Programmazione ad Oggetti
CityBuild

Ciavatti Michele, Dagos Jhon Anthony, Flamigni Mattia

Contents

1 Analisi

1.1 Requisiti

Il software *CityBuild* mira alla realizzazione di un videogioco gestionale ove il giocatore può costruire la propria cittadina scegliendo fra una varietà 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 città aumenterà e, di conseguenza, il giocatore dovrà espandere ulteriormente la città per soddisfare le richieste degli abitanti. Ogni edificio può essere sviluppato 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 più cittadini) ad un costo maggiorato per la relativa costruzione.

Il gioco prevede un menù di commercio per acquistare o vendere materiali. Il gioco non prevede un game over: tuttavia, il giocatore può vedersi ridurre la popolazione della città a causa del mancato approvvigionamento di beni.

Requisiti funzionali:

- All'avvio del gioco, verrà mostrato il menù principale per iniziare una partita oppure caricare la partita precedente.
- Il giocatore avrà a disposizione un terreno in cui poter inserire gli edifici, scegliendo questi ultimi da appositi menù.
- Il giocatore è continuamente notificato sullo stato della città, 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 può crescere o diminuire anche mentre il giocatore non è 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 città: qualora ci fossero sufficienti case, i cittadini entreranno automaticamente nella città per viverci all'interno. Al contrario, potrebbe succedere che i cittadini abbandonano la città a causa della mancanza di risorse e/o di abitazioni.

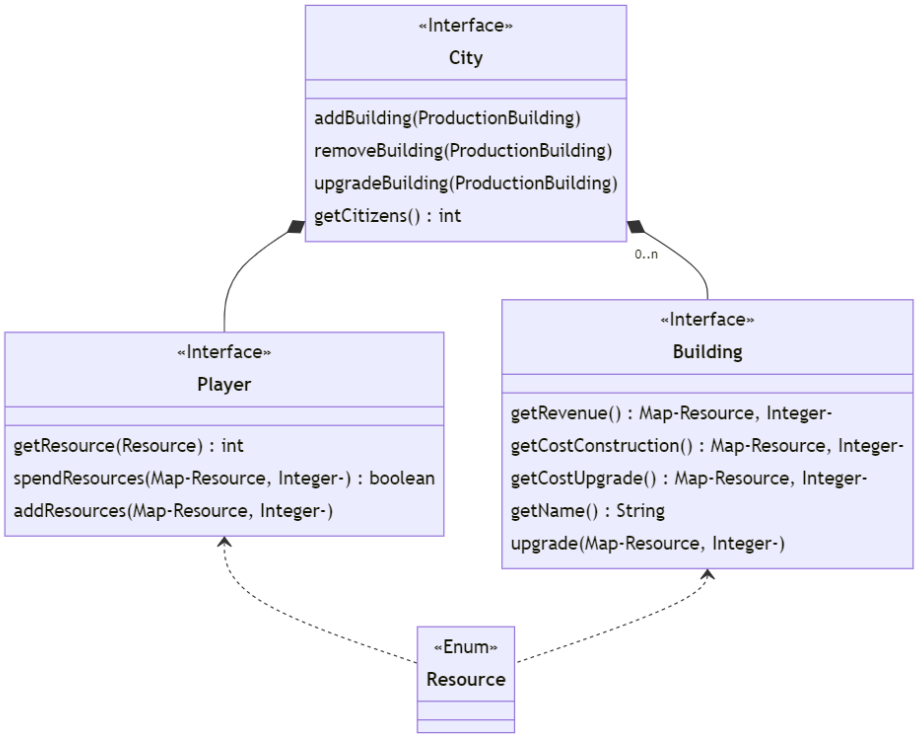


Figure 1: Diagramma UML con le principali entità, le loro funzionalità e le loro relazioni.

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 è *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 più una delega al *model*, e questo si può notare dall'UML (figura 2). Per il procedimento del gioco, si è 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 più 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 *EconomyFileReader*, 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 funzionalità messe a disposizione dalla libreria snakeYAML nella sua ultima versione (1.33). La figura 3 mostra lo schema UML. I metodi di *EconomyFileReader* restituiscono una lista di *Map*: l'idea è 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 più 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 aumentando la manuntenibilità del codice. La figura 4 mostra l'UML di questo design.

