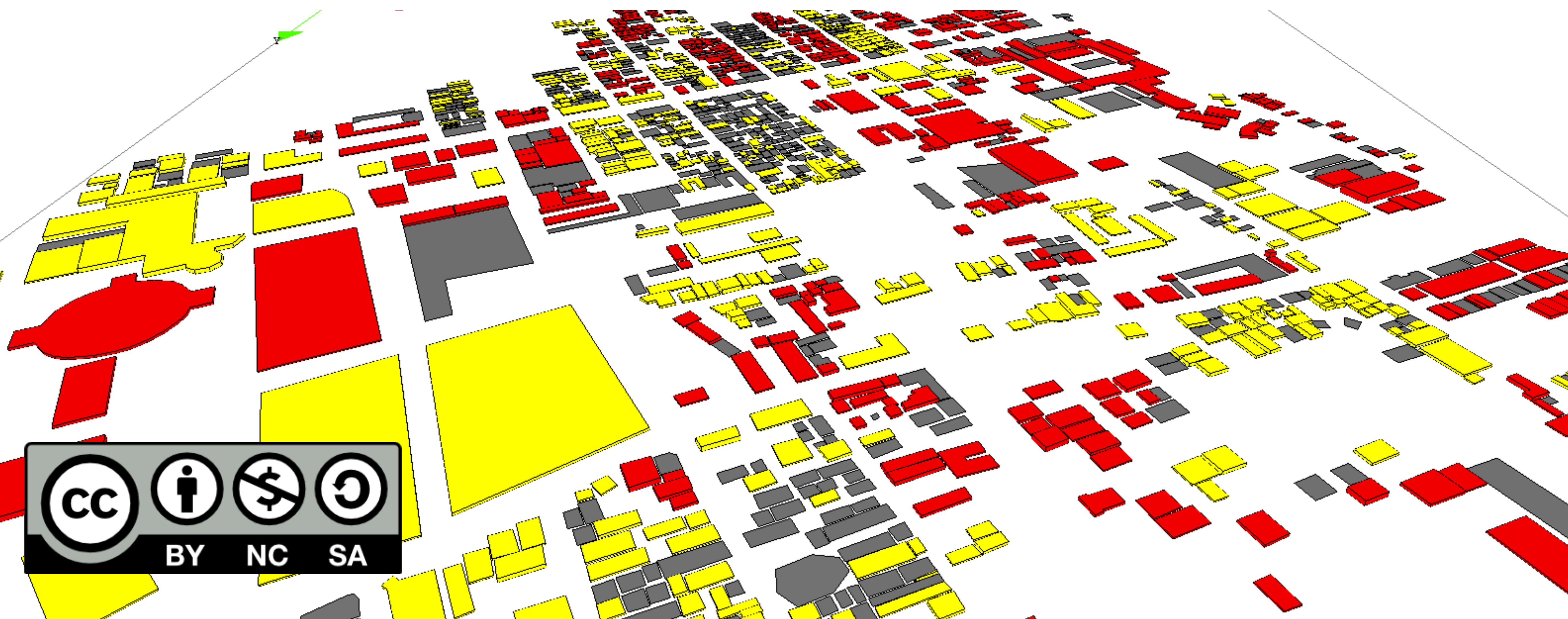
A Practical introduction to GAMA Through a Segregation model

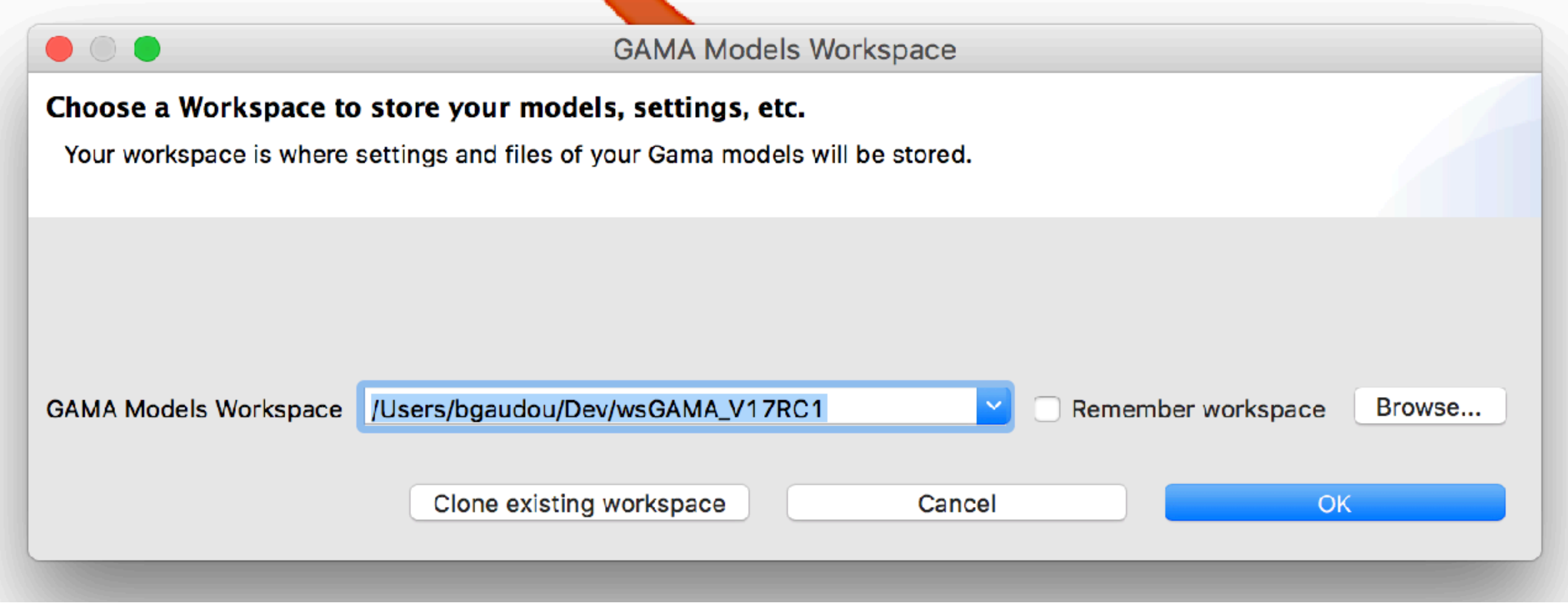
Benoit GAUDOU, *IRD UMMISCO, University Toulouse 1 Capitole, USTH; benoit.gaudou@gmail.com*



Introduction to the ***use*** of the Gama Platform

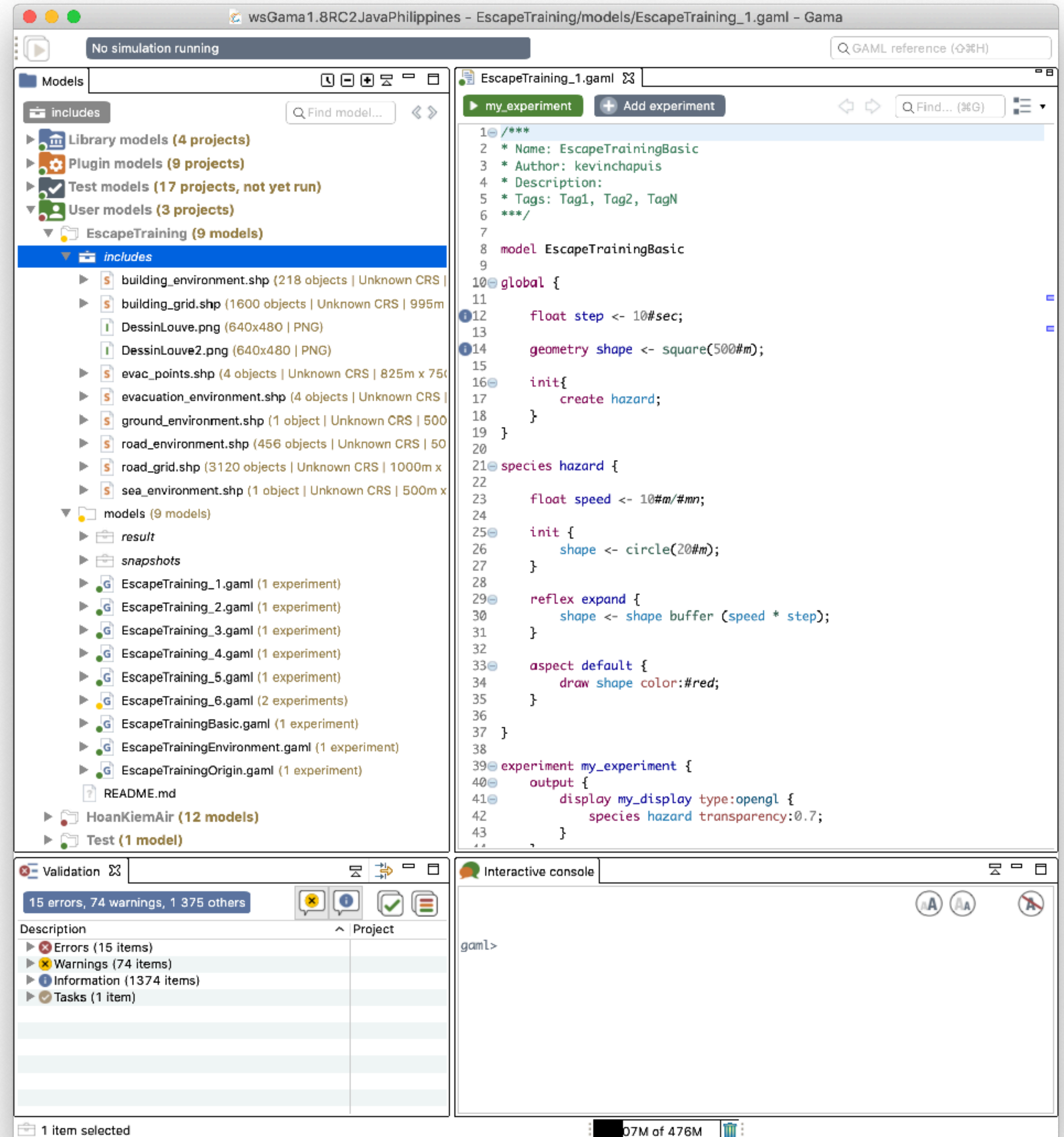
It is now time to run GAMA !

- First GAMA asks you to choose a **workspace**.
- A workspace is a folder that will contain all your own projects and models.
- You are free to choose the folder you want!



GAMA model files are stored in projects

- Each **project** may contain several **models**, as well as additional **resources** (GIS data, pictures,…).

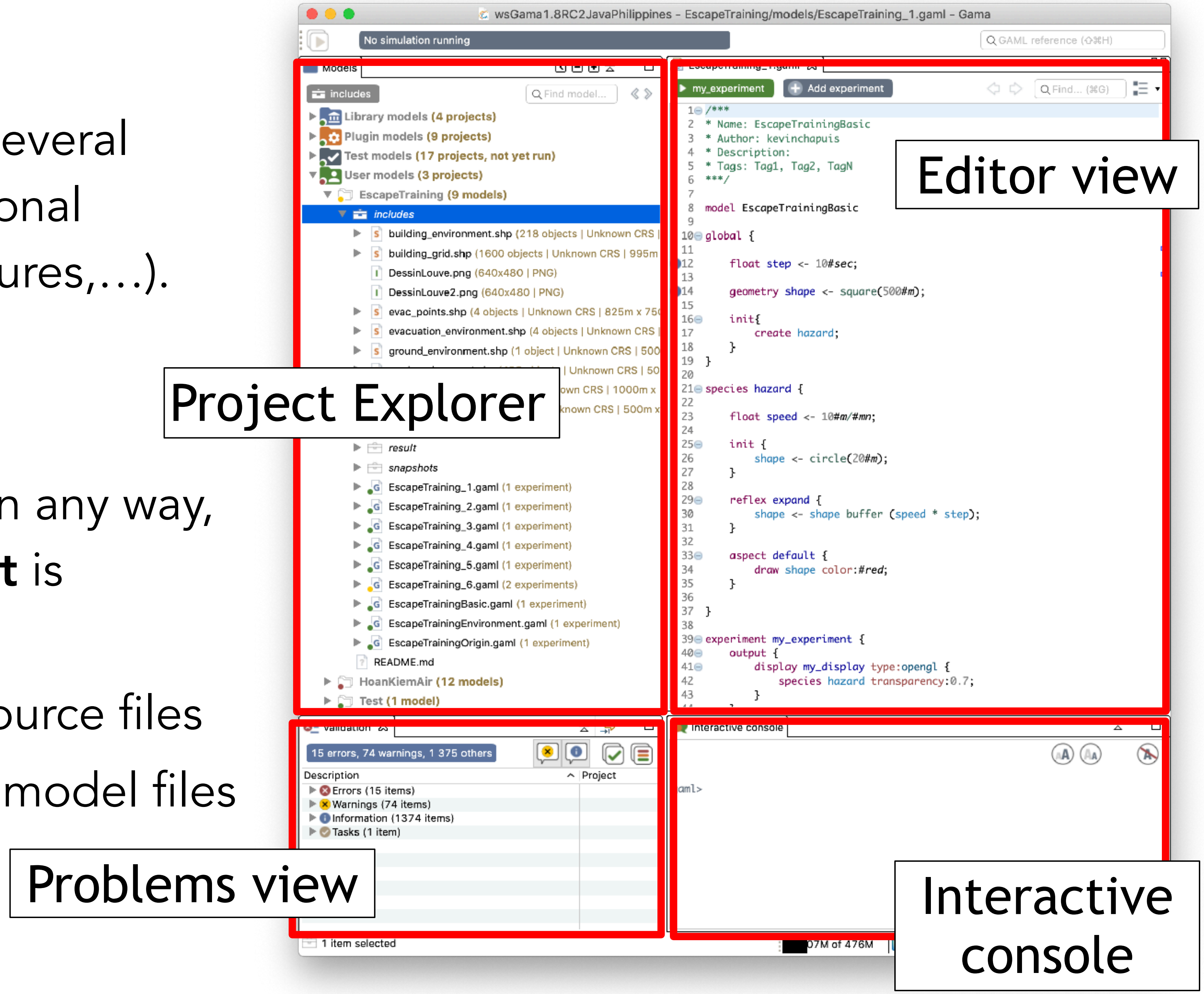


- Projects can be organised in any way, although a **default layout** is proposed:
  - ***includes*** : for all the ressource files
  - ***models*** : contains all the model files

GAMA model files are stored in projects

- Each **project** may contain several **models**, as well as additional **resources** (GIS data, pictures,…).

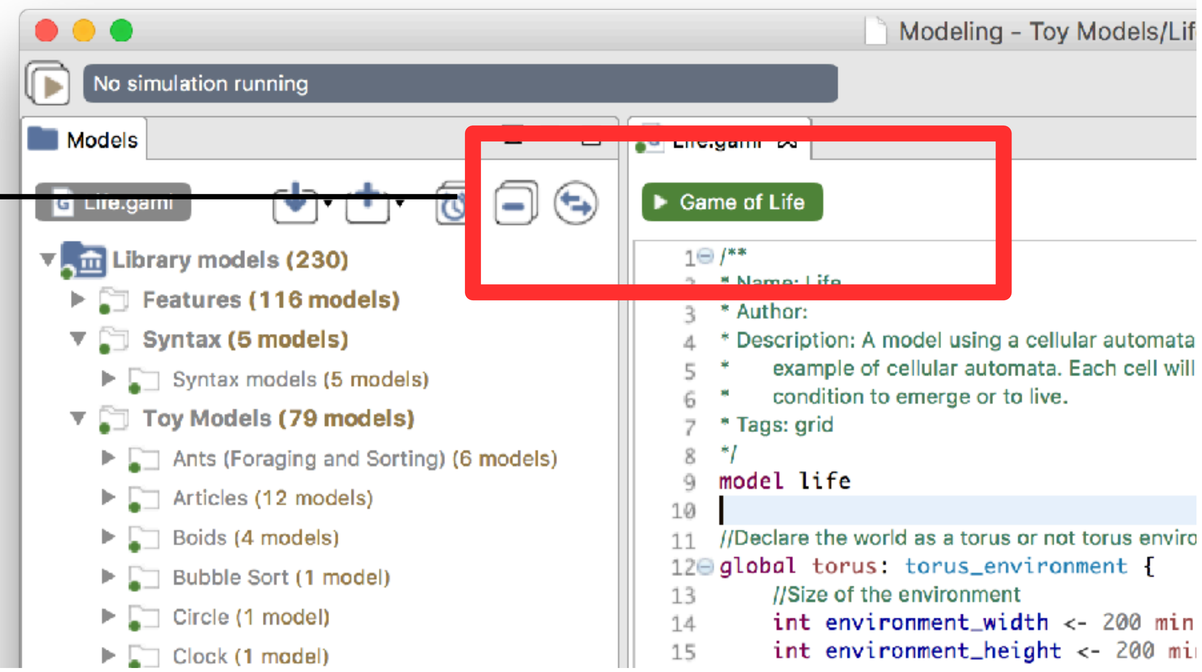
- Projects can be organised in any way, although a **default layout** is proposed:
  - ***includes*** : for all the ressource files
  - ***models*** : contains all the model files



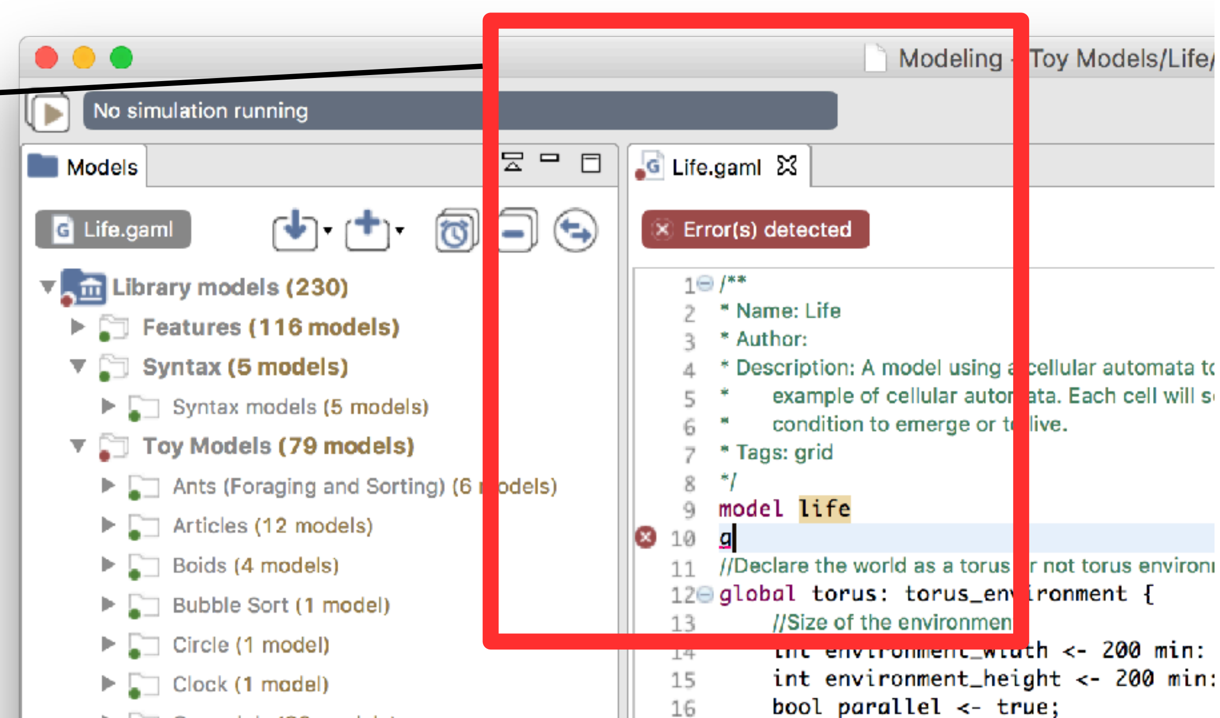
Take a look at “Game of life” model in library

