Francesco Regonesi 844604, Davide Rusconi 845592, Riccardo Sartori 845063

DOCUMENTAZIONE DEL SOFTWARE D-RISK

Progettazione

Sono stati prodotti i seguenti diagrammi (allegati alla documentazione):

- **Diagramma dei casi d'uso:** rappresenta i due casi d'uso e l'attore con il quale interagiscono. *VEDI DIAGRAMMA*
- **Modello di dominio**: è stato usato per identificare i principali concetti del gioco del Risiko e come base per lo sviluppo delle classi del package *model* (ovviamente sono state fatte ulteriori modifiche in questo passaggio). *VEDI DIAGRAMMA*
- **Diagramma delle classi:** descrive le classi del sistema, la loro struttura e le relazioni che intercorrono tra di esse. *VEDI DIAGRAMMA*
- **Diagramma SSD:** descrive il corso degli eventi relativamente al caso d'uso dell'inserimento di una nuova mappa. *VEDI DIAGRAMMA*
- **Diagramma di sequenza**: mostra tutte le interazioni tra le classi nel processo di inserimento di una mappa. *VEDI DIAGRAMMA*
- **Diagramma di attività**: mostra il flusso di operazioni e decisioni compiute da un giocatore nel corso di un turno di gioco. *VEDI DIAGRAMMA*
- **Diagramma di stato**: mostra lo stato della partita e tutti i suoi possibili cambiamenti. *VEDI DIAGRAMMA*
- **Diagramma generale** sull'architettura del software (raffinato al termine dello sviluppo anche a scopo di documentazione). *VEDI DIAGRAMMA*

Architettura generale del software

Il progetto si compone di due moduli software distinti: back-end e front-end. Questi moduli sono fra loro indipendenti e comunicano tramite protocollo HTTP.

Il back-end implementa la logica dell'applicazione senza curarsi di come questa verrà presentata all'utente, mentre il front-end si occupa di visualizzare nel browser l'interfaccia grafica e di gestire gli input dell'utente che, inoltre, vengono trasmessi al back-end per un'ulteriore elaborazione. Il front-end riproduce al suo interno soltanto alcune logiche di gioco, ossia quelle strettamente essenziali al suo corretto funzionamento e al controllo dell'input dell'utente: ovunque possibile, le logiche applicative sono delegate al backend. Data questa organizzazione è possibile in un secondo momento creare interfacce grafiche alternative da sostituire a quella proposta, senza bisogno di modificare il back-end.

Durante il processo di build del sistema i file statici (html / css / js) prodotti dalla build del front-end vengono copiati nella cartella delle risorse statiche del back-end e serviti direttamente dal server.

Tecnologie utilizzate

La parte di back-end è realizzata in linguaggio Java utilizzando il framework Spring Boot, mentre la parte di front-end è realizzata con il framework javascript Vue.js.

Per la gestione dei dati persistenti viene utilizzato come DBMS MySQL (versione 8), mentre per l'accesso applicativo ai dati è stata usata la tecnologia ORM offerta dalla libreria Spring Data JPA.

Come build system viene utilizzato Maven, mentre per il processo di quality assurance e di analisi del software sono stati utilizzati i tools Sonarqube e Understand. Il processo di build e quality assurance è stato automatizzato tramite l'utilizzo di Travis, un tool di continuous integration.

Per la rappresentazione dei grafi a front-end è stata utilizzata la libreria vis.js, mentre il framework CSS utilizzato è Vuetify (un framework CSS creato appositamente per Vue.js).

Architettura software back-end

L'architettura software del back-end è organizzata su tre layer distinti:

- *http*: è lo strato più esterno; si occupa della comunicazione HTTP con il front-end e delega il soddisfacimento delle richieste ai layer sottostanti. Contiene due ulteriori package:
    - *controller*: si occupano della comunicazione vera e propria e della gestione delle richieste.
    - *dto* (Data Transfer Object): modellano i dati inviati e ricevuti via HTTP in oggetti Java. Sono inoltre utilizzati per lo scambio di dati tra il layer *http* e il layer *service*.
- *service*: è lo strato che contiene la logica applicativa. Riceve le richieste dai controller e si interfaccia con lo strato sottostante per soddisfarle.
- *data*: è lo strato che gestisce i dati dell'applicazione e implementa le logiche di dominio. È diviso in tre package:
    - *model*: modella le classi del dominio e contiene la logica di dominio.
    - *creators*: contiene classi necessarie alla creazione di oggetti complessi che devono sottostare a una serie di vincoli.
    - *repository*: contiene classi che consentono di interfacciarsi con lo strato di persistenza (database).

Esiste inoltre un package *exceptions* che contiene le eccezioni specifiche del sistema, ciascuna mappata su un diverso status code HTTP.

Architettura software front-end

Javascript non supporta la divisione in package, tuttavia vi è un'organizzazione standard del codice in directory che viene promossa dal framework Vue:

- *components*: contiene i Component Vue, ossia dei file composti da 3 sezioni (*template*, *script* e *style*) che sono il cuore del framework. Essi supportano la reactivity, ossia

l'aggiornamento automatico dei dati presentati nel browser quando i dati sottostanti cambiano.

