

CosmoCity

“Programmazione ad Oggetti”

Yuri Collini, Elisa Simoni

31 Gennaio 2024

Sommario

Il progetto che si era prefissato di realizzare il nostro gruppo era quello di implementare un simulatore per la gestione di una colonia spaziale semi-autonoma. Nello specifico si voleva poter permettere agli utenti di gestire i principali aspetti di funzionamento e manutenzione dell'intera colonia spaziale come ad esempio gestire i coloni e le loro produzioni, monitorare la salute ed il benessere dei coloni nonché la gestione della sicurezza della colonia stessa.

Indice

1 Analisi	3
1.1 Requisiti	3
1.1.1 Requisiti funzionali	3
1.1.2 Requisiti non funzionali	4
1.2 Analisi e modello del dominio	4
2 Design	8
2.1 Architettura	8
2.2 Design dettagliato	10
2.2.1 Elisa Simoni	10
2.2.2 Yuri Collini	15
3 Sviluppo	21
3.1 Testing automatizzato	21
3.2 Metodologia di lavoro	22
3.2.1 Elisa Simoni	22
3.2.2 Yuri Collini	23
3.2.3 Collaborazione	24
3.3 Note di sviluppo	24
3.3.1 Elisa Simoni	24
3.3.2 Yuri Collini	25
4 Commenti finali	26
4.1 Autovalutazione e lavori futuri	26
4.1.1 Elisa Simoni	26
4.1.2 Yuri Collini	26
5 Guida utente	28
5.1 Main Menu	28
5.2 Impostazioni Partita	29
5.3 Inizio Partita	30

5.3.1	Sezione di Sinistra	30
5.3.2	Sezione Centrale	31
5.3.3	Sezione di Destra	31
5.4	Obbiettivo e Funzionamento della simulazione	32
5.4.1	Obbiettivo Principale	32
5.4.2	Funzionamento	32

Capitolo 1

Analisi

1.1 Requisiti

Lo scopo di questo progetto è la realizzazione di un simulatore, denominato CosmoCity, per la gestione di una colonia spaziale. L'obiettivo principale è quello di gestire diversi aspetti cruciali relativi alla colonia come le risorse, la salute dei coloni e la sicurezza della colonia stessa.

1.1.1 Requisti funzionali

- Gli utenti dovranno avere la possibilità di creare la propria colonia, aggiungere al suo interno diversi personaggi con ruoli specifici, come medici, chimici, ingegneri, cuochi, ecc.
- Ogni personaggio dovrà avere la capacità di svolgere azioni rilevanti per il proprio settore di competenza, influenzando direttamente lo stato globale della colonia.
- Gli utenti potranno registrare e monitorare eventuali variazioni dell'assegnazione dei personaggi ad altri settori attraverso l'interfaccia offerta dal portale.
- La colonia sarà soggetta a eventi casuali imprevisti come asteroidi, rivolte interne, e altri eventi simili che influenzano la sicurezza e l'ecosistema della colonia stessa.
- La gestione della colonia dovrà avvenire tramite una schermata simile a un pannello di controllo centralizzato.

1.1.2 Requisiti non funzionali

- Il sistema deve essere intuitivo e facile da utilizzare per gli utenti, indipendentemente dalle loro conoscenze ed abilità tecniche.

1.2 Analisi e modello del dominio

Il simulatore si basa su diverse entità e relazioni fondamentali tra di esse.

Quando un utente accede al simulatore, avrà la possibilità di creare una nuova **”Simulation” (Colonia)**, comprensiva di tutti gli elementi centrali del gioco. Gli elementi principali che costituiscono l’intero ecosistema del gioco sono :

- ”Resources” (Risorse)
- ”Settlers” (Coloni)
- ”Sectors” (Settori)
- ”Events” (Eventi)

Le **”Resources” (Risorse)** costituiscono gli elementi chiave del gioco, poiché la loro disponibilità o carenza influisce direttamente sulla vita e sulla sopravvivenza della colonia e della sua popolazione. La capacità e le competenze dei membri della colonia sono responsabili della generazione delle risorse utilizzate all’interno della comunità. Le risorse prodotte dai personaggi all’inizio della simulazione andranno distribuite in vari settori.

Le risorse oltre ad essere prodotte potranno essere consumate o perse a causa di eventi imprevisti come calamità o disastri di vario genere i cosiddetti **”Random Events” (Eventi Casuali)**.

Tutti i **”Settlers” (Coloni)** presenti all’interno del simulatore, devono avere ruoli specifici. Ogni settler viene assegnato ad un settore all’avvio della simulazione. Ogni colonia potrà ospitare al **massimo 10 personaggi**, ognuno dei quali avrà un compito diversificato.

All’interno del gioco saranno presenti **2 tipologie di settler** :

1. **Manadatory Settlers (Coloni Obbligatori)**
 - (a) Farmer (Contadino)
 - (b) Doctor (Dottore)

- (c) Blacksmith (Fabbro)
- (d) Gunsmith (Armaiolo)

2. Optional Settlers (Coloni Opzionali)

- (a) Chemist (Chimico)
- (b) Cook (Cuoco)
- (c) Technician (Tecnico)

I Mandatory Settler produrranno le seguenti risorse :

- Farmer = Food
- Doctor = Medicine
- Blacksmith = Screw
- Gunsmith = Weapons

All'interno del simulatore saranno presenti i **"Sector"** (**Settore**). Questi rappresentano un'astrazione di un elemento fondamentale per la vita e il funzionamento della colonia stessa. All'interno del simulatore saranno presenti 4 settori :

1. Farm (Fattoria)
2. Hospital (Ospedale)
3. Workshop (Officina)
4. Military Base (Base Militare)

All'interno del gioco vi sono 3 livelli di difficoltà del gioco :

1. Easy (Facile)
2. Medium (Medio)
3. Hard (Difficile)

La difficoltà scelta dall'utente durante l'inizio di una nuova partita sarà un criterio determinante per stabilire la quantità di risorse iniziali a disposizione dell'utente all'avvio della simulazione.

Il gioco termina in due casi principali :

1. La popolazione della colonia arriva a 0.
2. Una qualsiasi risorsa della colonia arriva a 0.

All'interno della colonia, vive la popolazione. Quest'ultima può aumentare con il passare del tempo e diminuire ogni volta si ammala e muore un abitante.

Oltre a una gestione inefficiente delle risorse, la colonia può essere sconvolta da imprevisti sotto forma di "Random Events" (Eventi Casuali). Questi eventi imprevisti possono causare danni e perdite alla colonia, aggiungendo ulteriori sfide e variabilità alla simulazione.

Durante il susseguirsi della simulazione, il tempo scorre e una gestione inadeguata delle risorse può causare un declino della colonia. Non esiste una vittoria definitiva; piuttosto, l'obiettivo del giocatore è prolungare la vita e la sopravvivenza della colonia. Gli elementi costitutivi il problema sono sintetizzati in Figura 1.1.