- **Open the model Life.gaml**
  Models library \ Toy models \ Life \ Life.gaml

The model can be experimented

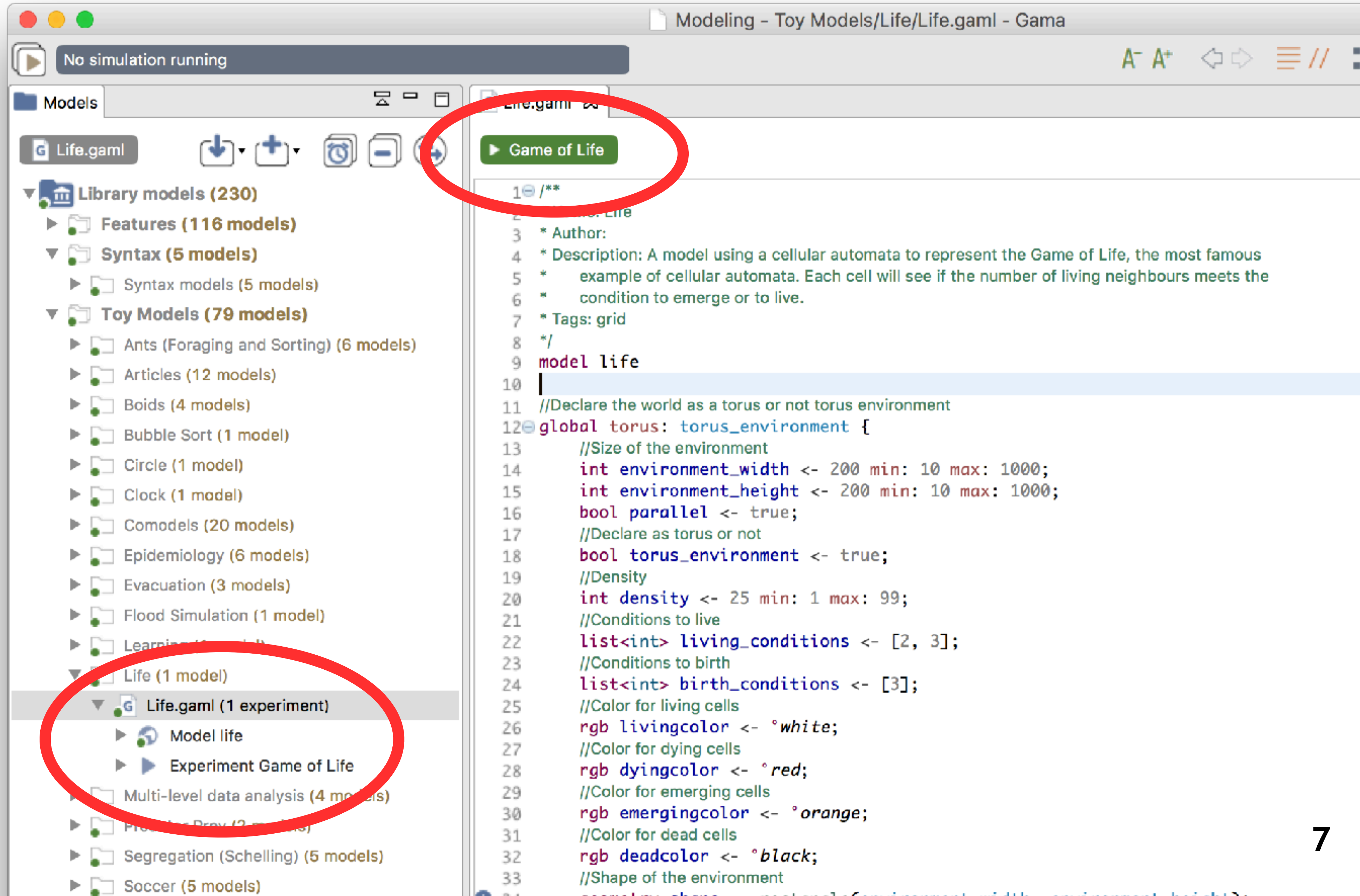


The model has errors



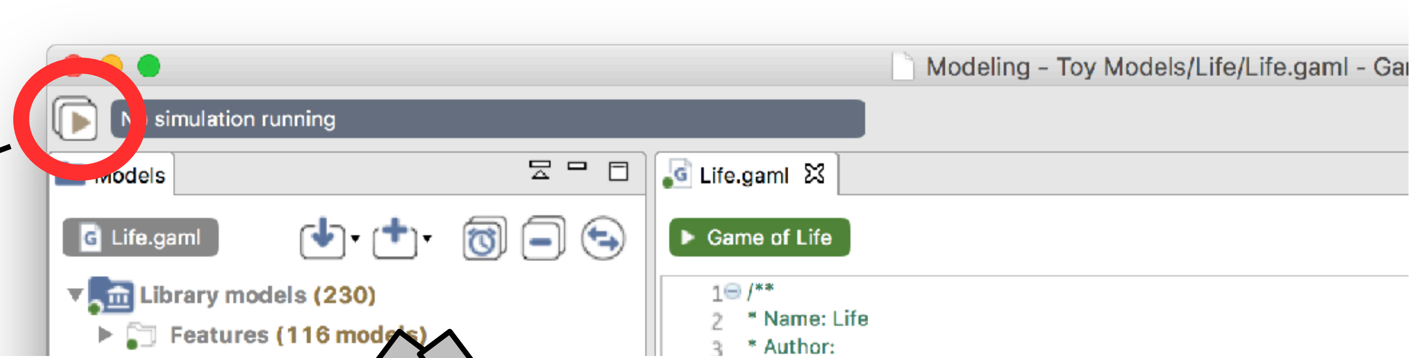
Launch experiment Game of life

- An **experiment** is a way to **"run" a model**.
- It can be reached either by:
  - Clicking on an Experiment button
  - in the Project Explorer, under the name of the model



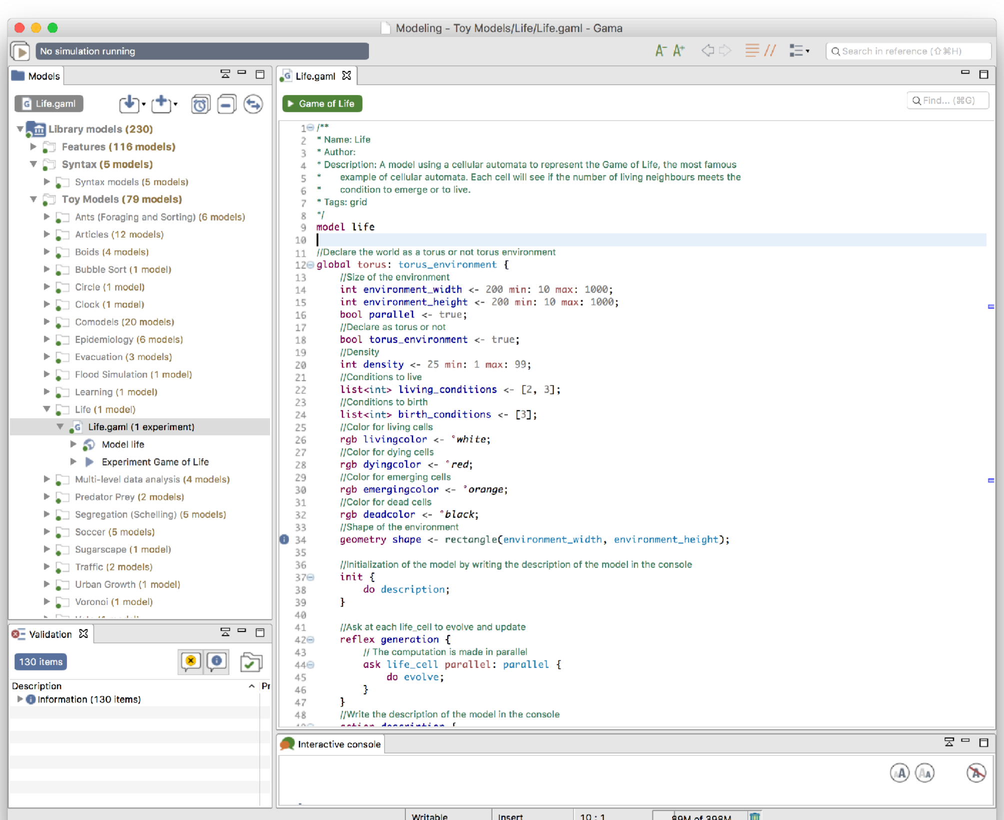
Launching an experiment will switch from *Modelling* to *Simulation* Perspective

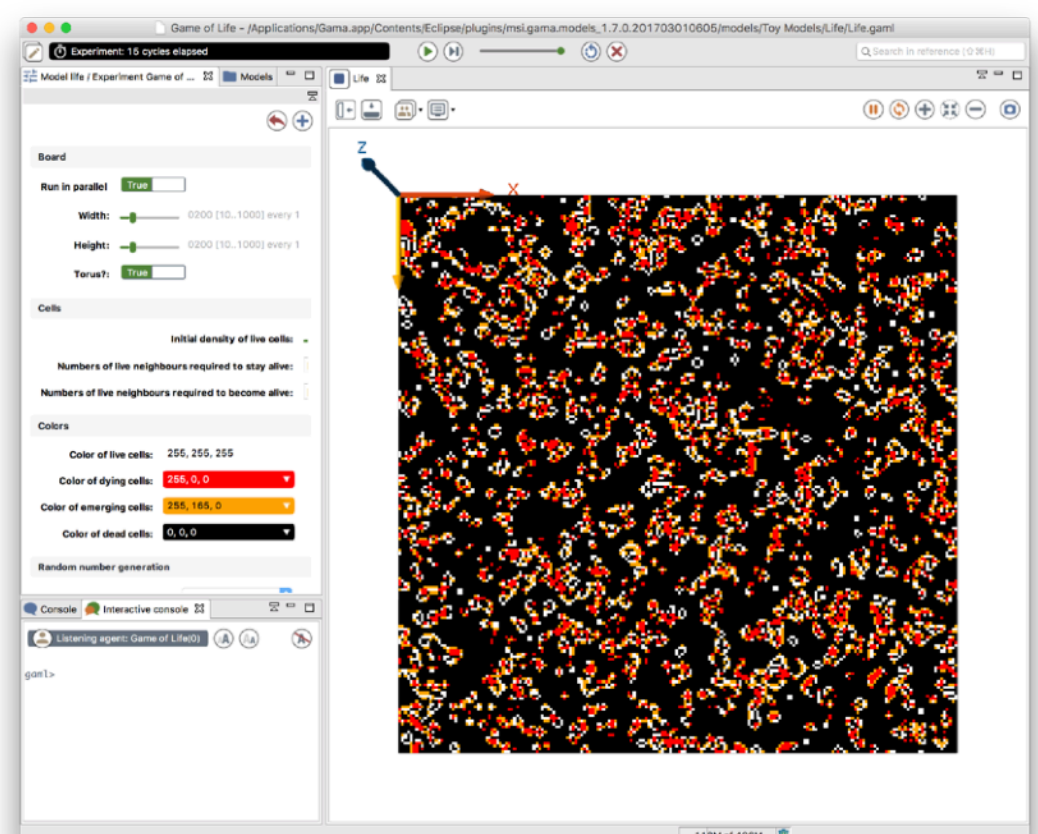
- Change the perspective



Modelling perspective

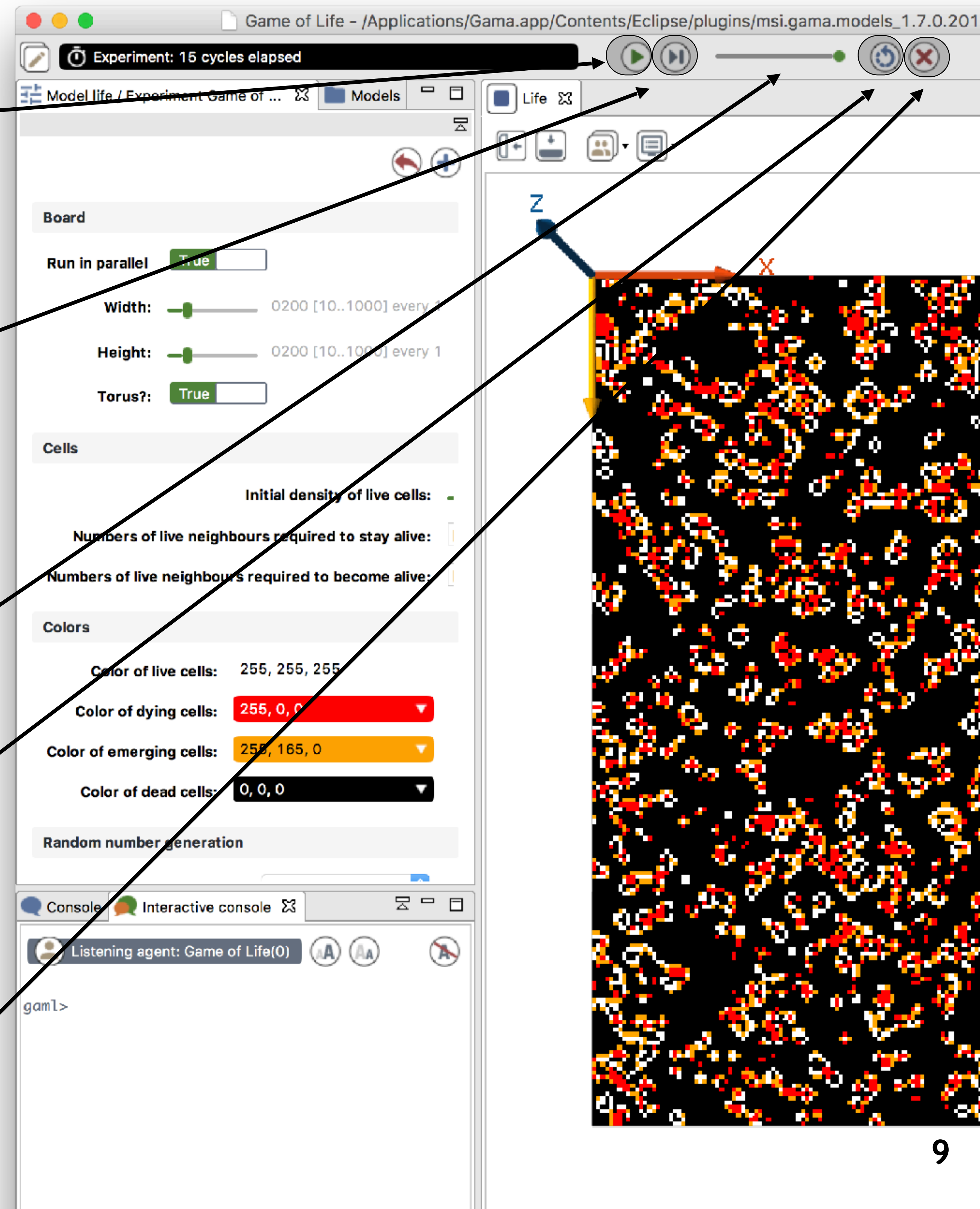
Simulation perspective





Exploring the Simulation perspective

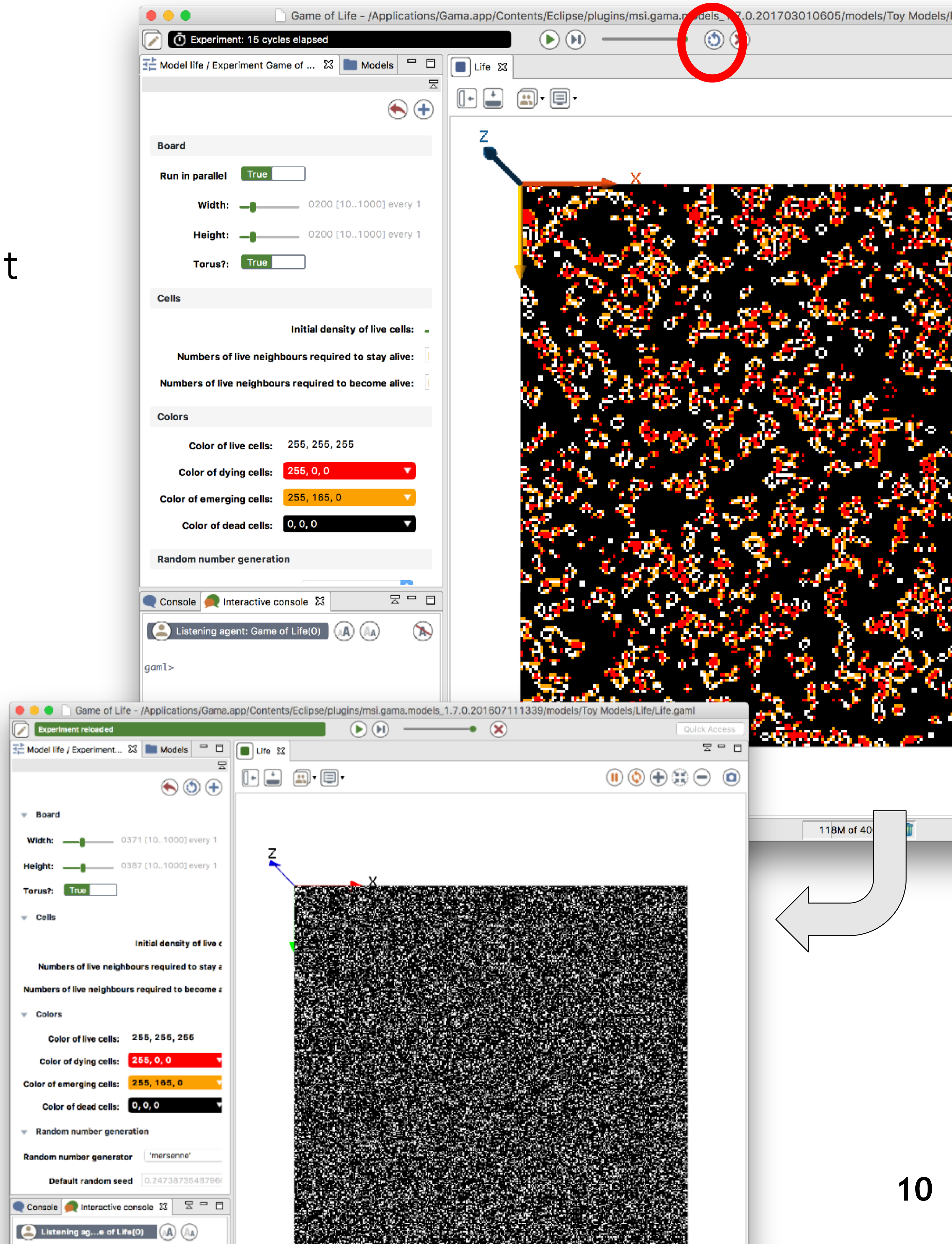
- **Start/pause** simulation (it will run until pause is clicked again)
- **Step** the simulation (it will run one cycle of the simulation)
- Adjust the **speed** of the simulation
- **Relaunch** the simulation (necessary after having changed the parameter values)
- **Interrupt** the simulation



