Relazione
“Farming Simulator”

Linda Fabbri,
Federico Raffoni,
Simone Rega,
Giacomo Montali

20 Aprile 2021

Indice

Capitolo 1

Analisi

1.1 Requisiti

Il gruppo si pone come obiettivo quello di realizzare "Farming Simulator", un software a scopo ludico basato sul videogioco "Hay Day" [1] sviluppato da Supercell. Il progetto mira alla realizzazione di un ambiente agricolo in 2D, in cui il giocatore interagisce con le colture e gestisce le proprie finanze. Lo scopo del gioco è arrivare all'apice della ricchezza, sbloccando tutti gli appezzamenti di terra e coltivando la maggior varietà di semi possibili.

Requisiti funzionali

- Questa applicazione deve permettere al giocatore di muoversi all'interno della mappa di gioco evitando di salire su zone di mappa non calpestabili.
- Il personaggio ha la possibilità di coltivare piantagioni di grano, patate, pomodori, carote, alberi di ciliege e altri frutti.
- Nella mappa deve comparire una zona per l'allevamento degli animali, grazie ai quali si possono raccogliere latte, uova e altri derivati.
- Farming Simulator mette a disposizione un negozio in cui è possibile vedere il proprio inventario, vendere i prodotti agricoli raccolti in cambio di monete virtuali e comprare nuovi semi da coltivare.
- Per aumentare la difficoltà di gioco vengono messi a disposizione appezzamenti di terreno coltivabili inizialmente bloccati, che si possono sbloccare tramite l'utilizzo delle monete virtuali.

[1] `https://supercell.com/en/games/hayday/`

Requisiti non funzionali

- Farming Simulator deve poter essere eseguito su dispositivi con schermi dotati di risoluzioni differenti.
- Il programma deve poter essere eseguito sia su Windows sia su sistemi operativi Unix

1.2 Analisi e modello del dominio

In Farming Simulator il soggetto principale è il personaggio virtuale, il quale si muove all'interno di una mappa, composta da diversi tipologie di blocchi.

All'interno della mappa possiamo trovare blocchi coltivabili (Field Block), blocchi non calpestabili dal personaggio (Wall), blocchi estetici(Water), il quale unico compito è quello di abbellire la mappa, blocchi che compongono la stalla in cui possono muoversi gli animali(Stall), ed infine blocchi "locked"(Unlockable Block), ovvero che inizialmente sono bloccati e possono essere sbloccati durante la partita in cambio di monete virtuali.

Il personaggio (Player), grazie all'inventario (Inventory), è in grado di mantenere risorse al suo interno. Tra le risorse collezionabili possiamo trovare: il cibo (Food) ottenuto dalla raccolta di coltivazioni o di animali oppure i semi (Seed). Questi ultimi possono essere comprati nel negozio (Shop) del videogioco, dentro il quale è anche possibile vendere il cibo che si è raccolto.

Il personaggio può interaggire con blocchi e animali (Animals), in particolare può piantare e raccogliere i semi all'interno dei blocchi coltivabili (FieldBlock), sbloccare i blocchi non ancora accessibili (UnlockableBlock) e raccogliere i prodotti animali.

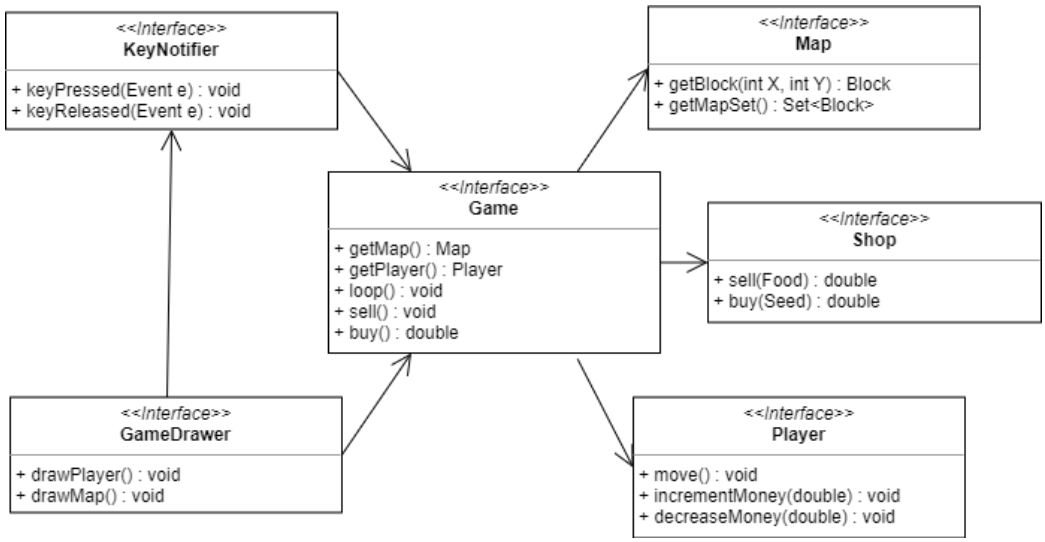

Figura 1.1: Schema UML dell'analisi del problema, con rappresentate le entità principali ed i rapporti fra loro

Capitolo 2

Design

2.1 Architettura

L'architettura di Farming Simulator è basata su un pattern MVC. In particolare è stata sviluppata l'interfaccia Game che funge da Controller e si posiziona tra View e Model e gestisce eventuali interazioni tra componenti. All'arrivo di un input, game, si occupa di ordinare al Model il cambio di stato.

