Relazione MedievalVillage

Luca Passeri
Diego Pergolini
Jacopo Mastrogirolamo

Sommario

1 Analisi

1.1 Requisiti

Il software mira alla realizzazione di un gioco di tipo RTS (Real Time Strategy), ispirato ai famosi “Sim City”, “Faraon” e“Stronghold Crusader", dove l’azione fluisce in modo continuo e l’utente deve cimentarsi nella gestione di una città medievale nei suoi vari aspetti.
MedievalVillage offre le specifiche di un gioco gestionale completo, consentendo all’utente di personalizzare vari aspetti del gioco quali la scelta della mappa, della difficoltà… e inoltre consentendo di salvare i propri progressi di gioco per poi riprendere la partita successivamente.
Una volta iniziata una partita sarà possibile sviluppare il villaggio a proprio piacimento, scegliendo tra una vasta gamma di edifici da costruire, ognuno adibito a specifiche mansioni. Sarà compito dell’utente gestire in maniera coscienziosa il villaggio nei suoi vari aspetti tentando di immedesimarsi il più possibile in un capace governante di un villaggio medievale sfruttando al meglio le risorse presenti disponibili.

Il software esporrà le seguenti funzionalità:

- Il giocatore avrà la possibilità di scegliere fra una varietà di mappe, ognuna delle quali dotata di diverse caratteristiche e difficoltà legate alla conformazione dell’ambiente circostante. Una miniatura della mappa consentirà di visualizzare la morfologia del territorio aiutando l’utente nella scelta della stessa.
- Si avrà la possibilità di costruire vari edifici, caratteristici dell’età medievale, disponendo delle giuste risorse naturali quali legno e pietra, in quantità relazionate all’importanza e tasso di produzione dell’edificio stesso. Si dovrà disporre inoltre del numero di popolani richiesto per il regolare svolgimento delle sue funzioni. Gli stessi edifici potranno anche essere rimossi o disabilitati.
- Sarà possibile visualizzare l’ammontare delle principali risorse associate alle condizioni economiche del villaggio (Popolazione, Oro, Cibo …).
- Visualizzazione degli indicatori sociali del villaggio (Popolarità, Sanità, Cultura …) che andranno ad influenzare le dinamiche di gioco; la bontà delle decisioni prese durante lo svolgimento della partita determineranno direttamente il miglioramento o peggioramento del villaggio stesso.
- Il giocatore potrà, in qualsiasi istante, interrompere e salvare l’attuale sessione di gioco per poi riprenderla in un secondo momento.
- Sarà possibile scegliere tra più livelli di difficoltà.

- Una serie di imprevisti potrà influenzare l'andamento del villaggio premiando o punendo il giocatore a seconda della condotta di gioco.
- Il giocatore potrà scegliere fra vari livelli di tasse e salari che influenzeranno le condizioni socio-economiche del villaggio.
- Per rendere più piacevole l'esperienza di gioco saranno presenti effetti sonori a seconda della situazione corrente.

1.2 Analisi e Modello del Dominio

MedievalVillage si propone di offrire al giocatore una divertente e quanto più verosimile simulazione di un villaggio medievale. Egli potrà sviluppare la propria città secondo le sue preferenze cercando di rendere il villaggio sempre più grande e florido cercando di sfruttare al meglio le risorse disponibili.
La mappa di gioco iniziale dovrà presentare più tipologie di terreni, ognuno con una determinata influenza verso gli edifici che potranno essere costruiti: infatti alcuni saranno posizionabili solo in presenza di determinate condizioni.
Il software dovrà controllare che il giocatore disponga delle risorse necessarie alla costruzione e in caso contrario notificare all'utente quelle mancanti.
Gli edifici possono essere costruiti, rimossi o disabilitati essendo elementi dinamici del gioco a differenza dei terreni, caratteristiche intrinseche e immutabili della mappa; sarà difficile modellare costruzioni con funzioni anche molto diverse tra loro ma realizzando comunque un'entità che li accomuni.
Altre entità dovranno gestire l'accumulo delle risorse in base alla loro tipologia, un caso particolarmente complesso sarà la gestione della popolazione: i popolani si suddivideranno in due categorie, i lavoratori e i disoccupati, entrambe fondamentali ma differenti tra loro. La popolazione consuma risorse, paga tasse, riceve stipendi, consente il funzionamento degli edifici ma può anche morire in caso di cibo insufficiente, potendo provocare la disattivazione dell'edificio nel quale lavoravano. È evidente quanto la complessità di queste relazioni sia difficilmente modellabile.
Non sarà semplice poi gestire la produzione ed il consumo di risorse considerando la mole di entità in relazione fra loro e tutte le possibili casistiche dovute alla complessità del villaggio.
Una specifica entità dovrà gestire la situazione economica del villaggio che potrà essere influenzata dall'utente in base ai livelli di tasse e salari scelti.
Andrà individuato un modo per restituire un valore che rispecchierà più fedelmente possibile la situazione corrente del villaggio sotto vari punti di vista.

Se sarà possibile nel gioco saranno introdotti anche eventi eccezionali dovuti alla situazione del villaggio, sarà difficile relazionare questi avvenimenti con le già complesse dinamiche di gioco; inoltre se il modello sarà ben organizzato non dovrebbe risultare troppo complicato aggiungere più livelli di difficoltà.
Il software dovrà essere in grado di funzionare su più sistemi operativi fornendo un'interfaccia grafica reattiva e gradevole.

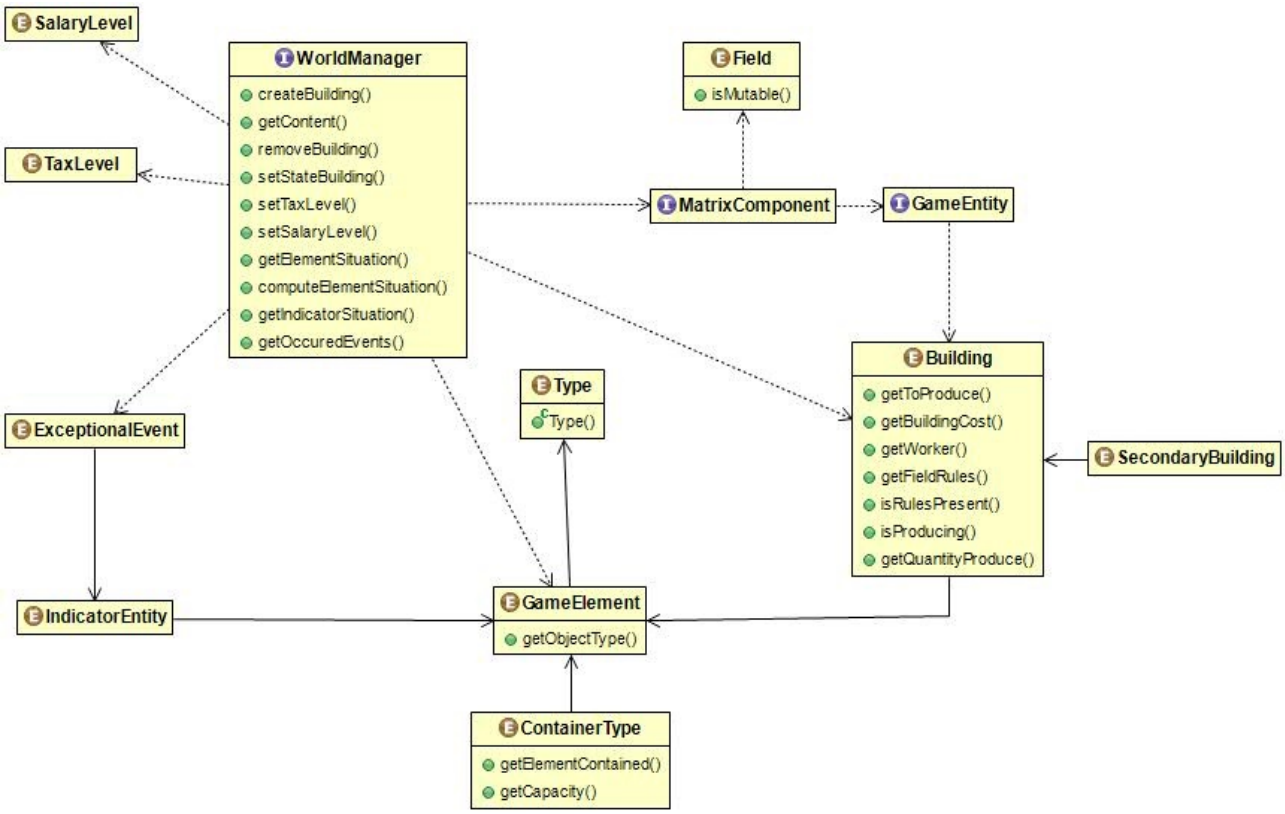

Figura 1
Schema UML del dominio, rappresentante le principali entità del gioco e le relazioni che intercorrono fra di esse.

2 Design

2.1 Architettura

Al fine di permettere l'interazione fra i vari componenti di gioco si è deciso di utilizzare il pattern architetturale MVC, definendo così tre principali sezioni: Model, View e Controller.

Il seguente diagramma UML mette in luce le relazioni fra i componenti architetturali.

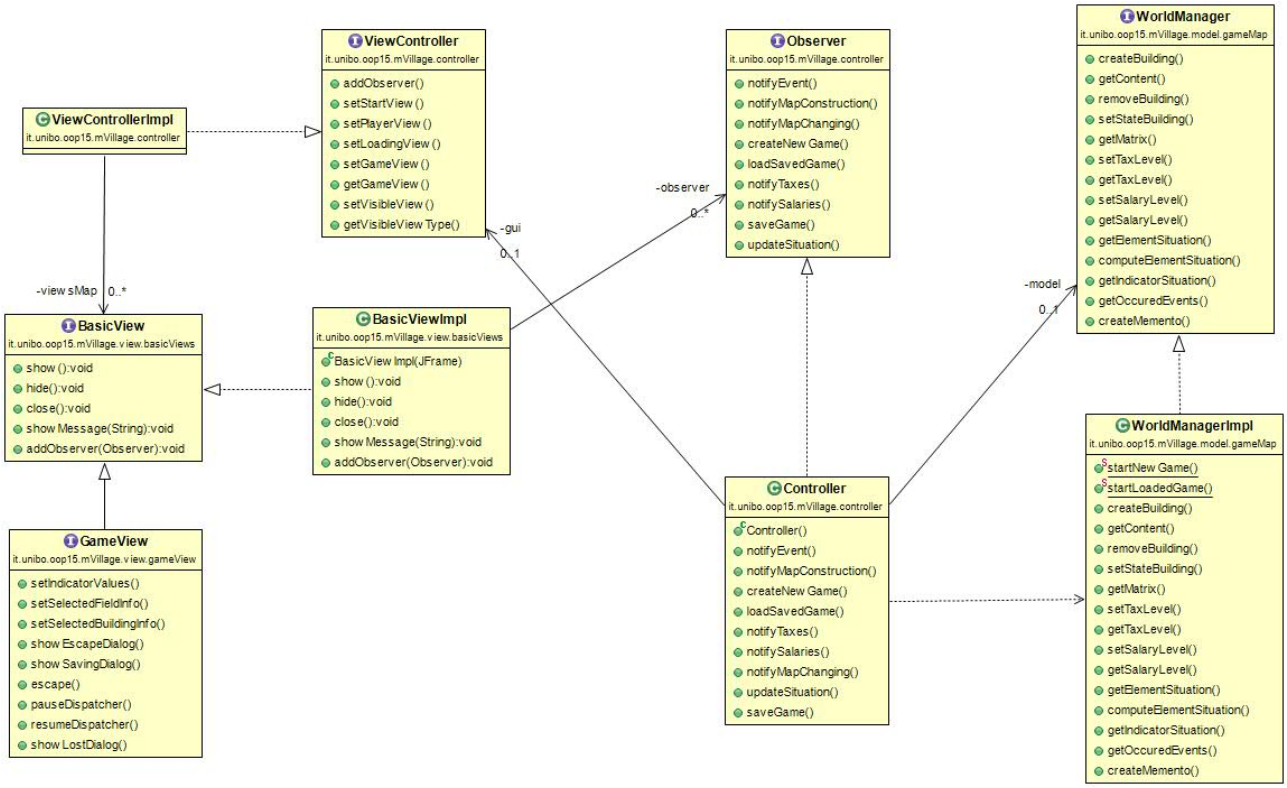

Figura 2
Overview dell'architettura che mostra i componenti principali del software:
il Controller si relaziona con il Model attraverso l'interfaccia WorldManager che offre tutte le funzionalità individuate in fase di analisi.
Per la View invece l'interfacciamento avviene indirettamente utilizzando un componente che si occupa di gestire le varie schermate.

Il Controller ha il compito di istanziare nel momento opportuno il Model e demandare ad esso le richieste dell'utente tramite un unico entry-point che maschera la complessità dei dettagli implementativi attraverso l'interfaccia *WorldManager*.

La View rappresenta la traduzione a livello grafico del dominio di gioco, in particolare, come si evince dallo schema UML, il Controller si deve interfacciare ad essa con due tipi di schermate:

- Quelle che permettono all'utente di scegliere le impostazioni di gioco sono delle implementazioni dell'interfaccia "*BasicView*"
- Quella che invece raccoglie le vere interazioni di gioco è una specializzazione di "*BasicView*", rappresentata dall'interfaccia *GameView*

Il Model è stato realizzato in modo da garantire la sua completa indipendenza, permettendo così eventuali modifiche successive senza avere impatti sulle altre parti.

Grazie al design definito precedentemente se si decidesse di sostituire la View (ad esempio utilizzando un'altra libreria grafica) basterebbe implementare le principali interfacce che descrivono le funzionalità indispensabili per questo gioco.

2.2 Design Dettagliato

Model -*Diego Pergolini*

Per implementare il dominio del gioco,nella fattispecie il nostro villaggio medievale,si è cercato di incapsulare più possibile le dinamiche strettamente computazionali, quindi nascoste al giocatore e su cui non devono agire direttamente le altre parti dell'architettura,esponendo soltanto i metodi che riguardano interazioni dirette con l'utente. L'utente durante la sessione di gioco può infatti costruire nuovi edifici, disabilitarli, rimuoverli, può cambiare il livello delle tasse e dei salari, salvare la partita; I metodi relativi a queste interazioni,insieme ai metodi che ritornano la situazione dei vari elementi costituenti del gioco (Indicatori,Risorse) o di una cella della mappa ed a quello che ordina di fare una nuova computazione costituiscono proprio l'interfaccia *WorldManager*.