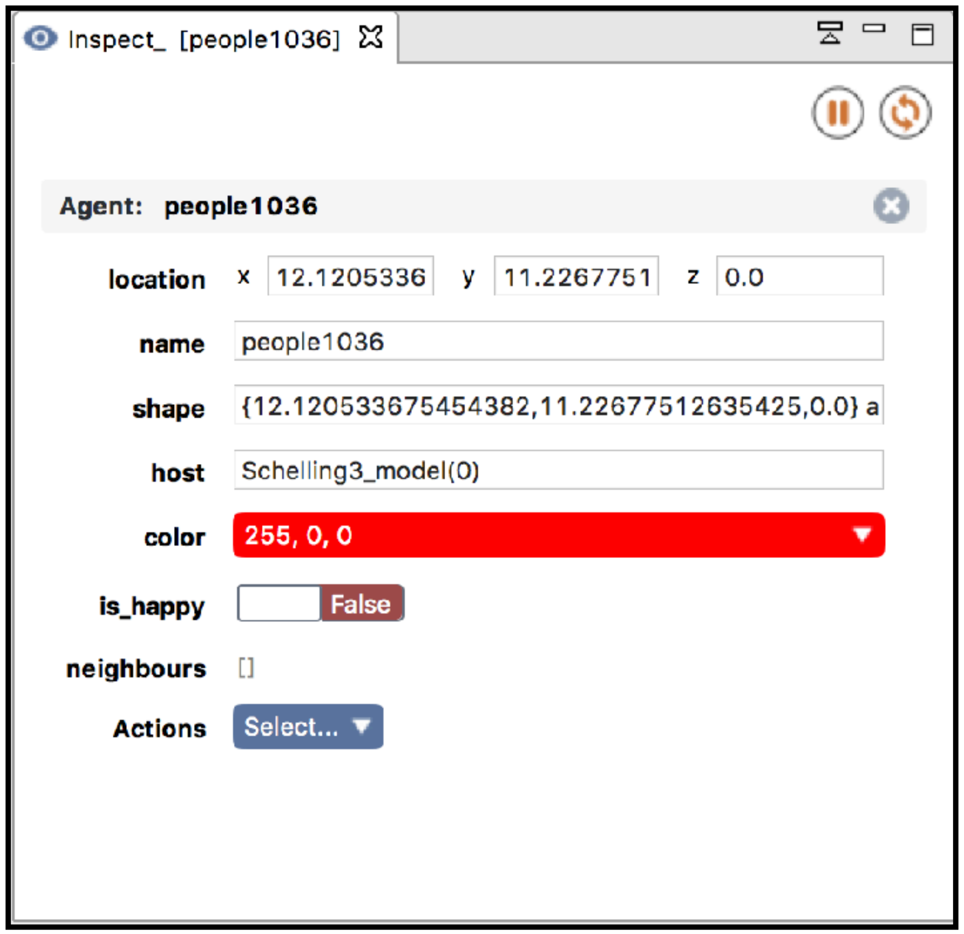
Explore the simulation with parameters modified from Parameters view

- The modifications made to the parameters are either:
  - Used **for the current simulation** when it makes sense (for instance, if the user changes a color)
  - Used **when the user reloads** the experiment otherwise (for instance, if the user changes the size of the grid)

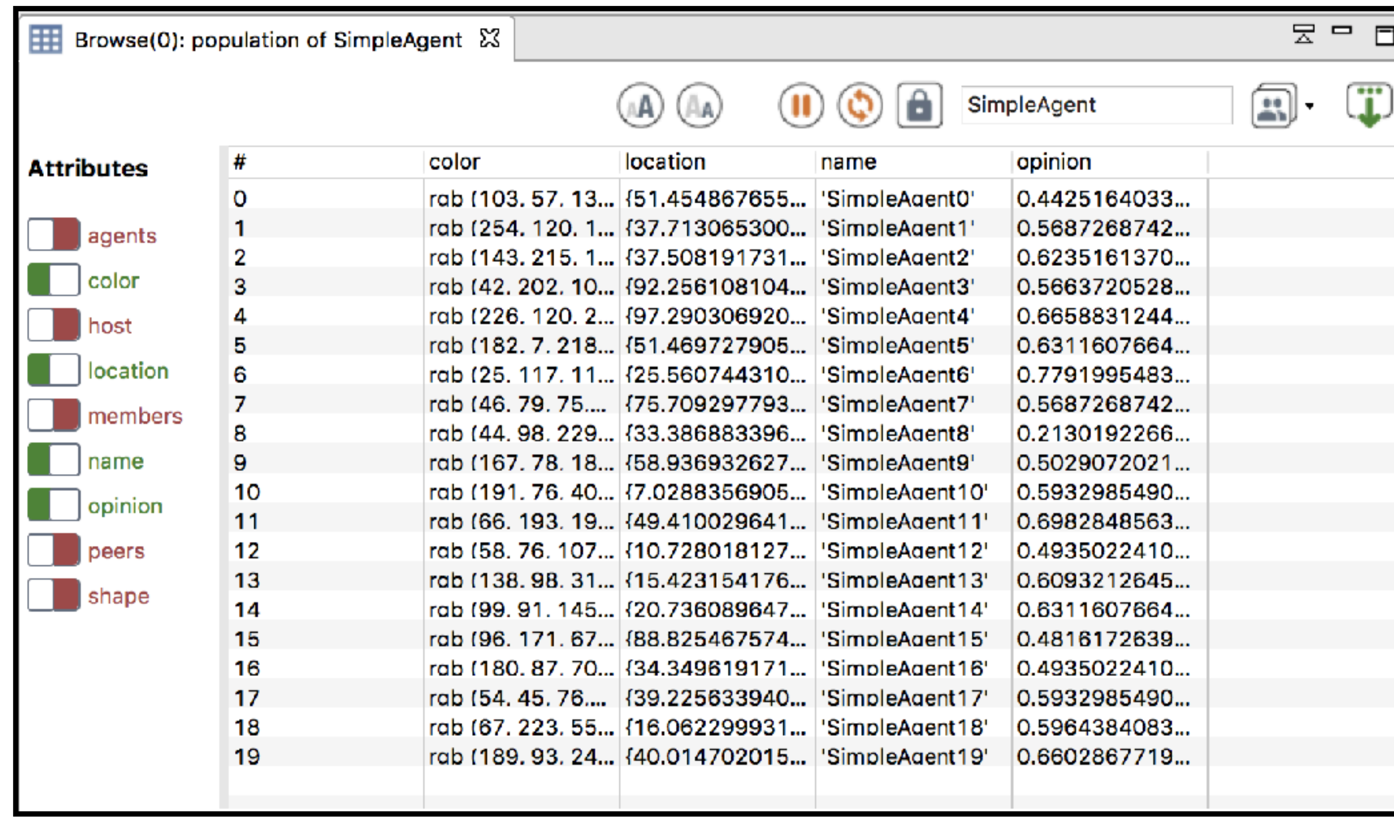
- Launching experiment again (from <u>the model editor</u>) will erase the modifications.



GAMA offers 2 views that display information about one or several agents



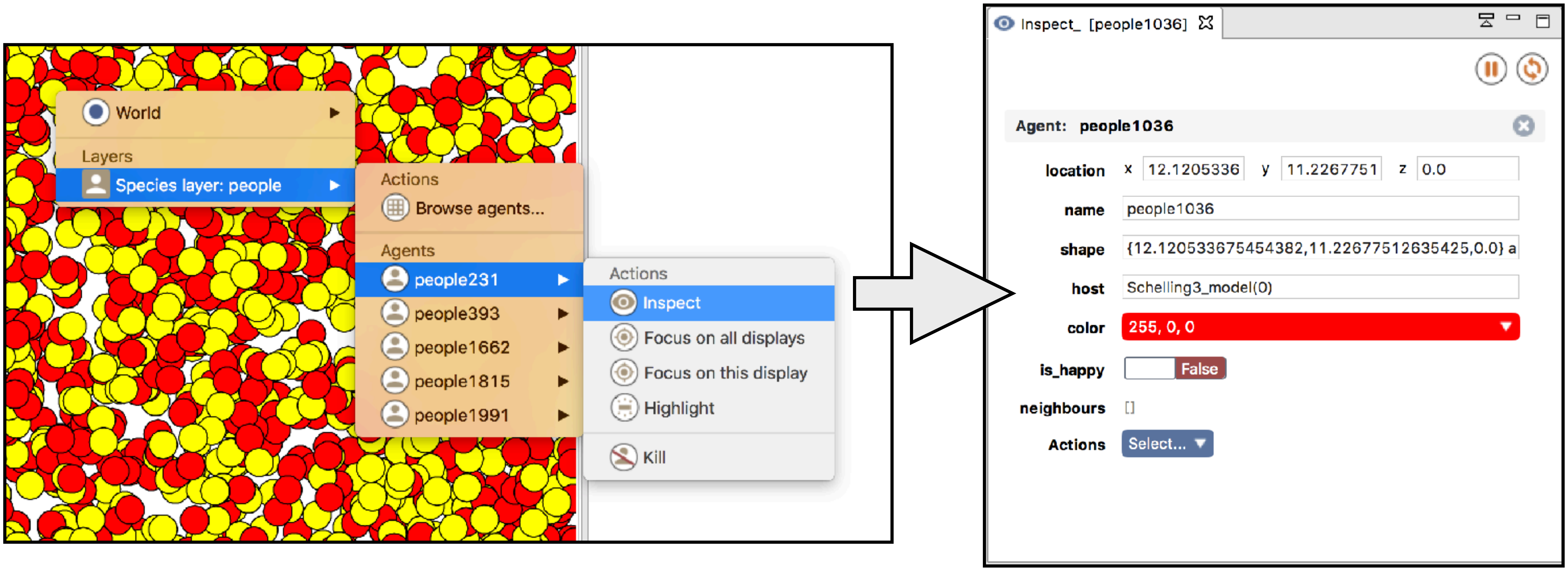
agent inspector



agent browser

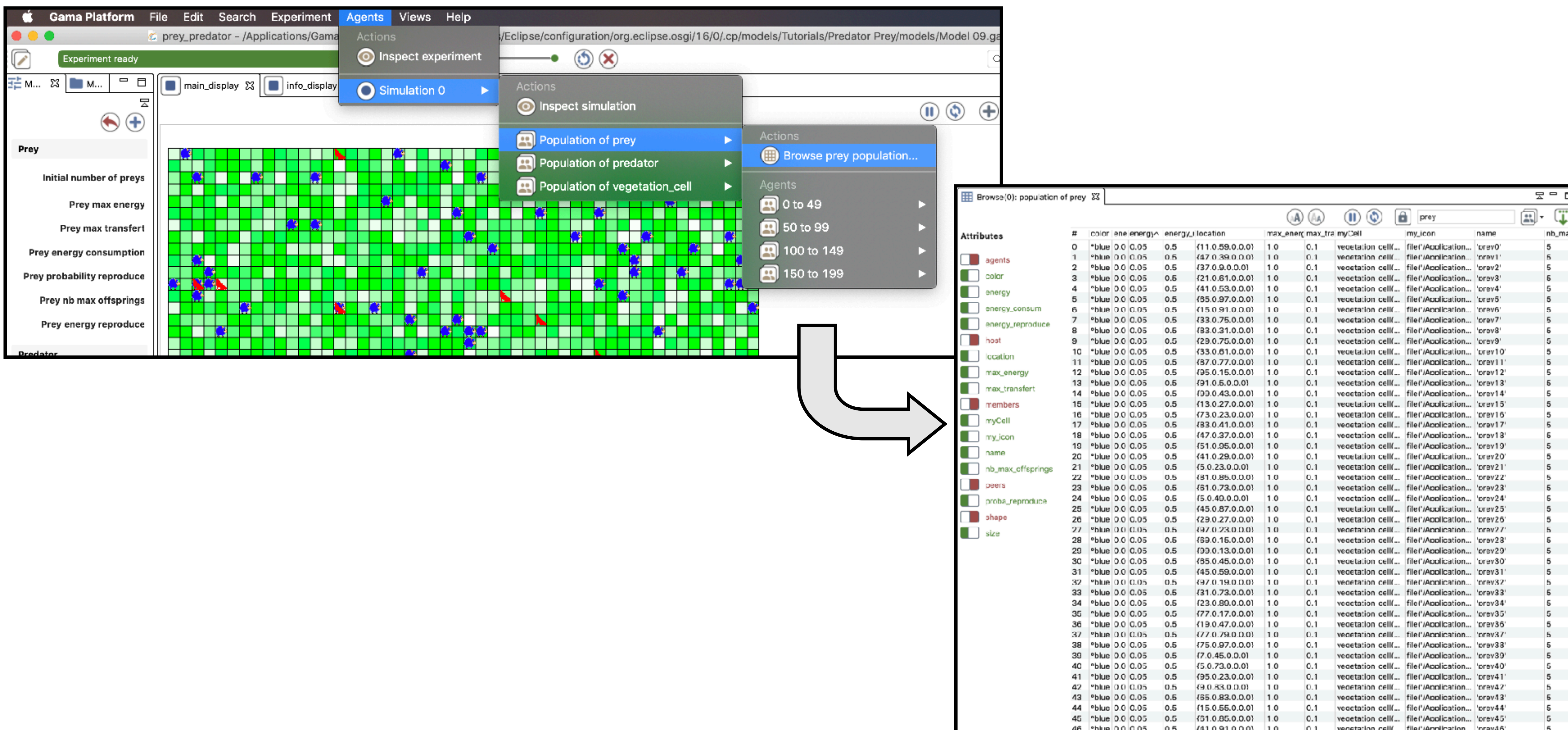
Inspect by right clicking on a agent in a display

- Provides information about **one specific agent**.
- It also allows to **change the values** of its variables during the simulation.
- It is possible to «highlight» the selected agent.



Inspect informations by agent browser

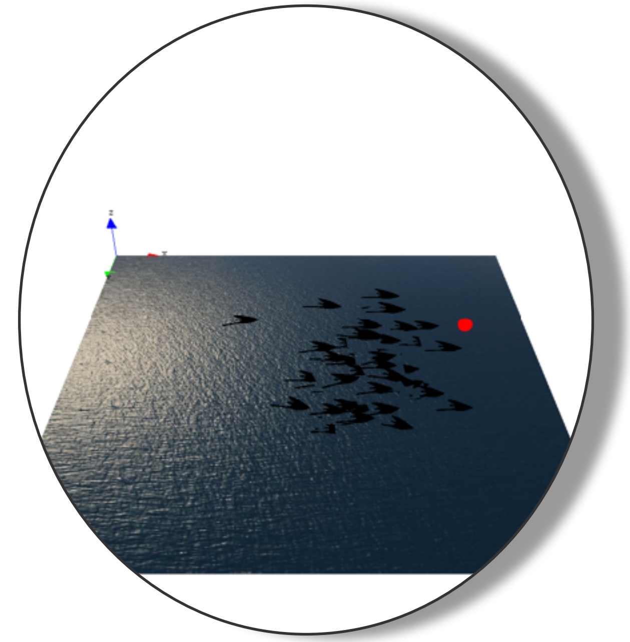
- The species browser provides informations about all or a selection of agents of a species.
- The agent browser is available through the **Agents** menu or by right clicking on a by right_clicking on a display



Take 5-10 minutes to explore some of the models of the Models Library.

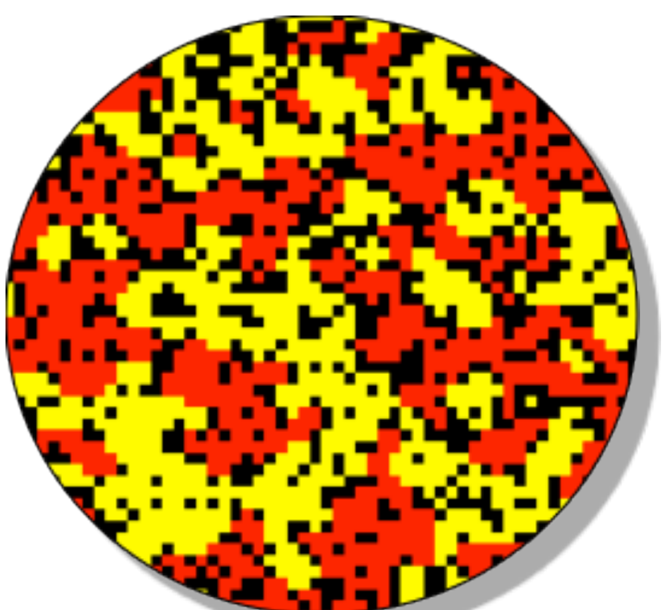
Toy Models\ Ants (Foraging and Sorintg) \ Ant Foraging.gaml

Experiment Classic



Toy Models \ Boids \ Boids 3D Motion.gaml

Experiment 3D



Toy Models\ Life \ Life.gaml

Experiment Game of Life

Toy Models\ Segregation (Schelling) \ Segregation (GIS).gaml

Experiment schelling

Write a ***first*** model: the Schelling's segregation model

Urban Segregation Model proposed by Schelling

- In 1969, Schelling introduced a model of segregation in which individuals of two different colours, positioned on a grid abstract representation of a district), choose where to live based on a **preferred percentage of neighbours of the same colour**.