Gli input da tastiera provenienti dall'utente passano attraverso la classe KeyNotifier, la quale si occupa di informare il game dell'avvenuto input.

La parte di View è formata da diversi GameDrawer, i quali si occupano di interrogare il game e mostrare a video il gioco.

Quando viene catturato un input da tastiera, il GameDrawer avvisa il keyNotifier dell'avvenuta pressione, così facendo il KeyNotifier è un observable per GameDrawer. In questo modo il GameDrawer non si deve occupare della gestione degli input, spostando il carico di lavoro al KeyNotifier; questa astrazione ci permette di non dipendere dall'implementazione e poter cambiare il binding[1] o addirittura dispositivo di input (ad esempio un Joystick).

In Figura 2.1 è esemplificato il diagramma UML che descrive l'architettura MVC del progetto, mostrando le classi principali.

[1]Binding = associazione di un tasto ad un determinato movimento e/o interazione visiva in gioco (ad esempio [WASD] per spostarsi nella mappa)

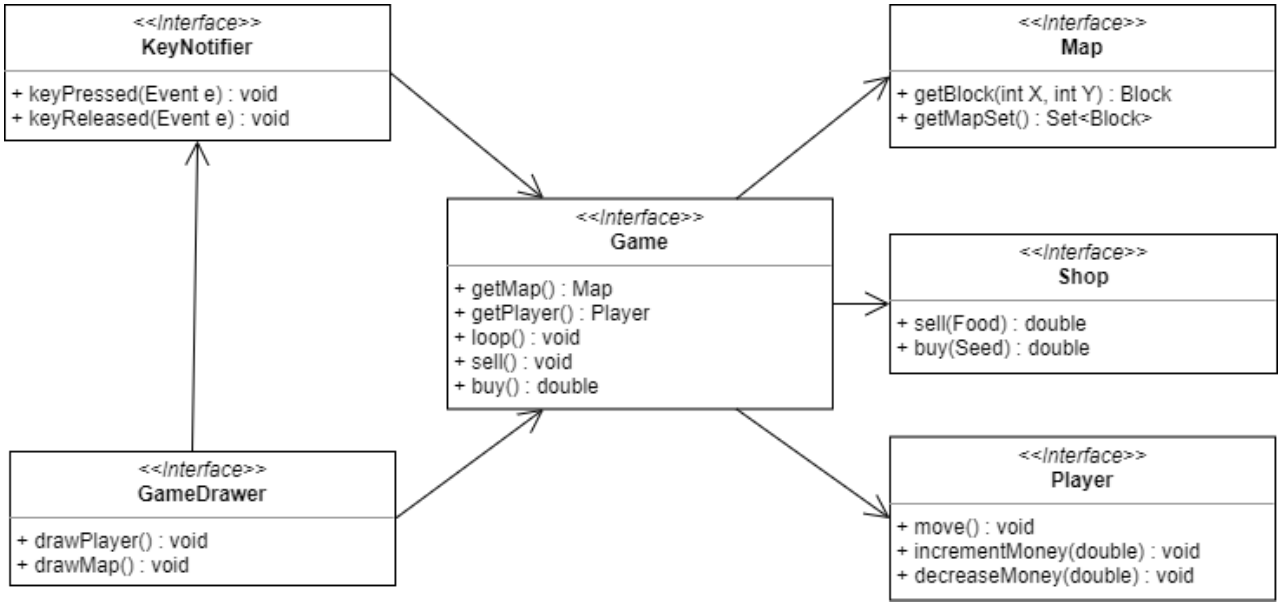

Figura 2.1: Schema UML architetturale di Farming Simulator, MVC.

2.2 Design dettagliato

In questa sezione vengono mostrati alcuni elementi di design con maggior dettaglio. Ogni membro del gruppo presenta gli elementi di design più significativi da lui prodotti, al fine di realizzare l'applicazione, predisponendola a future espansioni.

2.2.1 Simone Rega

Io nel progetto mi sono occupato della parte riguardante l'aggiornamento dinamico della GUI nella sezione dello Shop, in particolare mi preoccupo di aggiornare il JPanel dello shop e la visualizzazione dell'inventario ogni qualvolta che si interagisce con i JButton di compra/vendita, la JComboBox e il JSpinner per comprare semi.

Ho ritenuto utile usare un pattern Observer in modo che alla ricevuta notifica di un osservatore, quindi un Evento sui componenti grafici descritti prima, gli altri osservatori vengono notificati per poi essere aggiornati.