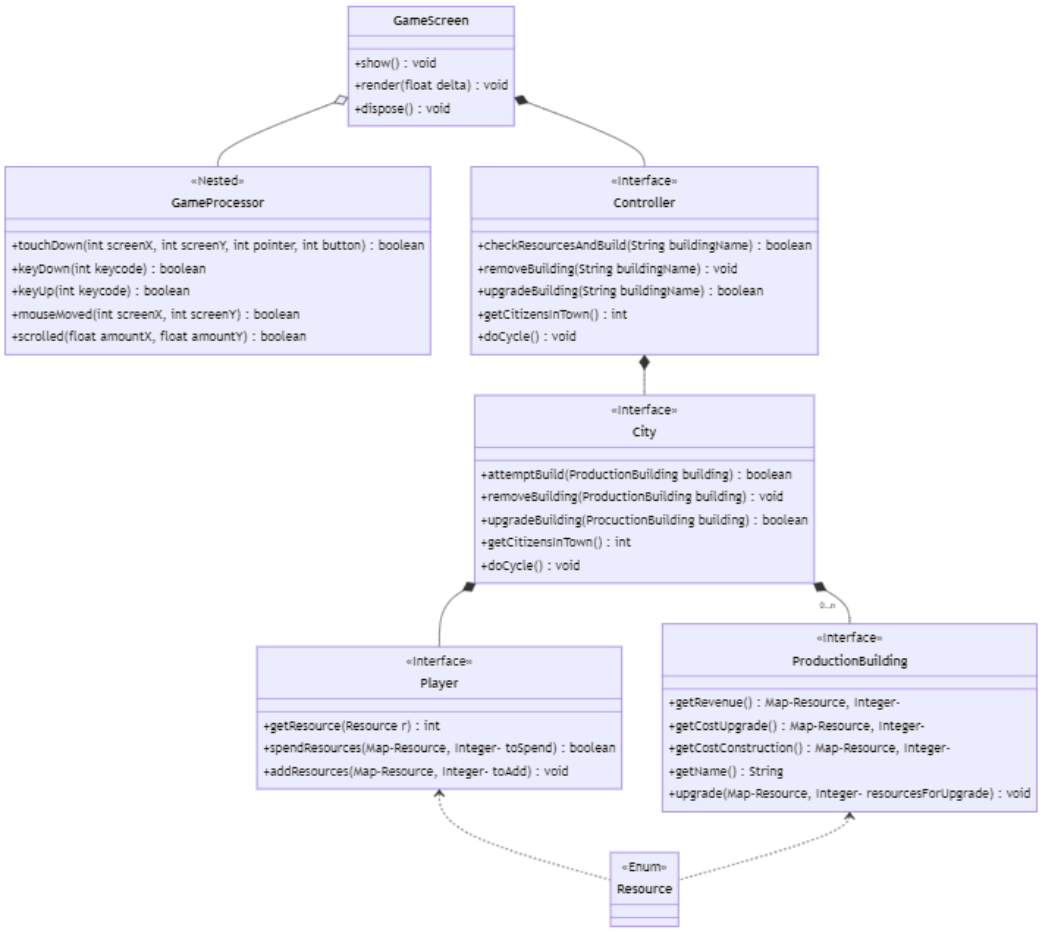


Figure 2: Diagramma UML dell'architettura

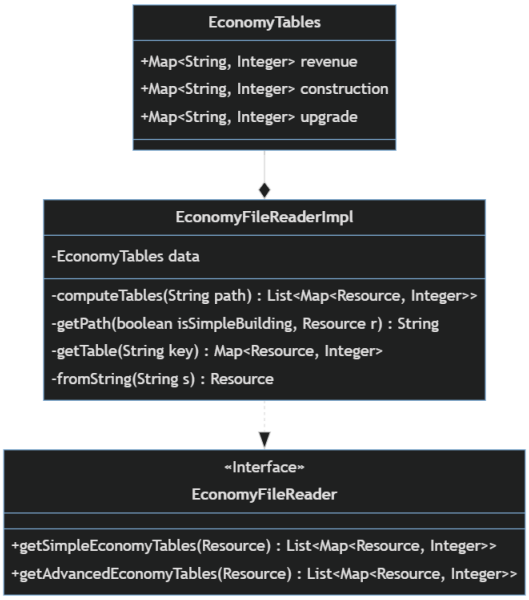


Figure 3: Diagramma UML del design *EconomyFileReader*. La classe *EconomyTables* possiede solo getter e setter per ogni campo; per motivi di sintesi, ho semplicemente riportato i campi nello schema UML (che sono in realtà privati, chiaramente).

