

# Relazione per “BarlugoFX”

Gobbi Giovanni, Fiordarancio Matteo, Lunadei Jacopo, Baroncini Ugo

25 aprile 2019

## **Sommario**

Il progetto BarlugoFX è un editor fotografico che fornisce vari strumenti di modifica per elaborare un'immagine. Il progetto nasce dalla necessità di avere a disposizione un editor facile da usare e disponibile su tutte le piattaforme.

Per ulteriori informazioni, si consulti il nostro repository Bitbucket:  
<https://bitbucket.org/Gobbees/oop18-barlugofx/>

# Indice

<b>1 Analisi</b>	<b>3</b>
1.1 Requisiti . . . . .	3
1.2 Analisi e modello del dominio . . . . .	4
<b>2 Design</b>	<b>6</b>
2.1 Architettura . . . . .	6
2.2 Design dettagliato . . . . .	7
2.2.1 Matteo Fiordarancio . . . . .	7
2.2.2 Giovanni Gobbi . . . . .	9
2.2.3 Jacopo Lunadei . . . . .	12
2.2.4 Ugo Baroncini . . . . .	12
<b>3 Sviluppo</b>	<b>16</b>
3.1 Testing automatizzato . . . . .	16
3.2 Metodologia di lavoro . . . . .	16
3.2.1 Matteo Fiordarancio . . . . .	17
3.2.2 Giovanni Gobbi . . . . .	18
3.2.3 Jacopo Lunadei . . . . .	20
3.2.4 Ugo Baroncini . . . . .	20
3.3 Note di sviluppo . . . . .	21
3.3.1 Matteo Fiordarancio . . . . .	21
3.3.2 Giovanni Gobbi . . . . .	22
3.3.3 Jacopo Lunadei . . . . .	23
3.3.4 Ugo Baroncini . . . . .	23
<b>4 Commenti finali</b>	<b>24</b>
4.1 Autovalutazione e lavori futuri . . . . .	24
4.1.1 Matteo Fiordarancio . . . . .	24
4.1.2 Giovanni Gobbi . . . . .	25
4.1.3 Jacopo Lunadei . . . . .	25
4.1.4 Ugo Baroncini . . . . .	26

<b>A Guida utente</b>	<b>27</b>
A.0.1 Crop . . . . .	28
A.0.2 Rotate . . . . .	29

# Capitolo 1

## Analisi

### 1.1 Requisiti

Il software si pone come obiettivo quello di fornire agli utenti di qualunque sistema operativo un editor di immagine di facile utilizzo, funzionale e soprattutto open source. Con editor di immagini si intende un software che permette di effettuare manipolazioni su un'immagine, quali, ad esempio, cambiarne la luminosità o ritagliarla.

#### Requisiti funzionali

- Il suddetto software dovrà disporre di un'interfaccia grafica user-friendly che permetta di applicare diverse manipolazioni su un'immagine scelta dall'utente.
- Il software dovrà inoltre dotarsi di strumenti che permettano di tenere traccia degli strumenti applicati, per permettere all'utente di annullare i cambiamenti fatti o sovrascriverli.
- Il software dovrà inoltre implementare i seguenti strumenti di manipolazione dell'immagine:
  1. Esposizione
  2. Luminosità
  3. Contrasto
  4. Trasformazione in bianco e nero
  5. Modifica selettiva dei colori RGB
  6. Bilanciamento del bianco

- 7. Saturazione
- 8. Tonalità
- 9. Vivacità colori
- Il software dovrà inoltre implementare i seguenti strumenti di modifica dell'immagine:
  1. Ritaglio
  2. Rotazione di angolo arbitrario
  3. Esportazione dell'immagine modificata in diversi formati (Jpeg, PNG, GIF), alcuni dei quali potranno anche essere compressi a piacere.
- Il software dovrà inoltre essere in grado, previa specifica creazione da parte dell'utente, di applicare una specifica sequenza di strumenti di manipolazione dell'immagine tutti in una volta (preset)

## 1.2 Analisi e modello del dominio

BarlugoFX dovrà essere in grado di visualizzare a schermo una qualunque immagine di un formato adeguato (png, jpeg e gif). Su tale immagine deve essere possibile applicare degli strumenti di manipolazione e modifica. Ogni strumento dipende da uno o più parametri numerici. Ove possibile, l'utente potrà specificare l'intensità del strumento. BarlugoFX dovrà inoltre permettere di sovrascrivere strumenti già applicati e/o rimuoverli. Sarà inoltre possibile salvare una sequenza di strumenti in un unico file (preset), che potrà poi essere applicato a piacere su qualunque altra immagine. L'immagine potrà essere inoltre ingrandita o rimpicciolita a piacere per evidenziare meglio le modifiche fatte in determinati punti. Dovrà inoltre essere possibile ruotare l'immagine.

Le difficoltà più importanti che ci si troverà da affrontare saranno sicuramente:

- Costruzione di una GUI che sia facile da utilizzare per l'utente e comunicativa anche per chi non ha esperienza nell'editing fotografico

- Implementazione dei possibili strumenti tenendo conto che essi possano essere eseguiti in parallelo o meno
- Gestione corretta della history delle modifiche per permettere all'utente di modificare strumenti già applicati in modo da ottenere un risultato esteticamente ottimale.
- Gestire correttamente la scrittura e lettura da file durante il salvataggio e caricamento dei preset

