

PixArt

Relazione del progetto Programmazione ad oggetti

Alessandro Ricci
Aldo Visconti
Beatrice Micolucci
Eduard Toni Alexandru

10 Giugno 2023

Indice

| | | |
|----------|--|-----------|
| 1 | Analisi | 3 |
| 1.1 | Requisiti | 3 |
| 1.2 | Analisi e modello del dominio | 4 |
| 2 | Design | 6 |
| 2.1 | Architettura | 6 |
| 2.1.1 | Model | 7 |
| 2.1.2 | Controller | 8 |
| 2.1.3 | View | 9 |
| 2.2 | Design dettagliato | 10 |
| 2.2.1 | Alexandru Eduard Toni | 10 |
| 2.2.2 | Micolucci Beatrice | 14 |
| 2.2.3 | Ricci Alessandro | 16 |
| 2.2.4 | Visconti Aldo | 20 |
| 3 | Sviluppo | 28 |
| 3.1 | Testing automatizzato | 28 |
| 3.1.1 | Alexandru Eduard Toni | 28 |
| 3.1.2 | Micolucci Beatrice | 28 |
| 3.1.3 | Ricci Alessandro | 29 |
| 3.1.4 | Visconti Aldo | 29 |
| 3.2 | Metodologia di lavoro | 29 |
| 3.2.1 | Alexandru Eduard Toni | 30 |
| 3.2.2 | Micolucci Beatrice | 31 |
| 3.2.3 | Ricci Alessandro | 31 |
| 3.2.4 | Visconti Aldo | 31 |
| 3.3 | Note di sviluppo | 33 |
| 3.3.1 | Alexandru Eduard Toni | 33 |
| 3.3.2 | Micolucci Beatrice | 34 |
| 3.3.3 | Ricci Alessandro | 34 |
| 3.3.4 | Visconti Aldo | 35 |
| 3.4 | Crediti | 35 |
| 4 | Commenti finali | 36 |
| 4.1 | Autovalutazione e lavori futuri | 36 |
| 4.1.1 | Alexandru Eduard Toni | 36 |
| 4.1.2 | Micolucci Beatrice | 36 |
| 4.1.3 | Ricci Alessandro | 37 |
| 4.1.4 | Visconti Aldo | 37 |
| 4.2 | Difficoltà incontrate e commenti per i docenti | 37 |
| 4.2.1 | Ricci Alessandro | 37 |

1 Analisi

Il Team si pone come obiettivo di creare un'applicazione intitolata PixArt. PixArt è un software che dà la possibilità all'utente di creare disegni in stile pixel art(da cui anche l'origine del nome).

”La pixel art è una forma di arte digitale, creata attraverso un computer, tramite l'uso di programmi di editing di grafica raster, in cui le immagini vengono modificate a livello di pixel. -Wikipedia”

Inoltre, tramite l'applicazione, l'utente potrà salvare i propri disegni creati e di giocare a dei minigame, sempre incentrati al mondo della pixel art.

1.1 Requisiti

Requisiti funzionali

- L'applicazione metterà a disposizione una griglia di pixel su cui l'utente potrà sviluppare i propri disegni attraverso l'utilizzo di diversi strumenti di disegno.
- Gli strumenti disponibili all'utente saranno i seguenti: Pencil, Eraser, Bucket, Darken, Lighten, Spray.
- Il software consentirà all'utente di gestire i progetti(Editare ed eliminare) da lui creati in modo semplice ed intuitivo.
- Il programma offrirà anche una modalità minigame dove l'utente potrà cimentarsi in sfide a tempo in cui mettere alla prova le sue abilità da disegnatore.
- L'applicazione consentirà di gestire più utenti contemporaneamente, ogni utente avrà associato il proprio percorso di salvataggio, sarà inoltre possibile accedere come guest(senza quindi necessità di registrazione) purchè si abbia selezionato un percorso di salvataggio.
- Durante l'utilizzo dell'applicazione sarà disponibile un pulsante ”UNDO” con il quale l'utente potrà riportare la griglia allo stato precedente.
- L'utente avrà la possibilità di scartare la griglia in uso in qualsiasi momento o di salvarla attraverso i pulsanti appositi.
- Durante la creazione di un progetto l'utente avrà la possibilità di scegliere il formato dell'immagine e la dimensione(in pixel) tra le opzioni prestabilite.
- Prima di salvare un progetto sarà possibile scegliere la scala delle immagini che rappresentano le griglie di un progetto.

Requisiti non funzionali

- L'applicazione dovrà garantire la sicurezza minima nel salvataggio delle password attraverso funzioni di hashing.
- L'organizzazione dei file dei progetti e delle immagini richieste dalle modalità minigame dovrà essere ottimizzata attraverso una strutturazione standard in modo da ridurre al minimo i tempi di ricerca.

1.2 Analisi e modello del dominio

L'applicazione dovrà essere in grado di gestire più utenti, di registrarli ed autenticarli. Ogni utente, chiamato **User**, avrà la possibilità di creare un progetto(**Project**) formato da una serie di matrici di pixel, chiamate **Matrix**. L'applicazione dovrà essere in grado di associare ad ogni progetto il relativo utente. PixArt dovrà rendere il progetto persistente in memoria, e renderlo disponibile in futuro. Ogni matrice di pixel verrà convertita in un'immagine del formato desiderato dall'utente(.png, .jpeg, .jpg). L'utente avrà a disposizione una serie di strumenti di disegno, chiamati **Tools**, tra cui scegliere, i quali coloreranno la matrice di pixel in modo diverso. Una volta modificata la matrice, essa dovrà poter ritornare agli stati precedenti. L'applicazione dovrà, inoltre, offrire la possibilità di concatenare le varie immagini create dall'utente per generare animazioni. Una funzionalità di PixArt sarà quella di poter far scegliere all'utente la velocità d'animazione(o delay) di ogni singola immagine. La principale difficoltà sarà quella di gestire la conversione del frame in file nel formato preferito dall'utente, e la gestione in memoria dei vari progetti creati dall'utente. L'applicativo offrirà due modalità di gioco, riguardanti la pixel art:

- **Mirror:** in cui l'utente dovrà replicare un'immagine mostrata sullo schermo.
- **Color book:** in cui l'utente dovrà colorare una matrice di pixel numerata. Ad ogni numero corrisponderà un colore, e l'utente dovrà colorare ogni pixel del colore associato al numero.

La seconda sfida sarà a tempo, perciò l'applicazione dovrà fornire una qualche forma di cronometro, definito **GameTimer**, con durata selezionabile dall'utente, e dovrà calcolare un punteggio in base alla correttezza del disegno generato.

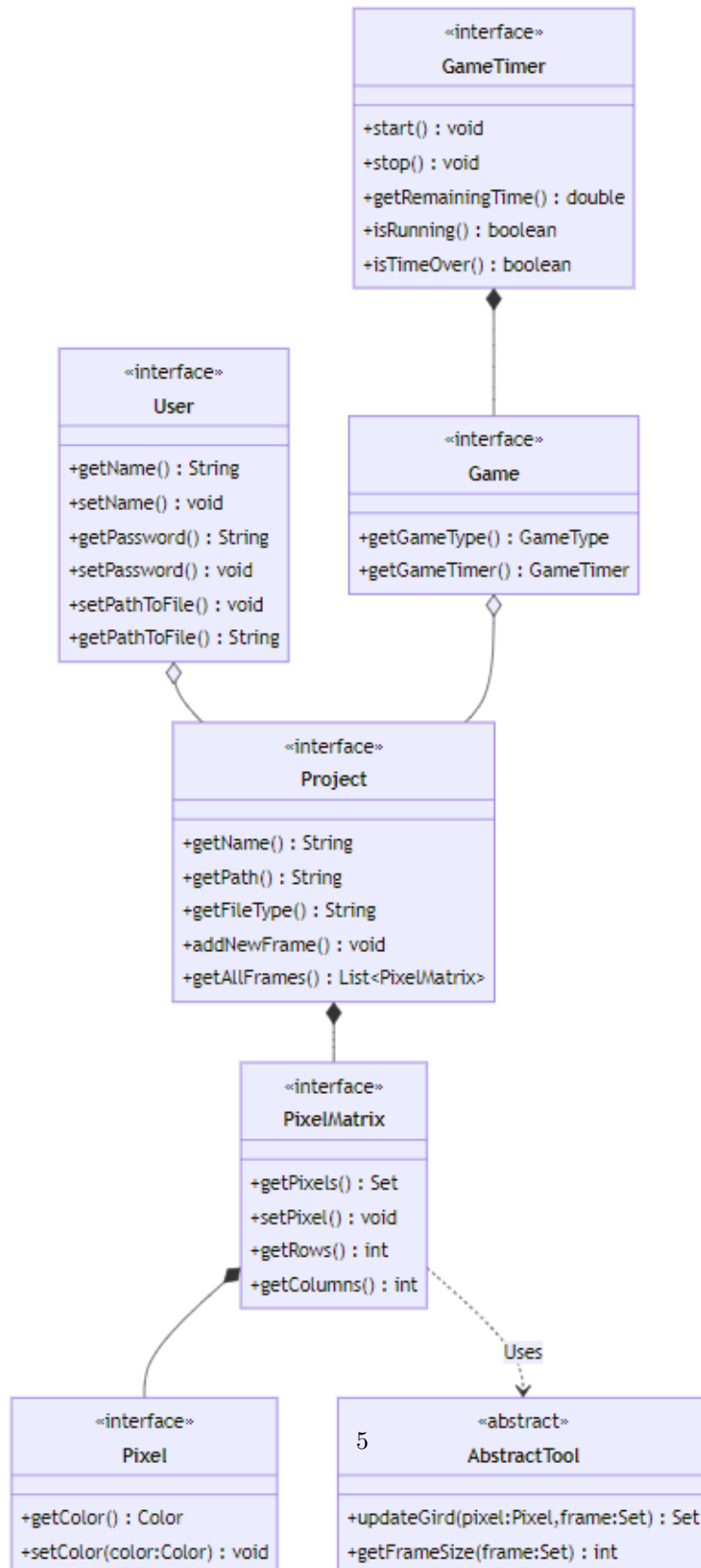


Figure 1: Schema UML del modello del dominio.

2 Design

2.1 Architettura

Per il progetto, abbiamo deciso di utilizzare un'architettura di tipo **MVC** (Model, View, Controller). Tale architettura ci consente di separare la parte di logica da quella grafica, dandoci la possibilità di rendere l'applicativo "cross-platform", senza dover apportare modifiche alla logica, ma soltanto alla parte grafica.

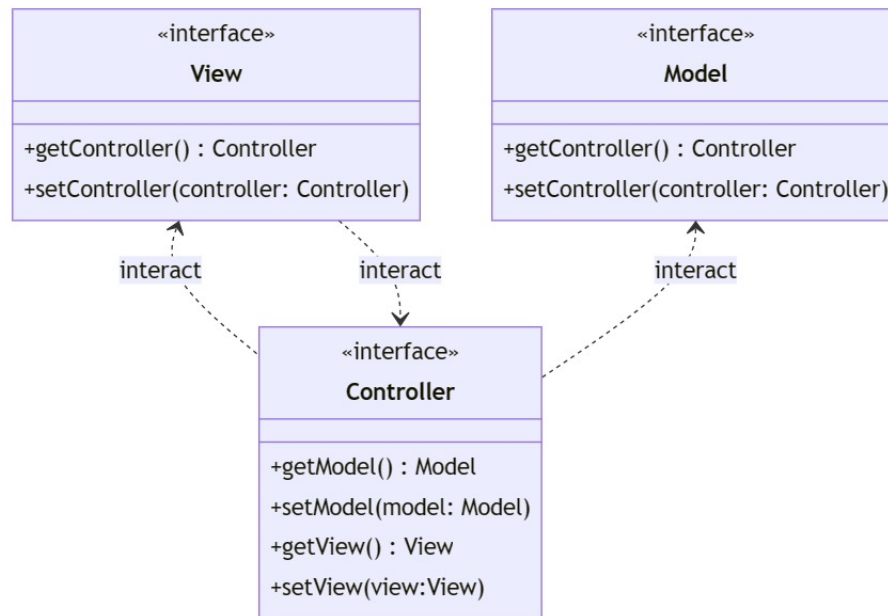


Figure 2: Schema UML dell'architettura MVC

Dopo un'attenta analisi, siamo riusciti ad individuare il numero delle possibili schermate a cui un utente potrebbe accedere. Ad ogni schermata abbiamo associato una View. Visto che ogni possibile View si comporta in modo diverso, e dovrebbe interagire con il Model in un modo diverso, abbiamo deciso di associare ad ogni View un proprio Controller specifico.

2.1.1 Model

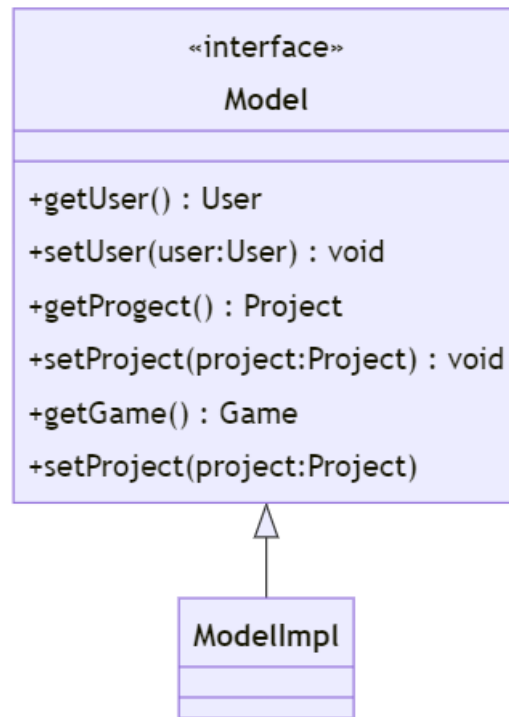


Figure 3: Schema UML del Controller

L'interfaccia **Model** racchiude la logica dell'applicativo, e contiene al suo interno i vari componenti logici. Il suo scopo è quello di fornire al Controller lo stato attuale dell'app. Il **Model** è singolo per tutto l'applicativo. Implementato nella maniera giusta, il Model dovrebbe poter essere riutilizzato con diverse view/controller. In questo caso, il Model può essere visto come un wrapper, che contiene al suo interno i componenti in grado di rappresentare un gameplay oppure il contesto di un user che lavora su un suo progetto. L'implementazione di tale interfaccia viene fatta dalla classe **ModelImpl**.