È stato deciso insieme a *Federico Raffoni* di creare le interfacce **Observer** e **Observable** di tipo generico, in modo da poter implementare le classi in più possibili situazioni eterogenee. Il ruolo di Observable è ricoperto dalla classe

ObservableShopGUI mentre il ruolo di Observer è ricoperto da ObserverShop (classe innestata all'interno di ShopDrawer.java).

In Figura 2.2 è esemplificato il diagramma UML che descrive l'architettura del pattern Observer usato da Simone Rega, mostrando le classi principali.

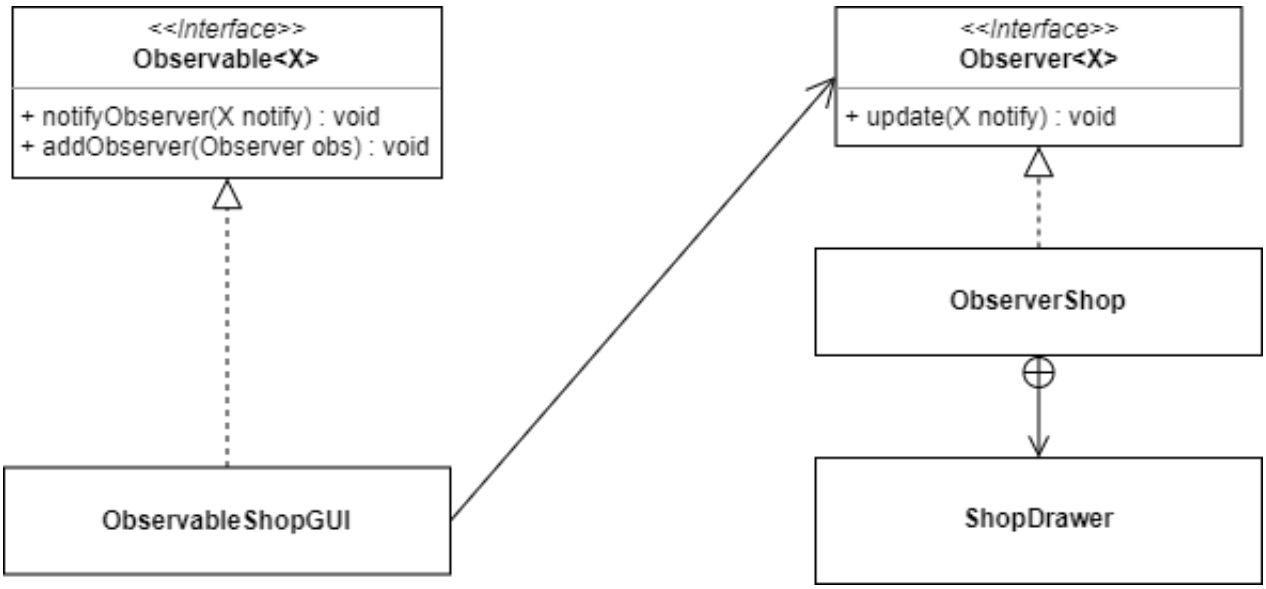

Figura 2.2: Schema UML architetturale del pattern Observer (Simone Rega).

2.2.2 Federico Raffoni

In questa sezione ci occuperemo principalmente del caricamento delle risorse grafiche e non solo. In particolare è stata creata la classe Resources la quale si occupa di caricare e distribuire tutte le risorse.

È stato usato il pattern SINGLETON per rendere unico nell'applicazione l'accesso alla classe Resources. In questo modo si evita che diversi GameDrawer carichino più volte le risorse sprecando memoria.

Le risorse verranno caricate solo alla prima chiamata, mentre le chiamate successive riceveranno una copia statica della classe risparmiando tempo e risorse.

In Figura 2.3 è esemplificato il diagramma UML che descrive l'architettura singleton, mostrando le classi principali.

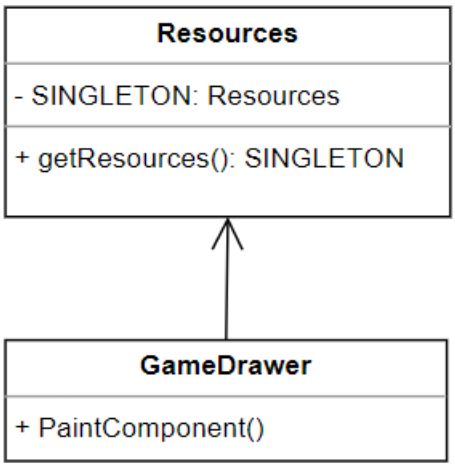

Figura 2.3: Schema UML architetturale del pattern Singleton (Federico Raffoni).

Per quanto riguarda il caricamento di una partita salvata, è stato utilizzato il pattern Observer per consentire all'Engine di rimanere in attesa mentre la classe GamePreloader si occupa di chiedere alll'utente se continuare una partita salvata oppure iniziare una partita da zero.

In particolare Engine è Observer, mentre la classe GamePreloader è Observable. Sono state realizzate inoltre le interfacce Observer e Observable generiche, in modo da poter essere riutilizzate anche in altri campi.

