softwareonion@gmail.com

Norme di progetto

Informazioni sul Documento

Versione	v4.0.0
Data approvazione	2019-07-12
Responsabili	Matteo Lotto
Redattori	Federico Brian Alessio Lazzaron Nicola Pastore
Verificatori	Federico Omodei
Stato	Approvato
Lista distribuzione	*Onion Software* *prof. Tullio Vardanega* *prof. Riccardo Cardin*

Scopo del Documento

L'attuale documento è adibito alla definizione di norme, convenzioni e modus operandi necessari allo sviluppo del progetto "Butterfly" adottate dal gruppo Onion Software .

Registro delle modifiche

Versione	Data	Autore	Ruolo	Descrizione
v4.0.0	2019-07-12	Matteo Lotto	Responsabile	Documento approvato per il rilascio
v3.1.0	2019-06-17	Federico Omodei	Verificatore	Verifica superata
v3.0.2	2019-06-16	Federico Brian	Amministratore	Fix struttura trattazione processi
v3.0.1	2019-06-15	Federico Brian	Amministratore	Consolidamento §2.1
v3.0.0	2019-05-29	Linpeng Zhang	Responsabile	Documento approvato per il rilascio
v2.1.0	2019-05-28	Nicola Zorzo	Verificatore	Verifica superata
v2.0.5	2019-05-26	Alessio Lazzaron	Amministratore	Correzione errori derivanti dalla verifica
v2.0.4	2019-05-24	Nicola Zorzo	Verificatore	Verifica non superata
v2.0.3	2019-05-23	Federico Brian	Amministratore	Aggiunta §2.2.4.3
v2.0.2	2019-05-21	Alessio Lazzaron	Amministratore	Aggiunte al §2.2.4.2 post valutazione RP
v2.0.1	2019-05-20	Alessio Lazzaron	Amministratore	Rettifica §2.1 post valutazione RP
v2.0.0	2019-05-09	Nicola Pastore	Responsabile	Documento approvato per il rilascio
v1.1.0	2019-05-09	Alessio Lazzaron	Verificatore	Verifica superata
v1.0.4	2019-05-06	Federico Brian	Amministratore	Ristrutturazione generale contenuti
v1.0.3	2019-05-03	Federico Brian	Amministratore	Rettifica §4 post valutazione RR
v1.0.2	2019-05-02	Federico Brian	Amministratore	Rettifica §2 post valutazione RR
v1.0.1	2019-04-25	Federico Brian	Amministratore	Aggiunte al §2.2.4.3
v1.0.0	2019-03-20	Alessio Lazzaron	Responsabile	Documento approvato per il rilascio
v0.1.0	2019-03-19	Matteo Lotto	Verificatore	Verifica superata
v0.0.9	2019-03-18	Federico Brian	Amministratore	Aggiornamento documento ai sensi delle modifiche evidenziate in v0.0.8
v0.0.8	2019-03-16	Matteo Lotto	Verificatore	Verifica non superata, da rivedere i paragrafi §2, §4
v0.0.7	2019-03-15	Nicola Zorzo	Amministratore	Stesura sezione §2 “Processi primari”

– *continua a pagina successiva*

– *continuazione da pagina precedente*

v0.0.6	2019-03-14	Federico Brian	Amministratore	Uniformazione documento agli standard decretati alla riunione del 2019-03-14
v0.0.5	2019-03-13	Nicola Pastore	Amministratore	Aggiunti §3.3 e §3.4
v0.0.4	2019-03-11	Federico Brian	Amministratore	Stesura §3 "Processi di Supporto"
v0.0.3	2019-03-11	Linpeng Zhang	Amministratore	Miglioramento generale e aggiunte sottoparagrafi a §4 "Processi organizzativi"
v0.0.2	2019-03-10	Linpeng Zhang	Amministratore	Stesura §1 "Introduzione" e §4 "Processi organizzativi"
v0.0.1	2019-03-06	Federico Brian	Amministratore	Creazione del documento LaTeX, definizione scheletro

Indice

Elenco delle figure

Elenco delle tabelle

1 Introduzione

1.1 Scopo del documento

Questo documento ha lo scopo di indicare le regole che tutti i componenti di *Onion Software* devono rispettare durante i processi dello sviluppo software per garantire uniformità nel modo di lavorare e nei prodotti finali. Si vuole utilizzare un approccio incrementale, volto a normare passo passo ogni decisione discussa e concordata tra tutti i membri del gruppo. Ciascun componente è tenuto a prendere visione di tale documento e a rispettare le norme in esso descritte allo scopo di perseguire la coesione internamente del gruppo.

1.2 Scopo del prodotto

Il capitolato C1 ha per obiettivo lo sviluppo di un'interfaccia chiamata *Butterfly*. Esso nasce dalla necessità di accentrare e standardizzare le segnalazioni di strumenti di terze parti, i quali Redmine, Gitlab e SonarQube; questi, infatti, forniscono messaggi di notifica specifici e di difficile gestibilità per l'utente finale. Inoltre, in caso di segnalazioni di bug è doveroso assicurarsi che il problema sia risolto tempestivamente, senza che debba accedere ad interfacce specifiche per comprendere. Butterfly si pone lo scopo di risolvere queste problematiche, garantendo una maggiore flessibilità e permettendo una gestione automatizzata e personalizzabile delle segnalazioni.

1.3 Glossario

Al fine di evitare possibili ambiguità relative al linguaggio utilizzato nei documenti formali, viene fornito il *Glossario v3.0.0*. In questo documento vengono definiti e descritti tutti i termini con un significato specifico. Per facilitare la comprensione, i termini saranno contrassegnati da una 'G' a pedice.

1.4 Riferimenti

1.4.1 Riferimenti normativi

- **Capitolato d'appalto C1 - Butterfly, monitor per processi CI/CD**:
  https://www.math.unipd.it/~tullio/IS-1/2018/Progetto/C1.pdf.

1.4.2 Riferimenti informativi

- **Slide L04 del corso Ingegneria del Software - Processi SW**:
  https://www.math.unipd.it/~tullio/IS-1/2018/Dispense/L03.pdf;
- **Slide L05 del corso Ingegneria del Software - Ciclo di vita del software**:
  https://www.math.unipd.it/~tullio/IS-1/2016/Dispense/L05.pdf;
- **Slide L06 del corso Ingegneria del Software - Gestione di progetto**:
  https://www.math.unipd.it/~tullio/IS-1/2018/Dispense/L06.pdf
- **Slide L10 del corso di Ingegneria del Software - Progettazione Software**:
  https://www.math.unipd.it/~tullio/IS-1/2018/Dispense/L10.pdf
- **Slide PD01 del corso di Ingegneria del Software - Regole del progetto didattico**:
  https://www.math.unipd.it/~tullio/IS-1/2018/Dispense/P01.pdf
- **Guide to the Software Engineering Body of Knowledge(SWEBOK), 2004**
  http://www.math.unipd.it/~tullio/IS-1/2007/Approfondimenti/SWEBOK.pdf;
- **Software Engineering, 9th edition (2010), Ian Sommerville**:
  formato cartaceo, numero capitolo di interesse: 2;

- **Documentazione Git**:
  https://www.atlassian.com/git;
- **Camel Case**:
  https://en.wikipedia.org/wiki/Camel_case;
- **Standard ISO/IEC 12207:1995**:
  https://www.math.unipd.it/~tullio/IS-1/2009/Approfondimenti/ISO_12207-1995.pdf;
- **Standard ISO 8601**:
  https://it.wikipedia.org/wiki/ISO_8601;
- **PEP-8 – Style guide for Python Code**:
  https://www.python.org/dev/peps/pep-0008/.

2 Processi primari

2.1 Fornitura

2.1.1 Descrizione

Il *processo di fornitura* descrive i compiti e le attività svolte dal *fornitore*$_G$ al fine di gestire il rapporto con il proponente. Dopo aver esaminato le richieste del proponente, il fornitore presenta uno *studio di fattibilità*$_G$ che, se ritenuto adeguato, porterà alla stipulazione di un contratto tra le due parti: il processo potrà quindi essere avviato, determinando le procedure e le risorse necessarie per gestire ed assicurare lo sviluppo e l'esecuzione del *piano di progetto*$_G$. Esso guiderà il completamento del prodotto software *Butterfly*, rispettando norme e termini precedentemente pattuiti.

2.1.2 Scopo

Il *processo di fornitura* norma tutte le fasi di progettazione, sviluppo e consegna del prodotto *Butterfly* che il gruppo *Onion Software* si impegna a rispettare per rivestire adeguatamente il ruolo di fornitore nei confronti del proponente *Imola Informatica S.p.A.* e dei committenti prof. Tullio Vardanega e prof. Riccardo Cardin.
Il *processo di fornitura* è composto dalle seguenti fasi:

- avvio;
- preparazione delle risposte alle richieste;
- contrattazione;
- pianificazione;
- esecuzione e controllo;
- revisione e valutazione;
- consegna e completamento.

2.1.3 Aspettative

Il gruppo intende instaurare un rapporto di collaborazione continuo con l'azienda *Imola Informatica S.p.A.*, in particolare con il referente sig. Davide Zanetti al fine di:

- identificare gli aspetti chiave del prodotto software;
- determinare i vincoli circa i requisiti ed i processi;
- effettuare una stima dei costi;
- concordare la qualifica del prodotto software.

Onion Software si impegna a mantenere un costante dialogo con il proponente, in modo da ottenere un riscontro efficace circa il lavoro svolto ogniqualvolta sia necessario.

2.1.4 Apprendimento delle tecnologie

Ogni membro del gruppo *Onion Software* si impegna a studiare individualmente ogni tecnologia utilizzata per lo sviluppo del prodotto software, al fine di dominare ogni possibile dubbio e inconsistenza.

2.1.5 Strumenti forniti da *Imola Informatica S.p.A.*

L'azienda proponente, in particolare il sig. Davide Zanetti, ha messo a disposizione un *cluster dati*$_G$ che dispone di 4 CPU e 32GB di RAM. La *virtual machine*$_G$[2] installata, monta il sistema operativo *CentOS7*$_G$, su cui è stato installato Docker. Ad *Onion Software* sono stati forniti i *permessi di amministratore*$_G$ ed è stata data piena libertà d'installazione di eventuali altre componenti software, con la clausola di operare unicamente all'interno di suddetta virtual machine e di non recare danno ad altre directory all'interno del cluster: pena la revoca totale del cluster.

2.1.6 Comunicazione con *Imola Informatica S.p.A.*

Il rapporto con l'azienda proponente dovrà essere costante. Le modalità con cui il gruppo *Onion Software* contatterà l'azienda saranno le seguenti:

- tramite **videochiamata di gruppo** utilizzando il software *hangouts*$_G$ per gli aggiornamenti importanti riguardanti lo sviluppo del prodotto software, come ad esempio una *baseline*$_G$;
- tramite **e-mail** per le comunicazioni informative e non urgenti;
- tramite **telegram** per comunicazioni brevi ed urgenti.

Le comunicazioni con la proponente saranno adibite alla comprensione profonda del capitolato d'appalto ed al consolidamento dei requisiti, contenuti nell'*analisi dei requisiti v3.0.0*. Il team *Onion Software* potrà rivolgersi all'azienda proponente anche per dubbi di natura tecnica, come ad esempio la richiesta di informazioni circa le tecnologie fornite.

2.1.6.1 Negoziazione

Nel corso della realizzazione del progetto può rendersi necessario discutere e negoziare gli accordi da contratto precedentemente presi con il cliente. Questo può accadere, ad esempio, se noi fornitori riteniamo particolarmente complicato il soddisfacimento di alcuni requisiti. In tal caso sarà necessario richiedere un colloquio all'azienda committente, in cui verranno discusse proposte alternative. Se queste dovessero essere accolte da entrambe le parti, i cambiamenti dovranno essere riportati all'interno dell'*analisi dei requisiti v3.0.0*, documento contenente tutti i requisiti e rappresentante il contratto che tutela fornitore e committente.

2.1.6.2 Richiesta materiale

L'azienda proponente può richiedere in ogni momento la visione del codice sorgente del prodotto software, oppure della documentazione esterna. Tale materiale potrà essere fornito in una delle modalità sottostanti:

- tramite tecnologie *cloud*$_G$, quali *Google Drive*$_G$, *GitHub*$_G$ o *GitLab*$_G$;
- tramite supporti fisici ottici, quali CD-ROM oppure DVD-ROM;
- tramite supporto fisico USB.

2.1.7 Comunicazione con i committenti

Il rapporto con i committenti potrà avvenire per i seguenti motivi:

- con il prof. Tullio Vardanega per:
  - chiarire eventuali dubbi ed illustrare le *best practice*$_G$ per lo sviluppo del prodotto software;

[2] altresì definita: *macchina virtuale*

  - mettere a disposizione il materiale didattico, esposto e descritto durante l'A.A. 2018/2019 per il corso di Ingegneria del Software del CdL[3] in Informatica, utile per il corretto sviluppo del prodotto software e per la stesura della documentazione;
  - illustrare l'avanzamento del progetto tramite apposite revisioni, pianificate e descritte nel *piano di progetto v.3.0.0*;

- con il prof. Riccardo Cardin per:
  - chiarire eventuali dubbi architetturali, richiedere suggerimenti utili per la definizione dell'architettura software e per lo sviluppo del codice.

Le modalità con cui il gruppo *Onion Software* contatterà i Committenti saranno le seguenti:

- tramite **videochiamata di gruppo** utilizzando il software *hangouts*$_G$;
- tramite **e-mail**.

2.1.8 Collaudo del prodotto

In ogni momento sarà possibile effettuare il rilascio di un prototipo in produzione: questo assicurerà un dialogo costante con l'azienda *Imola Informatica S.p.A.*, che ci permetterà di avere un riscontro circa il lavoro svolto ed operare di conseguenza. Tale prototipo dovrà necessariamente superare dei test descritti in §3.4.4.2.
L'adeguatezza del prodotto software è garantita dal superamento di questi test ed il materiale da consegnare alla fine dello sviluppo del prodotto comprenderà:

- il **codice sorgente** di *Butterfly*;
- la **documentazione del prodotto**, che includerà i documenti;
  - analisi dei requisiti;
  - piano di progetto;
  - piano di qualifica;
  - glossario;
  - manuale utente;
  - manuale sviluppatore.

Il manuale utente sarà consultato dall'utente finale al fine di garantire una corretta installazione ed uso del prodotto software; il manuale sviluppatore servirà ad agevolare la partecipazione da parte degli sviluppatori interessati al progetto: *Butterfly* è un progetto *open source*$_G$ ed è quindi accessibile a chiunque voglia apportare modifiche/migliorie al codice sorgente.

2.1.9 Preparazione alle revisioni

Ogni revisione necessita di un certo numero di materiale da fornire, come si può evincere dal seguente elenco.

- la lettera di presentazione;
- la documentazione richiesta;
- i diagrammi UML necessari;
- il codice necessario;
- le diapositive per la presentazione;
- (**solo in fase di RQ e RA**): una demo dimostrativa di *Butterfly*.

[3] Corso di Laurea

2.1.10 Verifica a valle di ogni revisione

Dopo ogni revisione di avanzamento, elencate nel §1.5 del *Piano di progetto v4.0.0*, viene assegnata una votazione sulla qualità dei documenti e della presentazione, con relativo commento. Il gruppo *Onion Software* si impegna a porre rimedio ad ogni eventuale valutazione negativa, intervenendo ove necessario. Qualora le valutazioni o i relativi commenti non fossero ben compresi, il team si riserva di contattare il proponente e/o il committente per chiarire ogni dubbio, preferibilmente fissando un colloquio, se possibile.

2.1.11 Rilascio prodotto finale

Una volta completata la fase di Validazione e Collaudo, il prodotto finale verrà rilasciato su un supporto **DVD dati** che comprenderà:

- l'utilità di installazione;
- i manuali d'uso;
- i manuali di collaudo;
- il codice sorgente completo dell'intero prodotto software;
- l'utilità di compilazione.