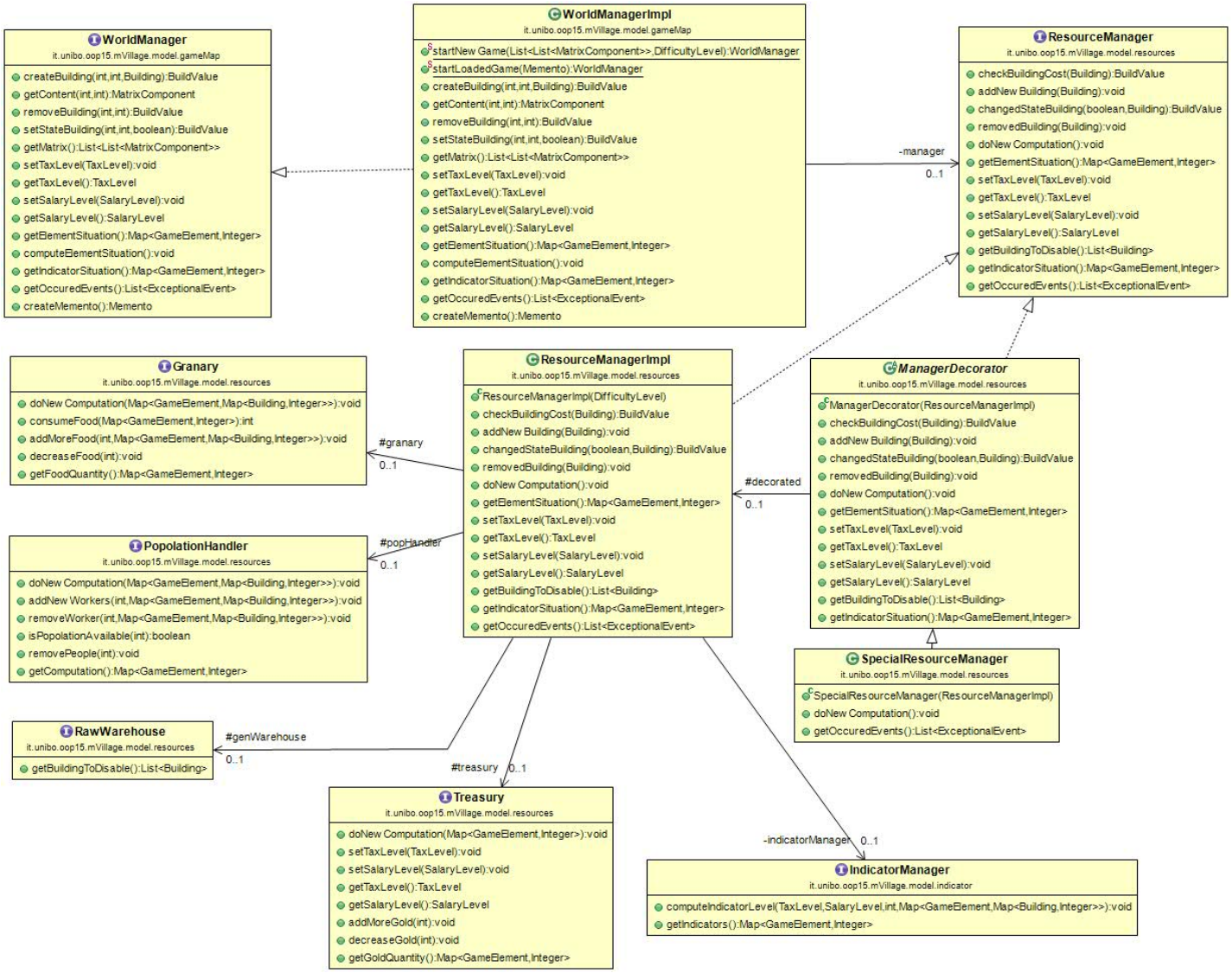

Figura 3
Schema UML che rappresenta l'architettura generale del Model, i vari dettagli saranno più chiari proseguendo nella lettura.

WorldManager:

Questa interfaccia è l'unica parte del Model a contatto diretto con il Controller, rendendo così semplice e diretto l'interazione fra essi,offrendo tutte le varie funzionalità senza doversi preoccupare dell'effettivo funzionamento interno. La sua implementazione concreta *WorldManagerImpl* è innanzitutto istanziabile solamente attraverso due metodi statici,a seconda che si voglia iniziare una nuova partita oppure caricarne una, infatti i costruttori sono stati resi privati per evitare di dover esporre due costruttori con numero e tipo di parametri diversi,usando invece static factory methods (***"Consider static factory methods instead of constructors"*** [1]) si offre una maggior leggibilità ed espressività del codice, questi metodi hanno solitamente lo svantaggio che se in presenza di soli costruttori

[1] Item 1- Effective Java - Second Edition- J.Bloch

privati non si può estendere la classe,in questo caso la cosa non risulta problematica in quanto la classe non è stata progettata per essere estesa ed infatti è dichiarata final("***Design and document for inheritance or else prohibitit"*** [2]).

Uso del pattern Facade:

WorldManagerImpl è stato implementato seguendo il pattern ***Facade***, nascondendo cioè la complessità intrinseca delle funzionalità offerte dal modello dietro un'unica interfaccia,detta appunto "Facciata". Per spiegare meglio il perché di questa scelta, oltre a mostrare il diagramma UML (Figura 4),ritengo sia utile fare un esempio: Ogni Task il controller deve ordinare al Model di fare una nuova computazione, quindi viene chiamato il metodo *computeElementSituation*(),ma che cos'è questa computazione? Essa comprende vari aspetti, interdipendenti ma diversi tra loro,infatti ogni ciclo vanno calcolate le nuove produzione di materiali causate dagli edifici produttivi che possono essere presenti nel villaggio, questi edifici produttivi possono però consumare altre risorse per produrre determinati elementi,risorse che potrebbero non essere disponibili, in tal caso l'edificio deve essere disabilitato automaticamente e non produrre più. Non tutti gli edifici però producono risorse, alcuni hanno effetto sui vari indicatori(salute,religione, cultura, popolarità, sicurezza). Negli edifici però lavorano popolani,che ovviamente consumano cibo, pagano tasse, ricevono stipendi,se il cibo non fosse abbastanza per tutti i popolani, alcuni necessariamente morirebbero, ovviamente i primi a morire saranno i disoccupati,ma nella malaugurata ipotesi che anche dei lavoratori morissero,gli edifici in cui lavoravano andrebbero disabilitati. È quindi evidente come tutti questi aspetti interni non devono essere esposti all'utilizzatore di *WorldManager*, vista l'elevata complessità e l'elevato numero di componenti in gioco.

La gestione del dominio può essere divisa logicamente in due parti:

- Gestione della mappa vera e propria
- Gestione delle risorse ed indicatori

WorldManagerImpl gestisce direttamente gli aspetti della mappa,mentre demanda ad un *ResourceManager* gli altri aspetti,che a sua volta si compone di altre classi più specifiche per

[2] Item 17- Effective Java - Second Edition- J.Bloch

svolgere determinate funzioni.

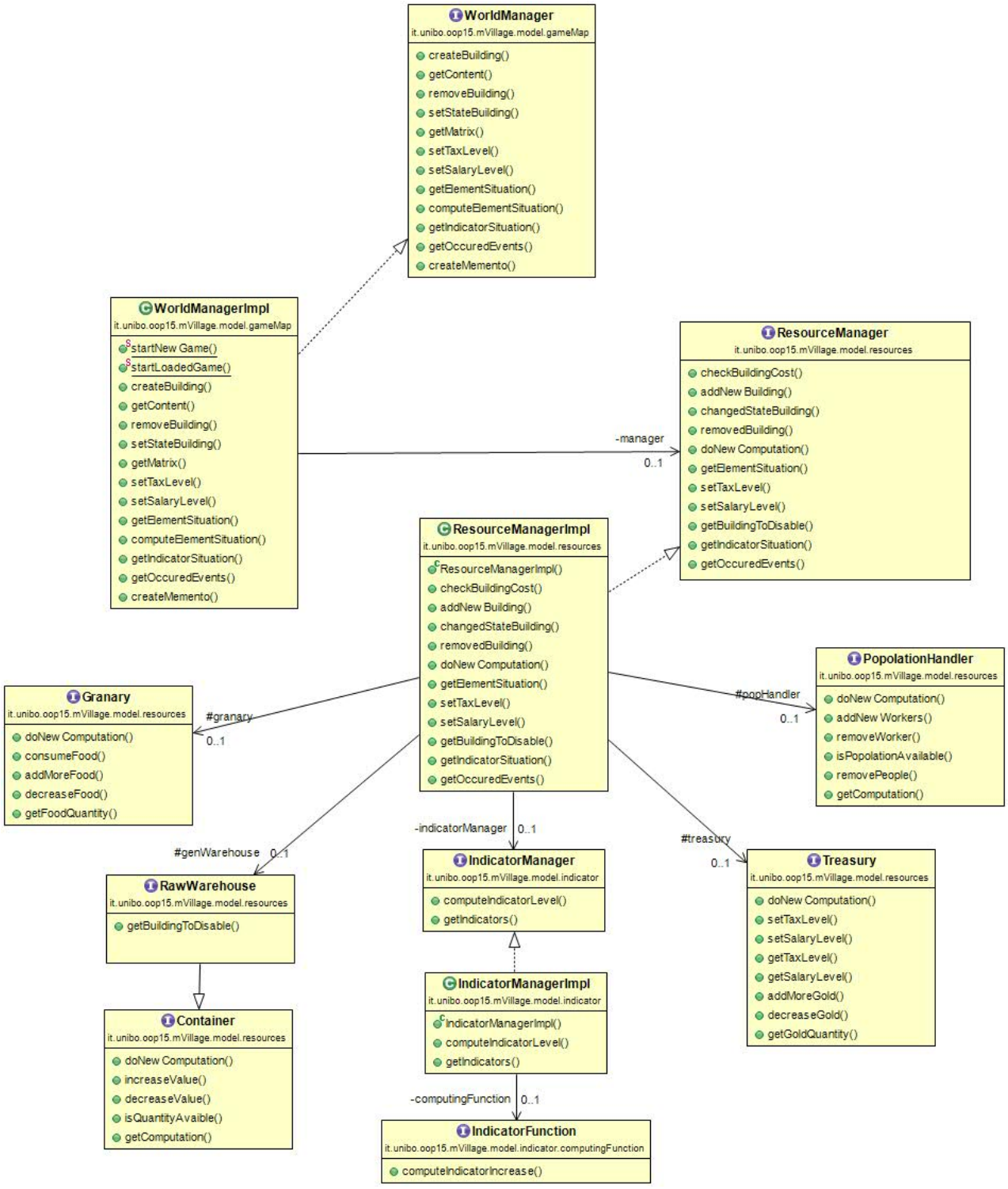

Figura 4

WorldManager *è la "facciata" dietro cui si nasconde tutta la complessità dei meccanismi che animano il dominio di gioco, infatti ogni suo metodo può chiamare più classi per arrivare espletare la propria funzione,il caso più eclatante è quello del metodo* ***computeElementSituation()****, in cui tutte le classi in figura vengono interrogate.*

MatrixComponent:

La mappa di gioco viene modellata come una matrice di *MatrixComponent*,questa interfaccia definisce i metodi offerti da un elemento della matrice,quali ottenere il tipo di terreno e l'eventuale l'edificio presente sullo stesso e di rimuovere/disabilitare quell'edificio. I tipi di terreni sono stati definiti nell'enumerazione *Field*, mentre gli edifici nell'enumerazione *Building*, questa scelta dona una grande estendibilità al progetto,potendo aggiungere nuovi edifici semplicemente aggiungendoli all'enum (definendo ovviamente i dati fondamentali richiesti che descrivono il funzionamento dell'edificio),ma come anche nuovi terreni,senza dover toccare altre parti del modello.

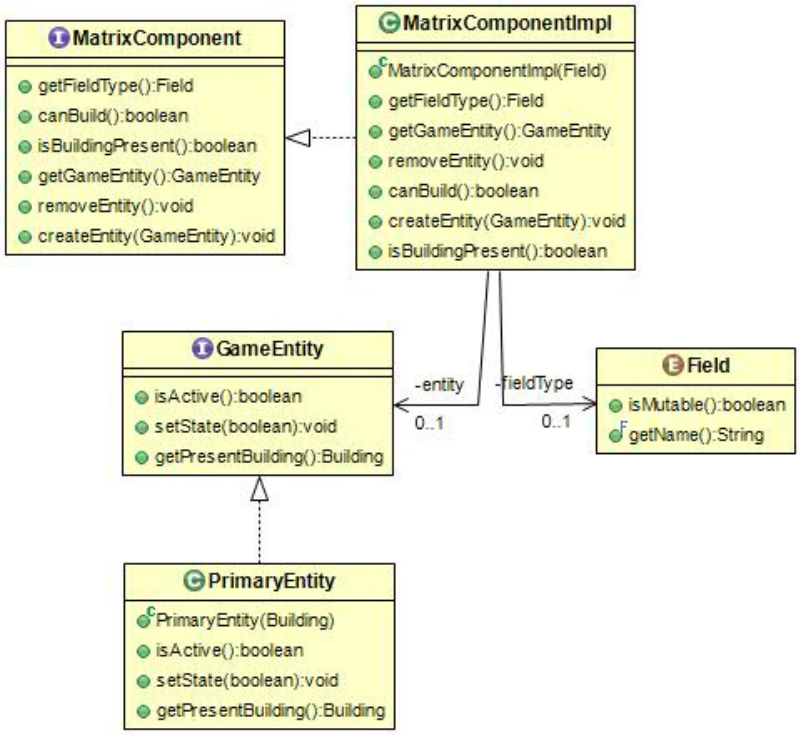

Figura 5
*Schema UML del design di **MatrixComponent,** cioè l'unità base della matrice che rappresenta la mappa di gioco. Esso è implementato concretamente dalla classe **MatrixComponentImpl**, che si compone di una **GameEntity**(Edificio ed il suo stato) e di un **Field** (tipo di campo).*

Uso del pattern Memento:

In *WorldManagerImpl* viene applicato anche il pattern ***Memento***, per salvare lo stato di un oggetto senza violarne l'incapsulamento e poterlo poi ripristinare in un secondo momento. Più specificatamente in questo caso ho implementato una classe statica Memento dentro *WorldManagerImpl*, essa mantiene al suo interno la *List<List<MatrixComponent>>,* (cioè la matrice di gioco) e il *ResourceManager* attuale (in cui sono contenute le varie informazioni relative alle

risorse,indicatori …) .Questa classe *Memento* ha costruttore privato che prende come parametri i due oggetti di cui sopra ed ha campi e getter privati. Così facendo un istanza di memento si può creare solo all'interno di *WorldManagerImpl*, in quanto è classe innestata statica,che nel Pattern Memento prende il nome di *Originator*, quando il giocatore decide di salvare la partita il controller può così chiamare il metodo *creare Memento()* che gli ritorna un Memento appena creato. Qualsiasi classe al di fuori di *WorldManagerImpl* non può effettuare alcuna operazione su questo oggetto, in quanto non ha metodi o campi pubblici,ciò porta a diversi vantaggi,innanzitutto si è assolutamente certi che lo stato della partita nel momento del salvataggio(e quindi della creazione del oggetto Memento) non verrà modificato in un secondo momento da classi esterne,evitando qualsiasi forma di problema al momento del caricamento della partita,inoltre il codice così facendo è molto più incapsulato,si nascondono dettagli implementativi interni al model,favorendo una più facile manutenzione futura. Se per esempio in futuro nascesse l'esigenza di salvare anche un'altra informazione all'atto del salvataggio dello stato attuale del dominio, dovremmo solo aggiornare la classe Memento, senza toccare alcuna riga di codice all'infuori del model. Se invece avessi creato due distinti metodi per ritornare la matrice ed il *ResourceManager* per poi salvarli, avrei potuto si creare delle copie difensive prima di ritornarli,ma non avrei avuto nessuna garanzia sul fatto che qualche altra classe non avrebbe modificato quei due oggetti prima di salvarli e ovviamente nel caso di modifiche di cui sopra, queste avrebbero impattato anche sulla gestione del salvataggio e caricamento nel controller. Riassumendo(Figura 6): *L'originator* (*WorldManagerImpl*) origina un oggetto *Memento* che racchiude lo stato attuale dell'Originator, questo Oggetto viene poi salvato su File dal *Caretaker*(il controller). Lo stesso oggetto all'atto del caricamento, verrà preso da File, passato come parametro al metodo *StartLoadedGame* di *WorldManagerImpl*, che prenderà dall'oggetto i campi che gli interessano per ripristinare una partita salvata (attraverso i getter privati).