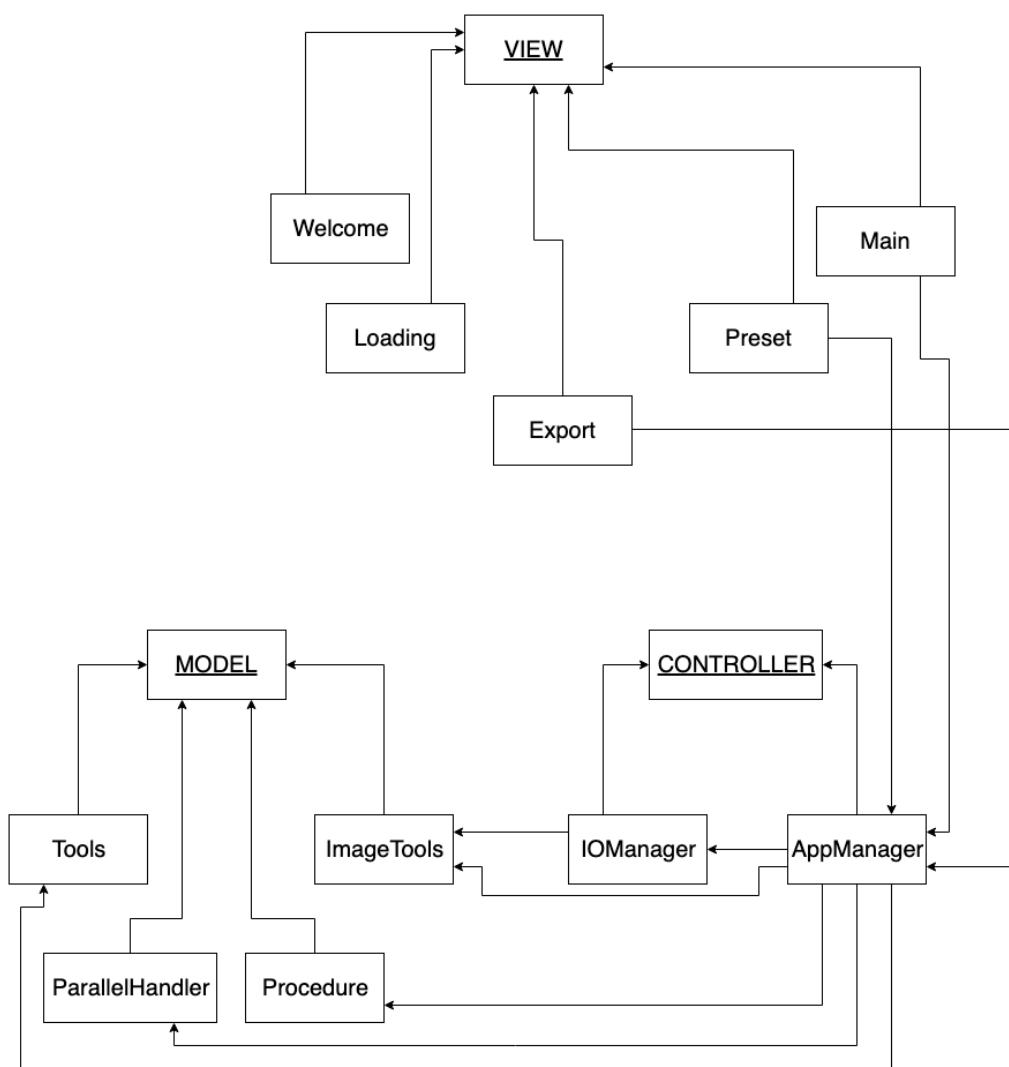


Figura 1.1: Schema UML dell'analisi del problema, con rappresentate le entità principali ed i rapporti fra loro

# Capitolo 2

## Design

### 2.1 Architettura

Per la realizzazione del sistema software è stato utilizzato il pattern architettonico *model-view-controller* dividendo le tre parti in tre diversi package:

-**barlugofx.model** contiene tutti i componenti relativi al model

-**barlugofx.view** contiene tutti i componenti relativi alla view

-**barlugofx.controller** contiene tutti i componenti relativi al controller

Il model si occupa di lavorare direttamente sulle matrici di pixel e di compiere le varie operazioni richieste dal controller. Il model puo' essere diviso a sua volta in due parti: una gestisce la history dei tool, cioè il modo in cui i vari tool sono stati applicati sulla immagine, la seconda parte gestisce invece propriamente i strumenti e le immagini.

Il model non utilizza nessun componente relativo nè al controller nè alla view, perciò è perfettamente scorporabile dal resto del sistema.

Il controller si occupa di "collegare" le richieste della view ad azioni del model. Si compone di due componenti principali: **AppManager** e **IOManager** (le cui funzionalità verranno spiegate nella parte 3.2.1).

**AppManager** si interfaccia al model tramite i riferimenti ai vari componenti **ImageTool** e all'utilizzo dello strumento di parallelizzazione **ParallelFilterExecutor** mentre **IOManager** utilizza il tipo di dato **Image**.

Entrambi utilizzano inoltre uno strumento di conversione presente nella classe Utilities **ImageUtils**.

Nessuno dei due componenti ha riferimenti a componenti della view.

La view contiene tutti i componenti relativi alla visualizzazione del siste-

ma all’utente. Si compone di diverse interfacce grafiche ciascuna con una specifica funzione. Ogni view (fatta eccezione per la `Welcome` e la `Loading`) mantiene un riferimento ad un oggetto `AppManager` e utilizza alcune delle sue funzionalità.

Si specifica che in nessuna view compare un elemento relativo alla parte di model o alla parte di controller `IOManager`.

Si è prestata particolare attenzione a rendere i tre componenti ben distinti perciò:

- un eventuale cambiamento della view non impatterebbe in nessun modo né su model né su controller.
- un eventuale cambiamento del controller comporterebbe un adattamento della sola view mentre il model rimarrebbe uguale.
- un eventuale cambiamento del controller impatterebbe solamente il controller, non la view.

## 2.2 Design dettagliato

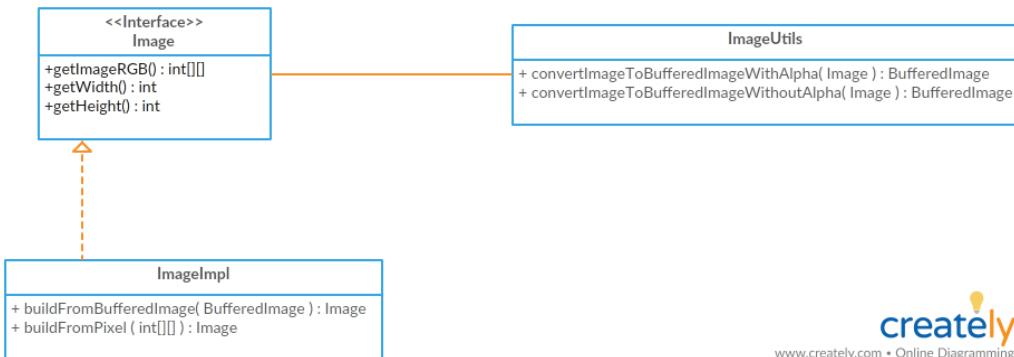
### 2.2.1 Matteo Fiordarancio

In questa sezione ci occuperemo di discutere tutte le scelte fatte all’interno del model ad esclusione della History.

Per prima cosa dovevamo scegliere quale implementazione, fra le varie possibili, usare per rappresentare un’immagine. Visto che volevamo una struttura che fosse il più possibile indipendente dalle varie implementazioni, abbiamo deciso di costruire un’interfaccia `Image` che fornisse i metodi a noi necessari per applicare i vari tool. Ci siamo avvalsi di una classe di utility, `ImageUtils`, che permette di gestire le conversioni fra `Image` e la state-of-art gestione delle immagini al momento, cioè una `bufferedImage`. In questo modo, se in un futuro l’implementazione passerà da una `bufferedImage` ad un’altra più efficiente, ci basterà cambiare solo alcuni metodi di `ImageUtils` e saremo a posto. Ultimo ma non meno importante, si è deciso di usare il design pattern della static factory per `ImageImpl`, in quanto considerato più corretto di un costruttore se la classe è finale (Effective Java)

In aggiunta a ciò, per questioni di efficienza abbiamo deciso di crearcì una

utility class, **ColorUtils**, che ci permette di operare sui pixel senza dover istanziare ogni volta istanze della classe `java.awt.Color`, cosa che su una matrice molto grande sarebbe stata molto pesante.



**creately**  
www.creately.com • Online Diagramming

Figura 2.1: Schema UML architetturale di Image, Image Utils e ImageImpl.

Andiamo ora ad analizzare la modellazione dei vari tool. Ogni tool implementa l’interfaccia `ImageTool` e viene istanziato utilizzando una static factory. Per evitare di scrivere sempre gli stessi metodi uguali nei vari tool, si è deciso di creare una classe d’appoggio astratta `AbstractImageTool`, la quale si occupa di implementare i metodi comuni ai vari strumenti, lasciando a ciascuno di essi l’implementazione di `applyTool()`. In questo modo si è eliminata la ridondanza dovuta alla ripetizione dei vari metodi.

Questa scelta ha inoltre portato all’uso del design pattern Template Method. Infatti, `AbstractImageTool`, per implementare `addParameter()` ha bisogno di sapere se il parametro è accettabile. Pertanto, `addParameter()` si configura come template method.

Si è presentato inoltre il problema della gestione di vari tipi di parametri, che potevano essere interi, double o float. Per poter risolvere tale problema (su suggerimento di Danilo Pianini) si è deciso che ogni parametro fosse inglobato all’interno di una classe finale `Parameter` e la gestione del corretto inserimento del parametro è stata delegata ad `AbstractImageTool`.

Infine, si è dovuta gestire la possibilità’ che alcuni strumenti potessero essere parallelizzati sulla CPU. Per fare ciò’, si è deciso di estendere l’interfaccia `ImageTool` con `ParallelizableImageTool`. Tale interfaccia usa anch’essa il template pattern: infatti, nel caso in cui un tool parallelizzabile debba essere eseguito in maniera non parallela, `ParallelizableImageTool` offre implementato di default il metodo `applyTool()`. Tale metodo è template method in quanto fa uso di `executeTool()`, implementato da tutte le classi che imple-

mentano `ParallelizableImageTool`.

