Programmazione ad Oggetti

CityBuild

Buldrini Matteo, Ciavatti Michele, Dagos Jhon Anthony, Flamigni Mattia

January 2023

Contents

1 Analisi

1.1 Requisiti

Il software *CityBuild* mira alla realizzazione di un videogioco gestionale ove il giocatore pu`o costruire la propria cittadina scegliendo fra una variet`a di edifici. La partita comincia in una mappa vuota, fornendo al giocatore dei materiali iniziali per poter costruire i primi edifici. Creando edifici, la popolazione della citt`a aumenter`a e, di conseguenza, il giocatore dovr`a espandere ulteriormente la citt`a per soddisfare le richieste degli abitanti. Ogni edificio pu`o essere svilup pato ulteriormente. Tutti gli edifici necessitano di materiali appositi per il loro sviluppo.

I principali edifici sono:

- Case. Per fare in modo che ogni casa sia abitabile, si necessita energia, fornita dalle centrali energetiche, e acqua fornita dai serbatoi/filtratori.

  Sviluppare una casa significa aumentare le persone che possono vivere al suo interno. La costruzione di una casa, come anche il suo miglioramento, necessita di materiali appositi.

- Strutture di produzione semplici. Queste strutture permettono la

produzione di materiali, indispensabili per la costruzione di edifici e per il benestare dei cittadini.

- Strutture di produzione avanzate. Varianti avanzate degli edifici base disponibili, che offrono maggiori risorse (o ospitano pi`u cittadini) ad un costo maggiorato per la relativa costruzione.

Il gioco prevede un men`u di commercio per acquistare o vendere materiali. Il gioco non prevede un game over: tuttavia, il giocatore pu`o vedersi ridurre la popolazione della citt`a a causa del mancato approvvigionamento di beni.

Requisiti funzionali:

- All'avvio del gioco, verr`a mostrato il men`u principale per iniziare una partita oppure caricare la partita precedente.
- Il giocatore avr`a a disposizione un terreno in cui poter inserire gli edifici, scegliendo questi ultimi da appositi men`u.
- Il giocatore `e continuamente notificato sullo stato della citt`a, in riferimento al numero di cittadini che la abitano e alle risorse che possiede.
- Il gioco continua anche ad applicazione spenta, per cui la popolazione pu`o crescere o diminuire anche mentre il giocatore non `e presente.

Requisiti non funzionali:

- Il software deve poter funzionare in maniera accettabile sui vari tipi di sistemi operativi.

1.2 Analisi e modello del dominio

Il gioco si ambienta in una mappa, inizialmente vuota. Al giocatore vengono forniti dei materiali di partenza per poter costruire i primi edifici. Ogni edificio richiede dei materiali specifici per essere costruito e sviluppato ulteriormente, per cui il giocatore deve possedere e spendere tali materiali per costruire e sviluppare l'edificio. Con la costruzione dei primi edifici, arriveranno le prime richieste da futuri cittadini per vivere nella citt`a: qualora ci fossero sufficienti case, i cittadini entreranno automaticamente nella citt`a per viverci all'interno. Al contrario, potrebbe succedere che i cittadini abbandonano la citt`a a causa della mancanza di risorse e/o di abitazioni.

Figure 1: Diagramma UML con le principali entit`a, le loro funzionalit`a e le loro relazioni.

2 Design

2.1 Architettura

L'architettura del software segue il pattern MVC con le classi *GameScreen* quale *view*, *Controller* quale *controller*, e infine *City* quale *model*. La libreria grafica scelta per il progetto `e *LibGDX*, la quale costringe a racchiudere nella classe di *view* elementi che normalmente sarebbero di appannaggio al *controller*, quali il *game loop* e pattern *observer* del *controller* sulla *view* per reagire allo stato di gioco (cosa che viene fatta dalla classe *GameProcessor*

). Di conseguenza, il *controller* risulta per lo pi`u una delega al *model*, e questo si pu`o notare dall'UML (figura 2). Per il procedimento del gioco, si `e pensato che ogni n secondi, la *view* avrebbe chiamato il metodo *doCycle()* del *controller*, il quale si occupa di aggiornare lo stato di gioco (rendimento degli edifici costruiti, spesa delle risorse richieste dai cittadini ed eventuale arrivo/abbandono di cittadini).

2.2 Design dettagliato

Michele Ciavatti

- *Lettura di file relativi al modello economico*.

  Problema: disporre dei dati grezzi relativi al modello economico nel modo pi`u efficiente e accessibile possibile.

  Soluzione: ho pensato di utilizzare dei file YAML e di creare delle classi di lettura dei medesimi. Nello specifico ho creato una interfaccia *Economy FileReader*, la cui implementazione reperisce i file YAML nella directory *resources* del progetto e li legge, passando per una classe di appoggio che ho denominato *EconomyTables*. Per la lettura dei file, ho sfruttato le fun zionalit`a messe a disposizione dalla libreria snakeYAML nella sua ultima versione (1.33). La figura 3 mostra lo schema UML. I metodi di *Econo myFileReader* restituiscono una lista di *Map*: l'idea `e che la prima *Map* si riferisce alla rendita dell'edificio, la seconda al costo di costruzione, e la terza al costo di sviluppo.