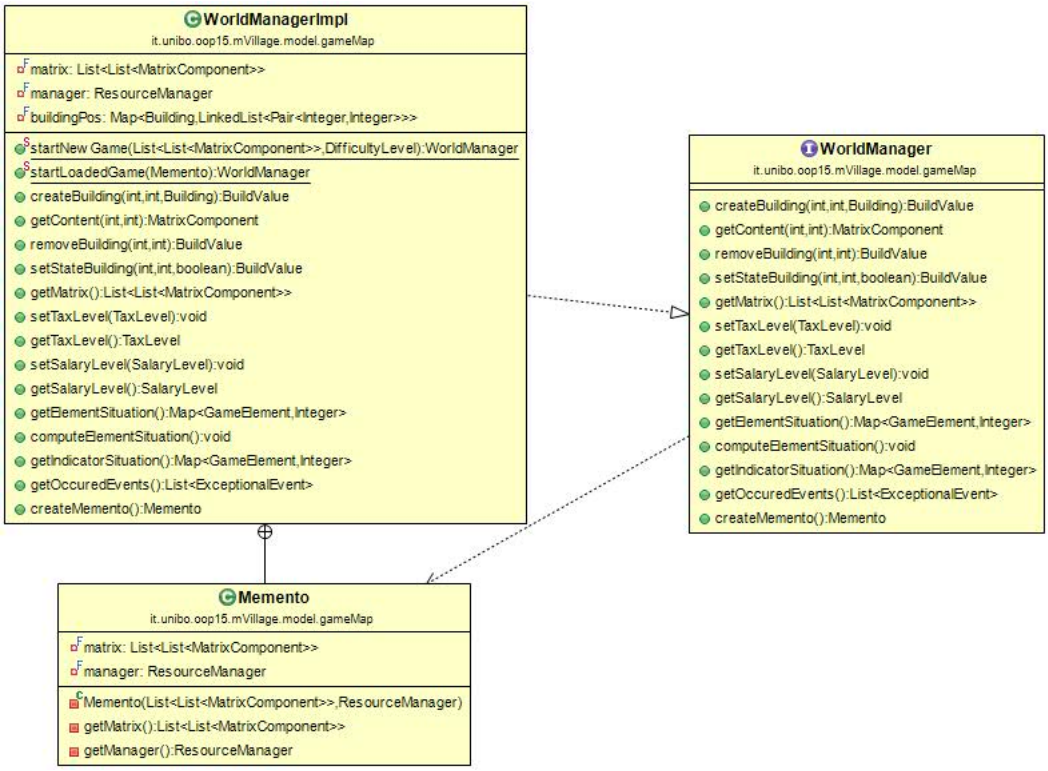

Figura 6
*L'originator è rappresentato da **WorldManagerImpl**, che genera un oggetto **Memento** ,il Caretaker (che è interessato al salvataggio dello stato dell'originator,nel nostro caso il Controller) ottiene questo oggetto grazie al metodo **createMemento.***

ResourceManager:

Penso sia importante soffermarsi sul design di *ResourceManager* e le sue implementazioni,ritengo infatti questa parte come il vero cuore del model, a prima vista guardando i metodi che offre sembra molto simile a *WorldManager*,in effetti le funzionalità di facciata che offre sono un suo sotto insieme, ma quello che cambia è l'ambito in cui vengono applicate,infatti per quelle in comune, in *WorldManager* si controlla direttamente soltanto che siano soddisfatti i vincoli fisici,di mappa, mentre i vincoli relativi alle risorse economiche(popolazione libera,risorse …) vengono demandati al *ResourceManager*. È quindi abbastanza chiara la maggior complessità dei problemi da affrontare in questa parte,causata dall'alto numero di interrelazioni fra gli elementi socio/economici del villaggio,motivo per cui l'implementazione pratica *ResourceManagerImpl* si compone di altri oggetti che vanno a gestire campi ancora più specifici del dominio. Più precisamente si compone di un *Granary* (Gestore di cibo), *PopulationHandler* (gestore della popolazione), *RawWarehouse* (gestore del magazzino di materie prime), *Treasury*(gestore dell'oro) ed *IndicatorManager* (gestore degli indicatori)(vedi Figura 7). Un'architettura di questo tipo è senz'altro più mantenibile e chiara,in quanto si

sono minimizzate le dipendenze fra le varie classi,isolando i vari comportamenti,così che modifiche future interni ad essi non vadano ad impattare sul resto del modello. Per realizzare i gestori di cibo,popolazione e materie prime già citati ho cercato di massimizzare il riuso ed evitare ripetizioni di codice,infatti come si può notare nella figura, tutte le funzionalità base che deve avere un contenitore(che sia di cibo,materie prime o persone) sono offerte dall'interfaccia Container e dalla sua implementazione *GeneralContainer* e quindi ognuno di essi contiene un oggetto di questo tipo. Fondamentale per un ottimo riuso del codice è stato poi definire l'Enum *ContainerType*, che elenca i tipi di *Container* che possiamo trovare,con le relative risorse che possono contenere e la capienza. Si possono istanziare così dei *Container* diversi semplicemente specificando il tipo di Container desiderato. I vari tipi di gestori si compongono quindi del *GeneralContainer* del tipo appropriato (ad esempio *RawWarehouse* avrà al suo interno un *GeneralContainer* istanziato passando come parametro *ContainerType.GENERAL_WAREHOUSE*). Questo approccio mi è sembrato più adatto piuttosto di usare strutture di ereditarietà,visto che le classi che avrebbero esteso *GeneralContainer* non sarebbero state suoi meri sottotipi, bensì *GeneralContainer* è più che altro un dettaglio implementativo(***"Favour composition over inheritance"*** [3]).

[3] Item 16- Effective Java - Second Edition- J.Bloch

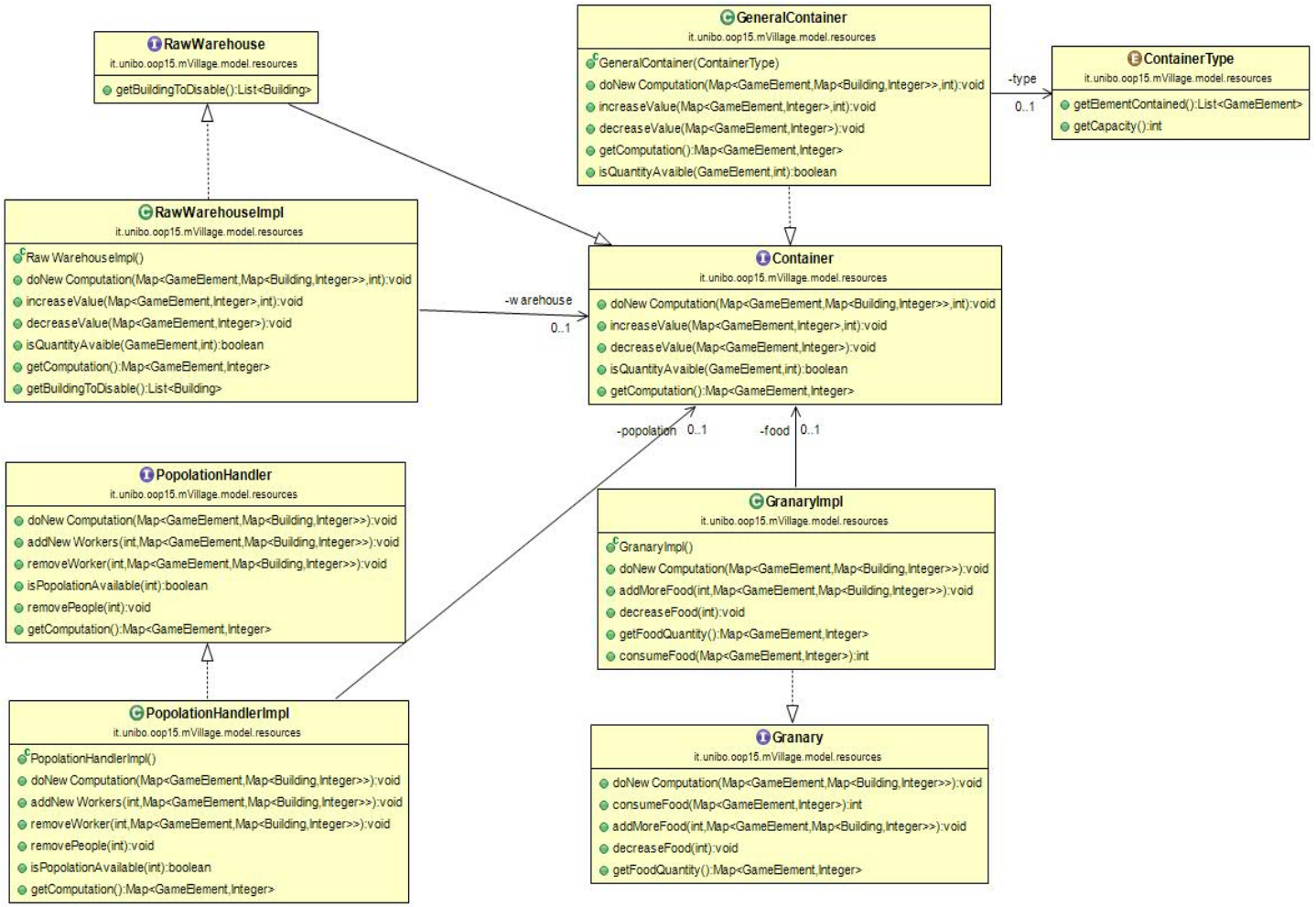

Figura 7
Schema UML rappresentante i vari gestori di Elementi del gioco. **PopolationHandlerImpl**, **RawWarehouseImpl**, **GranaryImpl** *si compongono di un* **Container** *a cui demandano le loro funzionalità base.*

IndicatorManager ed uso del pattern Strategy:

La parte relativa al calcolo degli indicatori sociali del villaggio viene invece assolta da un *IndicatorManager*, e quindi dalla sua implementazione *IndicatorManagerImpl*, in essa viene applicato il pattern ***Strategy***. Il pattern è stato applicato definendo una famiglia di algoritmi per calcolare la variazione del valore degli indicatori, rappresentati dal l'interfaccia funzionale *IndicatorFunction* che comprende il solo metodo *computeIndicatoIncrease()*. Nel costruttore di *IndicatorManagerImpl* deve essere quindi passata un oggetto facente parte di questa famiglia di algoritmi, per poterlo così utilizzare nella computazione dei valori degli indicatori in ogni task (Figura 8). L'utilizzo del pattern ***Strategy*** permette così una grande flessibilità nello scegliere come queste variazioni vadano calcolate, potendo usare qualsiasi

oggetto della famiglia degli *IndicatorFunction*, la strategia funzionale permetterebbe addirittura di usare le lambda, ampliando ancor di più la varietà di strategie adottabili.

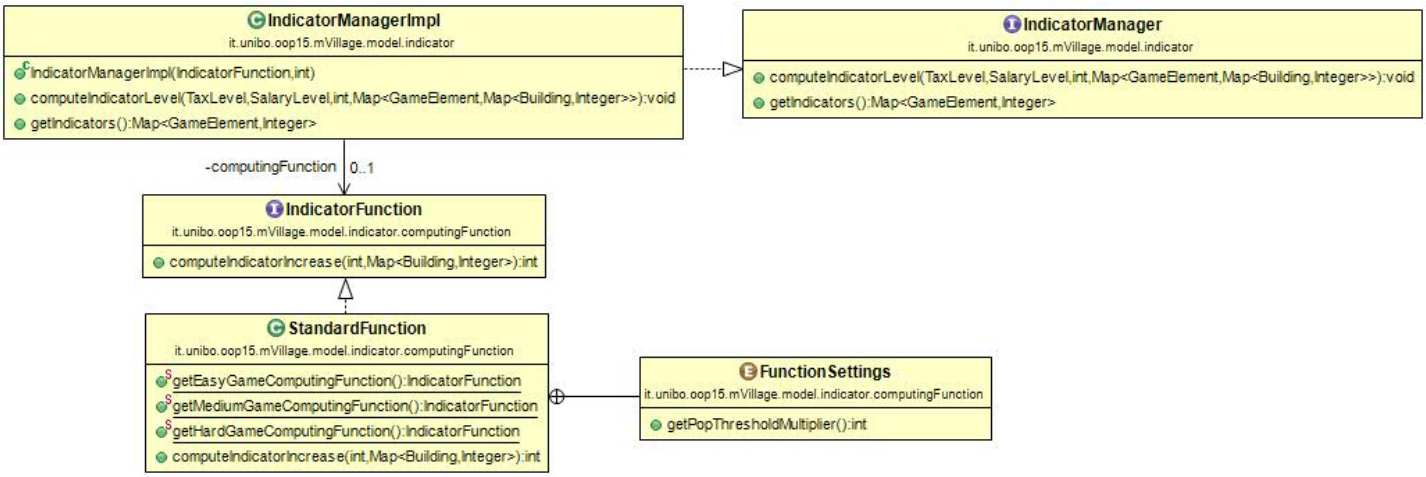

Figura 8
L'IndicatorManagerImpl *viene istanziato passandogli con quale oggetto della famiglia delle* ***IndicatorFunction*** *dovrà calcolare i valori degli indicatori, funzioni che vengono fornite dalla StaticFactory* **StandardFunction.**

Uso del pattern StaticFactory:

La scelta della *IndicatorFunction* da utilizzare è stata legata alla difficoltà scelta dal giocatore, infatti ogni livello di difficoltà ha una strategia di calcolo commensurata, per poter ottenere queste diverse funzioni è stato utilizzato il pattern ***StaticFactory***. La classe *StandardFunction* ha infatti un metodo statico che ritorna una sua implementazione per ogni livello di difficoltà (es. *getMediumGameComputinFunction()*). (Figura 8)

Uso del pattern Decorator:

Nel nostro mVillage il giocatore può imbattersi anche in eventi eccezionali relazionati alla sua gestione del villaggio, questa feature non è fondamentale per lo svolgimento del gioco, ma aggiunge un tocco di imprevedibilità e "realismo" alla game experience, visto però come questo aspetto sia per lo più "decorativo" ho pensato di realizzare due versioni di *ResourceManager*, una con il supporto agli eventi e l'altra no. Visto che le due classi avrebbero differito soltanto per questo aspetto, ho pensato che il pattern ***Decorator*** sarebbe stato perfetto per soddisfare questa esigenza, aggiungendo così dinamicamente il supporto agii eventi al "semplice" *ResourceManagerImpl*. In questo caso la classe astratta *ManagerDecorator*wrappa un *ResourceManagerImpl* ed implementa anche esso l'interfaccia *ResourceManager*, delegando quasi tutti i metodi all'oggetto wrappato. Il decoratore concreto

SpecialResourceManager estende la classe astratta e implementa quei metodi che permetteranno di aggiungere la nuova funzionalità(Figura 9). L'applicazione di questo pattern rende quindi facilissimo scegliere anche se il gioco supporti gli eventi oppure no, attraverso un meccanismo molto flessibile e che favorisce il riuso di codice, andando a modificare solo il contorno di una classe e non tutta la sostanza.

