



## Tutorial

Fables implementation of  
Game of Life

**Prepared by:**

Viktor Erdélyi, Márton Dávid Iványi, and  
Richárd Olivér Legéndi



**September 2008, Budapest**

# 1 Contents

1	Contents	2
2	Installation	3
2.1	System Requirements	3
3	Introduction	3
3.1	John H. Conway's Game of Life	3
4	Creating a new Fables Project	4
5	Working with the IME	6
5.1	The structure of the model	6
5.2	Compiling the Fables source	9
6	The Charting Wizard	11
6.1	Creating a Grid2D visualization	11
7	Running the Application	14
8	Document Generation	17
9	Multiple running and compiling	20
10	Conclusion	20
11	Appendix: Pure Code	21

## 2 Installation

To install the Fables IME run Fables\_Installer.exe. The installer guides you through the installation process.

### 2.1 System Requirements

#### 2.1.1 Hardware

Intel x86/x64 and 100% compatible processors are supported. A Pentium III 700 MHz or faster processor with at least 128 MB of physical RAM, and 200MB free disk space is recommended as a minimum.

#### 2.1.2 Operating Systems

Currently the following operating systems are supported:

- Linux
- Macintosh / MacOS X (any version that has Carbon Application Programming Interface support)
- Windows 2000 (SP3+)
- Windows XP Home (SP1+)
- Windows XP Professional (SP1+)
- Windows Server 2003 R2
- Windows Vista

## 3 Introduction

This tutorial will guide you through the use of *Fables* and will give you an insight into the language and its basics. With the help of this tutorial you will be able to create a basic simulation, and you will become familiar with the Integrated Modeling Environment. We will create a model for John H. Conway's *Game of Life*.

The *Functional Agent-Based Language for Simulations* (Fables) is a simple, easy-to-use programming language aimed at creating models for agent-based simulations. It has a modeling environment, specially intended for creating, scheduling and observing simulations. It requires minimal programming skills, as its formalism is similar to the mathematical formalism used in publications in the subject.

The *Integrated Modeling Environment* (IME) is based on the Eclipse platform. To use it you simply have to run the installation file provided. The IME has numerous features making coding in Fables easier, like the language assistant, syntax highlighting, context sensitive help, automatic document generation and such.

If you are familiar to programming you might note that Fables is multi-paradigm language. It merges the aspects of object-oriented, functional and procedural languages.

### 3.1 John H. Conway's Game of Life

The Game of Life is a cellular automaton devised by the British mathematician John Horton Conway in 1970. It is the best-known example of a cellular automaton. The model is actually a zero-player game, meaning that its evolution is determined by its initial state. There is no need for input from human players during the run, hence the agent-based model architecture is a perfect fit for it.

#### 3.1.1 Rules

The world of the Game of Life is an infinite two-dimensional orthogonal grid of square cells, each of which is in one of two possible states; living or dead. Every cell

interacts with all its eight neighbors, which are the cells that are directly horizontally, vertically, or diagonally connected to it. At each step in time, the following transitions occur:

1. Any living cell with fewer than two living neighbors dies, as if by loneliness.
2. Any living cell with more than three living neighbors dies, as if by overcrowding.
3. Any living cell with two or three living neighbors lives, unchanged, to the next generation.
4. Any dead cell with exactly three living neighbors comes to life.

The initial pattern constitutes the first generation of the system. The second generation is created by applying the above rules simultaneously to every cell in the first generation - births and deaths happen simultaneously, in ticks. The rules continue to be applied repeatedly to create further generations.

## 4 Creating a new Fables Project

To create a new *Fables Project*, you can either press CTRL + N, or you can reach the same function through the File → New → Project menu. The following screen will appear on the screen indicating you're to create a new project:

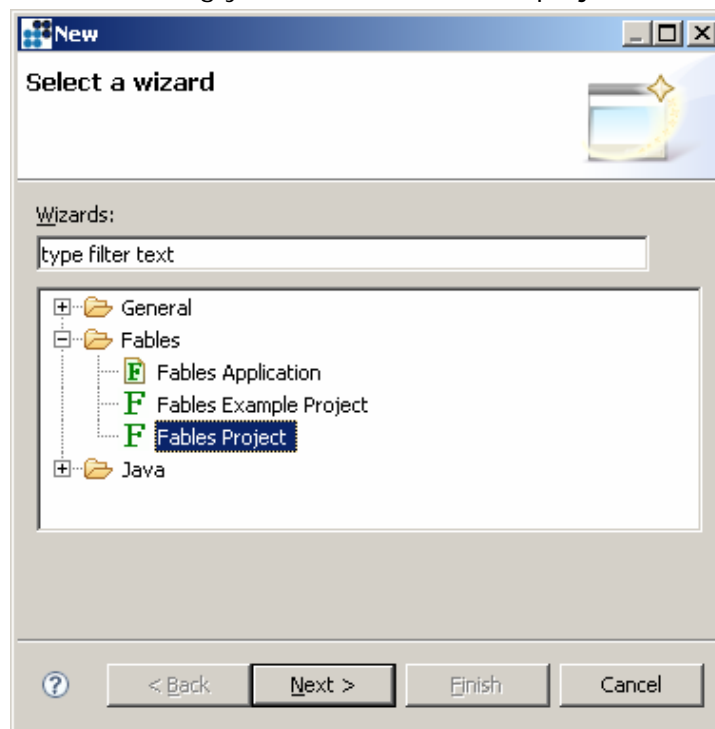


Figure 1

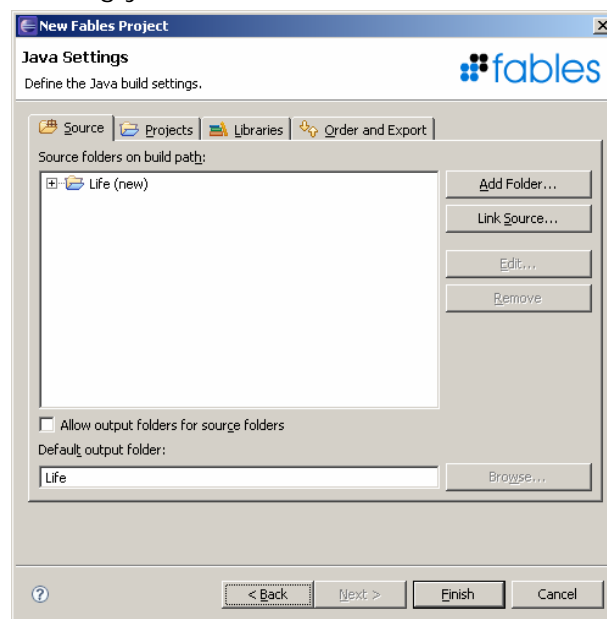
From the roll-down menu select *Fables Project*. A Fables project is a normal Java Eclipse project, extended to handle Fables files and capable to run Repast simulations.

After selecting to create a Fables Project and pressing *Next*, the following dialog will appear where you can name the project:

**Figure 2**

In the "Project name" field (Figure 2) type „Life“. The name of the project helps with distinguishing it from other projects; it has no effect on your model.

You can simply press *ENTER* or the *Finish* button to proceed and begin editing your model, or you can click on the *Next* button to set the Java settings (Figure 3), like other libraries, CLASSPATH settings and such. If you aren't familiar with Java, just press *Finish* to begin editing your model.