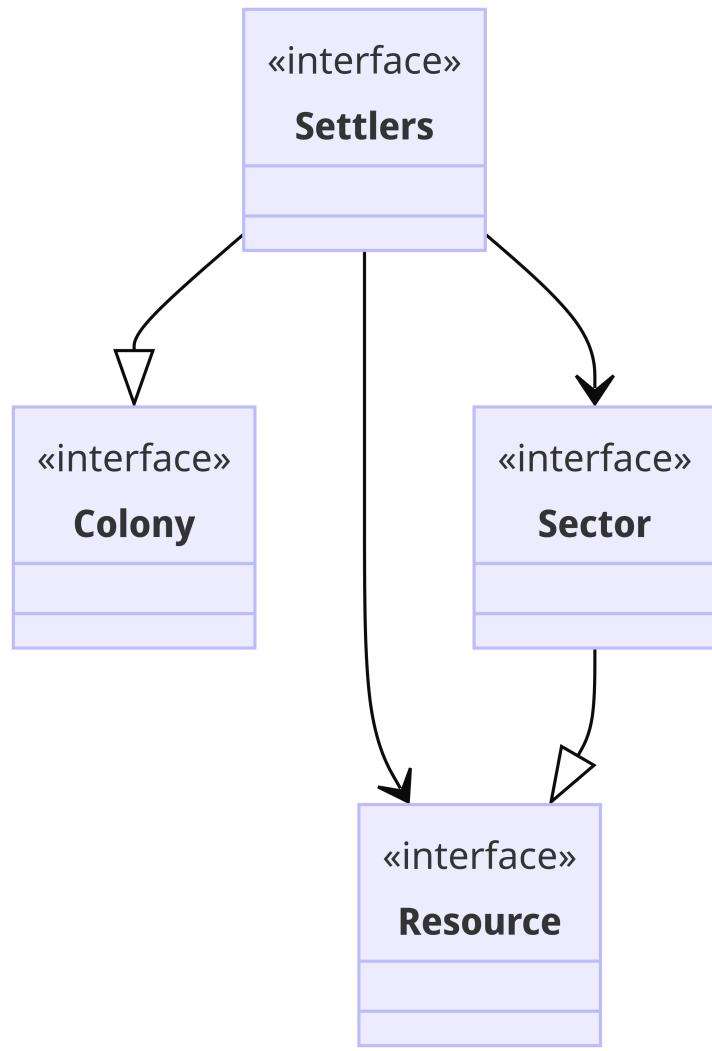


Figura 1.1: Schema UML dell'analisi del problema, con rappresentate le entità principali ed i rapporti fra loro

Capitolo 2

Design

2.1 Architettura

Il progetto è concepito secondo il paradigma MVC (Model-View-Controller), che consente di separare la logica della simulazione dalla sua rappresentazione grafica. L'elemento centrale del progetto è il `SimulationController`, responsabile della gestione degli stati del sistema e dell'avvio della simulazione. Questa simulazione coinvolge coloni che producono e consumano risorse. L'implementazione pratica di questo design si riflette nei vari controller, come `DashboardController`, `CreateColonyController`, e `AssignSettlerController`, ciascuno specializzato nella gestione di diverse fasi della simulazione. Ogni controller utilizza un'interfaccia grafica associata, come `DashboardView`, `CreateColonyView`, e `AssignSettlerView`. Al centro di questa architettura si trova il `SimulationController`, che assume il ruolo di controller principale. Essendo il cervello attivo della simulazione, il `SimulationController` gestisce gli stati del sistema e inizia l'esecuzione della simulazione. È responsabile di coordinare le interazioni tra il Modello (la simulazione stessa) e la View (rappresentazione grafica).

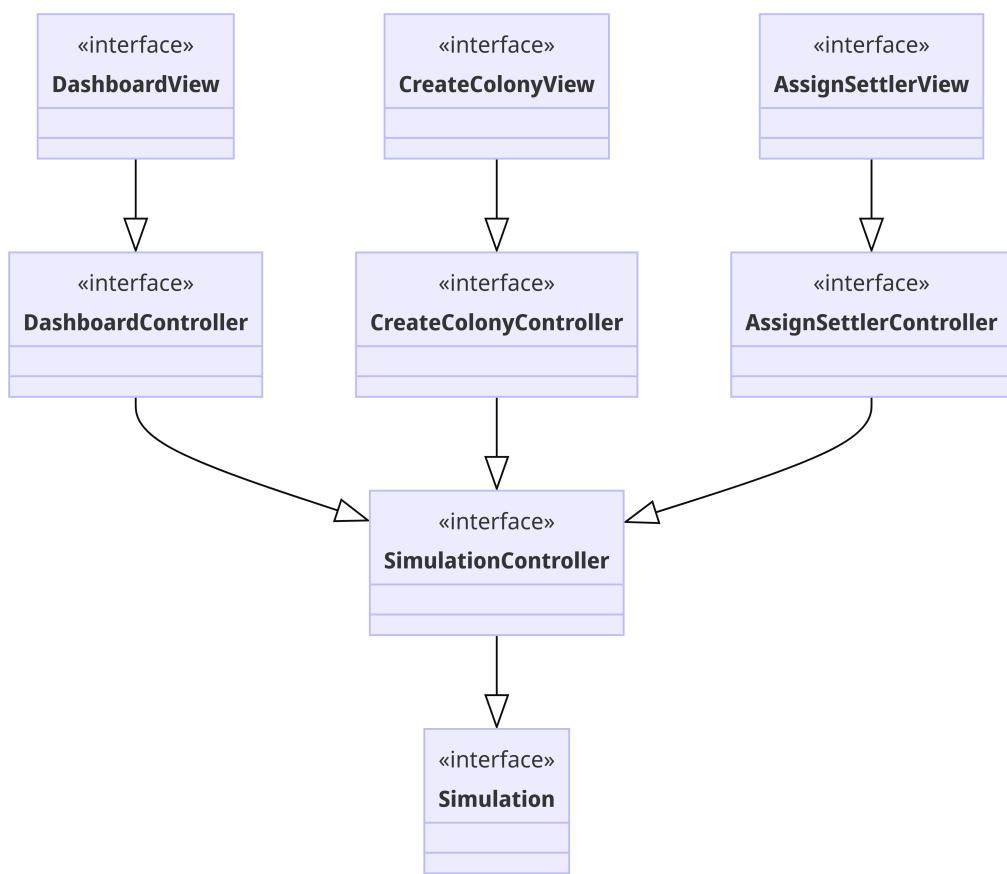


Figura 2.1: MVC UML

2.2 Design dettagliato

In questa sezione si andrà ad approfondire il design dell'applicativo nel dettaglio.

2.2.1 Elisa Simoni

Resource

Questa sezione fornisce una dettagliata descrizione dell'entità "Resource" e del suo impatto nella simulazione. Nel contesto della simulazione, i coloni generano risorse in base al settore in cui si trovano. Ad esempio, se si trovano nel settore "Fattoria", la risorsa prodotta sarà il cibo; nel settore "Base Militare", la risorsa sarà l'arma; nel settore "Ospedale", si produrranno medicinali; infine, nel settore "Officina", vengono prodotte viti.