2.1.2 Controller

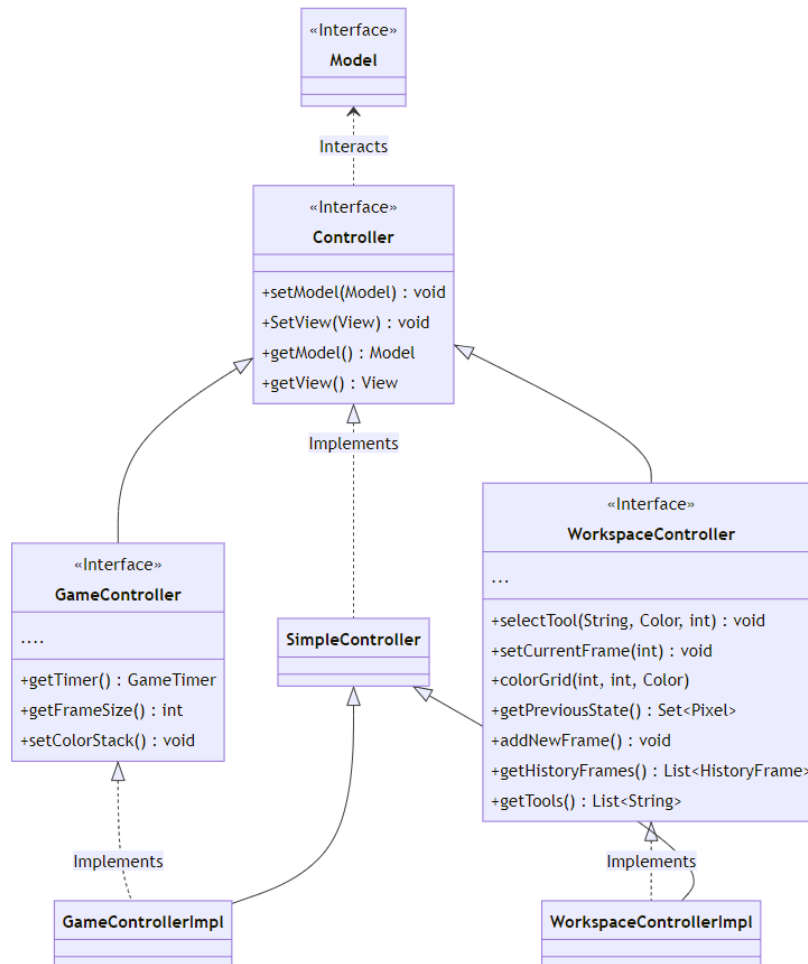


Figure 4: Schema UML del Controller

Il **Controller** permette il coordinamento tra il *Model* e la *View*. Riceve dal *Model* informazioni riguardo allo stato attuale dell'app, le elabora e informa la *View* in modo che questa possa essere aggiornata.

Abbiamo deciso di non utilizzare un unico *Controller*. Abbiamo infatti realizzato una struttura gerarchica in cui l'interfaccia *Controller*, che presenta i metodi principali per accedere a *View* e *Model*, viene poi specializzata da altre classi che comunicano ognuna con le relative *View* e i relativi *Model*.

Sono presenti alcune *View*, tuttavia, che non necessitano di particolari elaborazioni, pertanto abbiamo realizzato la classe **SimpleController** che estende

Controller, implementando quindi solo le funzioni di base per l'accesso a View e Model.

2.1.3 View

Abbiamo deciso di creare un numero di view pari al numero di schermate possibili a cui un utente può accedere. Lo scopo di ogni view è di comunicare al Controller eventuali cambiamenti avvenuti. Il Controller, a sua volta, va a modificare il Model. Per le View abbiamo deciso di utilizzare JavaFX; abbiamo scelto di non collegare direttamente l'interfaccia del Controller all'implementazione della View e quindi il file fxml al proprio "Controller" ma direttamente alla View; questo per far in modo che, separando il Controller dal framework utilizzato, in caso di cambio di libreria grafica si debbano cambiare solamente le classi di View, senza modificare quelle del Model e del Controller.

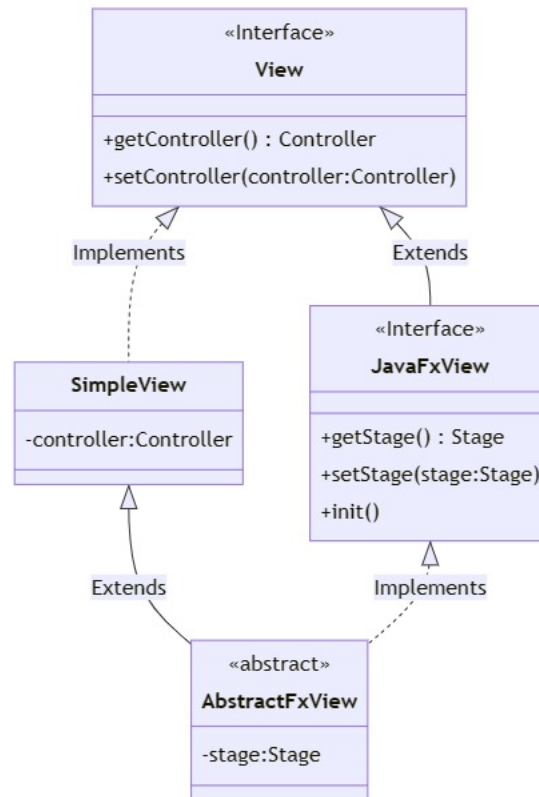
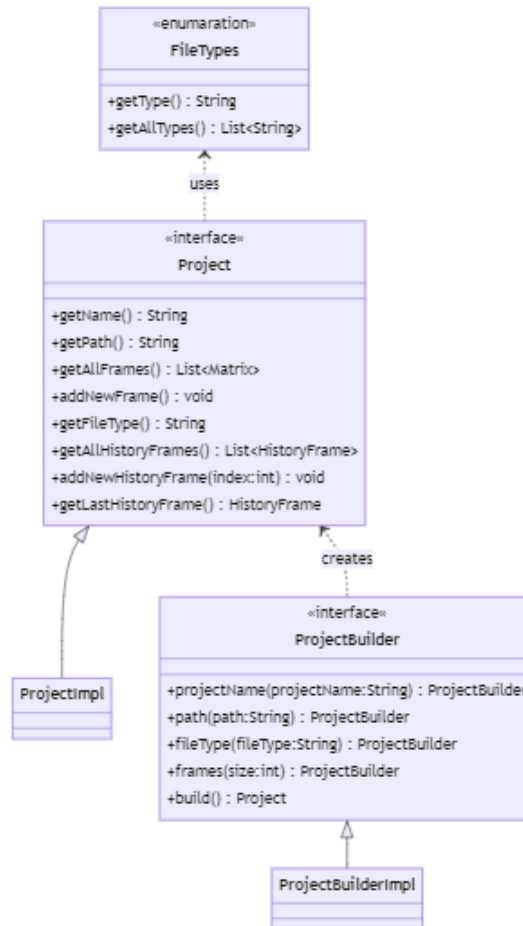


Figure 5: Schema UML della View

2.2 Design dettagliato

2.2.1 Alexandru Eduard Toni

Project



Problema: Uno dei primi problemi che abbiamo dovuto affrontare è stato creare una classe che potesse rappresentare la struttura di un progetto, considerando che ogni progetto deve avere:

- Un **nome**.
- Un **path** dove venga salvata una rappresentazione persistente del progetto stesso e le immagini relative.
- Il **formato** delle immagini relative.

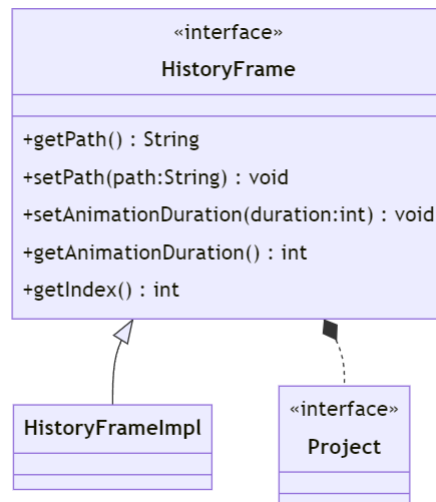
- La lista di **matrici** di pixel.
- La lista di **immagini**.

Soluzione: Avendo in mente i diversi requisiti, ho deciso di creare l'interfaccia **Project**, implementata dalla classe **ProjectImpl**. Tale classe contiene tutti i campi elencati precedentemente, con i relativi getters. Per quanto riguarda la lista di matrici di pixel, ho aggiunto come campo una lista di Matrix, e per la lista di immagini ho usato una lista di HistoryFrame, classe che analizzerò più in dettaglio successivamente. Grazie alla struttura scelta, è risultata triviale la conversione in .json di ogni istanza di Project, senza avere perdite di informazione.

Pattern: Vista la complessità della classe e i suoi numerosi campi, ho deciso di utilizzare il pattern **Builder**, per delegare la costruzione ad una classe specializzata. Tale pattern viene relizzato tramite l'interfaccia **ProjectBuilder**, e la sua implementazione **ProjectBuilderImpl**.

HistoryFrame

Problema: Una delle funzionalità dell'app è quella di poter generare animazioni, partendo dalle immagini di un progetto. L'animazione stessa avviene alternando le immagini ad intervalli di tempo (delay) anche diversi tra loro. Inoltre, nel Workspace, l'utente dovrebbe avere a disposizione una lista di immagini, ognuna delle quali corrispondente ad una Matrix.

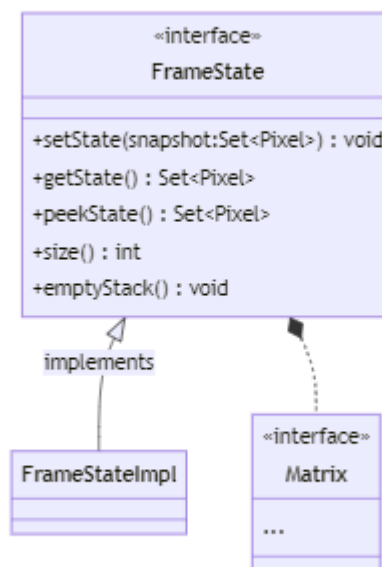


Soluzione: Dunque ho deciso di creare l'interfaccia **HistoryFrame**, con la relativa implementazione **HistoryFrameImpl**. Ogni istanza di un HistoryFrame contiene il path dell'immagine relativa e il delay di animazione. Ogni Project è legato ad uno o più HistoryFrame tramite composizione.

Pattern: Vista la semplicità della classe, non è stato necessario utilizzare alcun design pattern.

FrameState

Problema: Una delle funzionalità dell'app è quella di poter salvare i vari stati di una Matrix, e di poterla riportare ad uno stato specifico.

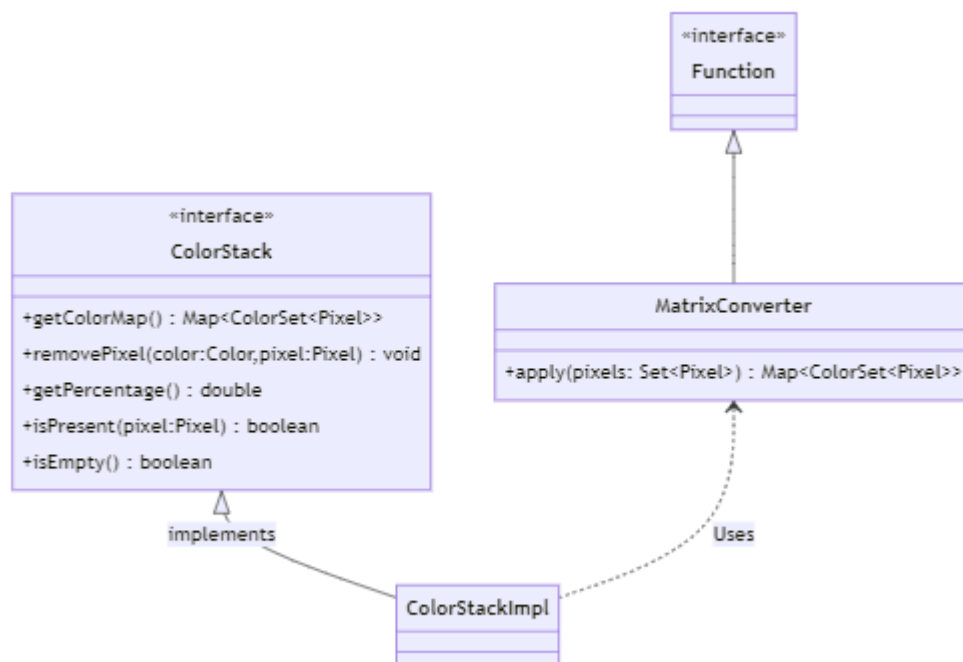


Soluzione: Dunque ho deciso di creare l'interfaccia **FrameState**, con la relativa implementazione **FrameStateImpl**, il cui compito è quello di memorizzare i diversi stati di una Matrix. Ogni volta che la Matrix viene modificata, il FrameState memorizza l'attuale stato all'interno di uno Stack. Ho deciso di utilizzare lo Stack grazie alla sua politica LIFO.

Pattern: Ho applicato il pattern **Memento**, in cui il *memento* è la classe FrameState e l'*originator* la classe Matrix.

ColorStack e MatrixConverter

Problema: Per la modalità di gioco "Color Book", è necessario avere una classe che elenchi tutti i colori presenti in una particolare Matrix, e tutti i pixel associati a quel determinato colore. Durante il gameplay, il player avrà una rappresentazione grafica di tale classe, da cui potrà selezionare il colore. Tale classe si occuperà anche di controllare la correttezza dei Pixel colorati in fase di gameplay, rimuoverli dall'elenco e di fornire un punteggio.



