# SOFTWARE LIFE CYCLE

Il nostro gruppo ha deciso di adottare un metodo di sviluppo agile per lo sviluppo di questo progetto. Essendo un tipo di prodotto non convenzionale e di dimensioni relativamente ridotte, eseguire un approccio model driven sarebbe stato controintuitivo e poco funzionale. Il nostro metodo di lavoro è stato simile all’extreme programming, ma senza delle finestre di tempo definite, ogni giorno bisognava fare il possibile. Definire i requisiti fin da subito era complesso, quindi abbiamo deciso di eseguire del prototyping (in particolare Evolutionary Prototyping). Poiché eravamo noi i clienti, i prototipi ci hanno aiutato a visualizzare il prodotto e ad applicare modifiche di volta in volta. Il codice doveva sempre essere funzionante ad ogni commit quindi abbiamo usato branch per evitare di implementare funzionalità che rompessero quello che era già presente, ed eseguire un merge una volta assicurato che tutto funzionasse. L’evoluzione è stata incrementale, ogni commit ha portato a delle nuove funzionalità, senza però stravolgere l’intero funzionamento. Gli incontri sono stati poco frequenti ed è stato eseguito molto refactoring.

# CONFIGURATION MANAGEMENT

Github è stato il punto centrale per la sincronizzazione del lavoro tra i membri del gruppo. Esso ci ha aiutato a definire le task da eseguire e a visualizzare il progresso del lavoro.

Ogni commit rappresenta i cambiamenti eseguiti nel codice con la corrispettiva descrizione di cosa è stato modificato (Change Oriented).

# PEOPLE MANAGEMENT AND TEAM ORGANIZATION

Per la gestione del team abbiamo deciso di adottare della burocrazia professionale, dove ognuno aveva autonomia nello svolgere la propria task.

Per quanto riguarda il focus, esso era rivolto verso il risultato finale (Task Directedness) dato che l’obiettivo era quello di ottenere un prodotto di qualità, gradevole da usare e di cui essere fieri.

In generale la team organization è stata del tipo: Chief Programmer Team. Dove Marchesi è il chief grazie alla sua discreta conoscenza di Flutter e visione del progetto, mentre il resto del gruppo ha agito come lavoratori specializzati che hanno contribuito allo sviluppo portando avanti task specifiche e producendo risultati di qualità.

# SOFTWARE QUALITY

Ecco la lista di qualità/qualità d’uso che abbiamo deciso di implementare nel nostro prodotto:

* Effectiveness & Satisfaction: Il prodotto ci ha sempre soddisfatto durante lo sviluppo per i risultati ottenuti sul prototipo. Il prodotto ha centrato l’obbiettivo di essere un gioco divertente, rigiocabile e informativo.
* Usability: pensiamo che i comandi siano intuitivi e facili da utilizzare. Qualsiasi giocatore dovrebbe essere in grado di capire in poco tempo come il gioco funziona e quali meccaniche esso comprende.
* Maintainability: Durante la scrittura del codice è stata eseguita una massiccia documentazione delle classi e dei metodi utilizzati. Questo in aggiunta alla documentazione esterna e ad una efficiente suddivisione delle cartelle, rende la manutenzione veloce e semplice, nonostante la notevole quantità di codice e documenti.
* Portability: Il codice è stato scritto con Flutter, il che rende il prodotto multiplatform, inoltre durante lo sviluppo del codice sono state tenute conto le best practices che permettono di far adattare il software a diversi dispositivi con diverse dimensioni di schermo.

Cosa si potrebbe migliorare:

* Reliability: aumentare il numero di test ed espanderli anche all’interfaccia grafica.
* Efficiency: Distribuire in modo migliore la creazione della partita
* Maintainability: Aumentare la suddivisione tra Model View e Controller.

# SOFTWARE ARCHITECTURE

Il nostro prodotto può essere suddiviso in componenti e connettori. Per quanto riguarda i componenti, essi sono rappresentati dalle classi. Ognuna di esse ha dei ruoli specifici, c’è che funge da memoria (es. Stanza, Personaggio), altri che fungono da elementi computazionali (es. CreazionePartita) ed infine il componente manager (Partita). I due tipi principali di connettori sono: Procedure Call e Implicit Invocation. Il primo viene utilizzato molto spesso da classi e widget che chiamano delle RPC per far eseguire delle operazioni, mentre il secondo tipo di connettore è utilizzato grazie alla libreria Provider e ChangeNotifier. Queste hanno permesso di costruire un sistema basato su eventi e su ascoltatori pronti a modificare il loro stato in base all’arrivo di eventi. Gli stili architetturali utilizzati sono quindi stati:

* Programma principale con subroutine: Programma principale definito dall’albero di widget, da cui ogni widget aveva a disposizione dei metodi da chiamare per poter far eseguire operazioni in remoto.
* Model View Controller: abbiamo cercato di dividere il più possibile la UI, dai dati e dalla logica del programma. Dentro i widget infatti, non viene eseguita quasi nessuna computazione e/o gestione dei dati. Tutto viene eseguito da funzioni in classi apposite.
* Implicit Invocation: Provider e ChangeNotifier ci hanno permesso di condividere le istanze di Partita e Personaggio lungo tutto l’albero di widget. Questo rappresenta lo stato della partita e tutte le informazioni necessarie per il funzionamento. Ogni volta che avvengono dei cambiamenti in queste due classi vengono generati degli eventi che vengono ascoltati da dei listeners che, in base al tipo di evento generato, reagiscono di conseguenza.

In generale per avere differenti punti di vista sull’applicazione a livello di granularità diversi, si possono utilizzare i diagrammi UML creati, che offrono sia una visione statica (Use case e Class) e sia una visione dinamica (Activity, Sequence, StateChart) della struttura.

# SOFTWARE DESIGN

Per il design oltre ai grafici UML, abbiamo utilizzato dei tool per il calcolo della complessità e per il rispetto delle best practices. In particolare, abbiamo usato questa libreria di dart: <https://pub.dev/packages/dart_code_metrics> che tramite dei comandi specifici ci permette di visualizzare il numero di classi, metodi, file non utilizzati, presenza di antipattern ecc.

Ci permette inoltre di impostare un valore massimo soglia per diverse proprietà, esempio: numero max di metodi in una classe, numero max di parametri, difficoltà massima calcolata con metodo Halstead

Per quanto riguarda i pattern utilizzati abbiamo:

* Observer: Tramite il Provider abbiamo reso Observable l’istanza di partita e di personaggio. Le altre classi posso creare dei listeners rivolti verso questi oggetti. Ogni volta che una delle due istanza subisce dei cambiamenti il metodo notifyListeners() viene chiamato che procede ad avvisare del cambiamento avvenuto tutti gli ascoltatori
* Player Role: In questo caso abbiamo un’associazione tra Nemico e Stanza, dove però entrambi sono super classi; infatti, nemico può poi essere una sottoclasse Boss o Scagnozzo, mentre la stanza può essere una sottoclasse StanzaCombattimento o StanzaEsplorazione.
* Model View Controller: come spiegato in precedenza è stato implementato il MVC pattern per rendere più funzionale e migliorare la qualità del codice.

# SOFTWARE TESTING

Abbiamo eseguito solamente Unit Test tramite gli strumenti forniti da dart. In un’applicazione pronta per essere mandata in produzione sarebbe bene anche scrivere test sul funzionamento della UI.

L’approccio di testing è stato quello di testare i metodi più critici e che in caso di fallimento potrebbero portare alla rottura dell’applicazione. Ovviamente l’ideale sarebbe stato scrivere test per qualsiasi metodo di qualsiasi classe utilizzata, ma data la quantità di tempo a disposizione e la dimensione del gruppo di sviluppo abbiamo puntato verso un approccio più mirato.

In particolare sono stati testati i metodi della classe Partita, della classe Stanza e di quella Personaggio utilizzando un approccio Error Based Testing.

# SOFTWARE MAINTENANCE

In generale la manutenzione del nostro prodotto è stata principalmente di due tipi: Correttiva ed Perfettiva. Anche se il prodotto non è stato effettivamente deployato, da quanto è stato possibile giocare una partita dall’inizio alla fine, abbiamo considerato il nostro prodotto completo e quindi da quel punto in poi le modifiche eseguite sono state di manutenzione.

Durante lo sviluppo sono state eseguite molte attività di refactoring, quelle più frequenti sono state:

* Spostamento di codice da una classe all’altra (per separare model, view e controller)
* Rinominazione di variabili
* Riscrittura di metodi per eseguire la stessa operazione in modo più efficiente
* Scrittura di commenti per descrivere il funzionamento di metodi e il ruolo delle variabili

Solitamente il refactoring è stato eseguito dopo l’implementazione di ciascuna funzionalità. Questo perché una volta implementato un nuovo pezzo di codice funzionante, l’obbiettivo era di adattarlo alla qualità del codice già presente e renderlo più leggibile e mantenibile.