2.1.12 Delimitazione dei rischi

Numerosi sono i problemi che si possono presentare durante lo sviluppo di un prodotto software come *Butterfly*. È compito del gruppo cercare di intercettare le anomalie in tempo e di riparare quanto danneggiato. Come "ultima spiaggia", il team si riserva di poter contattare il sig. Davide Zanetti, il prof. Tullio Vardanega o il prof. Riccardo Cardin allegando, ove possibile, un'eventuale soluzione.

2.2 Sviluppo

2.2.1 Descrizione

In accordo con lo standard ISO/IEC 12207:1995, il *processo di sviluppo* comprende tutte le attività e i compiti svolti durante la realizzazione del prodotto software, tra cui analisi dei *requisiti*$_G$, design, codifica, integrazione, testing, installazione e validazione finale del software. Gli sviluppatori sono tenuti ad eseguire queste attività in armonia col contratto stipulato.

Il *processo di sviluppo* del software si suddivide fondamentalmente in:

- analisi dei requisiti del prodotto;
- progettazione;
- codifica.

2.2.2 Scopo

Il *processo di sviluppo* attua tutte le attività ed i compiti necessari alla consegna del prodotto richiesto dal proponente.

2.2.3 Aspettative

Le aspettative di *Onion Software* per una corretta implementazione del processo di sviluppo sono le seguenti:

- la realizzazione di un prodotto software che sia:
  - conforme alle richieste del proponente;
  - che supera i test di verifica in modo soddisfacente;
  - che supera i test di validazione in modo soddisfacente;
- la creazione di obiettivi di *sviluppo*$_G$;
- la creazione di vincoli:
  - tecnologici;
  - di design.

2.2.4 Attività

2.2.4.1 Analisi dei requisiti

L'analisi dei requisiti è un'attività preliminare allo sviluppo (o alla modifica) di un sistema software e avviene normalmente come negoziazione tra analisti e clienti. Tale dialogo non è semplice: gli analisti possono aver difficoltà a comprendere il linguaggio ed il contesto culturale del cliente e viceversa; inoltre lo stesso cliente potrebbe aver difficoltà a mettere a fuoco i propri reali bisogni e di conseguenza le richieste o le proposte da mettere sul tavolo della discussione.

Le aspettative di questa attività sono, quindi, la corretta determinazione dei requisiti di vincolo, funzionali, prestazionali e di qualità del prodotto software. Tali requisiti saranno contenuti in un documento redatto dagli analisti, versionato e denominato *analisi dei requisiti*. I requisiti, quindi, devono essere estrapolati da più fonti:

- dal capitolato d'appalto;
- dai verbali di riunioni esterne ed interne;
- dai casi d'uso.

Il documento nato dall'analisi dei requisiti ha il compito di individuare i diversi requisiti (diretti ed indiretti) che il proponente richiede. I dati raccolti dagli analisti andranno a definire la struttura del documento, il cui scopo sarà la definizione delle funzionalità che il nuovo prodotto deve offrire, ovvero i requisiti che devono essere soddisfatti dal software che sarà sviluppato. Pertanto, il documento denominato *analisi dei requisiti* dovrà contenere:

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

Il documento, inoltre, dovrà:

- fornire ai progettisti riferimenti chiari ed adeguati;
- fissare le funzionalità e i requisiti concordati col proponente;
- fornire una base per raffinamenti successivi al fine di garantire un miglioramento continuo del prodotto e del processo di sviluppo;
- fornire ai verificatori riferimenti per l'attività di controllo dei test;
- calcolare la mole di lavoro per una stima dei costi e del personale necessario.

2.2.4.1.1 Casi d'uso

I casi d'uso sono una tecnica utilizzata per effettuare in maniera esaustiva e non ambigua la raccolta dei requisiti, al fine di produrre software di qualità. Essa consiste nel valutare ogni requisito focalizzandosi sugli attori che interagiscono col sistema, valutandone le interazioni. Un caso d'uso, quindi, è un insieme di scenari (sequenze di azioni) che hanno in comune uno scopo finale (obiettivo) per un utente (attore).
Il linguaggio che *Onion Software* utilizzerà per la rappresentazione dei diagrammi dei casi d'uso è **UML** (*Unified Modeling Language*) versione 2.0.
Ogni caso d'uso sarà identificato nel modo seguente:

`UC[codicePadre].[codiceFiglio]`

dove:

- **UC**: è l'acronimo di *use case*;
- **codicePadre**: è un numero che identifica in modo univoco un caso d'uso;
- **codiceFiglio**: è un numero che identifica in modo univoco un sottocaso d'uso del caso padre (può includere altri sottocasi propri).

Oltre al codice identificativo, la specifica di ogni caso d'uso dovrà riportare:

- **nome**: nome informale attribuito al caso d'uso;
- **attori**: nominano gli attori sia principali che secondari che partecipano al caso d'uso;
- **obiettivo**: descrive il motivo per il quale gli attori principali avviano il caso d'uso;
- **precondizioni**: descrivono le condizioni che devono essersi verificate per permettere l'esecuzione del caso d'uso;
- **postcondizioni**: descrivono le condizioni che devono essersi verificate dopo l'esecuzione del caso d'uso;
- **azioni**: elencano la sequenza delle azioni svolte dagli attori e dal sistema all'interno del caso d'uso;
- **inclusioni**: descrivono il comportamento in comune con un altro caso d'uso, utilizzate per non descrivere più volte lo stesso flusso di eventi;
- **estensioni**: elencano i casi d'uso generati dal caso d'uso in analisi;
- **diagramma**: diagramma UML utilizzato per rappresentare il caso d'uso.

2.2.4.1.2 Requisiti

Come i casi d'uso, ogni requisito verrà identificato nel modo seguente:

`R[tipologia][importanza][codice]`

dove:

- **tipologia**: rappresenta la tipologia del requisito, può assumere i valori:
  - **V**: requisito di *vincolo*, che deve essere soddisfatto per garantire una funzionalità richiesta dal proponente;
  - **F**: requisito *funzionale*, che definisce la funzione di uno o più componenti del sistema, la tipologia degli ingressi e delle uscite, nonché il comportamento;
  - **P**: requisito *prestazionale*, che descrive il modo di operare di un componente o dell'intero sistema per ottimizzare l'utilizzo delle risorse disponibili;

  - **Q**: requisito di *qualità*, che descrive la chiarezza e l'accessibilità del servizio.

- **importanza**: descrive l'importanza del requisito preso in esame, può assumere i valori:
  - **O**: requisito *obbligatorio*, che deve necessariamente essere soddisfatto per garantire le funzioni di base del prodotto;
  - **D**: requisito *desiderabile*, che non influisce sulle funzioni di base del prodotto ma garantisce una maggiore completezza;
  - **F**: requisito *facoltativo*, che non influisce sulle funzioni di base del prodotto, può garantire una maggiore completezza ma allo stesso tempo può causare il dispendio di ulteriori risorse, con conseguente aumento dei costi;
- **codice**: indica il codice univoco con cui è rappresentato un requisito, nella classica forma riportata in §2.2.4.1.1.

Inoltre, la specifica di ogni requisito riporterà una breve ma concisa descrizione e le fonti da cui è stato ricavato.

2.2.4.2 Progettazione

L'attività di progettazione si prefissa di individuare l'architettura software più adatta, come essa verrà implementata, i modelli di dati usati dal sistema, le interfacce per comunicare tra componenti diversi e gli algoritmi utilizzati. I progettisti non sviluppano immediatamente la struttura finale del prodotto, ma la modificano e aggiungono dettagli man mano che procedono con la progettazione del sistema, seguendo il modello incrementale: una prima istanza di questa architettura è attuata dal Proof of Concept della Technology Baseline. Tale architettura basilare subirà degli incrementi, che ne assicureranno l'adeguatezza e che soddisfaranno sempre più requisiti. L'architettura ottenuta sarà successivamente descritta in modo approfondito nell'allegato tecnico in sede di *Product Baseline*$_{\boldsymbol{G}}$.

2.2.4.2.1 Tipi di progettazione

Si possono identificare quattro tipi di progettazione:

- **architetturale**: dove si identifica la struttura complessiva del sistema, i componenti principali, le loro relazioni e come sono distribuiti;
- **database**: dove si progettano le strutture dati da utilizzare e come esse verranno rappresentate in un database;
- **interfaccia**: dove si definiscono le interfacce da utilizzare per i vari componenti di sistema;
- **componenti**: dove si cercano componenti riutilizzabili e, se non ce ne sono disponibili, se ne progettano di nuovi.

2.2.4.2.2 Fine ultimo di un'architettura software

L'architettura definita deve:

- soddisfare i requisiti elencati nell'*analisi dei requisiti*;
- essere comprensibile e modulare per facilitarne la manutenzione;
- essere sicura rispetto ad intrusioni e malfunzionamenti;
- essere affidabile e robusta, per svolgere al meglio il compito per la quale è stata creata e saper gestire situazioni anomale;
- avere componenti semplici, coesi, volti a raggiungere l'obiettivo finale e, ove possibile, con scarse dipendenze tra loro.

2.2.4.2.3 Qualità di un'architettura software

L'architettura definita deve possedere le virtù elencate nella tabella sottostante.

Nome	Descrizione
Sufficienza	È capace di soddisfare tutti i requisiti descritti nell'*analisi dei requisiti v3.0.0* garantendo versatilità, poiché alcuni possono variare nel tempo
Comprensibilità	Gli stakeholders devono essere in grado di comprenderne il funzionamento senza ambiguità
Modularità	È suddivisa in parti ben chiare e distinte, senza sovrapposizioni di funzionalità: solo in questo modo si garantisce la *separation of concerns*$_G$ nonché l'esecuzione dei task. Una modularità efficace si ottiene suddividendo l'attività nei suoi blocchi logici principali (e.g. gli stadi di una pipeline) e perseguendo il principio dell'*information hiding*$_G$
Robustezza	È capace di sopportare ingressi diversi di qualsiasi natura dall'utente e dall'ambiente senza incorrere stati d'errore non previsti
Flessibilità	Permette modifiche a costo contenuto al variare dei requisiti presenti nell'*analisi dei requisiti*
Riusabilità	le singole componenti dell'architettura possono essere utilmente impiegate in altre parti del sistema
Efficienza	L'architettura funziona e soddisfa i requisiti con un commisurato dispendio di risorse
Affidabilità (**reliability**)	Svolge bene i suoi compiti
Disponibilità (**availability**)	Necessita di zero o poco tempo di indisponibilità totale per manutenzione: non tutto il sistema deve essere interrotto se qualche sua parte è sotto intervento
Sicurezza rispetto a malfunzionamenti (**safety**)	È esente da malfunzionamenti gravi anche grazie ad un opportuno grado di ridondanza per restare operativo anche in caso di guasti locali
Sicurezza rispetto ad intrusioni (**security**)	I suoi dati e le sue funzioni non sono vulnerabili ad intrusioni
Dipendenze minimizzate	Evita il propagarsi delle modifiche causato da una singola modifica effettuata in un'altra parte del sistema
Semplicità	Ogni parte contiene il necessario, nulla di superfluo, seguendo il *principio KISS*$_G$
Incapsulazione (**information hiding**)	L'interno delle componenti di ogni singola parte dell'architettura non deve essere visibile all'esterno
Coesione	Le componenti che compongono la stessa parte di architettura hanno gli stessi obiettivi
Basso accoppiamento	Le parti distinte dipendono il meno possibile l'una dalle altre

Tabella 2: Qualità di un'architettura software

2.2.4.2.4 Test

In fase di progettazione architetturale e progettazione di dettaglio e codifica, i progettisti avranno il compito di sviluppare opportuni test, accompagnati da classi utili ad individuare eventuali errori ed anomalie. Tali test dovranno rispettare le convenzioni di nomenclatura illustrate nel paragrafo §3.4.4.2.

2.2.4.2.5 Diagrammi

Onion Software utilizzerà lo strumento *Astah UML* e la notazione offerta da *UML 2.0* allo scopo di rendere chiare e senza ambiguità le scelte progettuali adottate. Fanno parte di questa notazione i diagrammi descritti nella tabella a seguire.

Nome	Descrizione
Diagrammi delle attività	Descrivono il flusso di operazioni che definiscono un'attività attraverso una logica procedurale come, ad esempio, un algoritmo
Diagrammi delle classi	Descrivono una collezione di elementi di un modello: vengono definite le classi, i tipi, i metodi, gli attributi e le relazioni che vi intercorrono, senza specificare alcun linguaggio di programmazione
Diagrammi dei package	Descrivono raggruppamenti di classi di unità che verranno riusate in modo unito
Diagrammi di sequenza	Descrivono le iterazioni tra classi ed istanze di classi che implementano uno specifico comportamento
Diagrammi dei casi d'uso	Descrivono nel dettaglio le funzionalità dell'intero sistema
Diagrammi di macchine di stato	Modellano il comportamento di un singolo oggetto specificando la sequenza di eventi a cui è soggetto durante il suo ciclo di vita, in risposta agli eventi che subisce
Diagrammi di componenti	Descrivono le relazioni fra le differenti componenti del sistema
Diagrammi di deployment	Descrive l'esecuzione dell'architettura del sistema, comprensiva di componenti hardware, software e *middleware*$_G$ che li connettono

Tabella 3: Diagrammi UML 2.0

2.2.4.2.6 Diagrammi dei package

Questi diagrammi, insieme ai diagrammi delle classi, raggruppano gli elementi in un elementi di livello più alto. Nella progettazione adottiamo il principio del common closure principle, secondo il quale elementi dello stesso package condividono lo stesso motivo di cambiamento. Per identificare le dipendenze tra i package si utilizzeranno frecce tratteggiate. I vantaggi nell'utilizzo di questi diagrammi sono:

- l'individuazione immediata delle dipendenze circolari, evitandole;
- l'individuazione immediata degli elementi più stabili, ovvero quelle con più dipendenze entranti.

2.2.4.2.7 Diagrammi delle classi

Questi diagrammi strutturali descrivono ad alto livello le classi del sistema e le relazioni che intercorrono tra essi. Tali diagrammi sono rappresentate da un rettangolo che può contenere le seguenti informazioni:

- attributi: rappresentano i campi dato che ha una classe. Questi devono avere un nome, un tipo e opzionalmente una molteplicità;
- metodi: rappresentano le operazioni fornite dalla classe. Questi hanno una visibilità, un nome, una lista di parametri ed un tipo di ritorno.

Inoltre tra le classi rappresentiamo le seguenti relazioni di dipendenza:

- dipendenza: è rappresentata da una linea orientata tratteggiata che indica che la classe sorgente usa in modo non continuativo la classe d'arrivo. Ad esempio, il tipo della classe d'arrivo può essere il tipo di un parametro di un metodo della classe sorgente;
- associazione: è rappresentata da una linea orientata continua, indica che la classe sorgente ha un attributo del tipo della classe d'arrivo.
- aggregazione: è rappresentata da un rombo vuoto seguito da una linea continua, indica che la classe puntata dal diamante possiede un riferimento o un'istanza della classe d'arrivo tra gli attributi. Tale istanza o il suo riferimento possono essere condivisi.
- composizione: è rappresentata da un rombo pieno seguito da una linea continua, indica che la classe puntata dal diamante possiede un riferimento o un'istanza della classe d'arrivo tra gli attributi. Tale istanza o il suo riferimento non possono essere condivisi ad altre classi.
- generalizzazione: è rappresentata da una linea orientata continua la cui punta è una freccia vuota, indica che ogni classe sorgente estende la classe d'arrivo.

Una classe può essere suddivisa in:

- interfaccia: è rappresentata o da una sfera vuota con il nome dell'interfaccia, o da un rettangolo con la dicitura <<*interface*>>. Possono avere metodi che dovranno essere implementati da classi concrete, ma non possono avere attributi, e naturalmente un'interfaccia non può essere istanziata. Utilizzeremo la rappresentazione rettangolare per evidenziare la presenza di metodi;
- astratta: è rappresentata da un rettangolo il cui nome è indicato in corsivo. Come le interfacce non possono essere istanziate perché presentano almeno un'operazione astratta (con il nome in corsivo. Possono avere attributi;
- concreta: è rappresentata da un rettangolo. Proprietà e dipendenze sono rappresentate utilizzando le regole precedentemente elencate.

