Politecnico di Torino

Computer Vision

Learn OPENCV

Bartolotta Marco

Facchini Dario

Anno Accademico 2013/2014

Introduzione

Premessa

OpenCV è una libreria rivolta ai programmatori, interessati a realizzare applicazioni con risultati efficaci e con alte prestazioni. Data la natura del contesto, uno studio approfondito delle librerie, richiesto per progettare una buona applicazione di visione artificiale, non assicura comunque un immediato raggiungimento dell'obiettivo in maniera ottimale.
Nell'elaborazione delle immagini, in campi all'avanguardia o dove c'è carenza di esperienza, si affronta un lungo processo di trial-and-error che comporta più volte la riscrittura di parte del codice e uno studio sempre più approfondito della documentazione.
In definitiva, convincersi di aver raggiunto una soluzione sub-ottima significa spesso esplorare a fondo e nel dettaglio le molteplici possibilità offerte dalla libreria, integrandole di volta in volta nel codice.

Obiettivo

Si è avvertita l'esigenza di proporre una nuova interfaccia (logica e grafica) per l'interazione con le librerie openCV. I presupposti per la sua progettazione sono di orientamento didattico. L'obiettivo è di classificare le funzionalità offerte dalle suddette librerie, modellare un comportamento comune a tutti i prototipi e fornirne una astrazione, al fine di:

1. creare una raccolta omogenea delle funzioni di base di openCV, incapsulate in classi Java;
2. corredarle con informazioni e funzionalità comuni per lo studio e il test delle stesse;
3. fornire all'utente uno strumento intuitivo per combinarle tra loro, modificarne le proprietà, valutarne l'effetto finale e generare codice Java equivalente;
4. fornire al programmatore un buon meccanismo che gli consenta di implementare i propri metodi di interazione con le funzioni offerte dalla libreria in maniera intuitiva e trasparente per l'utente finale.

E' stato posto l'accento sulla facilità d'uso anziché sulle prestazioni, fatta eccezione (ove possibile) sull'esperienza interattiva dell'utente.

Soluzione proposta

Progettazione

Struttura file

Si è scelto di dividere il progetto in due package:

- *learnOpenCVBlocks*: contiene la classe astratta chiamata "LearnBlock" comune a tutti i blocchi della libreria. Ogni blocco quindi ne implementa le funzionalità in quanto sottoclasse specializzata di LearnBlock;
  sono qui inclusi i file relativi a descrizione (sottopackage "desc"), codice (sottopackage "code") e file fxml (sottopackage "fxml") locali ad ogni blocco;
- *learnOpenCV*: conterrà le classi per la gestione dell'interfaccia grafica e della logica globale dei blocchi;
  sono qui inclusi i file relativi a:
  - file fxml (sottopackage "fxml") che descrive l'interfaccia grafica principale. La loro modifica può avvenire solo se si ha dimestichezza con il codice del programma in quanto sono presenti diversi bindings (fx:id) necessari al suo funzionamento;
  - template del codice complessivo generato da programma ("code"), completamente personalizzabile. Di default, il template contiene una classe formata da:
    - più metodi privati (uno per blocco) con l'implementazione definita nel singolo file di codice, della versione selezionata;
    - un metodo pubblico che si occupa di eseguirle in sequenza e, dato un input, fornire l'output finale.

L'implementazione di un nuovo blocco prevede l'inserimento dei file .class, .fxml, .txt, ad esso relativi, nel primo package e la SOLA aggiunta del blocco alla lista presente nella classe EditBlock.