Soluzione: Perciò ho deciso di creare l'interfaccia **ColorStack**, implementata da **ColorStackImpl**. ColorStack mantiene i vari colori e i relativi Pixel all'interno di una mappa. Durante il gameplay, quando l'Utente seleziona un Pixel, la classe controlla l'esistenza di esso nella mappa e lo rimuove nel caso fosse presente. A fine del gameplay, il ColorStack è in grado di fornire un punteggio, pari alla percentuale di Pixel colorati correttamente. Visto che la mappa del ColorStack va inizializzata partendo da un set di Pixel, ho deciso di delegare l'operazione di conversione ad un'altra classe, chiamata **MatrixConverter**. MatrixConverter è una classe che implementa l'interfaccia Function, e si occupa di convertire i set di Pixel di una Matrix in una mappa. In questo modo, non si rischia di aggiungere troppa logica al costruttore di ColorStackImpl.

2.2.2 Micolucci Beatrice

Tools

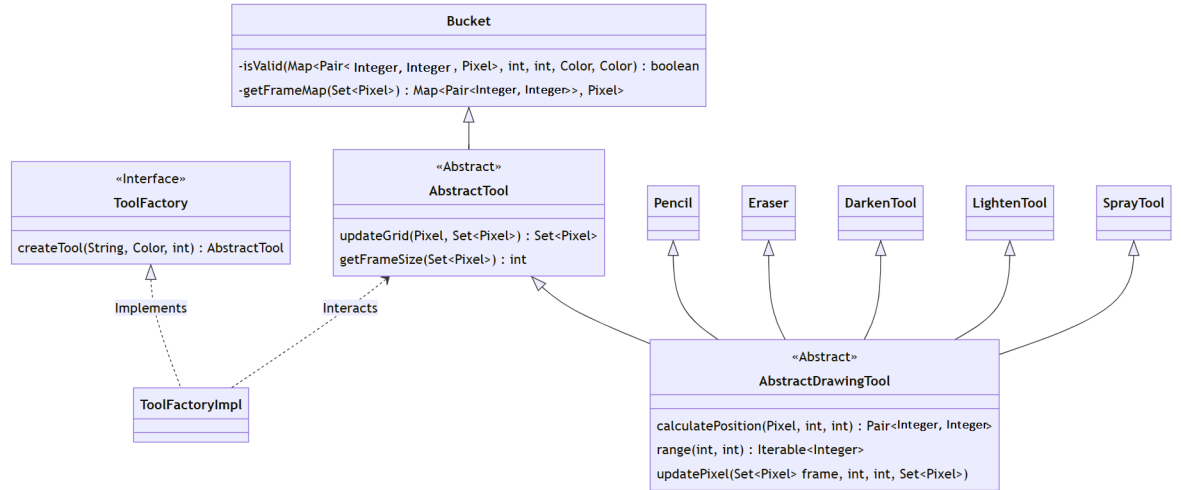


Figure 6: Creazione dei tools

L'applicazione permette l'utilizzo di diversi tools, i quali hanno tutti come funzione quella di aggiornare il colore di uno o più pixel. Ho pertanto deciso di utilizzare il pattern **Factory** tramite l'interfaccia **ToolFactory**. Questa ha il compito di creare delle istanze della classe **AbstractTool**.

Ogni tool implementa, a seconda della sua funzione, il metodo per aggiornare la matrice di pixel; per questo ho deciso di utilizzare una classe astratta che viene estesa dalle varie classi che implementano i tools.

In particolare, data la presenza di alcuni metodi comuni a tutti i tools tranne al **Bucket**, ho implementato una classe **AbstractDrawingTool**, anch'essa astratta, che viene estesa da tutti i tools ad eccezione di **Bucket** che estende direttamente **AbstractTool**.

User

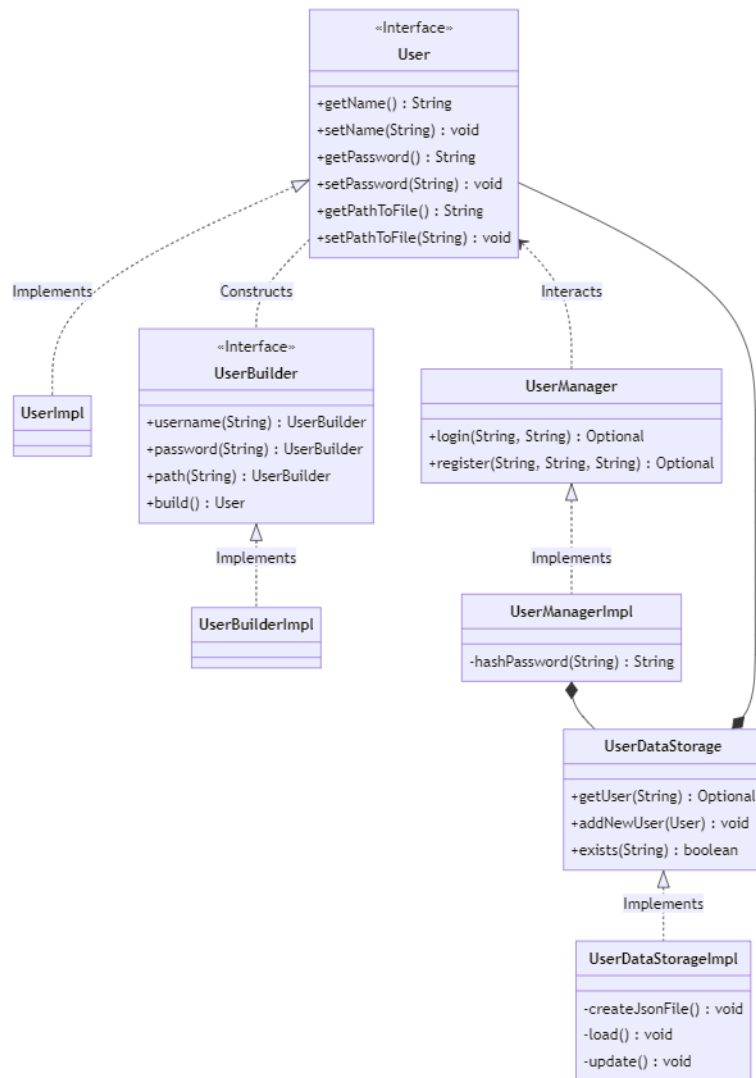


Figure 7: Gestione salvataggio degli User e dei dati associati

L'applicazione permette all'utente di registrarsi ed effettuare il login. A tal scopo ho implementato una classe **User** che mantiene le informazioni sull'utente, quali username, password e il path in cui verranno salvati i progetti da lui creati. Per la creazione delle istanze di **User** ho utilizzato il pattern **Builder** il quale permette di semplificare la manutenzione del codice: infatti se in futuro si vorranno apportare delle modifiche alla classe **User** basterà estendere la classe Builder. Ogni user ha un username che deve essere univoco. I dati degli utenti registrati vengono salvati all'interno di un file JSON. La memorizzazione di

tali dati viene gestita attraverso la classe **UserDataStorage** che definisce le operazioni che possono essere effettuate sul file. Questa viene utilizzata da **UserManager** che gestisce il login e la registrazione di un nuovo utente.

Timer

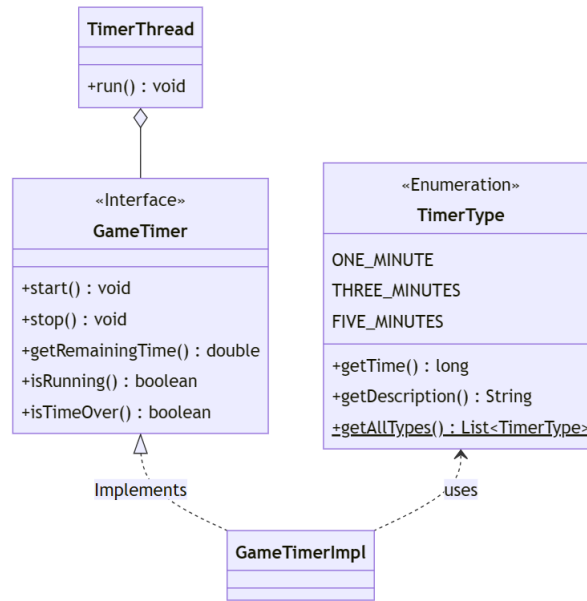


Figure 8: Gestione Timer per il game

All'interno dell'applicazione abbiamo deciso di realizzare un mini game che necessita di un timer. L'utente può scegliere vari tipi di timer che differiscono per il tempo messo a disposizione per completare il gioco.

La gestione del tempo viene delegata alla classe **GameTimer** che permette di far partire il timer, fermarlo e ottenere il tempo rimanente. Il timer viene realizzato tramite **TimerThread**, una classe che estende **Thread**.

2.2.3 Ricci Alessandro

Siccome la nostra applicazione si basa sulla creazione di immagini, una parte importante del progetto è quella della gestione delle immagini e dei file che rappresentano i progetti creati dall'utente. Per questo motivo ho dedicato una parte importante del tempo allo sviluppo di queste funzionalità.

Image Printer

Come già detto la nostra applicazione ha come obiettivo principale quello di creare immagini in Pixel Art; per fare ciò è stata creata la classe **ImagePrinter**.

In questa classe è stato utilizzato il pattern **Singleton** per far sì che, durante l'utilizzo dell'applicazione, venga garantita la creazione di una sola istanza e per rendere univoco l'accesso ad essa.

Nonostante le sue funzionalità non siano utilizzate pesantemente nella nostra applicazione ho preferito creare una classe a parte per evitare ripetizioni poco efficienti.

Per evitare che la immagini risultino poco nitide (soprattutto quelle con dimensioni di 16 pixel) è stato necessario implementare la funzionalità che rende possibile la scelta della scala delle immagini, preservandone la dimensione di pixel visibili, ma aumentandone la risoluzione.

File Handler

Anche per **FileHandler** ho deciso di utilizzare il pattern **Singleton** per le stesse motivazioni sopra elencate. Per poter utilizzare alcune sue funzionalità è stato necessario implementare la classe **Interface Serializer** in modo da poter registrare il TypeAdapter per interagire con le diverse interfacce implementate nel progetto.

Oltre alla gestione dei file JSON questa classe ha anche il compito di gestire le cartelle in cui vengono immagazzinati, eliminandole o creandole quando necessario.

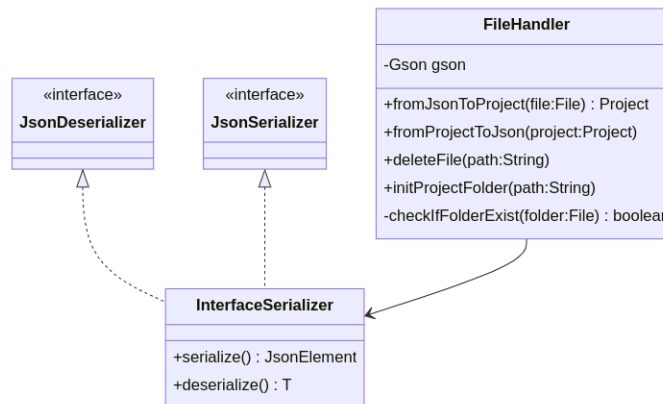


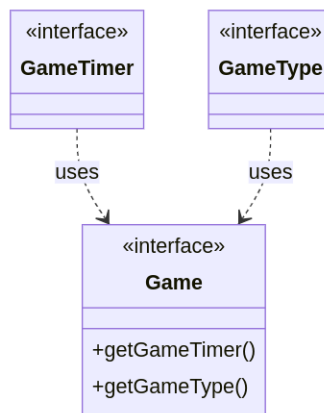
Figure 9: Gestione dei file tramite FileHandler

Game

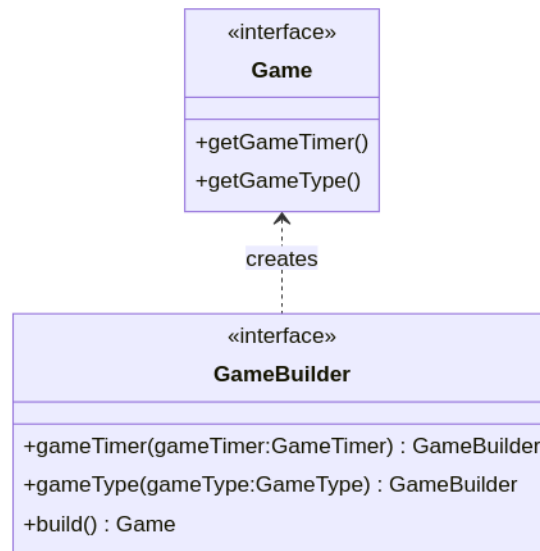
La nostra applicazione deve supportare due tipologie di minigame, **ColorBook** e **Mirror**.

Il game è principalmente composto da 2 elementi:

- **GameType**: una variabile, impostata tramite una enumeration, composta dal nome del Game e da una descrizione di come si svolge;
- **GameTimer**: un timer, scelto dall'utente prima iniziare una partita nel game.