In Figura 2.4 è esemplificato il diagramma UML che descrive l'architettura Observer, mostrando le classi principali.

2.2.3 Linda Fabbri

La parte di progetto riguardo al design di cui mi sono occupata personalmente è quella riguardante i blocchi.

Questa parte è composta da una classe enum (BlockType) per la classificazione delle tipologie dei vari blocchi, in base alla tipologia ho avuto la necessità di implementare metodi diversi, dunque ho creato tre interfacce: **Block** per tutti i generici blocchi, **FieldBlock** (estensione di Block) a cui ho

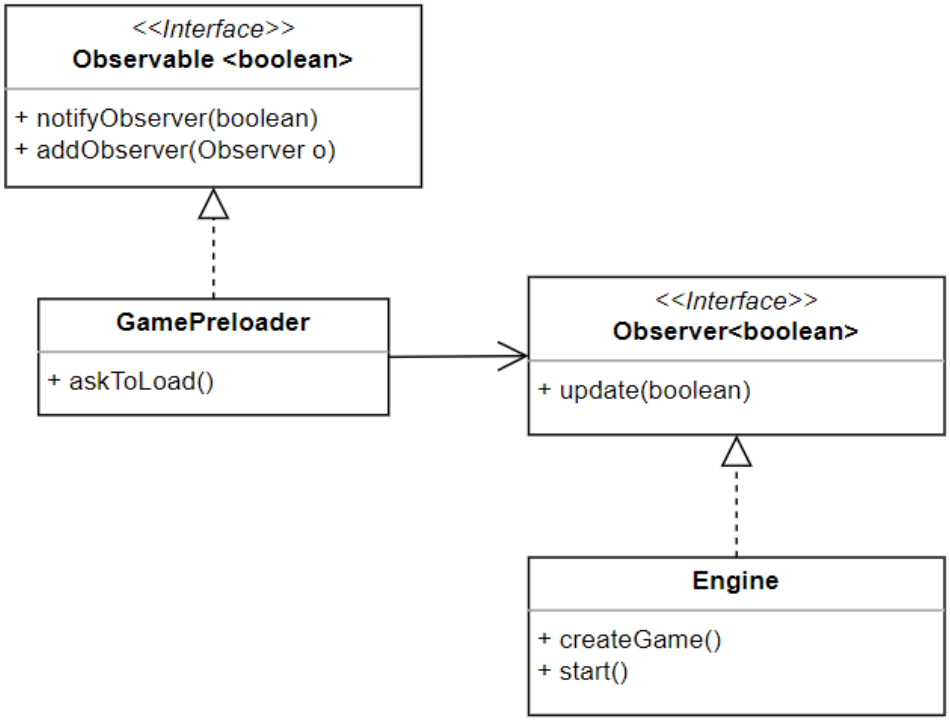

Figura 2.4: Schema UML architetturale del pattern Observer (Federico Raffoni).

aggiunto i metodi per i blocchi coltivabili (FIELD in BlockType), **UnlockableBlock**(estensione di FieldBlock) a cui ho aggiunto i metodi per i blocchi coltivabili bloccati (LOCKED in BlockType).

L'implementazione di queste interfacce si trova in FactoryBlock come classi innestate. Inoltre, per ogni tipologia di blocco si ha un differente costruttore al fine di determinare i vari campi.

Ho scelto l'utilizzo del pattern *"Factory method"* allo scopo di creare e ritornare blocchi in base al loro tipo mediante l'uso di costruttori differenti, delegando la responsabilità della scelta della creazione di ogni oggetto solo alla classe che implementa tale pattern.

In Figura 2.5 è esemplificato il diagramma UML che descrive l'architettura Factory Method, mostrando le classi principali.