- *Edifici di produzione risorse.*

  Problema: modellare tutti gli edifici di produzione nella maniera pi`u sintetica e riutilizzabile possibile.

  Soluzione: ho creato una interfaccia *ProductionBuilding* che si occupa di restituire le informazioni essenziali dell'edificio su richiesta e di gestire il processo di upgrade dell'edificio. Seguendo il design pattern *Factory*, ho deciso di delegare la creazione di istanze di *ProductionBuilding* ad un'altra classe, ovvero *BuildingFactory*, rendendo impossibile al client conoscere in alcun modo le implementazioni utilizzate per *ProductionBuilding* e au mentando la manuntenibilit`a del codice. La figura 4 mostra l'UML di questo design.

Figure 2: Diagramma UML dell’architettura

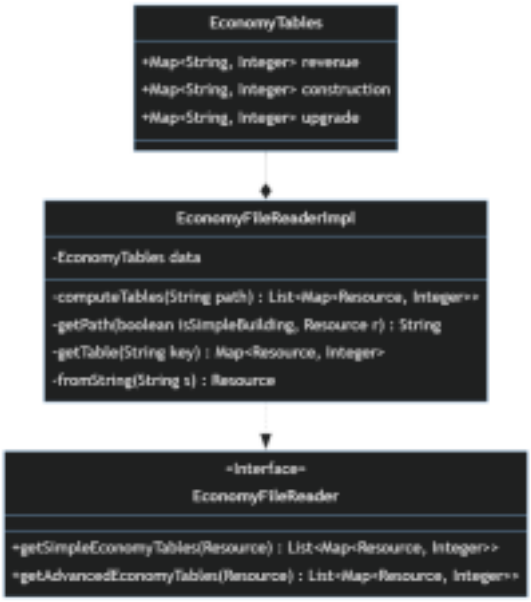

Figure 3: Diagramma UML del design *EconomyFileReader*. La classe *Econo myTables* possiede solo getter e setter per ogni campo; per motivi di sintesi, ho semplicemente riportato i campi nello schema UML (che sono in realt`a privati, chiaramente).

Figure 4: Diagramma UML del design *ProductionBuilding* e relativa *Factory*.

• *Men`u di accesso.*

Problema: creare il men`u principale del gioco, in grado di lanciare una partita nuova o caricarne una vecchia.

Soluzione: ho reso la classe *CityBuild* una estensione di *Game*, una classe di LibGDX, al fine di poter utilizzare metodi specifici in grado di switchare fra una schermata e la successiva. A questo punto, ho creato la classe *MainMenu* che `e la prima schermata di *CityBuild* e prende una istanza di quest'ultima per poterle cambiare schermata all'avvio di una nuova partita o al caricamento di una precedente. La figura 5 mostra l'UML di questo design.

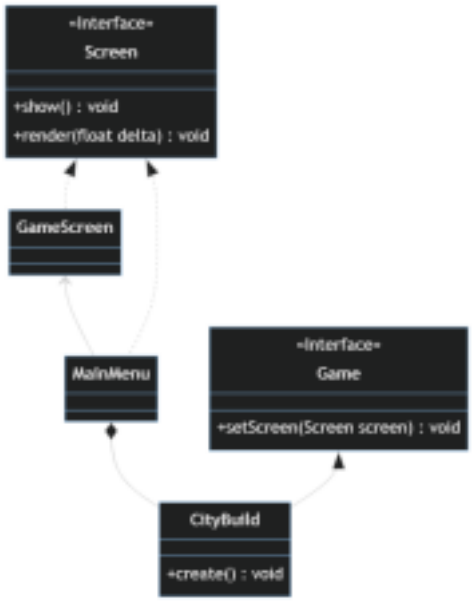

Figure 5: Diagramma UML del design del men`u principale di gioco. In realt`a, *GameScreen* estende *ScreenAdapter* ma tale classe `e solo una implementazione di base di *Screen* che consente di fare l'override solo sui metodi di *Screen* che ci interessano.

Mattia Flamigni

- menu di scelta degli edifici

  problema: creare un menu in modo che il giocatore possa scegliere l'oggetto da creare

  soluzione: si è optato per un menu fisso nella schermata di gioco, invece di un meccanismo di apertura finestra. E' stata creata una lista con i nomi degli oggetti da visualizzare con alcuni metodi.

  roundButtonList è utilizzato per scorrere attraverso una lista di image button. accetta un parametro (1 o -1) in base alla direzione richiesta dal

giocatore.

selectButton invece è utilizzato per selezionare un pulsante specifico all'interno di una tabella e “visualizzare” le informazioni relative all’immagine selezionata.