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

- ***Menù di accesso.***

  Problema: creare il menù 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 è 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.

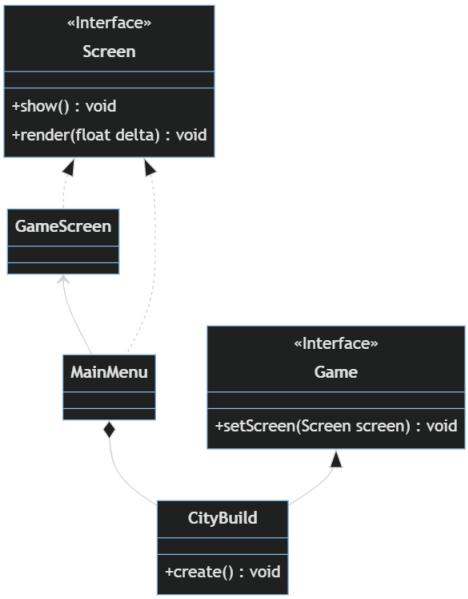


Figure 5: Diagramma UML del design del menù principale di gioco. In realtà, *GameScreen* estende *ScreenAdapter* ma tale classe è solo una implementazione di base di *Screen* che consente di fare l'override solo sui metodi di *Screen* che ci interessano.

Jhon Anthony Dagos

- ***Gestione del modello economico***

  Problema: Creare un modello economico che gestisca e manipoli le risorse, i costi e i guadagni del player e degli edifici.

  Soluzione: Ho creato due interfacce, *EconomyHandler* e *EconomyHanlderFactory*. L'interfaccia *EconomyHandler* si occupa di restituire le informazioni dei costi e dei guadagni, suddividendoli in due principali categorie: *SimpleCostTable* e *AdvancedCostTable*. Seguendo il pattern *fac-*

tory, ho delegato la creazione di istanze di *EconomyHandler* alla classe *EconomyHandlerFactory*.

Figure 6: Diagramma UML del design EconomyHandler con relativa factory

- ***Funzionalità shop*** **Problema**: Creare un menu shop all'interno del gioco, in cui il giocatore può sceglieredi acquistare una risorsa generata casualmente. **Soluzione**: È stato progettato un'interfaccia denominata *Shop*, accompagnata dalla relativa implementazione *ShopImpl*. La classe *GameScreen* comunica con l'interfaccia *Shop* per creare un *dialog* che visualizza un menu offrendo la risorsa al giocatore. In base alla scelta effettauta dal giocatore, lo shop restituisce la risorsa selezionata al *controller*, il quale si occuperà di gestire le risorse del giocatore. Questa soluzione permette al giocatore di interagire con lo shop all'interno del gioco. Inoltre, l'interfaccia e la sua implementazione sono facilmente estendibili per includere ulteriori funzionalità in futuro.

