1.4.5 Git

Git è il sistema per il controllo del versionamento utilizzato. Ha un'architettura distribuita ed è l'unico sistema di controllo di versione supportato da GitHub. Rispetto ad altri strumenti di controllo di versionamento, come $Mercurial_G$, ha un numero molto maggiore di comandi, il che rende difficile per un neofita l'utilizzo da riga di comando di git.

3.1.4.6 Gitkraken

$Gitkraken_G$ è un'applicazione desktop $multipiattaforma_G$ che semplifica l'utilizzo di git, esponendo una chiara interfaccia grafica che si sostituisce all'utilizzo di git da riga di comando. Grazie a ciò rende più semplice imparare ad usare lo strumento di versionamento e dà una visualizzazione grafica dello storico della repository.

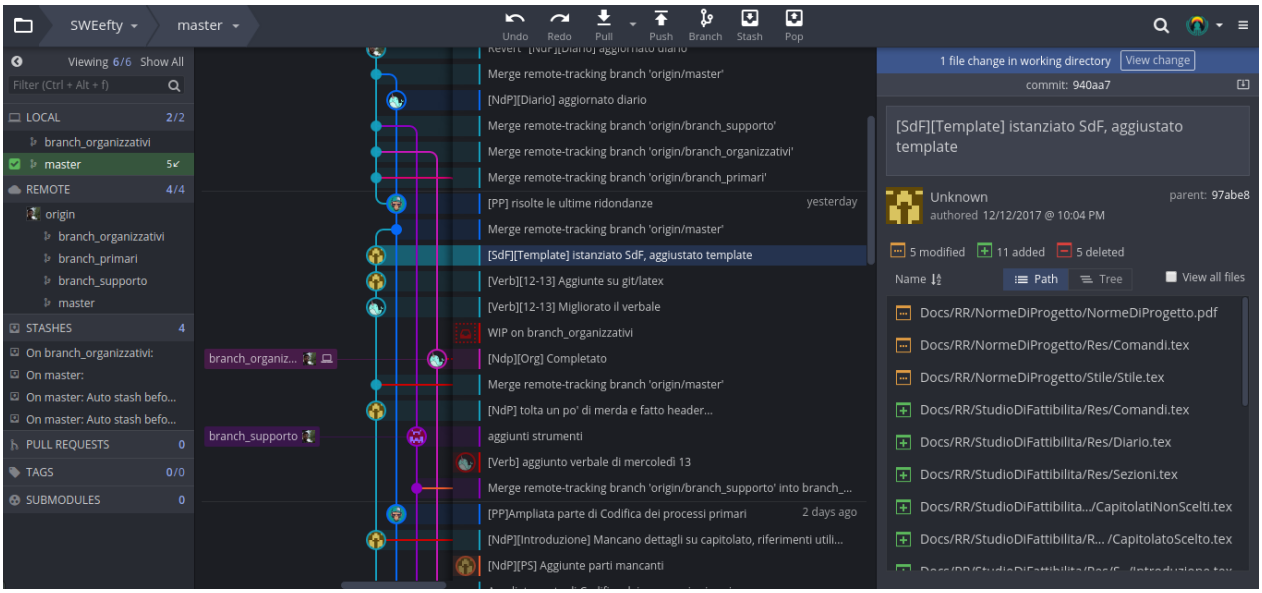


Figura 6: Interfaccia grafica del software Gitkraken su Ubuntu GNOME

4 Processi di Supporto

4.1 Documentazione

4.1.1 Scopo

Questo processo include e descrive le modalità di redazione e manutenzione dei documenti durante il ciclo di vita del prodotto software. Sono illustrate le norme, strumenti e convenzioni adottate per la scrittura di questi. Il tutto con l'obiettivo di consentire la stesura, senza ambiguità, di una documentazione valida, coerente e conforme a delle precise regole.

4.1.2 Procedure

Il gruppo si è servito del linguaggio di markup LaTeX per la stesura della documentazione.