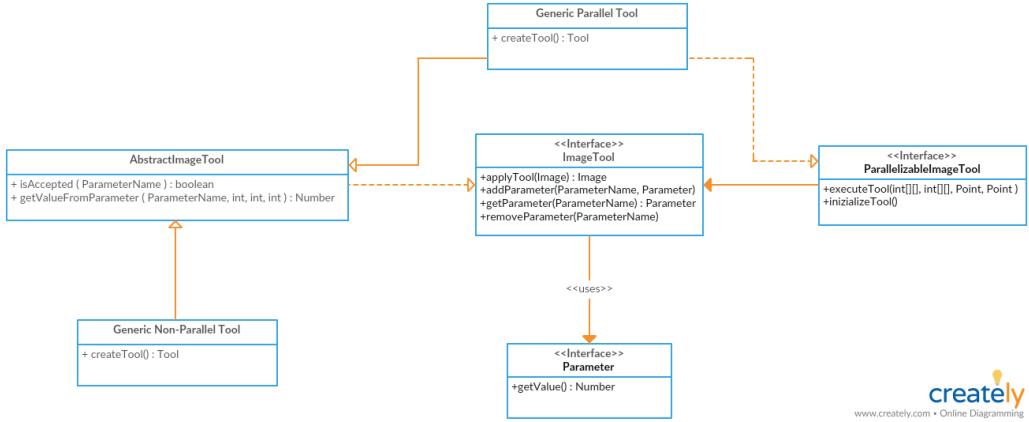


Figura 2.2: Schema UML architetturale di `ImageTool` e derivati

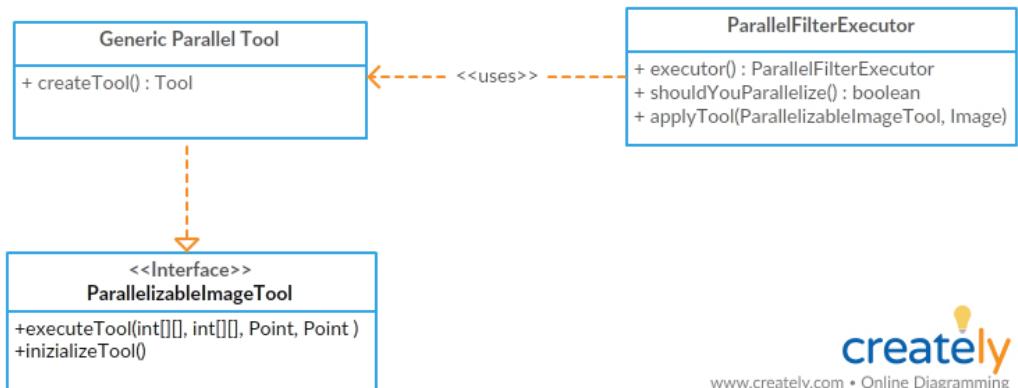
Ultimo ma non per importanza, si è dovuto decidere come gestire l'esecutore che avrebbe dovuto eseguire i strumenti in parallelo. A seguito di diversi ragionamenti, si è deciso di optare per un **Singleton**. Questo perché si vuole tassitivamente evitare che esistano più esecutori all'interno del programma. Infatti, ogni esecutore instanziato aggiungerebbe overhead di memoria e di tempo. Forzando l'istanza dell'esecutore a uno singolo e non chiudendo mai i thread istanziati ma lasciandoli solo in attesa, si è arginato il problema del possibile overhead della parallelizzazione. Inoltre, si è deciso di aggiungere il metodo `shouldYouParallelize()` che dovrebbe verificare (anche se da noi non è stato fatto) se fare la parallelizzazione sia effettivamente sensata. Infatti, su immagini piccole l'overhead dovuto alla parallelizzazione potrebbe costare di più della semplice applicazione del filtro in single-thread.

Si noti, infine, che all'interno del `ParallelFilterExecutor` è presente il pattern **Strategy**. Infatti, nella chiamata di `applyTool()`, la strategia attraverso la quale applicare un determinato strumento non è determinata dal `ParallelFilterExecutor` ma viene scelta dall'interfaccia `ParallelizableImageTool`, passata come parametro.

### 2.2.2 Giovanni Gobbi

In questa sezione verranno discusse numerose scelte relative all'architettura della parte di view e di controller.

Dato che i nomi dei componenti sono tutti molto simili e può essere facile confonderli, d'ora in avanti si utilizzerà la seguente terminologia:



**creately**  
www.creately.com • Online Diagramming

Figura 2.3: Schema UML architetturale di ParallelFilterExecutor

- Parte di View* per riferirsi alla parte di view del pattern MVC
- Parte di Controller* per riferirsi alla parte di controller del pattern MVC
- UI* per riferirsi ad una singola interfaccia grafica
- *View* per riferirsi alla classe che inizializza tutti i componenti di una UI
- *ViewController* per riferirsi al controller degli eventi di una UI

### Parte di View

Per la costruzione di ogni UI si utilizza la tecnologia JavaFX presente nativamente nei JDK Oracle fino alla versione 8 e installabile separatamente ad altre librerie (testato sia con Oracle JDK che con OpenJDK e OpenJFX).

La struttura di ogni singola UI si divide in tre componenti principali:

-**FXML\*.fxml**: file presente nella folder **res/fxml** contenente il codice fxml che genera l'aspetto grafico dell'UI. Per svilupparlo si è utilizzato il tool SceneBuilder.

-**\*View**: classe che inizializza tutti i componenti relativi all'interfaccia grafica (stage, scene, ecc.).

-**\*Controller**: classe che gestisce tutti gli eventi dell'UI (click, resize dei componenti, ecc.). I metodi/componenti contrassegnati dal tag @FXML si riferiscono ad oggetti presenti nel codice fxml.

I file fxml vengono caricati tramite un oggetto **FXMLLoader** e quindi applicati alla **Scene** all'interno della classe View.

Ciascuna delle classi View estende la classe **View** (non istanziabile) e ciò ha permesso di riutilizzare numerosi parti di codice e rendere quindi le singole classi molto compatte.

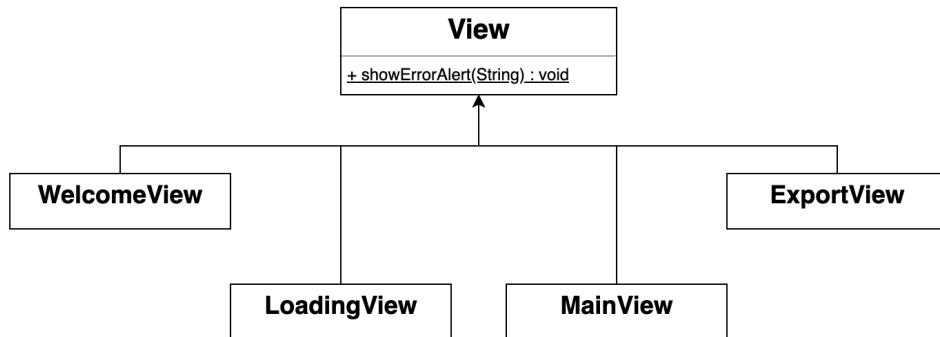


Figura 2.4: Schema UML architetturale dell'ereditarietà delle View

Per quanto riguarda le classi ViewController, ciascuna di esse implementa l'interfaccia `ViewController` tramite l'estensione di due classi astratte: `AbstractViewController` e `AbstractViewControllerWithManager`. Questa divisione è nata dalla necessità di avere due tipi di ViewController diversi: uno che comprende il manager e uno no. In questo modo, oltre a minimizzare le ripetizioni di codice, le classi ViewController hanno solamente metodi strettamente necessari agli eventi mentre i setter vengono ereditati dalle classi astratte.