Struttura classi: diagramma UML

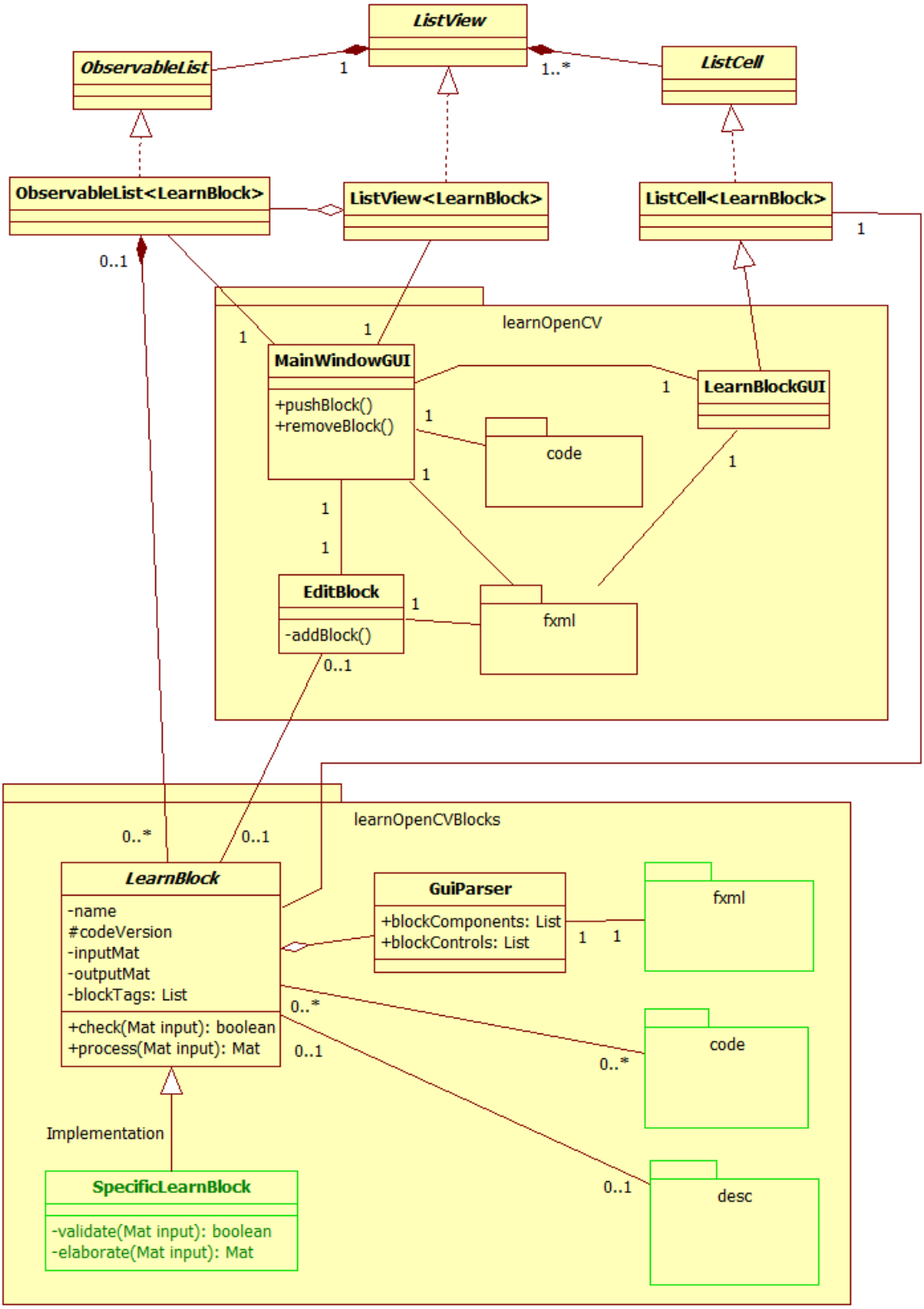

Figura A: *diagramma UML con i metodi di base per le funzioni principali*

Implementazione

Di seguito saranno descritti i metodi principali implementati per una migliore comprensione del codice, tralasciando le funzioni di utlità in ogni classe e le logiche di più basso livello. Si useranno d'ora in avanti il *corsivo* per identificatori interni al codice, i "doppi apici" per nomi o concetti specifici ad un contesto o per identificare elementi grafici, il sottolineato per nomi di file o risorse.