4.1.2.1 Approvazione dei documenti

Ogni documento non formale in corrispondenza del completamento della sua stesura dovrà essere sottoposto al *Responsabile di Progetto*, che dovrà delegare ai *Verificatori* il controllo del contenuto e della forma. Nel caso gli anzidetti *Verificatori* rilevino degli errori, sarà loro compito notificarli al *Responsabile di Progetto*, che a sua volta affiderà il redattore del documento il compito di correggerli. Questo ciclo va eseguito fino a quando il documento non viene considerato completamente corretto dai *Verificatori*. In caso di assenso sulla correttezza e sulla qualità il documento può essere reputato un documento formale. In caso contrario spetta al *Responsabile di Progetto* comunicare le ragione per cui il documento non è stato giudicato corretto, esplicitando le modifiche da apportare.

4.1.3 Template

Per agevolare la redazione della documentazione è stato creato un template LaTeXcontenente tutte le impostazioni stilistiche e grafiche citate in questo documento.

4.1.4 Struttura dei documenti

4.1.4.1 Prima pagina

Ogni documento è caratterizzato da una prima pagina che contiene le seguenti informazioni sul documento:

- Logo del gruppo;
- Titolo del documento;
- Nome del gruppo;
- Nome del progetto;
- Versione del documento;
- Cognome e nome dei redattori del documento;
- Cognome e nome dei verificatori del documento;
- Cognome e nome del responsabile approvatore del documento;
- Destinazione d'uso del documento;
- Lista di distribuzione del documento;
- Una breve descrizione del documento.

4.1.4.2 Registro delle modifiche

La seconda pagina di ogni documento contiene il diario delle modifiche del documento. Ogni riga del diario delle modifiche contiene:

- Un breve sommario delle modifiche svolte;
- Cognome e nome dell'autore;
- Ruolo dell'autore;
- Data della modifica;
- Versione del documento dopo la modifica.

La tabella contenente le modifiche è ordinata per data in ordine decrescente, affinchè la prima riga contenga la versione attuale del documento.

4.1.4.3 Indici
In ogni documento, esclusi i verbali, è presente un indice delle sezioni, un indice delle figure e un indice delle tabelle. Nel caso non siano presenti figure o tabelle i rispettivi indici verranno omessi. La notazione numerica di ogni indice deve partire da 1 e le sottosezioni devono essere separate da un punto. Anche la notazione della sottosezione parte da 1.

4.1.4.4 Formattazione generale delle pagine
Il template prevede dei margini orizzontali e verticali che devono essere rispettati in ogni pagina. Ad esclusione della prima, tutte le pagine contengono un'intestazione ed un piè di pagina.

4.1.4.4.1 Intestazione
L'intestazione è così strutturata:

- Logo del gruppo posto a sinistra;
- Indirizzo di posta elettronica del gruppo posto a destra;
- Ruolo dell'autore;

4.1.4.4.2 Piè di pagina
Il piè di pagina è così strutturato:

- Nome e versione del documento corrente, posti a sinistra;
- Numerazione progressiva della pagina rispetto al totale posta a destra.

4.1.4.5 Note a piè di pagina
In caso di presenza in una pagina interna di note da esplicare, esse vanno indicate nella pagina corrente, in basso a sinistra. Ogni nota deve riportare un numero e una descrizione.

4.1.5 Versionamento

Tutti i documento, esclusi i verbali, devono essere versionati, affinchè si possa in ogni momento conoscerne la storia. Ogni versione di un documento deve essere appuntata nel Registro delle Modifiche. Si deve adottare il seguente formato per registrare ogni modifica: **vX.Y.Z** dove:

- **X**
  - Parte da 0;
  - Viene incrementato dal *Responsabile di Progetto* in concomitanza con l'approvazione del documento;
  - Arriva fino al numero di revisioni a cui è sottoposto il docuemnto.
- **Y**

  - Parte da 0;
  - Viene incrementato dal $Verificatore_G$ in concomitanza di ogni verifica;
  - Quando X viene incrementato Y ritorna a 0.
- **Z**
  - Parte da 0;
  - Viene incrementato dal *Redattore* in concomitanza di ogni modifica;
  - Quando Y viene incrementato Z ritorna a 0.