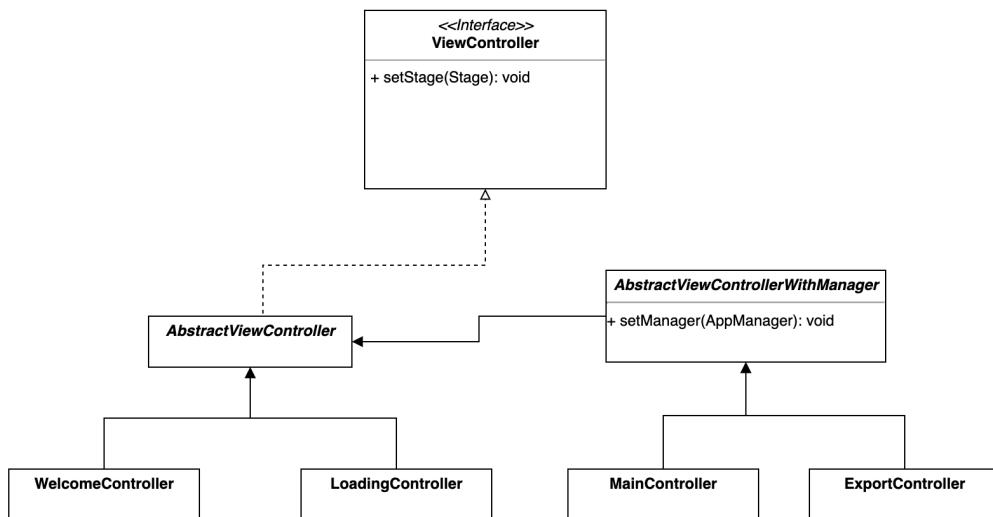


Figura 2.5: Schema UML architetturale dell'ereditarietà dei ViewController

**Parte di Controller** Dato che la parte di input/output può risultare molto laboriosa, si è deciso di dividere la parte di Controller in due componenti principali: **IOManager** e **AppManager**. Il primo si occupa di performare tutte le azioni che coinvolgono l'input/output, fra cui le più importanti: lettura e scrittura dell'immagine in vari formati.

Il secondo si occupa di fornire alla UI l'insieme di metodi che permettono di gestire tutti gli eventi eseguiti dall'utente.

Ciascuno dei due componenti è realizzato tramite l'utilizzo di una classe "Impl" che implementa la sua interfaccia.

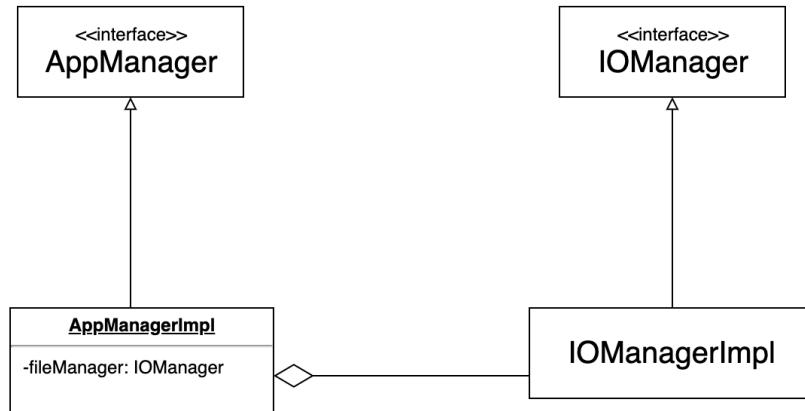


Figura 2.6: Schema UML architetturale dei vari componenti del Controller

### 2.2.3 Jacopo Lunadei

In questa sezione si discutono le scelte relative alla gestione dei preset. Per quanto riguarda le parti di architettura di **View** e **Controller** sono state prese le stesse accortezze delle altre classi simili((**MainView**, **MainController**), (**ExportView**, **ExportController**),...). Per quanto riguarda l'effettivo salvataggio di un preset si è scelto di utilizzare la classe **Properties**.

### 2.2.4 Ugo Baroncini

In questa sezione si discutono le scelte fatte nella progettazione della Procedura e della History.

Già dai primi momenti del progetto, è stata chiara la necessità di implementare un sistema di memorizzazione delle modifiche, e la possibilità di annullarle. Gli spunti principali sono stati la cronologia di modifiche e la feature dei layer di Adobe Photoshop, su cui ci siamo confrontati abbastanza per decidere come procedere. Durante la prima progettazione iniziale ho elaborato il concetto di History, un contenitore in cui venivano salvate le modifiche effettuate all’immagine, che dopo una prima implementazione si è rivelato completamente insufficiente.

La History infatti rappresentava le modifiche effettuate all’immagine iniziale, permettendo di aggiungere, rimuovere e modificare i SequenceNode, gli oggetti che rappresentavano la modifica, ma senza contenere in qualunque maniera le informazioni riguardanti l’ordine di applicazione di tali modifiche, rendendo impossibile un annullamento corretto. Terminata questa prima implementazione ho avviato una fase di upgrade della attuale History, mirata principalmente a separare il vero concetto di History dalla serie di modifiche applicate all’immagine. Sempre in questa fase decidiamo di limitare a una il numero di modifiche effettuabili con uno stesso ImageTool. Qui, la vecchia implementazione della History diventa Procedure (dall’idea di ”procedura di modifica”), e i SequenceNode diventano Adjustment (oggetti che rappresentano la singola modifica, applicata tramite un ImageTool, contenuto al loro interno).

A questo punto la Procedure contiene gli Adjustment applicati dall’utente all’immagine, ed ecco che il concetto di History diventa più chiaro: deve contenere (in ordine cronologico) le Action, ovvero le azioni fatte dall’utente che generano un Adjustment.