Logica a blocchi: LearnBlock

Questa classe è responsabile di tutte le funzioni del singolo blocco. Ognuno di essi progettato come un'entità atomica SISO stateless fornito di controlli. Gode inoltre della possibilità di essere "validato" ed "eseguito" dopo averne inpostato l'input, ovvero un oggetto di tipo Mat, al fine di ottenerne un output dello stesso tipo. LearnBlock contiene inoltre funzioni e variabili accessibili dall'esterno (contrassegnate come pubbliche) e dalle classi figlie (contrassegnate come protette). Ogni "LearnBlock" è descritto da un *name* (lo stesso della classe) e racchiude in sé:

- funzioni specifiche per la validazione e per l'esecuzione;
- un file fxml, che ne rappresenta l'interfaccia grafica con controlli settabili dall'utente;
- un testo di descrizione della funzione implementata nel blocco;
- uno o più file di testo (se si prevede di fornire più versioni della stessa funzione) per la generazione dinamica del codice equivalente del blocco.

Il tutto è definito esclusivamente dal programmatore del blocco. Si è invece scelto di non delegare a quest'ultimo l'onere di:

- caricare i file necessari al corretto funzionamento del blocco (saranno caricati automaticamente dalle directory predefinite in base al nome della classe);
- gestire l'interazione con le classi esterne (la GUI di base, la gestione, la validazione e l'esecuzione sono comuni a tutti i blocchi);
- garantire l'integrazione con il resto del programma (il caricamento del blocco avviene solo se sono valide tutte le condizioni espresse nella documentazione). Errori in esecuzione non gestiti , locali al blocco, saranno segnalati a runtime senza compromettere il funzionamento dell'intero ambiente.

Proprietà

In quanto unità di elaborazione, LearnBlock presenta una *inputMat* ed una *outputMat*, ovvero una copia clone dell'ultimo input/output impostato/generato, oltre che due viste Image bufferizzate *inputImg* e *outputImg* automaticamente generate, che potranno essere visualizzate da GUI JavaFX.

Si tiene poi traccia dello stato di esecuzione di un blocco attraverso più *SimpleBooleanProperty* accessibili all'esterno per effettuarne la valutazione ed il binding. Esse sono:

- *isActive* (default=true): quando è false l'esecuzione è bypassata e dunque l'output riportato è identico all'input;

- *isReady* (default=false): è true solo se tutti i file del blocco sono stati correttamente caricati e non presentano errori (impostato da *checkLoading()*). È solo a titolo informativo: se infatti il caricamento del solo file fxml fallisce, il blocco lancia *IOException* da costruttore (vedi *loadFXML()*);
- *isValid* (default=false): riporta lo stato dell'ultima controllo; in pratica riporta una previsione dell'esito della prossima esecuzione con le impostazioni correnti;
- *isExecuting* (default=false): è true solo durante l'esecuzione di *process()*, può essere utile per segnalare all'utilizzatore inizio e termine della stessa.
- *wasModified* (default=true): il suo utilizzo è demandato interamente all'utilizzatore esterno di LearnBlock (es. una GUI); il suo valore è impostato a false ogni qualvolta il blocco è eseguito, ma vale true solo quando il metodo *notifyModification()* è chiamato. In pratica può indicare se una modifica è stata fatta al blocco dall'ultima esecuzione.

Caricamento

LearnBlock presenta riferimenti ai tre file principali: fxmlPath, docPath, codPath. I path sono costruiti in base al nome della classe specifica che implementa LearnBlock (in Java è possibile con *getClass().getSimpleName()*). Il corretto caricamento di tutti e tre i file in fase di costruzione determina il valore della proprietà *isReady*. In particolare:

- *thisBlockDescription* contiene il testo del file di descrizione caricata da *loadDescription()*, se presente;
- *thisBlockSymbolicCode* contiene le diverse versione di codice individuate da *loadCode()*, se presenti e valide;
- *thisBlockRenderedCode* contiene l'ultima versione renderizzata di codice, ovvero con tag rimpiazzati da valori numerici o testuali, ogni qualvolta *renderCode()* è chiamata;
- *guiElements*, istanza della sottoclasse GuiParser: essa contiene i riferimenti ai singoli elementi grafici fxml del blocco (i "Components" in blockOrderedComponents) e, se presente anche il codice, ai singoli controlli veri e propri del blocco (i "Controls" in *blockOrderedControls*). GuiParser fornisce inoltre metodi per estrarre e reinserire tali oggetti dalla root fxml del blocco per passarli a GUI esterne[1]. Il Parent *root* conterrà l'oggetto padre caricato dal metodo *loadFXML()*.