- *assets*: file statici che devono essere inclusi nel progetto, ad esempio immagini.
- *plugins*: contiene file di configurazione per i plugin utilizzati, in questo caso:
    - Axios: per la gestione della comunicazione HTTP.
    - Vuetify: framework CSS per Vue.
- *services*: contiene funzioni Javascript per la comunicazione HTTP con il backend.
- *store*: gestisce in modo centralizzato i dati più importanti dell'applicazione, in modo che siano accessibili da ogni component. Espone inoltre delle operazioni utilizzabili per interagire con tali dati.
- *utils*: non previsto dal framework, qui sono stati inseriti script contenenti funzioni utili e di supporto.

Patterns e principles utilizzati

Design pattern

Nell'implementazione del sistema sono stati utilizzati diversi design pattern, sia implementati di default nei framework Spring Boot e Vue.js, sia realizzati ad hoc per esigenze specifiche.

- **Inversion of control**: il flusso principale dell'applicazione è gestito dal framework, sia a back-end che a front-end. L'obiettivo del pattern è favorire la riusabilità del software: il *boilerplate code* (relativo ad esempio a gestione richieste HTTP, connessione con database, ecc.) è integrato nel framework e reso robusto ed efficiente dalla community che lo supporta, liberando tempo ed energie per sviluppare la logica applicativa vera e propria.
- **Dependency injection**: il framework Spring Boot si occupa di fornire ai costruttori dei *beans* (ossia gli oggetti annotati come *@Component*, *@RestController, @Service*, ecc.) le dipendenze di cui questi hanno bisogno. Questo pattern rende possibile l'*Inversion of Control* e dunque favorisce la modularità e riutilizzabilità del software.
- **Singleton**: i *beans* di Spring Boot, ovvero in questo caso i service ed i controller, sono gestiti come singleton; a differenza del pattern Singleton GoF (per cui c'è solo un'istanza della classe nell'intera JVM), i Singleton di Spring sono relativi al singolo container Spring IoC.
- **Repository pattern**: la comunicazione con il database avviene tramite un'interfaccia che ne nasconde i dettagli (Spring Boot implementa in modo autonomo tale interfaccia a partire dalla sua definizione e la rende disponibile); ciò consente di concentrarsi sulla logica applicativa senza curarsi di come il framework implementa la connessione al database.
- **Builder**: siccome le logiche per la costruzione di una nuova mappa sono complesse, è stato usato il design pattern Builder per separare la logica della costruzione dalla rappresentazione dell'oggetto.

- **Observer**: internamente il framework Vue.js utilizza il pattern Observer per implementare la reactivity dei suoi component: quando cambiano i dati sottostanti cambia anche la rappresentazione nel DOM.

- **Transfer Object**: vengono utilizzati dei Data Transfer Object (DTO) per rappresentare i dati in entrata e uscita e per lo scambio di dati tra i package. In questo modo è possibile mantenere indipendenti gli strati di controller e model, e di controllare in modo più fine il formato dei dati inviati e ricevuti.

- **Use Case Controller**: le richieste relative ai due casi d'uso sono gestite tramite due controller separati che fungono da punto d'accesso per l'applicazione.

Design principles

- **Acyclic Dependencies Principle**: non sono presenti dipendenze cicliche tra i package.

- **Open Closed Principle**: le classi obiettivo sono mascherate da un'interfaccia. Tale interfaccia è chiusa alle modifiche, tuttavia è sempre possibile implementare delle nuove tipologie di obiettivo che implementino tale interfaccia.

- **Dependency Inversion Principle**: ad eccezione della factory, tutte le classi dipendono dall'interfaccia Obiettivo e mai dalle sue implementazioni.

- **Single Responsibility Principle**: applicato in generale ovunque possibile e ovunque non in contrasto con altri principi, ad esempio: controller, DTO, MappaBuilder ecc.

Pattern architetturali

- **Layered Architecture**: il software è diviso nei layer citati nell'architettura del back-end. Gli strati più esterni dipendono da quelli più interni e non viceversa. Questo pattern è stato utilizzato per migliorare la modularità e riutilizzabilità del software.

- **Active Record Pattern**: i dati salvati nel database sono mappati sulle classi del model. Ogni record corrisponde ad un'istanza della classe corrispondente. Questo pattern consente di maneggiare i dati salvati nel database in modo semplice ed intuitivo, inoltre è incoraggiato dalla libreria *Spring Data JPA*.

- **Service Layer**: la logica applicativa è separata da quella di dominio e gestita nello strato Service. Ciò serve e separare la logica applicativa da quella di dominio e a mantenere indipendenti i controllers dalle classi del model.

Quality assurance

Durante lo sviluppo dell'applicazione è stato utilizzato SonarQube per l'analisi della qualità del codice. La maggior parte delle problematiche evidenziate sono state risolte.