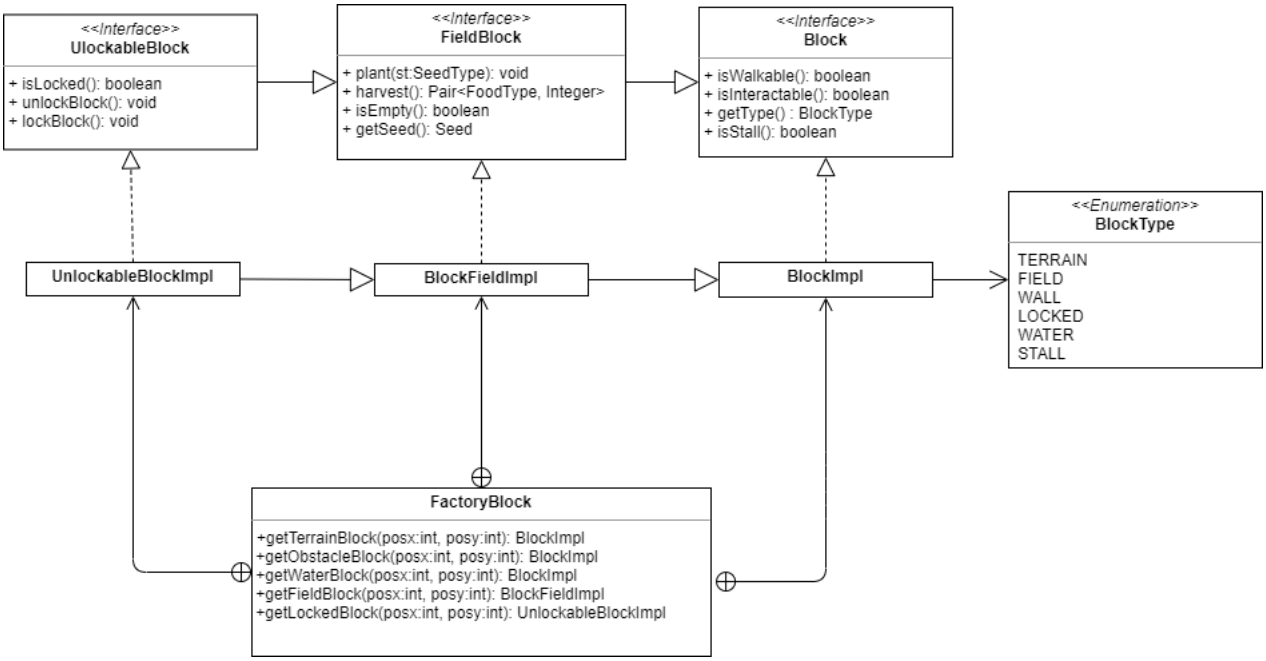

Figura 2.5: Schema UML architetturale del pattern Factory Method (Linda Fabbri).

2.2.4 Giacomo Montali

I principali punti dei quali mi sono occupato in questo progetto sono le varie entità presenti in essa, cioè il Player e i vari animali.

In particolare per questi ultimi ho utilizzato il pattern *"Simple Factory"* nella classe **FactoryAnimal**, con l'obiettivo di delegare la creazione degli animali a quest'ultima, separandola così dalla classe **AnimalImpl** per rendere più facile l'aggiunta di eventuali nuovi classi che implementano l'interfaccia Animal.

La suddivisione degli animali è stata effetuata grazie ad un Enum.

In Figura 2.6 è esemplificato il diagramma UML che descrive l'architettura del Pattern Simple Factory, mostrando le classi principali.

Inoltre ho creato la classe abstract **Entity** per renderla classe padre delle classi **PlayerImpl** e **AnimalImpl**, per evitare il riutilizzo di codice derivato dai metodi in comune come tutti i metodi che riguardano il movimento e la posizione sulla mappa.

In Figura 2.7 è esemplificato il diagramma UML che descrive l'architettura delle classe Abstract Entity, mostrando le classi principali.

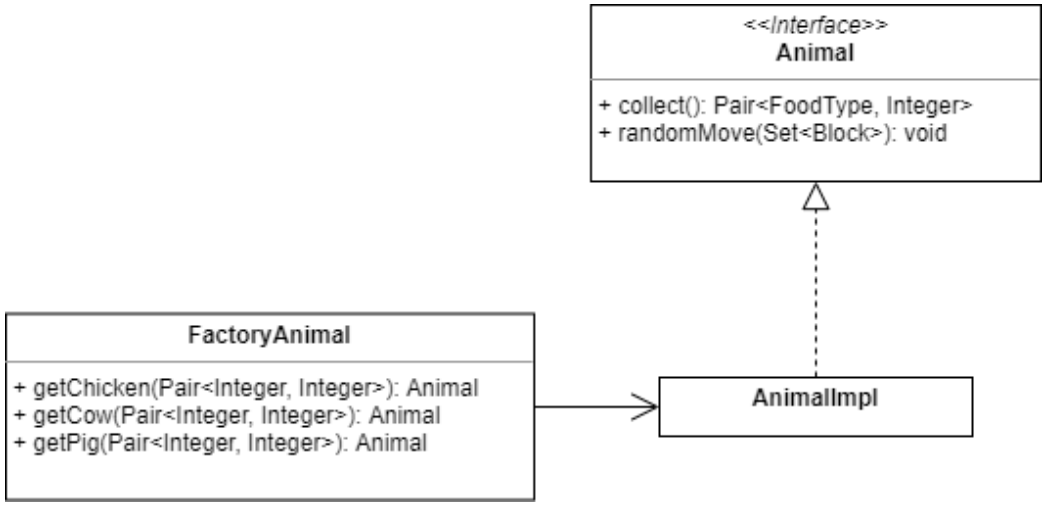

Figura 2.6: Schema UML architetturale del pattern Simple Factory (Giacomo Montali).