2.2.4.2.8 Diagrammi di sequenza

Questi diagrammi mostrano come alcuni oggetti collaborano tra di loro per svolgere determinate funzioni all'avanzare del tempo dall'alto al basso. I costrutti utilizzati sono:

- partecipanti: sono rappresentati da un rettangolo contenente il nome dell'istanza di una classe, il nome della classe e una barra di attivazione (rettangolo che si prolunga per tutta la durata in cui l'entità è attiva). Per facilitarne la lettura, i nomi delle istanze saranno i nomi delle classi precedute da una "a".
- messaggi: rappresentano le operazioni e il flusso dei dati tra i partecipanti. Possono essere suddivisi in:
  - sincroni: sono 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 una risposta dall'entità chiamata.

  - asincroni: sono 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 una risposta dall'entità chiamata, proseguendo con le sue altre funzionalità.

- ritorno: sono rappresentati da una linea tratteggiata orientata e indicano un messaggio di ritorno che risponde a un precedente messaggio di chiamata.

- creazione: sono rappresentati da una linea tratteggiata orientata e dalla notazione *<<create>>* . Indicano la creazione di una nuova entità da parte dell'entità chiamante.

- distruzione: sono rappresentati da una linea continua orientata e dalla notazione *<<destroy>>*. Indicano la distruzione di un'entità da parte dell'entità chiamante.

2.2.4.2.9 Architettura

L'architettura del software indica come si relazionano i vari elementi software, senza indicare come questi siano formati. Rappresentano quindi la visione più ad alto livello del software. In base al tipo di software che si vuole sviluppare si sceglie una delle seguenti architetture:

- layer: architettura che suddivide il software in più strati o livelli disposti uno sopra l'altro. Uno strato deve dipendere solamente da quello sottostante, comunicando con esso;

- event-driven: architettura i cui elementi software comunicano attraverso messaggi asincroni, ovvero elaborati in modo indipendente. Quest'architettura può essere sviluppata secondo le seguenti topologie:

  - *Mediator topology*$_G$: i messaggi vengono inseriti in una coda, smistati attraverso un mediatore in diversi canali e poi elaborati.
  - *Broker topology*$_G$: i messaggi viaggiano attraverso un broker che possiede più canali, per poi essere elaborati.

- microservizi: architettura che suddivide il software in diversi moduli indipendenti che comunicano attraverso *API REST*$_G$.

2.2.4.2.10 Design pattern

I *design pattern*$_G$ sono buone pratiche di sviluppo a problemi ricorrenti, ma la loro presenza non garantisce una buona struttura del software e allo stesso modo la loro assenza non implica una cattiva architettura software. I design pattern devono quindi essere applicati solamente se ben adatti al contesto. Questi si divono nelle seguenti categorie:

- strutturali: riguardano la struttura delle classi favorendo la loro comunicazione attraverso interfacce. Tra i pattern strutturali più conosciuti vi sono l'Adapter, il Decorator il Facade e il Proxy.

- creazionali: aiutano a separare una classe dalla sua istanziazione favorendo *information hiding*$_G$. Tra i pattern creazionali più conosciuti vi sono il Singleton, il Builder, il Factory Method e l'Abstract Factory.

- comportamentali: applicabili nel momento in cui un componente deve eseguire una specifica funzione per cui il design pattern è stato costruito. Tra i pattern comportamentali più conosciuti vi sono il Command, l'Iterator, l'Observer, lo Strategy e il Template Method.

- architetturali: strutturano il software con una visione ad alto livello. Tra i pattern architetturali più conosciuti vi sono il Dependency Injection e i Model-View pattern.

2.2.4.2.11 The Twelve-Factor App

Su richiesta del proponente e per quanto applicabile al componente specifico, *Onion Software* si impegnerà a seguire una metodologia di sviluppo orientata alla costruzione di applicazioni *software-as-a-service*$_G$ come indicato in The Twelve-Factor App.
Le applicazioni costruite in questo modo:

- seguono un formato dichiarativo per l'automazione della *configurazione*$_G$, minimizzando tempi e costi di ingresso per ogni sviluppatore che si aggiunge al progetto;
- si interfacciano in modo pulito con il sistema operativo sottostante, in modo tale da offrire la massima portabilità sui vari ambienti di esecuzione;
- sono adatte allo sviluppo sulle più recenti cloud platform, ovviando alla necessità di server e amministrazioni di sistema;
- minimizzano la divergenza tra sviluppo e produzione, permettendo il continuous *deployment*$_G$ per una massima "agilità";
- possono scalare significativamente senza troppi cambiamenti ai tool, all'architettura e al processo di sviluppo.

La metodologia twelve-factor può essere applicata a ogni software, scritto in qualsiasi linguaggio di programmazione, che fa uso di una serie di servizi come database, code, cache e così via.

2.2.4.3 Codifica

L'attività di codifica è la realizzazione effettiva dell'attività di progettazione. In questa fase si concretizza la soluzione attraverso la programmazione per ottenere il prodotto software finito. Come linguaggio di programmazione è stato scelto *Python*$_G$. Alcuni fattori-chiave che hanno determinato questa scelta sono: la scalabilità, la semplicità d'uso e la grande richiesta nel mondo del lavoro. Al fine di garantire una scrittura uniforme e leggibile è stata adottata la convenzione *PEP-8*$_G$, e per i commenti multilinea la convenzione *PEP-257*$_G$. Saranno presentati di seguito diversi esempi divisi per categoria, in modo da illustrare la corretta maniera di scrittura del codice.

2.2.4.3.1 Lingua di scrittura del codice

Funzioni, variabili, classi, metodi ed attributi di classe, costanti, moduli e package saranno denominati in lingua **inglese**. Tale convenzione dovrà essere rispettata anche per i commenti.

2.2.4.3.2 Indentazione

Usare *quattro* spazi per livello di indentazione. Il tasto *tab* per l'indentazione non è ammesso: si consiglia di configurare *PyCharm*$_G$ al fine di assicurare il rispetto di tale regola.

```
#Allineato con il delimitatore d'apertura

foo = long_function_name(var_one, var_two,
                         var_three, var_four)


#Aggiungi un ulteriore livello di indentazione per distinguere
#gli argomenti dal resto

def long_function_name(
        var_one, var_two, var_three,
        var_four):
```

```
    print(var_one)


#Le indentazioni pendenti dovrebbero aggiungere un livello

foo = long_function_name(
    var_one, var,two,
    var_three, var_four)
        print(var_one)
```

Listing 1: Indentazione SI

```
#Gli argomenti nella prima linea sono proibiti quando non
#si usa l'allineamento verticale

foo = long_function_name(var_one, var_two,
    var_three, var_four)


#Indentazione non distinguibile: serve un ulteriore livello

def long_function_name(
    var_one, var_two, var_three,
    var_four):
    print(var_one)
```

Listing 2: Indentazione NO

ECCEZIONE: le indentazioni pendenti possono essere indentate anche con un numero arbitrario di spazi, purché siano distinguibili dagli altri livelli.

```
#Le parentesi possono chiudere la struttura allineate con il primo
#elemento dell'ultima riga

my_list = [
    1, 2, 3,
    4, 5, 6,
    ]

result = some_function_that_takes_arguments(
    'a', 'b', 'c',
    'd', 'e', 'f',
    )


#...oppure con la prima lettera del nome della struttura dati

my_list = [
    1, 2, 3,
    4, 5, 6,
]

result = some_function_that_takes_arguments(
    'a', 'b', 'c',
    'd', 'e', 'f',
)
```

Listing 3: Indentazione parentesi

2.2.4.3.3 Lunghezza di riga

Ogni riga può essere lunga al massimo *settantanove* caratteri, mentre per i commenti il limite è di *settantadue*. Porzioni lunghe di codice possono essere suddivise in più linee utilizzando il simbolo "\" come in questo esempio:

```
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())
```

Listing 4: suddivisione riga

2.2.4.3.4 Operatori binari: interruzione di linea

```
income = (gross_wages
    + taxable_interest
    + (dividends
      - qualified_dividends)
    - ira_deduction
    - student_loan_interest)
```

Listing 5: Interruzione di linea SI

```
income = (gross_wages +
    taxable_interest +
    (dividends -
       qualified_dividends) -
    ira_deduction -
    student_loan_interest)
```

Listing 6: Interruzione di linea NO

2.2.4.3.5 Righe vuote

La distribuzione di righe vuote sarà avverrà in questo modo:

- la definizione di funzioni astratte e classi vanno circondate da due righe bianche;
- la definizione di metodi interni ad una classe va circondata da una riga bianca;
- righe bianche extra possono essere usate con parsimonia per:
  - separare gruppi di funzioni in relazione;
  - indicare sezioni logiche.

2.2.4.3.6 Imports

Di seguito, un confronto su come gestire il comando `import` (a sinistra) e su come non gestirlo (a destra).

```
import os
import sys
```

Listing 7: Imports SI

```
import os, sys
```

Listing 8: Imports NO

2.2.4.3.7 Codifica del codice sorgente

La codifica del codice sorgente, in accordo con lo standard Python 3.7.3 utilizzato da *Onion Software*, deve essere **UTF-8**.

2.2.4.3.8 Apici delle stringhe

```
my_string = "foo"
```

Listing 9: Apici SI

```
my_string = 'foo'
```

Listing 10: Apici NO

2.2.4.3.9 Commenti monolinea

I commenti situati in una singola linea prevedono l'uso di un *hash*: "**#**" seguito da uno spazio. I commenti relativi ad una porzione di codice sono indentati coerentemente con il codice. I commenti possono essere composti da più righe: in questo caso ogni riga comincerà con un **#** seguita da uno spazio.
I commenti *inline* non sono ammessi in quanto ritenuti non necessari e, di fatto, oggetto di distrazioni. Di seguito sono riportati alcuni esempi d'uso e non-uso.

```
# Increment x
x = x + 1

# Function that does something
def some_function(var_1, var_2)
    if var_1 == var_2
        # Print if equal
        # Just another line
        # and another
        print("They are equal")
```

Listing 11: Commenti SI

```
x = x + 1    #Increment x

#Function that does something
def some_function(var_1, var_2)
    if var_1 == var_2
#Print if equal
#Just another line of comments
        print("They are equal"
```

Listing 12: Commenti NO

2.2.4.3.10 Commenti multilinea

La convenzione **PEP-257** suggerisce l'uso di *docstring*$_G$ ogniqualvolta si presenti la necessità di scrivere un commento piuttosto verboso, come ad esempio prima della definizione di una classe. Di seguito è illustrato com'è opportuno implementare tale tipologia di commento.

```
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)
    """

    if imag == 0.0
      and real == 0.0:
        return complex_zero
    ...
```

Listing 13: Commenti SI

```
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)
    """
    if imag == 0.0
      and real == 0.0:
        return complex_zero
    ...
```

Listing 14: Commenti NO

2.2.4.3.11 Spaziature

Evitare strane spaziature:

- dentro alle parentesi:

```
spam(ham[1], {eggs: 2})
```

Listing 15: SI

```
spam( ham[ 1 ], { eggs: 2 } )
```

Listing 16: NO

- dopo una virgola che si trova dentro ad una parentesi:

```
foo = (0,)
```

Listing 17: SI

```
bar = (0, )
```

Listing 18: NO

- immediatamente prima di una virgola, un punto e virgola o due punti:

```
if x == 4: print x, y;
  x, y = y, x
```

Listing 19: SI

```
if x == 4 : print x , y ;
  x , y = y , x
```

Listing 20: NO

- dentro a parentesi quadre:

```
ham[1:9], ham[1:9:3],
  ham[:9:3], ham[1::3],
  ham[1:9:]

ham[low:up],
  ham[low:up:],
  ham[low::step]

ham[low+off : up+off]

ham[: up_fn(x) : step_fn(x)],
  ham[:: step_fn(x)]
ham[low + off : up + off]
```

Listing 21: SI

```
ham[low + offt:up + off]

ham[1: 9], ham[1 :9], ham[1:9 :3]

ham[low : : up]

ham[ : up]
```

Listing 22: NO

- immediatamente prima di aprire una parentesi tonda che contiene la lista dei parametri attuali di una funzione:

```
spam(1)
```

Listing 23: SI

```
spam (1)
```

Listing 24: NO

- immediatamente prima di aprire una parentesi quadra:

```
dct['key'] = lst[index]
```

Listing 25: SI

```
dct ['key'] = lst [index]
```

Listing 26: NO

- con operatori di assegnazione e non:

```
x = 1
y = 2
long_variable = 3
i = i + 1
submitted += 1
x = x*2 - 1
hypot2 = x*x + y*y
c = (a+b) * (a-b)
```

Listing 27: SI

```
x              = 1
y              = 2
long_variable = 3
i=i+1
submitted +=1
x = x * 2 - 1
hypot2 = x * x + y * y
c = (a + b) * (a - b)
```

Listing 28: NO

2.2.4.3.12 Convenzioni di denominazione

Di seguito sono delineate le norme di denominazione con relative casistiche d'uso.

Tipo	Norma di denominazione	Esempio
Funzioni	Usare parole in minuscolo separate da un *underscore*	`function`, `my_function`
Variabili	Usare lettere o parole in minuscolo separate da un underscore	`x`, `var`, `my_var`
Classi	Adottare stile *CamelCase* con prima lettera maiuscola e parole unite	`Class`, `MyClass`
Metodi di classe	**Privati:** i nomi saranno preceduti da un *underscore*, scritti in minuscolo e conterranno parole separate da un *underscore* **Pubblici:** usare lettere o parole in minuscolo separate da un underscore	`_method`, `_class_method` `method`, `class_method`
Costanti	Usare lettere o parole in maiuscolo separate da un underscore	`CONSTANT`, `PLANCK_CONSTANT`, `MY_LONG_CONSTANT`
Moduli	Usare lettere o parole in minuscolo separate da un underscore	`module.py`, `my_module.py`
Package	Usare parole in minuscolo unite	`package`, `mypackage`
Attributi di classe	**Privati:** i nomi saranno preceduti da un *underscore*, scritti in minuscolo e conterranno parole separate da un *underscore* **Pubblici:** usare lettere o parole in minuscolo separate da un underscore	`_attribute`, `_class_attribute` `attribute`, `class_attribute`

Tabella 4: Convenzioni di denominazione

ATTENZIONE: le denominazioni non useranno termini fuorvianti e che non identificano univocamente e il dato: ad esempio variabili denominate come `x` e `temp` sono proibite perché poco identificative. Nondimeno, è proibito l'utilizzo delle seguenti denominazioni: `l`, `O`, `o` perché poco distinguibili dai numeri uno e zero.

2.2.4.3.13 Strumenti

Di seguito saranno elencati gli strumenti utilizzati da *Onion Software* per lo sviluppo del prodotto software, indicandone il nome seguito dalla versione scelta.

- **Pycharm 2019.1.1**
  Come ambiente di sviluppo integrato del prodotto *Butterfly* è stato scelto PyCharm. È possibile eseguire il download del suddetto software prima iscrivendosi al sito come studenti universitari con il seguente link:

  https://www.jetbrains.com/shop/eform/students

  e successivamente procedendo all'installazione secondo il sistema operativo presente nella propria macchina:

  - **Windows** (versioni 7/8/8.1/10):

    https://www.jetbrains.com/pycharm/download/download-thanks.html?platform=windows

  - **Linux**: da riga di comando, digitare:

    ```
    sudo snap install pycharm-professional --classic
    ```

  È necessario disporre dei permessi di *root*$_G$ per eseguire l'installazione in ambiente Linux.

- **SonarQube 6.7.x**
  Utilizzato per eseguire automaticamente l'analisi statica del codice al fine di identificare eventuali *bug*$_G$, duplicazioni di codice e violazioni a standard di codifica.