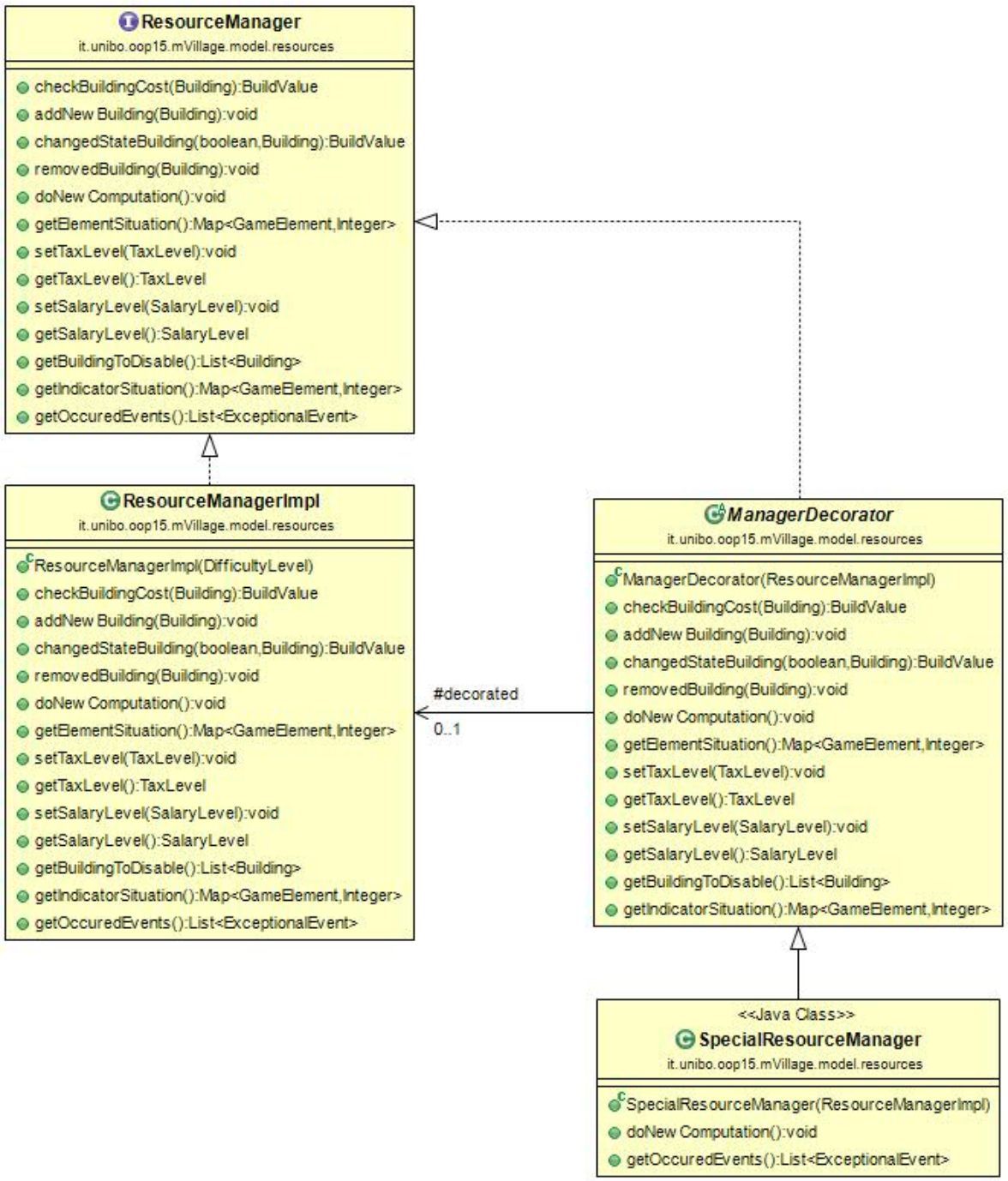

Figura 9
*Schema UML esplicativo dell'applicazione del pattern **Decorator**. La classe da **ManagerDecorator** funge da decoratore astratto,esso infatti wrappa un oggetto **ResourceManagerImpl** ed implementa l'interfaccia **ResourceManager, SpecialResourceManager** è invece il decoratore concreto che estende **ManagerDecorator** e ridefinisce alcuni suoi metodi.*

Gestione degli eventi eccezionali:

Vale la pena fare un veloce excursus su come sono stati gestiti questi eventi in pratica, essi sono stati legati infatti ai vari *IndicatorEntity* (un Enumerazione di indicatori sociali) presenti nel gioco, infatti ogni *IndicatorEntity* contiene un *EventBehaviour* correlato, che descrive se e come avvengono gli *ExceptionalEvent*(altra enumerazione di eventi eccezionali) specifici dell'indicatore. Le varie istanze delle classi della famiglia degli *EventBehaviour* sono fornite da una utility class composta da soli metodi statici, motivo per cui questa classe non ha senso che sia istanziabile, il costruttore è stato così reso privato per rafforzare la sua non istanziabilità (***"Enforce non instatiabilitywith a private constructor"*** [4])(Figura 10).

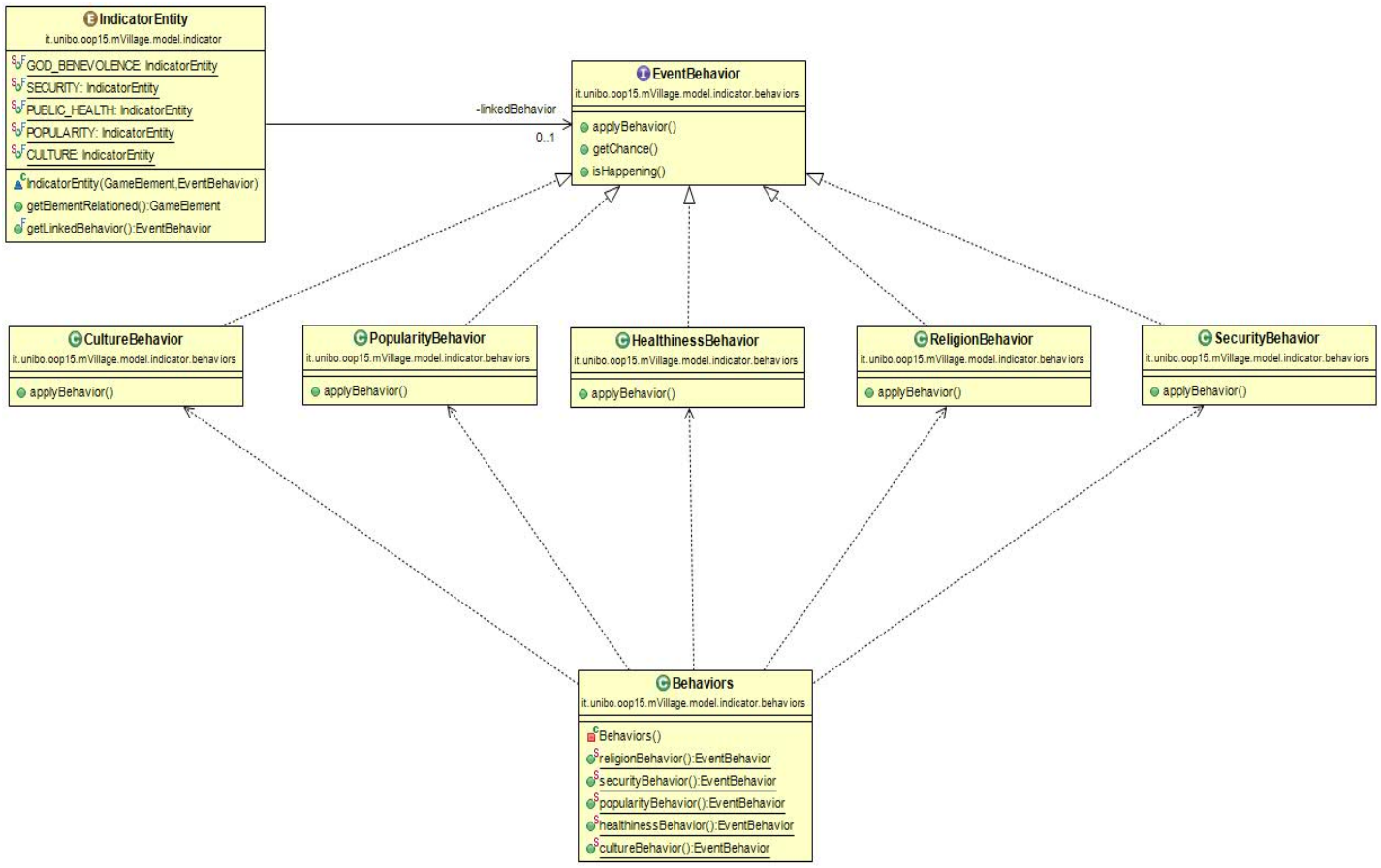

Figura 10
*Ogni **IndicatorEntity** ha un oggetto della famiglia degli **EventBehaviour** che gestisce gli eventi eccezionali appropriati, questi oggetti vengono forniti dalla utility class **Behaviours**.*

Considerazioni generali sul design del model:

Per inciso nel progetto si fa spesso uso dell'enum *GameElement*, essa comprende tutti gli indicatori sociali, tutte le materie prime ed i tipi di popolazione. Gli elementi di questa

[4] Item 4- Effective Java - Second Edition- J.Bloch

enumerazione sono tutti elementi prodotti/forniti/influenzati dalle varie entità del gioco, la definizione di questa enum è un'astrazione che ha facilitato molto la stesura del model, per maggiori dettagli relativi alla sua utilizzazione si veda il codice sorgente.

Come si evince navigando nei package del model, sono state utilizzate molte enumerazioni, questo tipo è perfetto per i nostri scopi, offrendoci la possibilità di definire un numero determinato di elementi aventi caratteristiche comuni, donando grandissima estendibilità al progetto, in qualsiasi momento potremmo infatti aggiungere nuovi edifici, terreni, materie prime, livelli di tasse o di salari, livelli di difficoltà, con il minimo sforzo, aggiungendo solo ciò che è relativo all'inserimento di un nuovo elemento in una enum. Mi preme fare un esempio pratico della reale estendibilità del progetto; Ad un certo punto dello sviluppo della mia parte, con gli altri compagni è emersa la volontà di inserire anche una locanda per allietare i popolani e che quindi consumi birra, prodotta a sua volta da un birrificio che consuma luppolo, a sua volta colto negli specifici campi. E' evidente come potrebbe non essere banale aggiungere tre elementi così interlacciati tra loro; Avevo però già descritto l'enum *SecondaryBuilding*, che elenca gli edifici che per espletare la propria funzione consumano un determinato elemento, è bastato così aggiungere alla lista gli edifici questi nuovi tre, mentre ai *SecondaryBuilding* il birrificio e la locanda e il gioco era fatto, non è stato necessario dover modificare nessun aspetto implementativo del model, pur avendo aggiunto dei nuovi elementi non banali. Il buon grado di disaccoppiamento fra le varie entità del Model hanno poi favorito una maggior facilità nell'individuare e correggere alcuni bug emersi in fase di testing.

Suddivisione in package:

Il package *it.unibo.oop15.model* è stato suddiviso in sotto packages coerentemente con l'architettura descritta finora:

- **it.unibo.oop15.mVillage.model.indicator:** Contiene ciò che concerne gli indicatori (*IndicatorEntity*) e gli *ExceptionalEvent* collegati.
  A sua volta suddiviso in:
  - **it.unibo.oop15.mVillage.model.indicator.behaviors:** Contiene i vari *Behaviour* relativi ad ogni *IndicatorEntity*.
  - **it.unibo.oop15.mVillage.model.indicator.computingFunction:** Contiene le funzioni per calcolare i valori degli indicatori.
- **it.unibo.oop15.mVillage.model.principalElement:** Contiene tutti gli elementi principali del dominio, come *Building*, *Field*, *DifficultyLevel*, ecc…

- **it.unibo.oop15.mVillage.model.resources:** Contiene tutti i componenti relativi alla gestione delle risorse umane e materiali.
- **it.unibo.oop15.mVillage.model.test:** Contiene il test delle funzionalità del model.

View- *Luca Passeri*

Rientrando nel caso del design architetturale MVC la View espone graficamente lo stato del Model e consente l'interazione con gli utenti; in questo caso inoltre si occupa della gestione dell'audio all'interno dell'applicazione.
La scelta della libreria grafica è ricaduta su Swing al fine di sfruttare gli insegnamenti del corso e inoltre per il fatto che questo gioco ben si presta alla modellazione con componenti di tale libreria, non necessitando (a differenza di numerosi giochi) di grande dinamicità.

La descrizione di questa sezione è stata suddivisa in alcuni punti principali che rispecchiano in maniera abbastanza fedele la suddivisione della stessa nei package.
A sua volta la distribuzione dei file all'interno dei package è stata svolta in maniera tale da massimizzare la coerenza del contenuto degli oggetti rapportati al nome del package interessato.
Più precisamente troviamo:

- **it.unibo.oop15.mVillage.view.basicViews**: contiene l'interfaccia base di ogni schermata, una sua standard implementazione e le prime tre schermate di gioco.
- **it.unibo.oop15.mVillage.view.gameView**: contiene tutti i file relativi alla principale schermata di gioco, vi è poi un sotto-package specifico per le sezioni del tab principale di questa schermata (verrà chiarito in seguito).
- **it.unibo.oop15.mVillage.view.customComponents**: contiene tutti i componenti personalizzati.
- **it.unibo.oop15.mVillage.view.utilities**: contiene alcune utility-class per la View.

Basic Views

La gestione delle schermate ha determinato un primo importante scheletro della View; come si può intuire dalla (Figura 11), infatti, ogni schermata risiede su un proprio frame e presenta i principali metodi per gestire tale schermata. Una valida alternativa sarebbe stata utilizzare

un unico frame ed eseguire un redesign di esso al variare delle situazioni ma, considerando che le schermate totali sono 4, di cui 2 immutabili e una potenzialmente tale, ho deciso di utilizzare più frame favorendo anche la velocità di visualizzazione (le prime 3 schermate infatti non vengono chiuse, bensì rimangono invisibili).
A questo punto una possibile scelta sarebbe stata quella di far sì che le varie schermate estendessero la classe JFrame, ma ciò avrebbe comportato una forte limitazione, poiché per poter poi accedere ai metodi di quest'ultima classe avrei dovuto dichiarare tali oggetti tramite la classe e non usando un'interfaccia, cosa che anche Josh Bloch afferma dover essere evitata (***"Refer to objects by their interfaces"***[5]).