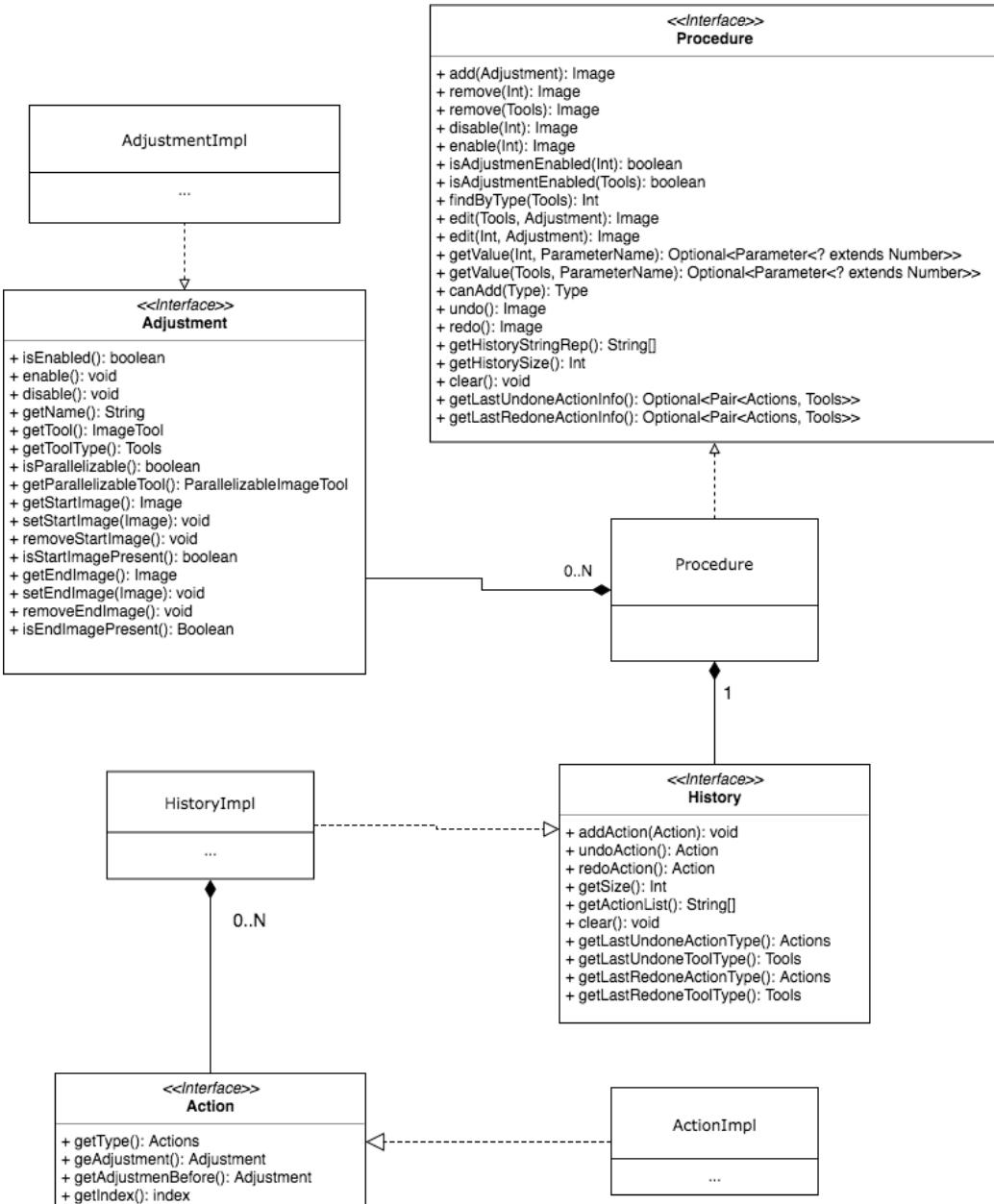


Figura 2.7: Schema UML architetturale di Procedure, History e componenti associati.

La `Procedure` contiene al suo interno una istanza di `History`, ogni volta che una azione di tipo `add` o `edit` viene eseguita è anche automaticamente salvata nella `History`. Per eseguire gli `undo` e `redo`, la `Procedure` espone dei wrapper agli stessi metodi della `History`, che prelevano la `Action` dalla `History` e

si occupano di adeguare il contenitore degli Adjustment in base alla richiesta.

Attualmente, nella GUI, vengono implementate solo alcune funzionalità tra quelle disponibili dalla parte Model della Procedure. In particolare, vengono offerte: la possibilità di rimuovere direttamente un Adjustment, in un qualunque punto arbitrario della Procedure, senza dover retrocedere manualmente tramite la History; la possibilità di abilitare e disabilitare un qualunque Adjustment, azione che non viene salvata nella History ma disattiva (o riattiva) temporaneamente il suddetto Adjustment, senza rimuoverlo dalla Procedure; e infine la possibilità da parte del controller di richiedere informazioni sulle ultime Action annullate o ripristinate, che risulta utile alla view per poter ripristinare i controlli (eg: slider) nella posizione corretta dopo, ad esempio, un undo.

La elaborazione delle immagini è gestita automaticamente dalla Procedure: dopo ogni azione che la modifica viene riprocessata la serie di **Adjustments** per generare l'immagine finale, che viene restituita al controller. Per limitare il costo di calcolo dell'immagine, ogni Adjustment può memorizzare l'immagine prima dell'applicazione del suo tool e l'immagine risultante. Grazie a questa cache è possibile evitare di ricalcolare tutto lo stack di filtri ad ogni modifica.

# Capitolo 3

## Sviluppo

### 3.1 Testing automatizzato

Si sono svolti diversi test automatizzati, soprattutto nella parte del model, per verificare che le cose funzionassero correttamente. In particolare, i test automatizzati principali organizzati sono stati 3:

- Il primo, `ParallelTest`, riguarda i strumenti da eseguire in parallelo. In particolare, il test si occupa di verificare che non siano presenti differenze di risultato fra l'applicazione dei strumenti in parallelo e quella in single-thread. Unica forse pecca di questo test è che è necessario inserire il percorso dell'immagine su cui eseguire i test.
- Il secondo, `AdjustmentTest`, si occupa di verificare che i vari `Adjustment` della `Procedure` siano stati implementati correttamente.
- Il terzo, `ProcedureTest`, verifica il corretto funzionamento della `Procedure` e della `History` integrata.

### 3.2 Metodologia di lavoro

Come DVCS, si è scelto di usare git. Per l'uso di tale strumento, si e' deciso di sviluppare la maggior parte del lavoro sul branch `develop`, creando, ove necessario, dei feature branch per implementare parti complesse. Una volta che feature importanti venivano completate, si provvedeva a fare il merge sul `master`. Sotto trovate le varie parti svolte da ciascun membro del progetto.

### 3.2.1 Matteo Fiordarancio

Matteo si è occupato del design dell'intero model con Baroncini Ugo, implementando poi la sezione inerente la gestione degli strumenti e dando supporto per l'integrazione con il controller. In particolare, si elencano qui le classi svolte da Matteo:

- `ColorUtils`
- `Image`
- `ImageImpl`
- `ImageUtils`
- `ParallelFilterExecutor`
- `AbstractImageTool`
- `ImageTool`
- `ParallelizableImageTool`
- `Parameter`
- `ParameterImpl`
- `ParameterName`
- `BlackAndWhite`
- `Brightness`
- `Contrast`
- `Cropper`
- `Hue`
- `Saturation`
- `Brightness`
- `Rotator`
- `SelectiveRGBChanger`
- `Vibrance`

- `WhiteBalance`
- `Tools`, in collaborazione con Giovanni Gobbi e Baroncini Ugo
- `SequenceNodeTest`
- `ParallelTest`
- `Timer`
- `MutablePair`, in collaborazione con Giovanni Gobbi

Matteo ha inoltre stabilito il design UML di:

- `Procedure` con Baroncini Ugo (la prima versione, che e' stata poi aggiornata da Ugo)
- Gestione delle Immagini con Baroncini Ugo
- Gestione dei Tools con Baroncini Ugo
- Parallelizzazione

### 3.2.2 Giovanni Gobbi

Giovanni si è occupato di parte fondamentale della View e dello sviluppo dei componenti del Controller. In seguito si elencano tutte le classi sviluppate da Giovanni:

- `AppManager` in collaborazione con Jacopo Lunadei
- `AppManagerImpl` in collaborazione con Jacopo Lunadei
- `IOManager` in collaborazione con Jacopo Lunadei
- `IOManagerImpl` in collaborazione con Jacopo Lunadei
- `Format`
- `View`
- `ViewController`
- `AbstractViewController`

- `AbstractViewControllerWithManager`
- `AnimationUtils`
- `InputOutOfBoundException`
- `ComplexNode`
- `CropArea`
- `RotateLine`
- `ZoomableImageView`
- `ZoomDirection`
- `ExportView`
- `ExportController`
- `LoadingView`
- `LoadingController`
- `ViewTools`
- `MainView`
- `MainController` in collaborazione con Jacopo Lunadei
- `WelcomeView`
- `WelcomeController`
- `BarlugoFX`

In aggiunta a ciò Giovanni ha sviluppato anche:

- `style.css` in collaborazione con Jacopo Lunadei
- `FXMLWelcome.fxml`
- `FMLLoading.fxml`
- `FXMLMain.fxml`, in collaborazione con Jacopo Lunadei
- `FMLExport.fxml`

Giovanni si considera autore di tutte le classi sopra elencate. Pertanto, eventuali errori saranno da imputarsi a lui soltanto.