  È possibile eseguire il download del software al seguente indirizzo:

  https://www.sonarqube.org/downloads/

  estrarre il contenuto dalla cartella appena scaricata e procedere all'installazione del server SonarQube secondo il sistema operativo presente nella propria macchina:

  - **Windows** (versioni 7/8/8.1/10):
    eseguendo il file individuato in

    ```
    C:\path\del\download\sonarqube\bin\windows-x86-xx\StartSonar.bat
    ```

  - **Linux**
    da riga di comando, spostarsi nella directory dove si è scaricato SonarQube e digitare:

    ```
    /sonarqube/bin/[OS]/sonar.sh console
    ```

  Successivamente, seguire le indicazioni di configurazione elencate all'indirizzo

  https://docs.sonarqube.org/latest/setup/install-server/

- **Docker Community 18.09.2**
  Utilizzato per automatizzare il *deployment*$_G$, Docker crea e gestisce container personalizzabili ed indipendenti che ne permettono l'esecuzione sulla stessa istanza di sistema operativo.
  Link di riferimento:

  https://www.docker.com

- **MongoDB 4.0**
  Il database documentale MongoDB è stato scelto da *Onion Software* perché offre scalabilità orizzontale, una semplice manipolazione dei dati e performance migliori. Si interfaccia e funziona con il linguaggio di programmazione Python grazie ai *package*$_G$ **PyMongo** e **mongoengine**.
  Link per il download:

  https://www.mongodb.com/download-center/community

- **Flask 1.0.2**
  Flask è un *microframework*$_G$ per Python, utilizzato da *Onion Software* in quanto risultava la scelta più adatta al contesto applicativo del nostro capitolato. Facile e flessibile da utilizzare, è utilizzato per:

  – la costruzione dell'interfaccia web dell'applicativo;
  – la comunicazione tra front-end e back-end;
  – l'ascolto degli eventi provenienti dai *producer*$_G$.

  Per il download, digitare da riga di comando:

  ```
  pip install -U Flask
  ```

- **Apache Kafka 2.2.0**
  Usato come piattaforma distribuita di streaming, Apache Kafka è in grado di:

  – pubblicare o sottoscriversi a flussi di informazioni, in modo analogo ad una coda di messaggi;
  – memorizzare flussi di informazioni in modo durevole e tollerante agli errori;
  – processare flussi di informazioni man mano che si verificano.

  Link per il download:

  https://www.apache.org/dyn/closer.cgi?path=/kafka/2.2.0/kafka-2.2.0-src.tgz

3 Processi di supporto

3.1 Documentazione

3.1.1 Scopo

Il *processo di documentazione* è una metodologia attuata per tenere traccia delle informazioni prodotte dal ciclo di vita di un processo od un'attività. Tali informazioni permettono di individuare norme per la stesura di una documentazione valida e coerente, con una struttura fissata.

3.1.2 Processo di documentazione

In accordo con lo standard ISO/IEC 12207:1995 che *Onion Software* ha deciso di adottare come riferimento informativo, il processo di documentazione è così strutturato:

- implementazione;
- progettazione e sviluppo;
- produzione;
- mantenimento.

Conoscere lo sviluppo e lo svolgimento di tali attività è di grande importanza per tutti gli interessati al prodotto software.

3.1.2.1 Implementazione

Con *implementazione* ci si riferisce alla suddivisione a cui sono soggetti i documenti da redigere lungo l'intero ciclo di vita del prodotto software. Tale suddivisione comprende due macro-categorie: documenti formali ed informali.

3.1.2.1.1 Documenti formali

Un documento viene definito *formale* solo se è *versionato*$_G$ e approvato dal *responsabile di progetto*. I documenti che soddisfano entrambi i requisiti sono da intendersi idonei alla distribuzione. Un documento formale che subisce modifiche è da intendersi informale fintanto che il *responsabile di progetto* non lo approva. Un documento formale con lo stesso nome può presentare più *versioni*$_G$ con lo stesso nome: in questo caso verrà considerato solo il documento avente il numero di versione più alto, le restanti versioni saranno classificate come obsolete.
I documenti formali vengono a loro volta suddivisi in due categorie:

- **documenti formali interni:** documentazione utile al team, di poco interesse per committenti e proponenti. I documenti formali interni comprendono:
  - lo **studio di fattibilità**, in cui vengono analizzati i diversi capitolati d'appalto. Il documento è comprensivo di motivazione per il quale si è scelto un capitolato rispetto ad altri, evidenziando per ognuno dei capitolati i rispettivi punti di forza e punti deboli.
  - le **norme di progetto**, in cui vengono decretate le regole e gli standard che il team ha scelto di adottare durante l'intera durata dello sviluppo software.
- **documenti formali esterni:** documentazione che verrà fornita ai Committenti ed il Proponente. I documenti formali esterni comprendono:
  - il **glossario**, che contiene una raccolta di termini con relativa spiegazione utilizzati nella documentazione che, potenzialmente, possono generare ambiguità;

- l'**analisi dei requisiti**, in cui vengono definiti i requisiti che devono essere soddisfatti dal prodotto software;
- il **piano di qualifica**, documento che descrive le operazioni di verifica e validazione seguite durante lo svolgimento del progetto;
- il **piano di progetto**, dove vengono delineate le modalità di sviluppo del progetto software, di gestione delle risorse umane e temporali. È infine fornita una stima dei costi e delle risorse necessari alla realizzazione del progetto.

3.1.2.1.2 Documenti informali

Un documento viene definito *informale* quando:

- si trova in fase di stesura, validazione o deve essere ancora approvato dal *responsabile di progetto*;
- non è soggetto a *versionamento*$_G$.

I documenti informali sono da considerarsi **interni**, di utilità unicamente del team. Un documento di tale tipologia può essere:

- un **verbale esterno**, contenente il resoconto di un colloquio avvenuto tra *Onion Software* e i committenti e/o il proponente;
- un **verbale interno**, breve rapporto di riunioni interne al gruppo circa gli argomenti affrontati e le decisioni prese.

3.1.2.2 Progettazione e sviluppo

3.1.2.2.1 Denominazione documenti

- **Formali:**
  I documenti formali saranno denominati nel seguente modo:

  nomeDocumento_vX.Y.Z

  in cui:
  - **nomeDocumento** identifica univocamente il documento riferito. Non è ammesso l'uso di spazi e le uniche lettere maiuscole devono essere le iniziali di ogni parola, eccetto la prima;
  - **_vX.Y.Z** rappresenta la versione del documento dove _ è adibito alla separazione del nome dalla versione, **X** è il numero incrementale di approvazione da parte del *responsabile di progetto* , **Y** è l'incrementale della verifica eseguita da parte di un verificatore designato e **Z** è il numero di modifiche apportate al documento, che (ri)parte da zero dopo ogni verifica e approvazione.
- **Informali:**
  I documenti informali saranno denominati nel seguente modo:

  verbale_[I/E][nIncontro]_AAAA-MM-GG

  in cui:
  - **verbale** indica la natura informale del documento;
  - **I/E** indica una scelta, rispettivamente, tra verbale interno od esterno;
  - **nIncontro** fa riferimento al numero incrementale della riunione, oggetto del verbale;
  - **AAAA-MM-GG** indica la data in cui ha avuto luogo la riunione, espressa nel formato specificato nel §3.1.2.3.3.

3.1.2.2.2 Formato di un documento

Il formato dei documenti individuato è il *Portable Document Format*, estensione: *".pdf"*$_G$, sviluppato da *Adobe Systems*. Finché il documento non sarà approvato dal *responsabile di progetto* si troverà in formato *".tex"*$_G$.

3.1.2.2.3 Template

Per rendere standard la struttura, la grafica e l'impaginazione della documentazione, *Onion Software* ha creato un *template*$_G$ LaTeX, a disposizione ed uso di tutti. Organizzando le informazioni secondo un pattern comune, si ottengono i seguenti benefici:

- la documentazione risulta più comprensibile e di facile lettura;
- ogniqualvolta sia necessaria una modifica al template, essa verrà effettuata una sola volta per poi essere propagata immediatamente in tutti i documenti;
- i redattori dei documenti possono focalizzarsi sullo sviluppo dei contenuti, piuttosto di preoccuparsi della loro struttura.

Ogni modifica al template deve essere vagliata dall'*amministratore di progetto* il quale, in caso di approvazione della modifica, mette al corrente tutti i membri del team. Il template è raggiungibile nel repository privato all'indirizzo

https://github.com/Onion-Software/Butterfly/latexTemplate

3.1.2.2.4 Struttura dei documenti

- **Formali:**
  Ogni documento formale presenta un frontespizio, composto da:
  - l'emblema dell'Università degli Studi di Padova come ***watermark***$_G$;
  - il **logo** *Onion Software*;
  - l'**indirizzo email** con la quale la lista di distribuzione del documento può contattare il gruppo;
  - il **titolo del documento** e la **versione**, che lo identificano univocamente. I nomi possibili per un documento sono elencati nel §3.1.2.1.1;
  - la **data d'approvazione** del documento, eseguita da parte del *responsabile di progetto*;
  - i **responsabili del documento**, coloro che lo approvano;
  - i **redattori del documento**, coloro che scrivono i contenuti;
  - i **verificatori del documento**, coloro che verificano periodicamente la validità e la coerenza di quanto scritto;
  - lo **stato del documento**, che ne qualifica l'avanzamento;
  - la **lista di distribuzione del documento**, che specifica a chi è indirizzato;
  - lo **scopo del documento**, che specifica che utilità ha il documento.

  Alla pagina seguente, che presenta un layout vuoto[4], è presente un registro delle modifiche, rappresentato in forma tabulare, in cui sono presenti:
  - il **numero di versione**;
  - la **data** in cui sono state eseguite le operazioni di modifica o verifica;
  - l'**autore** che ha eseguito tali operazioni;
  - il **ruolo** che ha avuto l'autore delle operazioni;

[4] senza header né footer

– una breve **descrizione** di quanto operato.

A pagina tre è presente un indice dei contenuti, uno per ogni sezione, immagine e tabella: questo dà luogo a tre indici distinti.
Le pagine in cui sono presenti gli indici dei contenuti, delle immagini e delle tabelle possiedono una numerazione a sé stante, rappresentata con numeri romani.
Successivamente, in una nuova pagina, sono presenti i contenuti del documento. Nelle pagine che trattano i contenuti sono presenti un header ed un footer caratterizzati da una struttura comune:

– il **logo** *Onion Software* in alto a sinistra;
– il **numero** ed il **nome** della **sezione** corrente in alto a destra;
– una **prima riga** di colore nero, che delimita il confine tra l'header ed i contenuti del documento;
– una **seconda riga** di colore nero, che delimita il confine tra i contenuti del documento ed il footer;
– il **numero di versione** in basso a sinistra;
– il **numero della pagina** corrente, in relazione con il totale delle pagine, in basso al centro.

Ogni sezione introdotta con il comando `\section` è preceduta dal comando `\pagebreak`, che interrompe il riempimento della pagina dai contenuti, assicurando che ogni sezione parta da un nuovo foglio. Questo assicura maggiore leggibilità e migliore organizzazione delle informazioni.

- **Informali**
I documenti informali possiedono un frontespizio avente le seguenti peculiarità:

– il **logo** *Onion Software*;
– un **nome caratteristico**, identificato come in §3.1.2.2.1;
– il **luogo** dov'è avvenuta la riunione;
– l'**ora** d'inizio e fine della riunione;
– il **segretario** della riunione;
– i **partecipanti** alla riunione;
– l'**ordine del giorno** oggetto della riunione;

A pagina seguente è presente un breve **resoconto** di quanto discusso durante la riunione, le decisioni prese e, in caso di verbale esterno, le domande formulate con le rispettive risposte.

3.1.2.3 Norme tipografiche

Aggiungendo regole che ne normano la stesura, si assicura consistenza e coerenza alla documentazione.

3.1.2.3.1 Stile del testo

Il testo in un documento può essere in:

- **grassetto**, utilizzato per titoli e sottotitoli, oggetti salienti in elenchi puntati[5], intestazione delle tabelle ed altre parole ritenute degne di nota dall'*amministratore di progetto*;

[5] seguiti da descrizione

- *corsivo*, utilizzato per i nomi propri caratteristici[6], ruoli, termini presenti nel glossario, path, didascalie;
- MAIUSCOLO, utilizzato unicamente per identificare acronimi.

Anche il **colore del testo** costituisce uno speciale stile del testo: esso sarà nero per la quasi totalità dei documenti, mentre saranno violacei i riferimenti a siti internet esterni al documento.

3.1.2.3.2 Composizione del testo

I contenuti presenti nella documentazione dovranno rispettare le seguenti norme:

- ogni paragrafo del documento che non sia il titolo di qualche sezione o sottosezione deve iniziare con il comando `\par`;
- **elenchi puntati:**
  - **simboli di livello:** per introdurre elementi in una lista organizzata a elenco puntato, per il primo livello sarà usato un *punto elenco* "•", per il secondo livello un *tratto d'unione* "-", per il terzo livello un *asterisco* "*" mentre per il quarto livello un *puntino* ".";
  - **punteggiatura:** al termine di ogni elemento di un elenco puntato sarà presente un *punto e virgola* ";" ad eccezione dell'ultimo elemento, che sarà chiuso da un *punto* ".";
- **elenchi numerati:** rappresentati da numeri romani in maiuscolo, utili per elencare sequenze che presentano un ordine preciso;
- **esempi:** gli esempi potranno essere preceduti dall'espressione "*e.g.*", abbreviazione di *exempli gratia*, talvolta contenuti in parentesi doppie "( )";
- **riferimenti interni:** i riferimenti interni al documento devono riportare il numero della sezione, preceduto dal simbolo di paragrafo "§";
- **punteggiatura:** il simbolo di punteggiatura sarà preceduto da un carattere diverso da una spaziatura e seguito da uno spaziatura;
- **note a piè pagina:** ogni nota a piè pagina dovrà iniziare con un riferimento numerico in apice, seguito da una descrizione e/o contestualizzazione;
- **doppi apici:** i doppi apici verranno utilizzati per includere esempi, formati e simboli.

3.1.2.3.3 Formati dei dati

Per uniformare la grafia ed evitare errori di interpretazione al fine di rilasciare un documento coerente, *Onion Software* ha deciso di normare l'utilizzo dei seguenti formati:

- la **data**, espressa nella notazione anno-mese-giorno, con quattro cifre per rappresentare l'anno e due per il mese ed il giorno. Nel caso sia sufficiente una sola cifra per rappresentare un mese e/o un giorno, la cifra più significativa sarà zero "0"[7];
- l'**ora**, espressa nella notazione ore:minuti con due cifre per le ore ed altrettante per i minuti, come previsto dal *sistema orario a 24 ore*. Come per la data, in caso sia sufficiente una sola cifra per rappresentare l'ora e/o il minuto, la cifra più significativa sarà zero "0";

[6] e.g. *Onion Software* oppure *norme di progetto*
[7] **e.g.** 4 marzo 2019: 2019-03-04

- i **nomi propri** di persone, ruoli ed organizzazioni saranno riportati in *corsivo* con la prima lettera di ogni parola in maiuscolo, ad eccezione dei ruoli di progetto e dei titoli dei documenti che saranno scritti in minuscolo, a meno di essere preceduti da un *punto* ".". Per i nomi che presentano una certa ricorrenza, sono stati creati dei comandi LaTeX;
- i **comandi** ed i **path**[8] per indicare la posizione specifica di un file o cartella all'interno di un archivio dati saranno evidenziati con il comando LaTeX `\texttt`.
- il nome del software usato per redigere i documenti LaTeX sarà richiamato usando l'apposito comando `\LaTeX`;
- gli ***snippet*** $_G$ di codice saranno in *font* $_G$ `monospace`.

3.1.2.3.4 Elementi grafici

All'interno della documentazione, l'informazione può presentarsi anche in sotto forma di elemento grafico. Un elemento grafico si distingue in due tipologie:

- l'elemento **immagine**, in formato "*pdf*" oppure *"png"* $_G$ seguita da didascalia;
- l'elemento **tabella** ogniqualvolta l'*amministratore di progetto* decida che sia conveniente rappresentare le informazioni in forma tabellare. La tabella presenta una scala di colorazioni del viola, in riferimento al colore del logo di *Onion Software*, in modo da renderla più leggibile. Talvolta, la tabella può essere seguita da una breve didascalia.