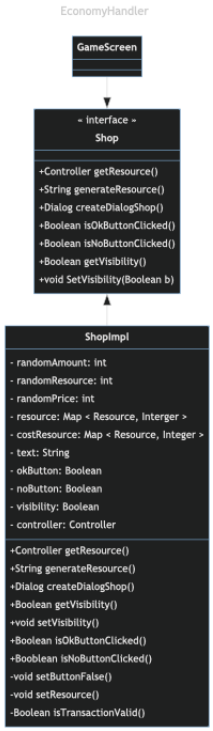


Figure 7: Diagramma UML del design Shop e relativa implementazione

Mattia Flamigni

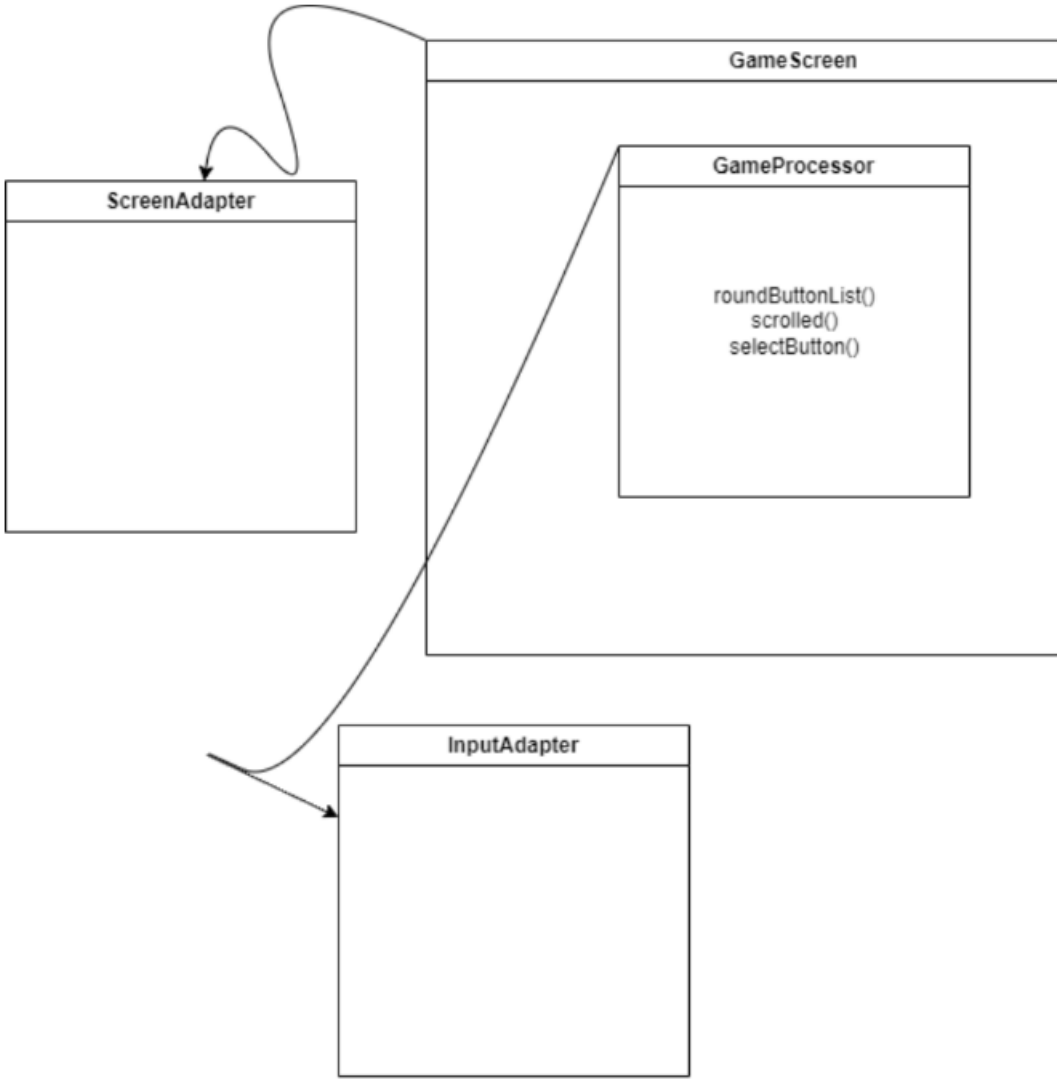
- ***Menù di scelta degli edifici***.

  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.