4.1.6 Norme tipografiche

Per la redazione della documentazione bisogna attenersi a queste norme.

4.1.6.1 Stile del testo

- **Grassetto:** il grassetto può essere utilizzato nei seguenti casi:
  - **Elenchi puntati:** in questi casi può essere utilizzato il grassetto per evidenziare il concetto sviluppato nella continuazione del punto;
  - **Altri casi:** per evidenziare particolari passaggi o parole chiave.
- **Corsivo:** il corsivo deve essere utilizzato nei seguenti casi:
  - **Citazioni:** quando si deve citare una frase questa va scritta in corsivo;
  - **Abbreviazioni:** quando possibile si deve preferire una parola completa ad un'abbreviazione;
  - **Nomi particolari:** il corsivo deve essere utilizzato quando si parla di figure particolari (es. *Progettista*);
  - **Documenti:** il corsivo deve essere utilizzato quando si parla di documenti (es. *Glossario*);
  - **Altri casi:** in altre situazione, il corsivo va utilizzato per mettere in rilievo passaggi o parole significativi, evidenziare riferimenti ai documenti interni o esterni.
- **Maiuscolo:** l'utilizzo di parole completamente in maiuscolo è riservato solo agli acronimi.

4.1.6.2 Elenchi puntati

Ogni punto dell'elenco deve terminare con un punto e virgola,tranne l'ultimo che deve terminare con un punto. La prima parola deve avere la lettera maiuscola, a meno di casi particolari (es. nome di un file);

4.1.6.3 Formati comuni Per rappresentare le seguenti entità vanno usate le seguenti modalità:

- **Orari: HH:MM**
  - **HH:** va da 0 a 23 e rappresenta l'ora;
  - **MM:** va da 0 a 59 e rappresenta i minuti.
- **Date: AAAA-MM-GG**
  - **AAAA:** rappresenta l'anno;
  - **MM:** rappresenta il mese;
  - **GG:** rappresenta il giorno.
- **Termini ricorrenti:**
  - **Nomi proprio:** ogni nome proprio va scritto con il formalismo *Nome Cognome*;
  - **Ruoli di progetto:** ogni ruolo va scritto con l'iniziale maiuscola e in corsivo;
  - **Nomi dei documenti:** ogni documenti va scritto con l'iniziale maiuscola e in corsivo.

4.1.6.4 Sigle
È previsto l'utilizzo delle seguenti sigle:

- AR: *Analisi dei Requisiti*;
- PP: *Piano di Progetto*;
- NP: *Norme di Progetto*;
- SF: *Studio di Fattibilità*;
- PQ: *Piano di Qualifica*;
- ST: *Specifica Tecnica*;
- MU: $Manualeutente_G$;
- DP: *Definizione di Prodotto*;
- RR: Revisione dei requisiti;
- RP: Revisione di progettazione;
- RQ: Revisione di qualifica;
- RA: Revisione di accettazione;
- Re: *Responsabile* di Progetto;
- Am: *Project manager*;
- An: *Analista*;

- Pt: *Progettista*;
- Pr: $Programmatore_G$;
- Ve: *Verificatore*.

4.1.7 Elementi grafici

4.1.7.1 Tabelle

4.1.7.2 Immagini

4.1.8 Classificazione dei documenti

4.1.8.1 Documenti informali

4.1.8.2 Documenti formali

4.1.8.3 Verbali

4.1.9 Strumenti

4.1.9.1 LaTeX

4.1.9.2 TexStudio

4.1.9.3 Lucidchart

4.2 Verifica

4.2.1 Scopo

4.2.2 Aspettative

4.2.3 Descrizione

4.2.4 Attività

4.2.4.1 Analisi

4.2.4.1.1 Analisi Statica

L'analisi statica è una tecnica utilizzata per identificare errori all'interno di documenti e del codice sorgente, senza la necessità eseguirlo, è applicabile durante tutto il loro ciclo di vita in due diverse modalità:

- **Walkthrough**:
  consiste nel leggere il documento/codice cercando anomalie senza avere un'idea chiara di che tipo di errori possono essere trovati, è un'attività onerosa e non efficiente ma necessaria durante le prime fasi di progetto dove sarà la principale forma di verifica adottata, in quanto non è chiaro fin dall'inizio che tipo di errori si possono fare. Questo metodo è utile per stilare una $checklist_G$ dove verranno archiviati gli errori più comuni. Vista la sua scarsa efficacia normalmente viene effettuata da più persone.
- **Inspection**:
  consiste nella lettura mirata del documento/codice per localizzare possibili errori specificati dalla lista di controllo. Ha un basso costo e diventa più efficace con l'esperienza e l'estensione della lista di controllo, per questo può essere effettuata da una persona sola.

Ogni errore rilevato va discusso con l'autore allo scopo di concordare una modifica.

4.2.4.1.2 Analisi dinamica

L'analisi dinamica è una forma di analisi del software che richiede l'esecuzione del codice sorgente. Viene effettuata tramite dei test che verificano il corretto funzionamento del prodotto, in caso di anomalie questi test aiutano ad identificare l'errore. I test devono essere ripetibili ovvero con lo stesso input e nello stesso ambiente devo essere in grado di ottenere sempre lo stesso output. Per ogni test sono quindi definiti i seguenti parametri:

- **Ambiente**: Sistema hardware e software sul quale si svolgerà il test.
- **Stato iniziale**
- **Input**
- **Output**
- **Istruzioni aggiuntive**: istruzioni su come va eseguito il test e su come vanno interpretati gli output

4.2.4.2 Test

4.2.4.2.1 Test di unità

I test di unità verificano che le singole $unita'_G$ di prodotto software funzionino correttamente, le unità non possono passare al test di integrazione se non passano questo test. Comunemente per i test di unità vengono utilizzati $driver_G$ e $stub_G$ che simulano il chiamante e un unità chiamata.

4.2.4.2.2 Test di integrazione

I test di integrazione sono il passo successivo ai test di unità servono a verificare che due o più unità, precedentemente verificate, funzionino correttamente una volta combinate. Questo tipo di test verifica la corretta collaborazione tra le varie unità e aiuta a trovare eventuali errori non rilevati nei test precedenti. L'obbiettivo finale di questi test è arrivare a testare l'intero prodotto costituito accorpando tutte le singole unità.

4.2.4.2.3 Test di sistema

I test di sistema validano il prodotto software ovvero si accertano che rispettino i requisiti concordati, vengono effettuati quando si ritiene che il prodotto abbia raggiunto una versione definitiva.

4.2.4.2.4 Test di regressione

I test di regressione devono essere fatti ogni volta che viene fatta una modifica a una componente software, consistono nel rifare i test di unità e integrazione necessari per accertarsi che tale modifica non causi errori nella componente in cui è stata fatta o in parti del software collegate. Un buon incapsulamento limita il numero di test da effettuare in questa fase.

4.2.4.2.5 Test di accettazione

Consiste nel collaudo del software in presenza del proponente, se questo test ha esito positivo il prodotto può essere rilasciato.

4.2.5 Strumenti

4.2.5.1 Verifica ortografica Viene utilizzato **GNU Aspell**, uno spell checker open source in grado di gestire più dizionari contemporaneamente.

4.2.5.2 Analisi statica Per l'analisi statica del codice JavaScript viene utilizzato **SonarSource**, questo prodotto è in grado di rilevare bug e problemi di sicurezza, è inoltre disponibile su vari IDE_G come Eclipse e IntelliJ.

4.2.5.3 Analisi dinamica Viene utilizzato **Jest** come piattaforma per l'esecuzione di test su codice JavaScript, comodo per la sua semplicità di utilizzo.

4.2.5.4 Metriche per il calcolo delle metriche viene utilizzato **SonarQube**, software che permette di stabilire un grado di qualità che tutto il gruppo deve rispettare, inoltre permette di visualizzare grafici per controllare la qualità del progetto nel periodo di sviluppo.