In entrambi i casi, l'elemento grafico sarà numerato e presente in un indice a parte, denominato "*indice delle immagini*" se si tratta di immagine, "*indice delle tabelle*" se si tratta di tabella.

3.1.2.4 Produzione e mantenimento

I documenti devono essere prodotti e forniti in modo conforme a quanto descritto di seguito.

3.1.2.4.1 Ciclo di vita di un documento

Ogni documento è soggetto al seguente ciclo di vita:

- **creazione:** il documento viene creato partendo da un template predefinito e reperibile dalla directory `latexTemplate` della repository;
- **strutturazione:** il documento viene provvisto di un registro delle modifiche dalla struttura predefinita, un indice dei contenuti testuali, un indice delle tabelle[9] ed un indice delle immagini[10]
- **stesura:** il documento viene redatto progressivamente, seguendo uno stile modulare ed incrementale;
- **in revisione:** ogni parte del documento è soggetta a verifiche, per assicurare validità e coerenza di quanto riportato. La revisione di una sezione è operata da una o più persone. Nel primo caso la persona che revisiona la sezione deve essere necessariamente diversa da chi l'ha scritta;
- **approvato:** terminata la fase di revisione delle sezioni, il *responsabile di progetto* vaglia il documento e determina la sua validità. Il documento, quindi, è pronto per il rilascio.

[8] siano essi locali o remoti
[9] ad eccezione del registro delle modifiche
[10] ad eccezione del logo *Onion Software* presente nel frontespizio

3.1.2.4.2 Organizzazione directory documento

Ogni documento scritto da *Onion Software* è locato in una directory dedicata che possiede le seguenti proprietà:

- il **nome** della directory deve essere privo di spazi ed in formato *camelCase*. Ciò significa che la prima lettera è minuscola mentre le iniziali di ogni parola vanno scritte in maiuscolo (e.g. `nomeDellaCartellaCheContieneIlDocumento`). All'interno vi sono 5 elementi:
  - la sottodirectory ***imgs***, che organizza le immagini visualizzate all'interno del documento: saranno sempre contenute, in questa cartella, le immagini *header.png, logo.png* e *unipd.png*;
  - il file *listOfSection.tex*, *script*$_G$ che include tutte le sezioni che interessano il documento;
  - il file *main.tex*, che inizializza il documento inserendo le informazioni che verranno visualizzate nel frontespizio in modo da lasciare inalterati i file di configurazione;
  - la sottodirectory ***res***, che organizza i contenuti e le configurazioni del documento nel seguente modo:
    - la sottodirectory ***config*** contiene quattro file:
      - *docuInfo.tex* che inizializza il documento inserendo informazioni specifiche del documento, diverse da tutti gli altri documenti;
      - *package.tex* in cui sono contenuti tutti i pacchetti da caricare per assicurare una corretta visualizzazione del testo;
      - *config.tex* in cui sono specificate le impostazioni secondo uno stile predefinito e scelto da *Onion Software*;
      - *commands.tex* che contiene tutti i comandi creati e ridefiniti da *Onion Software*.
    - la sottodirectory ***sections*** che contiene un file *.tex* per ogni `\section` che compone il documento.

3.1.2.4.3 LaTeX

Onion Software utilizza *Visual Studio Code* con l'estensione *Latex Workshop* e *TeXMaker* con *compilatore*$_G$ *MikTex 2.9* come editor di testo per LaTeX, linguaggio di markup utilizzato per redigere la documentazione. La directory contenente ogni documento presenta una struttura comune, come descritto in §3.1.2.4.2.

3.1.2.4.4 Modalità di distribuzione dei documenti

I documenti, in accordo con gli interessati, verranno inviati via email oppure potranno essere resi disponibili nelle directory dedicate di *GitHub*. *Onion Software* è attenta agli sprechi ed è disposta a fornire il cartaceo della documentazione solo se strettamente necessario e/o richiesto dalla lista di distribuzione del documento.

3.1.2.4.5 Archiviazione dei documenti

I materiali devono essere archiviati in conformità ai requisiti per la conservazione dei dati, la sicurezza, la manutenzione ed il backup. Tale motivazione ha spinto *Onion Software* ad operare un *backup*$_G$ ogni giorno alle 23.59 di tutta la repository comprensiva di ogni suo *branch*$_G$. Tale copia di backup viene custodita fisicamente in una memoria esterna di proprietà del *responsabile di progetto*.

3.1.2.4.6 Controllo termini di glossario

Una volta ultimate le verifiche, il verificatore utilizzerà un particolare script per accertare che le parole marcate con $_G$ nella documentazione siano state effettivamente inserite nel glossario. Per eseguire tale procedura, il verificatore dovrà:

- da riga di comando, spostarsi nella directory `script`;
- inserire nel file `conf.txt` il nome della directory dove sono conservati i file da controllare, seguito da "*.tex*". Ad esempio, se si vuole controllare che siano state inserite nel *Glossario* tutte le voci contrassegnate con $_G$ all'interno del documento *norme di progetto*, basterà inserire `[path]/normeDiProgetto/res/sections/*.tex` con, al posto di `[path]`, il percorso relativo per raggiungere la directory "normeDiProgetto" partendo dalla locazione del file *parser.py*;
- eseguire il comando `python3 parser.py`;
- verrà notificato un risultato che elencherà le eventuali mancanze, così da facilitarne l'inserimento.

3.2 Configurazione

3.2.1 Scopo

La documentazione e le parti soggette a versionamento del progetto verranno gestite sulla piattaforma *GitHub* utilizzando la tecnologia *Git*$_G$. I verbali interni ed esterni e le parti del progetto non versionate verranno gestite sulla piattaforma *Google Drive*$_G$.

3.2.2 Struttura

È presente una repository contenente tutti i documenti redatti da *Onion Software* così strutturata:

- **esterni**, contenente i documenti esterni:
  - analisiDeiRequisiti;
  - glossario;
  - pianoDiProgetto;
  - pianoDiQualifica;
  - verbali.
- **interni**, contenente i documenti interni:
  - verbali;
  - normeDiProgetto;
  - studioDiFattibilita.

Ogni documento sopracitato è organizzato in directory come spiegato nel §3.1.2.4.3.

3.2.3 Ciclo di vita di un branch

I branch principali vengono creati dall'*amministratore di progetto* attraverso l'estensione *GitFlow*$_G$. Dopo la creazione di una *issue*$_G$, la persona incaricata è responsabile nel crearsi il suo branch di lavoro con il nome riportato nella descrizione della issue. Al completamento del lavoro il verificatore si creerà il branch di *feature* dedicato alla verifica del lavoro dei redattori. A verifiche ultimate, il responsabile entrerà nel branch del verificatore e, in caso di approvazione, effettuerà il merge con il ramo *develop*. I redattori sono responsabili nel chiudere i loro branch una volta che i documenti saranno approvati. Il branch master verrà aggiornato attraverso le release in vista delle revisioni programmate.

3.2.4 Aggiornamento della directory

Il redattore ed il verificatore devono crearsi il proprio branch attraverso il comando `git flow feature start 'nome_branch'`.
Successivamente si eseguiranno i seguenti comandi da terminale:

- `git branch` per verificare di trovarsi nel branch di lavoro/verifica corretto[11];
- `git pull origin develop` per aggiornare il proprio ramo con quello di *develop*;
- `git add nomeFile` per aggiungere "nomeFile" al *commit*$_G$ (per aggiungerli tutti, digitare `git add .`);
- `git commit -m ''messaggio''` inserendo un messaggio utile agli altri membri del team per comprendere le modifiche intraprese;
- `git push` per inviare le modifiche sul repository remoto *GitHub* rendendo visibili le proprie modifiche anche agli altri membri del team.

3.2.4.1 Bad Commit

Può verificarsi l'eventualità che la repository venga aggiornata da un commit contenente errori di diversa natura[12], operato da parte di un *amministratore di progetto*. Ciò invalida immediatamente l'intero branch dov'è stato operato il commit, rendendo necessaria un'operazione di ripristino dell'ultimo commit valido. Tale procedura ha diversi modi per essere attuata. Se nessuna delle due soluzioni proposte di seguito è soddisfacente, si prega di contattare il *responsabile di progetto*.

3.2.4.1.1 Annullare un commit

Il comando da terminale per annullare completamente un commit, avendo a disposizione il suo codice identificativo, è il seguente:

```
git revert idCommit
```

dove `idCommit` è il codice identificativo (visibile sulla piattaforma GitHub) del bad commit.

3.2.4.1.2 Annullare l'ultimo commit

Per eliminare l'ultimo commit, basta inserire il comando da terminale

```
git push OnionSoftware +idCommit^:nomeBranch
```

dove `idCommit` è il codice identificativo dell'ultimo commit (visibile sulla piattaforma GitHub) e `nomeBranch` è il nome del branch dov'è stato operato tale commit.

3.3 Qualità

3.3.1 Scopo

Poiché fornire un prodotto software di qualità sia al proponente che cliente è l'obiettivo finale del progetto, la presenza di questa sezione è essenziale. L'obiettivo è quello di garantire che il prodotto e i servizi rispettino uno standard qualitativo accettabile per il proponente. Ci aspettiamo di garantire qualità nell'organizzazione, nel prodotto e in tutti processi e sotto-processi derivanti grazie all'uso di verifiche e controlli opportunamente selezionati.

[11] il branch corrente è contrassegnato da un asterisco "*" a sinistra del nome del ramo
[12] d'ora in poi denominato: *bad commit*

3.3.2 Classificazione metriche

Dopo un'opportuna riflessione siamo giunti a tale notazione soddisfacente:

`M[settore di utilizzo][numero]`

dove:

- *settore di utilizzo*: si riferisce all'ambito in cui viene applicata la metrica, che si distingue in processi, prodotti oppure test. I vari acronimi di riferimenti a metriche si suddividono in:
  - PR: per processi;
  - PD: per prodotti;
  - T: per test;
  - a loro volta possono essere seguiti da una S o una D:
    - S: sta per software, quindi la metrica si applica in ambito software;
    - D: sta per documento, quindi la metrica si applica nella documentazione;
    - se non è presente nel la S ne la D significa che la metrica si riferisce ad ambiti più generici.
- *numero*: si riferisce al codice d'identificazione della metrica inteso come numero intero a due cifre.

3.3.3 Organizzazione del processo

Per la realizzazione di un processo il gruppo ha deciso di ricercare la qualità fin dal prima fase di sviluppo adottando il metodo $PDCA_{\boldsymbol{G}}$, descritto nell'appendice A. Grazie a questo metodo è possibile ottenere un miglioramento continuo della qualità di tutti i processi, verifiche e prodotti.
Le seguenti norme danno un'idea del tipo di ragionamento attuato per ogni processo stanziato:

- **pianificazione**: stipulazione di attività, scadenze, responsabilità, risorse utili a raggiungere specifici obbiettivi di miglioramento;
- **esecuzione**: rispettare rigorosamente la strategia pianificata;
- **valutazione**: verificare l'esito delle azioni di miglioramento rispetto alle attese;
- **consolidazione**: consolidare il buono e cercare modi per migliorare il resto.

3.3.4 Procedure per la qualità del processo

In questa sezione si analizzano delle procedure per garantire la qualità dei vari processi; la classificazione alfanumerica fa riferimento al paragrafo §3.3.2.

3.3.4.1 MPR01 Schedule Variance

Questa metrica è un indicatore del fatto che un'attività in fase di sviluppo è in anticipo o in ritardo secondo la schedulazione fissata durante la pianificazione di quest'ultima.
Si calcola con la seguente formula:

$$Schedule\ Variance = EV - PV$$

dove EV sta per lavoro effettivamente completato mentre PV indica il valore del lavoro previsto nella data presa in considerazione.
Dalla formula si può concludere che:

- si è in anticipo rispetto alla pianificazione se il valore ottenuto è positivo;

- si è in ritardo rispetto alla pianificazione se il valore è negativo;
- si è in perfetta tabella di marcia se il valore è zero.

Se si ricade nei primi due casi, le pianificazioni future dovranno ridurre o aumentare le previsioni di durata.
I valori possono essere misurati in minuti o in Euro;

3.3.4.2 MPR02 Budget Variance

Permette di rendersi conto dei costi sostenuti alla data corrente rispetto al budget preventivato. Si calcola con la seguente formula:

$$Budget\ Variance = \log_{10}(PCWS - ACWP)^2$$

dove $PCWS$(Planed Cost of Work Scheduled) indica il costo previsto per la realizzazione delle attività completate in data odierna mentre $ACWP$(Actual Cost of Work Performed) sta per costo sostenuto per la realizzazione delle attività fino alla data corrente.
Se il risultato ottenuto è positivo significa che il budget sta venendo speso entro i canoni previsti, mentre se è negativo il budget sta venendo speso velocemente e di conseguenza si rischia di sforare i costi pattuiti con il proponente.

3.3.4.3 MPRS01 Code Coverage

Indica il numero di righe di codice percorse dai test durante la loro esecuzione. Per linee di codice totali si intende tutte quelle appartenenti all'unità in fase di test. Per poter garantire un'ottima copertura di verifica al code coverage si utilizzano:

- MPRS02 Function coverage: verificare che ogni funzione sia stata chiamata;
- MPRS03 Statement coverage: verificare che ogni statement del codice sia stato eseguito;
- MPRS04 Branch coverage: verificare se tutti i possibili branch(derivanti da if e case statement) sono stati eseguiti;
- MPRS05 Condition coverage: ogni espressione booleana è stata valutata sia a vero che a falso;
- MPRS06 Lines coverage: verificare se ogni linea è stata eseguita e/o percorsa. È l'applicativo del code coverage.

$$LC = \frac{linee\_di\_codice\_percorse}{linee\_di\_codice\_totali}$$

3.3.5 Procedure per la qualità dei prodotti

In questa sezione si analizzano delle procedure per garantire la qualità dei vari prodotti; la classificazione alfanumerica fa riferimento al paragrafo §3.3.2.

3.3.5.1 MPDD01 Indice Gulpease

È un indice di leggibilità per testo in lingua italiana. Rispetto ad altri ha il vantaggio di utilizzare la lunghezza delle parole in lettere anziché in sillabe, semplificandone il calcolo automatico.
La formula per il suo calcolo è la seguente:

$$Gulp = 89 + \frac{300 * (totale\_frasi) - 10 * (totale\_lettere)}{totale\_parole}$$

I risultati sono compresi tra 0 e 100, dove il valore 100 indica la leggibilità più alta e 0 la leggibilità più bassa. In generale risulta che testi con un indice:

- inferiore a 80 sono difficili da leggere per chi ha la licenza elementare;
- inferiore a 60 sono difficili da leggere per chi ha la licenza media;
- inferiore a 40 sono difficili da leggere per chi ha un diploma superiore.

3.3.5.2 MPDD02 Errori ortografici

Il controllo ortografico si attua come descritto nel paragrafo §3.4.5.1 presente nella sezione strumenti di questo documento.

3.3.5.3 MPDS01 Copertura requisiti obbligatori

Indica la percentuale dei requisiti obbligatori coperti dall'implementazione. La sua formula di misurazione è la seguente:

$$CRO = \left(\frac{N_{\text{os}}}{N_{\text{o}}}\right) * 100$$

dove N_{os} è il numero di requisiti obbligatori soddisfatto mentre N_o è il numero di requisiti obbligatori.

3.3.5.4 MPDS02 Copertura requisiti accettati

Indica la percentuale dei requisiti desiderabili e facoltativi coperti dall'implementazione. La sua formula di misurazione è la seguente:

$$CRA = \left(\frac{N_{\text{as}}}{N_{\text{a}}}\right) * 100$$

dove N_{as} è il numero di requisiti accettati soddisfatti mentre N_a è il numero di requisiti accettati.