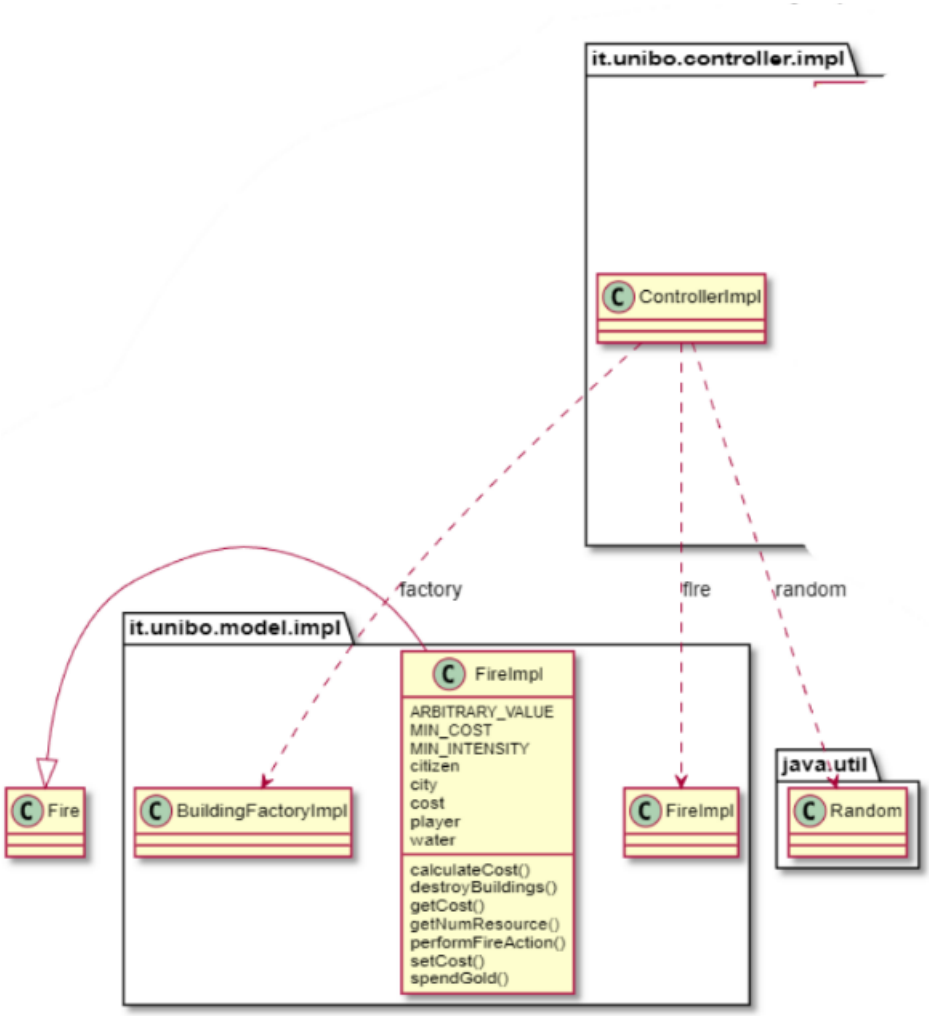
  selectButton invece è utilizzato per selezionare un pulsante specifico all'interno di una tabella e "visualizzare" le informazioni relative all'immagine selezionata.



- ***Funzionalità di incendio***.

  Problema: creare un imprevisto al giocatore. In questo caso la città può prendere fuoco andando a decrementare risorse.