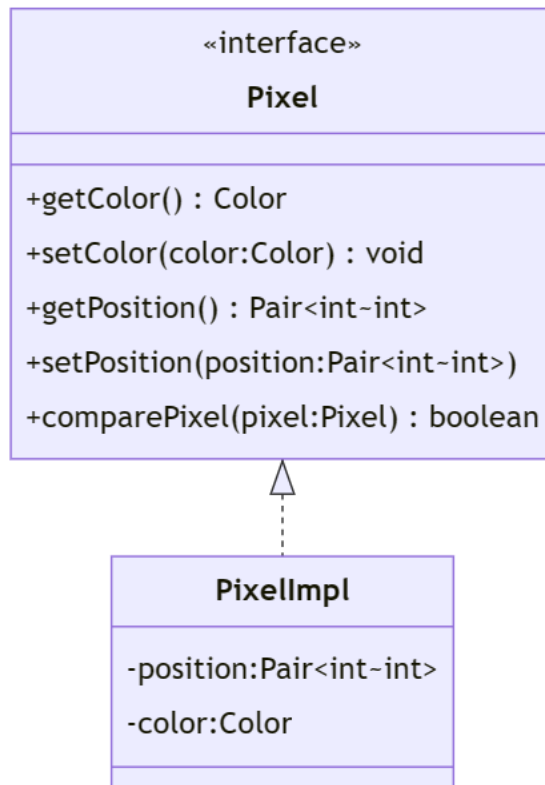
Per la creazione del Game ho deciso di utilizzare il pattern **Builder**, questo per fare in modo che la sua costruzione tramite l'interfaccia **GameBuilder** sia separata dalla rappresentazione nell'interfaccia **Game**. Nonostante non siano molti i campi che serve inizializzare nel Game ho preferito utilizzare comunque questo pattern, oltre per il motivo elencato in precedenza, anche per rendere più leggibile la parte di codice.



Per il game è stata implementata la classe **GameLevels** dove vengono immagazzinati i dati dei diversi livelli dei minigiochi rappresentati con una enumeration.

2.2.4 Visconti Aldo

Pixel

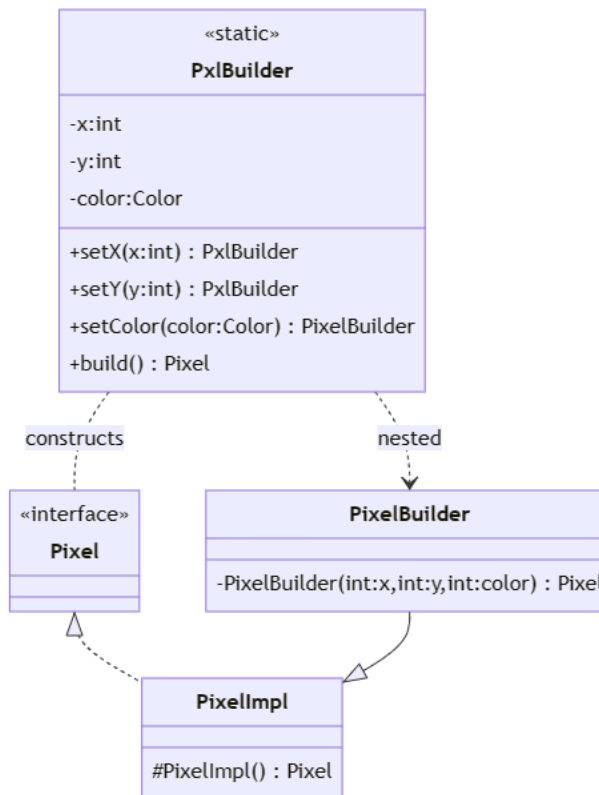


Problema: La rappresentazione nel model delle unità elementari di cui è composta la matrice su cui l'utente disegna, ovvero i "Pixel", i quali devono avere:

- Un **colore**.
- Una **posizione** che indichi l'ubicazione all'interno della matrice.

Soluzione: Ho deciso di creare l'interfaccia **Pixel**, implementata dalla classe **PixelImpl**. Tale classe contiene tutti i campi elencati precedentemente, con i relativi getters e setters.

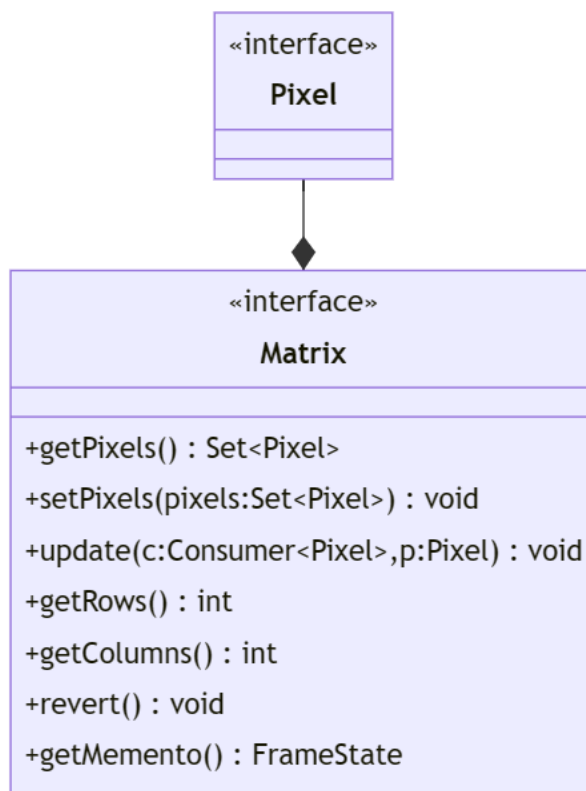
Pattern: Nonostante la semplicità dell'entità, ho deciso di utilizzare il pattern **Builder**, per delegare la costruzione ad una classe specializzata **PxlBuilder**. Tale scelta è stata effettuata per limitare l'accesso al costruttore della classe **PixelImpl**, probabilmente non è una scelta ottimale, ma consente di creare entità leggermente più complesse, che però facilitano la fase di testing, la quale necessita l'utilizzo di molti "Pixel" differenti tra loro.



Matrix

Problema: La rappresentazione nel model della matrice su cui l'utente disegna, la quale deve possedere:

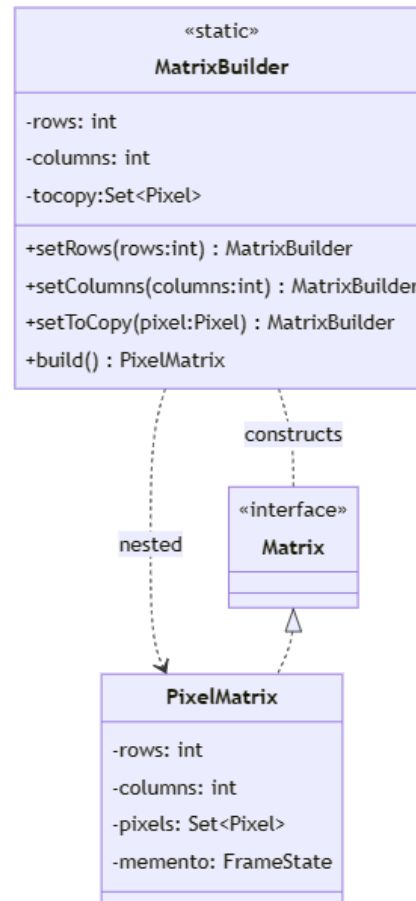
- Delle **colonne**.
- Delle **righe**.
- Un **elenco** dei pixel che la compongono.



Soluzione: Un'interfaccia **Matrix**, implementata dalla classe `textbfPixelMatrix`. Tale classe contiene tutti i campi elencati precedentemente, con i relativi getters e metodi che permettono di aggiornare lo stato di quest'ultima e salvare gli stati precedenti.

(Non tratterò in questa sezione il pattern che salva gli stati precedenti e che consente la funzione **UNDO** in quanto realizzata dal mio collega Alexandru)

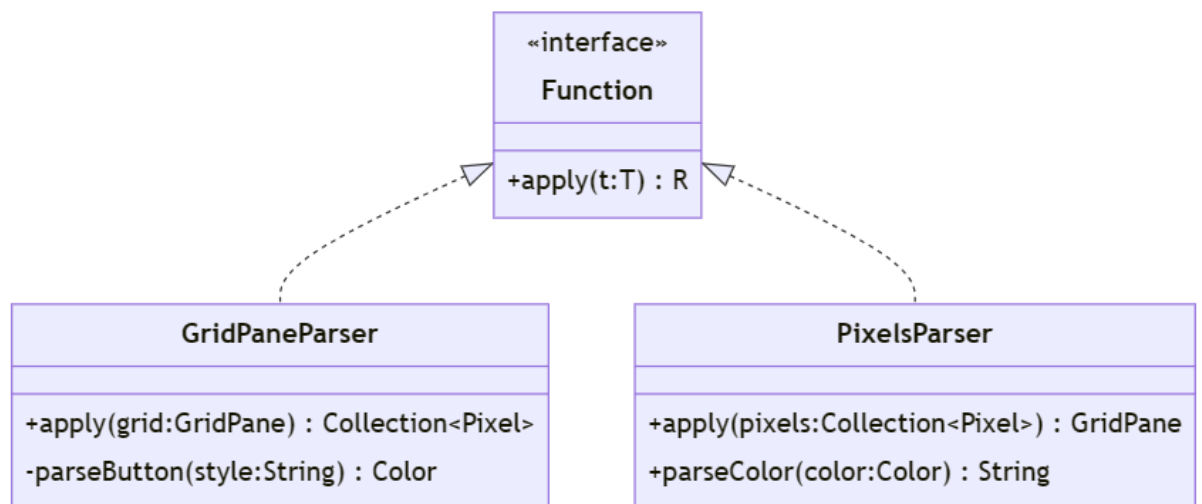
Pattern: anche qui come in precedenza ho utilizzato il pattern **Builder** attraverso una classe inestata **MatrixBuilder** all'interno di **PixelMatrix** limitando anche qui la visibilità del costruttore, questa volta non a livello di package ma a livello di classe.



Parser

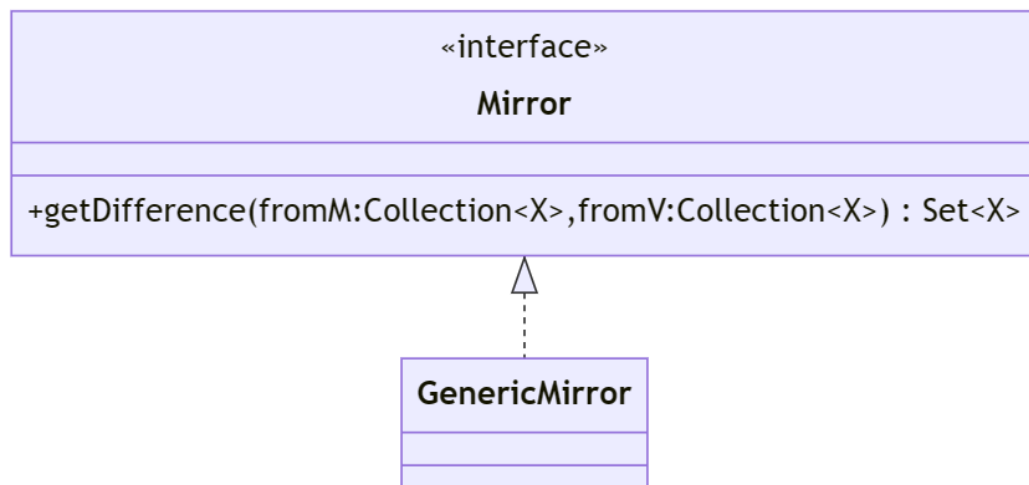
Problema: Avendo scelto di rappresentare la matrice nel model sotto forma di **PixelMatrix**, per il **Controller** ed altre classi risultava abbastanza macchinosa la conversione da componente grafico di javafx a **PixelMatrix** e viceversa.

Soluzione: Due classi che implementano l'interfaccia **Function**.



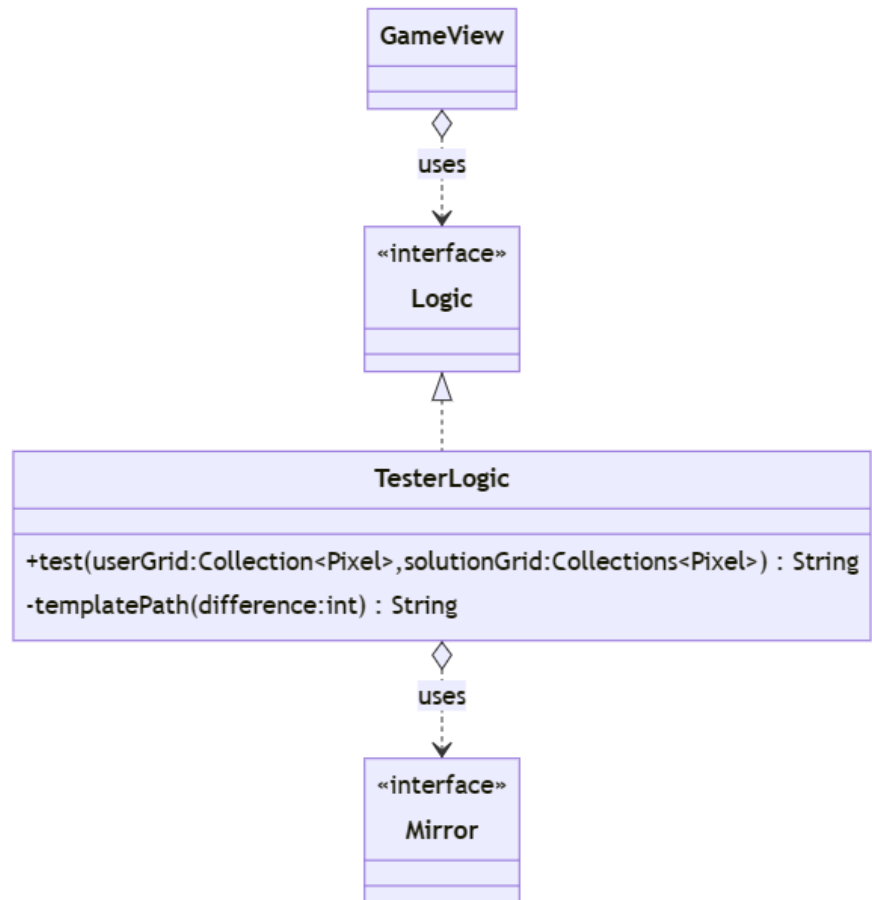
Mirror

Problema: Dopo aver deciso di creare il minigioco **Mirror**, è sorta la necessità di avere le differenze tra il disegno che l'utente avrebbe dovuto riprodurre ed il disegno realizzato dall'utente, per assegnare un punteggio.