**Figure 3**

## 5 Working with the IME

Well done, you have just created your first blank Fables model. As you can notice on the left of the screen, your model has been added to the IME workspace, and it already contains some folders, namely the following:

- **Charts:** this directory will contain the charts created for your model.
- **Documents:** the documents generated from your model will be placed in here.
- **Repast:** the executable Repast/Java code will be placed here.
- **Lib:** place additional libraries under this directory.  
(required only to PET simulations)
- **Resources:** place additional resources under this directory  
(required only to PET simulations)

One more new item can be recognized in the left column, a file named `Life.fab`. The `.fab` files are the Fables files, in which you can create your models. At the moment it is almost empty, it contains only a minimal model definition that sets the default random generator's seed value to 1:

```
model Life {
    startUp {
        seed(1);
    }
}
```

On the bottom of the screen you can see the *Fables Console*, the field where the compiler informs you about the results of the compilations (like warnings and errors).

On the right of the screen you can see the *Outline* window which presents the structure of your model in a tree view.

Sometimes it comes handy to enlarge the editor window to full screen mode within the IME. To do this, double-click on the `Life.fab` editor tab atop the editor, or press CTRL + M within the editor window.

### 5.1 The structure of the model

To set up the model first we define some constants that indicate whether a cell in the model world (where the agents interact) is empty or not:

```
dead = 0 ;
live = 1 ;
```

To define the size of the model world we use the `worldSize` constant according to the following definition:

```
worldSize = 50 ;
```

There is a function we will use to make sure everything is displayed in a discrete interval and to make the borders permeable, the normalizing function `norm`.

It is really expedient to write comments in your code because it will be more understandable both for you and for anybody else who reads your code. The documents automatically generated from the code will also use these comments. See more on this in section 8 Document Generation. Commenting can be via placing `/**` before and `*/` after the text to be commented.

The comment and the declaration for the normalization is the following:

```
/** To normalize x depending on worldSize. */
norm (x) = x mod worldSize ;
```

The `mod` operator used above is a standard Fables operator for determining modulo. It returns `x`, if it is in the interval `[0..worldSize]` (i.e. `norm(7)` will be 7), but it transforms every other value to fit into the interval `[0..worldSize]`. For instance, `norm(55)` returns 5, `norm(-1)` returns 49, etc. You can find a more detailed description in the Fables Language Manual.

The model world will be represented by a matrix that is dynamically changing hence we define it as a variable (see below). It is dynamically changing as each turn in the simulation creates a new state of world.

```
var world;
```

By declaring the world as a variable, declaring world size, introducing the norm function, and defining empty and used cells you have the created the base structure of the model.

To initialize the model you implement the `startUp()` method where we initialize the `world` variable and set the random seed to zero for now.

```
startUp(worldSize) {
  seed(0);
  world := [ [ discreteUniform( dead, live ) ]
            : x is [0..worldSize-1]
            : y is [0..worldSize-1] ] ;
}
```

The code above indicates that the `worldSize` is a parameter of the model (model parameters have to be put into the brackets after the keyword `startUp`). That generally means it can be directly set during the runs. Since `worldSize` is used for setting the size of the matrix (`world`), generally it's 50x50. We initialize our world randomly depending on discrete uniform, and the initialized seed value.

According to the Game of Life rules the neighbors of all cells have to be counted to determine their states in the next generation. To do this the connected (horizontally, vertically and diagonally) cells have to be iterated. To determine the number of neighbors of a given cell  $(x, y)$  we define the following function:

```
numberOfNeighbours(x, y) = size ( [ 1 :
  dx is [-1..1], dy is [-1..1]
  when not (dx == 0 == dy)
  and ( world(norm(x+dx))(norm(y+dy)) == live )
] ) ;
```

At first that could be hard to understand, but this example shows how to declare a sequence and its conditions and iterations.

Basically this code will return the size of a given sequence. The sequence consists of exactly as many '1's as living neighboring cells  $(x,y)$  has in the following matrix:

$(x-1, y-1)$	$(x-1, y)$	$(x-1, y+1)$
$(x, y-1)$	$(x, y)$	$(x, y+1)$
$(x+1, y-1)$	$(x+1, y)$	$(x+1, y)$

The iteration based on `x` is done by `dx`, while by `y` is done by `dy`. The condition `not (dx == 0 == dy)` takes  $(x,y)$  out of consideration as it can't be its own neighbor.

We need to implement only one more method, the evolution rule. The simple „23/3“ rule (see 3.1.1) is used:

```
step(x,y) = ( n==3 or ( world(x)(y)==live and n==2 )
  => live
  otherwise
  => dead
  where (
    n = numberOfNeighbours(x,y)
  ) ;
```

The definition above is a partial function. It will get the `x` and `y` coordinates of a cell as a parameter, and will return `live` (1) if it fits the rules (if it has three neighbors or it was alive in the previous generation and has two neighbors), `dead` (0) otherwise. There is also a local constant definition after the `where` keyword. It creates a new local constant (`n`), which can be used only in this function definition, and contains the number of neighbours of the current cell.

We need another matrix that will contain the new world, since we have to store both for comparing them (counting the neighbors):

```
newWorld = [ [ step(x, y)
              : y is [0..worldSize-1] ]
             : x is [0..worldSize-1] ] ;
```

The new world is created with the previously defined `step` function. There is one last thing to add to our model, and that is the schedule that makes the evolution process cyclical, controls the turns:

```
schedule Main cyclic 1 {
  1 : world := newWorld ;
}
```

This schedule is named `Main` and is activated in every turn processing the `world := newWorld` assignment in the first turn after activation (`cyclic 1`).

If you did everything according to the instructions the model code in the Fables IME should look something like what you see on the figure below. If you had troubles with the implementation, you can find the full source code in the Appendix: Pure Code section.