In **figura B** è riportato una visione di insieme del caricamento in LearnBlock: la descrizione non è necessaria, così come il codice. L'unica condizione necessaria al caricamento del blocco è la presenza di un fxml valido. Se *loadFXML()* riesce a caricare il file, ogni Component è riconosciuta attraverso l'uso dell'fx:id o id

[1] In JavaFX non è possibile avere due gerarchie che contengono un riferimento alla stessa istanza di una classe; in altre parole un oggetto grafico JavaFX non può comparire in due Scene separate contemporaneamente. Le alternative sono due: creare due istanze dello stesso oggetto e effettuarne il bidirectionalBind oppure spostare la stessa istanza da un albero all'altro tramite metodo addChildren. Quest'ultima è più corretta nel caso considerato, in quanto i controlli di un LearnBlock sono visualizzati in EditBlock (su creazione) e in MainWindow in due momenti differenti e se ne preserva lo stato senza inutili copie.

“control”, richiesto al programmatore del blocco, e inserita nella suddetta lista senza ulteriori controlli. I Components saranno gestiti as-is dall’interfaccia esterna lasciando al designer del blocco massima libertà nel disegno del suo contenuto (con tool come Scene Builder di JavaFX) e dunque sulle impostazioni visibili all’utente finale. Un Component è un tipo generico **javafx.scene.Parent** (possono essere di qualsiasi tipo, da AnchorPane per posizionamento assoluto che GridPane per contenuto relativo). Fintanto che nessun codice è fornito, non è di interesse il contenuto dei Components.
Se del codice è individuato da LearnBlock, sono necessarie regole più restrittive per collegare semanticamente le variabili interne al codice con gli elementi grafici visibili all’utente. Prerogativa dei blocchi è infatti quella di essere trasparenti, sia come funzioni che come codice. La funzione *loadCode()* ha l’onere di individuare tali file di codice, enumerarne le versioni e caricarne il contenuto solo se tutte le condizioni sono rispettate. In sintesi:

1. in ogni versione del codice dovranno comparire i tag <!INPUT> e <!OUTPUT>
2. ad ogni tag (nella forma <! nometag > individuato tra le diverse versioni dovrà corrispondere un oggetto grafico sottoclasse di **javafx.scene.control.Control** con omonimo fx:id o id contenuto in un Component
3. ogni Component dovrà contenere almeno e al più un Control descritto da un tag (ma potrà contenere qualsiasi altro oggetto con qualsiasi altro id o fx:id)

Queste tre regole assicurano che ogni parte grafica del blocco abbia un suo scopo ben preciso, descriva un solo setting del blocco, e al contempo permettono di corredare al controllo qualsiasi altro elemento grafico di supporto. Una classe che eredita da LearnBlock potrà dunque accedere a tutti gli oggetti attraverso fx:id, ma solo ai Controls sarà permesso avere una controparte nel codice che sarà generato da *renderCode()*.

Esecuzione

LearnBlock mette a disposizione di chi crea un nuovo blocco due metodi astratti:

- *boolean validate(final Mat input)*
- *Mat elaborate(Mat input)*

La guida al loro uso è descritta nel dettaglio nel paragrafo di documentazione per programmatori. Ci limitiamo qui a dire che le variabili locali alla classe *error* e *codeVersion* sono fornite al programmatore di classi figlie di LearnBlock per riportare la descrizione di errori o la scelta della versione del codice da renderizzare.