Soluzione: Un'interfaccia **Mirror**, implementata dalla classe **GenericMirror**; la classe si compone di un solo metodo, il quale restituisce un **Set** contenente tutti gli oggetti presenti nella **Collection fromM** che non sono presenti nella seconda **Collection fromV**, per poter utilizzare questa classe in maniera efficace è necessario che le classi generiche "**X**" abbiano fatto l'**override** del metodo **equals**.

Pattern: Nel minigame il punteggio non ha valore numerico, ma è rappresentato dal meme **"Mr.Incredible becoming uncanny"**, viene quindi mostrata una versione differente di Mr.Incredibile a seconda della correttezza del disegno fatto dall'utente. Utilizzo quindi uno **Strategy** all'interno della **GameView**, per stabilire il **path** dell'immagine da mostrare.

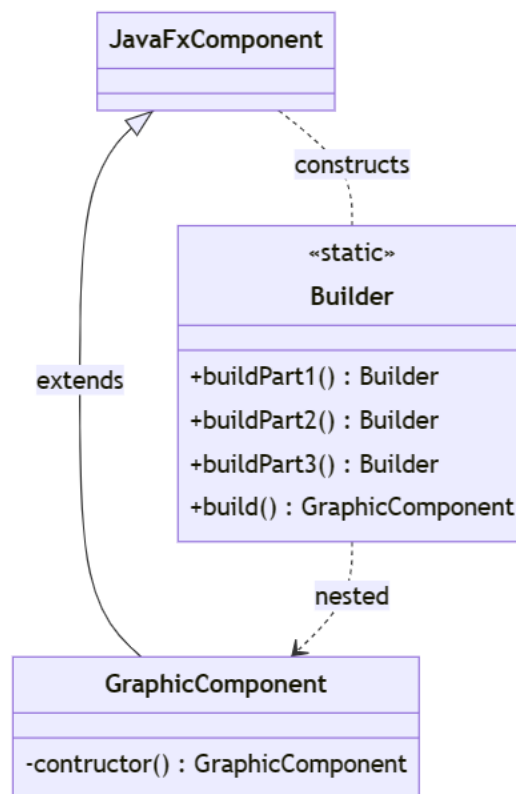


GameView usa Logic per sapere quale immagine dovrà mostrare, **Tester-Logic** stabilisce il **path** dell'immagine, utilizzando **Mirror** per conoscere le differenze tra il disegno corretto e quello fatto dall'users.

Components Package

Problema: All'interno delle classi della view era necessario inizializzare componenti grafici di **javafx** molto complessi, questa operazione impiegava un notevole numero di righe rendendo il codice prolisso e a mio parere meno comprensibile.

Soluzione: Ho deciso quindi di creare diverse classi le quali estendono ognuna un componente differente di **javafx**, all'interno di ognuna ho poi innestato una classe che fungesse da **Builder**.



3 Sviluppo

3.1 Testing automatizzato

Ogni membro del gruppo si è occupato di eseguire i test riguardanti le proprie classi, dando particolare valore ed importanza a tutti i componenti costituenti del Model, quindi la parte logica. Una volta testata una classe, e controllando che i vari test dessero esito positivo, si procedeva con la sua integrazione al resto del progetto e consecutivo push sul branch main. Per quanto riguarda tutta la parte di view, siamo riusciti ad individuare bug e problemi testando manualmente l'applicativo. Per la scrittura dei test, abbiamo utilizzato il framework JUnit 5.

3.1.1 Alexandru Eduard Toni

Durante la fase di testing, ho deciso di dedicare particolare enfasi alle classi costituenti del model; quindi mi è sembrato di vitale importanza testare il **Project** e la sua corretta costruzione tramite il **ProjectBuilder**. Vista la semplicità della classe **HistoryFrame**, ho deciso di non testarla singolarmente, ma nel contesto di un Project. Di seguito, la lista di test:

- ColorStackTest
- FrameStateTest
- MatrixConverterTest
- ProjectTest
- ProjectBuilderTest

3.1.2 Micolucci Beatrice

Avendo sviluppato i tools necessari per disegnare sulla griglia di pixel, ho pensato fosse importante realizzare dei test per verificarne il corretto funzionamento. Ho, inoltre, realizzato dei test per verificare il corretto salvataggio, all'interno di un file JSON, degli user.

- AbstractDrawingToolTest
- AbstractToolTest
- BucketTest
- PencilTest
- UserDataStorageTest
- UserImplTest
- UserManagerImplTest

3.1.3 Ricci Alessandro

Siccome una mia parte del progetto consisteva nella gestione dei salvataggi dei progetti creati dall'utente, ho ritenuto importante il test dei metodi che si occupano di questi ultimi. Un'altra parte dei test è dedicata al testing del Model del Game e a quello della sua costruzione attraverso il Builder descritto in precedenza.

- FileHandlerTest
- GameTest

3.1.4 Visconti Aldo

- PixelMatrixTest
- ImplPixelTest
- PixelBuilderTest
- GenericMirrorTestq

3.2 Metodologia di lavoro

La prima parte del processo è consistita in una fase di "brainstorming". Dunque, abbiamo dedicato una buona quantità di tempo a discutere e a confrontarci. Abbiamo elencato le diverse funzionalità che avremmo voluto che l'applicativo fosse in grado di eseguire, e perciò siamo riusciti ad individuare i principali componenti che avrebbero formato il **Model**. Vista la natura del progetto, siamo stati in grado di predire in anticipo il numero di **View** che l'app avrebbe avuto, e dunque anche il numero di **Controller** relativi. Una volta ottenuto uno schema generale della struttura dell'app, abbiamo proseguito con la suddivisione del lavoro e l'assegnazione dei vari componenti. Ogni membro del gruppo, dunque, ha avuto il compito di creare e gestire la View e il Controller relativa alla propria parte di Model. Essendo i vari componenti abbastanza isolati tra di loro, ognuno ha lavorato in relativa autonomia, tranne nei casi in cui ci fosse una forte interazione tra le classi (ex: **FrameState** e **Matrix**). I primi componenti ad essere sviluppati sono stati il **Project**, il **Matrix** e il **Pixel**, essendo il "nocciolo" del Model dell'applicativo, e la view **WorkSpace** e relativo controller **WorkSpaceController**. Abbiamo proceduto con l'implementazione dell'intero workflow di salvataggio di un determinato Project. Le funzionalità di login/registrazione e l'intero workflow di gameplay sono state implementate successivamente. Una volta implementati i requisiti obbligatori dell'app, siamo passati da una metodologia di lavoro simile alla tipologia "*Waterfall*" ad una più "*Agile*", sviluppando ogni micro-funzionalità in piccole finestre di tempo. Durante questa fase è diventata molto più frequente la collaborazione tra i membri del gruppo. Ogni implementazione di una certa funzionalità veniva seguita dalla sua integrazione con il resto del progetto. Durante la fase implementativa

del progetto abbiamo utilizzato Git come DVCS. Visto la buona separazione del lavoro, abbiamo deciso di usare soltanto un branch, il main, su cui venivano "pushati" i nuovi cambiamenti. Non sono risultati particolari problemi di merge conflicts, a prova della suddivisione del lavoro e della comunicazione all'interno del gruppo.

3.2.1 Alexandru Eduard Toni

In autonomia mi sono occupato di:

- Creazione dell **Project** con relativo builder **ProjectBuilder**, ed enum di possibili formati di file immagine **FileTypes**.
- Creazione delle classi **HistoryFrame** e **FrameState**, analizzate in dettaglio precedentemente.
- Creazione della classe **ColorStack** e **MatrixConverter**.
- Creazione della view di animazione **AnimationView** e relativo controller **AnimationControllerImpl**.
- Classe **Animator** che estende la classe **Thread** e si occupa di animare le immagini di un progetto, alternandole. E' una classe innestata all'interno del **AnimationController**.
- Creazione della view di setup di progetto, in cui l'utente sceglie le specifiche per un determinato progetto(Nome, Path,Formato..), chiamata **SettingsView**, con relativo controller **SettingsController**.
- Creazione della view di setup di game, in cui l'utente decide il timer, la modalità di gioco e il disegno da replicare, chiamata **GameSetup** e relativo controller **GameSetupControllerImpl**.

In collaborazione ho lavorato:

- con Visconti Aldo:
 - Sulla view di lavoro **WorkspaceView**, in cui l'utente crea i propri disegni.
 - Sulla view di gameplay chiamata **GameView**, in cui Visconti ha implementato la grafica per la modalità di gioco "Mirror".
- con Micolucci Beatrice:
 - Sulla **GameView**, in cui abbiamo implementato la grafica per la modalità di gioco "Color Book", in particolare sull'inizializzazione della matrice di pulsanti.

Per quanto riguarda le classi **HomeView**, **Model**, **ModelImp**, **SimpleController**, **Controller**, abbiamo lavorato tutti insieme, comunicando tra di noi nel caso di avvenute modifiche.

3.2.2 Micolucci Beatrice

In autonomia:

- Realizzazione delle classi che implementano e gestiscono i tools.
- Realizzazione delle classi che permettono il salvataggio degli utenti in memoria, quindi implementazione del **LoginController** e realizzazione della grafica relativa al login, ovvero implementazione della classe **LoginView**.
- Realizzazione delle classi che implementano il **Timer** per il game.

In collaborazione:

- Realizzazione della **GameView**, in particolare la parte riguardante la modalità di gioco *Color Book*.

3.2.3 Ricci Alessandro

Per quanto riguarda il mio lavoro in autonomia mi sono focalizzato su:

- Realizzazione **FileHandler**, classe che gestisce la creazione e il caricamento dei file JSON.
- Realizzazione **ImagePrinter**, classe che stampa le immagini e le salva nei 3 formati possibili(PNG, JPG, JPEG).
- Realizzazione della ProjectView e del suo controller.
- Realizzazione della parte di Model del **GameBuilder** e del **Game** e della relativa parte di Controller.

3.2.4 Visconti Aldo

Il lavoro svolto in autonomia comprende i file presenti nei seguenti package:

- **pixart/model/grid**
- **pixart/model/pixel**
- **pixart/utilities/mirror**
- **pixart/utilities/parser**
- **pixart/view/components**
- **pixart/view/abilitytest**

Il mio lavoro si concentra, quindi, principalmente nel model, view e nelle classi di utilità. Vorrei sottolineare che le classi presenti nel package **pixart/view/components** si trovano in uno stato embrionale, e **necessitano** un affinamento. Per quanto riguarda la collaborazione con il gruppo mi ritengo più che soddisfatto, una volta stabilite le interfacce è stato facile lavorare in maniera autonoma, Oltretutto ho notato che l'utilizzo dei pattern facilita il lavoro in collaborazione, infatti una volta distribuite le classi che devono interagire è molto difficile creare conflitti nello sviluppo (Es. **PixelMatrix**_i—**FrameState**_i—FrameStateImpl, per la funzione **UNDO**).

3.3 Note di sviluppo

3.3.1 Alexandru Eduard Toni

- **JavaFX** per la realizzazione di **WorkSpace**, **GameSetup**, **SettingsView**, **AnimationView**.

- <https://github.com/Visco0/00P-pixArt-gen/blob/54000c0b6b2abd44fa9c2e7d7a6d1eb6e96d88/src/main/java/it/unibo/pixArt/view/workspace/WorkSpace.java#L1>
- <https://github.com/Visco0/00P-pixArt-gen/blob/54000c0b6b2abd44fa9c2e7d7a6d1eb6e96d88/src/main/java/it/unibo/pixArt/view/animation/AnimationView.java#L1>
- <https://github.com/Visco0/00P-pixArt-gen/blob/54000c0b6b2abd44fa9c2e7d7a6d1eb6e96d88/src/main/java/it/unibo/pixArt/view/impl/SettingsView.java#L1>
- <https://github.com/Visco0/00P-pixArt-gen/blob/54000c0b6b2abd44fa9c2e7d7a6d1eb6e96d88/src/main/java/it/unibo/pixArt/view/impl/GameSetupView.java#L1>

- **Stream** in diverse parti del codice, ed in particolare in **ColorStack**:

- <https://github.com/Visco0/00P-pixArt-gen/blob/54000c0b6b2abd44fa9c2e7d7a6d1eb6e96d88/src/main/java/it/unibo/pixArt/model/colorstack/ColorStackImpl.java#L51>
- <https://github.com/Visco0/00P-pixArt-gen/blob/54000c0b6b2abd44fa9c2e7d7a6d1eb6e96d88/src/main/java/it/unibo/pixArt/view/impl/GameSetupView.java#L53>
- <https://github.com/Visco0/00P-pixArt-gen/blob/54000c0b6b2abd44fa9c2e7d7a6d1eb6e96d88/src/main/java/it/unibo/pixArt/model/project/FileTypes.java#L24>

- **Lambda** in diverse parti del codice, ad esempio in:

- <https://github.com/Visco0/00P-pixArt-gen/blob/54000c0b6b2abd44fa9c2e7d7a6d1eb6e96d88/src/main/java/it/unibo/pixArt/model/colorstack/ColorStackImpl.java#LL32C1-L32C1>
- <https://github.com/Visco0/00P-pixArt-gen/blob/54000c0b6b2abd44fa9c2e7d7a6d1eb6e96d88/src/main/java/it/unibo/pixArt/model/colorstack/ColorStackImpl.java#LL44C6-L44C6>

- **Optional** Utilizzato in **ColorStack**:

- <https://github.com/Visco0/00P-pixArt-gen/blob/54000c0b6b2abd44fa9c2e7d7a6d1eb6e96d88/src/main/java/it/unibo/pixArt/model/colorstack/ColorStackImpl.java#LL30C6-L30C6>

3.3.2 Micolucci Beatrice

- **JavaFX** per la realizzazione di *LoginView* e *GameView*

<https://github.com/Visco0/00P22-pixArt-gen/blob/aba8dab0c638019a2fe16f0ae118ca7c757cbef6/src/main/java/it/unibo/pixart/view/login/LoginView.java#L1C1-L182>

- **Stream** utilizzati in diverse parti come, ad esempio, nel *Bucket* e in *AbstractDrawingTool*.