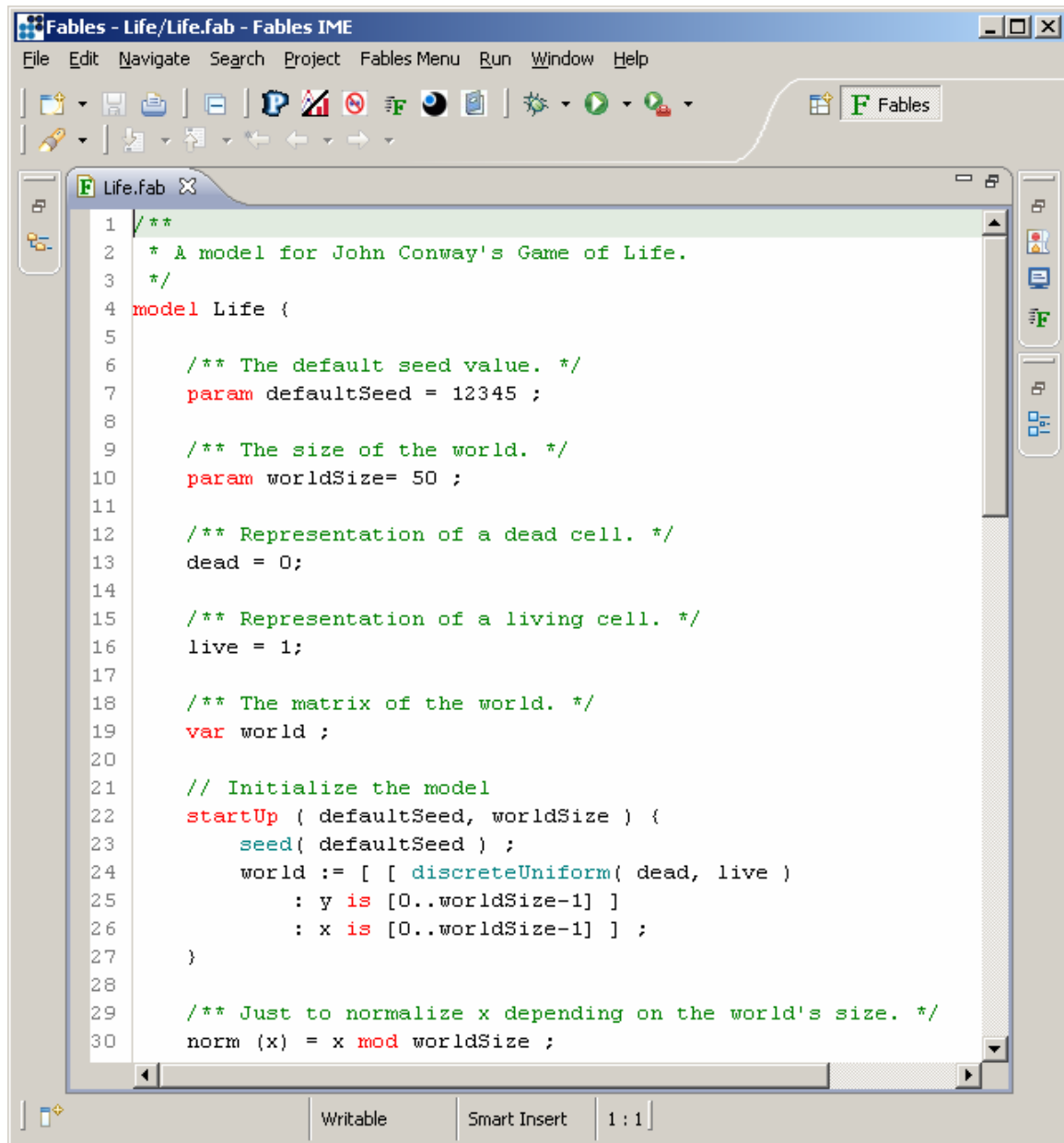


Figure 4

## 5.2 Compiling the Fables source

Now that the Fables program code is done the model can be compiled to Repast. Note that the whole point behind using Fables is that it does not require nearly as much programming as it would in Repast. Compiling the Fables code to Repast is totally automatic and it can be done in several ways in the IME. One way is to right-click with your mouse on the `Life.fab` and select *Run As* → *Fables application* in the dialogue that appears (see Figure 5).

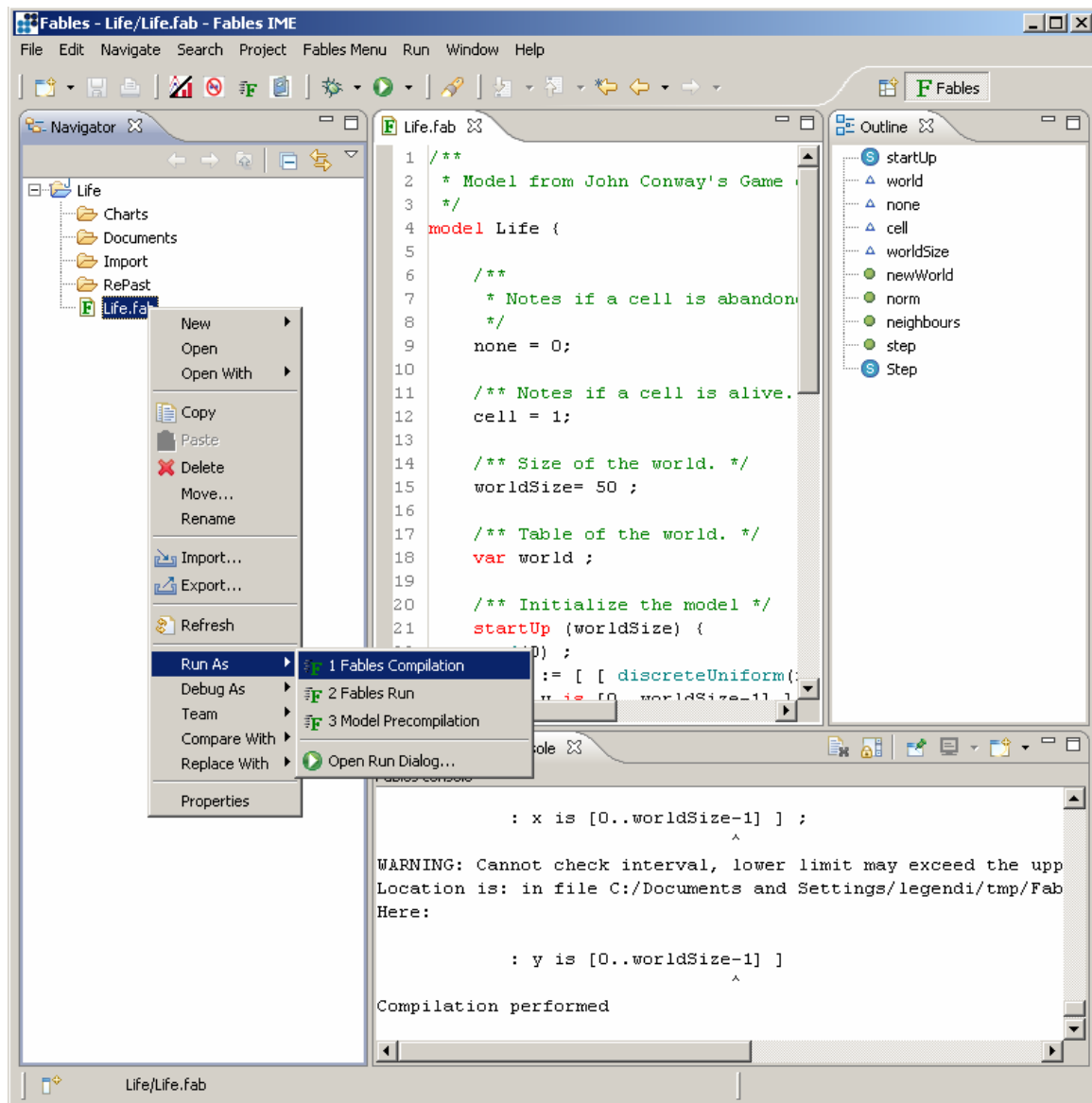


Figure 5

You are informed about the progress of the compilation through the Fables Console on the bottom of your screen. As a result of the compilation the Repast source code will be generated and stored in the RePast directory in the Life project. Your workspace is updated automatically when the compilation ends with the Java files generated.

If you haven't created any visualization before the compilation – if you have followed the instructions, you haven't - you will be asked to create one during the process. This is covered in the next section (6 The Charting Wizard).

**Note:** The compiler writes some warning messages onto the console. The reason is that the compiler tries to check every lower and upper bound of an interval definition, because if the lower bound exceeds the upper one, it leads to a runtime error (for instance, for the [3..1] interval). The compiler is able to perform this check, but for the [0..worldSize-1] interval it fails due the upper limit is a model parameter, that can be assigned to any value during runtime. That's the reason why the warning messages are there.

## 6 The Charting Wizard

Fables gathers all information available from your model and determines what kinds of charts could be generated from the variables, constants and functions declared in the model. Upon selecting any given chart type (6.1, Figure 6) you will be offered only the adequate variables for each type (Figure 7). If a chart type is selected that the model contains no adequate variables for the IME won't let you create it.

In our case a two-dimensional matrix is declared which is perfect to be displayed in a Grid2D chart.