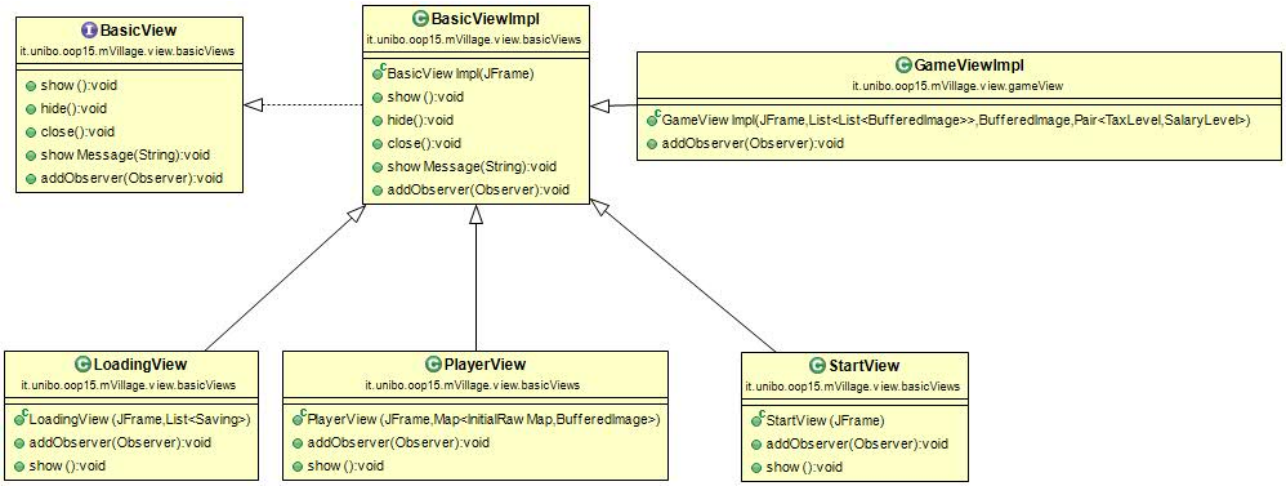

Figura 11

Come accennato precedentemente esiste un'interfaccia comune a tutte le schermate, BasicView, che viene implementata da una classe BasicViewImpl, la quale istanzia il frame e definisce tutti i metodi principali di ogni sotto-view eccetto il metodo "addObserver"; perciò ogni schermata estende BasicViewImpl.

Tale costruzione ha reso possibile evitare la ripetizione di codice e l'identificazione di più oggetti tramite un'interfaccia comune.

Aprendo il gioco, verrà mostrata per prima la schermata iniziale, nella quale è possibile scegliere l'opzione di avviare una nuova partita o di caricarne una; in entrambi i casi comparirà un'altra interfaccia antecedente a quella di gioco:

- nel caso di "New Game" sarà possibile impostare le caratteristiche del gioco (mappa e difficoltà) e il vostro nome che vi accompagnerà durante il gioco
- nel caso di "Load Game" sarà possibile scegliere uno dei salvataggi (se presenti) e riprendere da dove è stato salvato

[5]Item 52 -Effective Java - Second Edition - J.Bloch

Queste prime schermate non necessitano di particolari organizzazioni in quanto le funzioni che offrono sono minimali; ben diverso invece è il discorso riguardante la schermata di gioco vera e propria, parte fondamentale della View di questa applicazione.

Game View

La corposità di quest'ultima sezione mi ha portato a pensare di scorporare la stessa in più parti e ho adottato la soluzione che dal mio punto di vista attribuisse più autonomia possibile ad ogni sotto-sezione e comportasse più equilibrio possibile; ho scelto infatti di suddividerla per componenti principali.
Più precisamente la suddivisione non è avvenuta scorporando i veri e propri componenti grafici ma utilizzando classi che contengono essi, le quali, se necessario, implementano interfacce che offrono metodi per interagire con questi ed altri componenti, altrimenti direttamente un'interfaccia comune a tutte le sotto-parti (BasicGameView).
In Figura 12 viene mostrata la composizione delle sezioni.

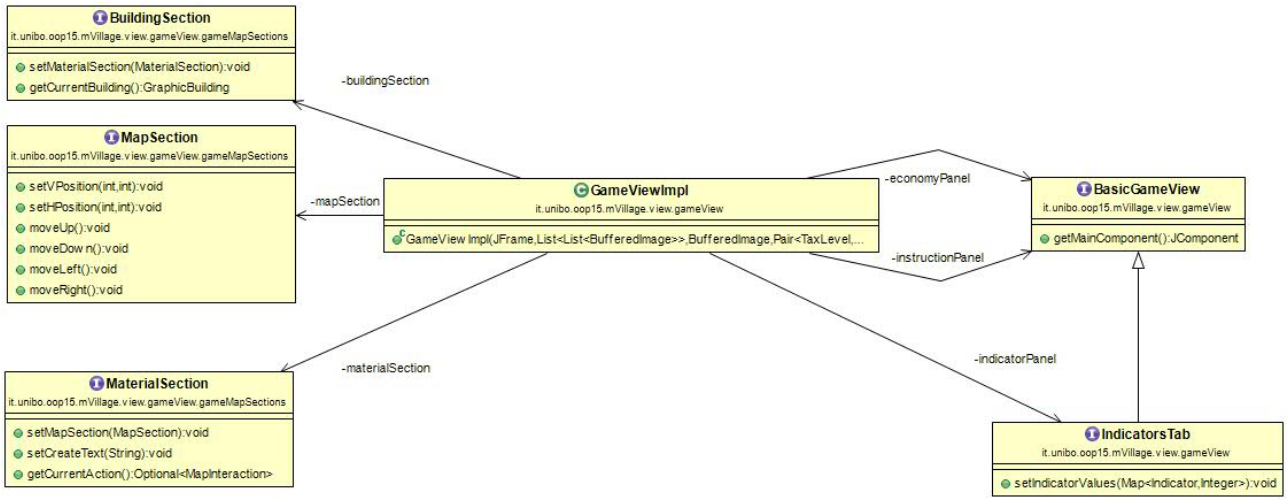

Figura 12
In questa figura viene mostrata la suddivisione della schermata di gioco in più sezioni:

GameViewImpl presenta come campi a sinistra i 3 oggetti che contengono i pannelli che insieme costituiscono il tab principale (Figura 14), contenete la mappa di gioco e le funzioni principali per interagire con essa. Mentre a destra i 3 oggetti che contengono i pannelli riguardanti gli altri tab, Indicators, Economy e Instructions (Figura 13).

Uno zoom di tale suddivisione viene mostrato nelle seguenti due immagini.

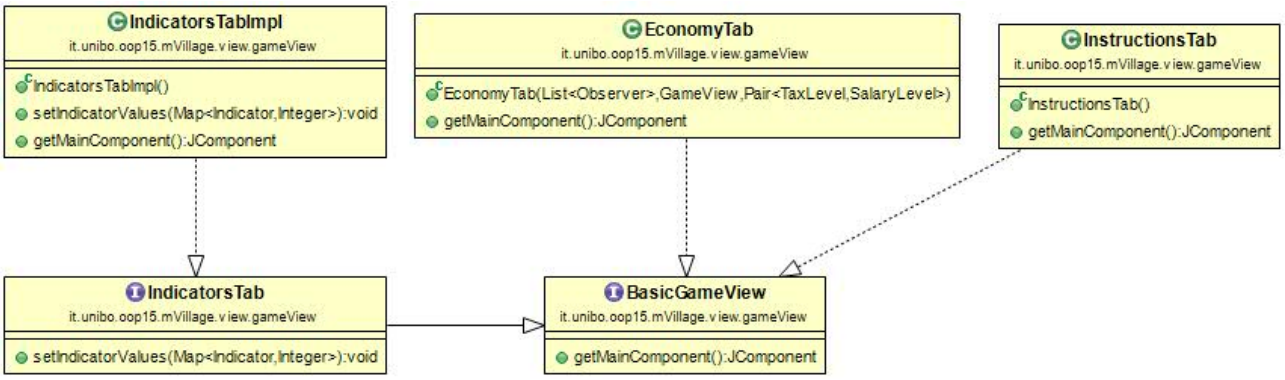

Figura 13

Come già accennato ogni classe implementa direttamente o indirettamente BasicGameView,

le classi EconomyTab e InstructionsTab non necessitano di metodi addizionali a quelli di quest'interfaccia poiché il Model non vi può interagire, mentre quest'ultimo può intervenire sugli indicatori tramite IndicatorsTabImpl, il quale implementa indirettamente BasicGameView passando per l'interfaccia IndicatorTab che aggiunge gli ulteriori metodi necessari.

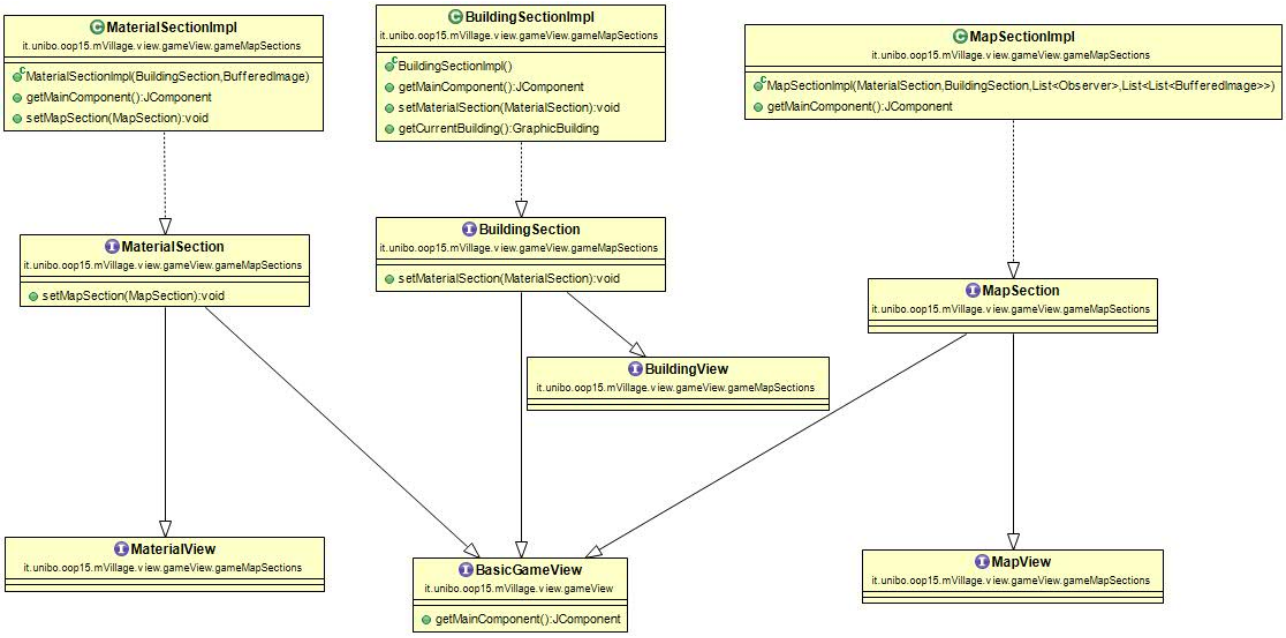

Figura 14

Tale architettura è frutto del massimo incapsulamento possibile per le sotto-parti del tab principale:

per la costruzione di ogni sezione servivano infatti dei metodi fini al gioco, che devono essere perciò visibili dall'esterno, mentre altri volti esclusivamente a collegare le sezioni tra di loro.

Perciò, per evitare di rendere visibili metodi non solo superflui ma potenzialmente pericolosi per la view ho deciso di utilizzare due interfacce per ogni sezione. In particolare le interfacce il cui nome termina per "view" indicano l'oggetto che ha come contratto tutto ciò che deve essere visto dall'esterno, mentre quelle il cui nome termina per "section" il contratto di quelle appena descritte più i metodi necessari alla sua costruzione.

Riassumendo quindi, per quanto riguarda il tab principale, GameViewImpl ha come campi MapSection, BuildingSection e MaterialSection al fine di crearle ma essa estende MapView, BuildingView e MaterialView (Figura 15).

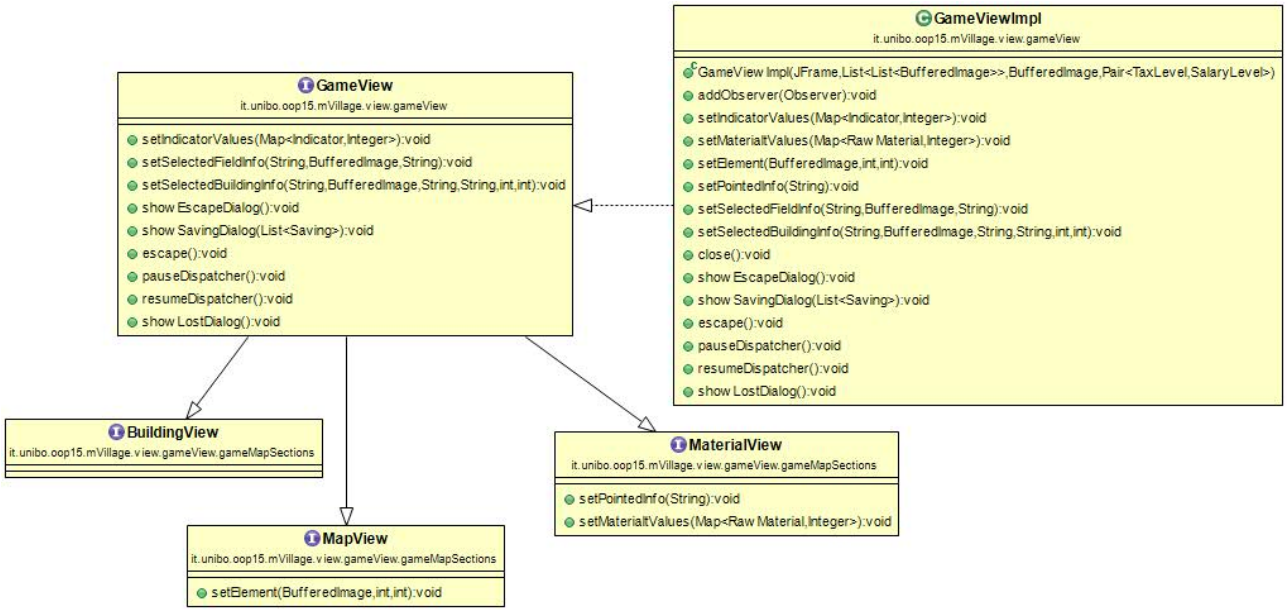

Figura 15

Tale figura mostra l'organizzazione di GameView effettuando uno zoom sul tab della mappa di gioco:

GameView infatti estende MapView, BuildingView e MaterialView i quali contengono esclusivamente i metodi visibili al Model (ovviamente tramite il Controller) evitando così ripetizione di codice, e in particolare ridefinisce i metodi delle classi estese delegando ad esse il compito di svolgere l'operazione richiesta.

Custom Components

Una parte molto significativa è quella riguardante la modellazione dei componenti: ogni componente infatti è stato utilizzato indirettamente tramite una classe che lo estende.
Ciò ha permesso di creare oggetti di due tipi:

- componenti standard che vengono utilizzati molto frequentemente, evitando così ripetizione di codice
- componenti custom in cui è possibile effettuare rapidamente dall'esterno operazioni che altrimenti avrebbero richiesto più passaggi.