Segue un solo esempio:

<https://github.com/Visco0/00P22-pixArt-gen/blob/aba8dab0c638019a2fe16f0ae118ca7c757cbef6/src/main/java/it/unibo/pixart/model/tools/filltools/Bucket.java#L69-L72>

- **Lambda** utilizzate in varie parti di codice

Segue un solo esempio:

<https://github.com/Visco0/00P22-pixArt-gen/blob/aba8dab0c638019a2fe16f0ae118ca7c757cbef6/src/main/java/it/unibo/pixart/model/user/storage/UserDataStorageImpl.java#L74-L77>

- **Gson** libreria per serializzare e deserializzare oggetti Java in JSON, utilizzata per il salvataggio dei dati degli utenti

Segue un solo esempio:

<https://github.com/Visco0/00P22-pixArt-gen/blob/dfdb6a9241f965ad43a2a4de3c30449d66468a90/src/main/java/it/unibo/pixart/model/user/storage/UserDataStorageImpl.java#L52-L62>

- **Optional** utilizzati soprattutto nell'*UserManager* per gestire i casi in cui venissero restituiti valori nulli

Segue un solo esempio:

<https://github.com/Visco0/00P22-pixArt-gen/blob/dfdb6a9241f965ad43a2a4de3c30449d66468a90/src/main/java/it/unibo/pixart/model/user/manager/UserManagerImpl.java#L27-L37>

- **Runnable** per far partire il timer

<https://github.com/Visco0/00P22-pixArt-gen/blob/dfdb6a9241f965ad43a2a4de3c30449d66468a90/src/main/java/it/unibo/pixart/model/timer/TimerThread.java#L19-L45>

3.3.3 Ricci Alessandro

- **JavaFX**: per la realizzazione di **ProjectView**.

Permalink: <https://github.com/Visco0/00P-pixArt-gen/blob/54000c0b6b2abd44fa9c2e7d7a6d1eb6/src/main/java/it/unibo/pixArt/view/impl/ProjectView.java>

- **Stream e Lambda:** utilizzati in molte delle classi create, in particolare in **ImagePrinter**
Permalink: <https://github.com/Visco0/00P22-pixArt-gen/blob/a883893c282fb7dee9a3844e50fe1f/src/main/java/it/unibo/pixart/utilities/ImagePrinter.java#L45>
- **GSON:** libreria utilizzata per convertire i progetti in file JSON e viceversa.
Permalink: <https://github.com/Visco0/00P22-pixArt-gen/blob/a883893c282fb7dee9a3844e50fe1f/src/main/java/it/unibo/pixart/utilities/FileHandler.java#L30>
- **Graphics2D e ImageIO:** librerie utilizzate per la creazione e la gestione delle immagini.
Permalink: <https://github.com/Visco0/00P22-pixArt-gen/blob/a883893c282fb7dee9a3844e50fe1f/src/main/java/it/unibo/pixart/utilities/ImagePrinter.java#L91>

3.3.4 Visconti Aldo

- **Stream e Lambda:** generic mirror, gridpaneparser, Utilizzati in molte classi come ad esempio: <https://github.com/Visco0/00P22-pixArt-gen/blob/a883893c282fb7dee9a3844e50fe1faa4661dcdb/src/main/java/it/unibo/pixart/utilities/mirror/GenericMirror.java#L21>, <https://github.com/Visco0/00P22-pixArt-gen/blob/a883893c282fb7dee9a3844e50fe1faa4661dcdb/src/main/java/it/unibo/pixart/utilities/parser/GridPaneParser.java#L28>
- **Generics:** Utilizzati in: <https://github.com/Visco0/00P22-pixArt-gen/blob/a883893c282fb7dee9a3844e50fe1faa4661dcdb/src/main/java/it/unibo/pixart/utilities/mirror/GenericMirror.java#L14>

3.4 Crediti

- **SceneManager** classe che si occupa dello switch tra views, stages e controller vari, ispirata dal progetto Jhaturanga.
- **User** strutturazione delle classi per il salvataggio degli user ispirata al progetto Jhaturanga.
- **TimerThread** classe TimerThread ispirata all'omonima classe del progetto Jhaturanga
- **Algoritmo Flood Fill** per l'implementazione del Bucket
- **Interface Serializer** per la registrazione di nuovi TypeAdapter per le funzionalità di serializzazione e deserializzazione

4 Commenti finali

4.1 Autovalutazione e lavori futuri

4.1.1 Alexandru Eduard Toni

Lo sviluppo di questo progetto è stata una sfida molto ardua ed impegnativa. Non avendo mai lavorato su un progetto così grande, devo ammettere di essermi trovato molto spesso in difficoltà. Ma è stata proprio questa condizione di difficoltà in cui mi sono ritrovato, che mi ha spinto ad uscire dalla mia comfort zone e a provare ad imparare cose nuove. Lavorando su questo progetto, ho sviluppato un interesse per ciò che concerne la qualità del codice e i design patterns. Nonostante sia soddisfatto del mio lavoro, ammetto che ci sia ancora molto da migliorare e molto da imparare, ma ciò non mi fa scoraggiare poichè fa parte del mio percorso. Mi piacerebbe approfondire di più i design patterns in futuro, svolgendo individualmente qualche progetto più piccolo in cui ci sia la necessità di implementarli. Continuerei a programmare volentieri in Java, visto che mi piace molto la sua natura strutturata e organizzata. Tutto sommato, sono soddisfatto di questa esperienza, poichè mi ha fatto imparare tante cose. Da un punto di vista pragmatico, di "hard skills", ho imparato ad usare JavaFX, e tools come Git e gradle, che sono di vitale importanza per qualsiasi software engineer. Inoltre, ho anche sviluppato l'occhio per il codice pulito. Da un punto di vista di "soft skills", ho imparato a lavorare in gruppo, a comunicare con altre persone in un contesto lavorativo e a trovare sempre una soluzione nel caso di pareri discordanti.

4.1.2 Micolucci Beatrice

Lavorare a questo progetto è stata un'esperienza molto impegnativa ma anche stimolante.

È stato per me il primo approccio alla programmazione ad oggetti, ma soprattutto alla realizzazione di un progetto di gruppo di tali dimensioni.

Mi sono inizialmente occupata della sezione riguardante i tools; è stata, in particolare, impegnativa la ricerca di un design pattern adeguato per la creazione di quest'ultimi.

La gestione degli user e del loro salvataggio in memoria mi ha, inoltre, permesso di imparare qualcosa sull'utilizzo dei file Json in Java.

Certamente ho ancora molto da imparare e il codice da me scritto può sicuramente essere migliorato, tuttavia mi ritengo abbastanza soddisfatta del lavoro svolto e di ciò che ho appreso in questi mesi.

La comunicazione con i miei compagni è stata molto importante e ci ha permesso di lavorare in sintonia, dividendoci le diverse sezioni da realizzare in modo da poterle implementare in autonomia.

Infine ho avuto l'opportunità di acquisire familiarità con l'utilizzo dei comandi di base di Git, che spero di poter studiare in modo più approfondito in futuro.

In conclusione posso affermare che la realizzazione di questo progetto è stata per me un'occasione di crescita.

4.1.3 Ricci Alessandro

Mi ritengo molto soddisfatto del lavoro svolto con i miei compagni e di come abbiamo lavorato in gruppo. Anche incontrando qualche difficoltà iniziale siamo riusciti comunque a lavorare bene insieme, risolvendole comunicando tra di noi e confrontandoci. Nonostante, rispetto ai miei compagni, io abbia svolto una mole meno importante di lavoro per quanto riguarda il model ho cercato di compensare con lo sviluppo della parte che gestisce i file e le immagini, parte che risultava difficile da dividere in più membri del gruppo.

Lo sviluppo di questo progetto mi ha sicuramente aiutato ad entrare in un'ottica di lavoro di gruppo, cosa che ritengo molto importante, soprattutto per me dato che è stato il primo progetto in team di questa dimensione.

In un futuro penso di portare avanti questo progetto implementando alcune funzionalità a cui ho pensato in quanto penso che abbia molto potenziale se continuato ad essere sviluppato.

4.1.4 Visconti Aldo

Sono sufficientemente soddisfatto del progetto, anche se ritengo che ci sia ancora molto da raffinare all'interno dell'architettura del progetto, in modo da renderlo più performante. In principio non ero molto sicuro di sviluppare questa applicazione in quanto avrei preferito lavorare su un gioco, e pensavo sarebbe stato poco il lavoro da svolgere, ma mi sono ricreduto. Sarei felice di portare avanti questo progetto insieme ai miei compagni, spero che loro abbiano lo stesso desiderio. Penso di non aver sfruttato al meglio DVCS, mi piacerebbe approfondire le mie conoscenze a riguardo, in maniera particolare sul **DVCS WORKFLOW**. Non sono riuscito ad integrare al meglio tutte le classi del package **pixart/view/components**, fatta eccezione per la classe `PixelsPane.java`, che penso sia la meglio riuscita, ma potrebbe essere raffinata.

4.2 Difficoltà incontrate e commenti per i docenti

4.2.1 Ricci Alessandro

Inizialmente ho incontrato alcune difficoltà nella parte di codice, in quanto io lavoro su un Sistema Operativo diverso da quello dei miei compagni, difficoltà che sono riuscito a risolvere confrontandomi con il mio team.

Per quanto riguarda il corso lo ritengo un ottimo corso da frequentare, dove durante le lezioni in laboratorio viene spiegata in maniera efficace la parte più pratica, molto utile per lo sviluppo in gruppi del progetto (Git, Gradle), mentre, durante il resto del corso vengono spiegate, oltre ad alcune parti pratiche, parti più teoriche, utili per la progettazione e l'analisi dell'elaborato da sviluppare.

Appendice A

Guida utente

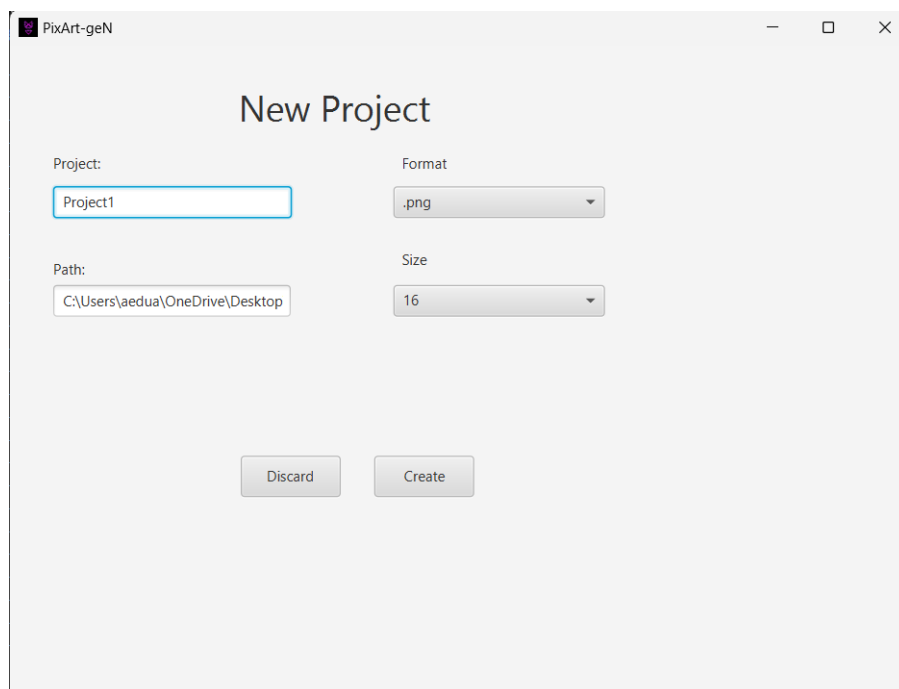
A.1 Login

The screenshot shows a web application window titled "PixArt-geN". The interface is divided into two main sections by a vertical line. On the left, under the heading "Login", there are input fields for "Username" and "Password", a "Login" button, and a "Login as guest" button. On the right, under the heading "Create Account", there are input fields for "Username", "Password", and "Path", along with "Register" and "Browse" buttons. The PixArt-geN logo is visible in the top left corner of the application area.

Figure 10: Schermata Login

Una volta avviata l'applicazione verrà mostrata all'utente una schermata tramite cui egli potrà eseguire il login, registrarsi, oppure eseguire l'accesso come *guest* specificando dapprima il path in cui vorrà salvare eventuali progetti.

A3. Settings



The screenshot shows a 'New Project' dialog box from the application 'PixArt-geN'. The dialog is titled 'New Project' and contains the following fields and controls:

- Project:** A text input field containing 'Project1'.
- Format:** A dropdown menu showing '.png'.
- Path:** A text input field containing 'C:\Users\aedua\OneDrive\Desktop'.
- Size:** A dropdown menu showing '16'.
- Buttons:** Two buttons at the bottom: 'Discard' and 'Create'.

Figure 11: Schermata Settings.

In questa sezione, l'utente seleziona gli attributi che il proprio progetto avrà. L'utente può scegliere il nome del progetto, che di default è "Project1", il formato delle immagini, che di default è ".png", e la dimensione della matrice, che può essere 16x16, 32x32, 64x64 (default 16x16). Nel campo sotto il label "Path", viene indicato il path in cui verrà salvata la cartella relativa al progetto. Tale path viene scelto dall'utente inn fase di registrazione, e non può essere cambiato. Cliccando il pulsante "Create", viene generato il progetto e la relativa cartella, e si passa alla WorkSpace view. Nel caso ci fosse già un progetto con lo stesso nome nel path dell'utente, un pop-up di warning appare sullo schermo, informando l'user del problema. Il progetto vecchio può essere sovrascritto. Cliccando su "Discard", si ritorna alla sezione Home.

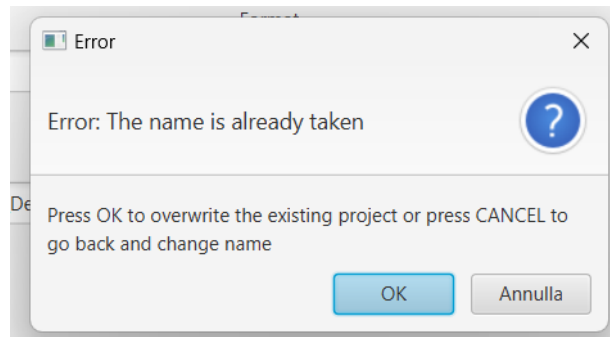


Figure 12: Warning pop-up che informa l'utente della presenza di un progetto con lo stesso nome nel path.