### 6.1 Creating a Grid2D visualization

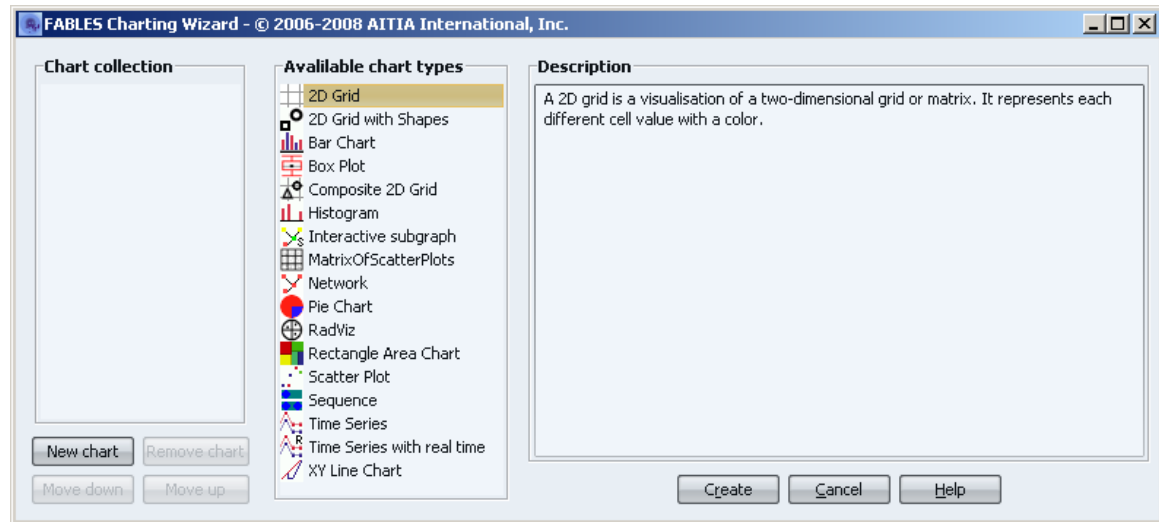


Figure 6

To create a Grid2D chart either double-click on it in the *Available chart types* menu or select it and click on the *Create* button (Figure 6). Upon doing so the wizard proceeds to the window shown on Figure 7 below. This is where the characteristics of the given chart can be specified.

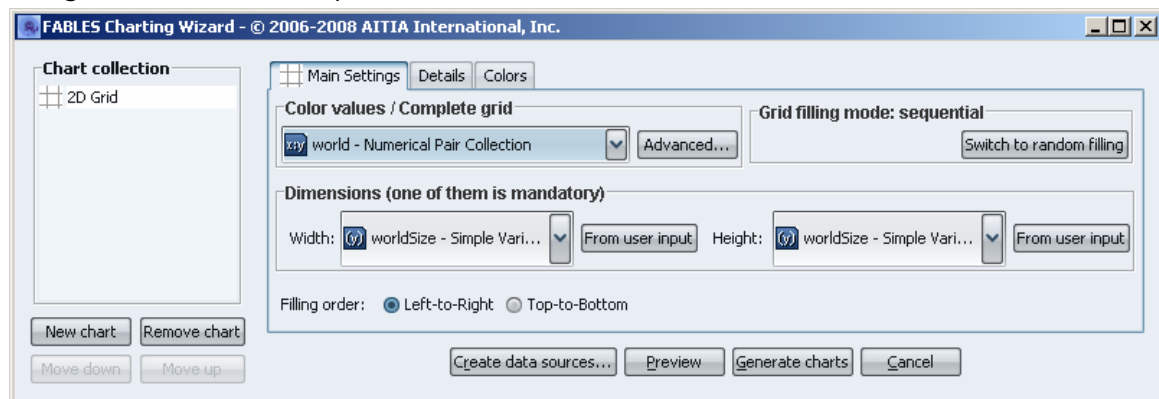
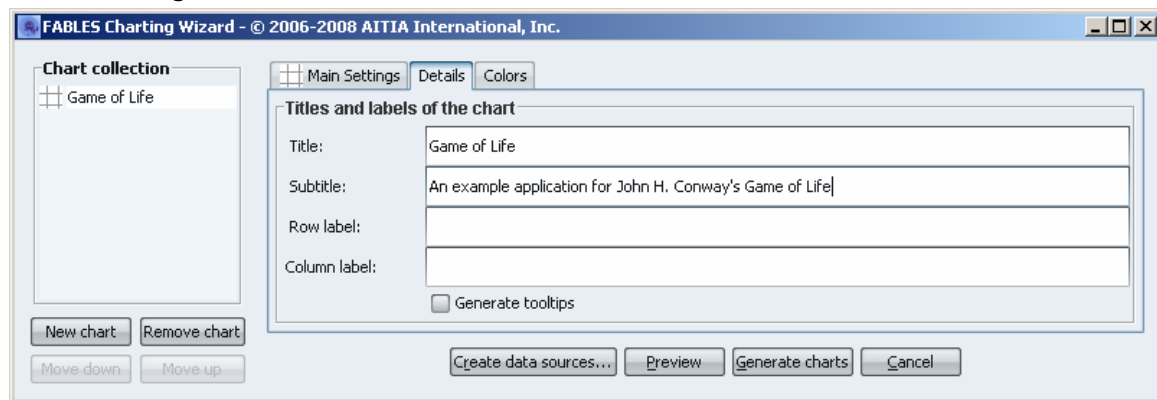


Figure 7

*On the Main Settings tab* a data source – a variable providing data for the chart - has to be selected first. In this case select `world` and then set the *width* and *height* values. These values determine the dimensions of the chart. They can be set to a constant value: 50 would be an obvious choice, but please note that this constant value won't change if the model parameter `worldSize` changes, and the chart has to be edited manually after each run. For these cases, the Charting Wizard is able to determine the width and size values of the model from other data sources. Using the simple variable `worldSize` for both width and height is a perfect choice.

**On the Details tab** the various labels and titles can be set for the chart as it is shown on Figure 8 below.



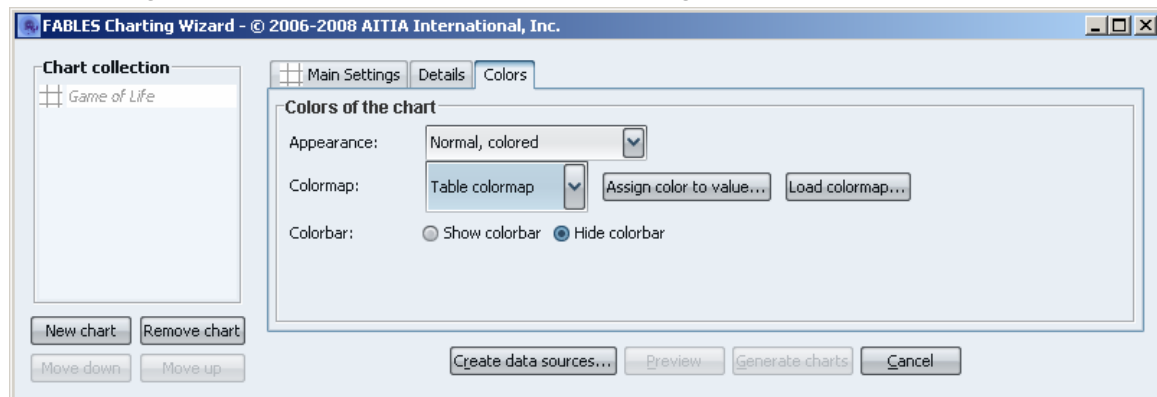
**Figure 8**

Set the following values:

- **Title:** Game of Life
- **Subtitle:** An example application for John H. Conway's Game of Life.