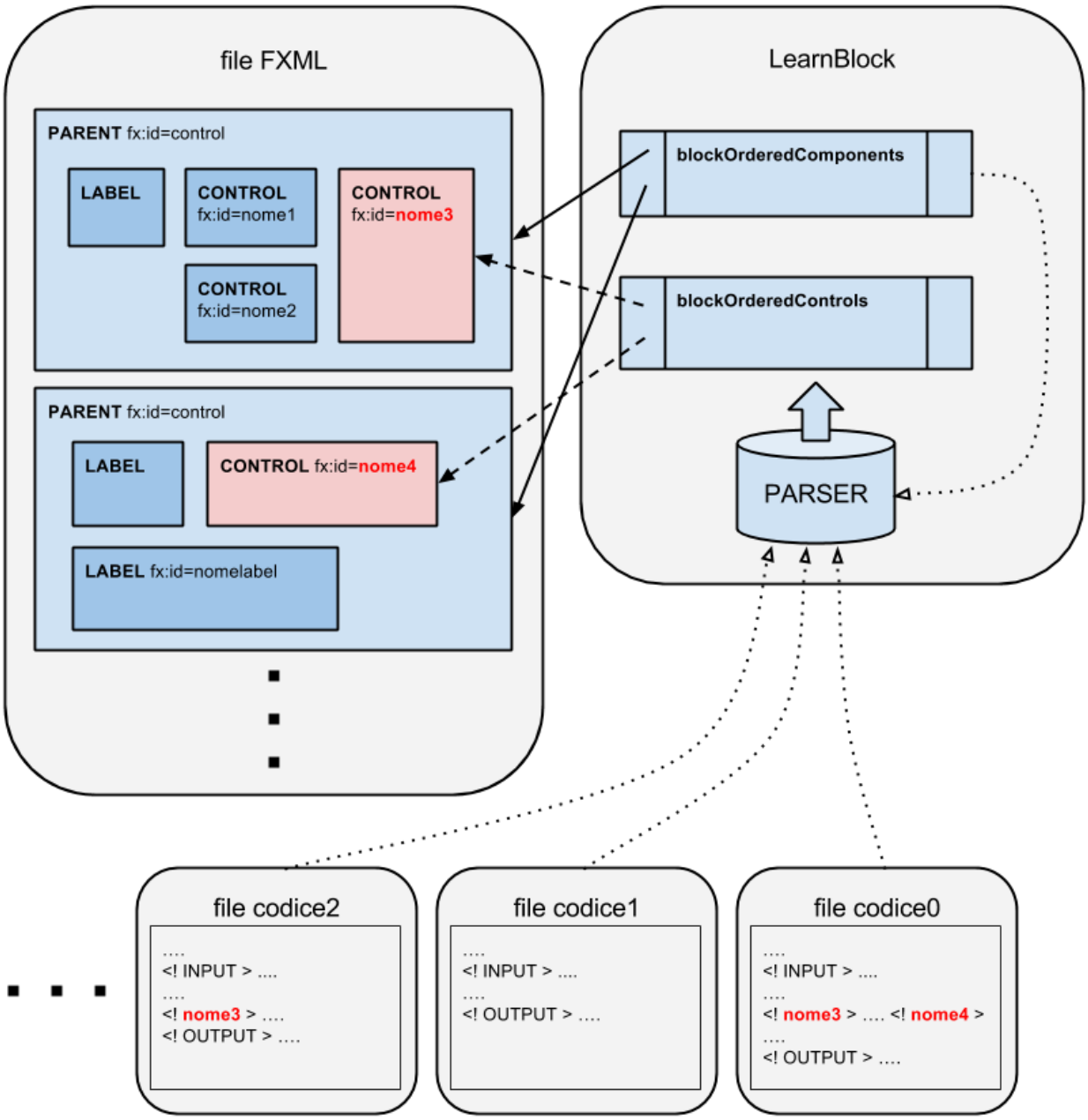

Figura B: *Le frecce continue rappresentano i collegamenti necessari alla costruzione di un LearnBlock. I Container con fx:id corretto saranno SEMPRE indicizzati come "Component" e riportati ordinati nella scermata principale. Le frecce discontinue si realizzano invece SOLO in presenza di codice. In tal caso il Parser considera il numero totale di tag distinti presenti nei diversi file di codice e li confronta con il numero dei Components: se uguale, confronta il loro nome con gli fx:id presenti. Il parsing ha successo solo se per ogni tag esiste solo un Container con un omonimo Control, solo allora il metodo renderCode() può restituire un risultato.*