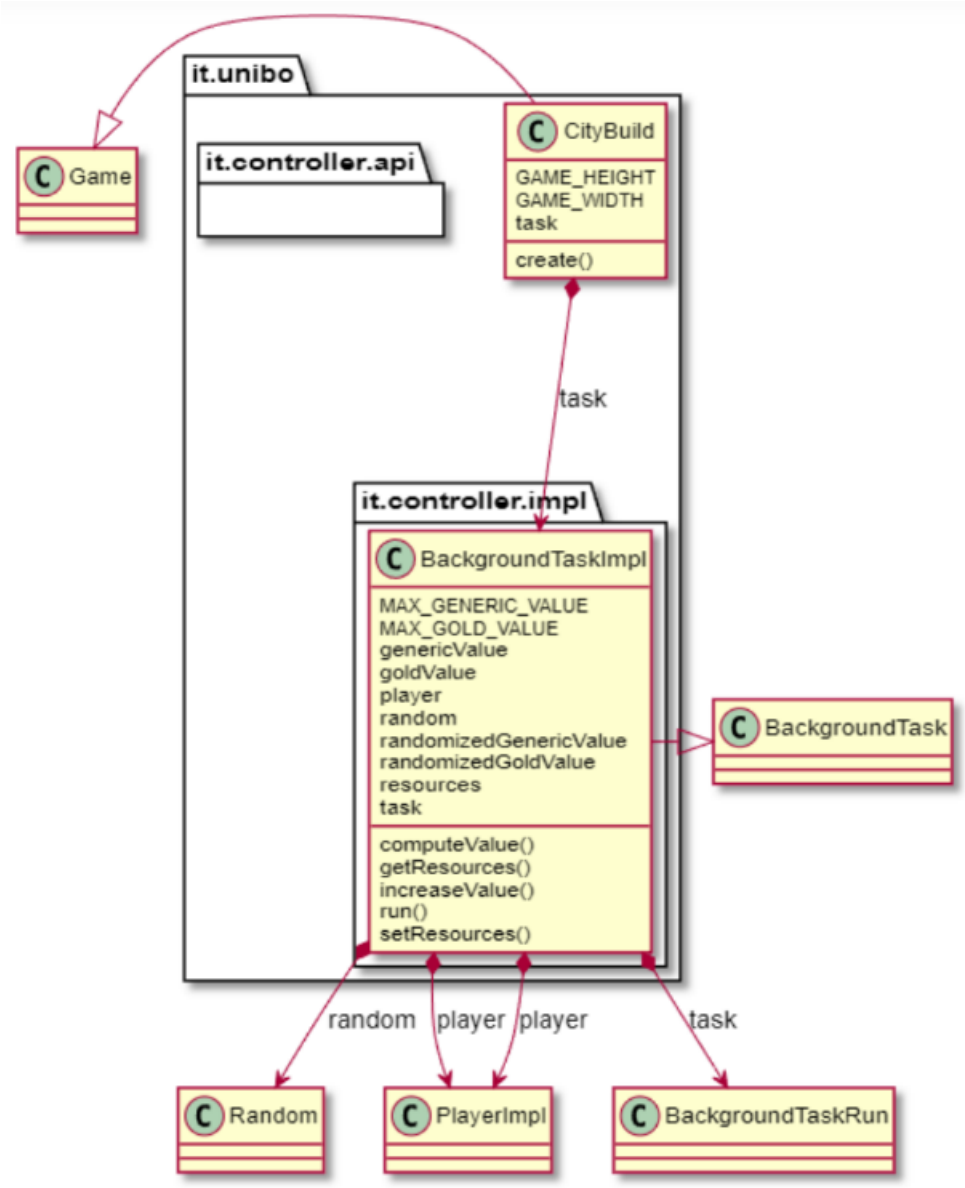
  Soluzione: è stata creata un interfaccia nella quale l'implementazione calcola intensità che influisce poi sul calcolo del costo. nel file GameScreen viene chiamato il metodo per incendiare la città.



- ***Funzionalità di aggiornamento in background***. Se il giocatore decide di salvare la partita, quando il programma viene riaperto è come se la città non si fosse mai fermata.

  Problema: trovare un modo per fare in modo che venga eseguito un thread ad applicazione chiusa.

  Soluzione: documentandomi sul web ho trovato una funzionalità della libreria 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*, *ProductionBuilding*, *Econo-*

myFileReader, *EconomyHandler* e *City*.

3.2 Metodologia di lavoro

I lavori sono stati divisi come segue:

- **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 menù principale. Progettazione del model (ovvero la classe *City*) e gran parte della *view*.
- **Jhon Anthony Dagos:** Gestione del modello economico e gestione dello shop.
- **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 differenti, e anche commit minori sul branch *master*.

3.3 Note di sviluppo

Michele Ciavatti

- ***Utilizzo della libreria di terze parti snakeYAML:*** la classe *EconomyFileReaderImpl* utilizza tale libreria. Permalink alla prima riga di codice in cui si usa la libreria. `https://github.com/MicheleCiavatti/OOP22-CityBuild/blob/864d62cd87667a2b47ce890060f0aa766becd1ae/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 è la classe all'url `https://github.com/MicheleCiavatti/OOP22-CityBuild/blob/864d62cd87667a2b47ce890060f0aa766becd1ae/core/src/main/java/it/unibo/view/MainMenu.java#L26`.
- ***Utilizzo di lambda:*** usate pervasimante nel codice, un esempio si trova al link `https://github.com/MicheleCiavatti/OOP22-CityBuild/blob/864d62cd87667a2b47ce890060f0aa766becd1ae/core/src/main/java/it/unibo/model/impl/ProductionBuildingImpl.java#L68`
- ***Utilizzo di stream:*** usati pervasimente nel codice. Un esempio al link `https://github.com/MicheleCiavatti/OOP22-CityBuild/blob/864d62cd87667a2b47ce890060f0aa766becd1ae/core/src/main/java/it/unibo/model/impl/ProductionBuildingImpl.java#L87`