A4. Workspace

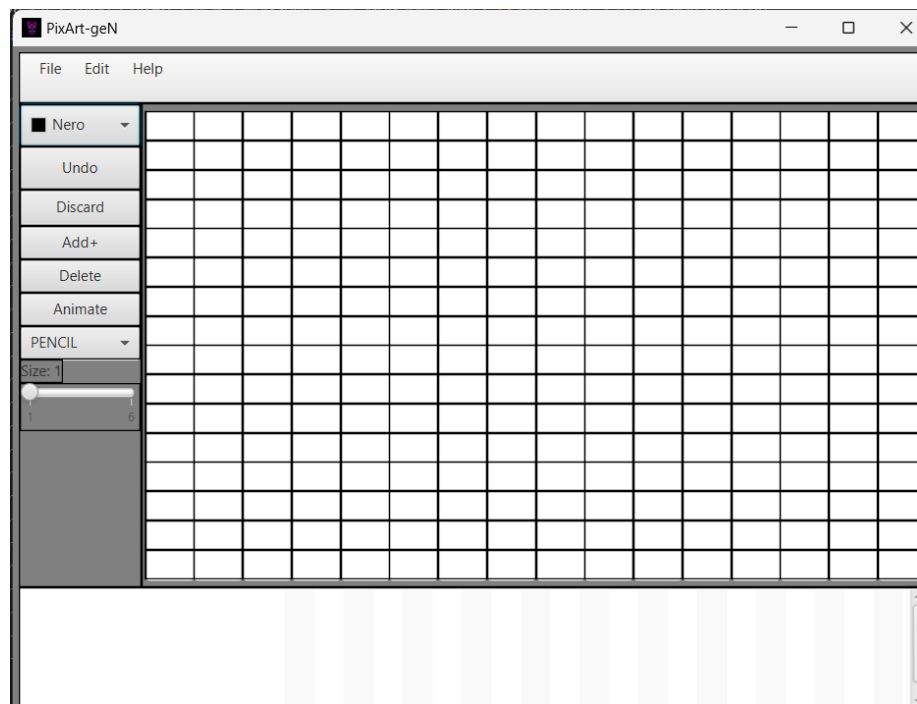


Figure 13: Schermata di Workspace

Workspace è la view principale, in cui l'utente crea, edita, elimina i vari frame di un progetto. Al centro è presente una matrice, che si colora muovendo

il cursore sulle vari celle. Cliccando una sola volta, si riesce a colorare una sola cella. Cliccando due volte di seguito, si ottiene una colorazione continua, in cui ogni volta che il cursore passa sopra una cella, essa viene colorata del colore selezionato. La colorazione continua si ferma cliccando di nuovo 2 volte di seguito sulla matrice.

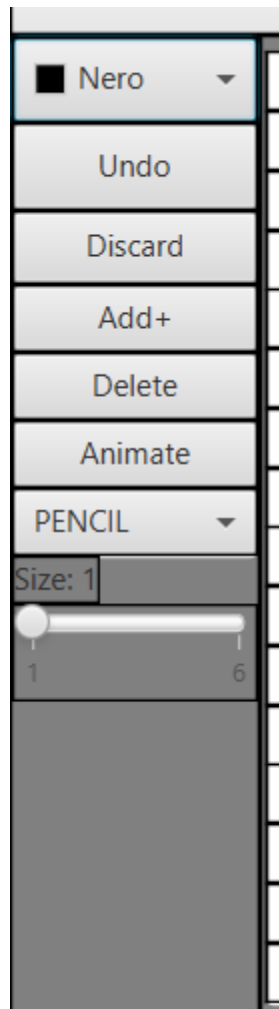


Figure 14: Barra dei pulsanti laterali del Workspace.

A sinistra della schermata, è presente una barra dei pulsanti, ognuno dei quali svolge una determinata funzione.

- Lo slider in fondo serve ad indicare la dimensione del tool, che va da 1x1 a 6x6(default 1x1).

- Il dropdown serve a selezionare il tool, che di default è il "PENCIL". Ci sono 6 tool tra cui scegliere.
- Cliccando sul pulsante "Animate", si passa alla via di animazione, in cui vengono animati i vari frame del progetto.
- Il pulsante "Delete" serve ad eliminare il frame attuale su cui si sta lavorando.
- Il pulsante "Add+" serve ad aggiungere un nuovo frame in coda alla lista di frame del progetto.
- Il pulsante "Discard" serve a ripulire l'intero frame attuale, rendendo tutte le celle bianche.
- Il pulsante "Undo" serve a riportare la matrice agli stati precedenti, "disfando" le ultime modifiche.
- L'elemento più in alto è un color picker, che serve a selezionare il colore da utilizzare(di default è nero).

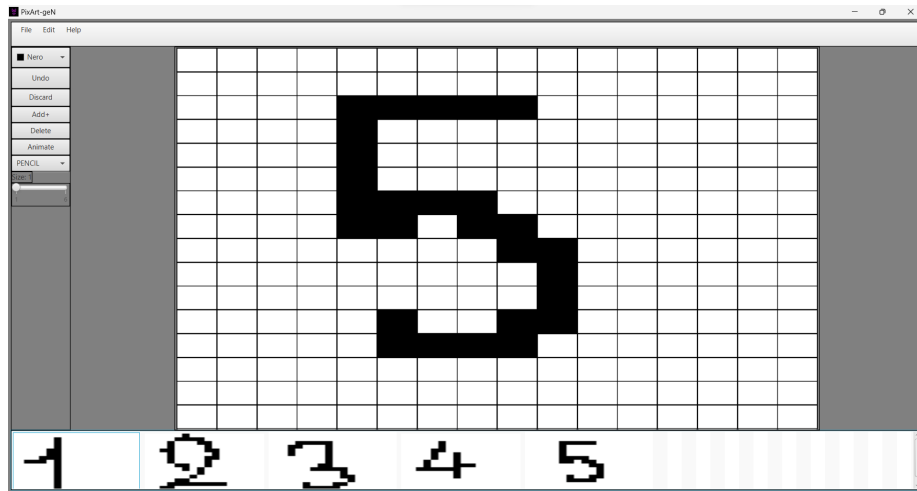


Figure 15: Esempio di una serie di frame.

Una volta creati e colorati diversi frame, l'utente può selezionarli nella barra di HistoryFrame in basso, cliccando sul frame in interesse. La barra di HistoryFrame serve a navigare tra la lista di frame, ma anche di offrire un'anteprima delle immagini del progetto.

Nella barra più in alto, l'utente può salvare il progetto cliccando File->Save. Cliccando File->Close, si ritorna alla schermata di Home, senza salvare le

modifiche.

Nel momento in cui l'utente decide di salvare un progetto, e clicca "Save", appare un popup tramite il quale si seleziona la scala delle immagini, in rapporto alla dimensione della matrice; infatti esse possono essere ingrandite di un fattore pari a x1, x4 o x16 volte la dimensione della matrice.

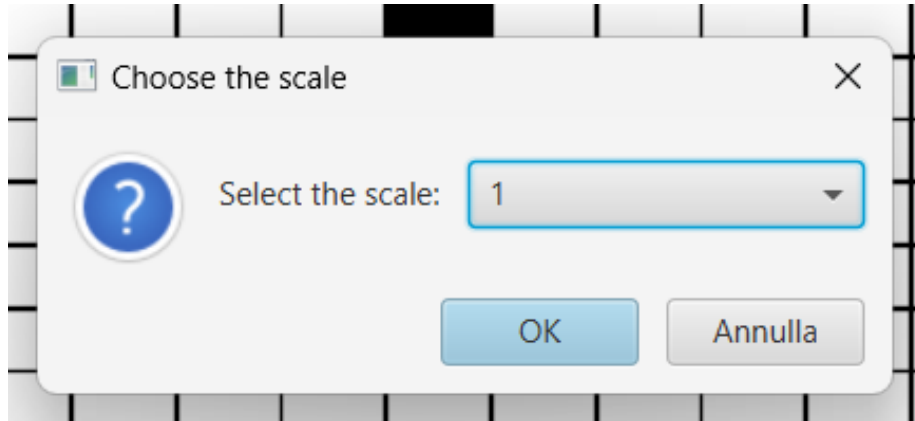


Figure 16: Scale popup.

A5. ProjectView

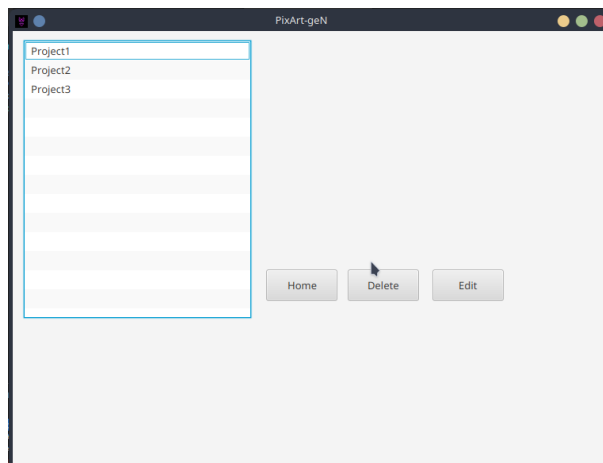
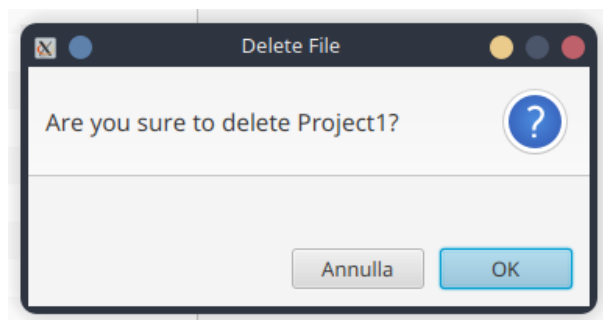


Figure 17: Visualizzazione della ProjectView

Nella schermata "ProjectView" sarà possibile visualizzare i progetti creati dall'utente presenti nel suo path di salvataggio, una volta selezionato uno di questi è possibile caricarlo per modificarlo attraverso il pulsante "Edit" oppure eliminarlo premendo il pulsante Delete.



Nell'ultimo caso si aprirà un pop-up di conferma, in caso si preme il bottone "Ok" il progetto selezionato verrà eliminati altrimenti si tornerà alla ProjectView.

A6. AnimationView

Nella schermata relativa all'AnimationView, l'utente può animare le immagini ottenute dai vari frame del progetto. Cliccando sul pulsante con l'icona play ,l'animazione inizia, alternando le varie immagini tra loro, mostrandole per un certo intervallo di tempo, o delay. L'immagine attuale viene mostrata nel

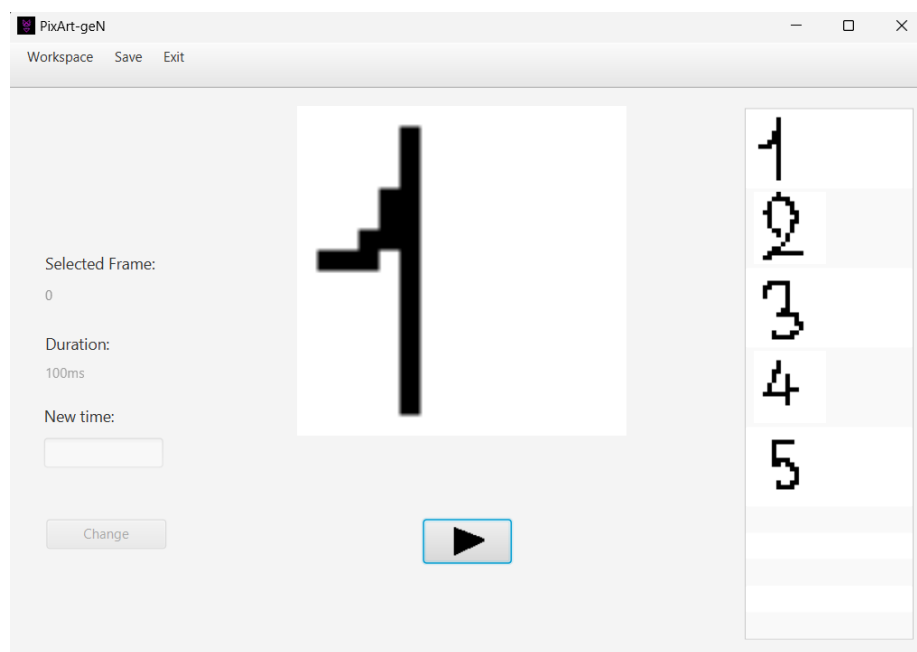


Figure 18: Schermata dell'AnimationView.

riquadro centrale. L'animazione viene fermata ricliccando sullo stesso pulsante, che adesso mostra l'icona dello stop. Il delay è di default di 100ms, quando viene creato un HistoryFrame.

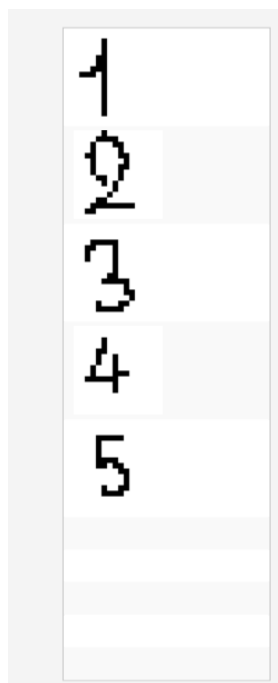


Figure 19: Lista di frame.

A destra della schermata, è presente la lista dei frame. Cliccando su uno di loro, esso viene mostrato nel riquadro centrale.