Un metodo ***check()*** incapsula la sola funzione di *validate()* e permette di verificare dall'esterno che il blocco sia pronto ad eseguire, eventualmente fornendo una Mat di input oppure tentando con la *inputMat* attualmente impostata. In caso di input nullo il risultato è sempre false, ma *validate()* è comunque eseguita, concatenando al contenuto di *error* (contenente quanto la *validate()* è riuscita a riportare prima di ritornare o la lanciare eccezioni) una stringa "Null input is currently set, a valid input is required to get full validation". Cattura inoltre eccezioni *NullPointerException* qualora *validate()* non gestisca bene questo caso. Lo scopo di *check()* è di non permettere che *execute()* sia eseguita con impostazioni o input non validi, preannunciandone per quanto possibile gli errori.
Il metodo ***process()*** implementa l'esecuzione vera e propria, incapsulando *check()* ed *elaborate()* in sequenza, eseguendo la seconda solo se la prima ritorna true. Implementa inoltre controlli su input nullo e sulle diverse proprietà di LearnBlock che ne modificano il comportamento. Si noti che, anche se *check()* ritorna true, *elaborate()* potrebbe comunque fallire: è voluto infatti che *process()* gestisca tutte le eccezioni riportandole come stringa in *error* in quanto è responsabilità del programmatore del blocco sottostante gestirle e della GUI soprastante almeno visualizzarle se non gestite. La *process()* assicura inoltre che ad essere utilizzata nella *elaborate()* è SEMPRE un clone della Mat di input (a scapito delle prestazioni ma consente ad ogni blocco di conservare propria copia di ultimo input e output). Ciò non è vero per la *check()* poiché *validate()* usa input di sola lettura.

Codice

LearnBlock è in grado da solo di generare codice renderizzato (ovvero con valori numerici e stringa testuali) a partire dalle versioni simboliche fornite dal programmatore. Egli è responsabile di impostare la variabile *codeVersion* nella *elaborate()* al valore numerico corrispondente alla versione da renderizzare in base alla casistica da lui decisa. Di default il valore è 0, ovvero la prima (oppure l'unica) versione fornita. E' buona norma dunque chiamare il metodo *renderCode()* solo se *check()* ha successo, in modo che siano rispettati i paradigmi del programmatore del blocco. Per dettagli su come i valori contenuti nei Controls siano convertiti in testo si rimanda alla documentazione per programmatori al paragrafo di implementazione di codice. **Tra la scelta dell'elemento grafico e la rappresentazione del valore nel codice esiste una corrispondenza biunivoca.**

Block Loader: EditBlock

La classe EditBlock si occupa della fase di selezione ed aggiunta dei singoli blocchi. Il costruttore prende come argomento un'istanza (puntatore) di MainWindowGUI al fine di poter terminare l'esecuzione nel caso la schermata principale venga chiusa. All'interno del costruttore viene caricato l'fxml relativo all'interfaccia e ne vengono inizializzate tutte

le primitive grafiche. Infine vengono associati dei listner alle primitive di tipo list al fine di gestire l'interazione dell'utente. I metodi *BlocksImgProc()*, *BlocksCore()* oppure *BlocksAlgorithm()* vengono invocati nel caso l'utente clicchi sull'elemento della lista corrispondente. Essi interpreteranno la selezione dell'utente in base all'indice della lista da quest'ultimo selezionato.
Tramite costrutto switch/case verrà caricato il blocco opportuno e settato il relativo stile grafico designato. Vengono anche settate le textArea presenti nella schermata affinchè mostrino il testo relativo alla descrizione della funzione implementata e il relativo codice invocando la funzione *setGUI()*.
La funzione *preview()* è legata all'interazione dell'utente con il bottone execute: dopo aver controllato che il blocco sia stato correttamente istanziato e che la *check()* del blocco abbia dato esito positivo, preleva l'output restituito e lo mostra a video. Un listener si occupa di effettuare automaticamente il refresh del codice. L'esito negativo di almeno uno dei due controlli mostrerà invece il relativo messaggio d'errore.
E' possibile aggiungere il blocco alla pipe di esecuzione principale tramite bottone "Add", preservandone i parametri impostati. Il metodo *addBlock()* corrispondente ne passa l'istanza a MainWindowGUI tramite funzione *pushBlock()*. Un nuovo blocco dello stesso tipo è caricato in EditBlock, pronto per essere editato. Le funzioni *openImg()* e *saveImg()* si occupano infine dell'apertura di una immagine da elaborare per la Preview e del salvataggio su disco del risultato. Entrambe sfruttano la classe FileChooser.

