FiatLux : Developer documentation

Application developed by Nazim Fatès

April 13, 2020

Contents

Part I

Use case

Chapter 1

Creating a model

1.1 Implementation of the model

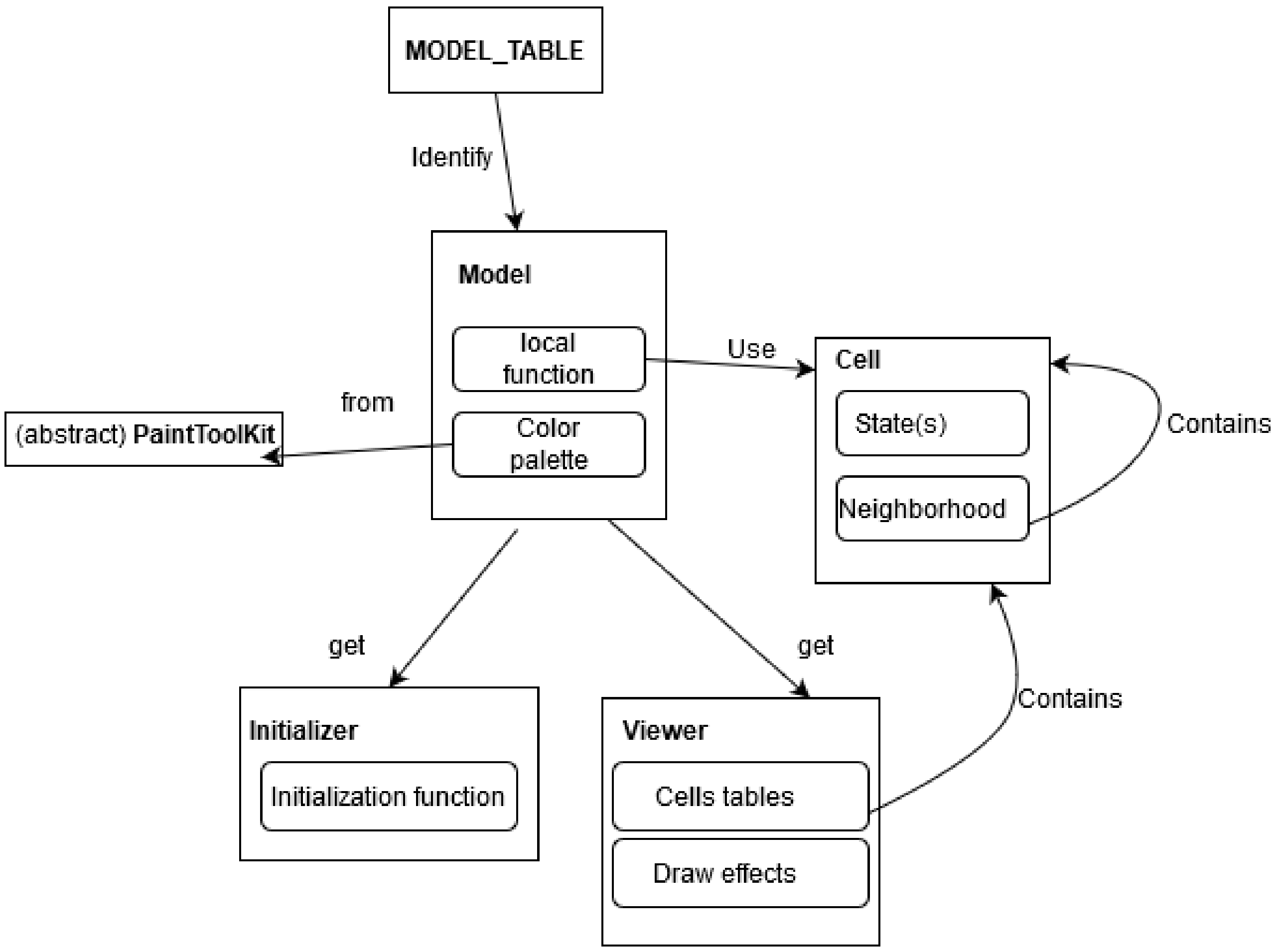

Figure 1.1: Structure : Implementation of a model (simplified)

At the start, you need to develop a class which will contain the model's methods. The package `models` are exclusively created for this and contain packages for each complex system type. Chose the most convenient package for your model. We recommend you create your own package in the ones you took for your model to clarify the hierarchy of the code's application but this's not obligatory.