3.3.5.5 MPDS03 Accuratezza rispetto alle attese

Indica la percentuale di risultati concordi alle attese. La sua formula di misurazione è la seguente:

$$ARA = \left(1 - \frac{N_{\text{td}}}{N_{\text{te}}}\right) * 100$$

dove N_{td} è il numero di test che producono risultati discordi alle attese e N_{te} è il numero di test eseguiti.

3.3.5.6 MPDS04 Tempo medio di risposta

Indica il tempo medio che intercorre fra la richiesta software di una determinata funzionalità e la restituzione del risultato all'utente. La sua formula di misurazione è la seguente:

$$TR = \frac{\sum_{i=1}^{n} T_i}{n}$$

dove T è il tempo intercorso fra la richiesta i di una funzionalità ed il comportamento delle operazioni necessarie a restituire un risultato a tale richiesta.

3.3.5.7 MPDS05 Adeguatezza del tempo di risposta

Rapporto tra il tempo medio di risposta e il tempo previsto dai requisiti.

3.3.5.8 MPDS06 Densità errori

Indica la percentuale dei test che si sono concluse con un fallimento. La sua formula di misurazione è la seguente:

$$DE = \left(\frac{N_{\text{er}}}{N_{\text{te}}}\right) * 100$$

dove N_{er} è il numero di errori rilevati durante l'attività di testing mentre N_{te} è il numero di test eseguiti.

3.3.5.9 MPDS07 Capacità di analisi degli errori

Indica la percentuale di modifiche effettuate in risposta agli errori che hanno portato all'introduzione di nuovi errori in altre componenti del sistema. La sua formula di misurazione è la seguente:

$$CAE = \left(\frac{N_{\text{ei}}}{N_{\text{er}}}\right) * 100$$

dove N_{ei} è il numero di errori delle quali sono state individuate le cause mentre N_{er} è il numero di errori rilevati.

3.3.5.10 MPDS08 Efficienza delle modifiche

Indica la percentuale di modifiche effettuate in risposta agli errori che hanno portato all'introduzione di nuove errori in altre componenti del sistema. La sua formula di misurazione è la seguente: la seguente:

$$EM = \left(1 - \frac{N_{\text{ere}}}{N_{\text{er}}}\right) * 100$$

dove N_{ere} è il numero di errori risolti con l'introduzione di nuovi errori mentre N_{er} è il numero di errori risolti.

3.3.5.11 MPDS09 Accoppiamento di componenti

In un prodotto/sistema software quanto strettamente sono indipendenti i componenti e quanti componenti sono esenti da impatti da cambiamenti negli altri componenti. Si calcola con un valore intero che rappresenta la totalità di componenti con metodi o variabili definiti da altri componenti. Più basso è il valore più alta sarà l'indipendenza dei componenti.

3.3.5.12 MPDS10 Interfaccia utente auto-esplicativa

Percentuale degli elementi d'informazione e dei passi che sono presentati all'utente inesperto in modo che questi possa completare un'attività senza un addestramento preliminare o assistenza esterna. La sua formula di misurazione è la seguente:

$$CF = \left(\frac{N_{\text{fc}}}{N_{\text{fo}}}\right) * 100$$

dove N_{fc} è il numero di funzionalità comprese in modo immediato dall'utente durante l'attività di testing del prodotto e N_{fo} è il numero di funzionalità offerte dal sistema.

3.3.5.13 MPDS11 Comprensibilità dei messaggi d'errore

Percentuale dei messaggi d'errore che dichiarano la ragione dell'errore e suggeriscono come risolverlo. La sua formula di misurazione è la seguente:

$$CF = \left(\frac{N_{\text{ec}}}{N_{\text{e}}}\right) * 100$$

dove N_{ec} è il numero di errori comprese in modo immediato dall'utente durante l'attività di testing del prodotto e N_e è il numero di errori emessi dal sistema durante l'attività di testing.

3.3.5.14 MPDS12 Duplicazione del codice

Percentuale del codice duplicato rilevato dall'applicazione Sonarqube.
Una percentuale tollerabile di linee di codice duplicato non deve eccedere del 20% rispetto al numero totale.

3.4 Verifica

3.4.1 Scopo

Il processo di verifica ha il fine di garantire la correttezza e completezza del prodotto finale. Sono soggetti a verifica il software e la documentazione.

3.4.2 Aspettative

Un corretto utilizzo del processo di verifica si svolge con:

- l'estrapolazione di una procedura;
- l'uso di criteri adeguati;
- l'esecuzione dopo ogni fase di produzione;
- la catalogazione dei difetti, se presenti, che dovranno essere segnalati e corretti.

3.4.3 Descrizione

Il processo di verifica è composto da due diverse attività:

- analisi, ossia un controllo del codice sorgente con successiva esecuzione. Si compone di analisi statica e analisi dinamica;
- test, vari test eseguiti sul software prodotto.

3.4.4 Attività

3.4.4.1 Analisi

3.4.4.1.1 Statica

È una tecnica che permette l'individuazione di errori all'interno della documentazione e codice sorgente. Per eseguire l'analisi manuale, diversa dalla formale, si può scegliere tra due diversi metodi:

- ***walkthrough***: consiste in un'analisi a tappeto di tutta la parte da analizzare. È un'attività onerosa in quanto richiede la cooperazione di più membri del team, tuttavia è molto semplice e di conseguenza sarà utilizzata almeno nelle prime fasi del progetto;
- ***inspection***: consiste in una ricerca mirata e strutturata degli errori più comuni segnalati nella lista di controllo. La lista viene progressivamente estesa in risposta alla scoperta di errori, ciò rende l'inspection sempre più efficace. Tale tecnica è solitamente svolta da un singolo elemento del team. Di seguito è riportata la tabella con gli errori più comuni.

	Errori più comuni
1	utilizzo inappropriato di lettere maiuscole
2	non utilizzo del simbolo “;” all’interno di elenchi puntati
3	utilizzo involontario di doppi spazi tra due parole
4	date scritte senza seguire le regole di stesura

Tabella 5: *Inspection* degli errori più comuni

3.4.4.1.2 Dinamica

È una tecnica che necessita del prodotto software e della sua esecuzione in quanto permette l’identificazione di anomalie e verifica il corretto funzionamento del prodotto, previo utilizzo di opportuni test.

3.4.4.2 Test

Per produrre un buon prodotto è essenziale testarlo, e per testare un prodotto è fondamentale creare buoni test. Per produrre test di ottima fattura si devono rispettare diversi parametri:

- **ambiente**: il sistema hardware e software sul quale viene eseguito il test;
- **stato iniziale**: stato dal quale viene eseguito il test;
- **input**: dati inseriti;
- **output**: risultato ottenuto;
- **istruzioni aggiuntive**: informazioni sul test in questione, ossia istruzioni di utilizzo, interpretazione del risultato, eccetera.

Affinché i test si possano definire “buoni”, devono:

- essere ripetibili;
- essere provvisti di ambiente di esecuzione;
- avere input/output identificati;
- avvertire riguardo possibili eccezioni;
- fornire un file log con i risultati.

Il gruppo *Onion Software* adotta come modello di sviluppo del software il *modello a* V_G, il quale prevede la creazione dei test in concomitanza alle attività di analisi e progettazione. In questo modo i test permetteranno di verificare la correttezza delle parti di programma parallelamente al loro sviluppo. Per definire lo stato in cui ogni test si trova, vengono utilizzate le seguenti sigle:

- **IP**: indica che il test è implementato;
- **NIP**: indica che il test non è implementato;
- **V**: indica che il test è stato verificato e quindi soddisfa la richiesta;
- **NV**: indica che il test non è stato verificato.

Per facilitare la classificazione dei test abbiamo adottato la seguente notazione:

`testType[requisito][tipo][id]`

dove:

- `testType` si riferisce al tipo di test:
  - **TU**: test di unità;
  - **TI**: test di integrazione;
  - **TS**: test di sistema;
  - **TA**: test di accettazione.
- `requisito` si riferisce al tipo di requisito, come descritto in §2.2.4.1.2:
  - **O**: requisiti obbligatori;
  - **D**: requisiti desiderabili;
  - **F**: requisiti facoltativi.
- `tipo` si riferisce all'ambiente di applicazione del test:
  - **F**: test di funzionalità;
  - **Q**: test di qualità;
  - **P**: test di prestazioni;
  - **V**: test di vincolo.
- `id` si riferisce all'identificativo numerico del test.

Ognuno dei test ha un diverso scopo e oggetto di verifica.

3.4.4.2.1 Test di unità

Viene prettamente eseguito su singole unità software, ossia su componenti di programma con funzionamento autonomo; solitamente questo test si esegue prima dell'integrazione di un'unità ad un'altra unità o sistema. Il test in questione richiede la preparazione di elementi software, *driver*$_G$ e *stub*$_G$: il driver simula l'unità chiamante, mentre lo stub l'unità chiamata.

3.4.4.2.2 Test di integrazione

Si potrebbe definire l'estensione logica del test di unità in quanto si esegue quando si integrano più unità in un unico componente. Il test lavora anche sulle interfacce coinvolte, quindi non solo verifica il corretto funzionamento di ogni singolo elemento, ma anche le relazioni tra le unità integrate.
Iniziando da questo sistema composto da singole unità, si giungerà progressivamente a testare moduli di un gruppo con quelli di altri gruppi; la procedura si ripete fino al raggiungimento della totalità del sistema. Utilizzare una strategia di test di integrazione sistematica e incrementale è da preferire alla strategia di *Big Bang testing*$_G$, dove tutti i componenti sono integrati nello stesso momento;

3.4.4.2.3 Test di sistema

Lo scopo di questo test è valutare la conformità del sistema ai requisiti specificati; come quelli di sicurezza, velocità, accuratezza e affidabilità. Il superamento dei test di sistema sancisce il raggiungimento del prodotto finale.

3.4.4.2.4 Test di accettazione

In questo livello di test il software è stato testato per l'accettabilità. Lo scopo di questo test è valutare la conformità del sistema con i requisiti aziendali e valutare se è accettabile per la consegna.

3.4.4.2.5 Metriche di test

Queste misurazioni servono per tenere traccia delle esecuzioni dei test e relativi successi fallimenti tramite le seguenti metriche:

- TS01 percentuale di test passati. Indica la percentuale di test passati, molto utile per capire a che punto si è nella fase di sviluppo della componente. La sua formula di misurazione è la seguente:

$$TP = \left(\frac{TP}{TE}\right) * 100$$

  dove TP indica il numero di test passati e TE il numero di test eseguiti;

- TS02 percentuale di test falliti. Indica la percentuale di test falliti, molto utile per capire a che punto si è nella fase di sviluppo della componente. La sua formula di misurazione è la seguente:

$$TF = \left(\frac{TF}{TE}\right) * 100$$

  dove TF indica il numero di test falliti e TE il numero di test eseguiti;

- TS03 totalità dei test eseguiti in rapporto ai requisiti: indica la percentuale di test eseguiti sui requisiti totali, utile per capire l'avanzamento dei test. La sua formula di misurazione è la seguente:

$$TF = \left(\frac{TE}{TT}\right) * 100$$

  dove TE indica il numero di test eseguiti e TT il numero di test totali;

3.4.5 Strumenti

3.4.5.1 Controllo ortografico

Per il controllo ortografico viene utilizzata la correzione in tempo reale integrata nell'editor *Texmaker*, che permette la visualizzazione immediata degli errori grazie alla sottolineatura delle parole non corrette secondo la lingua italiana; altri strumenti verranno integrati nel paragrafo durante l'avanzamento del progetto.

3.4.5.2 Indice di Gulpease

Per ottenere l'indice di Gulpease abbiamo utilizzato lo strumento web disponibile all'indirizzo `https://farfalla-project.org/readability_static/`.

3.5 Validazione

3.5.1 Scopo

La validazione è un processo per determinare se il prodotto finale soddisfa i requisiti del progetto commissionato. È buona norma eseguire la validazione di un progetto quando si è giunti a un prodotto definitivo testato e verificato.

3.5.2 Aspettative

Un corretto sviluppo del processo di validazione si raggiunge con:

- identificazione dei prodotti da validare;
- creazione di una procedura adeguata di validazione;
- utilizzo di criteri adeguati per la validazione;
- comparazione dei risultati ottenuti con quelli auspicati;

3.5.3 Attività

La validazione si ottiene con il superamento dell'analisi dinamica, che permette di giungere al prodotto definitivo solo dopo l'esecuzione opportuni test e verifiche. La validazione si dice compiuta quando i risultati di test e verifiche soddisfano i parametri standard forniti dal *piano di qualifica*. I *verificatori di progetto* hanno il compito di rieseguire tutti i test, ponendo attenzione ai risultati, in particolare per i test di validazione. Il *responsabile di progetto* dovrà analizzare i risultati ottenuti e decidere se rieseguire i test.

4 Processi organizzativi

4.1 Gestione

4.1.1 Scopo

Il processo di gestione si occupa di pianificare e controllare l'esecuzione delle varie attività dei processi. In particolare, lo scopo è:

- istanziare processi nel progetto;
- stimare i costi e le risorse necessarie;
- pianificare le attività e assegnarle alle persone;
- controllare le attività e verificare i risultati.

4.1.2 Istanziazione del processo

L'istanziazione dei processi nel progetto avviene in due passi:

- i processi aziendali sono istanziati da standard di processo (quale ISO/IEC 12207:1995);
- i processi di progetto sono istanziati da processi aziendali.

Uno standard di processo definisce un insieme di processi standard trasversali a diverse tipologie di *way of working*$_G$; nell'insieme i processi identificano il way of working adottato. L'istanziazione dei processi di progetto può dipende da diversi fattori, i quali:

- dimensione e complessità del progetto;
- rischi identificati;
- disponibilità di risorse;
- vincoli contrattuali.

4.1.3 Pianificazione

Il responsabile deve pianificare l'esecuzione del processo di gestione compilando un piano di progetto, documento ad uso esterno che include:

- la pianificazione delle attività nel tempo, a partire dall'obiettivo;
- la stima dei costi e delle risorse necessarie per completare le attività dei processi di progetto;
- la quantificazione dei rischi associati allo svolgimento dei processi;
- l'assegnazione delle diverse responsabilità ai componenti del gruppo;
- i controlli di qualità e di verifica delle attività dei processi.

Ogni attività ha specifici obiettivi e vincoli. Come obiettivo comune si vuole perseguire l'economicità, nonché l'efficacia e l'efficienza.

4.1.4 Ruoli

Nel corso del progetto ogni membro del gruppo deve ricoprire un ruolo, a rotazione, che gli viene assegnato. Nel piano di progetto sono organizzate e pianificate le attività assegnate agli specifici ruoli, descritti di seguito.

4.1.4.1 Responsabile di progetto

Il responsabile di progetto è la figura professionale che possiede maggiori responsabilità, in particolare quelle riguardanti la scelta e l'approvazione di documenti, la gestione, il controllo ed il coordinamento del gruppo. Inoltre, egli rappresenta il progetto presso il fornitore e il *committente*$_G$, facendo quindi da intermediario. In particolare, si occupa di:

- controllare, coordinare e gestire le attività;
- gestire le relazioni esterne;
- gestire le risorse umane;
- approvare i documenti.

4.1.4.2 Amministratore di progetto

L'amministratore è la figura professionale che gestisce e controlla l'ambiente di lavoro. In particolare, si occupa di:

- amministrare le infrastrutture di supporto;
- risolvere i problemi legati alla gestione dei processi;
- gestire la documentazione di progetto;
- controllare le versioni e le configurazioni

4.1.4.3 Analista

L'analista è una figura professionale che possiede maggiori conoscenze per quanto riguarda il dominio del problema. In particolare, si occupa di:

- studiare approfonditamente il problema, nonché il suo contesto applicativo;
- capire il problema;
- definire la complessità e le diverse tipologie di requisiti, sia espliciti, sia impliciti;
- redarre il documento *analisi dei requisiti*.

4.1.4.4 Progettista

Il progettista è una figura professionale che possiede competenze tecniche e tecnologiche costantemente aggiornate; si occupa di:

- effettuare scelte legate agli aspetti tecnici e tecnologici del progetto;
- sviluppare un'architettura che mira all'economicità ed alla manutenibilità a partire dal lavoro svolto dall'analista;
- redarre il documento *studio di fattibilità*.