Pipeline manager: MainWindowGUI e LearnBlockGUI

MainWindowGUI implementa lo scopo del programma: caricare una immagine in *inputMat* tramite il metodo *loadImage()*, processarlo attraverso la pipeline *blocksPipe* e mostrarne il risultato. *compareFinalResult()* permette di confrontare *outputMat* con *inputMat* mentre *saveImage()* di salvarlo su file. Una TextArea *logArea* tiene traccia di tutti i passi del processo (informazioni ed errori).
Mediante metodo *addBlock()* associato a pulsante "Add" una istanza di EditBlock è creata (legata alla variabile *blockStore*): una sola istanza è permessa e un handler si occupa di chiuderne lo stage se lo stage di MainWindow è terminato. Un metodo *pushBlock()* è dunque chiamato da EditBlock per ogni nuovo blocco da inserire in *blocksPipe*.
La ListView *blocksPipeView* è la vista personalizzata che mostra a video i singoli blocchi. Al clic su un blocco, un Listener provvede a caricarne:

- la descrizione in *blockDescription* (tramite *loadDescription()* e visualizzata nel Tab "Info");
- i controlli in *blockControlsGrid* (tramite *loadControls()* e visibili nel Tab "Control"): i Components del blocco (ovvero gli elementi grafici con fx:id="control" e che contengono almeno un elemento Control) restituiti da *getGui().getControls()* sono inseriti uno per riga nella tabella *blockControlsGrid*. Il Listener stesso dunque reinserisce i controlli nella gerarchia originale prima di caricarne i nuovi ogni qualvolta è selezionato un nuovo blocco, chiamando il metodo *unloadControls()*[2];

[2] vedi punto [1]

- l'input e l'output locali del blocco (se presenti, tramite *loadDetails()* e visibili nel Tab "Details): le Mat locali al blocco sono caricate rispettivamente in *blockImgIn* e *blockImgOut* ed è possibile confrontarle tramite metodo *compareLocalResult()*.

Ogni cella di *blocksPipeView* disegna per ogni blocco una interfaccia grafica dedicata (CellFactory) implementata dalla classe LearnBlockGUI, che presenta:

- *blockActiveFlag*: una Checkbox che se disabilitata permette l'override del blocco (il suo input non viene processato ma inoltrato direttamente come output). E' in binding con la proprietà *isActive* di LearnBlock;
- *blockNumber*: una label che riporta l'ordine di esecuzione del blocco nella pipeline
- *checkLed*: un elemento grafico che riporta lo stato corrente del blocco, ovvero
    - spento: il blocco è stato disattivato, ovvero la proprietà *isActive* è false;
    - rosso: il blocco ha riportato un errore in esecuzione, ovvero la proprietà *isValid* è false;
    - giallo: il blocco non ha riportato errori in esecuzione, ma segnala che descrizione o codice sono assenti o non caricati correttamente; la proprietà *isValid* è true ma *isReady* è false;
    - verde: il blocco non ha riportato errori e tutte le funzioni sono operative; le proprietà *isActive*, *isValid* e *isReady* sono tutte true.