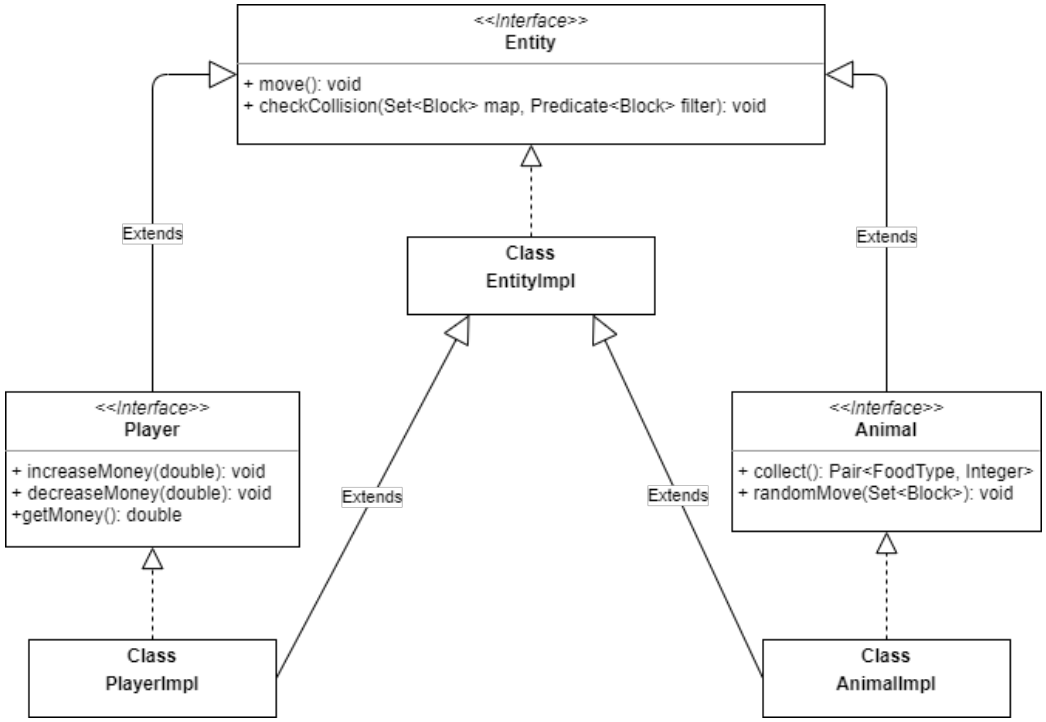

Figura 2.7: Schema UML architetturale della classe Entity (Giacomo Montali).

Capitolo 3

Sviluppo

3.1 Testing automatizzato

In questo progetto abbiamo deciso di testare prevalentemente la parte di Model poiché riteniamo sia la più propensa ad avere bug; non sono però mancati test riguardanti la View e salvataggio del gioco. Per svolgere test completamente automatizzati sul progetto è stato usato JUnit5.

Le classi che sono state testate sono:

- **testPlayerMovement:** con questo test muovo il personaggio simulando la pressione dei tasti
- **testPlayerCollision:** con questo test controllo che non avvenga il movimento su blocchi o in direzioni in cui mi è permesso
- **testPlayerPlant:** con questo test simulo l'interazione del personaggio con un blocco di tipo FieldBlock, piantando un seme e poi controllando che quel blocco non sia vuoto
- **testUnlockBlock:** con questo test simulo l'interazione con un blocco di tipo UnlockableBlock
- **testUnlockBlockWithoutMoney:** con questo test simulo l'interazione con un blocco di tipo UnlockableBlock senza soldi
- **testPlayerAnimalInteraction:** con questo test mi sposto nella zona "Stalla" della mappa e aspetto che un Animal qualunque sia pronto per poi poter raccogliere il suo *Food* simulando l'interazione tra *Player* e *Animal*
- **testBuy:** con questo test provo a comprare un *Seed* dallo *Shop*, controllando poi che sia effettivamente presente nell'*Inventory*
- **testBuyWithoutMoney:** questo test è simile al precedente, ma ho zero money, verificando che non avvenga l'acquisto

- **testSell:** in questo test simulo una normale interazione con un FieldBlock, ricavandone poi il raccolto che proverò a vendere, verificandone l'esistenza nell'inventario e l'ottenimento dei soldi ricavati dalla vendita
- **testSaver:** con questo test simulo il salvataggio del gioco su file con relativa apertura da esso in una nuova esecuzione del gioco, verificando che i nuovi e vecchi *Game* e *Player* siano gli stessi e abbiano le stesse caratteristiche
- **testSaverPlayer:** questo test è simile al precedente, differisce semplicemente per il fatto che il Player viene mosso nella nuova istanza di gioco, così da far risultare falso il fatto che l'istanza salvata e l'istanza nuova creata siano uguali

3.2 Metodologia di lavoro

• Metodologia riguardante il progetto in generale
Lo sviluppo del progetto è stato preceduto da una fase preliminare di analisi e progettazione, durante la quale abbiamo lavorato tutti insieme per definire prima le basi dell'applicazione, poi successivamente specificare i vari dettagli implementativi tramite UML sulla piattaforma Miro.

In queste fasi la collaborazione e il confronto sono state molto intense.

La suddivisioni delle parti di Farming Simulator è avvenuta in modo abbastanza equo assegnando a ciascuno una parte significativa e importante del progetto, anche in base alle nostre preferenze.

• Distributed Version Control System
Come DVCS è stato usato Git, creando un repository su GitHub. Abbiamo lavorato sullo scheletro del progetto su un unico branch main, per poi creare diversi branch per implementare e/o mettere a punto la realizzazione di feature nuove o più sostanziose.