4.1.4.5 Programmatore

Il programmatore è una figura professionale responsabile della codifica di parte dell'architettura prodotta dal progettista e delle componenti di supporto che serviranno per effettuare le prove di verifica e validazione sul prodotto. Il programmatore si occupa di:

- implementare le scelte progettuali del progettista;
- creare o gestire componenti di supporto per la verifica e validazione del codice;
- redarre il *manuale utente* relativo alla propria codifica.

4.1.4.6 Verificatore

Il verificatore è una figura professionale presente per l'intera durata del progetto che possiede competenze tecniche e conoscenza delle norme, affidandosi agli standard definiti nelle *nome di progetto*, nonché alla propria esperienza e capacità di giudizio, per effettuare correzioni e controlli del lavoro svolto dagli altri componenti del gruppo. Si occupa di:

- verificare la conformità dei prodotti ai requisiti sia funzionali sia di qualità;
- controllare e correggere i prodotti in fase di revisione;
- segnalare ed evidenziare gli errori trovati nei vari prodotti;

4.1.5 Procedure

4.1.5.1 Gestione delle comunicazioni

4.1.5.1.1 Comunicazioni interne

Le comunicazioni interne riguardano solamente i membri del gruppo e avvengono utilizzando Telegram. Questa è un'applicazione di messaggistica istantanea multipiattaforma che permette di creare gruppi che possono essere anche separati per ruolo all'interno del team. Possiede inoltre diverse funzionalità, tra cui quelle di videochiamata e messaggi vocali, nonché di bot per la creazione di sondaggi. La facilità d'uso e le vaste funzionalità di Telegram ai fini progettuali ne hanno indotto la scelta.

4.1.5.1.2 Comunicazioni esterne

Il responsabile si occupa delle comunicazioni con soggetti esterni al gruppo. Si utilizza come strumento l'indirizzo di posta elettronica del gruppo **softwareonion@gmail.com**.
Per le comunicazioni con il committente *Imola Informatica S.p.A.*, come descritto nel capitolato, a causa della distanza si utilizzeranno principalmente i seguenti strumenti:

- Telegram per la chat;
- Google Hangouts o Skype per le videochiamate.

In casi eccezionali sarà comunque possibile richiedere incontri di persona e/o definire ulteriori strumenti di comunicazione.

4.1.5.2 Gestione degli incontri

4.1.5.2.1 Incontri interni

Gli incontri interni riguardano tutti i componenti all'interno del gruppo. Sono richiesti ed organizzati dal responsabile che dovrà stabilire data e ora mediante l'utilizzo di *Google Calendar*$_{G}$, applicativo che consente ad ogni membro del gruppo di segnalare gli orari in cui non è disponibile. Altri applicativi utilizzati verranno specificati in §4.1.5.4.
Se necessario, qualsiasi altro membro può avanzare una richiesta di riunione interna, che sarà eventualmente approvata dal responsabile.

4.1.5.2.2 Verbali di incontri interni

Ad ogni incontro interno un componente del gruppo nominato dal responsabile avrà il compito di redarre un *verbale*. Questo documento dovrà contenere almeno le informazioni specificate nel paragrafo §3.1.2.2.4.

4.1.5.2.3 Incontri esterni

Il responsabile è la sola figura professionale ad avere la possibilità di organizzare incontri con il proponente o il committente. Il responsabile deve essere presente agli incontri esterni e può, eventualmente, essere accompagnato da al più altri due componenti del gruppo. Sarà sempre compito del responsabile informare gli altri membri del gruppo, attraverso comunicazioni o incontri interni.

4.1.5.2.4 Verbali di incontri esterni

Come per le riunioni interne, anche per le esterne viene redatto un *verbale*. La struttura delle due tipologie è analoga, ma le riunioni esterne presentano una maggiore criticità per la presenza di persone esterne al gruppo, quali il committente e/o il proponente.

4.1.5.2.5 Tracciamento delle decisioni

Per ogni verbale redatto viene inclusa una tabella che riporta il tracciamento delle decisioni prese durante l'incontro. Ogni decisione sarà identificata nel modo seguente:

`VER-AAAA-MM-GG.id`

dove:

- **VER**: sigla della parola *verbale*;
- **AAAA-MM-GG** indica la data in cui ha avuto luogo la riunione, espressa nel formato specificato nel §3.1.2.3.3;
- **id**: indica il numero identificativo della decisione.

4.1.5.3 Gestione degli strumenti di coordinamento

4.1.5.3.1 Ticketing

Il responsabile può assegnare compiti ai componenti del gruppo e controllare l'andamento di questi attraverso il ticketing; ciò consiste nella creazione di ticket che devono avere almeno le seguenti informazioni:

- titolo;
- descrizione;
- assegnatario;
- data di scadenza;
- stato del ticket;
- eventuali tag per facilitare l'identificazione;
- eventuali commenti per segnalare particolarità riscontrate e/o utili per lo svolgimento del compito.

Per creare ticket si utilizza Github. L'assegnatario dovrà aggiornare periodicamente lo stato del ticket, che dovrà sempre essere uno dei seguenti:

- da fare (TODO);
- in lavorazione (DOING);
- in revisione;
- completato (DONE).

4.1.5.4 Gestione dei rischi

La gestione dei rischi è un'attività del responsabile di progetto che deve documentare l'attività nel *piano di progetto*. La procedura da seguire è la seguente:

- rilevare ed identificare nuovi rischi;
- verificare periodicamente lo stato dei rischi già rilevati e l'efficacia delle strategie di risposta pianificate;
- registrare aggiornamenti sia di nuovi rischi, sia di quelli già esistenti nel *piano di progetto*;
- se inefficaci, ridefinire le strategie adottate per la gestione dei rischi;

Il responsabile di progetto ha il compito di rilevare i rischi e di renderli noti, documentando quest'attività nel *piano di progetto*. La procedura da seguire per la gestione dei rischi è la seguente:

4.1.5.4.1 Codifica dei rischi

Le tipologie di rischi sono così codificate:

- **RT**: rischi tecnologici (e.g. rischi correlati all'utilizzo di tecnologie sconosciute o troppo nuove);
- **RI**: rischi interpersonali (e.g. rischi legati a problemi di natura sociale);
- **RO**: rischi organizzativi (e.g. rischi relativi a possibili errori nella pianificazione).

4.2 Formazione

4.2.1 Scopo

È il processo che fornisce e mantiene la formazione del personale. L'acquisizione, la disponibilità, lo sviluppo, l'esecuzione e la manutenibilità dei prodotti software sono fortemente dipendenti dalle conoscenze e dalle capacità dei componenti del gruppo. Pertanto, ciascun individuo del gruppo dovrà procedere in modo autonomo e periodico con lo studio individuale delle tecnologie utilizzate.

4.2.2 Studio individuale

Ogni componente del gruppo è tenuto ad apprendere le conoscenze necessarie a svolgere il processo di cui è responsabile. È altresì incoraggiato ad approfondire quegli argomenti che possano portare a un miglioramento delle proprie prestazioni o dei processi che sta eseguendo. In particolare, si evidenziano i seguenti strumenti utilizzati:

- **Google Hangouts**: servizio che offre la possibilità di fare videoconferenze e chiamate VoIP, utilizzato per alcuni incontri interni;
- **Telegram**: strumento di messaggistica utilizzato per la gestione del gruppo e per le comunicazioni sia interne che esterne;
- **GitDesktop**: interfaccia con GUI per utilizzare Git più comodamente sul proprio desktop;
- **GitHub**: per il versionamento e il salvataggio in remoto di tutti i file riguardanti il progetto.
- **Google Calendar**: utilizzato per organizzare le riunioni interne;

- **Google Drive**: utilizzato per la stesura di file che sono soggetti a molti cambiamenti e devono essere visibili a tutti nella loro versione più aggiornata, come ad esempio il *glossario*;
- **GanttProject**: per il supporto alla pianificazione del progetto e alla realizzazione di diagrammi di Gantt;
- **LaTeX**: come linguaggio di markup per la stesura dei testi;
- **Visual Studio Code**: editor di testo per LaTeXche offre un'interfaccia grafica, attraverso l'estensione *Latex Workshop*, in grado di compilare e visualizzare una preview dei documenti;
- **Sistemi operativi**: i requisiti non indicano la necessità di usare un sistema operativo specifico, verranno quindi utilizzati Windows, Linux e Mac OS dai diversi membri del team.

4.2.3 Scambi di esperienza

Una delle norme del progetto obbliga i componenti a svolgere ogni ruolo almeno una volta. A causa di tale restrizione, per mantenere un buon livello di efficienza, è necessario ridurre quanto possibile il periodo di formazione di un membro in preparazione a svolgere un differente ruolo, e a continuare il processo svolto dal precedente collega.
Per perseguire tale scopo, ogni settimana, *Onion Software* svolge degli incontri di scambio di esperienza, nel quale i membri condividono le proprie esperienze acquisite, e il proprio *way of working* nello svolgimento di quel ruolo, formando gli altri colleghi con le proprie conoscenze.

A Appendice

A.1 Ciclo di Deming (PDCA)

Ogni processo deve essere organizzato basandosi sul principio del miglioramento continuo:

- ***plan*** (pianificare): viene definito un piano che basandosi sulla definizione di problemi e obiettivi pianifica compiti, assegna responsabilità, studia il caso, analizza le cause della criticità e definisce azioni correttive;
- ***do*** (eseguire): vengono implementate le attività secondo le linee definite durante la fase di pianificazione;
- ***check*** (valutare): viene verificato l'esito delle azioni di miglioramento rispetto alle attese;
- ***act*** (agire): vengono applicate le correzioni necessarie per colmare le carenze rilevate e vengono standardizzate le attività correttamente eseguite.

A.1.1 Utilità

Il ciclo di Deming è uno strumento molto utile per sviluppare:

- procedure quotidiane di gestione per l'individuo e/o la squadra;
- processi per la soluzione di problemi;
- gestione di progetti;
- sviluppo continuo;
- sviluppo del fornitore;
- sviluppo delle risorse umane;
- sviluppo di nuovi prodotti;
- verifiche e revisioni.

Nel contesto della continuità aziendale, il ciclo PDCA è la metodologia fondamentale di approccio per gestire le attività in corso dei piani di *business continuity*.

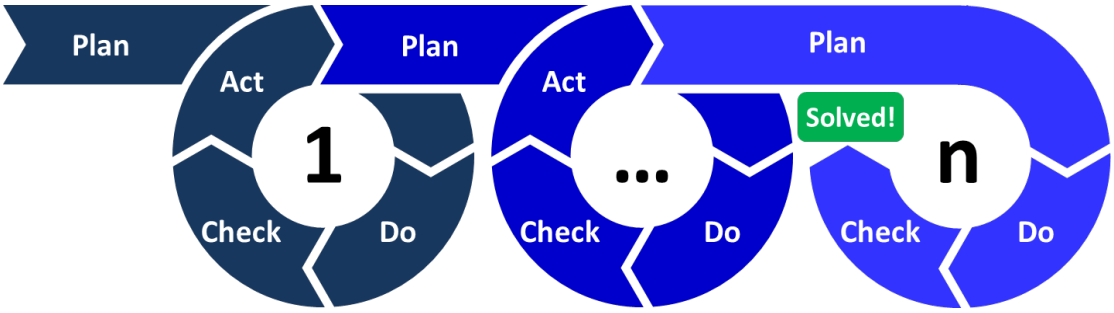

Figura 1: *Ciclo di Deming*[13]

[13] https://upload.wikimedia.org/wikipedia/commons/4/42/PDCA-Multi-Loop.png

A.2 ISO/IEC 15504

ISO/IEC 15504, anche conosciuta come SPICE (*Software Process Improvement and Capability Determination*), è un insieme di nove documenti di standard tecnici relativi ai processi di sviluppo del software, alle loro funzioni di business e, in particolare, alla loro valutazione.
La norma contiene un modello di riferimento che definisce una dimensione del processo e una dimensione della capacità (rispettivamente *process dimension* e *capability dimension*). La *process dimension* comprende i seguenti processi di:

- cliente/fornitore;
- ingegneria;
- supporto;
- gestione;
- organizzazione.

La *capability dimension* definisce una scala di maturità a cinque livelli (più il livello base, detto *livello 0*) così definiti:

- livello 0: processo incompleto;
- livello 1: processo eseguito;
- livello 2: processo gestito;
- livello 3: processo stabilito;
- livello 4: processo prevedibile;
- livello 5: processo ottimizzato.

La capacità (o maturità) dei processi è misurata tramite attributi definiti a livello internazionale, che sono:

- performance del processo: capacità di un processo di raggiungere gli obiettivi trasformando input identificabili in output identificabili;
- gestione delle performance: capacità del processo di elaborare un prodotto coerente con gli obiettivi fissati;
- gestione del prodotto di lavoro: capacità del processo di elaborare un prodotto documentato, controllato e verificato;
- definizione del processo: l'esecuzione del processo si basa su standard di processo per raggiungere i propri obiettivi;
- distribuzione del processo: capacità del processo di attingere a risorse tecniche e umane appropriate per essere attuato efficacemente;
- misurazione del processo: gli obiettivi e le misure di prodotto e di processo vengono usati per garantire il raggiungimento dei traguardi definiti in supporto ai target aziendali;
- controllo del processo: il processo viene controllato tramite misure di prodotto e processo per effettuare correzioni migliorative al processo stesso;
- innovazione del processo: i cambiamenti strutturali, di gestione e di esecuzione vengono gestiti in modo controllato per raggiungere i risultati fissati;
- ottimizzazione del processo: le modifiche al processo sono identificate e implementate per garantire il miglioramento continuo nella realizzazione degli obiettivi di business dell'organizzazione.

Ciascuna attributo di processo consiste di una o più pratiche generiche che a loro volta sono elaborate in *indicatori della pratica* che aiutano nella fase di valutazione delle prestazioni. Ciascun attributo del processo è valutato secondo una scala a quattro valori (*n-p-l-f*):

- *not achieved* (non raggiunto), 0 - 15%;
- *partially achieved* (parzialmente raggiunto), >15% - 50%;
- *largely achieved* (ampiamente raggiunto), >50% - 85%;
- *fully achieved* (completamente raggiunto), >85% - 100%.

A.3 ISO/IEC 9126

Con la sigla ISO/IEC 9126 si individua una serie di normative e linee guida, sviluppate dall'ISO (Organizzazione internazionale per la normazione) in collaborazione con l'IEC (Commissione Elettrotecnica Internazionale), preposte a descrivere un modello di qualità del software. Il modello propone un approccio alla qualità in modo tale che le società di software possano migliorare l'organizzazione, i processi e, come conseguenza concreta, la qualità del prodotto sviluppato.
La norma tecnica relativa alla qualità del software si compone di quattro parti:

- modello della qualità del software;
- metriche per la qualità esterna;
- metriche per la qualità interna;
- metriche per la qualità in uso.