Questi oggetti, combinati con il pattern Builder (che in seguito verrà illustrato) hanno conferito maggior comodità durante la programmazione ed eleganza alla View.
In alcuni casi inoltre, visto la quantità di componenti simili creati, ho usufruito di alcune utility-class in due differenti modi:

- con metodi per ottenere tutti i componenti di quel tipo; essendo questi ultimi con visibilità package private questa classe statica è l'unico modo per costruirli
- con metodi per ottenere un particolare componente con differenti caratteristiche

Rimanendo all'interno dei componenti, un argomento un po' più nel dettaglio sul quale ho però voluto porre attenzione riguarda i JDialog.
Questo è un gioco nella quale la comunicazione con l'utente è fondamentale e, a volte, molto frequente, motivo per cui ho deciso di utilizzare questi componenti.
I JDialog infatti gestiscono in maniera efficiente e comoda il focus e la loro disposizione all'interno del frame padre e il loro utilizzo rimane intuitivo; tuttavia presentano un problema non indifferente:
essi infatti vengono istanziati solo dopo esser stati chiusi.
Non potendo interagire con essi mentre sono aperti perciò ho dovuto affidare ad essi stessi compiti che (senza tale problema) avrei preferito evitare: ad esempio quando si intende chiudere un JDialog l'unico modo per farlo è gestire tale evento all'interno del componente stesso, non permettendo così di effettuare questa operazione dall'esterno.
Il funzionamento del gioco non è assolutamente compromesso e l'impatto che questo imprevisto ha avuto sull'eleganza e incapsulamento del codice è risultato comunque minimo.

KeyEventDispatcher

Per quanto riguarda il movimento all'interno della mappa ho deciso di offrire come possibilità anche quella di usare i comandi WASD, trovando scomodo utilizzare solamente le barre offerte dal JScrollPane.
Per poter permettere di accedere a questa funzionalità indipendentemente dal componente che in quel momento possiede il focus ho trovato molto pratico ridefinire un mio personale KeyEventDispatcher aggiungendo metodi necessari. GameKeyEventDispatcherImpl inoltre, per consentire di gestire più comandi contemporaneamente, contiene un Thread che memorizza e gestisce tali eventi tramite un set.

Pattern Utilizzati

Per intrinseca conformazione della View, e probabilmente per mia limitata conoscenza, ho trovato utile nella realizzazione di questa parte solamente il pattern Builder.

Sebbene sia l'unico pattern utilizzato il suo utilizzo è stato intensivo viste le sue proprietà: esso ha consentito infatti di realizzare componenti con parametri opzionali evitando la proliferazione di costruttori; un esempio significativo viene illustrato in Figura 16.

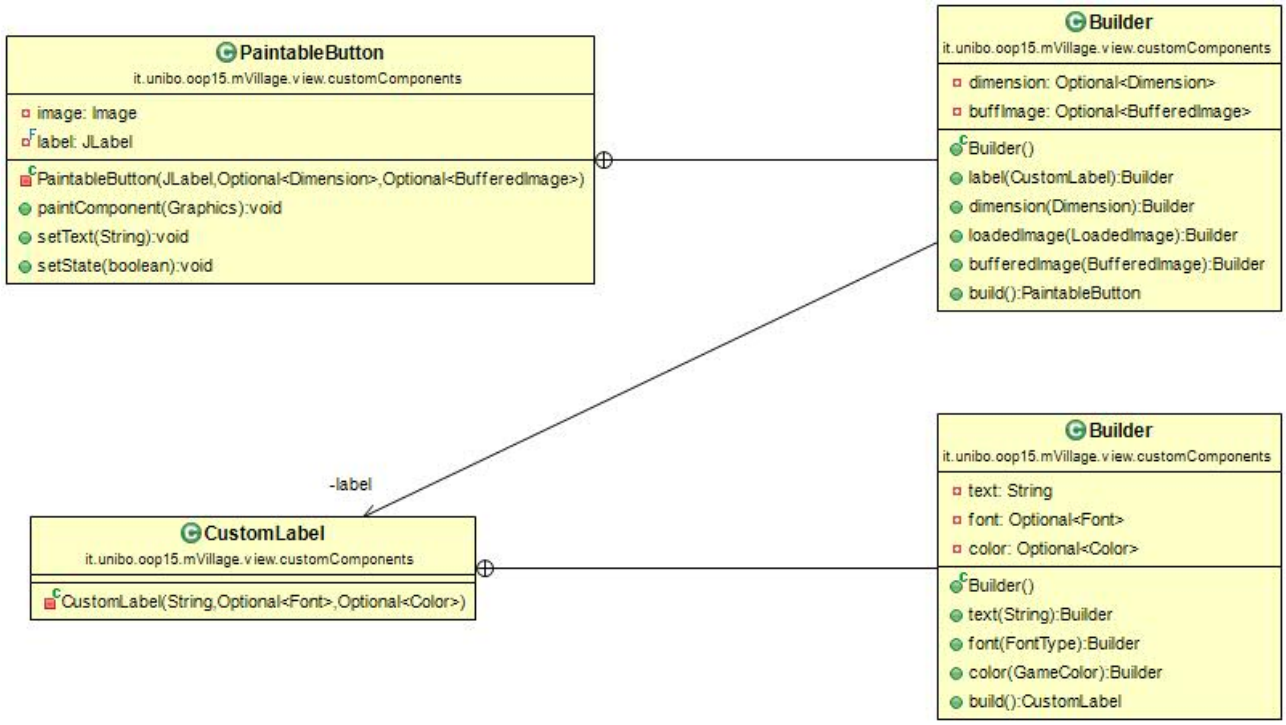

Figura 16

Questa figura mostra uno dei casi dell'utilizzo del pattern Builder.

In questo esempio viene utilizzato un Builder all'interno di un altro: vi è un PaintableButton (che estende JButton) che ha come parametri impostabili l'immagine, la dimensione e una CustomLabel che è obbligatoria. La CustomLabel (che estende JLabel) a sua volta viene costruita utilizzando un Builder avente come parametri il testo (obbligatorio), il font e il colore.

Senza questo pattern avrei dovuto definire numerosissimi costruttori con differenti parametri o limitare i parametri impostabili di tali componenti.

Controller – *Jacopo Mastrogirolamo*

Come da patter MVC, il controller, agisce sia su model che su view andando a gestire i dati presenti all'interno della definizione del model stesso ed aggiornando la view quando questi dati subiscono delle modifiche.

Il controller, utilizzando il pattern Observer, osserva constantemente la view in modo da recepire gli input da parte dell'utente. Al fine di gestire in maniera efficacie le varie schermate della view e settarvi l'observer dopo che le stesse siano state inizializzate, si è creata l'interfaccia ViewController. Quest'ultima, tramite l'enumerazione *TypeView*, mette in pratica l'alternarsi delle corrette schermate di gioco.

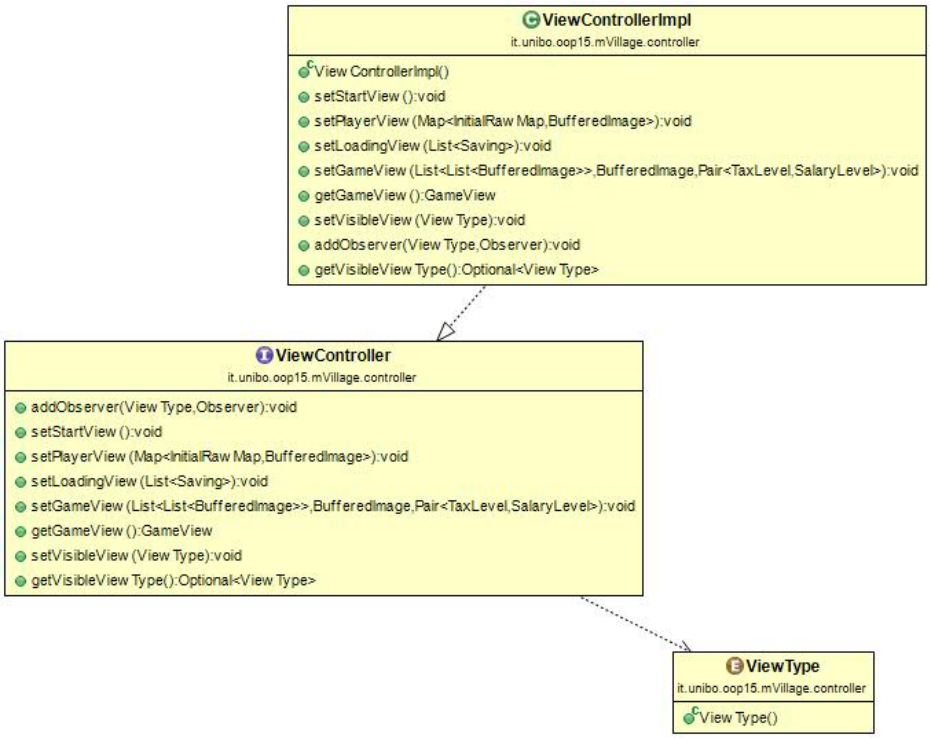

Figura 17
*Schema Uml che mostra i metodi dell'interfaccia **ViewController** che trova la sua implementazione nella classe **ViewControllerImpl**. Fondamentale il metodo **addObserver** che permette al Controller di monitorare la view e gestire l'alternarsi delle schermate, rappresentate idealmente dall'enum **ViewType**.*

Utilities e Enum per interazione con view

Al fine di gestire la corretta comunicazione tra controller e view, soprattutto per quel che riguarda gli elementi di gioco, si sono utilizzate diverse enum aventi la funzione di creare un linguaggio comune che permetta agli stessi di comunicare e mettere in pratica il pattern Observer. Enum quali *GraphicBuilding* o *InitialRawMap* contengono descrizioni degli elementi di gioco necessarie alla view per una corretta esperienza di gioco. Le stesse enum vengono utilizzate nella classe di utilità statica *MapUtility* che contiene al suo interno mappe

che, a determinate enum rappresentanti elementi di gioco, associano la relativa immagine : il tutto è necessario per ridurre i tempi di caricamento delle immagini utilizzate nella view e migliorare in maniera drastica i tempi di reattività dell'applicazione in generale. Il caricamento di immagini e font avviene utilizzando i metodi della classe *StaticLoader*.

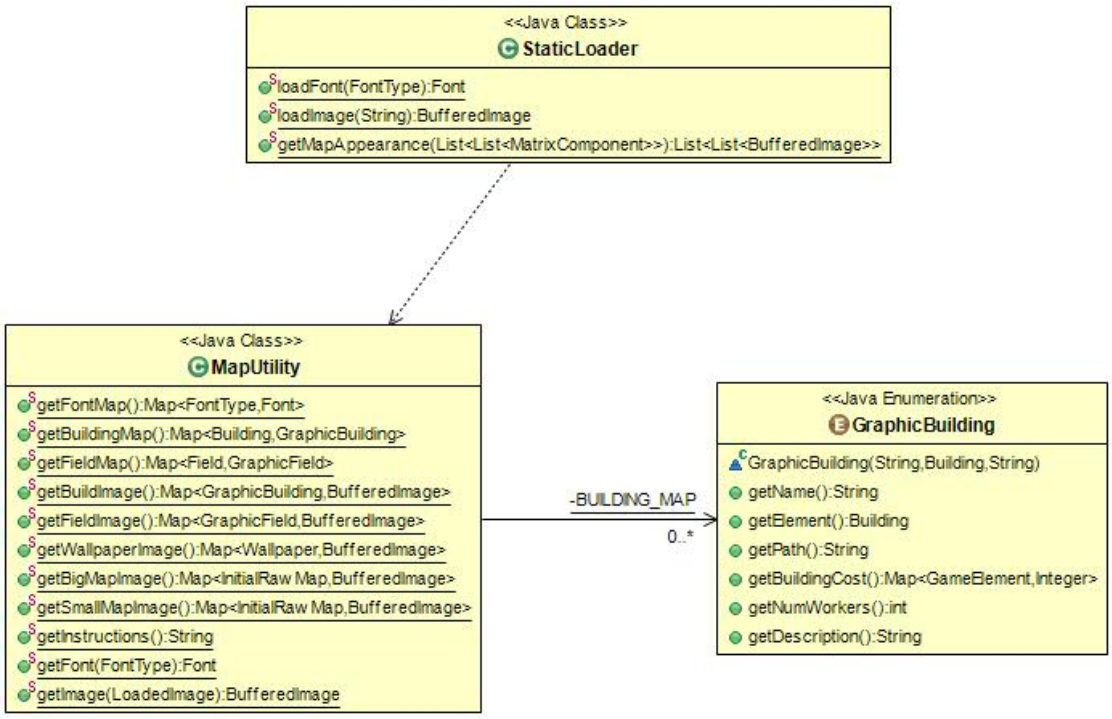

Figura 18
*Visione d'insieme delle classi di utilità statica. I metodi **loadFont** e **loadImage** di **StaticLoader** vengono utilizzati nella creazione di tutte quelle mappe che, ad elementi di gioco (qui in rappresentanza l'enum **GraphicBuilding** che definisce alcune caratteristiche degli edifici presenti nella mappa di gioco), associano la relativa immagine tramite classe **MapUtility**.*

Saving

L'interfaccia *Save* si occupa della gestione dei salvataggi relativi alle partite di gioco. Questi vengono memorizzati all'interno di una cartella locale alla home dell'utente creata all'avvio dell'applicazione. I metodi principali della classe *Save*, *loadGame() e saveGame()*, lavorano in stretta collaborazione con la classe *GameAttributes*. Questa classe presenta al suo interno il pattern *Builder*: questo è stato utilizzato per costruire passo passo un oggetto complesso che ricostruisca le condizioni di gioco da salvare o ricaricare. L'utilizzo di questo pattern permette e favorisce l'aggiunta di future condizioni di gioco delle quali è necessario tenere traccia: basterà modificare la classe *GameAttributes* e di conseguenza avremo la creazione di un oggetto che rispecchierà le modifiche fatte.

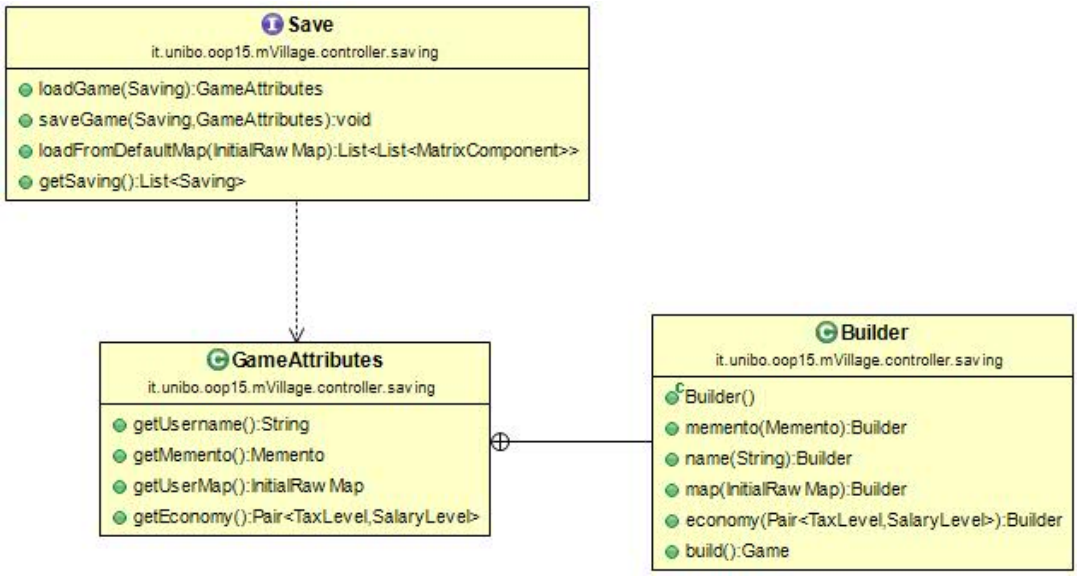

Figura 19
*L'interfaccia **Save** restituisce un oggetto di tipo **GameAttributes**, che rappresenta tutti quegli elementi di gioco che concorrono al corretto funzionamento dell'applicazione stessa, o ne riceve uno come argomento per ricavarne i medesimi elementi rispettivamente nei metodi **loadGame** e **saveGame**.Come si evince dall'Uml, **GameAttributes**, si compone di un **Builder** che ne definisce le proprietà intrinseche.*

MedievalThread

L'interfaccia *MedievalThread*, rappresenta il "cuore" pulsante dell'applicazione. Questa estende la classe Thread e si occupa appunto di creare un thread il cui compito è quello di, ogni lasso di tempo determinato come politica interna alla classe stessa e tramite il metodo *updateSituation()* di *Observer*, aggiornare i dati relativi al dominio del model e modificare di conseguenza la view. Importante l'utilizzo della variabile di mutua esclusione *Lock* per permettere all'utente di mettere in pausa il gioco senza che il thread continui ad aggiornare i dati dello stesso.