The name of the class's model must be like `NameOfTheCellularAutomatonModel.java` with no space, uppercase for each word and finish with "-Model". The class will need to extend the type of model's class. You will find this class in the `CAmodels` package. The most used one is `ClassicalModel` but there are the other's as well.

For every model you must define a name. It's important because this variable will serve as a unique reference to the model. Use the same rules than before without the "-Model" termination:

```java
public static final String NAME = "NameOfTheCellularAutomaton";
```

1.1.1 Classical cellular automata's model implementation

ApplyLocalFunction `ClassicalModel` extends `CellularModel` and is the model this uses `OneRegisterIntCell`. `OneRegisterIntCell` is a cell's type whose state is defined by an integer. ClassicalModel implements the method `ApplyLocalFunction(OneRegisterIntCell in_Cell)` which returns an integer : the following state of the cell.

This is the first function you must define in your model. The object `OneRegisterIntCell` contains methods to read the neighbourhood state and the state of the cell itself. With this data you can implement in this function the update of the cells.

```java
@Override
public int ApplyLocalFunction(OneRegisterIntCell in_Cell) {
        // Your function ...
        return intNextState;
}
```

GetDefaultInitializer The second method you must define is `GetDefaultInitializer()` where you will choose the initializer of your model. Just instantiate the one this you wish and return this instance.

In the case where you are modelling a binary cellular automaton this's all you will need. But if you want to implement more specific behavior, you must use different palettes and viewers.

Palette

The color of the simulation is managed by `PaintToolKit`. Many colors stack are already in the application you can choose the one you want for your model by overwriting the `GetPalette()`

method in your model. In this function you need to instantiate a color's stack with `PaintToolKit.GetThePa` and define some color if you want (Particularly the 0 state if you want a rule special for it. For example turn it white like most of the cellular automata).

Here an example with the Rainbow palette but you can found others in the section `PaintToolKit`.

```java
@Override
public PaintToolKit GetPalette() {
        PaintToolKit palette= PaintToolKit.GetRainbow(256);
        palette.SetColor(0, FLColor.c_black);
        return palette;
}
```

1.1.2 Use the model

The model is now usable by command-line but if you want it to appear in the menu you need to register it in the tables of the app. For that, in the package `main¿tables` there is a class named `MODEL_TABLE` which list the different model you can use in the app. In the method `FillTable()` you have to add your model in the list with the command line with the others in the section you'll find the more appropriate:

```java
AddList(NameOfTheCellularAutomatonModel.class);
```

Your will find that the list is stratified by the method `AddSeparator()` which add a strait line in the menu as a separator. And by the line `m_modelType = MT.*TypeOfModel*` this one separate the list into other tabs in the menu. Choose the most convenient tabs and add separators if you wish in order to keep the menu tidy.

Part II

Structure

To implement a model, in addition to the rule to be developed as above, it is necessary to use viewer and initializer classes in order to be able to employ it in the software. The hierarchy of these classes is explained here.