Esistono due categorie di risorse: "BaseResource" e "StackedResource". Possono essere Cibo, Viti, Medicinali e Armi. Ciascuna risorsa può essere prodotta. Inoltre, esistono le "StackedResource", che possono essere modificate durante lo svolgimento del gioco poiché definiscono le risorse disponibili nella simulazione.

Ogni colono consuma cibo e produce risorse, mentre la popolazione rappresenta un tipo particolare di "StackedResource" che consuma e richiede tutte le tipologie di risorse.

Per implementare questa logica, è stato utilizzato un design pattern ispirato al Template Pattern:, garantendo una struttura flessibile ed estensibile. Di seguito una descrizione e una rappresentazione grafica. Il pattern Template Pattern è stato scelto per introdurre comportamenti specifici nelle risorse in modo dinamico, senza dover modificare la loro interfaccia di base. Questa scelta offre maggiore flessibilità e consente di comporre diverse funzionalità in modo modulare. In generale una risorsa può essere usata in diversi modi e avere nomi e aggiunte diverse, ma sempre una risorsa rimane.

- Resource rappresenta un'interfaccia generica per le risorse all'interno del sistema, garantendo uniformità attraverso un metodo comune per ottenere la quantità della risorsa. Nel pattern costituisce le fondamenta della struttura gerarchica di risorse, fornendo un'interfaccia comune e unificata.
- La classe astratta BaseResource è progettata per servire come punto di partenza per la creazione di risorse specifiche. Gestisce la quantità come attributo comune e offre un costruttore per l'inizializzazione. fornisce una struttura comune per tutte le risorse, garantendo

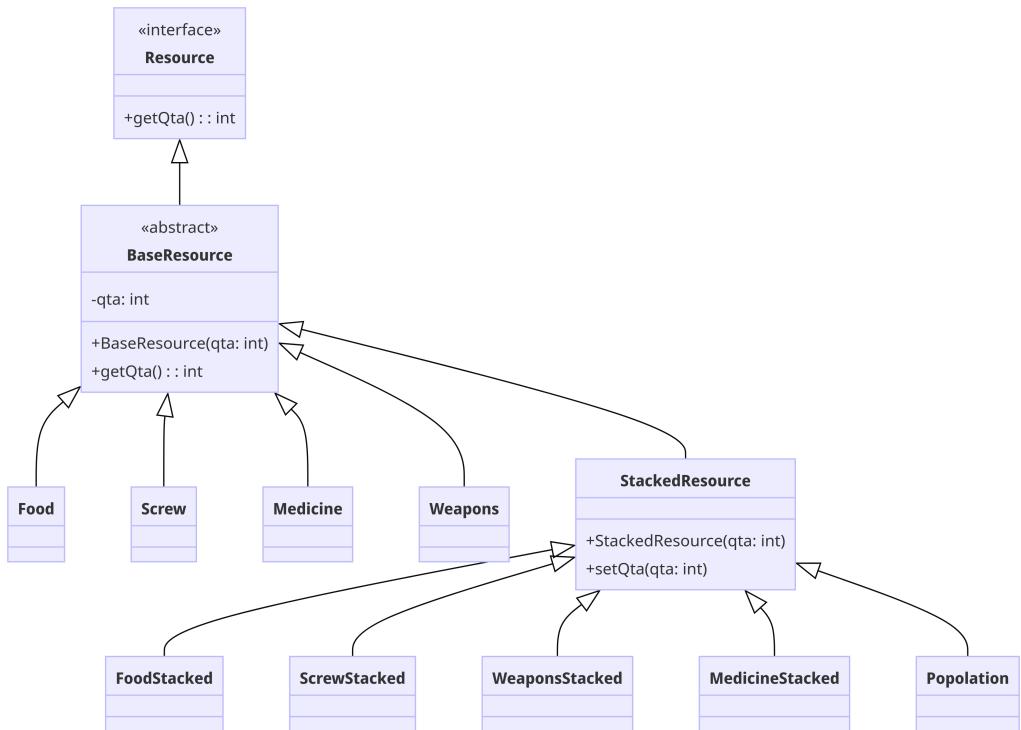


Figura 2.2: Resource UML - Template Pattern

che l'accesso alla quantità sia uniforme indipendentemente dal tipo di risorsa.personalizzare.

- La classe **StackedResource** offre una variante della base resource fornendo maggiore flessibilità del concetto di **Resource**.
- La classi **Food/Screw/...** estendono **BaseResource** e specializzano applicando le caratteristiche specifiche utilizzando l'ereditarietà. L'ereditarietà consente di specializzare il comportamento del metodo template, adattandolo alle esigenze specifiche di ciascuna risorsa.
- La classi **Popolazione/FoodStacked/...** estendono **StackedResource** e specializzano applicando le caratteristiche specifiche utilizzando l'ereditarietà.

In conclusione il metodo template è `getQta()`.2.2

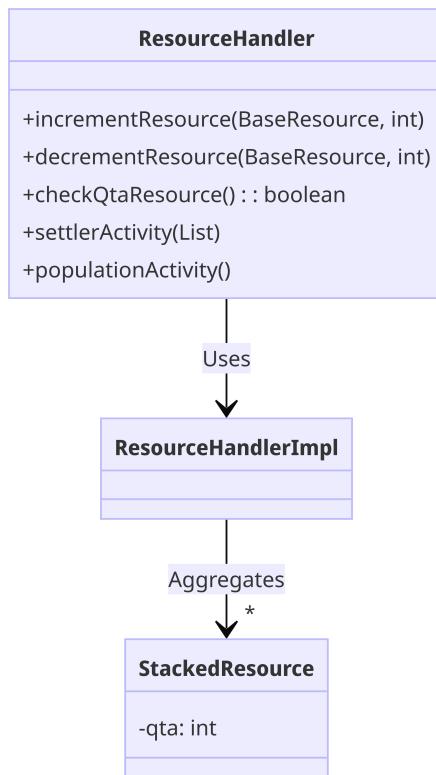


Figura 2.3: Resource Handler

ResourceHandler

Le StackedResource della simulazione vengono gestite da un handler che si occupa del loro decremento e incremento e gestione della quantità in generale. Vedere figura 2.3

Settlers

Questa sezione fornisce una dettagliata descrizione dell'entità "Settler". Nel contesto della simulazione, i coloni generano risorse in base al settore in cui si trovano. Esistono due tipi di settler: Mandatory Settler e Optional Settler. Il pain point era creare una struttura gerarchica che incapsulasse le decisioni di creazione delle istanze in classi concrete. La scelta è ricaduta su una variante Abstract del Factory Method Pattern.

- L'interfaccia Base Settler definisce l'interfaccia comune per tutti i settler fornendo la definizione dei metodi base per getAppetite e sectorAssigned.