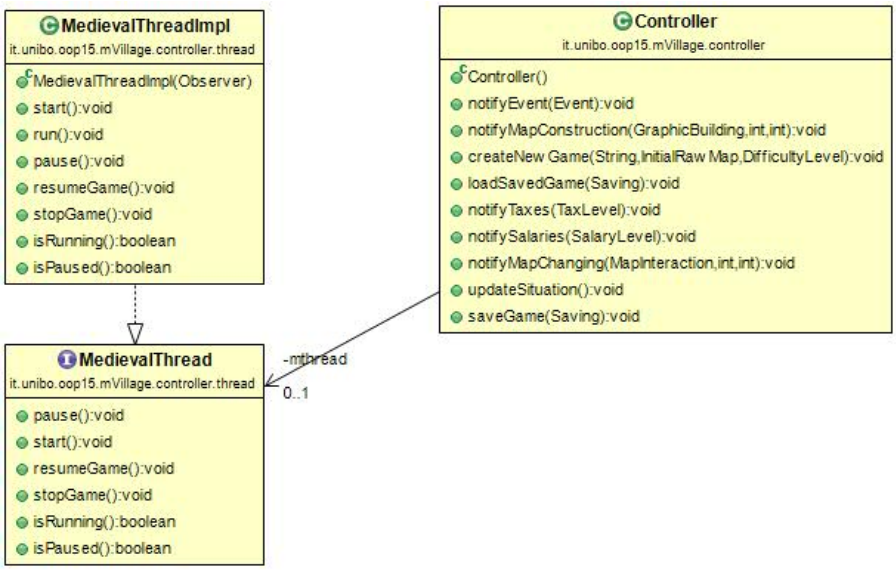

Figura 20
***Controller** utilizza, instanziandone il riferimento, la classe **MedievalThreadImpl** che a sua volta, attraverso la politica interna definita, richiama ciclicamente il metodo **updateSituation()**.*

Observer

L' interfaccia *Observer*, che trova la sua implementazione nella classe *Controller*, definisce i metodi necessari ad una corretta gestione di model e view, inizializzati all'interno della classe stessa. Vengono inoltre inizializzate anche tutte le referenze alle utility-class per la gestione degli audio e al thread principale. Ogni classe che volesse implementare la funzione di controller dovrebbe seguire l'interfaccia *Observer*, costituente una linea guida, i cui metodi più specifici quali ad esempio *updateSituation()* possono essere definiti a piacemento per definire la politica di gestione e comportamento generale dell'applicazione. Altri metodi fondamentali sono la *notifyEvent,* che ricevendo come argomento un particolare evento, definito dall'enum *Event*, va ad aggiornare la view utilizzando *ViewController,* oppure la *notifyMapConstruction,* metodo fondamentale in un RTS game dove la costruzione di elementi costituisce la parte centrale del gioco. Questo metodo, ricevendo come argomento l'elemento che si vuole costruire e le coordinate che definiscono dove costruirlo, interroga il model per conoscere se è possibile effettuare l'azione desiderata andando successivamente ad aggiornare la view secondo quando ricevuto come output dal model. Il tutto costituisce dunque un elemento chiave nella politica di interfacciamento MVC che vuole mantenere completamente "sconosciuti "model e view. Gli altri metodi, visibili nell'UML, si centrano sul salvataggio e caricamento dati, necessari per implementare una politica di salvataggio dati se desiderata come feature dell'applicazione.

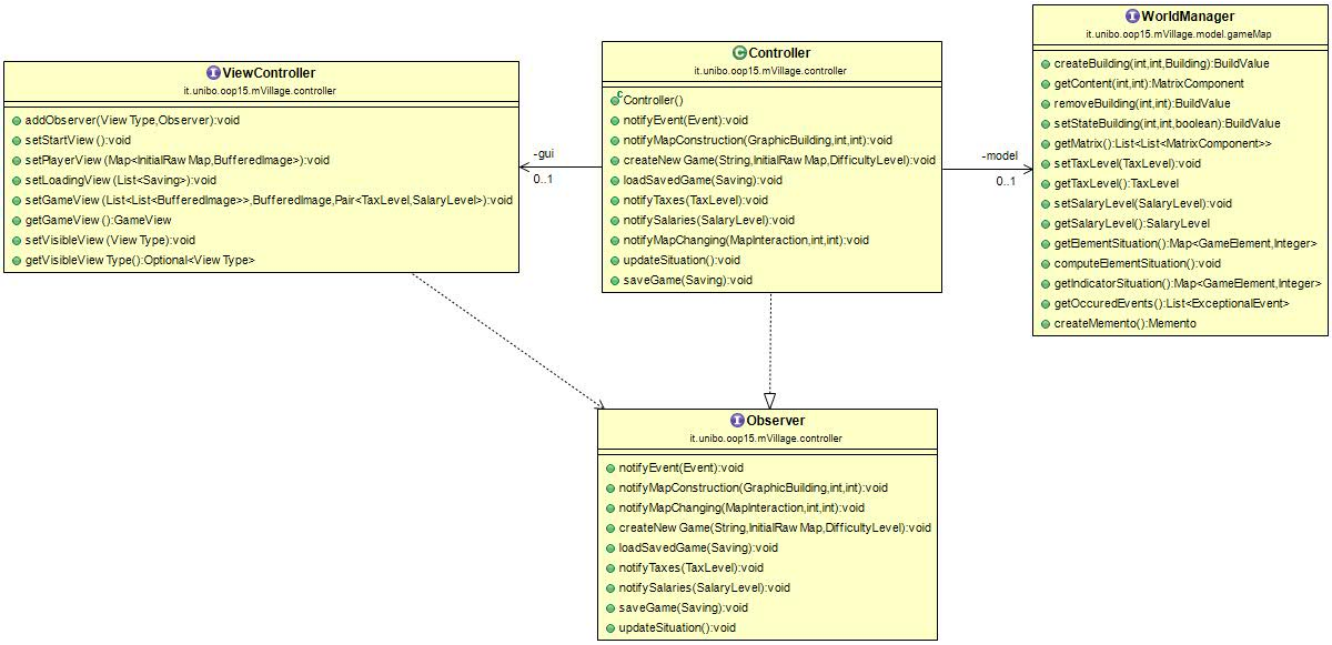

Figura 21
*Visione generale del funzionamento del **Controller** che, in posizione centrale, collega da un lato la **View**, gestita come specificato in precedenza da **ViewController**, e dell'altro il **Model**, la cui complessità viene fornita al controller stesso tramite **WorldManager**.*

Audio Management

Per rendere l'esperienza di gioco più piacevole sono stati aggiunti effetti sonori che accompagnano l'utente durante determinate azioni di gioco. L'interfaccia *Audio* definisce i metodi principali da utilizzare nella gestione degli audio. Questa interfaccia trova la sua realizzazione nella classe *AudioPlayer* che utilizza il pattern *Singleton*, dettato dalla natura pragmaticamente funzionale della classe e per impedire il proliferare di oggetti aventi le stesse identiche funzioni (riprodurre audio). Importanti i metodi *play()*, che definisce con quali parametri (quali ad esempio volume) riprodurre il media, *playOnLoop()* che riproduce la traccia in loop e *playShortSound(),* utilizzato per riprodurre suoni brevi e sporadici. Tutti questi metodi accettano come argomento una *Track* o *ShortTrack*, enum che definiscono il tipo di traccia e la sua posizione all'interno delle risorse di progetto. Infine, la classe *AudioManager*, definisce un thread che, utilizzando una politica di scelta random tra una lista di *Track* correttamente selezionate, riproduce, durante la fase di gioco attiva da parte dell'utente, degli audio a tema medievale in sequenza. L'utilizzo di variabile di mutua esclusione quali la *Lock*, permettono al thread di gestione degli audio di essere messo in pausa proprio come *MedievalThread*.

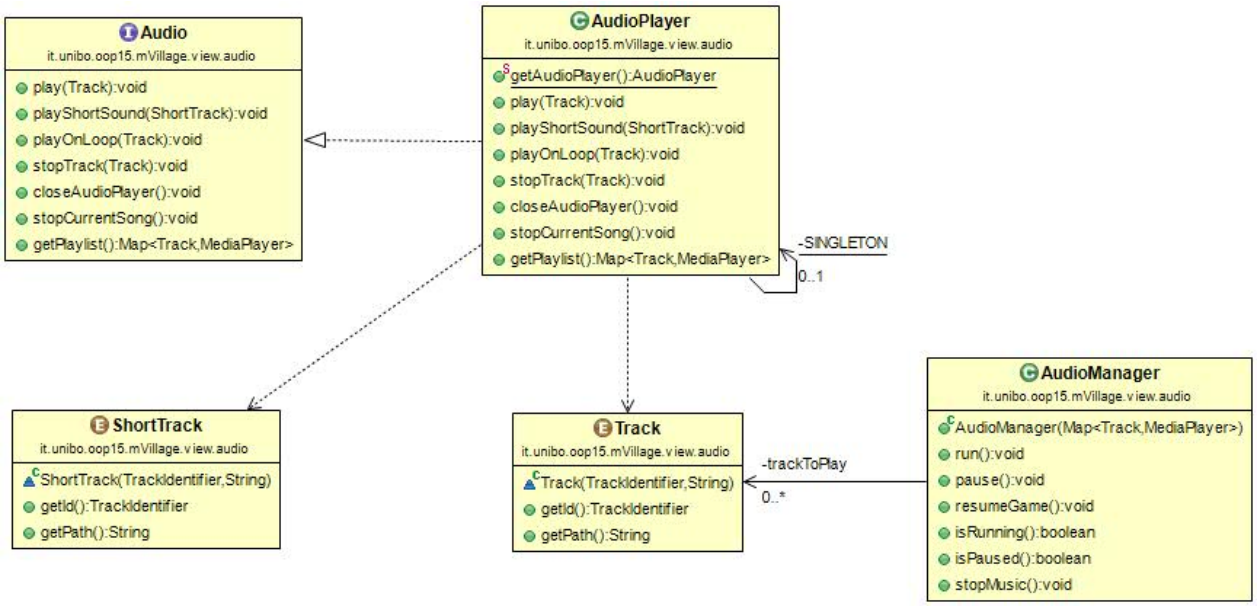

Figura 22
*Uml che mostra l'interazione tra i vari componenti volti alla gestione degli audio. L'interfaccia **Audio**, che definisce i metodi principali per la gestione di **Track** e **ShortTrack** (enum che rappresentano e definiscono le caratteristiche dei vari suoni), fornisce anche la playlist, tramite metodo **getPlaylist**, che la classe **AudioManager** andrà ad utilizzare nella definizione del suo metodo **run()**.*

Suddivisione in package:

Il package contenente le classi relative al controller è stato suddiviso in sotto packages al fine di ordinare e mantenere coerente la logica descritta in precedenza:

- **it.unibo.oop15.mVillage.controller:** contiene le classi volte alla gestione di model e view.
- **it.unibo.oop15.mVillage.controller.saving:** contiene le implementazioni necessarie ad una corretta gestione della politica di salvataggio e caricamento dati.
- **it.unibo.oop15.mVillage.controller.thread:** troviamo la definizione della politica di aggiornamento dei dati dell'applicazione.
- **it.unibo.oop15.mVillage.controller.utilities:** presenta classi di utilità con metodi prettamente statici.
- **it.unibo.oop15.mVillage.controller.viewInteractionEnum:** contiene diverse enum volta ad una corretta interazione view/controller.

3 Sviluppo

3.1 Testing Automatizzato

Per il testing della parte di model è stata utilizzata la suite JUnit, il codice dei test realizzato si trova nella classe TestModel del package **it.unibo.oop15.mVillage.model.test ;** Il test è abbastanza prolisso in quanto era di fondamentale importanza assicurarsi del corretto funzionamento di ogni funzionalità prima ancora di interfacciare il model con le altre parti, infatti grazie a questi test sono emersi alcuni bug in situazioni particolari, risolti senza problemi, ma che con un testing esclusivamente di tipo manuale difficilmente sarebbero emersi; Tutte le componenti funzionalmente più importanti sono state testate, dapprima isolatamente e poi in quelle che sarebbero state le condizioni di effettivo utilizzo.

Classi testate:

- **it.unibo.oop15.mVillage.model.resources.GeneralContainer:** è la classe cha fattorizza tutte le funzioni base che deve offrire un contenitore generico, in quanto base per gli altri contenitore, è stata la prima ad essere creata e testata, in modo da avere una base solida su cui sviluppare altri componenti.
- **it.unibo.oop15.mVillage.model.resources.GranaryImpl**: è la classe che gestisce il cibo, che quindi computa la produzione, il consumo e gestisce l'immagazzinamento. Sono stati testate oltre alle funzionalità base anche il comportamento nei casi limite, come ad esempio quando il cibo non è sufficiente per tutti.
- **it.unibo.oop15.mVillage.model.resources.PopolationHandlerImpl:** di questa classe sono state testate le funzionalità di base relative alla gestione delle nascite, della trasformazione di popolani disoccupati in lavoratori e viceversa.
- **it.unibo.oop15.mVillage.model.resources.RawWarehouseImpl:** sono state testate brevemente le funzionalità che vengono quasi totalmente demandate al *GeneralContainer*, mentre in maniera più intensiva è stato verificato la corretta gestione degli edifici secondari, cioè quegli edifici che per produrre hanno bisogno di un'altra materia prima, nel caso del test si è presa in considerazione la produzione della birra (eseguita dal birrificio, consumando luppolo),controllando che l'edificio fosse disattivato in caso di mancanza di luppolo.
- **it.unibo.oop15.mVillage.model.resources.TreasuryImpl :** sono state testate le funzionalità di base quali aumento e diminuzione di una certa quantità d'oro e calcolo della nuova situazione economica al netto delle imposte e dei salari.
- **it.unibo.oop15.mVillage.model.indicator.IndicatorManagerImpl:** è stata testata la corretta computazione dei valori degli indicatori, in questo caso prendendo

in esame l'indicatore *Popolarità*, influenzata da un certo tipo di edifici e dal livello di tasse e salari.