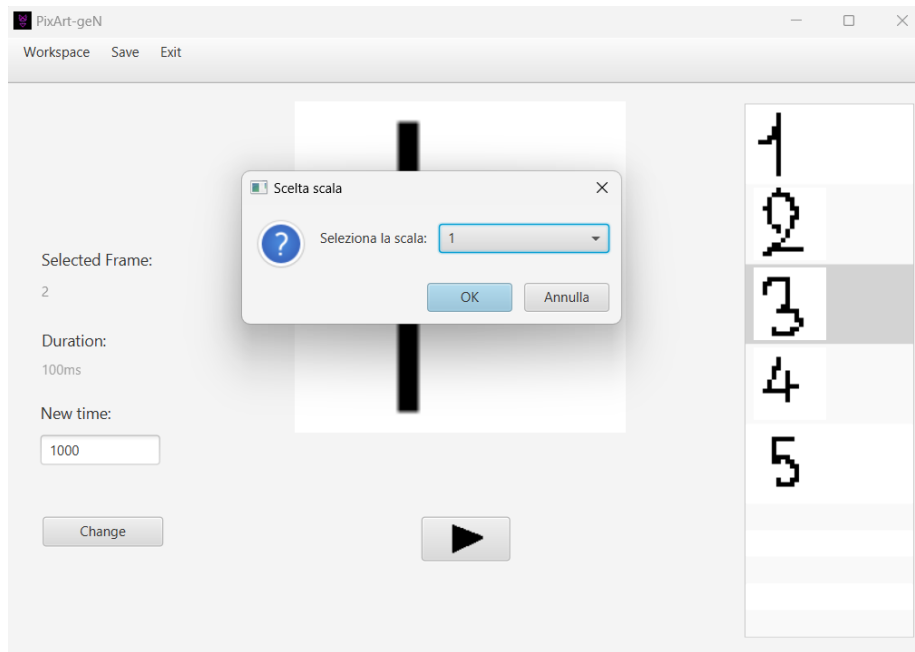


Figure 20: Campi per modificare il delay del frame attuale.

A sinistra della schermata, ci sono dei label che indicano l'indice del frame attuale e il suo delay in ms. Inserendo un nuovo delay nel campo specifico, e cliccando "Change", il delay della relativa immagine viene sovrascritto.

Nella parte sopra della schermata, è presente una barra con 3 diversi pulsanti:

- Cliccando su "WorkSpace", si ritorna alla schermata di WorkSpace.
- Cliccando su "Save", il progetto viene salvato e si ritorna alla Home.
- Cliccando su "Exit" si ritorna alla Home, senza salvare le modifiche apportate al progetto.

Quando si sceglie di salvare il progetto, appare lo stesso popup del WorkSpace, che fa scegliere all'utente la scala delle immagini.

A7. GameSetup

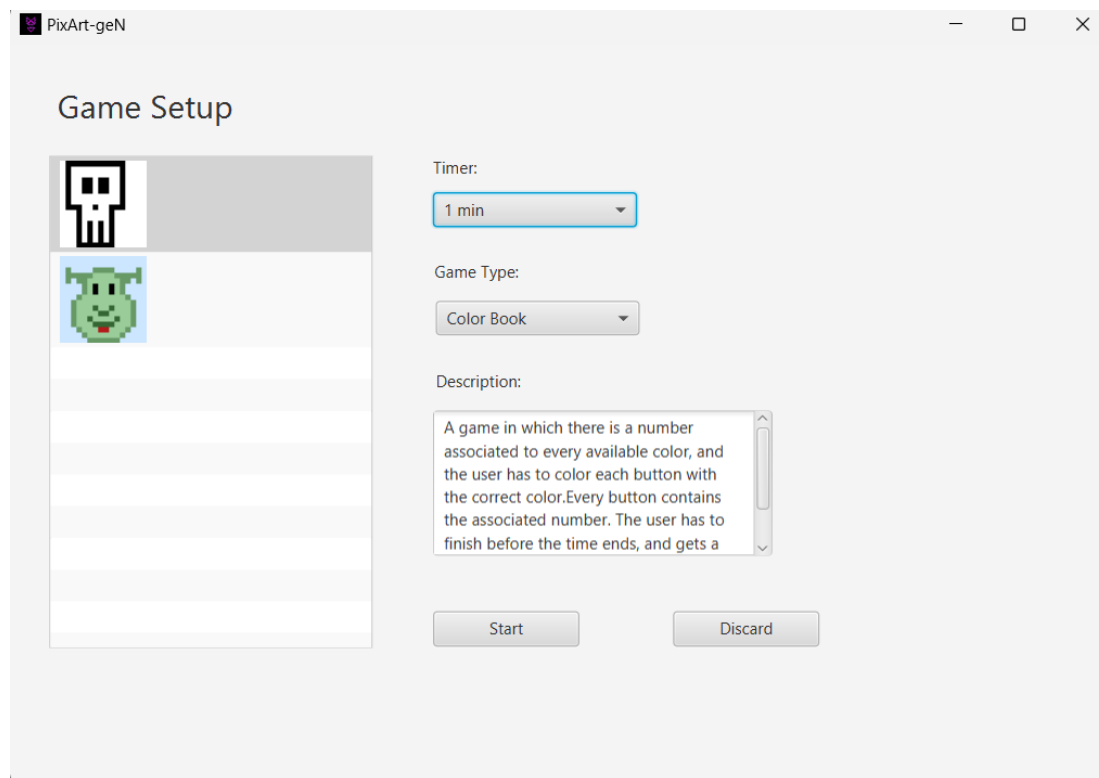


Figure 21: Schermata di GameSetup

In questa schermata, l'utente decide in quale modalità di gioco cimentarsi. A sinistra c'è una lista di progetti già pronti messi a disposizione, che possono essere selezionati per giocare. A destra ci sono 2 dropdown: la prima serve a scegliere il timer del gioco. Ci sono 3 tipi di timer (1 minuto, 3 minuti, 5 minuti). Sotto al dropdown del timer c'è il secondo dropdown, che serve a scegliere la modalità di gioco. Quando viene selezionata una modalità di gioco, in basso viene esposta una breve descrizione che spiega il funzionamento della relativa modalità di gioco. Di default, il timer è da 1 minuto, e la modalità di gioco è la "Color Book". Cliccando su "Start" si passa alla GameView. Cliccando su "Discard", si ritorna alla Home.

A.8 Game: modalità *Color Book*

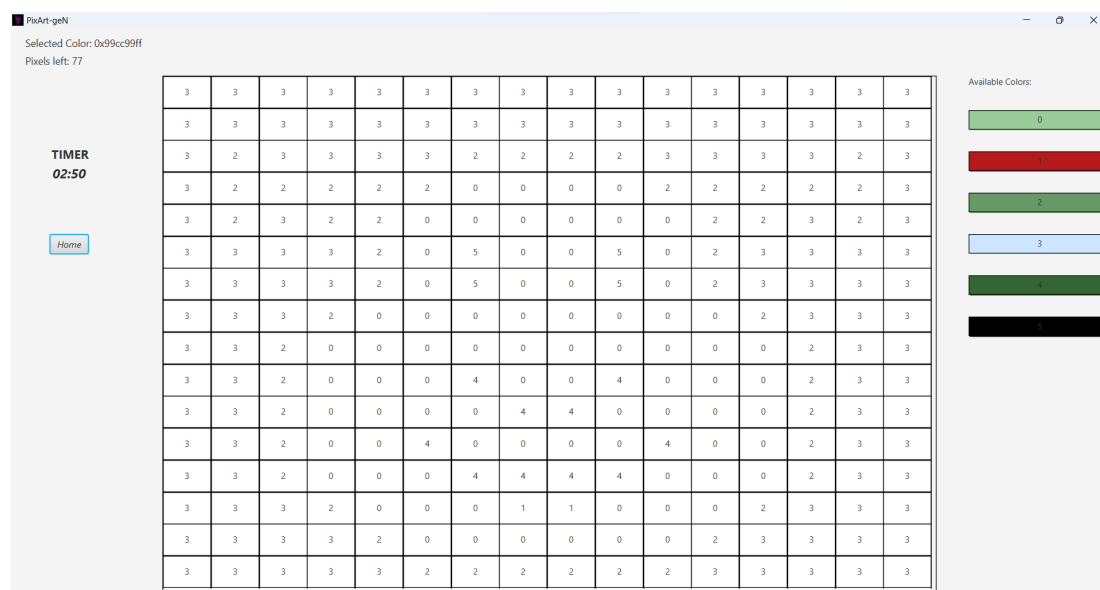


Figure 22: Schermata game in modalità Color Book

Una volta scelta la modalità di gioco *Color Book* apparirà una schermata con una griglia di pixel, ad ognuno dei quali è associato un numero. Ogni numero corrisponde a un colore. I colori con i relativi numeri sono mostrati sulla destra. Lo scopo del gioco è quello di riprodurre l'immagine scelta nel *GameSetup*. L'utente sceglie quindi quello desiderato e clicca su un pixel. Se il colore è giusto il pixel si colorerà, altrimenti rimarrà bianco.

In alto a sinistra viene mostrato il numero di pixel che devono ancora essere colorati, più in basso vi è un timer che mostra il tempo rimanente.

Una volta scaduto il tempo o quando tutti i pixel sono stati colorati apparirà un pop-up che esibirà il numero di pixel colorati in modo corretto. Qui l'utente potrà scegliere se tornare al menù principale o se iniziare un nuovo gioco.

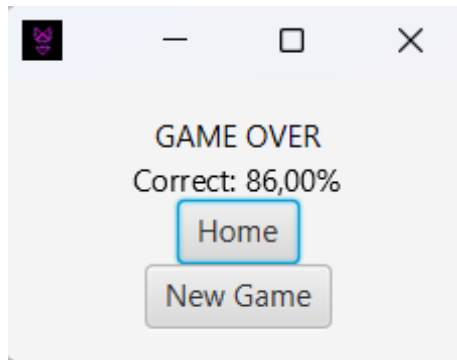


Figure 23: GameOver pop-up.

A.9 Game: modalità Mirror

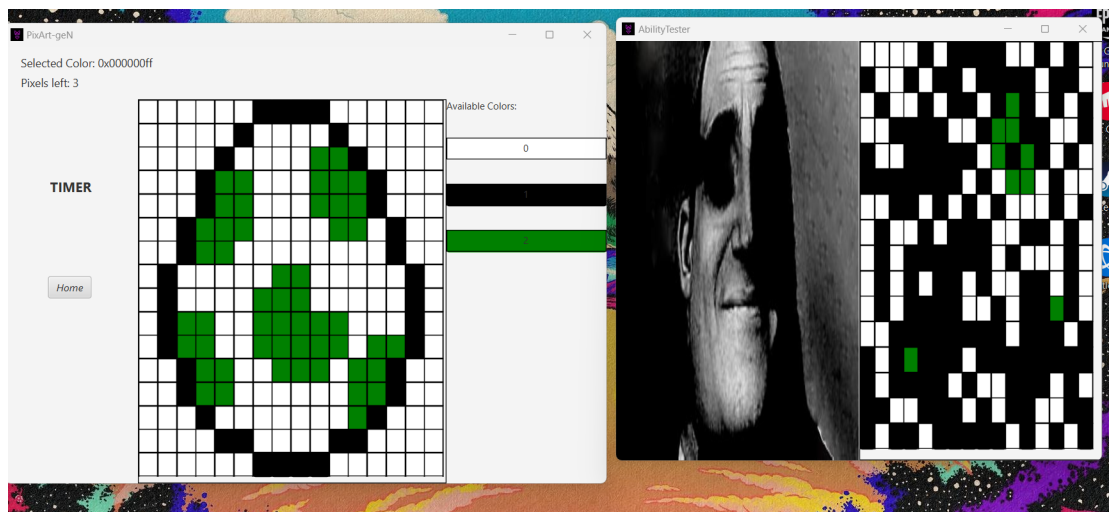


Figure 24: Schermata game in modalità Mirror

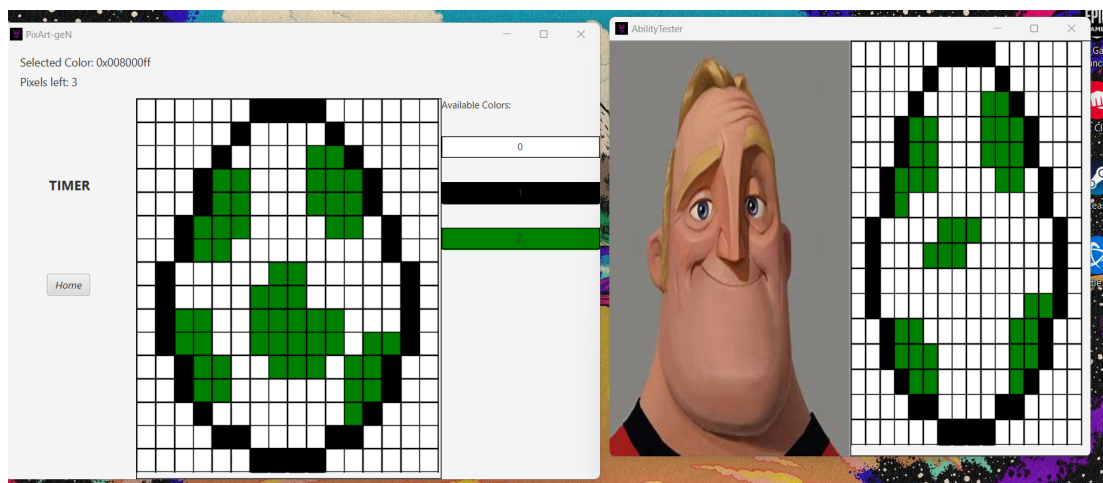


Figure 25: Schermata game in modalità Mirror

Dopo aver scelto la modalità **Mirror** verranno avviati due stage, nel primo si avrà la possibilità di rivelare l'immagine (esattamente come nel gioco **ColorBook**) e scegliere i colori da utilizzare, nel secondo si avrà un'altra matrice su cui disegnare come qualsiasi cosa si desideri, **MR.INCREDIBILE** valuterà in tempo reale il vostro disegno, cambiando espressione in base al livello di correttezza del disegno.