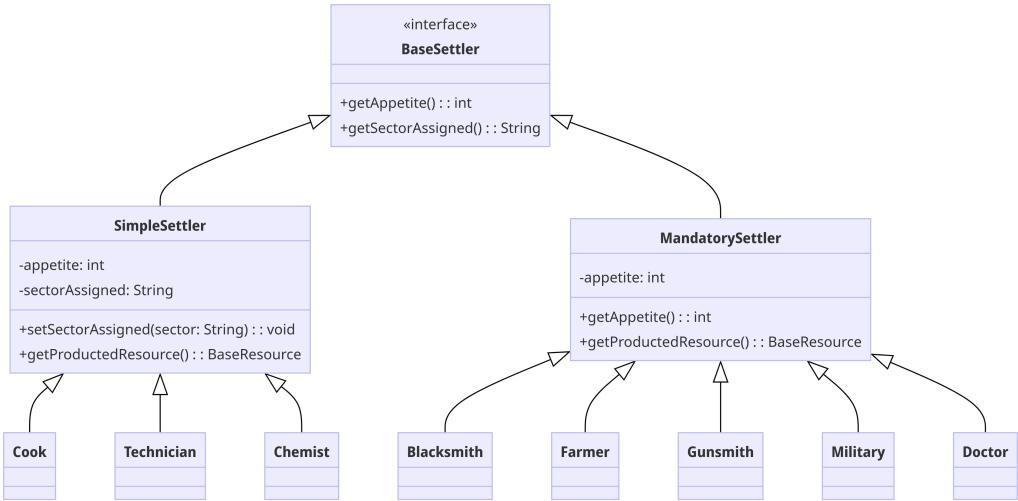


Figura 2.4: Settler UML - Abstract

- SimpleSettler sstende BaseSettler e fornisce un’implementazione di base per le funzionalità comuni dei settler. Contiene anche un Factory Method (setProducedResource) che consente alle sottoclassi di produrre risorse in base al settore assegnato. La classe astratta è una sorta di ”creator” nel contesto del pattern
- MandatorySettler estende BaseSettler e fornisce un’implementazione parziale delle funzionalità comuni dei settler. La sua principale differenza rispetto a SimpleSettler è nella semantica del Factory Method, adattandosi alle necessità specifiche dei settler obbligatori.
- Le classi Cook/Technician/Chemist estendono SimpleSettler e forniscono un’implementazione specifica per il settore dove vengono utilizzate. Nel pattern rappresenta una ”concrete creator” specializzata per la creazione di uno specifico settler.
- Le altre classi Doctor/Blacksmith/... estendono mandatory settler forniscono specifiche implementazioni per i vari tipi di settler. Anche questo è un ”concrete creator” nel pattern.

Di seguito un rappresentazione grafica:

Simulation e SimulationController

- La classe Simulation agisce come il modello del sistema, contenendo i dati e la logica del dominio. Rappresenta lo stato corrente della simu-

lazione, inclusi il nome della colonia, i coloni, le risorse e la difficoltà. Il modello gestisce il recupero e la manipolazione dei dati, mantenendo una separazione chiara dalla logica di presentazione

- La classe `SimulationController` funge da controller principale che si occupa di gestire le interazioni utente, coordinando il modello e la vista. È il punto di ingresso per l'aggiornamento dello stato della simulazione e gestisce gli eventi generati dall'interfaccia utente. Il controller si occupa dell'aggiornamento del modello (`Simulation`) in risposta alle azioni dell'utente e si interfaccia con la vista (`Dashboard`) per presentare i dati. Vedere immagine: 2.5

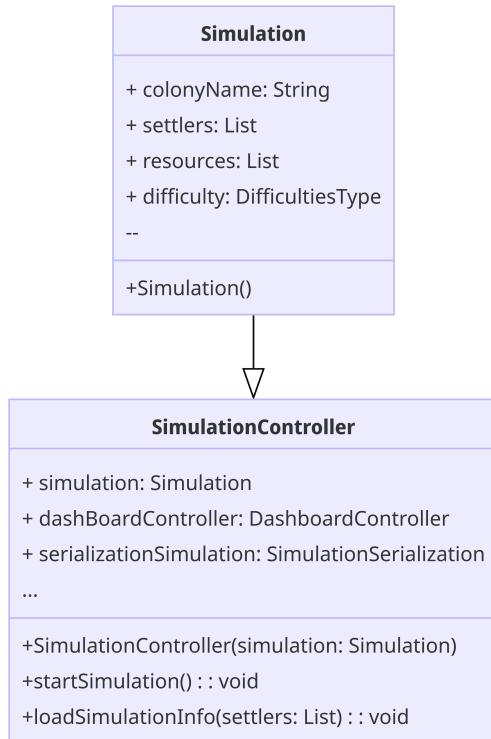


Figura 2.5: Simulation UML

View e Controller

Rispettando il pattern MVC abbiamo suddiviso in maniera capillare ogni View dal suo controller. Un esempio è `CreateColony` e il suo controller di cui verrà riportato il design di seguito. JavaFX è utilizzato per la creazione

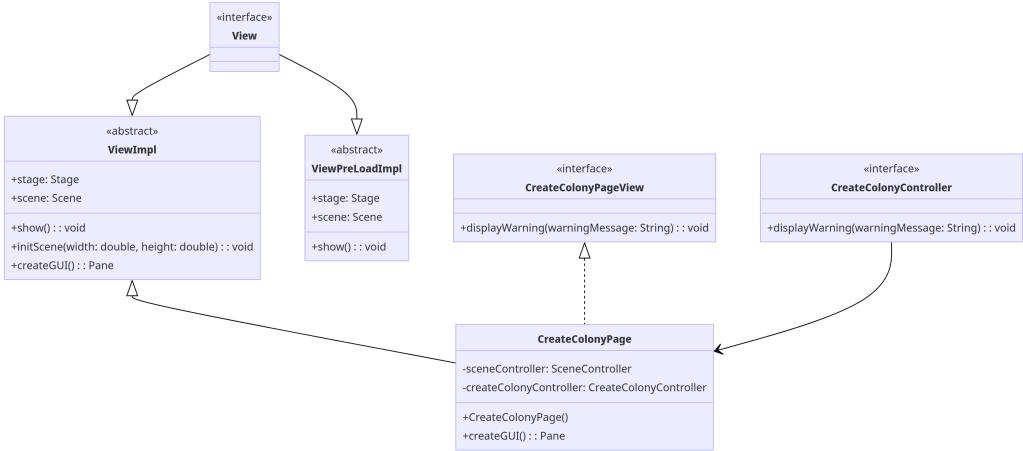


Figura 2.6: Create Colony and View UML

di tutte le view in quanto uno strumento potente e semplice da usare, che permette la creazione di una GUI responsive ed efficace. La classe CreateColonyPage agisce come la componente View nell'architettura MVC. È responsabile di definire l'aspetto e la disposizione dell'interfaccia utente, nonché di gestire gli input dell'utente durante la creazione della colonia. Di seguito una rappresentazione grafica: 2.6

2.2.2 Yuri Collini

Observable

Questa sezione offre una dettagliata descrizione di come è stata gestita l'implementazione per la gestione del tempo e degli eventi ad esse correlati nel contesto della simulazione della nostra colonia spaziale. Nello specifico, gli elementi presentati di seguito, si attengono al pattern MVC e più precisamente al controller denominato : TimeController, svolgono un ruolo chiave per quanto riguarda gestione temporale e la sincronizzazione dell'interfaccia grafica.