I restanti problemi, che sono stati ignorati, sono i seguenti:

- **Make "Partita" serializable or don't store it in the session.**

  Come indicato nel commento su SonarQube, l'eventualità che questo problema si verifichi è molto remota e non rilevante per la nostra applicazione, e non giustifica l'implementazione dell'interfaccia Serializable.

- **Make sure that using this pseudorandom number generator is safe here.** Il generatore random è utilizzato solo per il lancio dei dadi durante il combattimento, non rappresenta quindi un security hotspot.

- **Codice duplicato in mappa - *mappaDTO* - *compactMappaDTO***: non è stato ritenuto opportuno legare queste classi tramite ereditarietà per diminuire il codice duplicato, che in ogni caso è limitato ad alcuni attributi e ai loro metodi getter.

- **Codice duplicato in networkUtils.js:** si tratta di un array con i colori dei giocatori. La segnalazione è stata quindi ignorata.

Analisi del codice

Dopo aver sviluppato una versione funzionante dell'applicazione (tag Git: **versione_funzionante_pre_refactoring**) è stata effettuata una prima analisi con Understand (*CLUSTER CALL* – *METRICS TREEMAP*). L'unico problema serio individuato è stato un eccessivo numero di dipendenze verso altre classi per quanto riguarda la classe *PartitaService*, sia interne che esterne al package (**global breakable**), accompagnato da un'eccessiva complessità ciclomatica della classe stessa.

Si è pertanto deciso di lavorare alla riduzione della complessità ciclomatica di *PartitaService* e di rivedere la suddivisione delle responsabilità tra le classi. Prima di effettuare tali modifiche sono stati scritti i test in modo da raggiungere una coverage dell'80% sul codice Java.

Le principali modifiche sono state:

- riduzione della complessità ciclomatica di alcune classi.
- eliminazione di due service e spostamento delle logiche nelle classi del model ritenute più appropriate.
- raffinamento della struttura dei package in più punti.
- rimozione di classi superflue nel model.
- introduzione delle classi del package creators.

Al termine del refactoring è stata effettuata una nuova analisi che ha confermato una riduzione della massima complessità ciclomatica e una migliore distribuzione delle logiche nelle varie classi. Il grafico *Cluster Call* conferma l'assenza di dipendenze cicliche tra package.

(CLUSTER CALL – METRICS TREEMAP)

Possibili evolutive

La naturale evoluzione dell'applicazione prevede la possibilità di giocare una partita in modo asincrono su più dispositivi (uno per giocatore).

L'applicazione è stata sviluppata in modo da favorire questa evolutiva: ad esempio le richieste HTTP *attacco* e *difesa* sono separate in quanto richiedono l'interazione di due giocatori diversi: l'attaccante lancia un attacco e il difensore indica con quante armate si difende. Ciò non è strettamente necessario in una partita mono-browser ma certamente lo è se si gioca con browser differenti.

Lo sviluppo di questa evolutiva comporta inoltre una serie di complicazioni tecniche e rende necessarie numerose considerazioni in merito ai requisiti dell'applicazione, di cui in seguito vengono riportate le principali.

Innanzitutto sul lato tecnico andrebbe gestito in maniera differente il salvataggio delle informazioni sulla partita in corso, in quanto queste non sarebbero più vincolate ad una singola sessione. Per fare ciò sarebbe necessario implementare un punto di accesso centralizzato e veloce a tali informazioni, ad esempio un database Redis.

Inoltre il server dovrebbe essere in grado di comunicare informazioni ai client in modo asincrono e di propria iniziativa, in modo da informarli in tempo reale sulle mosse compiute dagli altri giocatori. Per far si che ciò avvenga si potrebbe utilizzare la tecnologia delle WebSocket.

La struttura a layer del software back-end e la separazione netta tra back-end e front-end dovrebbe permettere l'implementazione di queste modifiche senza bisogno di modificare i layer Service e Model, bensì agendo solamente sullo strato di Controller (se necessario aggiungendo un layer per la gestione delle WebSocket e della cache Redis) e sul Frontend.

Una volta implementata la possibilità di giocare su più dispositivi contemporaneamente, si può decidere di implementare un sistema di autenticazione. Questo permetterebbe di registrare una serie di informazioni e statistiche sui giocatori, inoltre si potrebbe consentire ai giocatori di condividere le mappe create con gli altri utenti.

Con l'implementazione di tale evolutiva sorgerebbero una serie di problematiche in più che andrebbero gestite, ad esempio:

- il collegamento di più giocatori ad una stessa partita: si potrebbe gestire tramite la creazione di un link o un codice condivisibile tramite il quale i giocatori potrebbero connettersi alla sessione di gioco desiderata; inoltre potrebbe essere fornita la possibilità di partecipare a partite pubbliche.
- la gestione di un'interruzione improvvisa della connessione da parte di uno dei giocatori nel mezzo della partita (volontaria o dovuta a ragioni di rete).
- l'assegnazione di un tempo massimo per effettuare una mossa o per giocare un turno.
- l'implementazione di un sistema di censura automatica su tutto ciò che viene inserito dagli utenti (dato che potrebbe avere visibilità pubblica), o quantomeno un ruolo di moderatore e un meccanismo di segnalazione.