- ***Utilizzo di Optional:*** usati più volte nel codice, un esempio è nella classe *GameScreen* (permalink alla dichiarazione del campo): `https://github.com/MicheleCiavatti/OOP22-CityBuild/blob/864d62cd87667a2b47ce890060f0aa766becd1ae/core/src/main/java/it/unibo/view/GameScreen.java#L55`

I file per lo stile dei tasti nel menù iniziale sono stati presi dal link: `https://github.com/czyzby/gdx-skins`. Inoltre, il metodo *setColorLabel()* della classe *GameScreen* è 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 Autovalutazione e lavori futuri

Michele Ciavatti

Il codice da me creato è 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 però ben conscio di un punto molto debole della mia progettazione, ovvero l'utilizzo di una enumerazione per le risorse del software. Tale scelta è chiaramente molto comoda per i programmatori, che possono facilmente accedere alle funzionalità legate alle risorse e fare confronti veloci e precisi fra risorse differenti, ma ha anche il forte svantaggio di rendere il codice meno manuntenibile.

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. In futuro si potrebbero implementare molte più opzionalità in riferimento al gioco o migliorare quelle presenti (esempio l'upgrade che può essere fatto solo una volta per edificio).

Mattia Flamigni

Riconosco che il mio codice Java potrebbe non essere perfetto, 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.

Jhon Anthony Dagos

Nella mia attività di programmazione, ho utilizzato il pattern factory per creare codice che è in parte mantenibile e riutilizzabile. Tuttavia, durante la progettazione dello shop, sono consapevole che il mio codice potrebbe non essere perfetto e che ci sono alcune parti che avrei potuto migliorare per renderlo più efficiente e meno ripetitivo.

Nonostante ciò, ho cercato di seguire il più possibile il pattern factory e ho lavorato diligentemente per aggiungere funzionalità alle classi progettate dai miei colleghi. Credo di aver avuto successo nel contribuire al lavoro di squadra e nel creare codice che soddisfa gli obiettivi del progetto.

In futuro, continuerò a migliorare le mie abilità di programmazione e a cercare di applicare le migliori pratiche nella progettazione del codice. Sono aperto alle critiche costruttive e ai suggerimenti dei miei colleghi per migliorare il mio lavoro e raggiungere i nostri obiettivi comuni.

4.2 Difficoltà incontrate e commenti per i docenti

Michele Ciavatti

Riporto qui alcune criticità riscontrate nel progetto. In sintesi, la mia opinione è che un progetto di gruppo difficilmente è equo nel lavoro e nella valutazione, e non di rado capitano imprevisti che costringono membri attivi a fare più lavoro di quello che il progetto richiederebbe. Nel nostro caso, è eclatante che una delle persone non ha di fatto partecipato in alcun modo, per cui gli altri hanno dovuto fare più lavoro per cercare di coprire la sua parte. La mia opinione è che i progetti dovrebbero essere singoli per permettere a ciascuno di fare effettivamente le 80 ore concordate senza squilibri. Inoltre, io stesso mi sono ritrovato a dover far fronte a problematiche del gruppo relative al progetto che esulavano dalle parti di progetto che mi erano state assegnate, ed è chiaramente impossibile conciliare il fatto di dover fare al massimo 80 ore con problemi del progetto che altri membri non sanno risolvere o, nel caso peggiore, non vogliono risolvere (come appunto è successo con Matteo Buldrini).

5 Guida Utente

La guida è riportata nel file README del progetto. Dentro tale file sono anche riportati i credits per musiche, suoni e sprite. Per riuscire a giocare efficacemente, si consiglia di costruire inizialmente una *Foundry*, poi un *Woodcutter* e *Depurator*, infine un *Power plant* appena possibile.

Si noti che il gioco è molto più rapido di quanto dovrebbe essere: tale rapidità è giustificata proprio dal poter mostrare ai professori tutte le caratteristiche del software in tempi brevi.