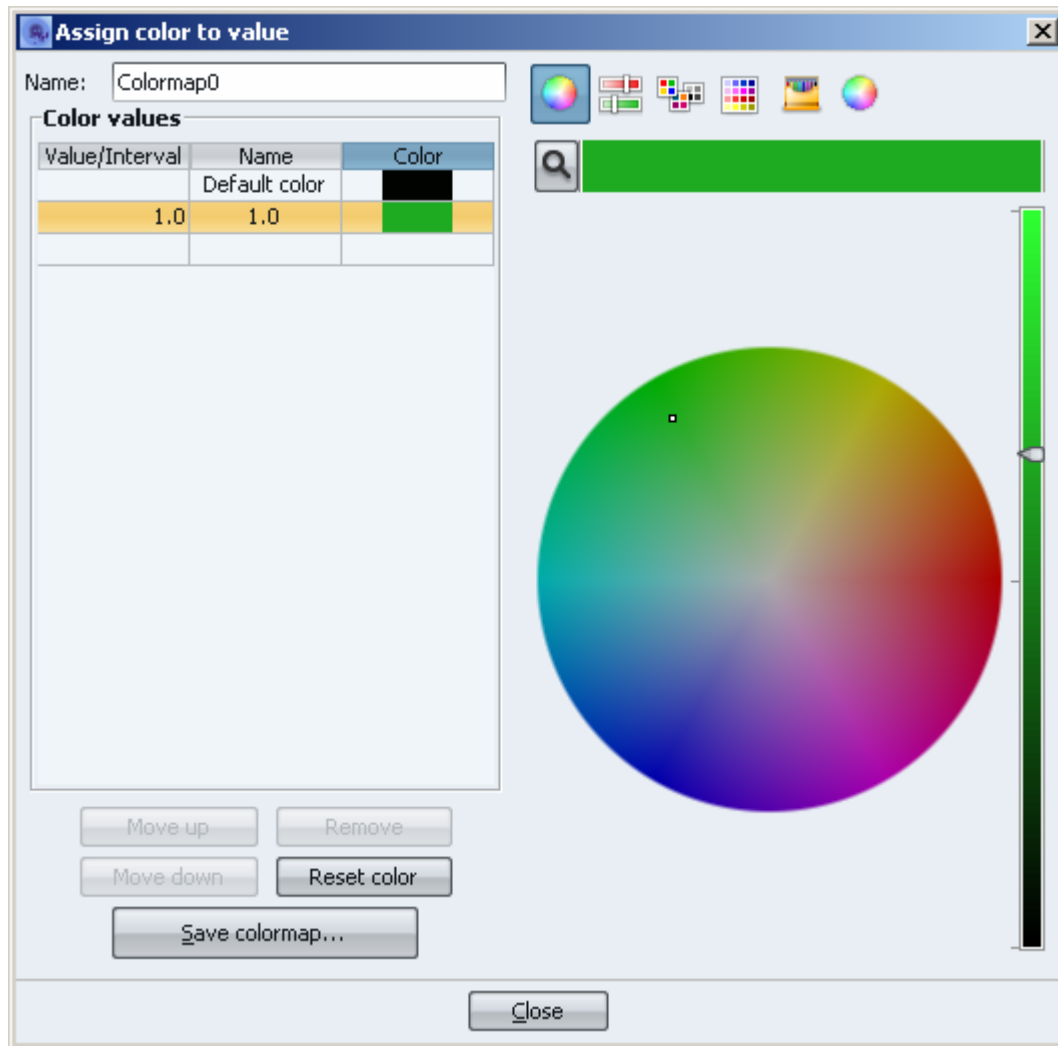
These texts will appear on the displayed chart as additional information.

**The third tab is the Colors tab** (Figure 9) which is really important in case of a Grid2D chart, setting the color values fills the chart with meaning. There are several available settings, use the *TableMap* option for now. This setting will enable you to define color-value pairs that are displayed on the grid. If you have defined colormaps previously you can use *Load colormap...* to invoke it but in our case you have to press *Assign color to value...* to set one up (see Figure 10).



**Figure 9**

The world variable holds two values; 0 for dead and 1 for living cells. Set the default value – by pressing *Define default color...* and selecting the color in the popup - to black (that will represent empty cells) and add value 1 to the color table and assign an arbitrary color value to it (Figure 10). Press *Close* when you're done to return to the wizard's main window.

**Figure 10**

Upon returning to the main window you can examine your settings in practice by pressing *Preview*. If you are satisfied press the *Generate Charts* button and the new chart will be generated and placed in the project's *Charts* folder.

If the compilation was interrupted to create the chart (see the end of section 5.2 Compiling the Fables source) it is suspended until the visualization is created. In this case it continues as soon as the chart is generated.

## 7 Running the Application

To run the by now compiled Fables application one of the generated Java sources has to be run. In simpler models like ours there are only four of them. The first one is the pure model ([Life.java](#), Figure 11), the second is the one providing the graphical interface ([LifeGUI.java](#)), and the third and fourth are the producer files are there for creating the data for the charts.

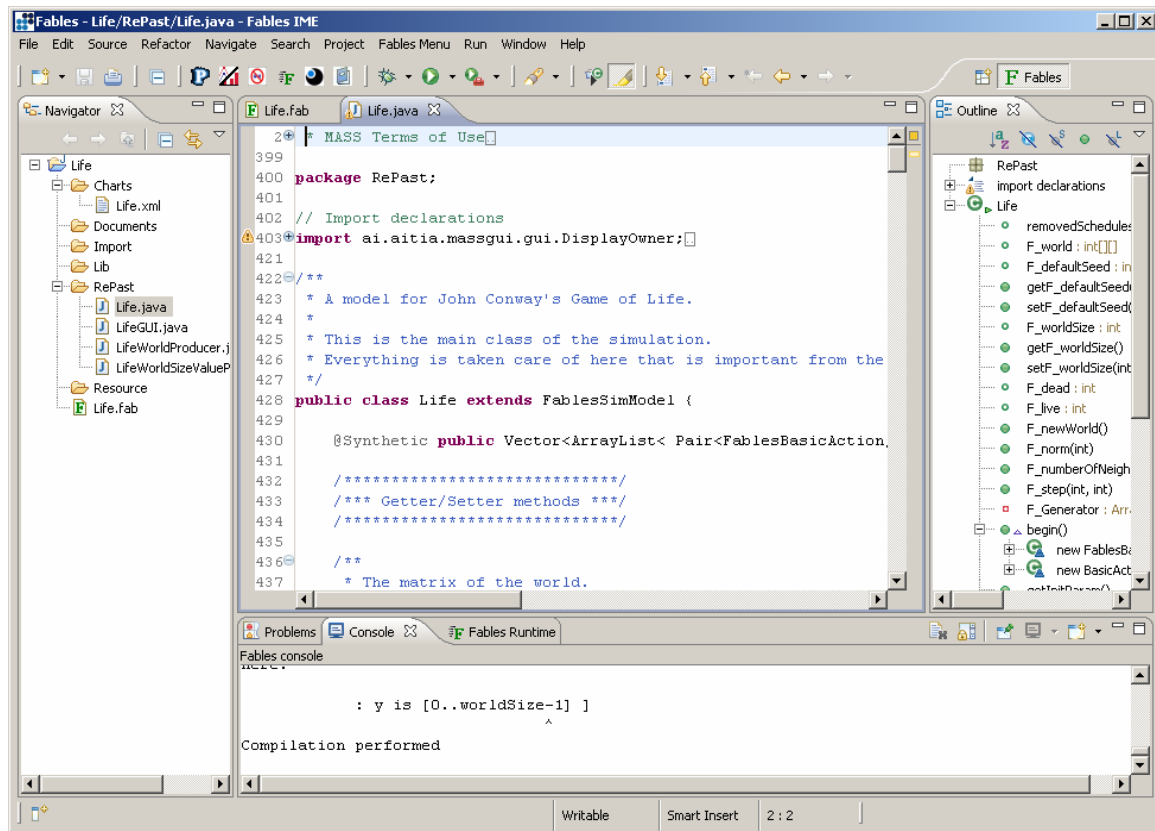


Figure 11

If you run the pure model there'll be no charts, the model code will be executed and results are only displayed in the run console if `print` or `println` function calls are placed in the code.

To run the graphical one simply right-click on the `LifeGUI.java` file and select *Run As* → *Java Application* (See Figure 12 below).