Nel contesto del simulazione, la gestione del tempo è un elemento fondante. Infatti, all'avvio della partita di gioco, una volta settate tutte le caratteristiche come i settlers ed il livello di difficoltà, parte un timer il quale resta attivo per tutta la durata della simulazione.

Per la gestione del tempo di gioco ho deciso di prendere ispirazione dall'Observer Pattern.

L'Observer Pattern è un design pattern comportamentale che definisce una dipendenza uno-a-molti tra gli oggetti, in modo tale che quando uno

degli oggetti cambia stato, tutti i suoi osservatori (da cui prende appunto il nome) vengano notificati e aggiornati automaticamente.

Questo pattern si rivelato essere particolarmente utile per gestire le notifiche agli "ascoltatori / osservatori"

Nel contesto del nostro progetto, l'Observer Pattern è stato applicato per gestire la notifica degli eventi temporali da parte del TimerObservable a tutti gli oggetti interessati ed in ascolto, come ad esempio l'EventObserver.

Quando all'inizio di una partita viene fatto partire il tempo, tutti gli oggetti in ascolto riscono a capire il tempo trascorso dall'avvio perché vengo costantemente aggiornati tramite l'update che si occupa di notificare tutti gli osservatori che si sono "registrati" di quanto tempo sia trascorso, consentendo loro di reagire / comportarsi di conseguenza a tali eventi temporali come ad esempio facendo generare gli eventi pseudo casuali di cui tratteremo di seguito.

L'interfaccia Observer definisce un metodo update, che ogni classe che agisce come osservatore deve obbligatoriamente implementare. In questo progetto, l'EventObserver implementa questa interfaccia, consentendo di definire il comportamento specifico che deve verificarsi quando un qualsiasi oggetto riceve una notifica di aggiornamento da parte del TimerObservable.

In sintesi, il TimerObservable funge da soggetto osservabile, mentre l'EventObserver si "registra" a questo componente per ricevere notifiche costanti allo scorrere del tempo.

Questa implementazione aderisce al principio di separazione delle responsabilità, ed al pattern MVC, promuovendo anche la modularità del codice. Ciò ci permette una gestione degli eventi nel contesto della simulazione che non solo è efficiente ma risulta essere anche facilmente estensibile, semplificando la manutenzione e l'aggiunta di nuove funzionalità nel tempo.

TimerObservable

La classe **TimerObservable** rappresenta il modello nel contesto dell'applicazione e gioca un ruolo cruciale nella gestione degli eventi temporali. Questa classe implementa come detto in precedenza, il pattern Observable, consentendo ad altri componenti, come ad esempio l'EventObserver, di registrarsi e ricevere notifiche in risposta a determinati eventi temporali.

Il metodo **addObserver** presente all'interno della classe permette l'iscrizione / aggiunta degli osservatori, mentre **removeObserver** consente, come dice esattamente il nome, la loro rimozione. Componenti molto importanti sono i metodi **run** e **notifyObservers** i quali si occupano rispettivamente di gestire lo scorrere del tempo e la notifica agli osservatori in ascolto.

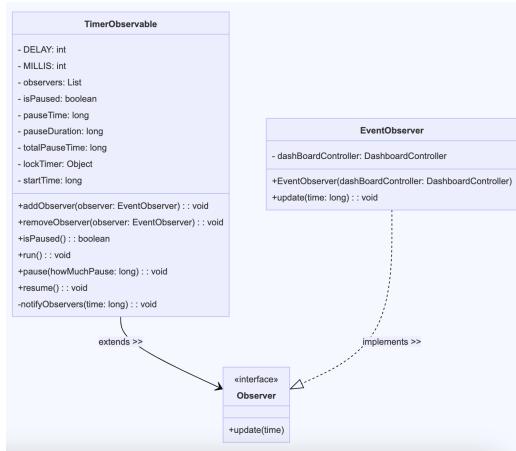


Figura 2.7: Enter Caption

EventObserver

Altro elemento fondante è la classe **EventObserver**. Questa classe è il controller che reagisce agli eventi temporali notificati dal TimerObservable. Essa implementa l’interfaccia Observer, fungendo da osservatore nel contesto del pattern Observable.

Il design di EventObserver segue il pattern MVC, dove il controller è responsabile di reagire agli eventi temporali e di aggiornare ad esempio la view. Questa classe è costruita intorno al concetto di observable, consentendo una separazione pulita delle responsabilità.

Observer

L’interfaccia **Observer** rappresenta il contratto che gli osservatori devono seguire. La classe definisce un unico metodo ovvero l’**update** che verrà invocato quando il soggetto osservabile, ovvero il timer di gioco, effettua un cambiamento di stato, nel nostro caso, lo scorrere del tempo ogni secondo.

Ho scelto consapevolmente di prendere ispirazione dal design pattern ”Observer” per favorire la modularità, la manutenibilità dell’applicativo favorendo al contempo la sua estendibilità ed i campi di applicazione.

Questo approccio consente agli osservatori di reagire dinamicamente agli eventi temporali senza essere direttamente vincolati dal soggetto osservabile.

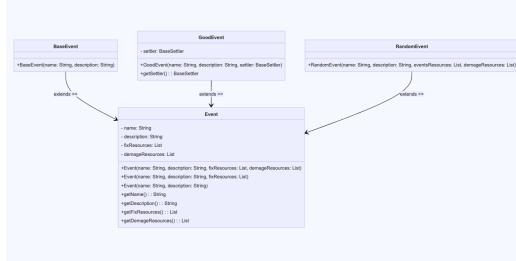


Figura 2.8: Gestione Eventi

Eventi

Nel nostro progetto di gestione di una fantomatica colonia spaziale, la gestione degli eventi è un aspetto importante ed interessante per comprendere e modellare le dinamiche su cui poggia le basi la simulazione. Di seguito tratterò nel dettaglio gli aspetti di design più rilevanti.

Le classi chiave coinvolte nella gestione degli eventi sono **BaseEvent**, **GoodEvent**, e **RandomEvent**, ognuna progettata per rappresentare categorie specifiche di eventi che si possono verificare in vari momenti all'interno della simulazione in base a determinate condizioni del gioco ed ad alcuni stati.

Tutta la gestione degli eventi è stata realizzata implementando il pattern MVC, il quale ci fornisce una chiara separazione tra la logica di business, la gestione degli eventi e l'interfaccia utente.

Le elementi coinvolti all'interno della gestione degli eventi sono :

1. Event
2. BaseEvent
3. GoodEvent
4. RandomEvent
5. EventManager

Event

La classe **Event** è al centro della gestione degli eventi che si possono verificare all'interno del nostro applicativo. Ho progettato la classe Event come classe base. Fornisce una struttura comune per rappresentare diversi tipi di eventi che possono verificarsi durante il corso del gioco.