3.2.1 Federico Raffoni

Sviluppo del motore grafico, in particolare l'Engine e la classe WindowManager, e le varie interazioni tra questi due elementi. Ideazione e creazione della classe abstract GameDrawer che ha portato alla realizzazione delle varie scene di gioco. Sviluppo della classe KeyNotifier che permette l'interazione tra la view e il controller (Game).

Implementazione della classe Resources.

Ho inoltre sviluppato la classe che permette il salvataggio e il caricamento della partita (GameSaver) .

Ho inoltre contribuito all'implementazione della classe Game e allo sviluppo della parte grafica nella classe MainScreenDrawer

Classi sviluppate singolarmente:

- KeyNotifier
- Engine
- GameSaver
- GameDrawer
- GamePreloader
- Resources
- WindowManager
- MusicPlayer

3.2.2 Linda Fabbri

Si è occupata principalmente della parte riguardante la logica di gioco legata alle interazioni fra le entità e la parte che si occupa della creazione dei blocchi che compongono la mappa di gioco, in particolare le classi Interaction, InteractionImpl nel package engine e tutte le classi del package block.

Legato alle interazioni ha implementato i metodi riguardanti la compravendita di prodotti che si trovano nella classe ShopImpl, in particolare i metodi sellAll e buy.

In collaborazione con Simone Rega si è implementata la classe ShopDrawer per la parte che si occupa della grafica dei pannelli legati alla compravendita di semi e prodotti.

Ha collaborato all'implementazione della classe Game, in particolare per chiamare i metodi implementati in InteractionImpl.

In collaborazione con Simone Rega e Federico Raffoni ha eseguito il refactoring per correggere errori evidenziati dal checkstyle.

•Problemi

In fase di progettazione nella classe Inventory si era pensato di creare solo i metodi gotFoods e gotSeeds, per verificare l'esistenza di un certo numero di Food o Seed nell'inventario, tuttavia per i metodi delle classi che sono state citate precedentemente, si è trovata la necessità di un metodo che ritornasse la mappa di tutte le tipologie di Food e Seed.

3.2.3 Simone Rega

Sviluppo delle classi all'interno del package Item che riguardano principalmente la realizzazione degli elementi Seed e Food del gioco.

Sviluppo della parte di grafica riguardante lo shop (ShopDrawer); le informazioni del gioco (InfoDrawer e implementazione di vari JPanelHUD nel MainScreenDrawer).

Ideazione e creazione delle classi nel package gameShop riguardante la logica base dello Shop

Ideazione e sviluppo della classe test in JUnit5 a cui hanno dato una mano allo sviluppo Linda Fabbri e Federico Raffoni.

Classi sviluppate singolarmente:

- FoodType
- SeedType
- Seed
- SeedImpl
- ItemConstants
- UnlockableBlock
- InfoDrawer
- QuitDrawer
- JPanelINFO
- JPanelHUD
- ObservableShopGUI

Classi sviluppate insieme ad altre persone

- MainScreenDrawer : implementazione di generateHUD per generare la barra di gioco superiore
- ShopDrawer : quasi tutta la parte grafica
- FarmingSimulatorTestClass
- Observer
- Observable
- Shop
- ShopImpl

3.2.4 Giacomo Montali

Sviluppo del package entity, che riguarda principalmente Player, Animal e le relative classi ad esse connesse.

Sviluppo della mappa di gioco dentro la quale interagiranno le entità. Caricamento della mappa di gioco da file.

3.3 Note di sviluppo

3.3.1 Federico Raffoni

- **Lambda:** ho usato le lambda all'interno del metodo loop della classe GameImpl per muovere tutte le entità e per impostare i blocchi camminabili dal player
- **Reflection:** All'interno della classe GameSaver

Ho utilizzato la libreria Gson per serializzare e deserializzare i componenti di gioco per permettere il salvataggio e il caricamento della partita. Nell'implementazione della classe GameSaver è stato necessario sviluppare un serializzatore di interfacce il quale è stato scritto prendendo spunto da questo thread (source: https://stackoverflow.com/questions/62358189/gson-stackoverflowerror-during-serialization)

Per la creazione della classe MusicPlayer, non essendo pratico delle funzioni di libreria di javax.sound, ho fatto riferimento a snippet presenti sul web

3.3.2 Simone Rega

Per quanto riguarda le Feature Avanzate del linguaggio:

- **Lambda :** utilizzate nella classe FarmingSimulatorTestClass per i test sul Player e sui Blocchi
- **IntStream:** utilizzate nella classe FarmingSimulatorTestClass .

Per quanto riguarda le librerie utilizzate

- **Java AWT**
- **Java Swing**

So che queste due librerie sono state introdotte a lezione spiegandone i principali elementi di maggior rilevanza ma ci tengo comunque a precisare lo studio e l'approfondimento per poter realizzare elementi di grafica più complessi o che purtroppo non abbiamo avuto modo di trattate a lezione .