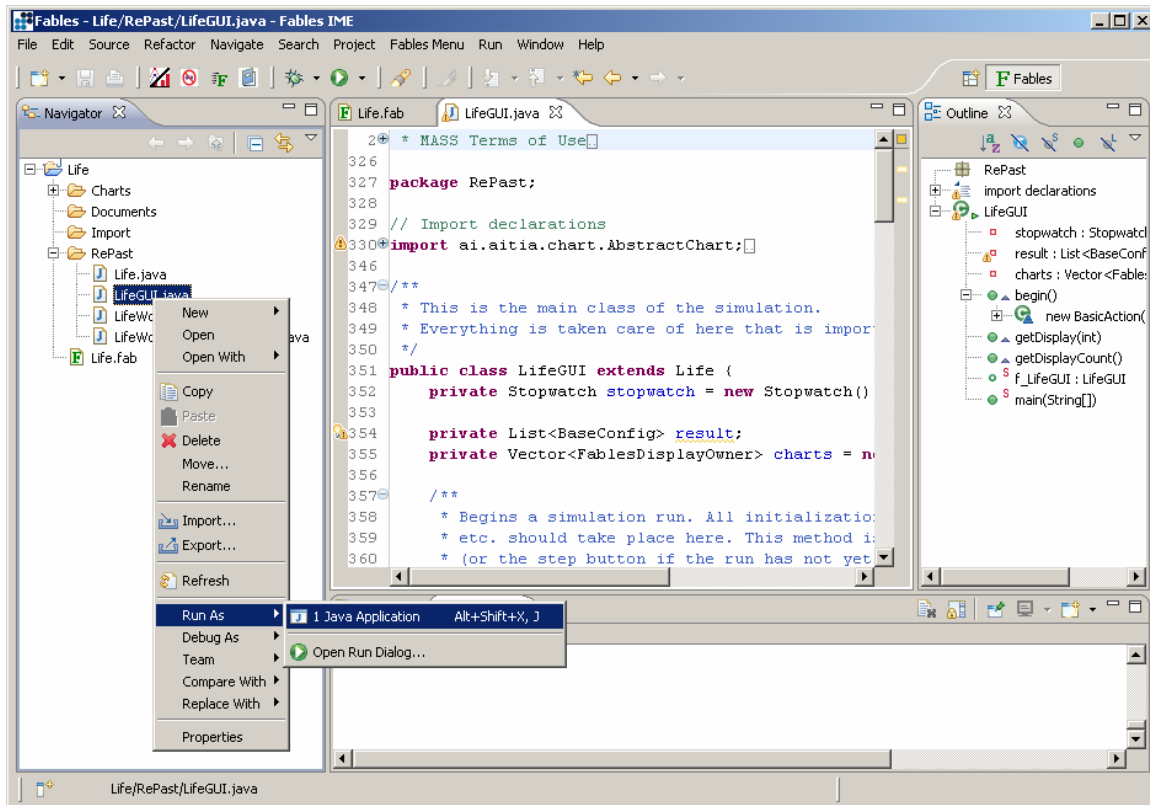


Figure 12

Upon selecting *Java Application* the Repast controls pop-up on the screen that enable initializing the model for the run by pressing *Initialze* (Figure 13).

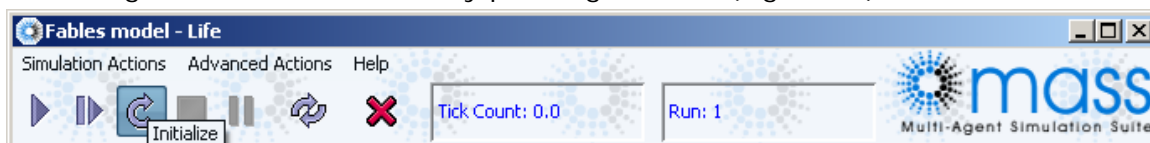
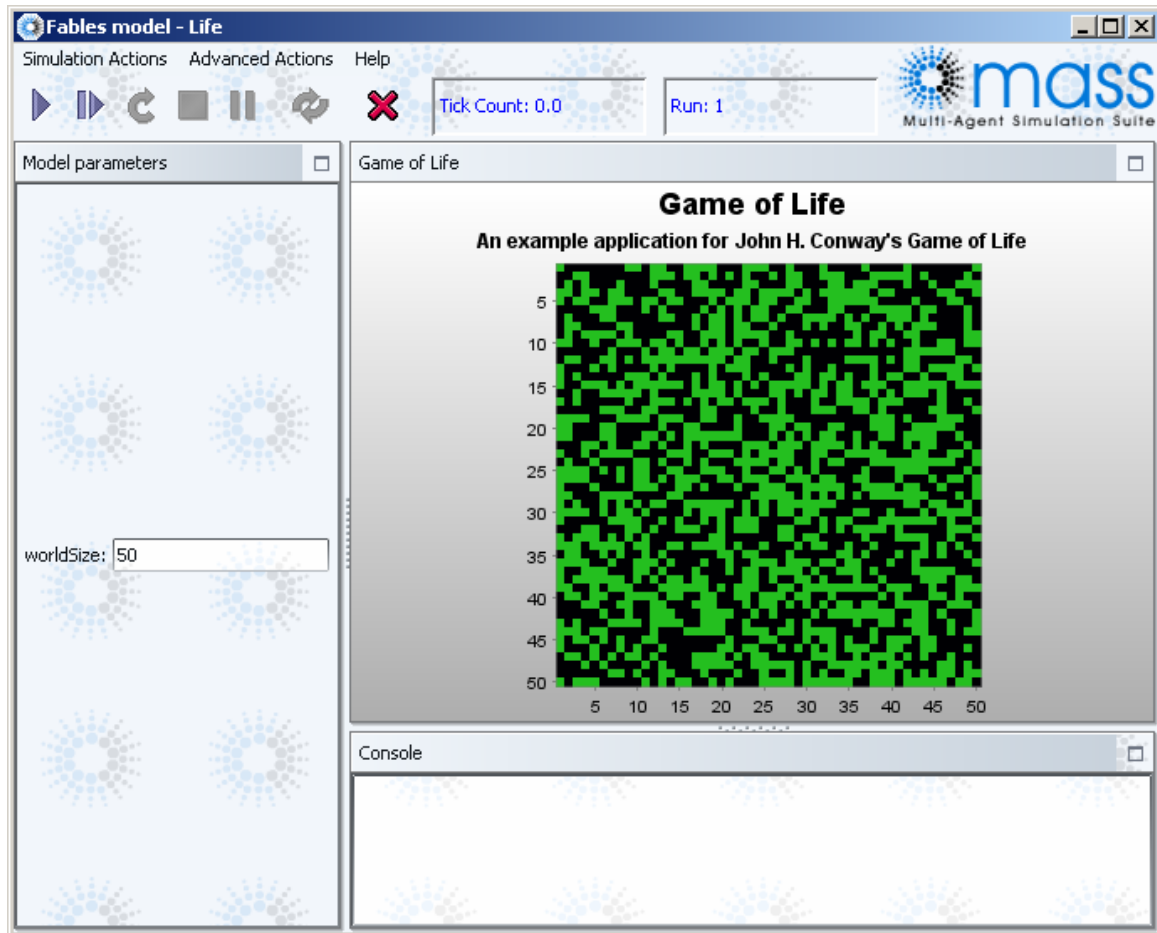


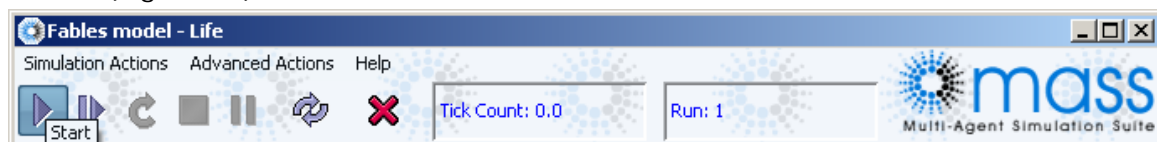
Figure 13

If everything went correctly a frame appears that is similar to the one displayed below on Figure 14.