Durante una partita infatti possono verificarsi vari eventi come : eventi pseudo casuali, i cosiddetti RandomEvent (verranno analizzati di seguito),

GoodEvent che per l'appunto come dice il nome stesso sono eventi buoni / positivi per il giocatore come la crezione di un nuovo settler. In fine, ma non per importanza troviamo i BaseEvent.

La classe **Event**— gioca un ruolo fondamentale nel permettere una rappresentazione flessibile e modulare degli eventi all'interno della simulazione. La sua struttura ben definita facilita l'estensione del sistema per gestire nuovi tipi di eventi, garantendo al contempo coerenza e chiarezza nel modello di dominio dell'applicazione.

La classe **BaseEvent** è dedicata alla rappresentazione di eventi fondamentali all'interno della simulazione. Tali eventi forniscono una base sulla quale costruire situazioni di eventi più complessi. Questo rende il codice più estendibile e maggiormente efficiente.

Come detto anche poco sopra, la gestione dei cosiddetti "eventi positivi" è stata affidata alla classe GoodEvent.

Questa classe **GoodEvent** estende il concetto di evento definito tramite la classe Event, presente all'interno dell'applicativo inoltre aggiunge altre funzioni come ad esempio getSettler.

Gli eventi positivi sono stati concepiti all'interno della simulazione come evento "premio" per l'utente. Infatti questa classe fornisce, in base a determinate caratteristiche e stato di risorse, un nuovo settler al giocatore. Questa tipologia di evento è stata scelta per contribuire al progresso e all'arricchimento della colonia e favorendo la soddisfazione del giocatore.

La modellazione degli eventi pseudo casuali è stata implementata mediante la classe **RandomEvent**.

Questa classe ricopre un ruolo molto importante all'interno del gioco. Lo scopo di questa classe è appunto la creazione eventi pseudo casuali.

Ho detto appositamente pseudo casuali in quanto i possibili eventi che si possono verificare vengono selezionati in maniera casuale da un file JSON prestabilito di eventi. Questa tipologia di eventi è stata scelta di essere inserita nel contesto di gioco per introdurre elementi di sfida all'interno del sistema, fornendo un livello di imprevedibilità.

Questo approccio rispecchia i principi del pattern MVC, fornendo anche la possibilità di aggiungere in futuro nuovi eventi casuali senza la necessità di modificare il codice dell'applicativo.

All'interno del sistema è presente anche la classe **EventManager**.

Quest'ultima svolge un ruolo centrale nella gestione e generazione degli eventi all'interno dell'applicazione. Il suo scopo principale è coordinare la generazione e la gestione degli eventi, fornendo un'interfaccia per accedere a diverse tipologie di eventi all'interno del sistema.

L'utilizzo della serializzazione consente di caricare gli eventi casuali da una sorgente esterna, in questo caso una risorsa come un file JSON, offrendo una maggiore flessibilità nella configurazione del gioco.

L'implementazione di queste classi è modulare, inoltre consente una facile estensione e adattamento alle esigenze future del sistema in esame.

Dasboard

Adesso ci occuperemo di analizzare il design di alcuni elementi della view, nello specifico della bashbord principale di gioco.

Quando l'utente ha impostato tutta la configurazione di base del gioco viene "catapultato" in una schermata, quella principale del gioco, ovvero la Dashboard.

Tramite questa schermata di gioco, l'utente interagisce con il gioco ed allo stesso tempo può vedere dati ed informazioni di vario genere.

Per implementare questa grafica si sono realizzati i seguenti elementi:

1. View
2. ViewImpl
3. Dashboard
4. DashboardImpl

L'elemento principale responsabile di implementare la grafica della dashboard è appunto la classe Dashboard.

Questa classe si occupa di gestire la presentazione dell'interfaccia utente e risponde alle interazioni effettuate dell'utente quando magari preme sui bottoni alla sua sinistra e si occupa di rappresentare visivamente le varie informazioni.

All'interno della classe, troviamo il metodo `createGUI()`, il quale trae ispirazione dal Factory Method Pattern, in maniera implicito. Questo metodo è responsabile della creazione e inizializzazione dell'interfaccia utente (GUI) restituita dalla classe `Pane`.

Per questa classe come per altre mi sono nuovamente ispirato al pattern Observer, specialmente nella gestione degli aggiornamenti dell'interfaccia grafica in risposta ai cambiamenti nello stato del gioco. Ad esempio, tramite i metodi come `updateTimeLabel`, `updateResourceLabel`, e `updateCircle`.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

I test presenti nel seguente progetto sono stati scelti per la loro importanza nello stesso. Quest'ultimi verificano semplicemente la corretta istanziazione e caricamento della simulazione e delle classi che ne girano attorno. La libreria utilizzati per la loro creazione JUnit 5.

Si è scelto di non testare la view in quanto durante la progettazione sono state gestite in maniera capillare le eccezioni che ne prevedevano l'update.

In particolare si è deciso di testare:

- EventManager per testando la consistenza della creazione
- ResourceHandler nelle sue tre operazioni fondamentali ovvero decremento, incremento ed check della quantità delle risorse-
- Serialization per controllare se la serializzazione e la de-serializzazione restituivano effettivamente componenti della simulation

Per quanto riguarda la parte del model

- BaseResource e le sue declinazioni per la loro corretta creazione e assegnazione
- Settler e le sue specificazioni, viene controllata la loro produzione e l'assegnazione al settore corretto
- Sector per vedere se la sua creazione rispetta la prassi designata

3.2 Metodologia di lavoro

Il progetto è stato suddiviso in base a 4 macro-sezioni, ognuno dei partecipanti al progetto doveva provvedere alla creazione della view, controller e model di ciasuna macro sezione.

3.2.1 Elisa Simoni

La creazione delle risorse, dei coloni e della loro gestione è stata affidata ad Elisa Simoni, che ha provveduto anche a creare il sistema simulazione per la loro completa integrazione. Di questi elementi si è dovuta creare la view e i controller per la loro assegnazione all'interno della simulazione. Inoltre sempre da quest'ultima è stato gestito il salvataggio, e quindi di conseguenza la serializzazione e la de-serializzazione di alcuni elementi. Infine le è stata affidata anche la gestione dei multimedia. In termine di importanza l'implementazione delle risorse richiedeva altrettante elasticità, e l'utilizzo di interfacce comuni e astrazione è stato fondamentale

La gestione dei settler è stata l'elemento cardine della progettazione, ogni settler compie azioni e produzioni differenti. Si è cercato di mantenere la soluzione semplice e facilmente estendibile in caso di ampliamento del programma.

```
└── settlers
    BaseSettler.java
    Blacksmith.java
    Chemist.java
    Cook.java
    Doctor.java
    Farmer.java
    Gunsmith.java
    MandatorySettler.java
    Military.java
    SimpleSettler.java
    Technician.java
```

Figura 3.1: Settler file list

```
└── resources
    BaseResource.java
    Food.java
    FoodStacked.java
    Medicine.java
    MedicineStacked.java
    Population.java
    Resource.java
    Screw.java
    ScrewStacked.java
    StackedResource.java
    Weapons.java
    WeaponsStacked.java
```

Figura 3.2: Resource File list

3.2.2 Yuri Collini

A me è stata affidata una serie di responsabilità (alcune anche rilevanti come la gestione del tempo) all'interno del progetto. Tra i miei compiti principali, mi sono dedicato alla creazione e gestione degli eventi, come ad esempio i RandomEvent. La mia attività di sviluppo è stata incentrata sull'implementazione di un sistema robusto e dinamico per gestire il timer di gioco, elemento cruciale per il corretto funzionamento e la sincronizzazione degli eventi nel contesto della simulazione.