A.3.1 Modello di qualità

Il modello di qualità stabilito nella prima parte dello standard (ISO/IEC 9126-1) è classificato da sei caratteristiche generali (funzionalità, affidabilità, efficienza, usabilità, manutenibilità, portabilità) e varie sotto-caratteristiche misurabili attraverso delle metriche. La funzionalità è un requisito funzionale a differenza delle ultime cinque caratteristiche, chiamate invece *requisiti non funzionali* o *requisiti di qualità*; in particolare, questi ultimi sono materia fondamentale nello studio dell'ingegneria del software. Il modello è articolato nel seguente modo:

- **funzionalità**: è la capacità di un prodotto software di fornire funzioni che soddisfano esigenze stabilite, necessarie per operare sotto condizioni specifiche e garantire:
  - **appropriatezza**: rappresenta la capacità del prodotto software di fornire un appropriato insieme di funzioni per i compiti e gli obiettivi prefissati all'utente;
  - **accuratezza**: è la capacità del prodotto software di fornire i risultati concordati o i precisi effetti richiesti;
  - **interoperabilità**: è la capacità del prodotto software di interagire ed operare con uno o più sistemi specificati;
  - **conformità**: è la capacità del prodotto software di aderire a standard, convenzioni e regolamentazioni rilevanti al settore operativo a cui vengono applicate;
  - **sicurezza**: è la capacità del prodotto software di proteggere informazioni e dati negando in ogni modo che persone o sistemi non autorizzati possano accedervi o modificarli, e che a persone o sistemi effettivamente autorizzati non sia negato l'accesso ad essi;
- **affidabilità**: è la capacità del prodotto software di mantenere uno specificato livello di prestazioni quando usato in date condizioni per un dato periodo e garantire:
  - **maturità**: è la capacità di un prodotto software di evitare che si verifichino errori, malfunzionamenti o siano prodotti risultati non corretti;
  - **tolleranza agli errori**: è la capacità di mantenere livelli predeterminati di prestazioni anche in presenza di malfunzionamenti o usi scorretti del prodotto;
  - **recuperabilità**: è la capacità di un prodotto di ripristinare il livello appropriato di prestazioni e di recupero delle informazioni rilevanti in seguito a un malfunzionamento. A seguito di un errore, il software può risultare non accessibile per un determinato periodo di tempo, questo arco di tempo è valutato proprio dalla caratteristica di recuperabilità;
  - **aderenza**: è la capacità di aderire a standard, regole e convenzioni inerenti all'affidabilità;

- **efficienza**: è la capacità di fornire appropriate prestazioni relativamente alla quantità di risorse usate e garantire:
  - **comportamento rispetto al tempo**: è la capacità di fornire adeguati tempi di risposta, elaborazione e velocità di attraversamento sotto determinate condizioni;
  - **utilizzo delle risorse**: è la capacità di utilizzo di quantità e tipo di risorse in maniera adeguata;
  - **conformità**: è la capacità di aderire a standard e specifiche sull'efficienza;
- **usabilità**: è la capacità del prodotto software di essere capito, appreso, usato e benaccetto dall'utente quando usato sotto condizioni specificate; deve inoltre garantire:
  - **comprensibilità**: esprime la facilità di comprensione dei concetti del prodotto, permettendo all'utente di comprendere se il software è appropriato;
  - **apprendibilità**: è la capacità di ridurre l'impegno richiesto agli utenti per imparare ad usare l'applicazione;
  - **operabilità**: è la capacità di mettere in condizione gli utenti di utilizzarlo per i propri scopi e controllarne l'uso;
  - **attrattiva**: è la capacità del software di essere piacevole per l'utente che ne fa uso;
  - **conformità**: è la capacità del software di aderire a standard o convenzioni relativi all'usabilità;
- **manutenibilità**: è la capacità del software di essere modificato, includendo correzioni, miglioramenti o adattamenti e garantire:
  - **analizzabilità**: rappresenta la facilità con la quale è possibile analizzare il codice per localizzare un errore;
  - **modificabilità**: è la capacità del prodotto software di permettere l'implementazione di una specificata modifica (sostituzioni componenti);
  - **stabilità**: la capacità del software di evitare effetti inaspettati derivanti da modifiche errate;
  - **testabilità**: è la capacità di essere facilmente testato per validare le modifiche apportate al software;
- **portabilità**: è la capacità del software di essere trasportato da un ambiente di lavoro ad un altro. Ambiente che può variare dall'hardware al sistema operativo; deve inoltre garantire:
  - **adattabilità**: la capacità del software di essere adattato a differenti ambienti operativi senza dover applicare modifiche rispetto a quelle fornite per il software considerato;
  - **installabilità**: la capacità del software di essere installato in uno specificato ambiente;
  - **conformità**: la capacità del prodotto software di aderire a standard e convenzioni relative alla portabilità;
  - **sostituibilità**: è la capacità di essere utilizzato al posto di un altro software per svolgere gli stessi compiti nello stesso ambiente.

A.3.2 Qualità esterne

Le metriche esterne misurano i comportamenti del software sulla base dei test e dell'operatività durante la sua esecuzione, in funzione degli obiettivi stabiliti in un contesto tecnico rilevante o di business.

A.3.3 Qualità interne

Le qualità interne (o metriche interne), si applicano al software non eseguibile durante le fasi di progettazione e codifica. Le misure effettuate permettono di prevedere il livello di qualità esterna ed in uso del prodotto finale, poiché gli attributi interni influiscono su quelli esterni e quelli in uso. Le metriche interne permettono di individuare eventuali problemi che potrebbero influire sulla qualità finale del prodotto prima che sia realizzato il software eseguibile. Esistono metriche che possono simulare il comportamento del prodotto finale tramite simulazioni.

A.3.4 Qualità in uso

La qualità in uso rappresenta il punto di vista dell'utente sul software. Il livello di qualità in uso è raggiunto quando è stato raggiunto sia il livello di qualità esterna sia il livello di qualità interna. La qualità in uso, quindi, permette di abilitare specifici utenti ad ottenere specifici obiettivi con efficacia, produttività, sicurezza e soddisfazione. La qualità in uso deve garantire:

- **efficacia**: è la capacità del software di permettere gli utenti di raggiungere gli obiettivi specificati con accuratezza e completezza;
- **produttività**: è la capacità di permettere agli utenti di spendere una quantità di risorse appropriate in relazione all'efficacia ottenuta in uno specifico contesto d'uso;
- **soddisfazione**: è la capacità del prodotto software di soddisfare gli utenti;
- **sicurezza**: rappresenta la capacità del prodotto software di raggiungere accettabili livelli di rischio di danni a persone, al software, ad apparecchiature o all'ambiente operativo in uso.

A.4 The Twelve-Factor App

È una metodologia di sviluppo già descritta in sezione §2.2.4.2.11. Come da nome, si basa su dodici fattori:

1. **codebase**: una sola *codebase*$_G$ sotto controllo di versione, tanti deployment;
2. **dipendenze**: dipendenze dichiarate ed isolate;
3. **configurazione**: memorizzare le informazioni di configurazione nell'ambiente;
4. **backing service**: trattare i *backing service*$_G$ come risorse;
5. **build, release, esecuzione**: separare in modo netto lo stadio di build dall'esecuzione;
6. **processi**: eseguire l'applicazione come uno o più processi stateless;
7. **binding delle porte**: esportare i servizi tramite *binding delle porte*$_G$;
8. **concorrenza**: scalare attraverso il process model;
9. **rilasciabilità**: massimizzare la robustezza con avvii veloci e chiusure non brusche;
10. **parità tra sviluppo e produzione**: mantenere lo sviluppo, staging e produzione simili il più possibile;
11. **log**: trattare i log come stream di eventi;
12. **processi di amministrazione**: eseguire i task di amministrazione/management come processi una tantum.

A.4.1 Codebase

L'app deve sempre attenersi al controllo di versione. Esiste una relazione *uno-a-uno* tra codebase e applicazione, ma ci saranno comunque tanti deployment. In uno stesso istante possono esistere più versioni della codebase. Sono possibili alcune varianti di questa modalità:

- se ci sono più codebase, non si parla più di applicazione ma di sistema distribuito;
- più app non possono condividere la stessa codebase, ciò è possibile soltanto con un apposito sistema di dipendenza.

A.4.2 Dipendenze

Un'applicazione che aderisce alla twelve-factor:

- non si basa mai sull'esistenza implicita di librerie system-wide, come PAN per Perl e Rubygems per Ruby;
- le dipendenze vengono tutte dichiarate tramite un file manifest dedicato;
- viene contemplato l'uso di un tool di isolamento delle dipendenze durante l'esecuzione, in modo tale da assicurarsi che non ci siano delle dipendenze implicite che creino interferenze nel sistema in cui ci si trova;
- la specifica completa ed esplicita delle dipendenze si applica in modo uniforme, sia in production che in sviluppo;
- le operazioni di dichiarazione ed isolamento vanno sempre effettuate, non ha importanza quale sia il toolchain usato;
- l'applicazione non si deve mai basare sull'esistenza di un qualsiasi tool di sistema, come per ImageMagick e Curl, perché non si ha la certezza che questo tool possa essere presente in tutti i sistemi in cui girerà in futuro.

A.4.3 Configurazione

Un'app conforme alla metodologia richiede una separazione ben definita delle impostazioni di configurazione dal codice. Per fare ciò si deve memorizzare tutte le impostazioni in variabili d'ambiente per le seguenti ragioni:

- sono molto semplici da cambiare da deployment a deployment senza dover toccare direttamente il codice;
- c'è una probabilità molto bassa che siano erroneamente incluse all'interno del repository, a differenza dei classici file di configurazione;
- sono file indipendenti sia dal linguaggio che dal sistema operativo utilizzato.

Affinché il prodotto finale ne possa risentire positivamente in termini di scalabilità, le variabili d'ambiente non sono mai raggruppate e classificate sotto ambienti specifici che vengono gestiti in modo totalmente indipendente in ogni deployment.

A.4.4 Backing service

Il codice di un'app twelve-factor non fa distinzioni tra servizi in locale o third party. Per l'applicazione entrambi sono risorse connesse, accessibili via url e dispongono di credenziali memorizzate nell'opportuno file di configurazione.
Ad un qualsiasi deployment di un'applicazione twelve-factor si deve poter permettere di passare velocemente da un database MySQL locale ad uno third party senza alcuna modifica al codice. A cambiare dovrebbero essere solo i file di configurazione necessari. Ogni backing service è quindi definibile come una risorsa connessa. Un database MySQL è una risorsa. Due database MySQL saranno visti come due distinte risorse. Un'app twelve-factor vede questi database come risorse anche per sottolineare la separazione dal deployment a cui fanno riferimento. Le risorse possono essere collegate e scollegate da un deployment a piacimento.

A.4.5 Build, release, esecuzione

Una codebase viene trasformata in deployment attraverso tre fasi:

- build: converte il codice del repo in un'applicazione eseguibile, utilizzando una determinata versione del codice ad una specifica commit. Nella fase di build vengono compilati i binari con gli asset appropriati includendo anche le eventuali dipendenze. Una fase di build può essere avviata manualmente da uno sviluppatore oppure avere un sistema automatico che la esegue ad ogni modifica effettuata al codice;
- release: preleva la build prodotta nella fase precedente e la combina con l'attuale insieme di impostazioni di configurazione del deployment specifico. La release risultante quindi contiene sia la build che le impostazioni. Ogni release dovrebbe possedere un ID univoco di rilascio. I tool di deployment offrono tipicamente dei tool di gestione delle release, in particolare alcuni dedicati ad un rollback verso una release precedente;
- esecuzione: conosciuta anche come "runtime", vede l'applicazione in esecuzione nell'ambiente di destinazione attraverso l'avvio di processi della release scelta.

A.4.6 Processi

L'app viene eseguita nell'ambiente di esecuzione come uno o più processi. I processi twelve-factor sono *stateless*, cioè senza stato, e *share-nothing*, cioè con nessuna condivisione. Tutti i dati che devono persistere devono essere memorizzati in un backing service, come ad esempio un database. I packager di asset usano il file system come cache per gli asset compilati. Un'app twelve-factor richiede che questa compilazione sia effettuata durante la fase di build e non a runtime. Le *sticky session* sono una palese violazione della metodologia twelve-factor. I dati di sessione sono un ottimo candidato per quei sistemi di immagazzinamento dati che offrono la feature di scadenza.

A.4.7 Binding delle porte

L'applicazione twelve-factor è completamente *self-contained* (contenuta in se stessa) e non si affida ad un altro servizio, come un webserver, nell'ambiente di esecuzione. La web app esporta *Http*$_G$ come un servizio effettuando un binding specifico ad una porta, rimanendo in ascolto su tale porta per le richieste in entrata. Tale funzionalità viene frequentemente implementata tramite dichiarazione delle opportune dipendenze, aggiungendo una libreria webserver all'applicazione. Http non è l'unico servizio che può essere esportato tramite *port binding*, in realtà quasi ogni tipo di software può essere eseguito tramite uno specifico binding tra processo e porta dedicata. Da notare inoltre che usare il binding delle porte permette ad un'applicazione di diventare il backing service di un'altra applicazione.

A.4.8 Concorrenza

In un'applicazione twelve-factor, i processi sono definiti *first class citizen*. La visione del concetto di processo prende spunto dai demoni in *Unix*$_G$, che equivalgono ai servizi su sistemi Windows. Attraverso l'uso di questo modello, lo sviluppatore può realizzare la propria applicazione in modo tale da farle gestire senza problemi diversi livelli di carico di lavoro, assegnando ogni tipo di lavoro ad un tipo di processo ben definito. Come per le richieste Http possono essere gestite da un web process, mentre i compiti più lunghi, svolti in background, possono essere gestiti da un processo separato.
Il modello di processo così come presentato favorisce la scalabilità del sistema, in quanto la natura orizzontalmente partizionabile, e non soggetta a condivisioni, di un processo twelve-factor permette di gestire la concorrenza in modo semplice ed affidabile. I processi di un'applicazione twelve-factor non dovrebbero essere soggetti a *daemonizing*, cioè la possibilità di essere avviati come demoni o servizi, ma dovrebbero fare affidamento a sistemi di *process manager* del sistema operativo, ad esempio Systemd o SysVinit, in modo da rispondere adeguatamente ed automaticamente a crash o riavvii improvvisi.

A.4.9 Rilasciabilità

I processi di un'applicazione twelve-factor sono rilasciabili, cioè possono essere avviati o fermati senza problemi al momento del bisogno. Questa caratteristica ovviamente facilita le procedure di scaling, deployment rapido della codebase o cambi dei file di configurazione. Quindi i processi dovrebbero:

- ambire a minimizzare i tempi di avvio. Idealmente, un processo impiega pochi secondi dal tempo di lancio al momento in cui tutto è pronto per ricevere nuove richieste. Dei tempi brevi di avvio inoltre forniscono una maggiore agilità in fase di release, il tutto a vantaggio della robustezza dell'applicazione;
- terminare in modo tutt'altro che brusco, quindi graduale, in caso di ricezione di un segnale SIGTERM, cioè di una richiesta di terminazione da parte del sistema operativo. Per un'applicazione web la giusta terminazione di un processo viene ottenuta quando si cessa innanzitutto di ascoltare sulla porta dedicata al servizio, evitando quindi di ricevere altre richieste, permettendo poi di terminare le richieste esistenti ed infine di terminare definitivamente;
- essere robusti nei confronti di situazioni di crash improvviso, cosa che si verifica come in caso di problemi a livello di hardware sottostante. Nonostante questa seconda evenienza si verifichi meno frequentemente di una chiusura con SIGTERM, può comunque succedere. L'approccio raccomandato, in questi casi, è l'uso di un sistema robusto di code che rimette nella lista delle richieste il lavoro che non può essere immediatamente completato.

Ad ogni modo, una buona applicazione twelve-factor deve poter gestire senza problemi le terminazioni inaspettate, senza che questo generi problemi alla successiva esecuzione.

A.4.10 Parità tra sviluppo e produzione

Un'applicazione twelve-factor è progettata per il rilascio continuo. Il suo obiettivo è quello di minimizzare le differenze di tempo, personale e strumenti. Per fare ciò sono necessari alcuni accorgimenti:

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

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

A.4.11 Log

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

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

Uno stream di eventi di un'applicazione può essere quindi indirizzato verso un file, o visionato in tempo reale su un terminale. Questi sistemi hanno ottimi tool per effettuare un lavoro di analisi del comportamento dell'applicazione, come:

- ricerca di specifici eventi nel passato;
- grafici per rappresentare dei trend, come le richieste per minuto;
- attivazione di avvisi specifici in base a regole definite dall'utente, come nel caso in cui la frequenza di eventi al minuto sale oltre una certa soglia.

A.4.12 Processi di amministrazione

È l'array dei processi che vengono usati durante le normali operazioni dell'applicazione. Lo sviluppatore potrebbe richiedere l'esecuzione di alcuni task come:

- esecuzione delle modifiche alla struttura del database;
- utilizzo di una console in modo tale da avviare del codice arbitrariamente o analizzare alcuni aspetti dell'applicazione specifici;
- esecuzione *one-time* di alcuni script specifici.

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