### 3.2.3 Jacopo Lunadei

Jacopo si è occupato della parte di gestione dei preset, sia per quanto riguarda la parte di view che quella dei controller. In seguito si elencano tutte le classi sviluppate da Jacopo:

- `PresetView`
- `PresetController`
- `IntegerStringConverter`
- `DoubleStringConverter`
- `AppManager`, nella parte riguardante i preset
- `AppManagerImpl`, nella parte riguardante i preset
- `IOManager`, nella parte riguardante i preset
- `IOManagerImpl`, nella parte riguardante i preset
- `MainController`, nella parte riguardante i preset

In aggiunta a ciò Jacopo ha sviluppato anche:

- `style.css`, nella parte riguardante i preset
- `FXMLMain.fxml`, nella parte riguardante i preset
- `FXMLPreset.fxml`

Jacopo si considera autore di tutte le classi sopra scritte, tranne di quelle segnalate. Pertanto, eventuali errori saranno da imputarsi a lui soltanto.

### 3.2.4 Ugo Baroncini

Ugo si è occupato della progettazione e realizzazione della Procedure e della History. In seguito si elencano tutte le classi sviluppate da Ugo:

- `Procedure`
- `ProcedureImpl`
- `History`

- `HistoryImpl`
  - `Adjustment`
  - `AdjustmentImpl`
  - `Action`
  - `ActionImpl`
  - `Actions`
  - `HistoryActions`
  - `Tools`, in collaborazione con Matteo Fiordarancio e Giovanni Gobbi
  - `AdjustmentAlreadyPresentException`
  - `NoMoreActionsException`
  - `ProcedureTest`, in collaborazione con Matteo Fiordarancio
  - `AdjustmentTest`, in collaborazione con Matteo Fiordarancio
- .

### 3.3 Note di sviluppo

#### 3.3.1 Matteo Fiordarancio

Dovendo denotare le cose piu' interessante svolte da Matteo, fra queste spiccano sicuramente:

- Gestione dei colori ARGB (alpha + red + green + blue) attraverso un unico intero con la creazione della classe `ColorUtils`
- Uso dei Raster della `bufferedImage` per la scrittura su di essa
- Approfondimento dei java `Executor` per la parallelizzazione dei vari tools, con scelta finale, per motivi di performance, sui `CachedThreadPool`
- Approfondimento dei `CountDownLatch` per la gestione dell'inizio e terminazione dei vari thread evitando la chiusura degli stessi

- Scelta dell'algoritmo di parallelizzazione, concretizzatosi in una divisione in bande verticali, in modo che si possa sfruttare al meglio il caching (attraversare un array per righe è sempre meglio che attraversarlo per colonne)
- Utilizzo della wildcard bounded covariante per la gestione dei possibili valori accettati dai tools
- Uso degli optional per `AbstractImageTool`
- Uso del default per l'implementazione di un metodo nell'interfaccia `ParellalizableImageTool`
- Uso delle lambda e stream in `WhiteBalance` e in `ParallelFilterExecutor`
- Uso delle librerie `AffineTransform` e `AffineTransformOp` per la rotazione
- Tutti gli algoritmi sono stati trovati online e riadattati in java. Cito alcune delle fonti maggiori di algoritmi che ho segnato (trovate comunque tutto nella javadoc o nei commenti al codice)
  1. <https://adadevelopment.github.io/gdal/white-balance-gdal.html> per il `WhiteBalance`, ispirato da GIMP
  2. <https://www.dfstudios.co.uk/articles/programming/image-programming-algo> per gli algoritmi più semplici di manipolazione di immagini
  3. <https://stackoverflow.com/questions/10426883/affinetransform-truncates-rq=1> per l'algoritmo di rotazione
  4. <http://redqueengraphics.com/category/color-adjustments/> per la Vibrance.

### 3.3.2 Giovanni Gobbi

Le sfide più interessanti affrontate da Giovanni sono:

- Creazione di un'interfaccia grafica moderna e accattivante in grado di essere semplice e intuitiva anche a chi non ha mai utilizzato un software di editing prima d'ora
- Adattamento dell'interfaccia grafica ai moderni display con un range di possibili risoluzioni molto ampio

- Gestire gli eventi di Zoom e Panning dell’utente (sviluppo del componente `ZoomableImageView`)
- Sviluppo di strumenti grafici di ritaglio e rotazione dell’immagine (`CropArea` e `RotateLine`)
- Sviluppo di un buon design per la parte di View utilizzando le feature fornite dalla tecnologia JavaFX (divisione in file fxml e controller degli eventi)
- Estensione della classe `View` per l’utilizzo di essa con tutte le View presenti nel sistema (utilizzo del tipo generico `<T extends ViewController>`)
- Estensione dell’interfaccia `ViewController` in grado di adattarsi ai due tipi di `ViewController` presenti nel sistema (`AbstractViewController` e `AbstractViewControllerWithManager`)
- Utilizzo della libreria `ImageIO` per l’esportazione dell’immagine in vari formati, tra cui JPEG in formato compresso

### 3.3.3 Jacopo Lunadei

Le sfide più interessanti affrontate da Jacopo sono:

- Utilizzo di stream e lambda in `PresetController`
- Utilizzo della classe `Properties` per il salvataggio e l’applicazione dei preset
- Utilizzo di `Reflection` per l’applicazione filtri di un preset
- Utilizzo delle wildcards per la gestione di spinner di tipi diversi in `PresetController`

### 3.3.4 Ugo Baroncini

Le sfide più interessanti affrontate da Ugo sono:

- Progettazione della Procedure e History allegata
- Implementazione delle suddette
- Test e controlli adeguati alla verifica del funzionamento di classi così sostanziose.

# Capitolo 4

## Commenti finali

### 4.1 Autovalutazione e lavori futuri

#### 4.1.1 Matteo Fiordarancio

Questo lavoro mi è stato molto utile in generale. Ho cercato di prestare particolare attenzione al modo in cui il codice è stato scritto e architettato. Per questo, ritengo che il mio codice sia di buon livello. Eventuali punti di debolezza penso possano essere trovati nella conversione dei parametri a valori (ma anche li, non saprei come farlo meglio) e magari una gestione migliore della parallelizzazione (di cui al momento non sono a conoscenza). In generale, mi ritengo molto soddisfatto per l'architettura da me scelta.

Riguardo il mio ruolo nel gruppo, ritengo di essere stato un buon elemento per il team. Ho sempre cercato di essere trainante e disponibile (anche se delle volte avrei potuto fare meglio probabilmente).

Da questo progetto ho verificato che il design prima della effettiva scrittura del codice è fondamentale. Penso di aver imparato in generale come si costruisce del software relativamente complesso e come scrivere correttamente una relazione. Ho migliorato la mia capacita' di stare in team e appreso varie cose che mi saranno utili per interview per posti di lavoro ai quali ho applicato o applicherò.

Una cosa che sicuramente dovrò imparare sarà come svolgere benchmark affidabili in Java ma in generale in qualunque linguaggio. Avere questa conoscenza mi sarebbe stata molto utile per determinare, in maniera precisa, alcune parti di codice come *shouldYouParallelize()* di `ParallelFilterExecutor`

### **4.1.2 Giovanni Gobbi**

Complessivamente ritengo di aver svolto un buon lavoro e di aver contribuito in maniera significativa alla realizzazione del progetto. La parte principale da me svolta, ovvero la View, ha richiesto molto tempo per apprendere come sviluppare un buon design per ogni singola interfaccia grafica utilizzando la tecnologia JavaFX e, nonostante non avessi mai sviluppato con questa tecnologia prima d'ora, credo di aver diviso in maniera adeguata le varie sezioni in modo che ogni classe/file fxml svolga una precisa funzione separata dalle altre.