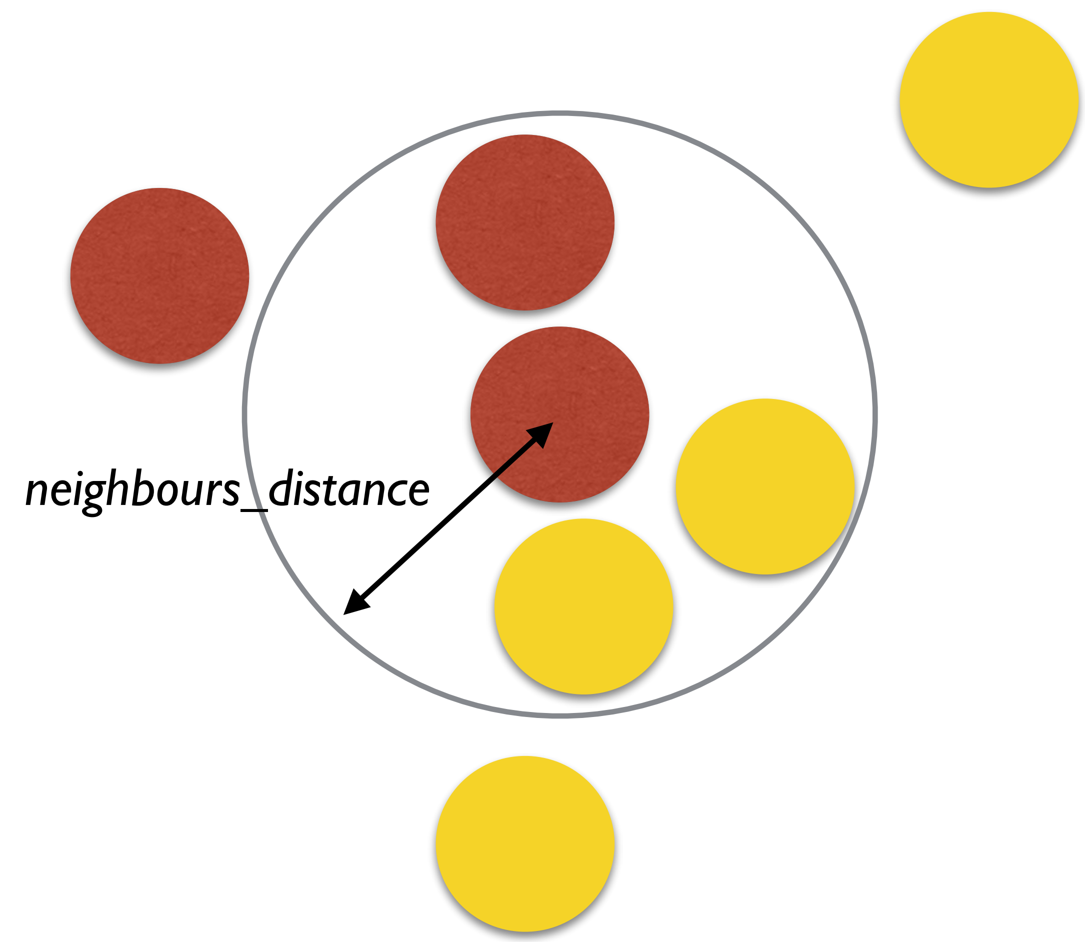
- Using coins on a board, he showed that a **small preference for one's neighbours** to be of the same colour could lead to **total segregation**.

- It is a good example of a generative model, where the emergence of a phenomenon here, segregation) is not directly predictable from the knowledge of individual

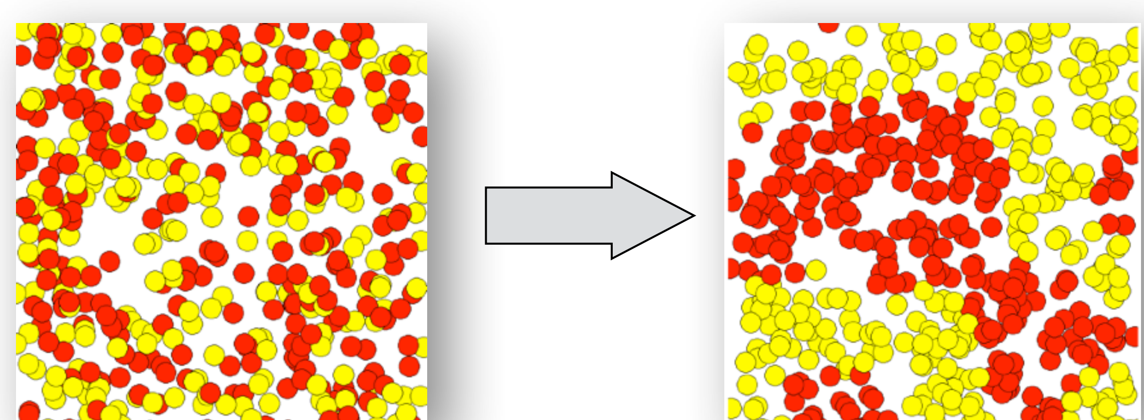
Proposed implementation of the Model

- ***People* agents** of 2 different colors (red and yellow) live in a continuous environment

- **At each simulation step**, each people agent:
  - **computes if it is happy:** it is happy if the rate of people agents at a distance *neighbours_distance* of the same color is higher or equals to the threshold *similar_rate_wanted*
  - if it is **not** happy, it **moves to a random location**

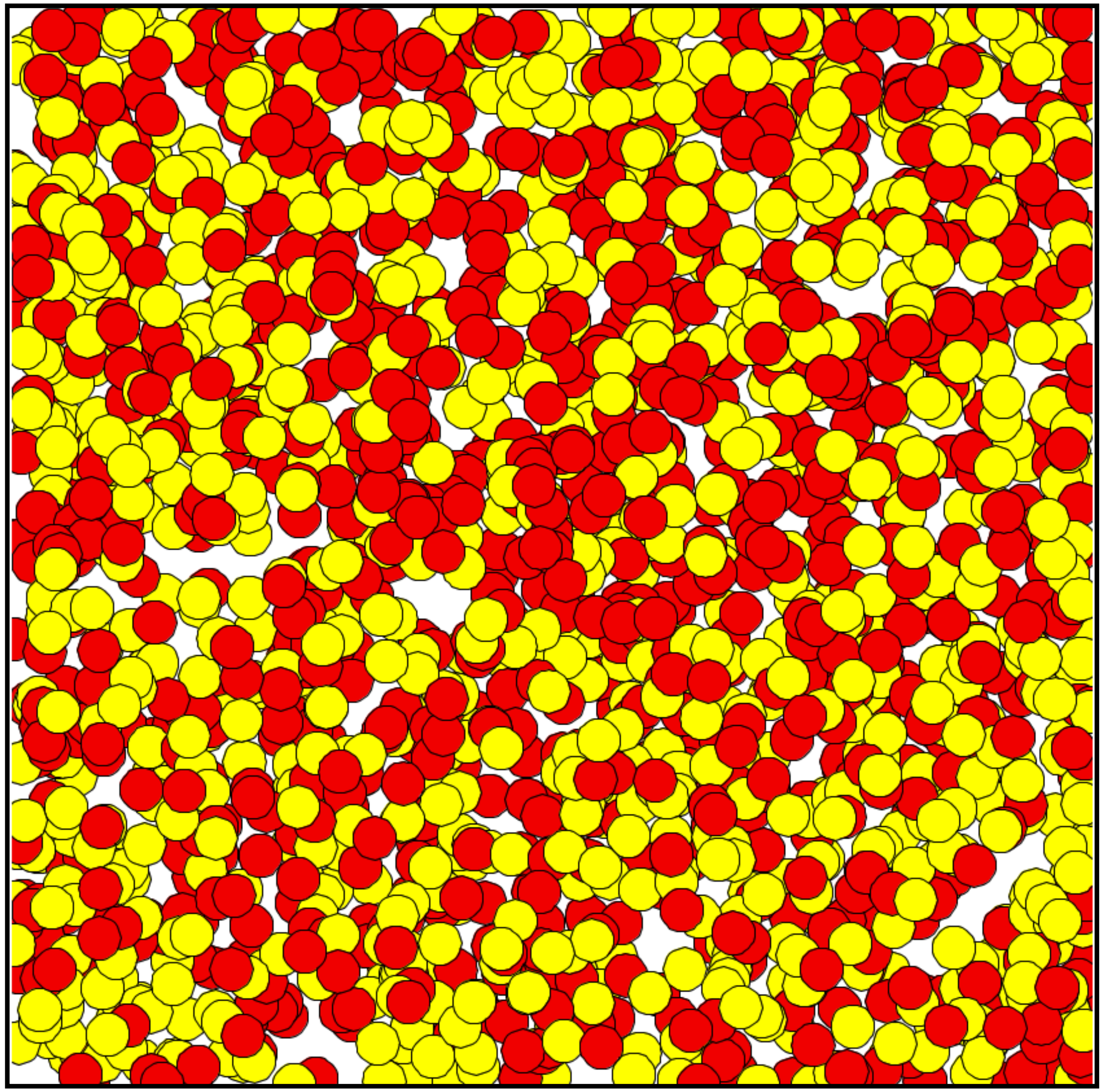


Similar_rate = 1/3 = 0.333
happy if similar_rate >= *similar_rate_wanted*

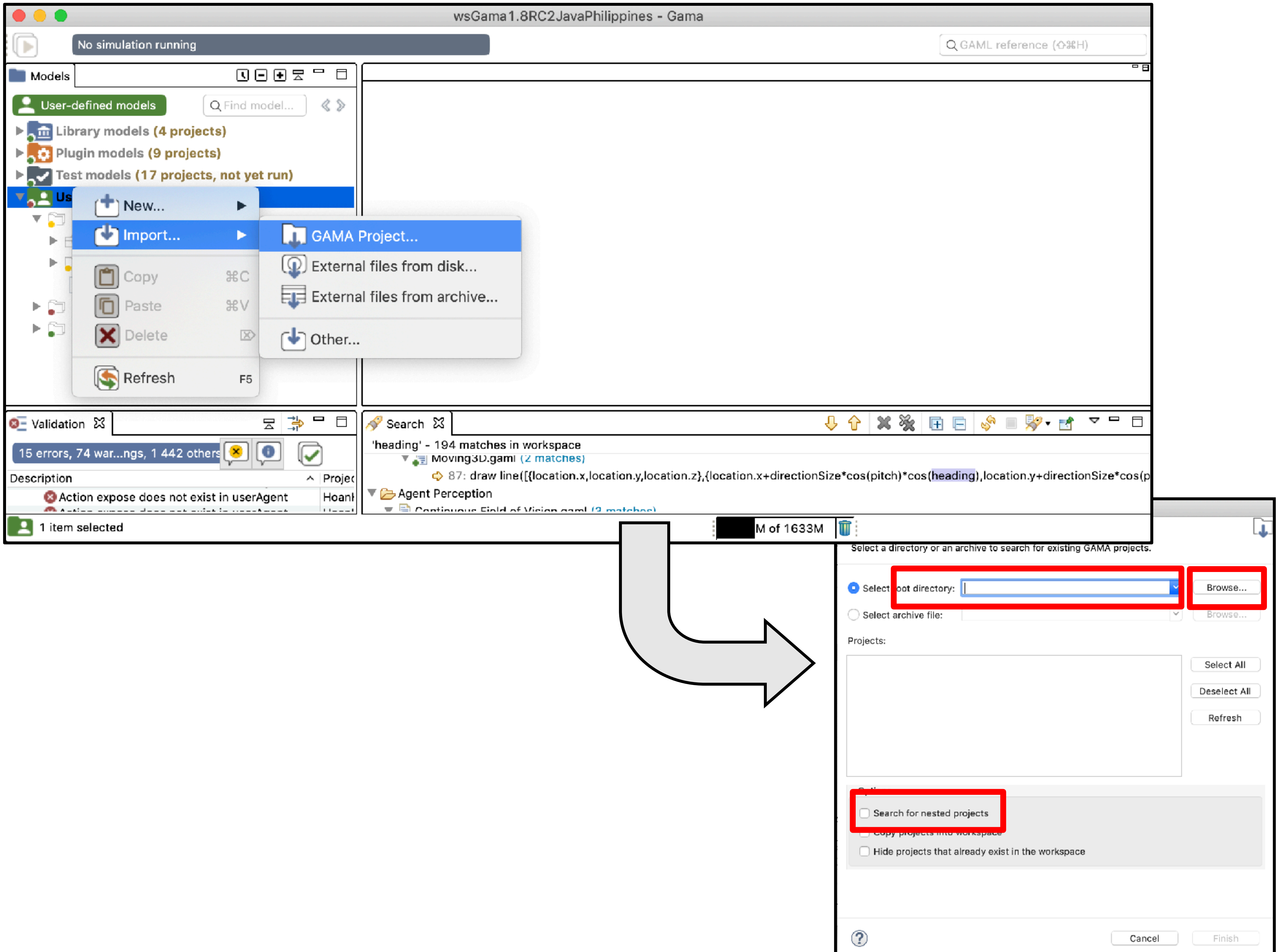


Step 1: definition and display of the people species

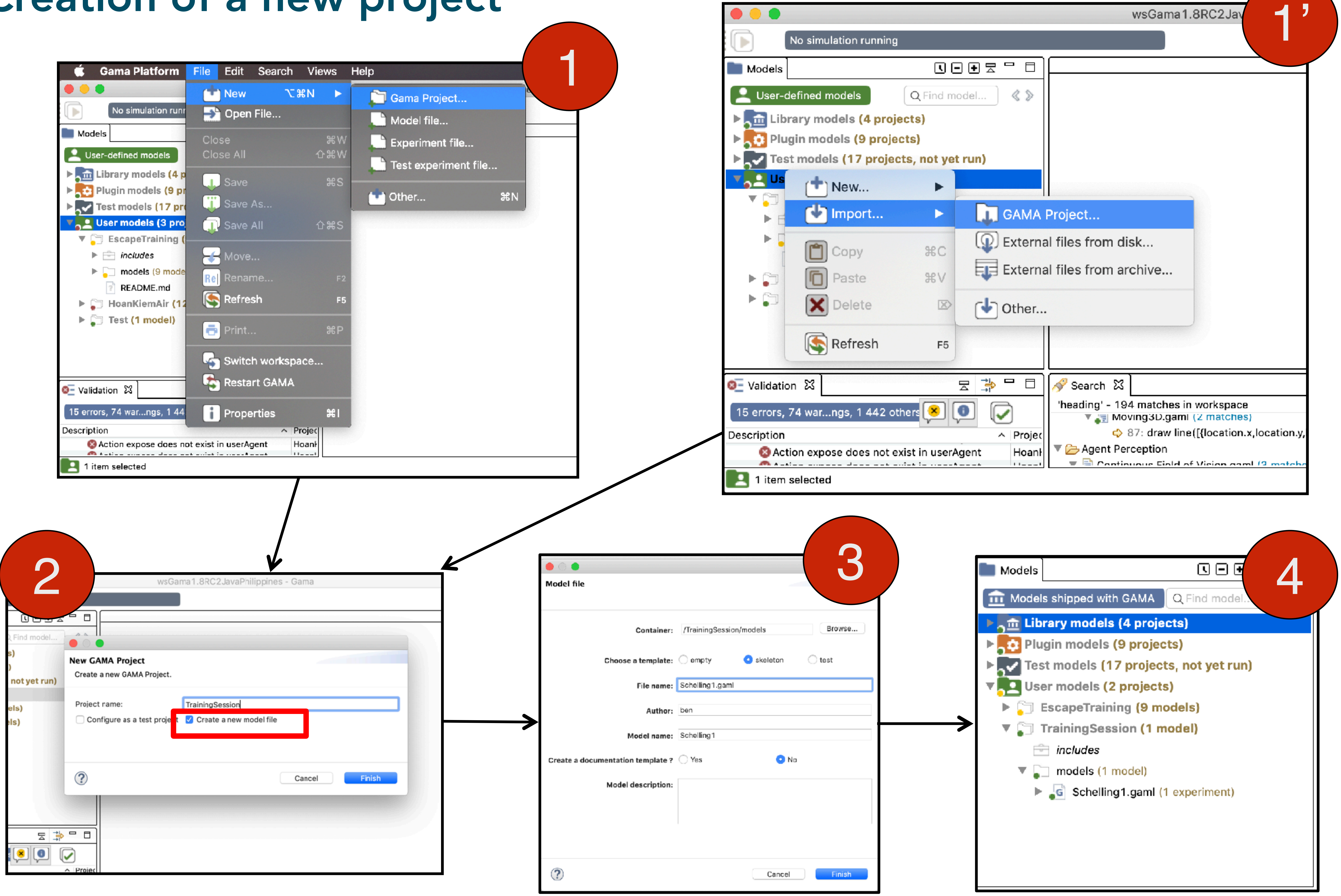
- Objectives:
  - ➡ Definition of the *people* species
  - ➡ Creation of *2000* people agents randomly located in the environment
  - ➡ Display of the agents