1.2 Viewer

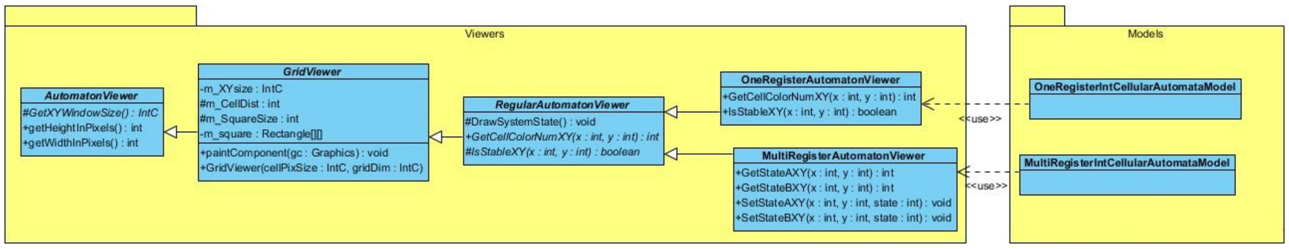
Viewers
Models
AutomatonViewer
#GetXYWindowSize() : IntC
+getHeightInPixels() : int
+getWidthInPixels() : int
GridViewer
-m_XYsize : IntC
#m_CellDist : int
#m_SquareSize : int
-m_square : Rectangle[][]
+paintComponent(gc : Graphics) : void
+GridViewer(cellPixSize : IntC, gridDim : IntC)
RegularAutomatonViewer
#DrawSystemState() : void
+GetCellColorNumXY(x : int, y : int) : int
#IsStableXY(x : int, y : int) : boolean
OneRegisterAutomatonViewer
+GetCellColorNumXY(x : int, y : int) : int
+IsStableXY(x : int, y : int) : boolean
MultiRegisterAutomatonViewer
+GetStateAXY(x : int, y : int) : int
+GetStateBXY(x : int, y : int) : int
+SetStateAXY(x : int, y : int, state : int) : void
+SetStateBXY(x : int, y : int, state : int) : void
OneRegisterIntCellularAutomataModel
MultiRegisterIntCellularAutomataModel
<<use>>
<<use>>

Figure 1.2: Class diagram : Viewers hierarchy (simplified)

The `Viewers` objects manage the table of cells. This table contains all the cells of the simulation with their values and methods. It also manage the grid view of the simulations. Like the size of the cells, the distribution of the colors, the border of the cells or theirs specials effect visualization. There are four hierarchical levels presented here from the higher to the specifier.

1.2.1 AutomatonViewer

The abstract class `AutomatonViewer` contains the most general methods about the display of the grid. It extends `JComponent`[1] it is the most elevated hierarchical level. It associates a palette to each viewer.

The abstract method `GetXYWindowsSize()` needs to be defined by inheritance. This method must return a couple of int which contains the size of the window's panel in pixels in the form of a `IntC`.

1.2.2 GridViewer

This abstract class defines the methods which display the components of the grid in the `paintComponent()`, which cannot be overwritten. The class also contains the variables about the grid:

m_XYsize: the grid size (in number of cells).

m_CellDist: an integer for the distance between two cells.

m_SquareSize: an integer which represents the size of the cells.

[1]`https://docs.oracle.com/javase/7/docs/api/javax/swing/JComponent.html`

m_square: a two-dimensional table of java's Rectangle class[2] which will be used to draw the grid.

The constructor of the class requires two parameters:

- A couple of int which contains the size of the cells, in pixels, in an `IntC`.
- The size of the grid in number of cells. An `IntC` or a `GridSystem` can be used.

The abstract method `DrawSystemState()` must be defined in an inherited class in order to display the type of grid (the space of the automaton) and other objects into the simulation (for example the direction of the cell or the stability point).

1.2.3 RegularAutomationViewer

Last abstract class of the hierarchy which implements the usage of the colours, a display of a grid system and the stability point in overriding the method `DrawSystemState()`. This class is used to be inherited by the regular cellular automata viewers which don't contain any particular rules. There is two abstract methods to define here: `GetCellColorNumXY()`, that must return an int for colour id within parameters two int which are the X and Y position of the cell. `IsStableXY()` must return a boolean: true if the state's cell is stable and will not be changed the next time step and false if the cell can be changed. With parameters two int X and Y again for position.