Un altro compito che mi è stato affidato è relativo alla progettazione della dashboard, elemento centrale dell'interfaccia di gioco con cui interagisce.

In aggiunta, ho lavorato anche alla creazione dei settori all'interno del gioco. Questi settori sono componenti della simulazione e includono elementi grafici, informazioni sullo stato e altri dettagli che contribuiscono alla ricchezza dell'esperienza di gioco.

Nel corso dello sviluppo del progetto mi sono occupato anche di creare alcuni pannelli di dialogo. Questi pannelli sono stati progettati per fornire un mezzo efficace per gestire i cambiamenti nello stato del gioco, consentendo agli utenti di interagire con gli eventi e le situazioni emergenti in modo chiaro e informativo.

3.2.3 Collaborazione

In aggiunta alla progettazione architetturale del progetto, abbiamo collaborato attivamente nella gestione degli eventi ricorrenti, focalizzandoci sulle variazioni di risorse e su eventi specifici all'interno della simulazione. La responsabilità della generazione e gestione dei test è stata equamente suddivisa, coinvolgendo entrambi i membri del team.

L'integrazione iniziale è stata complessa, principalmente a causa della mancanza di una definizione chiara dello scheletro del progetto. Tuttavia, mediante un rafforzamento della fase di progettazione, siamo riusciti a facilitare notevolmente la fusione degli elementi creati. Questo processo ha evidenziato l'importanza di una pianificazione dettagliata e ha contribuito a ottimizzare l'armonizzazione delle diverse componenti del progetto.

3.3 Note di sviluppo

3.3.1 Elisa Simoni

Elementi avanzati di Java utilizzati

- Utilizzo di Stream, lambda expressions e Optional (L'utilizzo degli Stream è stato copioso di seguito farò vedere solo alcuni esempi più corposi). <https://github.com/elisasimoni/CosmoCity/blob/2778f6abfc86410977374c2458996ce>
- Utilizzo di JavaFX <https://github.com/elisasimoni/CosmoCity/blob/2778f6abfc86410977374c2458996cec2c1a2820/src/main/java/it/unibo/cosmocity/view/Lan>

Contribuzioni

- Come già segnalato nella class AudioManager si è preso spunto dalla guida <https://www.baeldung.com/java-play-sound>

- Gestione Immagini: http://www.java2s.com/Tutorial/Java/0261_2D-Graphics/ReadanImagefromim
- Gestione Serialization e Deserialization: <https://medium.com/@bubu.tripathy/json-serialization-and-deserialization-in-java-2a3f08266b70>

3.3.2 Yuri Collini

Elementi avanzati di Java utilizzati

- Utilizzo dei Thread Java : <https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html>

Contribuzioni

- Gestione Observer Pattern (preso ispirazione) : <https://refactoring.guru/design-patterns/observer>
- Factory Method (preso ispirazione) : <https://refactoring.guru/design-patterns/factory-method>

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

4.1.1 Elisa Simoni

Durante lo sviluppo del progetto ho riscontrato qualche difficoltà che però nel breve sono riuscita a risolvere. Questo è anche grazie al fatto che preparando nuovamente l'esame scritto mi ha portato a comprendere meglio alcuni concetti della programmazione a oggetti che ho subito messo in opera. Non nascondo anche il fatto che l'ansia, soprattutto in questo che è l'ultima tappa prima di conseguire la laurea, ha giocato un ruolo fondamentale soprattutto negli ultimi giorni. Questo mi ha portato a scrivere in maniera frettolosa e a volte poco ordinata portandomi di conseguenza a far fatica a leggere il mio stesso codice: questa è l'importanza di scrivere un codice pulito e con metodo. Una grande difficoltà l'ho avuto nel fattore di organizzazione e gestione del tempo in quanto sia io che il mio collega con cui ho condiviso il progetto siamo studenti-lavoratori (tempo pieno) e questo ha fatto sì che il tempo a disposizione per portare avanti il progetto insieme o anche singolarmente è diminuito in maniera drastica. Concludo dicendo che potrebbe essere buona innovazione creare strumenti per andare incontro agli studenti lavoratori del nostro settore in quanto anche rappresentano rispetto alle altre facoltà il numero più alto.

4.1.2 Yuri Collini

Durante lo svolgimento di questo progetto non nascondo che ci sono state alcune complessità e difficoltà.

Purtroppo sia io che la mia collega Elisa Simoni studiamo e lavoriamo a tempo pieno entrambi, questo elemento purtroppo ci ha portato a ridurre

il tempo a nostra disposizione, per poter realizzazione questo progetto in maniera considerevole. Sempre per il fattore tempo abbiamo lavorato in alcune occasioni anche ad orari proibitivi a causa dei nostri turni di lavoro.

Nonostante il fattore tempo a nostro svantaggio sono soddisfatto del risultato prodotto, anche se non esente da imperfezioni e sbavature.

Questo progetto mi è piaciuto particolarmente in quanto mi ha permesso a differenza di quello precedente (fatto anni fa), di apprendere nuovi concetti ed approfondire le mie conoscenze teoriche e pratiche pregresse, In particolare sono molto contento di aver scoperto due nuovi pattern che non conoscevo. Nello specifico mi riferisco all'Observer pattern, che è risultato per me molto importante all'interno del progetto per una corretta ed efficiente gestione del tempo e degli eventi, oltre che a migliorare la gestione della sincronizzazione tra interfaccia grafica e dati.