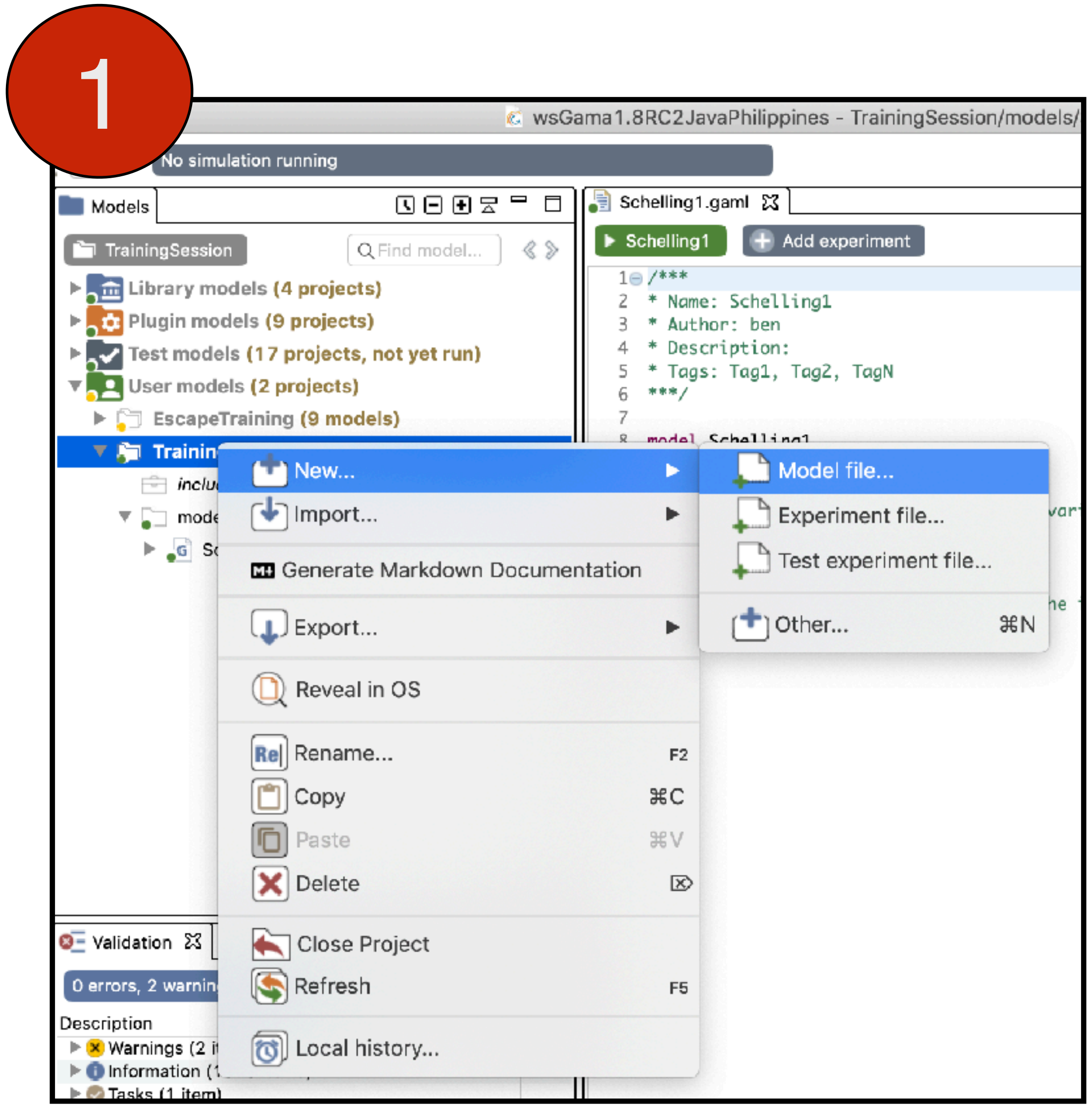
Import existing projects into the workspace

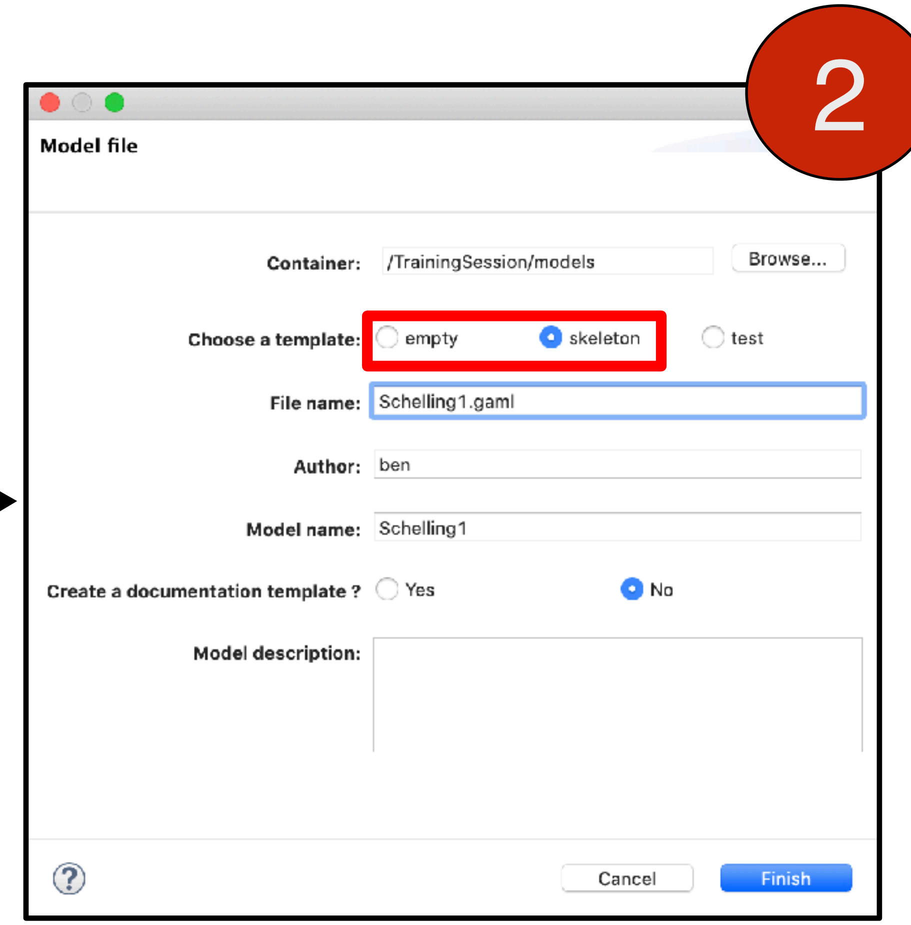


Creation of a new project



Creation of a new model file





3

```
el mymodel

global {
     /** Insert the global definitions, variables and actions here */
}

experiment mymodel type: gui {
     /** Insert here the definition of the input and output of the model */
     output {
     }
}
```

Introduction to the main concepts of the GAMA Modelling Language - GAML

- The role of GAML is to support modellers in writing **models**, which are specifications of **simulations** that can be executed and controlled during **experiments**, themselves specified by experiment plans.

- Agents in GAML are specified by their **species**, which provide them with a set of **attributes** (*what they are, know…*), **actions** (*what they can do*), **behaviours** (*what they actually do*) and also specifies properties of their **population**, for instance its **topology**

- **Everything is an agent** in GAML: the model itself (called the *world*), the agents defined in it, the experiments…

Therefore, the structure of a model in GAML is simply a set of *species declaration statements*

- 3 types of block declaration (equivalent to species statements) are supported:
  - **Global (unique)**: global attributes, actions, dynamics and initialisation.
  - **Species** and **Grid**: agent species. Several species statements can be defined in the same model.
  - **Experiment** : simulation execution context, in particular inputs and outputs. Several experiment

2 ways to write commentaries (texts that are not just part of the model but here for information purpose):

- //… : for one line. Example : //this is a commentary
- /* … */ : can be used for several lines. Example : /* this is as well a commentary */

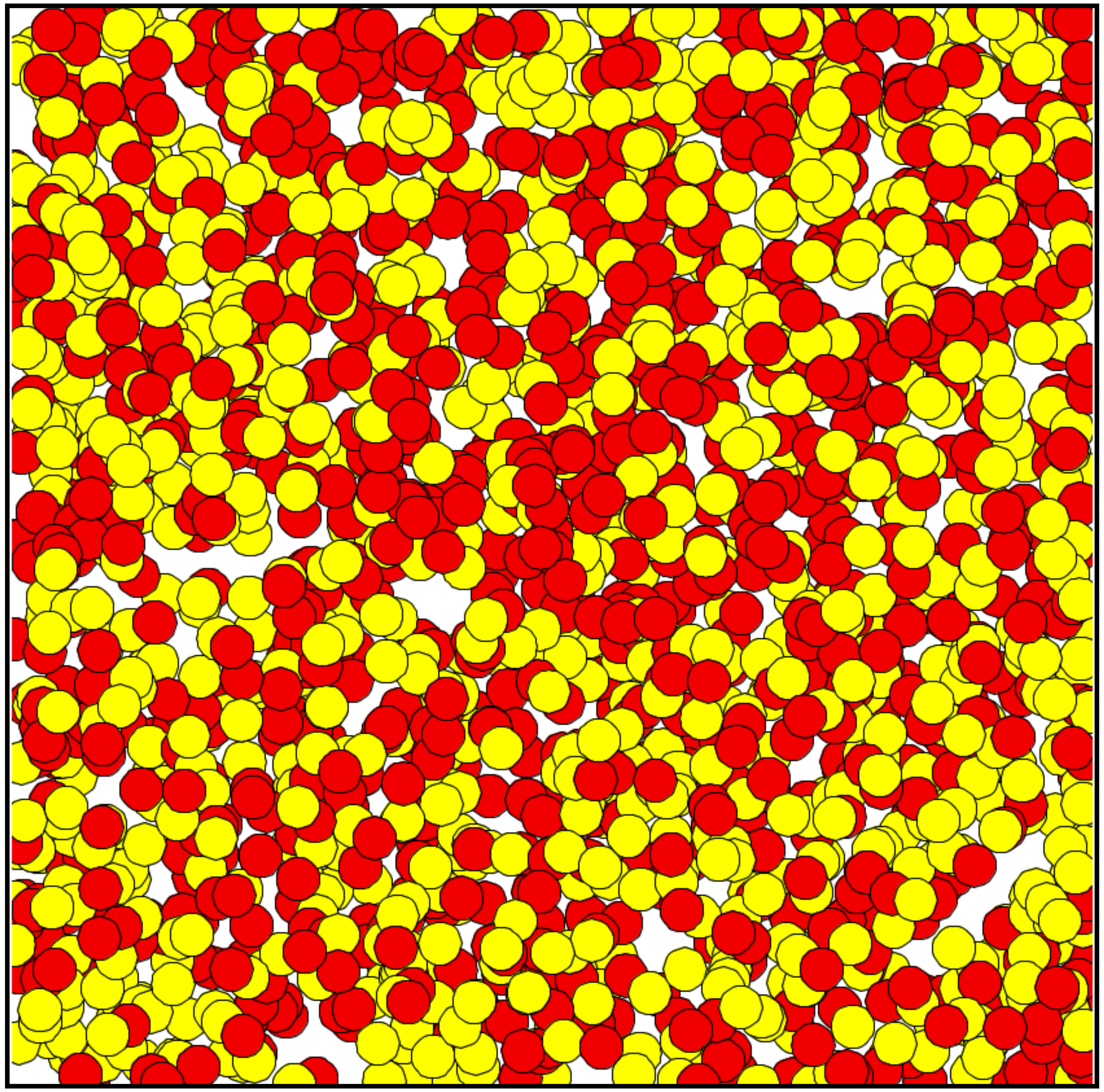
General Structure of a model	Model

```
model my_model

global {
    /** Insert the global definitions,
     * variables and actions here
     */
}

species my_species{
    /** Insert here the definition of the
     * species of agents
     */
}

experiment my_model type: gui {
    /** Insert here the definition of the
     * input and output of the model
     */
}
```

Segregation model 1: People Species

- **To do:** We want to create and display 2000 people agents.

- **Steps to follow:**
  - Definition of the people species
  - Creation of 2000 people agents randomly located in the environment
  - Display of the agents



Segregation model 1:
Step 1. People Species definition

- **To do:** define the species *people*:

- **Solution:**

People species definition	Model

```
model my_model

global { }



species people{

}


experiment my_model type: gui { }
```

Segregation model 1:
Step 2. Creation of 2000 people agents

- **To do:** create 2000 *people* agents

- **Hint:** this done at the **initialization** of the simulation, so in the **init** block of the global

- **Solution:**

Creation of 2000 people agents	Global

```
model my_model

global {
    init {
        create people number: 2000;
    }
}


species people{ }

experiment my_model type: gui { }
```

The GAML corner:
THE first cause of error in writing models is at the end of the line!

The rule:

- A line (i.e. a statement) always **ends** with either **« ; »** or a block of statements
- A block of statements is marked out by **« { … } ».**
  - A block allows to execute a set of instructions in the context of another statement (create agents during the initialization).

Creation of 2000 people agents	Global

```
model my_model

global {
    init {
        create people number: 2000;
    }
}


species people{ }

experiment my_model type: gui { }
```

Segregation model 1:
Step 3. Display of the people agents

- **To do:** display the 2000 *people* agents

- **Hint:** the definition of the displays is made in an experiment

Display the people agents	Experiment

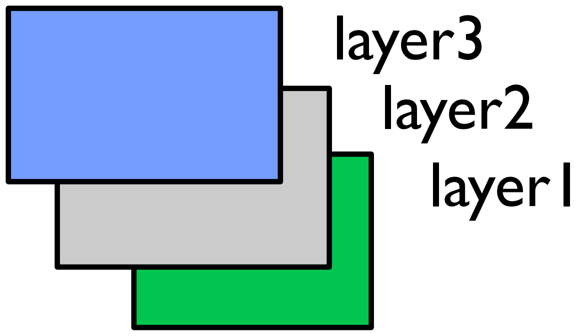
- **Solution:**

```
model my_model

global {
    init {
        create people number: 2000;
    }
}

species people{ }

experiment Schelling1 type: gui {
    output {
        display people_display {
            species people;
        }
    }
}
```

The GAML corner: *experiment* block: output definition

- The *output* block has to be defined in an *experiment* block

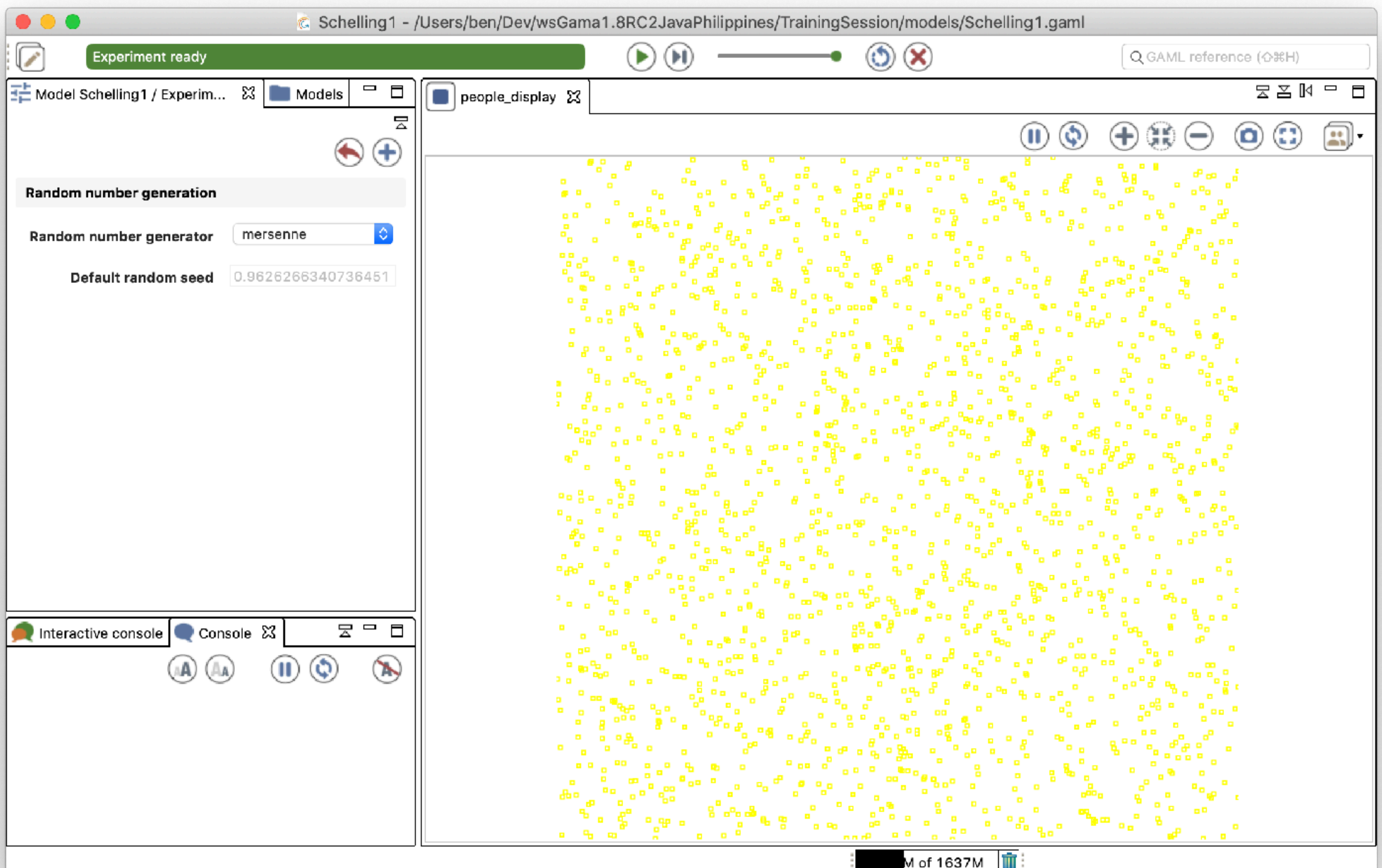
- It allows to define **displays**:
  - Each *display* can contain different displays:
    - Agent species (all the agents of the species) :
      species *my_species* **aspect:** *my_aspect*
    - list of agents :
      agents *layer_name* **value:** *agents* **aspect:** *my_aspect;*
    - Grids: optimised display of grids:
      grid *grid_name* **lines**: *my_color;*
    - Images:
      image *layer_name* **file:** *image_file*;
    - Charts: see later
  - A refreshing rate can be defined: facet **refresh:** *nb (int)*



Segregation model 1:
Step 3. Display of the people agents

- **To do:** display the 2000 *people* agents

- **Result:** people are only displayed as points, with the same color for all the agents.



Segregation model 1:
Step 4. Define the way agents are displayed through an aspect

- **To do:** define an aspect for the *people* agents

- **Solution:** define an aspect in the people species and use it in the display.