**Figure 14**

This is a typical random position for the model. Note that title and subtitle texts set in the wizard are also displayed on the chart. To run your model press the *Start* button (Figure 15).



**Figure 15**

After several rounds of running the model will possibly get to a stable state looking something like Figure 16.

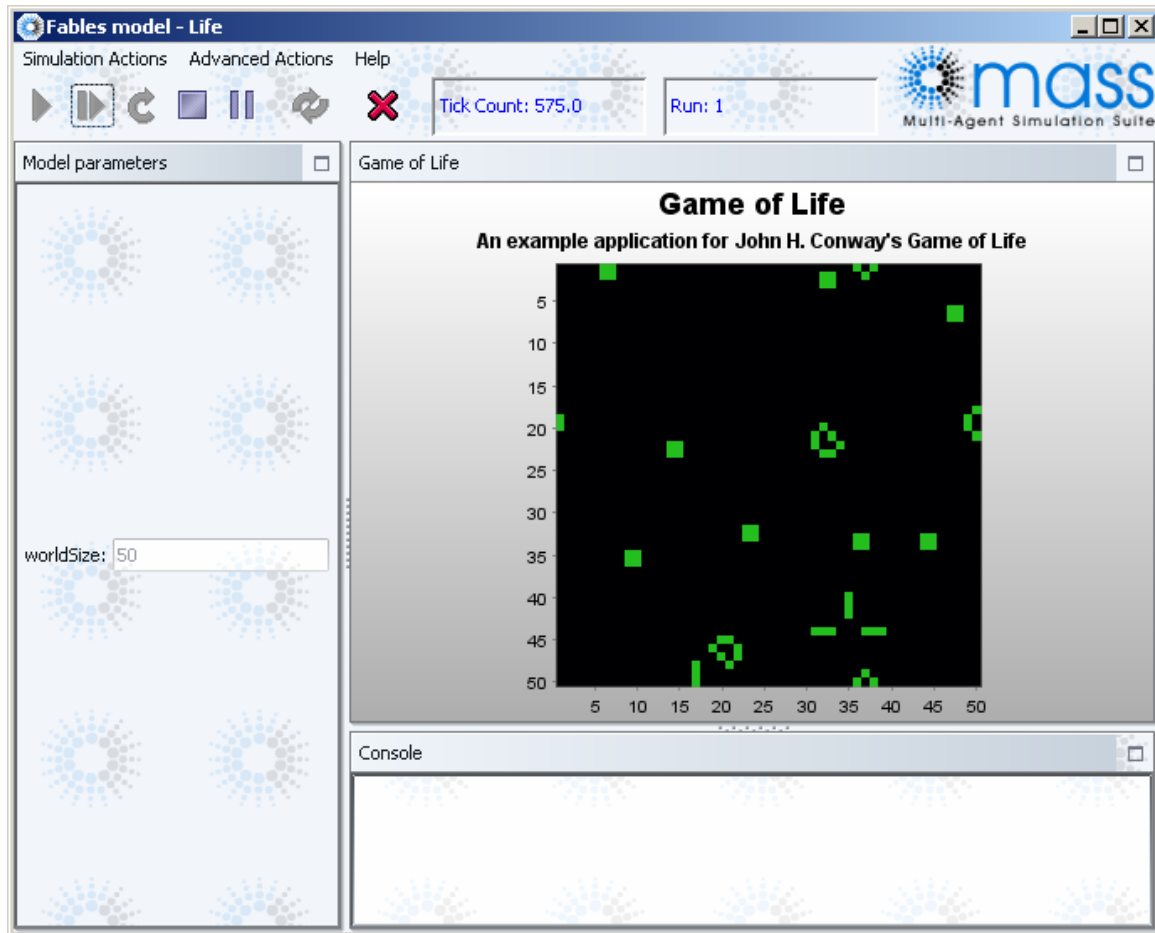


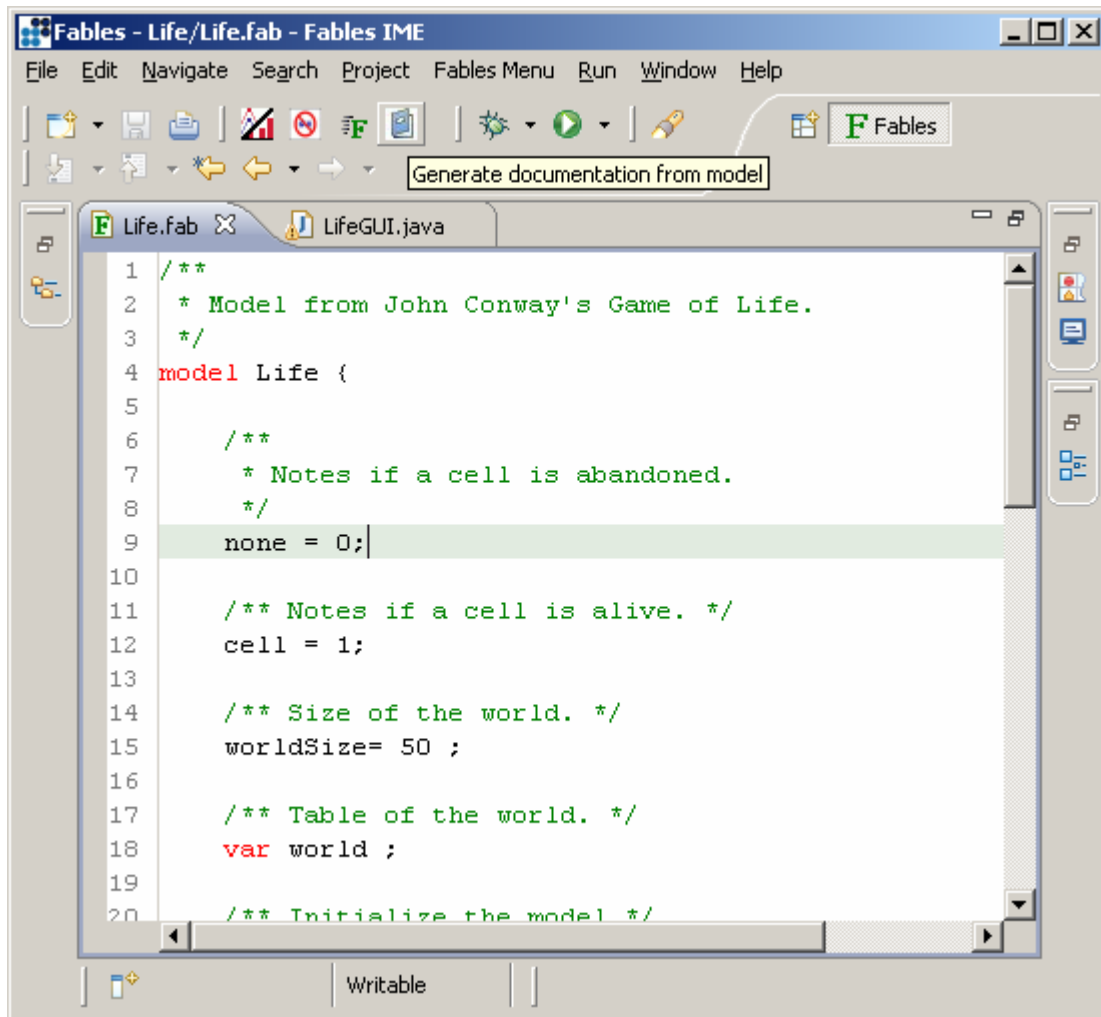
Figure 16

Most of the evolved states of the model are either stay-alive or cyclic. Stay-alive states are stable, the cells neither move, nor develop. Cyclic states are ones that reproduce in a few generations.