  Una animazione segnala se invece lo stato è aggiornato:
    - led fisso: il blocco non è stato modificato dall'ultima esecuzione; la proprietà *wasModified* è false; un listener di MainWindowGUI in ascolto sui controlli si occupa di impostare a true la proprietà quando rileva input dell'utente;
    - led lampeggiante: il blocco è stato modificato ma non ancora eseguito; la proprietà *wasModified* è true;
- *blockName*: riporta il nome del blocco;
- *buttonF*: chiama *showFourier()* e apre una finestra in cui è visualizzata la trasformata di Fourier dell'output del blocco;
- *buttonH*: chiama *showHistogram()* e apre una finestra in cui è visualizzato l'istogramma dell'output del blocco;
- *deleteBlock*: un pulsante che chiama l'omonima funzione per cancellare se stesso dalla pipeline.

LearnBlockGUI inoltre implementa un meccanismo di cambiamento di posizione dei blocchi con trascinamento del mouse (mediante EventHandlers nel costruttore).

In ogni momento è inoltre possibile cancellare tutti i blocchi dalla lista tramite metodo *clearPipe()* accessibile da pulsante "Clear All". Una sezione di log è strettamente legata all'output finale della pipeline, la cui gestione avviene con metodi logicamente simili al singolo LearnBlock:

- ***validatePipe()***: nessun blocco viene eseguito, ma sono chiamate in sequenza le singole *check()* per tentare una stima preventiva dell'errore fornendo come input la Mat del risultato precedente (nel ciclo for chiamata *currentBlockInput*). Per il primo blocco l'input della *check()* sarà sempre presente perché corrisponde all'*inputMat* caricata. Per blocchi successivi, se non sono stati eseguiti in passato,

risulterà *currentBlockInput* nulla: la *check()* è SEMPRE chiamata, in modo che se ben implementata errori riguardanti impostazioni del blocco siano comunque riportati ma il risultato resti false per assenza di input. Con input nullo *check()* concatena sempre al contenuto di *error* (contenente quanto la *validate()* è riuscita a riportare prima di ritornare o la lanciare eccezioni) una stringa “Null input is currently set, a valid input is required to get full validation”. Si prega di notare che, anche se *validatePipe()* ritorna true, *executePipe()* potrebbe comunque fallire a causa di eccezioni non gestite lanciate da *process()* e quindi *elaborate()*! In tal caso il problema è responsabilità dell’implementazione del blocco: se necessario occorrerà scrivere una nuova versione di codice per quel caso.

- ***executePipe()***: i blocchi sono eseguiti in sequenza, fino a generare un errore (errata impostazione oppure una Mat nulla) o una *outputMat* finale non nulla. L’output di ogni *process()* è direttamente usato come input della *process()* successiva. Una Mat di output nulla è considerato errore: l’intera procedura termina immediatamente, riportando in logArea l’ultimo errore/eccezione incontrato tramite *getError()* sul blocco incriminato. Se tutti i blocchi eseguono correttamente *outputMat* è mostrata a video ed è chiamata la funzione *loadCode()*, che sarà descritta a breve. Ricordiamo che è LearnBlock ad assicurarsi che la *elaborate()* del blocco sia chiamata solo se la *validate()* è true e l’input è non nullo. Si può dire che *validatePipe()* esegue un sottoinsieme di operazioni di *executePipe()*, pur non essendo chiamata esplicitamente da quest’ultima. Qualsiasi eccezione lanciata dalla *elaborate()* (e dunque inoltrata da *process()*) del blocco è catturata indistintamente e mostrata a video in *logArea* e rappresenta un caso non gestito dal programmatore del blocco.

Il metodo *loadCode()* genera il render finale del codice della pipeline di blocchi. Si è scelto di chiamare in *executePipe()* ogni qual volta è generata una *outputMat* valida, in modo che si abbia la certezza di un codice valido. Il file template.txt contiene tag e scheletro del codice. Per ogni blocco è chiamato il metodo *renderCode()* (il quale ricordiamo genera un codice valido per gli input e la versione correntemente impostati) e la sequenza è sostituita al tag *<!FUNCTIONS>*. Al tag *<!CALLS>* sono invece sostituite le chiamate delle singole funzioni. I nomi seguiranno il pattern *nomebloccoNUMERO()* dove “NUMERO” è un intero incrementato ad ogni omonimia (se ad esempio lo stesso tipo di blocco è usato più volte nella pipeline). Attenzione: non è la versione del codice usata da quel blocco!