Define an aspect for people agents	People

```
species people {
    aspect asp_circle {
        draw circle(1.0) color: #red border: #black;
    }
}
```

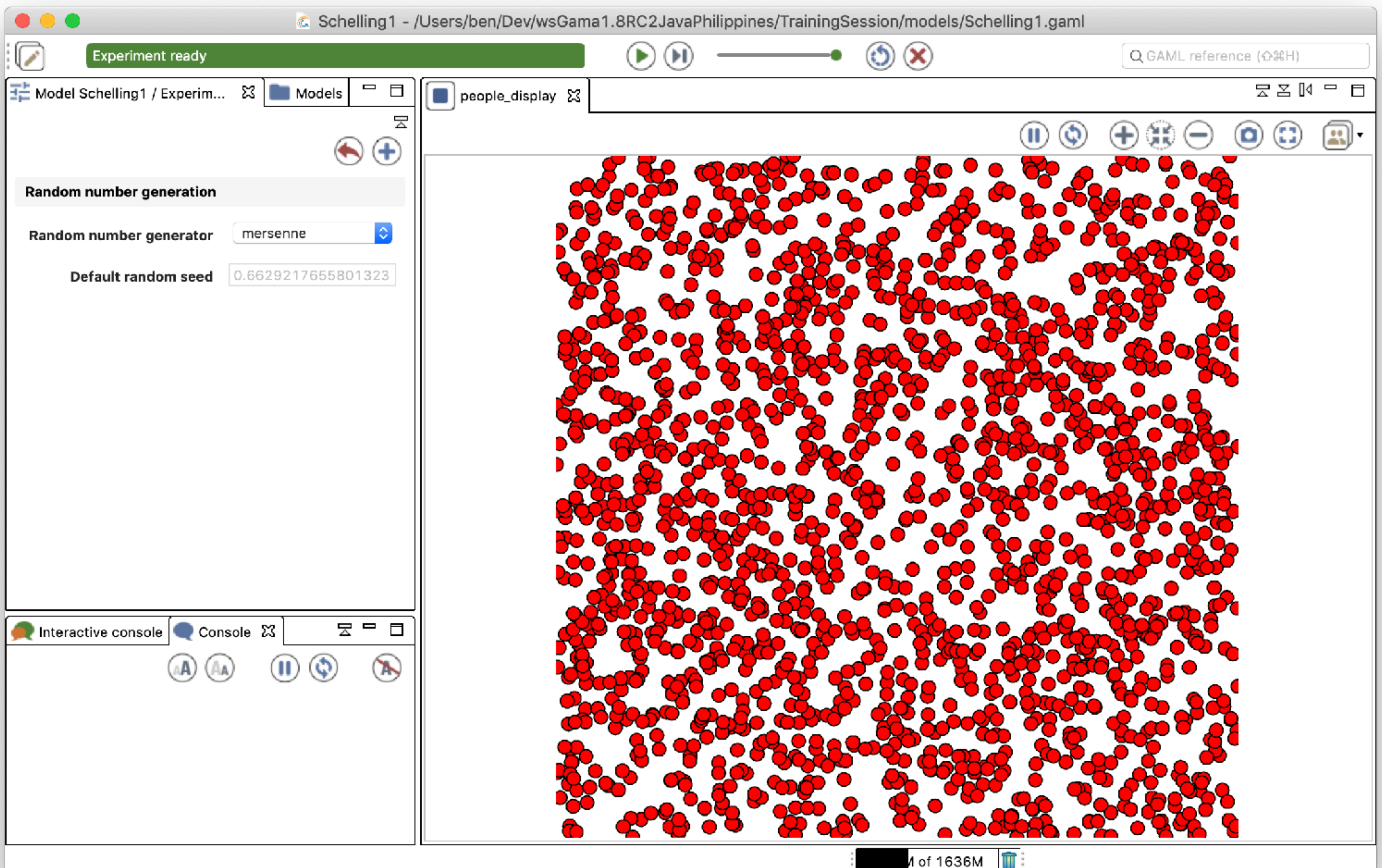
Display the people agents	Experiment

```
experiment Schelling1 type: gui {
    output {
        display people_display {
            species people aspect: asp_circle;
        }
    }
}
```

Segregation model 1:
Step 4. Define the way agents are displayed through an aspect

- **To do:** define an aspect for the *people* agents

- **Results:**



The GAML corner: A statement represents either an imperative command or a declaration

- **Each line in a GAML model is a statement.**

- It consists in a **keyword**, followed by a list of **facets** (some of them mandatory), ended by ";" or a **block** of statements.

- A **facet** is a keyword, followed by ":", and an **expression**.
  - Note that the keyword of the first facet can usually be omitted.
  - If the statement is a declaration, the first facet contains an **identifier.**
- A **block** is a set of statements enclosed into curly brackets ("{" and "}")

Example of statements	Model

```
global {
    init {
        create people number: 2000;
    }
}

species people{ }
```

The GAML corner: A statement represents either an imperative command or a declaration

- **Each line in a GAML model is a statement.**

- It consists in a **keyword**, followed by a list of **facets** (some of them mandatory), ended by “;” or a **block** of statements.

- A **facet** is a keyword, followed by “:”, and an **expression**.
  - Note that the keyword of the first facet can usually be omitted.
  - If the statement is a declaration, the first facet contains an **identifier.**
- A **block** is a set of statements enclosed into curly brackets (“{” and “}”)

Example of statements	Model

```
global {
    init {
        create people number: 2000;
    }
}

species people{ }
```

keyword (create)

First **Facet** without a **keyword** (people)

Facet with a **keyword** (*number*) and an **expression** (2000)

The GAML corner: A statement represents either an imperative command or a declaration

- **Each line in a GAML model is a statement.**

- It consists in a **keyword**, followed by a list of **facets** (some of them mandatory), ended by ";" or a **block** of statements.

- A **facet** is a keyword, followed by ":", and an **expression**.
  - Note that the keyword of the first facet can usually be omitted.
  - If the statement is a declaration, the first facet contains an **identifier.**

- A **block** is a set of statements enclosed into curly brackets ("{" and "}")

Example of statements	Model

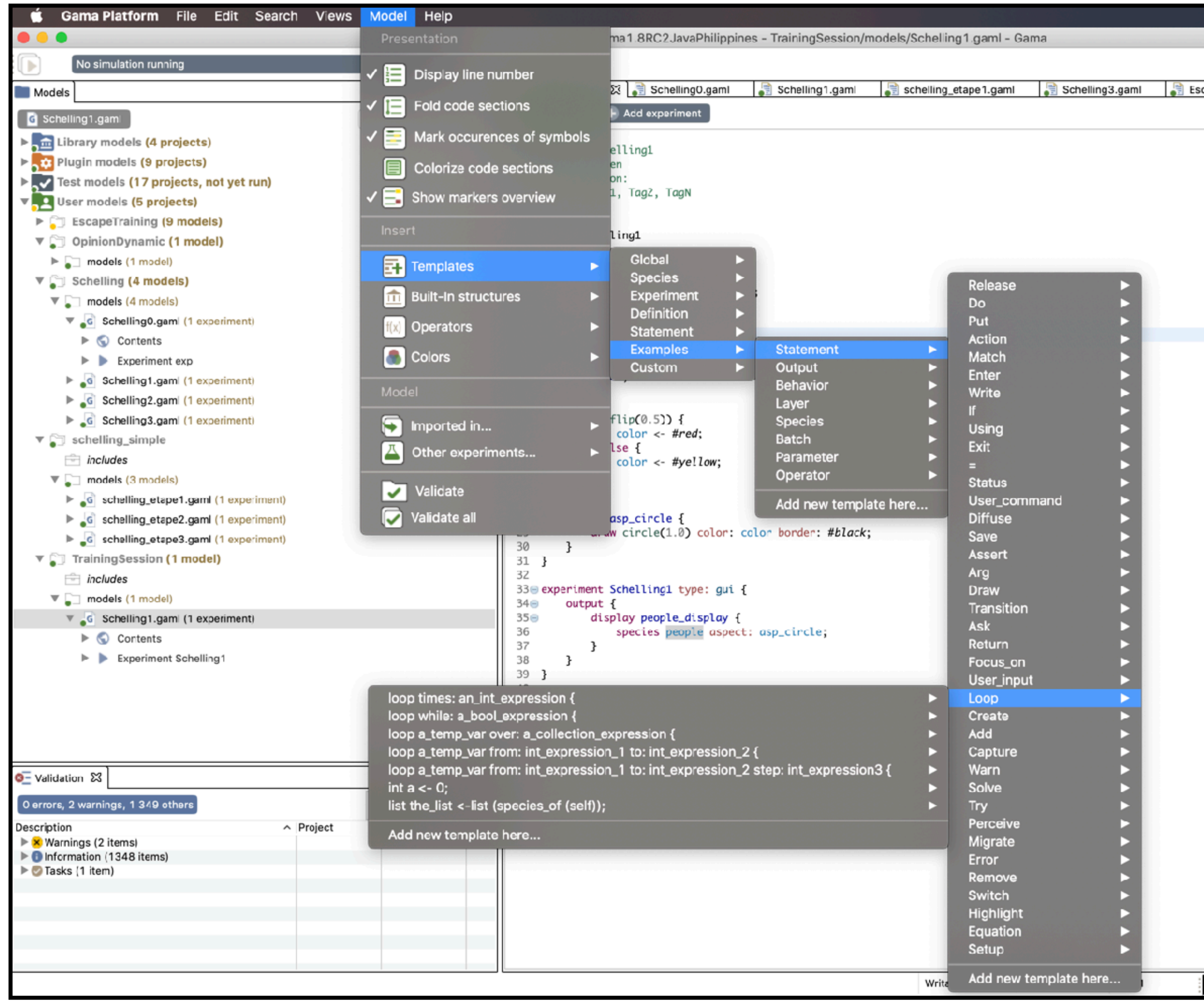
Examples of statement **keywords**

```
global {
    init {
        create people number: 2000;
    }
}

species people{ }
```

keyword

The GAML Corner: example of statements

- The GAML language contains many statements:
  - draw
  - create
  - loop
  - If - else
  - declaration
  - …



- Example of the if - else:

```
if( condition ) {
    set of statements to perform if the condition is true
} else {
    set of statements to perform otherwise
}
```

Segregation model 1:
Step 5. Define the color of each agent

- **To do:** each agent is displayed with a color (red or yellow) that characterize it
- **Hints:** each people agent will be characterized by a color value, which is initialized to a random color (among red and yellow).

Add the color attribute and initialize it | People

```
species people {

    rgb color;

    init {
        if( flip(0.5) ) {
            color <- #red;
        } else {
            color <- #yellow;
        }
    }

    aspect asp_circle {
        draw circle(1.0) color: color border: #black;
    }
}
```

- **Solution:**
  - Add a color attribute to the people species
  - Initialize it to a random color value among red and yellow.
  - Use the color in the display

Segregation model 1:
5. Define the color of each agent

- **To do:** each agent is displayed with a color (red or yellow) that characterize it
- **Hints:** each people agent will be characterized by a color value, which is initialized to a random color (among red and yellow).
- **Solution:**
  - **Add a color attribute to the people species**
  - Initialize it to a random color value among red and yellow.
  - Use the color in the display

Add the color attribute and initialize it | People

```
species people {

  rgb color;

  init {
      if( flip(0.5) ) {
          color <- #red;
      } else {
          color <- #yellow;
      }
  }

  aspect asp_circle {
      draw circle(1.0) color: color border: #black;
  }
}
```

color **attribute**
for the *people* species
of type rgb

Segregation model 1:
5. Define the color of each agent

- **To do:** each agent is displayed with a color (red or yellow) that characterize it
- **Hints:** each people agent will be characterized by a color value, which is initialized to a random color (among red and yellow).

- **Solution:**
  - Add a color attribute to the people species
  - **Initialize it to a random color value among red and yellow.**
  - Use the color in the display

Add the color attribute and initialize it	People

```
species people {

    rgb color;

    init {
        if( flip(0.5) ) {
            color <- #red;
        } else {
            color <- #yellow;
        }
    }

    aspect asp_circle {
        draw circle(1.0) color: color border: #black;
    }
}
```

Init block can be used to initialize the agent when it is created

Initialize randomly color to red or yellow With equal probability

Segregation model 1:
5. Define the color of each agent

- **To do:** each agent is displayed with a color (red or yellow) that characterize it
- **Hints:** each people agent will be characterized by a color value, which is initialized to a random color (among red and yellow).
- **Solution:**
  - Add a color attribute to the people species
  - Initialize it to a random color value among red and yellow.
  - **Use the color in the display**

Add the color attribute and initialize it	People

```
species people {

    rgb color;

    init {
        if( flip(0.5) ) {
            color <- #red;
        } else {
            color <- #yellow;
        }
    }

    aspect asp_circle {
        draw circle(1.0) color: color border: #black;
    }
}
```

color **attribute**
Is used in the **aspect**

The GAML Corner: definition of a species

- **4 kinds of elements** can be defined in a species:
  - The internal state of the agents of this species (**attributes**).
  - Their capabilities (**action**): blocks that will be executed only when called.
  - Their behavior (**reflex**): blocks that will be executed at each step.
  - Their way of being displayed (**aspect**).
- In addition, an unique **init** block can be used to initialized agents at their creation.

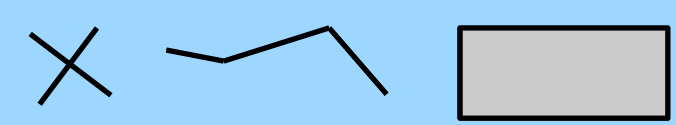
- Note: **global**, **grid**, **experiment** are kinds of species and have the same structure.

General structure of a species	Species

```
species my_species {
    string a_variable;

    init { }
    action my_action { }
    reflex my_behavior { }
    aspect my_aspect { }
}
```

All GAMA agents are provided with some ***built-in attributes*** :

- ***name*** *(string)*
- ***shape*** *(geometry)*



- ***location*** *(point)* : centroid of its shape

The GAML corner: Init block

- For each species, an init block can be defined
- It allows to execute a sequence of statements at the creation of the agents
- Activated only once when the agent is created, after the initialisation of its variables, and before it executes any reflex

- Only one instance of init per species

```
global {
    ............
    //Only executed when world agent is
created
    init {
            write "Executing initialisation";
        }
}
```

GAML: declaration of an attribute

- **General declaration of a variable:** `data_type a_variable;`

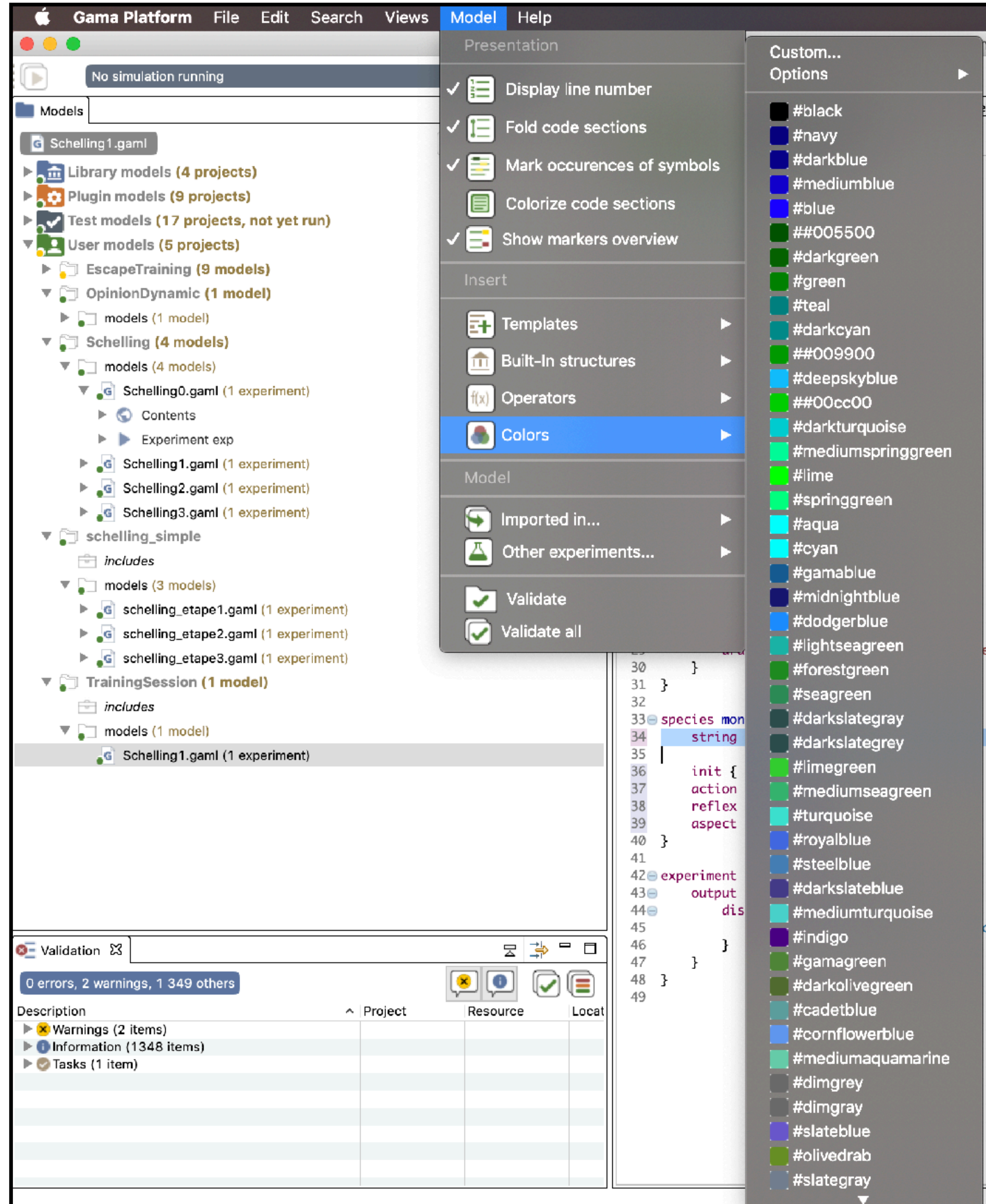
- The data_type describes the kind of data stored in the variable. It can be:
  - int (integer), float, string, bool (boolean value, i.e. that can be only true or false), point, list, pair, map, file, matrix, species name, rgb (for the colors), graph, path…

- **Additional facets:**
  - **<-** : (initial value),
  - **update:** (value computed at each simulation step),
  - ->: (value computed each time it is called),
  - **min:** (minimum value, if the value should become lower than the min, it is set to the min value).
  - **max**

```
species people {
   rgb color <- #red;
   int age <- 1 min:1 max: 120 update: age + 1;
}
```

The GAML corner: built-in constants

- GAML provides a set of built-in constants, starting with **#**
  - **colors**: #red, #yellow, #darkgrey…
  - **units**: #s, #h, #mn, # day, #m, #km…
  - **mathematical**: #pi, #e, #infinity…
  - **Graphical units:** #zoom, #camera_location



The GAML corner: operators

- Whereas **statements** are commands or declaration, **operators** are functions that compute a value on one or several operands.

- Unary operators are written:
  - operator(operand1)
- Binary operators are written:
  - Op1 operator Op2
  - operator(Op1, Op2)
- When there are more than 2 operands:
  - Op1 operator(Op2, …)
  - operator(Op1, Op2, …)

Add the color attribute and initialize it	People

```
species people {

    rgb color;

    init {
        if( flip(0.5) ) {
            color <- #red;
        } else {
            color <- #yellow;
        }
    }

    aspect asp_circle {
        draw circle(1.0) color: color border: #black;
    }
}
```

Operator ***flip****, that computes randomly the value true or false with a given* ***probability***

Operator ***circle*** *computes a circle geometry with a given* ***radius***

Back to the model

- **Notes:**
  the three following ways of initializing color are equivalent in this case.