3.3.3 Linda Fabbri

- **Optional:** nei blocchi per il loro contenuto
-

3.3.4 Giacomo Montali

Capitolo 4

Commenti finali

In quest'ultimo capitolo si tirano le somme del lavoro svolto e si delineano eventuali sviluppi futuri.

Nessuna delle informazioni incluse in questo capitolo verrà utilizzata per formulare la valutazione finale, a meno che non sia assente o manchino delle sezioni obbligatorie. Al fine di evitare pregiudizi involontari, l'intero capitolo verrà letto dai docenti solo dopo aver formulato la valutazione.

4.1 Autovalutazione e lavori futuri

4.1.1 Simone Rega

Sin dall'inizio sono stato partecipe e disponibile per la progettazione e l'analisi del progetto.

Purtroppo non ho potuto lavorare troppo sulla logica di gioco (ho semplicemente svolto la parte riguardante la logica che può avere un seme quando piantato)

Sono abbastanza contento del risultato finale riguardante il lavoro da me svolto e della modesta qualità del mio codice, che è stata equilibrata e trasparente, permettendone un corretto e utile uso.

Devo dire che sono leggermente dispiaciuto perché avrei voluto essere più partecipe o comunque dedicarmi di più allo sviluppo del programma. Avrei potuto migliorare l'ottimizzazione del mio codice e le sue funzionalità, o magari osare di più con elementi di programmazioni più avanzati introducendo più Lambda e Stream ad esempio.

Sono contento del Team con cui ho avuto modo di lavorare a questo progetto. La comunicazione è stata di fondamentale importanza.

4.1.2 Federico Raffoni

4.1.3 Linda Fabbri

4.1.4 Giacomo Montali

4.2 Difficoltà incontrate e commenti per i docenti

4.2.1 Simone Rega

buon corso, oop uni ¿ iti gradle spiegata meglio

Purtroppo il mio percorso di sviluppo non è stato lineare nel tempo; nonostante fosse stimolante per me avere per la prima volta un progetto di programmazione così bello e complesso da portare avanti, questo non è stato sempre possibile poiché non sempre sono riuscito a trovare il tempo per dedicarmi ad esso dopo le lezioni o lo studio. È effettivamente il primo progetto di sviluppo software da zero a cui ho partecipato e mi ha fatto crescere dal punto di vista organizzativo, rendendomi cosciente di ciò che mi aspetta una volta uscito dall'università.

4.2.2 Federico Raffoni

4.2.3 Linda Fabbri

4.2.4 Giacomo Montali

Appendice A

Guida utente

Capitolo in cui si spiega come utilizzare il software. Nel caso in cui il suo uso sia del tutto banale, tale capitolo può essere omesso. A tal riguardo, si fa presente agli studenti che i docenti non hanno mai utilizzato il software prima, per cui aspetti che sembrano del tutto banali a chi ha sviluppato l'applicazione possono non esserlo per chi la usa per la prima volta. Se, ad esempio, per cominciare una partita con un videogioco è necessario premere la barra spaziatrice, o il tasto "P", è necessario che gli studenti lo segnalino.

Elementi positivi

- Si istruisce in modo semplice l'utente sull'uso dell'applicazione, eventualmente facendo uso di schermate e descrizioni.

Elementi negativi

- Si descrivono in modo eccessivamente minuzioso tutte le caratteristiche, anche minori, del software in oggetto.
- Manca una descrizione che consenta ad un utente qualunque di utilizzare almeno le funzionalità primarie dell'applicativo.

Appendice B

Esercitazioni di laboratorio

In questo capitolo ciascuno studente elenca gli esercizi di laboratorio che ha svolto (se ne ha svolti), elencando i permalink dei post sul forum dove è avvenuta la consegna.

Esempio

B.0.1 Paolino Paperino

- Laboratorio 04: `https://virtuale.unibo.it/mod/forum/discuss.php?d=12345#p123456`
- Laboratorio 06: `https://virtuale.unibo.it/mod/forum/discuss.php?d=22222#p222222`
- Laboratorio 09: `https://virtuale.unibo.it/mod/forum/discuss.php?d=99999#p999999`

B.0.2 Paperon De Paperoni

- Laboratorio 04: `https://virtuale.unibo.it/mod/forum/discuss.php?d=12345#p123456`
- Laboratorio 05: `https://virtuale.unibo.it/mod/forum/discuss.php?d=22222#p222222`
- Laboratorio 06: `https://virtuale.unibo.it/mod/forum/discuss.php?d=99999#p999999`
- Laboratorio 07: `https://virtuale.unibo.it/mod/forum/discuss.php?d=22222#p222222`

- Laboratorio 08: https://virtuale.unibo.it/mod/forum/discuss.php?d=99999#p999999
- Laboratorio 09: https://virtuale.unibo.it/mod/forum/discuss.php?d=22222#p222222
- Laboratorio 10: https://virtuale.unibo.it/mod/forum/discuss.php?d=99999#p999999
- Laboratorio 11: https://virtuale.unibo.it/mod/forum/discuss.php?d=22222#p222222

Bibliografia