Ritengo inoltre di aver realizzato un'interfaccia grafica abbastanza carina che non sia troppo difficile da capire per un qualsiasi utente.

Una critica che mi posso fare è quella di non aver architettato una buona ereditarietà delle varie classi sin da subito, ma la maggior parte delle modifiche che hanno portato al design finale sono state implementate durante lo sviluppo e non erano note a priori.

Per quanto riguarda il ruolo nel gruppo, credo che la mia presenza sia stata apprezzata dagli altri componenti in quanto ho sempre rispettato i compiti assegnatimi terminandoli nelle scadenze prefissate. Credo di essere stato maggiormente d'aiuto nella fase di design dell'intero software evidenziando le funzionalità che il software doveva offrire.

Infine aggiungo che, se dovessimo riscontrare un effettivo utilizzo del software da parte di vari utenti, sarei più che felice di continuare lo sviluppo integrando nuovi strumenti e funzionalità tra cui, secondo me fondamentale in un editor fotografico, la possibilità di effettuare modifiche direttamente sul file RAW nativo ed esportare in formato compresso.

### **4.1.3 Jacopo Lunadei**

Ritengo che questo lavoro sia stato utile per comprendere il funzionamento di JavaFX e la suddivisione in classe/file fxml, utilizzato da noi con il fine di creare un buon design. Considero questo lavoro molto utile dato che sono riuscito ad approfondire temi più complessi quali wildcards e reflection, molto utili per il mio compito di gestione dei preset.

Una critica che posso fare riguardo al mio ruolo nel gruppo è quella di non aver contribuito abbastanza alla realizzazione del progetto. Ciò è dovuto al fatto che, essendo rimasto indietro con gli esami, ho dovuto posticipare l'inizio del progetto e, quando ho iniziato, non ho valutato correttamente le tempistiche della realizzazione e i problemi che avrei dovuto affrontare. Credo comunque di aver fatto un buon lavoro per i compiti a me assegnati dagli altri componenti del gruppo.

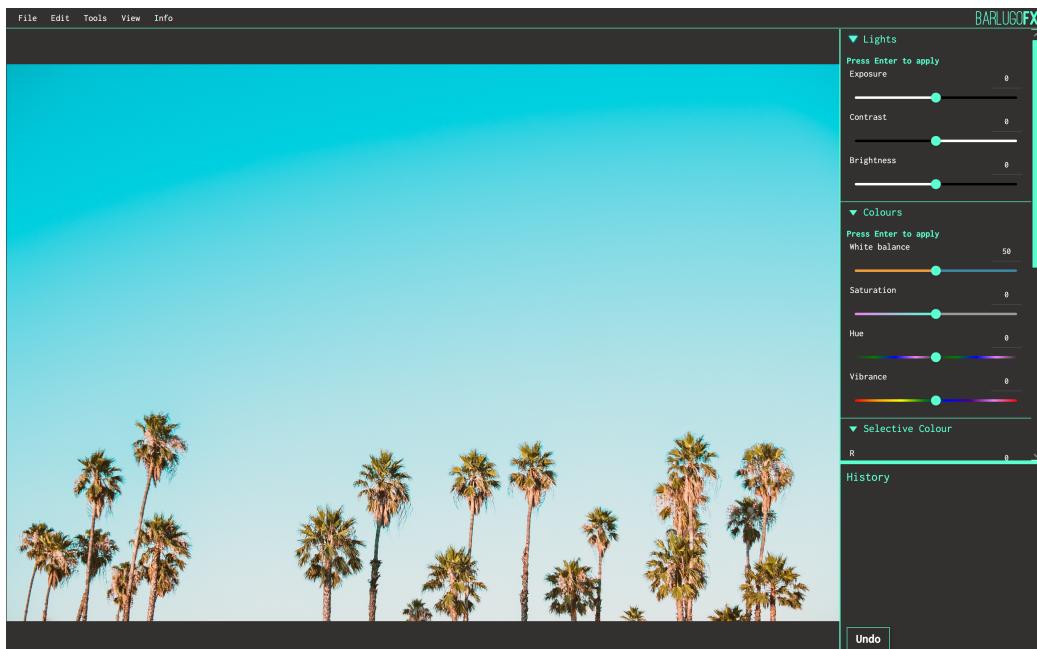
#### **4.1.4 Ugo Baroncini**

Sono molto soddisfatto del lavoro svolto, un interessante upgrade futuro che vorrei gestire è un alleggerimento del sistema di caching delle immagini nella History.

# Appendice A

## Guida utente

Questa breve guida è sviluppata per guidare l'utente nell'utilizzo dell'applicazione. Una volta scelta l'immagine da caricare dalla schermata di benvenuto del programma (o CTRL + N) la schermata dell'applicazione sarà la seguente:



Tenendo premuto il tasto sinistro del mouse, è possibile spostare l'immagine a proprio piacere. Muovendo invece la rotella del mouse, è possibile zoomare avanti o indietro a piacere. Facendo un rapido **doppio click** sull'immagine, è possibile ritornare alla situazione iniziale.

Sulla destra si trova una serie di strumenti che è possibile applicare ad un'immagine. Per ogni strumento è possibile settare un valore sia muovendo lo slider che scrivendo il valore nella casella di testo a fianco.

Per gli strumenti presenti nelle sezioni **lights** e **colours**, una volta modificato il valore, basterà premere **enter** (return su macOs) per applicare le modifiche.

Per le sezioni **selective colour** e **black n white**, è presente un tasto **apply** in fondo alla singola sezione che applica la modifica all'immagine.

### A.0.1 Crop

Selezionando il Crop (CTRL + C o dal menu Tools), comparirà un rettangolo che è possibile ridimensionare o muovere. Una volta selezionata l'area da ritagliare, basta premere **enter** (return su macOs) per applicare il Crop.