Add the color attribute and initialize it	People

```
species people {
    rgb color;

    init {
        if( flip(0.5) ) {
            color <- #red;
        } else {
            color <- #yellow;
        }
    }
}
```

Add the color attribute and initialize it	People

The operator **? :**

Condition ? valueIfTrue : valueIfFalse

```
species people {
    rgb color;

    init {
        color <- (flip(0.5) ? #red : #yellow);
    }
}
```

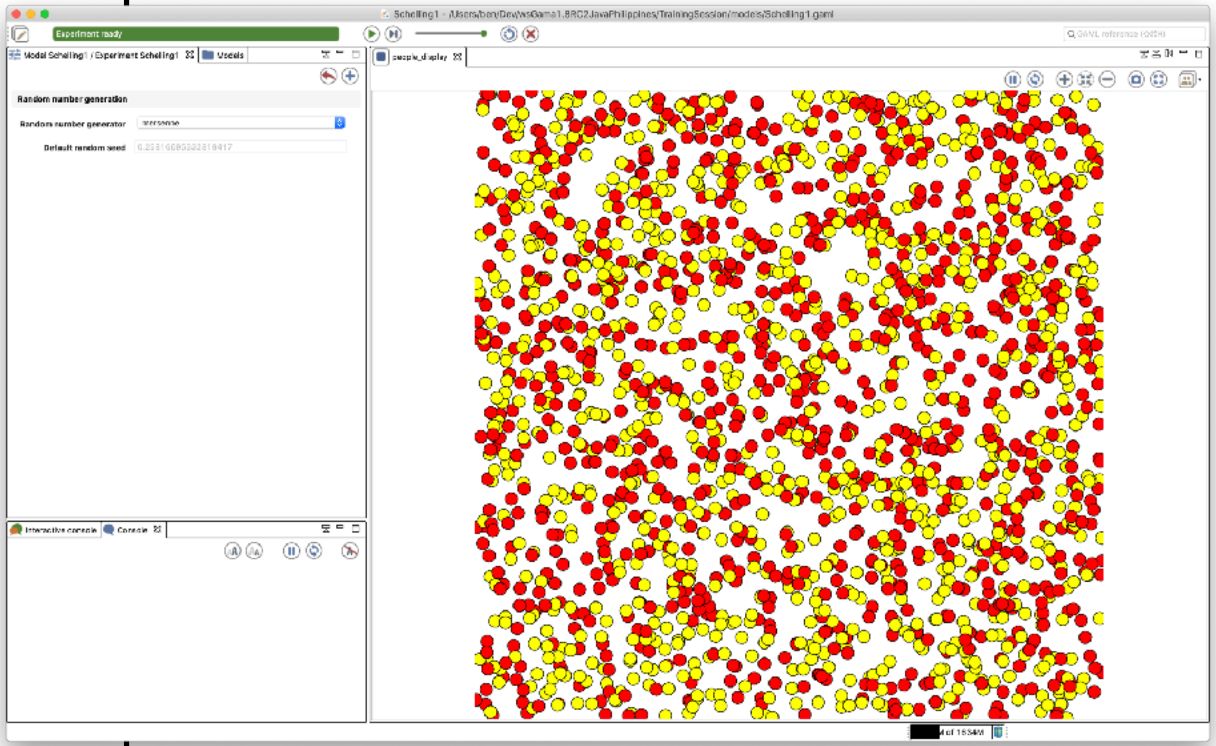
Add the color attribute and initialize it	People

```
species people {
    rgb color <- (flip(0.5) ? #red : #yellow);
}
```

Summary of the model 1

Summary of model 1	Model

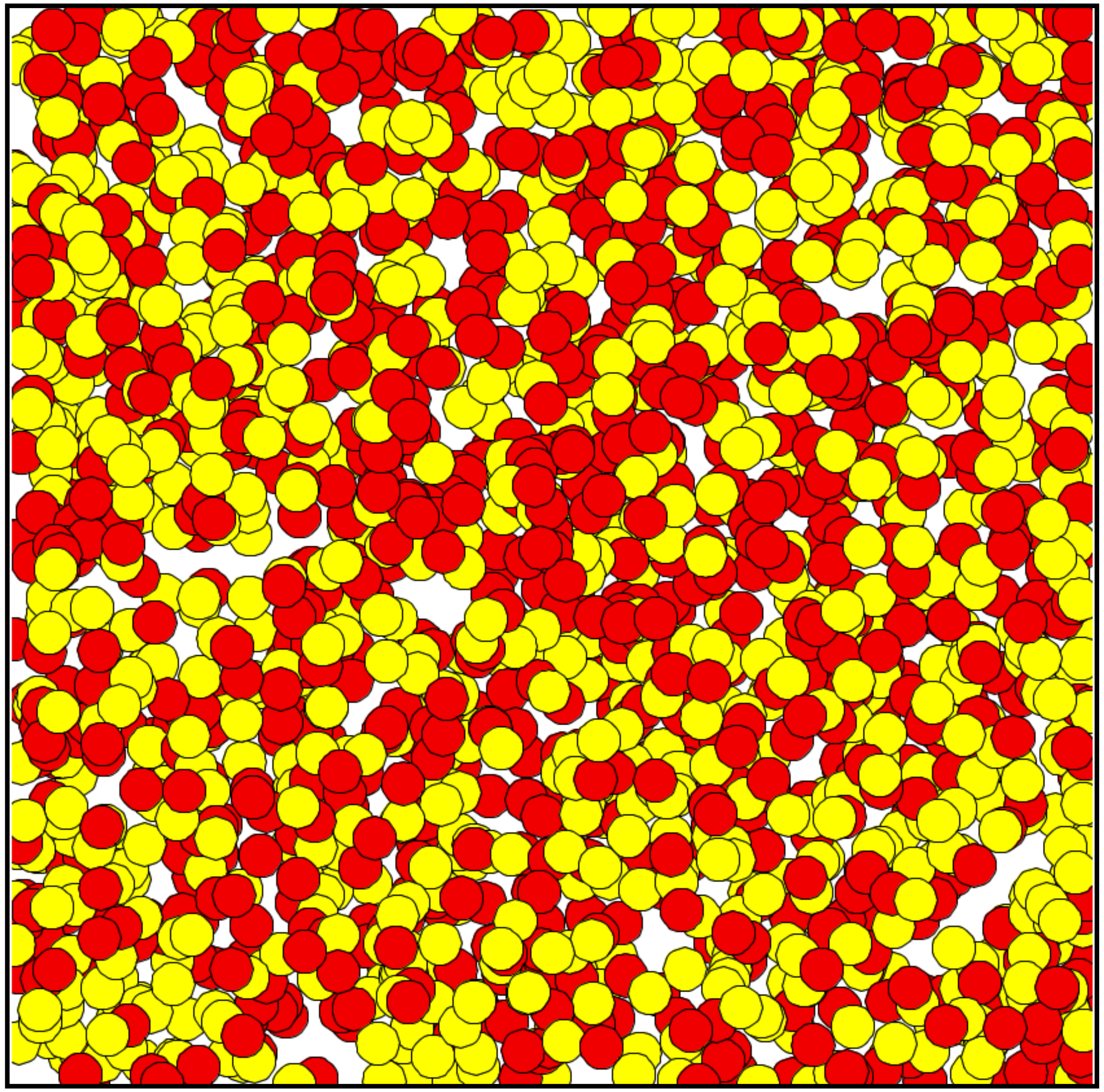
```
model Schelling1

global {
    init {
        create people number: 2000;
    }
}

species people {

    rgb color <- (flip(0.5) ? #red : #yellow);

    aspect asp_circle {
        draw circle(1.0) color: color border: #black;
    }
}

experiment Schelling1 type: gui {
    output {
        display people_display {
            species people aspect: asp_circle;
        }
    }
}
```



Try to run the simulation!

Step 1.5 (dummy model): introduce agent move

- Objectives:
  - ➡ Definition a random move of people agents at each simulation step



Segregation model 1.5: move
Step 1. Define a random move for agents

- **To do:** define a random move behavior for the people agents
- Hints: for an agent to move is simply to change its location.

- **Solution:**

Define a move behavior	People

```
species people {
    rgb color <- (flip(0.5) ? #red : #yellow);

    reflex move {
        location <- any_location_in(world.shape);
    }

}
```

Segregation model 1.5: move
Step 1. Define a random move for agents

- **To do:** define a random move behavior for the people agents
- Hints: for an agent to move is simply to change its location.

- **Solution:**

Reflex *is used to define a behavior*

Define a move behavior	People

```
species people {
    rgb color <- (flip(0.5) ? #red : #yellow);

    reflex move {
        location <- any_location_in(world.shape);
    }

}
```

Compute a random location in a geometry

world is the global agent. Its shape is the spatial environment in which agents are located.

Segregation model 1.5: move
Step 1. Define a random move for agents

- **To do:** define a random move behavior for the people agents
- Hints: for an agent to move is simply to change its location.

- **Solution:**

Reflex is used to define a behavior

Define a move behavior	People

```
species people {
    rgb color <- (flip(0.5) ? #red : #yellow);

    reflex move {
        location <- any_location_in(world.shape);
    }

}
```

Compute a random

world is the global agent. Its shape is the spatial environment in which agents are located.

Note 1: all GAMA agents are provided with a **location** attribute that defined the coordinate of the centroid of its shape. When the location is modified, the shape of the agent is translated to the new location and when the shape is modified, the location is re-computed.

The GAML Corner:
Species are provided with a simple behavioural structure, based on reflexes (*what they actually do*)

- A **reflex** is a sequence of statements that can be executed, at each time step, by the agent.

```
reflex name when: condition{
     [statements]
  }
```

- If no facet **when** are defined, it will be executed every time step.
- If there is one, it is executed only if the boolean expression evaluates to true.

- Several reflex blocks can be defined in each species. Each will be executed at each simulation step.

Note: The init block is a specific reflex that is activated only once at the creation of the agent

Segregation model 1.5: move
Step 1. Define a random move for agents

- **To do:** define a random move behavior for the people agents
- Hints: for an agent to move is simply to change its location.

- **Solution:**

Reflex *is used to define a behavior*

Define a move behavior	People

```
species people {
    rgb color <- (flip(0.5) ? #red : #yellow);

    reflex move {
        location <- any_location_in(world.shape);
    }

}
```

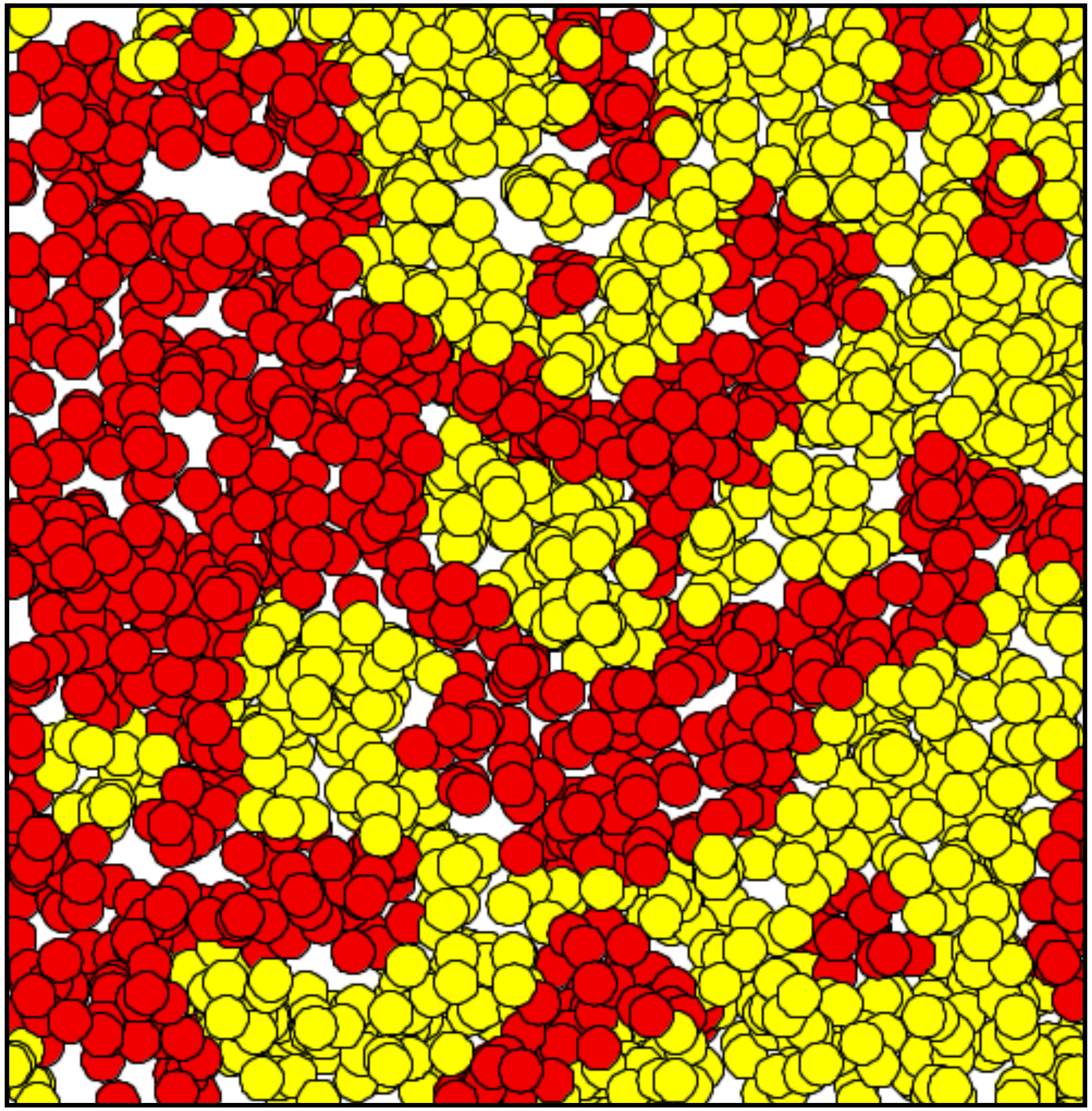
Compute a random location in a geometry

world is the global agent. Its shape is the spatial environment in which agents are located.

Run the simulation!

Step 2: definition of the people agent dynamics

- **Objectives:**
  - ➡ Definition of global variables (nb_people, similarity_rate, neighbour_distance…)
  - ➡ Definition of the neighbours attribute for the people agents
  - ➡ Definition of a computing neighbours similarity behaviour for the people agents
  - ➡ Definition of a moving behaviour for the people agents



Segregation model 2:
Step 1. Global attribute definitions

- In the following we need to compute:
  - the neighbourhood of each people agents, that is **the agents at a given distance**
  - The satisfaction in its neighborhood: that is the rate of agents of a different color compared to a rate of similarity wanted.

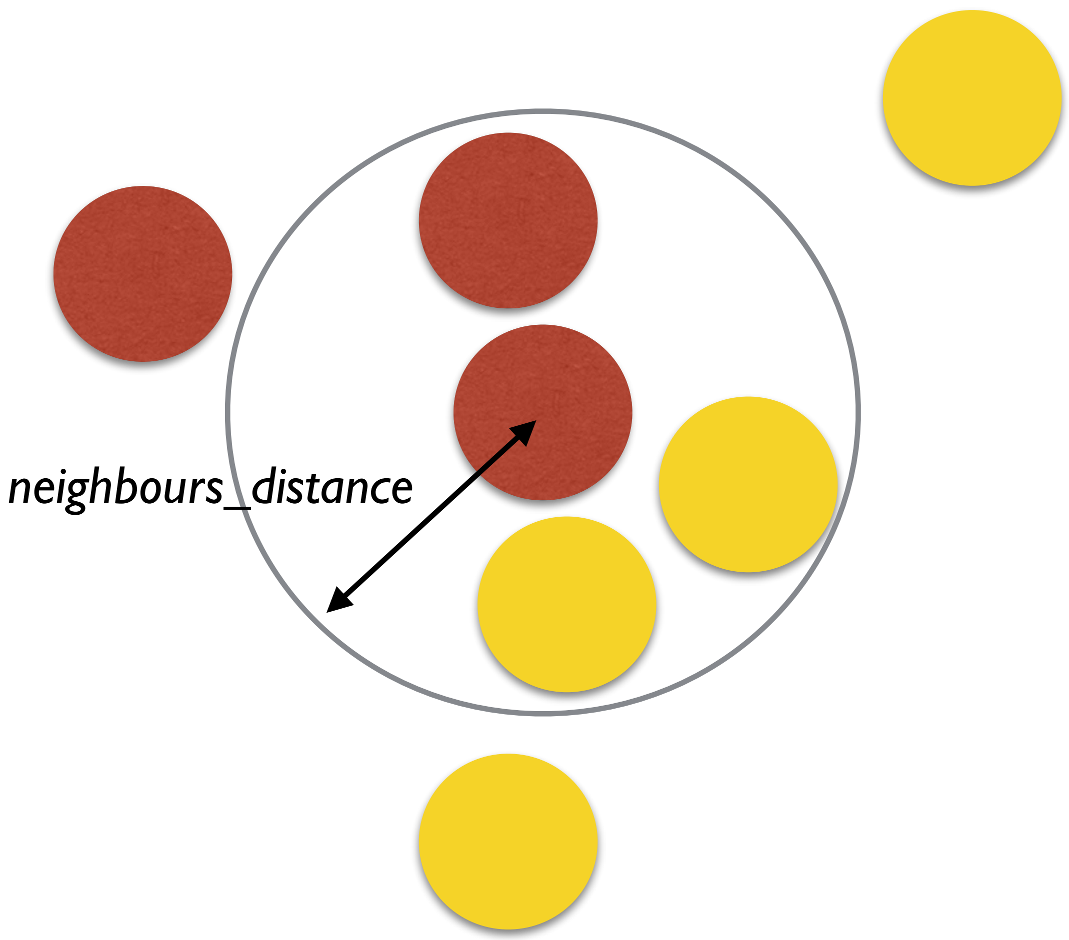
- These 2 values (**neighbours_distance** and **rate_similarity_wanted**) will have the same value for all the agents, we define thus them as global attributes.