- **it.unibo.oop15.mVillage.model.resources.ResourceManagerImpl:** vista l'importanza della classe sono state testate varie situazioni limite che potevano essere problematiche, come ad esempio la rimozione di case(con relativa diminuzione della popolazione e conseguente disattivazione automatica di alcuni edifici); l'indisponibilità di cibo per i popolani(che provoca la morte degli stessi e di conseguenza la disattivazione degli edifici dove lavoravano); la rimozione di un magazzino dopo averne costruiti almeno due, che provocherà quindi uno scarto delle risorse fino a che il loro ammontare sia compatibile con la capienza dei magazzini disponibili. Sono state poi ovviamente testate anche le altre funzionalità base, meno problematiche ma comunque fondamentali.
- **it.unibo.oop15.mVillage.model.gameMap.WorldManagerImpl:** Sono state testate le funzionalità relative alla gestione della mappa vera e propria, in quanto gli aspetti socio economici sono gestiti dal *ResourceManager*, già precedentemente testato, è stato verificato anche il corretto funzionamento dei metodi per creare/caricare/salvare una partita.

Sono stati poi seguiti alcuni test manuali al fine di verificare il corretto funzionamento dell'interfaccia grafica su sistemi operativi diversi, quali Windows XP, Windows 8.1, Windows 10, Linux Mint 17.2.

3.2 Metodologia Di Lavoro

La suddivisione del lavoro è avvenuta quasi completamente considerando le tre parti costitutive del pattern architetturale MVC (Pergolini: Model, Passeri: View, Mastrogirolamo: Controller) fatta eccezione della parte di gestione degli audio che, per equilibrare il carico di lavoro, è stata attribuita a Mastrogirolamo insieme alla creazione delle mappe di gioco.

Il DVCS è risultato molto utile per massimizzare la collaborazione e facilitare il lavoro di gruppo, è stata un'esperienza significativa che ci ha permesso di scoprire una tecnica avanzata di cooperazione al fine di realizzare un unico software lasciando però possibilità di indipendenza dei singoli membri all'interno del proprio settore.

Più in particolare in fase di analisi abbiamo definito le interfacce base contenenti i metodi che dovevano essere chiamati da altri componenti, in seguito le parti di implementazione vera e propria sono state sviluppate in autonomia e pubblicate una volta ottenuto un risultato soddisfacente.

Diego Pergolini:

Come concordato nella proposta di progetto, ho sviluppato tutta la parte del progetto inclusa nel package ***it.unibo.oop15.mVillage.model*** . Con gli altri componenti del gruppo abbiamo da subito concordato le funzionalità base che il dominio del gioco doveva garantire, tutte incluse nell'interfaccia *WorldManager*, in modo da garantire un alto grado di indipendenza fra le parti; Alcuni elementi principali del dominio e necessari anche alle altre parti sono stati inseriti nel package **it.unibo.oop15.mVillage.model.principalElement** (Qui vengono definiti, con delle enum, i vari elementi del gioco,come gli edifici, i terreni, le materie prime, i livelli di difficoltà, di tasse, di salari).Ho cercato di implementare il model nascondendo il più possibile i dettagli implementativi all'utilizzatore, fornendo solo un insieme di funzionalità, non permettendo alcuna modifica o interazione con i dettagli interni, questa scelta mi ha consentito di sviluppare in maniera totalmente indipendente la mia parte. Ho aggiunto alcune funzionalità opzionali in un secondo momento, vista la disponibilità di tempo, ma queste modifiche non hanno apportato problemi nell'integrazione fra componenti in quanto l'architettura generale del progetto era stata pensata anche per supportare nuove funzionalità.

Luca Passeri:

Mi sono occupato della parte riguardante la View, come definito nella proposta del progetto, cioè di tutto il codice contenuto all'interno del package ***it.unibo.oop15.mVillage.view*** fatta eccezione della parte degli audio.

La parte più complicata è stata quella di definire a priori le interfacce principali con le quali il Controller avrebbe interagito cercando di non fornire elementi superflui; il lavoro svolto assieme al compagno Mastrogirolamo è stato perciò di fondamentale importanza e l'attenzione rivolta alla costruzione di una solida architettura ci ha permesso di non dover più apportare strutturali modifiche alle interfacce.

Definite queste interfacce ho poi sviluppato il lavoro in totale autonomia, aggiungendo funzionalità che per la maggior parte delle volte non hanno richiesto alcuna modifica alla parte di interazione con il Controller.

Ho cercato di seguire il più possibile il processo di sviluppo "top-down" con raffinamento successivo partendo dalla definizione di interfacce, specializzate o implementate in seguito da oggetti che sono stati perfezionati nella fase finale.

Jacopo Mastrogirolamo:

Mi sono occupato della realizzazione delle classi contenute all'interno del package controller e, per cercare di equilibrare il carico di lavoro, della gestione e ricerca audio (classi contenute all'interno del package **view.audio**). Ho inoltre creato le mappe di default iniziali (presenti all'interno della cartella **rawMap.binMap** nella risorse di progetto) che l'utente è chiamato a scegliere prima di iniziare una nuova partita. Le mappe si differenziano principalmente per disposizione degli elementi "naturali" di gioco (acqua, terra, montagne, foreste) oppure per caratteristiche particolari quali assenza di elementi (vedi mappa Desert) in modo da stimolare la creatività dell'utente e mettere alla prova le proprie capacità. Ho cercato di implementare il tutto rispettando la logica MVC ed ovviamente il confronto con i miei compagni è stato determinante. Principalmente è stata fondamentale la definizione dell'interfaccia *Observer* con Passeri in modo da definire chiaramente la logica di comunicazione legata alla View.

3.3 Note di sviluppo

Diego Pergolini:

Nella classe *WorldManagerImpl* è stata utilizzata la classe *SerializationUtils* della libreria

Apache Commons Lang, per ritornare una copia difensiva dell'elemento della matrice di gioco che l'utilizzatore ha richiesto, è infatti di fondamentale importanza che nessun'altra classe esterna possa modificare lo stato interno del *WorldManager*. La scelta è ricaduta su questa classe anche al netto dei suoi due principali svantaggi:

- Non è utilizzabile su oggetti non serializzabili… Ma gli oggetti di tipo *MatrixComponentImpl* sono serializzabili.
- È più lenta di una copia diretta… ma il contesto in cui viene utilizzata non è così time critical.

È evidente quindi come questi due svantaggi nel nostro contesto non ne precludano l'utilizzo.

In *MatrixComponentImpl* sono stati invece utilizzati gli Optional offerti dalla libreria Google Guava, in quanto era necessario disporre di Optional Serializzabili, sia per quanto concerne il salvataggio su file, sia per sfruttare la serializzazione in-memory fornita da SerializationUtils (punto precedente).

Luca Passeri:

Durante lo sviluppo della parte del software riguardante la View ovviamente era necessario disporre di elementi grafici come immagini o sfondi; purtroppo per mancanza di tempo (e di abilità) mi è stato impossibile crearle personalmente, ho quindi provveduto a ricercarle sul web.

Jacopo Mastrogirolamo:

Per quanto concerne la gestione degli audio è stata utilizza la libreria JavaFX. La scelta è stata presa per la possibilità di gestire varie tipologie di file (tra i quali quelli con estensione .mp3) ed utilizzare la classe MediaPlayer che offre varie modalità di gestione dei media (pausa, riproduzione partendo da un punto predefinito, loop dell'audio...). I file audio utilizzati all'interno dell'applicazione sono stati ricercati sul web oppure presi dall'insieme di media del gioco "Stronghold 2".

4.4 Autovalutazione e Lavori Futuri

Diego Pergolini:

Sono soddisfatto del mio lavoro, la modellazione del dominio di una, seppur semplificata, città medievale, non era affatto banale vista la mole di entità e interrelazioni in gioco. La parte più difficile è stata senz'altro quella iniziale, in cui ho dovuto pensare analiticamente a quale fossero le astrazioni migliori per implementare il dominio, dovevano essere infatti modellati edifici con funzionalità completamente diverse, garantendo però una estendibilità della gamma degli stessi. Trovate queste astrazioni è stato poi relativamente semplice implementare le varie funzionalità da offrire, la divisione dei macroproblemi e l'affidamento a vari componenti che ne gestivano aspetti specifici si è rivelata vincente, in quanto alcuni bug scovati in fase di testing sono stati agilmente risolti andando a correggere solo minimi aspetti interni, non andando mai ad intaccare l'architettura generale del modello.

I punti di forza della mia parte, sono a mio parere:

- La facilità di inserimento di nuovi edifici, materie prime, livelli di difficoltà/tasse/salari/indicatori
- L'incapsulamento dei vari dettagli dietro un'interfaccia che nasconde all'utilizzatore la complessità del dominio
- Il non permettere a l'utilizzatore di *WorldManager* di arrecare danni all'interno dell'oggetto, in quanto sono forniti solo metodi relativi alle funzionalità offerte e laddove sono ritornati dati sensibili sono passate copie difensive.
- L'uso di pattern e tecniche di programmazione efficace

Un aspetto rivedibile riguarda un caso particolare della disattivazione/rimozione di edifici, cioè quando si tenta di fare queste azioni su una casa, le case sono un tipo di edificio molto particolare, esso "produce" e contiene cittadini(fino alla sua capienza), come noto i cittadini possono essere disoccupati oppure lavorare in qualche altro edificio, ma se il giocatore rimuove una casa, quei cittadini devono necessariamente abbandonare il villaggio in quanto non c'è spazio a sufficiente ad ospitarli, se tali cittadini garantivano il funzionamento di un

determinato edificio, esso va disabilitato. Si può capire come questo aspetto sia molto complesso e l'ho gestito facendo una nuova computazione dello stato del villaggio, questa soluzione non mi aggrada particolarmente ma visto che è stata gestita internamente al model,senza impattare su altre parti, ho deciso di non cercare una strada alternativa magari più "logicamente pulita". In generale tutte le situazioni che riguardavano la morte/abbandono di cittadini ha richiesto molta attenzione, in quanto i cittadini nella mia astrazione non sono gestiti uno ad uno,ma nella loro totalità.

È stata sicuramente una bella esperienza creare un progetto insieme ad altri compagni,che mi ha permesso di capire cosa significhi lavorare in team, nel gruppo penso di aver avuto un ruolo propositivo in quanto grande appassionato di questo tipo di giochi.

Luca Passeri:

Sono personalmente soddisfatto del mio lavoro e ritengo che il tempo e l'impegno dedicato abbiano portato ad un risultato positivo.
Ogni singolo aspetto della mia parte è stato affrontato col pensiero di poter essere migliorato riflettendo costantemente su quali aspetti potevano essere perfezionati, inoltre ho dato molta importanza alla fase di analisi e ciò mi ha portato a costruire un'architettura solida che non è mai stata modificata durante lo sviluppo della View.
Gli aspetti che più mi soddisfano sono:

- l'estendibilità delle varie schermate o sotto-schermate dovuta alla presenza di numerose interfacce, molte volte combinate assieme per formare oggetti più complessi
- La suddivisione dei compiti e funzionalità rispettando le competenze degli oggetti
- I componenti personalizzati combinati con il pattern Builder che hanno conferito eleganza ed estendibilità al codice

Gli aspetti che invece potrebbero essere migliorati sono:

- La non immediata interazione con il controller: avendo optato per una View con più frame si sono andate a formare 4 principali sotto-view istanziate come campi all'interno di ViewControllerImpl. Ciò però è anche dovuto al fatto che non esiste un'unica interfaccia con tutti i metodi possibili, in quanto, come già detto, sono ripartiti in interfacce separate.

Forse modificando l'architettura poteva essere minimizzati gli entry-point con il Controller.

- Il fatto che utilizzando la libreria grafica Swing molti procedimenti sono risultati più semplici ma macchinosi, probabilmente l'utilizzo di una libreria più avanzata come JavaFX avrebbe attribuito maggior qualità al codice.

È stata una bella esperienza lavorare in un team rimanendo comunque responsabili della propria parte, mi ha permesso di capire l'importanza delle modalità di sviluppo del software e le dinamiche di interazione con gli altri membri.
Da grande appassionato di questa tipologia di giochi ritengo di aver dedicato particolare impegno per la realizzazione di questo progetto, rimanendo molto soddisfatto del gioco nel complesso e del lavoro svolto da me e dagli altri componenti del gruppo.

Jacopo Mastrogirolamo:

Sono generalmente soddisfatto del lavoro compiuto e della collaborazione che si è venuta a creare con il gruppo. Importanti sono state le riflessioni comuni che hanno portato ad una applicazione che ritengo piacevole sia graficamente che a livello implementativo. Per quando riguarda la mia parte di lavoro, nonostante miglioramenti siano sempre possibili, sono soddisfatto dell'implementazione della logica di salvataggio della classe *Save* ed anche del thread principale *MedievalThread* con l'utilizzo di variabili di mutua esclusione. Mi sono divertito nell'implementazione delle classi di gestione degli audio avendo carta bianca e non essendo strettamente legato agli altri elementi implementativi, e credo che l'aggiunta di suoni dia quel tocco in più al gioco. Mi posso ritenere quindi generalmente soddisfatto di come è stato gestito il lavoro nella sua totalità.

5 Guida Utente

In questa parte verrà mostrata una sintetica guida utente per istruirlo nelle funzioni principali del gioco.

Una volta lanciato il software la prima schermata sarà d'introduzione e consentirà al giocatore di iniziare una nuova partita o caricarne una, nel primo caso verrà mostrata un'altra schermata che permette di impostare le caratteristiche del gioco (attenti a compilare e selezionare tutti i campi richiesti), mentre nell'altro vi verranno mostrati tutti i salvataggi effettuati, se presenti, altrimenti vi verrà riportato che non ve ne sono.

Nota: tutta la descrizione che segue viene riproposta all'interno della schermata di gioco in un tab apposito ("instructions") in inglese ma in maggior dettaglio.

Entrati nella schermata di gioco vera e propria troverete una mappa con differenti tipi di terreno, ognuno presentante differenti caratteristiche; in basso un set di edifici che possono essere costruiti mentre a destra un pannello che offre diverse funzionalità:

- Una scritta che vi indicherà l'elemento della mappa sul quale risiede il puntatore
- Una piccola mappa che mostra la miniatura della mappa di gioco, è possibile muoversi all'interno di quest'ultima cliccando su un punto della mini-mappa
- Lista di risorse di cui disponete
- Due bottoni che vi consentono di costruire o selezionare un elemento nella mappa di gioco (inizialmente la scritta del bottone è chiara, se vi cliccate diventerà rossa e significa che quella è l'azione che state effettuando)

Durante il gioco potrebbero apparirvi dei messaggi di varia natura (mancanza di risorse, eventi straordinari…), il gioco non procederà fin tanto che sono aperti, perciò leggeteli con attenzione perché spesso possono portare notizie importanti.

In qualunque momento desideriate interrompere il gioco basta che premiate il tasto "esc" (eccetto nella schermata di "instruction" e quando un altro messaggio è già aperto) che metterà in pausa il gioco e vi offrirà varie funzionalità come salvare in uno degli slot disponibili, tornare al menu o uscire.

Noterete nella schermata di gioco in alto altri tab, i quali permettono di avere un'aggiuntiva gestione del vostro villaggio, quello degli "indicators" vi permetterà di conoscere più in dettaglio la condizione sociale della vostra città, mentre quello di "economy" consentirà di modificare la condizione economica della stessa.

Ponete particolare attenzione alla quantità di "gold", "food" e "population", poiché nel caso di assenza del primo o degli ultimi due assieme la vostra partita terminerà.