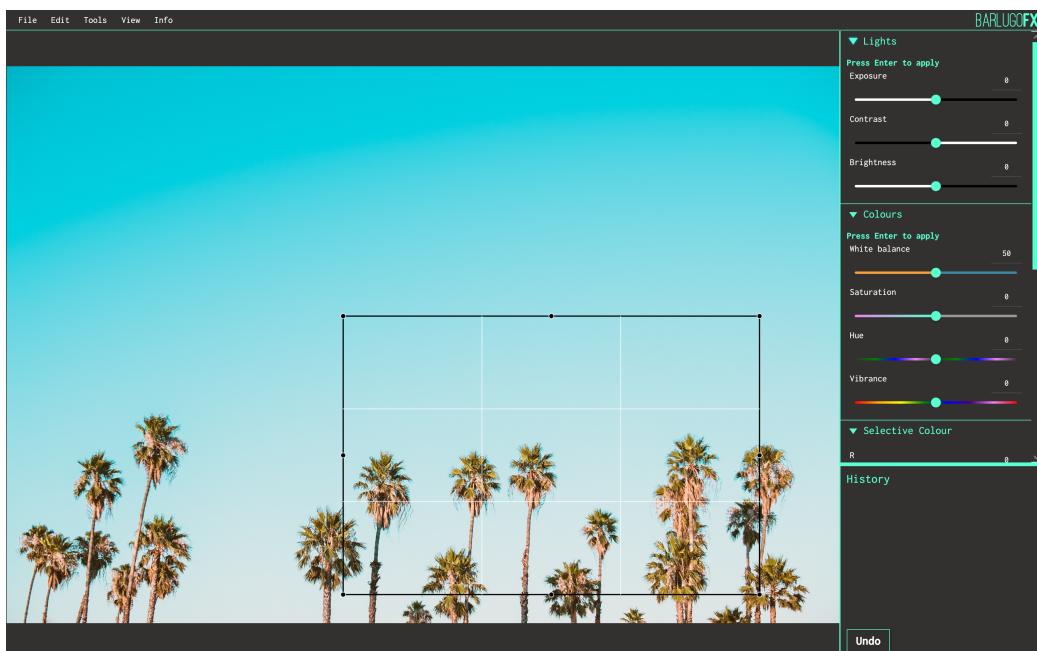


Figura A.1: Il rettangolo che appare selezionando lo strumento di crop

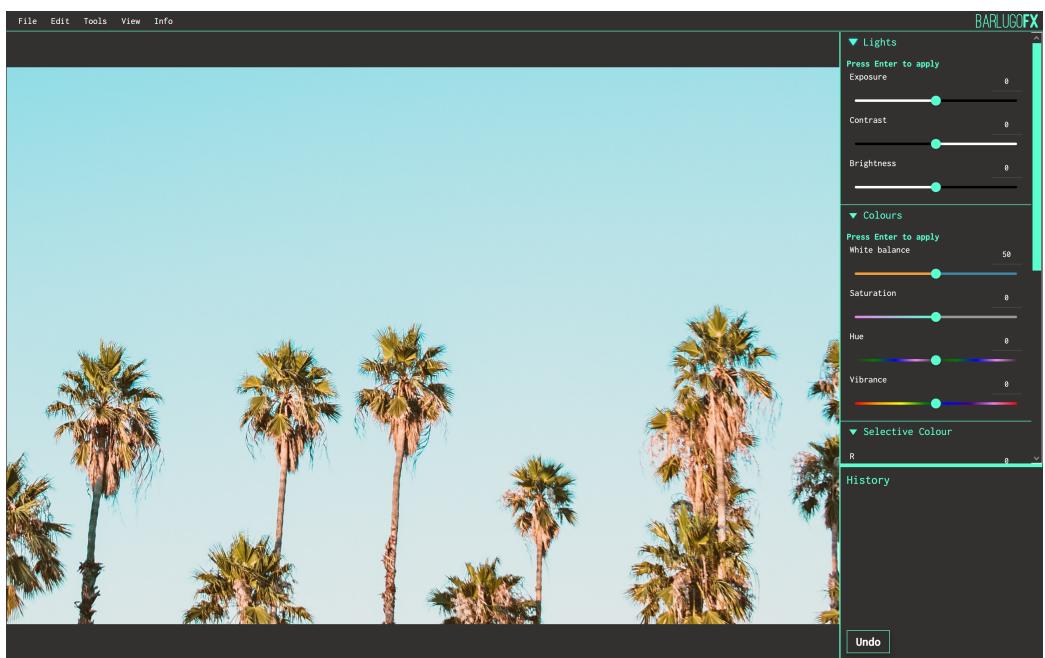


Figura A.2: L'immagine risultante una volta applicato il Crop

### A.0.2 Rotate

Per utilizzare il rotate (CTRL + R o dal menu Tools), premere il pulsante sinistro del mouse e tracciare una linea per determinare di quanto ruotare l'immagine.

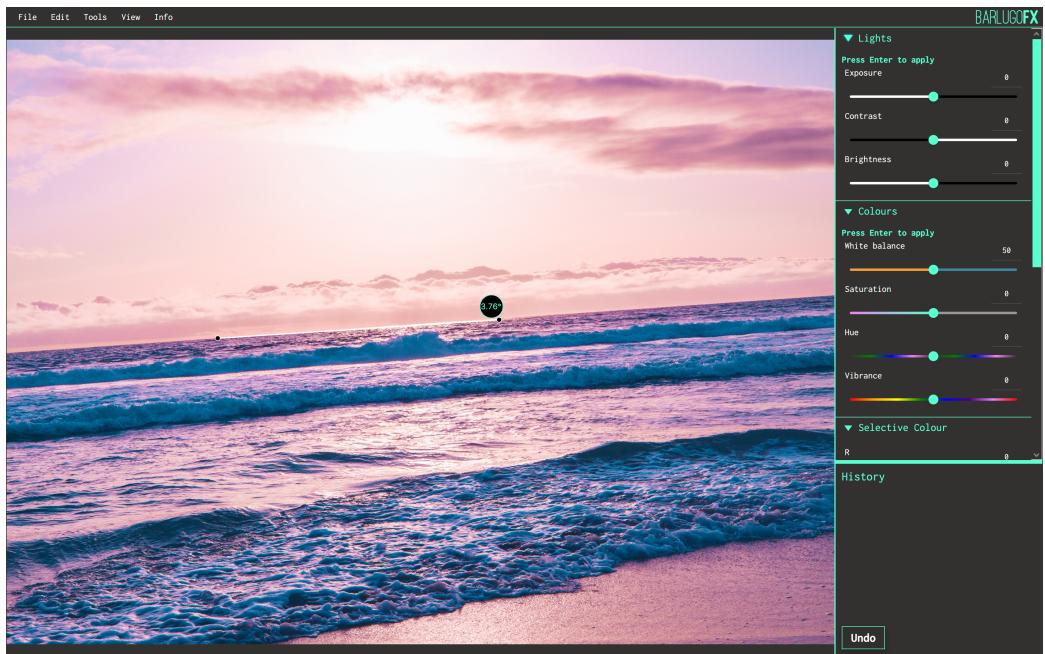


Figura A.3: la linea che appare mostrando anche l'angolo di rotazione

Lo strumento è particolarmente adatto in situazioni in cui l'orizzonte di una foto non è perfettamente allineato, come da foto d'esempio. Una volta soddisfatti dell'angolo di rotazione, basterà rilasciare la pressione del mouse per avere l'immagine ruotata.

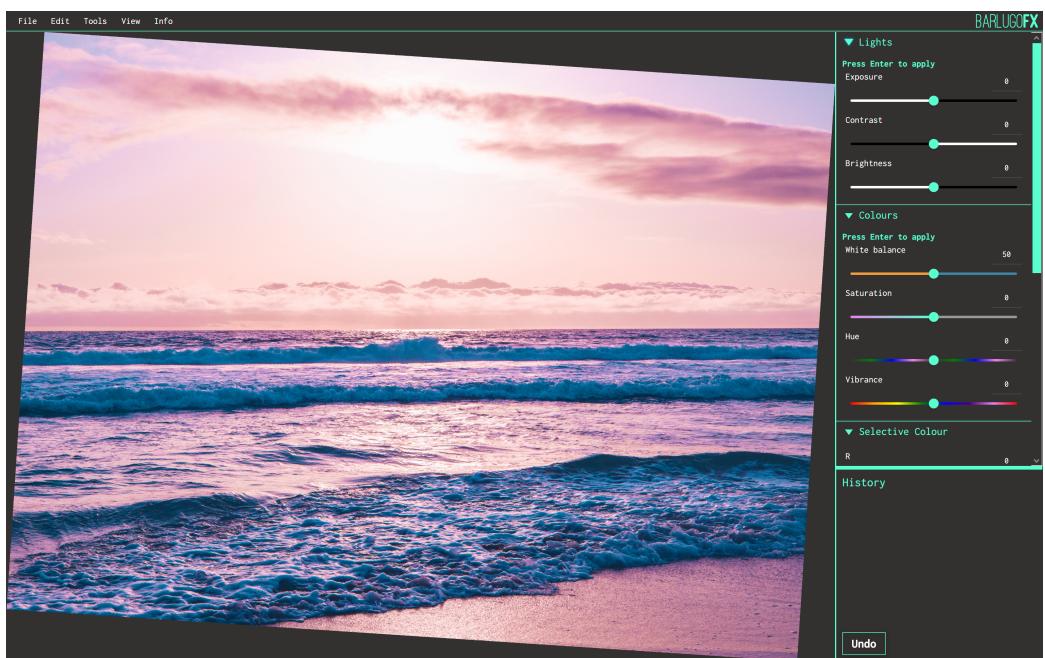


Figura A.4: L'immagine risultante una volta applicato il Rotate