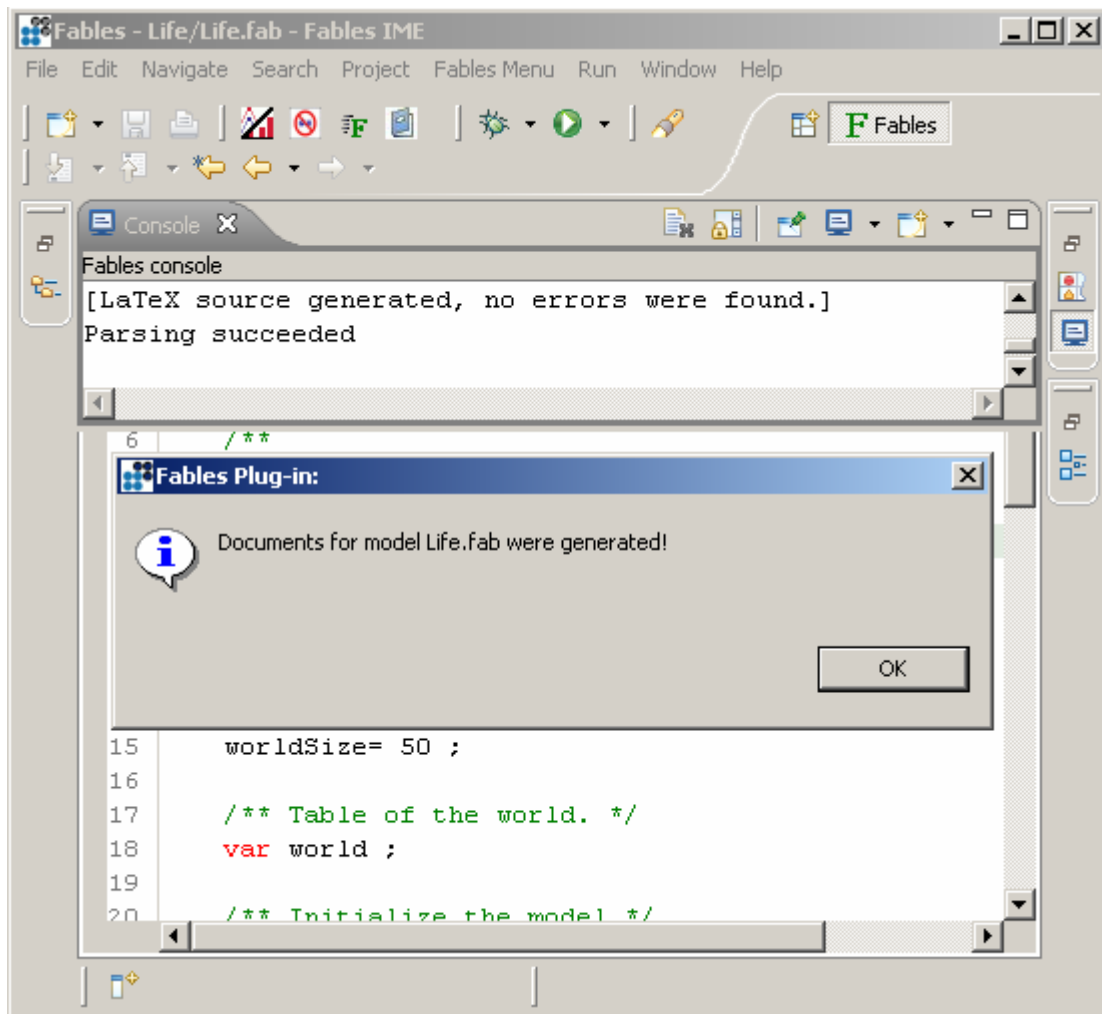
## 8 Document Generation

The Fables IME is able to generate documents from the models in a number of popular formats. Read more about the formats and how document generation works in the *Fables Manual*.

To generate a document double click (open) any of the Fables files you have (`Life.fab` is perfect) and press the *Generate Document* button either on the toolbar or on the menu bar at the top of the screen (Figure 17).

**Figure 17**

Note that the model has to be syntactically correct to be able to be processed. When the parsing is over you are notified by a dialog about the documents that were created in the project's Documents directory (Figure 18 below).

**Figure 18**

See the *HTML* description generated by Fables IME from the model Life for example.

## 9 Multiple running and compiling

Multiple running and compiling is when a model is run repeatedly. Normally Fables learns the launching configurations for each model when they are first compiled, and after that running the model again is done without detecting the launching configurations again.

For Fables and Repast that would interfere with setting variables before and in the run. Hence we have created a menu bar button "*Run Fables code*" (Figure 19) that learns the launching configuration, compiles and runs the models upon pressing it, and does so every time it is pressed. Note that a Fables file has to be active in the editor in order to do this.

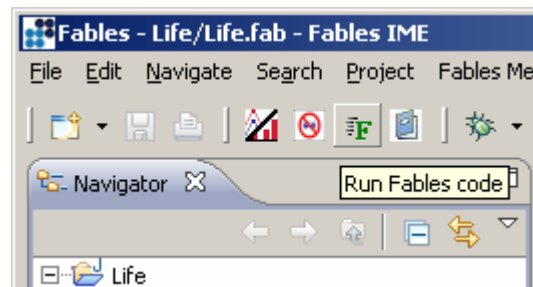


Figure 19

## 10 Conclusion

If you followed the instructions in this tutorial you have created your first simulation in Fables. It could hardly be stated it was too hard or involved too much programming. Of course creating more complex models is not this easy but that is due to the mathematical complexity of the given problems. Fables is intended to help you in every aspect of creating any model that you can describe with mathematical formulas and agent-based simulation can be applied to.

Upon installing Fables you will probably notice that the package already contains some example models in it. Studying these models is a great way to extend your knowledge and understand the possible uses and advantages of the language and the environment.

## 11 Appendix: Pure Code

If you have any problems with the code that is needed for this model here it is without any interruptions and comments.

```

/**
 * A model for John Conway's Game of Life.
 */
model Life {

  /** The size of the world. */
  worldSize= 50 ; // PARAMETER

  /**
   * Notes if a cell is abandoned.
   */
  dead = 0;

  /** Notes if a cell is alive. */
  live = 1;

  /** The table of the world. */
  var world ;

  // Initialize the model
  startUp (worldSize) {
    seed(0) ;
    world := [ [ discreteUniform(dead,live)
                : y is [0..worldSize-1] ]
              : x is [0..worldSize-1] ] ;
  }

  /** Just to normalize x depending on the world's size. */
  norm (x) = x mod worldSize ;

  /** Determines the neighbourship relation. */
  numberOfNeighbours(x, y) = size ( [ 1 :
    dx is [-1..1], dy is [-1..1]
    when not (dx==dy==0)
    and world(norm(x+dx))(norm(y+dy)) == live ] ) ;

  /** Determines the rule, now it is just the simple "23/3". */
  step(x,y) =
    (n==3 or ( world(x)(y)==live and n==2 )
     => live
    otherwise
     => dead
    where (
      n = numberOfNeighbours(x,y)
    ) ;

  /**
   * Creation of the next generation.
   */
  newWorld = [ [ step(x, y)
                : y is [0..worldSize-1] ]
              : x is [0..worldSize-1] ] ;

  /** Will perform the generation change. */
  schedule Generator cyclic 1 {
    1 : world := newWorld ;
  }

}; // model{ Life }

```