Capitolo 5

Guida utente

5.1 Main Menu

All'avvio del programma l'utente si troverà il menù principale composto da tre buttoni che hanno le seguenti funzionalità:

- New Game = inizia una nuova partita
- Load Game = carica una partita precedentemente salvata
- Exit = esce dall'applicazione

Si veda di seguito la figura che rappresenta quanto appena descritto

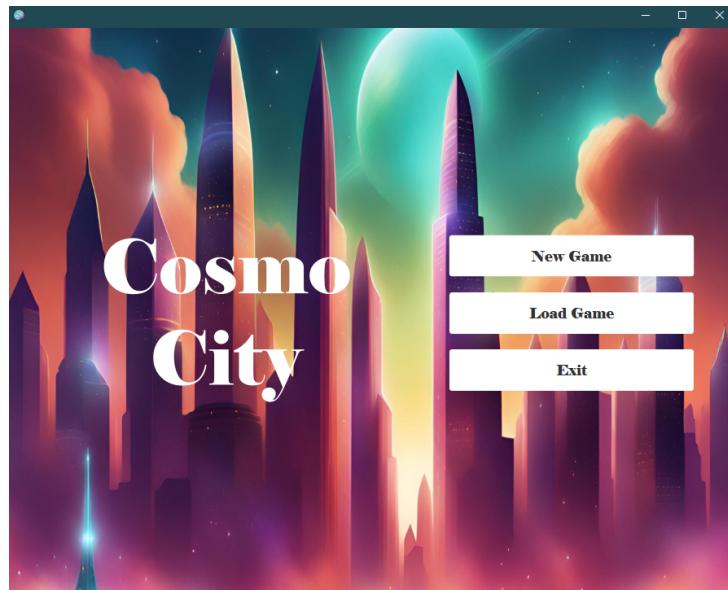


Figura 5.1: Main Menu

5.2 Impostazioni Partita

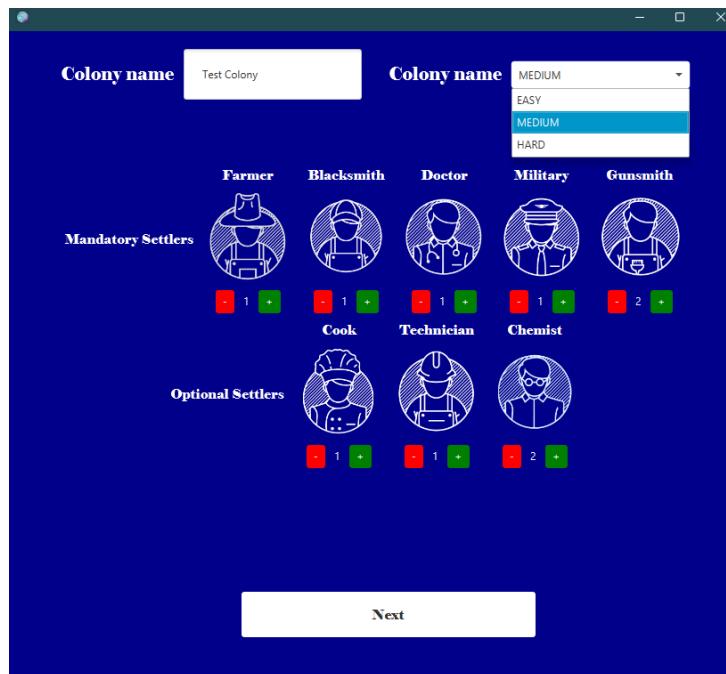


Figura 5.2: Impostazioni Partita

Dopo aver cliccato il comando "nuova partita" citato nel paragrafo precedente, l'applicativo vi presenterà una schermata in cui si potranno effettuare alcune scelte. Partendo dalla sezione in alto:

- Colony Name = Si inserisce il nome della colonia
- Difficoltà = tramite un menù a tendina si potrà scegliere il livello di difficoltà del gioco

Nella sezione centrale invece possiamo decidere dal numero di persone con cui la nostra colonia sarà composta e distinguiamo due categorie:

- Mandatory Settlers = producono una sola risorsa e solo nel settore in cui operano (esempio: 1 Farmer - 1 Food).
- Optional Settlers = producono in base al settore in cui operano producono ogni risorsa del settore.

Nota Bene Si consiglia all'utente di non superare il numero di 10 abitanti. Si consiglia di assegnare almeno un Mandatory Settlers per ogni categoria. Nella sezione in basso troviamo il comando "NEXT" che se cliccato farà iniziare la partita.

5.3 Inizio Partita



Figura 5.3: Dashboard

Iniziata la partita la nuova schermata si presenterà divisa in tre sezioni:

5.3.1 Sezione di Sinistra

Nel menu a sinistra troviamo quattro bottoni:

- Pausa = mette in pausa il gioco
- Save = permette di salvare la partita in corso
- Resoucers = mi apre un nuovo menu (si veda la figura di seguito) che mi da la possibilità di spostare gli **Optional Settlers** negli altri settori.

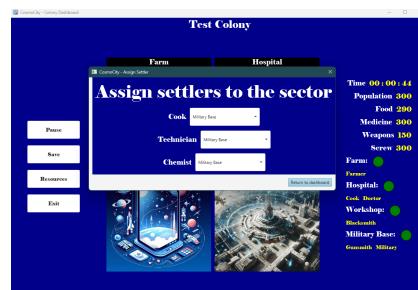


Figura 5.4: pulsante Resources

- Exit = comando per uscire dalla partita

5.3.2 Sezione Centrale

Nella sezione centrale troviamo quattro immagini che rappresentano le quattro categorie su cui si basa la colonia:

- Farm = La fattoria serve per produrre cibo
- Hospital = l'ospedale produce medicine
- Workshop = workshop produce viti
- Military Base = la base militare produce armi

5.3.3 Sezione di Destra

Nella sezione di destra sono rappresentati tutti i valori con cui l'utente dovrà lottare per fare sopravvivere la colonia:

- Time = indica da quanto tempo la partita è iniziata
- Population = indica il numero di abitanti che vivono nella colonia
- Food = indica il numero di cibo presente nella colonia
- Medicine = indica il numero di medicine presente nella colonia
- Weapons = indica il numero di armi presenti nella colonia
- Screw = indica il numero di viti presenti nella colonia

Troviamo sotto questa sezione nuovamente i quattro settori della colonia affiancati da un bollino(nella foto sono tutti e quattro verdi) che cambierà colore all'avvenire di un determinato evento legato al numero di **abitanti**:

- Bollino Verde = Indica che un determinato prodotto ha un valore superiore al 75 per cento rispetto alla popolazione
- Bollino Giallo = Indica che un determinato prodotto ha un valore compreso tra il 75 e il 50 per cento rispetto alla popolazione
- Bollino Rosso = Indica che un determinato prodotto ha un valore sotto il 50 per cento rispetto alla popolazione

5.4 Obiettivo e Funzionamento della simulazione

5.4.1 Obiettivo Principale

L'obiettivo principale della simulazione è quella di seguire nel tempo l'andamento della colonia.

5.4.2 Funzionamento

La simulazione basa la sua evoluzione sulla base di eventi che accadono nel tempo. In particolare:

- La popolazione ogni *tot* di tempo mangerà consumando una determinata quantità di cibo
- Si possono riscontrare eventi randomici che portano a decideri cosa sacrificare tra le risorse per sopravvivere all'evento in atto.
- La popolazione nel tempo si può ammalare causando una diminuzione della popolazione
- La popolazione nel tempo genera nascite causando l'aumento della popolazione

Nota Bene Sia i **Mandatory Settlers** che **Optional Settlers** mangiano ma al contrario della popolazione producono anche risorse.

Bibliografia

JavaFX Documentation: <https://openjfx.io>