- funzionalita di incendio
  problema: creare un imprevisto al giocatore. in questo caso la citta puo prendere fuoco andando a decrementare risorse

  soluzione: è stata creata un interfaccia nella quale l’implementazione calcola intensita che influisce poi sul calcolo del costo. nel file GameScreen viene chiamato il metodo per incendiare la citta.

- funzionalita di aggiornamento in background:
  se il giocatore decide di salvare la partita, quando il programma viene riaperto è come se la citta non si fosse mai fermata.

  problema: trovare un modo per fare in modo che venga eseguido un thread ad applicazione chiusa

  soluzione: documentandomi sul web ho trovato una funzionalita della liobreria che consente di far partire un thread appena prima della chiusura dell’applicazione. i valori vengono aggiornati random.

3 Sviluppo

3.1 Testing automatizzato

Abbiamo realizzato test automatici basati sulla suite di JUnit, localizzati nei package it.unibo.controller e it.unibo.model.

Sono stati implementati test relativi a *Player*.

3.2 Metodologia di lavoro

I lavori sono stati divisi come segue:

- Matteo Buldrini:
- Michele Ciavatti: lettura dei file YAML relativi al modello economico, progettazione e gestione degli edifici di produzione delle risorse. Gestione della logica relativa al giocatore. Progettazione del men`u principale. Pro gettazione del *Controller* e gran parte della *view*.
- Jhon Anthony Dagos:

• Mattia Flamigni:

Abbiamo utilizzato il DVCS Git lavorando su branch separati per ogni feature sviluppata: una volta realizzata la feature, veniva eseguito un merge sul branch *master*. Occasionalmente sono stati eseguiti dei merge fra branch-feature differ enti, e anche commit minori sul branch *master*.

3.3 Note di sviluppo

Michele Ciavatti

- *Utilizzo della libreria di terze parti snakeYAML:* https://github.com/MicheleCiavatti/OOP22-CityBuild/blob/ed827142143a128f969beeab58e175cb06ee1d66/core/src/main/java/it/unibo/controller/impl/EconomyFileReaderImpl.java#L49
- *Utilizzo della libreria di terze parti libGDX:* usata per tutta la resa grafica. Un esempio `e la classe all'url https://github.com/MicheleCiavatti/OOP22-CityBuild/blob/c2ce894150e9e4e0aa457acd35b41e6ce3618848/core/src/main/java/it/unibo/MainMenu.java#L17.
- *Utilizzo di lambda:* usate ripetutamente nel codice, un esempio si trova al link https://github.com/MicheleCiavatti/OOP22-CityBuild/blob/ed827142143a128f969beeab58e175cb06ee1d66/core/src/main/java/it/unibo/model/impl/ProductionBuildingImpl.java#L55
- *Utilizzo di stream:* usati pi`u volte nel codice. Un esempio al link https://github.com/MicheleCiavatti/OOP22-CityBuild/blob/ed827142143a128f969beeab58e175cb06ee1d66/core/src/main/java/it/unibo/model/impl/ProductionBuildingImpl.java#L74
- *Utilizzo di Optional:* nella classe che ora si chiama ScreenExample.

I file per lo stile dei tasti nel men`u iniziale sono stati presi dal link: https://github.com/czyzby/gdx-skins. Inoltre, il metodo *set ColorLabel()* della classe *GameScreen* `e stato preso dal seguente post su Stack Overflow: https://stackoverflow.com/questions/18166556/how-to-change-the-color-of-the-background-in-libgdx-labels.

4 Commenti finali

4.1 Autovalutezione e lavori futuri

Michele Ciavatti

Il codice da me creato `e sufficientemente manuntenibile grazie all'abbondante utilizzo di interfacce, del pattern *Factory* e file terzi per la gestione di dati grezzi relativi all'economia del gioco.

Sono per`o ben conscio di un punto molto debole della mia progettazione, ovvero l'utilizzo di una enumerazione per le risorse del software. Tale scelta

`e chiaramente molto comoda per i programmatori, che possono facilmente ac cedere alle funzionalit`a legate alle risorse e fare confronti veloci e precisi fra risorse differenti, ma ha anche il forte svantaggio di rendere il codice meno ma nuntenibile.

A parte questo, non vedo gravi pecche nel mio lavoro considerando che ho cercato di seguire tutte le best practices di cui sono a conoscenza.

Mattia Flamigni

Riconosco che il mio codice Java potrebbe non essereperfetto, tuttavia ho cercato di garantire che funzioni come richiesto. riconosco di non aver utilizzato tutti i pattern, come ad esempio factory. dall'altra parte, oltre all'utilizzo di interfacce, mi sono impegnato a migliorare il codice da me scritto con if ternari, lamda expression e impegno nel rispettare il pattern MVC, scelto in fase di progettazione.