1.2.4 OneRegisterAutomatonViewer

This class is used for the cellular automata of type `OneRegisterIntCellularAutomataModel`. It implements the MouseInteractive interface in order to update directly some cell by the viewer. `GetColorNumXY()` and `IsStableXY()` are defined.

1.2.5 MultiRegisterAutomatonViewer

This abstract class is used for the cellular automata of type `MultiRegisterIntCellularAutomataModel`. It implements methods for the `MultiRegisterCell`'s manage like `GetCell()`, `SetState()` A and B and `GetStateXY()` for the first and the second int (respectively named A and B).

1.3 Initializer

[2]`https://docs.oracle.com/javase/8/docs/api/java/awt/Rectangle.html`

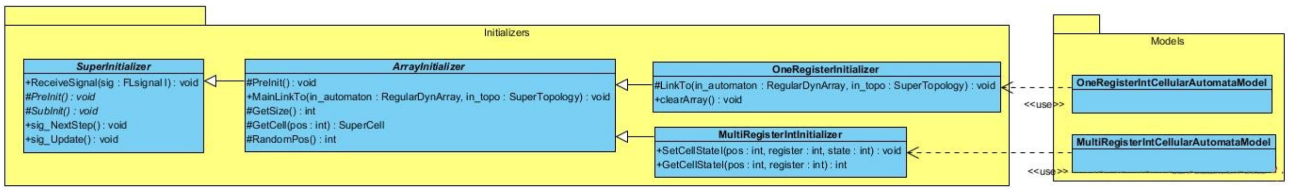
Initializers
SuperInitializer
+ReceiveSignal(sig : FLsignal I) : void
#PreInit() : void
#SubInit() : void
+sig_NextStep() : void
+sig_Update() : void
ArrayInitializer
#PreInit() : void
+MainLinkTo(in_automaton : RegularDynArray, in_topo : SuperTopology) : void
#GetSize() : int
#GetCell(pos : int) : SuperCell
#RandomPos() : int
OneRegisterInitializer
#LinkTo(in_automaton : RegularDynArray, in_topo : SuperTopology) : void
+clearArray() : void
MultiRegisterIntInitializer
+SetCellStateI(pos : int, register : int, state : int) : void
+GetCellStateI(pos : int, register : int) : int
Models
OneRegisterIntCellularAutomataModel
MultiRegisterIntCellularAutomataModel
<<use>>
<<use>>

Figure 1.3: Class diagram : Initializers hierarchy (simplified)

1.3.1 SuperInitializer

SuperInitializer is the most elevated level of the initializers. It contains a signal managers' method ReceiveSignal(). Which essentially call the abstract function that need to be defined in the inherited class SubInit() and PreInit ()called in this order.

This methods are called in the order PreInit() and after SubInit(). The first one is used to prepare what you require before the initiation. The second one must contain the code of the proper initiation.

If needed the functions sig_NextStep() and sigUpdate() can be overridden.

1.3.2 ArrayInitializer

ArrayInitializer is an abstract class than define PreInit() and verify if an error has occurred during the initiation. But most importantly, it defines the method MainLinkTo() which will link these two parameters: a RegularDynArray which represents the core of a cellular automaton. Literally only a one-dimensional table containing cells (link ref RegularDynArray to-do) and a topology (inherited from SuperTopology). With that function a table of cells and the strategy to read it will be linked.

This class also contains an abstract method to manage the array like GetSize(), GetCell() and RandomPos() which return, in order, the size of the table, the cell at the position given (in an int parameter) and an arbitrary position in the table.

1.3.3 OneRegisterInitializer

This abstract class is used for the cellular automata of type OneRegisterIntCellularAutomataModel. It defined the method LinkTo() and provides method to use the cell table like ClearArray(). It also implemented Get and Set method specific to 2D automata.

1.3.4 MultiRegisterIntInitializer

This abstract class is used for the cellular automata of type MultiRegisterIntCellularAutomataModel. It doesn't define the method LinkTo(), so the method must be overridden if need to be used. The class implemented Get and Set method for the position I (an int parameter).

Part III

Annexes

List of Figures