Define global attributes	Global

```
global {
    float rate_similar_wanted <- 0.4;
    float neighbours_distance <- 5.0;

    init {
        create people number: 2000;
    }
}
```

Segregation model 2:
Step 2. Neighbours definition for people species

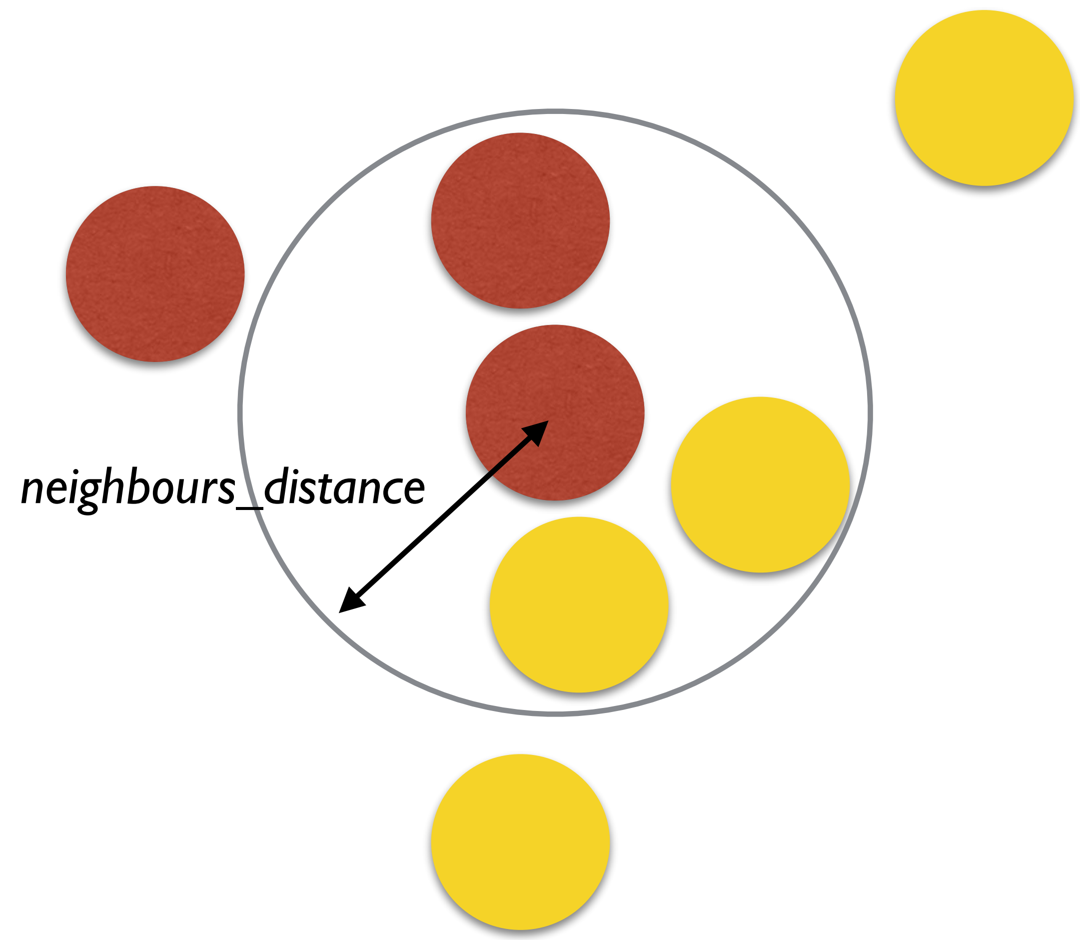
- **To do:** define an attribute for the people species called *neighbours* (containing the agents in the neighborhood) and compute its value.
  - **Type:** list of people agents;
  - **Value:** update at each simulation step with the people agents that are at a distance lower or equal to neighbours_distance



- **Solution:**

Segregation model 2:
Step 2. Neighbours definition for people species

- **To do:** define an attribute for the people species called *neighbours* (containing the agents in the neighborhood) and compute its value.
  - **Type:** list of people agents;
  - **Value:** update at each simulation step with the people agents that are at a distance lower or equal to neighbours_distance



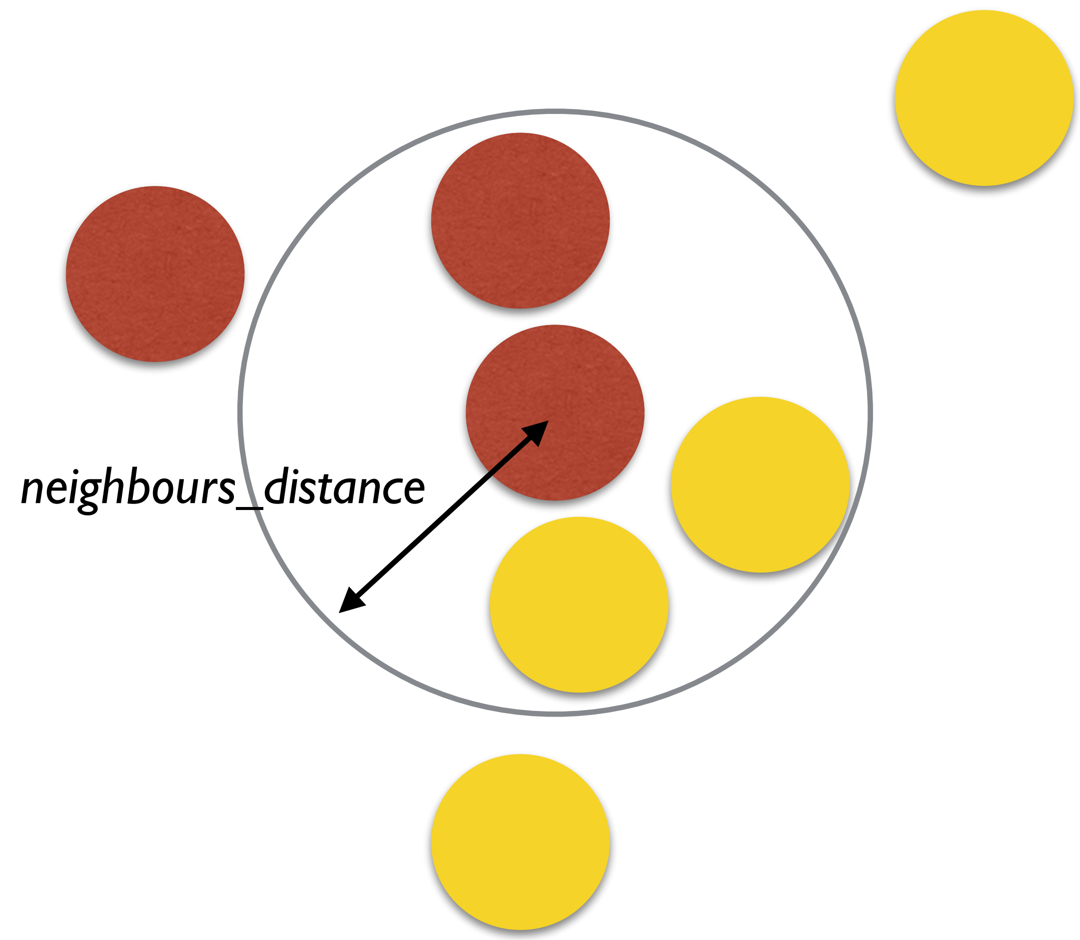
- **Solution:**

Define and compute neighborhood	People

```
species people {

   rgb color;


   list<people> neighbours update: people at_distance neighbours_distance;
  // Behavior and aspects
}
```

Segregation model 2:
Step 2. Neighbours definition for people species

- **To do:** define an attribute for the people species called *neighbours* (containing the agents in the neighborhood) and compute its value.
  - **Type:** list of people agents;
  - **Value:** update at each simulation step with the people agents that are at a distance lower or equal to neighbours_distance



- **Solution:**

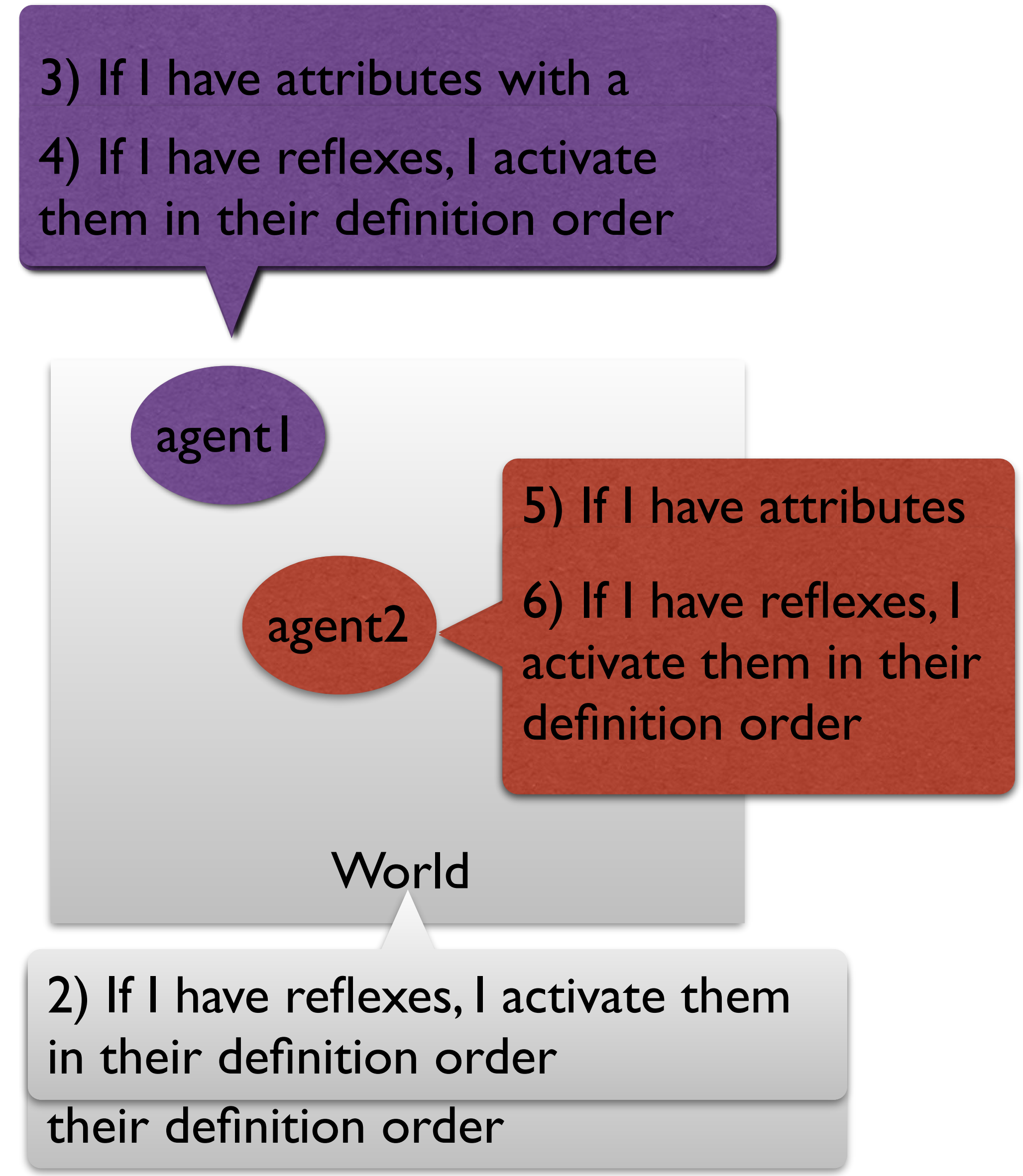
Define and compute neighborhood	People

```
species people {

   rgb color <- (flip(0.5) ? #red : #yellow);

   list<people> neighbours update: people at_distance neighbours_distance;
  // Behavior and aspects
}
```

Species **at_distance** *float_value*
computes the list of agents
of the given species at the given distance

The GAML corner: The scheduler of GAMA

- The basic scheduler of GAMA works as follows:
  - GAMA activates the world agent (global) then all the other agents according to their order of creation
  - **When an agent is executed**, first its update its attributes (facet **update** of the attributes), then it **activates its reflexes** in their definition order

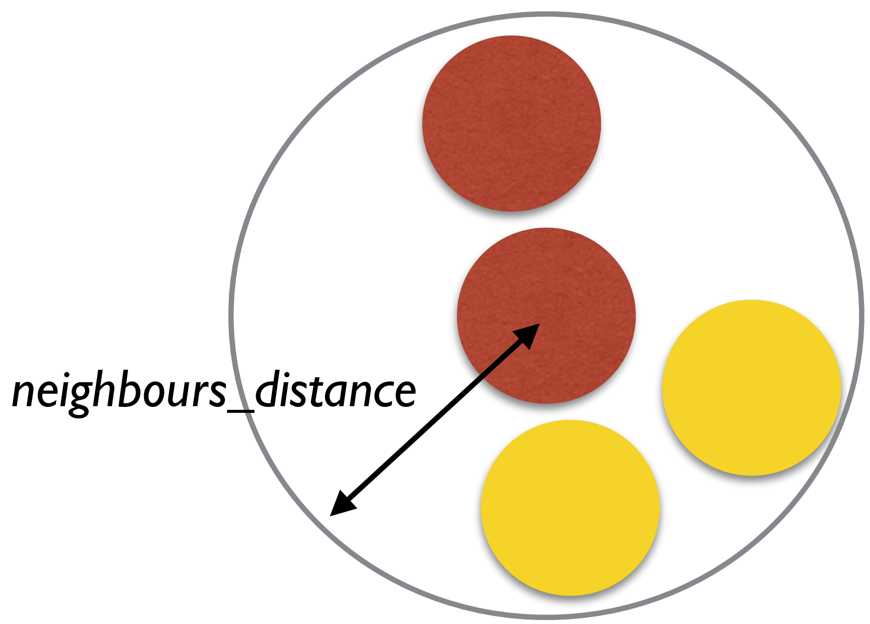


- Of course the Scheduler can be easily tuned through the GAML language:
  - modification of the order of activation of the agents (than can be dynamic)
  - Fine activation of the agents using actions (e.g.: agent1 executes a first action, then agent2 executes an action, then agent1 executes again another action….)

Note: GAMA offers some specific control architectures (finite state machine, tack-oriented architectures…) that can be added to species

Segregation model 2:
Step 3. Compute similarity rate and happiness level for the people species

- **To do:** define a reflex called computing_similarity for the people species:
  - if the neighbours is empty, set the rate_similar to 1.0
  - Otherwise, compute the **number of neighbours**, then the **number of neighbours with the same colour** as the agent, then set the rate_similar to the number of similar neighbours divided by the number of neighbours
  - Compute the happiness state of the agent (and store it in an attribute)



Similar_rate = 1/3 = 0.333
happy if similar_rate >= *similar_rate_wanted*

Segregation model 2:
Step 3. Compute similarity rate and happiness level for the people species

To do:

- define a reflex called computing_similarity for the people:
- if the neighbours is empty, set the rate_similar to 1.0
- Otherwise, compute the number of neighbours, then the number of neighbours with the same colour as the agent, then set the rate_similar to the number of similar neighbours divided by the number of neighbours
- Compute the happiness state of the agent (and store it in an attribute)

Compute of similarity	People

```
species people {
    // other attributes
    list<people> neighbours update: people at_distance neighbours_distance;
    bool is_happy <- false;

    reflex computing_similarity {
        float rate_similar <- 0.0;
        if (empty(neighbours)) {
            rate_similar <- 1.0;
        } else {
            int nb_neighbours <- length(neighbours);
            int nb_neighbours_sim <- neighbours count (each.color = color);
            rate_similar <- nb_neighbours_sim /nb_neighbours ;
        }
        is_happy <- rate_similar >= rate_similar_wanted;
    }
    //other reflex and aspect definition
}
```

Segregation model 2:
Step 3. Compute similarity rate and happiness level for the people species

To do:

- **define a reflex called computing_similarity for the people:**
- if the neighbours is empty, set the rate_similar to 1.0
- Otherwise, compute the number of neighbours, then the number of neighbours with the same colour as the agent, then set the rate_similar to the number of similar neighbours divided by the number of neighbours
- Compute the happiness state of the agent (and store it in an attribute)

Compute of similarity | People

```
species people {
    // other attributes
    list<people> neighbours update: people at_distance neighbours_distance;
    bool is_happy <- false;

    reflex computing_similarity {
        float rate_similar <- 0.0;
        if (empty(neighbours)) {
            rate_similar <- 1.0;
        } else {
            int nb_neighbours <- length(neighbours);
            int nb_neighbours_sim <- neighbours count (each.color = color);
            rate_similar <- nb_neighbours_sim /nb_neighbours ;
        }
        is_happy <- rate_similar >= rate_similar_wanted;
    }
    //other reflex and aspect definition
}
```

New reflex

Segregation model 2:
Step 3. Compute similarity rate and happiness level for the people species

To do:

- define a reflex called computing_similarity for the people:
- **if the neighbours is empty, set the rate_similar to 1.0**
- Otherwise, compute the number of neighbours, then the number of neighbours with the same colour as the agent, then set the rate_similar to the number of similar neighbours divided by the number of neighbours
- Compute the happiness state of the agent (and store it in an attribute)

Compute of similarity | People

```
species people {
    // other attributes
    list<people> neighbours update: people at_distance neighbours_distance;
    bool is_happy <- false;

    reflex computing_similarity {
        float rate_similar <- 0.0;
        if (empty(neighbours)) {
            rate_similar <- 1.0;
        } else {
            int nb_neighbours <- length(neighbours);
            int nb_neighbours_sim <- neighbours count (each.color = color);
            rate_similar <- nb_neighbours_sim /nb_neighb
        }
        is_happy <- rate_similar >= rate_similar_wanted;
    }
    //other reflex and aspect definition
}
```

*Definition of a **local variable** (same syntax as attribute), to store the similarity rate*

Note: **local variables** are variables that exist only inside a block. These variables are delete from the computer memory at the end of the block

```
type my_local_variable <- init_value;
```

Segregation model 2:
Step 3. Compute similarity rate and happiness level for the people species

To do:

- define a reflex called computing_similarity for the people:
- **if the neighbours is empty, set the rate_similar to 1.0**
- Otherwise, compute the number of neighbours, then the number of neighbours with the same colour as the agent, then set the rate_similar to the number of similar neighbours divided by the number of neighbours
- Compute the happiness state of the agent (and store it in an attribute)

Compute of similarity	People

```
species people {
    // other attributes
    list<people> neighbours update: people at_distance neighbours_distance;
    bool is_happy <- false;

    reflex computing_similarity {
        float rate_similar <- 0.0;
        if (empty(neighbours)) {
            rate_similar <- 1.0;
        } else {
            int nb_neighbours <- length(neighbours);
            int nb_neighbours_sim <- neighbours count (each.color = color);
            rate_similar <- nb_neighbours_sim /nb_neighbours ;
        }
        is_happy <- rate_similar >= rate_similar_wanted;
    }
    //other reflex and aspect definition
}
```

*The **empty** operator returns true is the list operand is empty*

Segregation model 2:
Step 3. Compute similarity rate and happiness level for the people species

To do:

- define a reflex called computing_similarity for the people:
- if the neighbours is empty, set the rate_similar to 1.0
- Otherwise, **compute the number of neighbours, then the number of neighbours with the same colour as the agent**, then set the rate_similar to the number of similar neighbours divided by the number of neighbours
- Compute the happiness state of the agent (and store it in an attribute)

Compute of similarity	People

```
species people {
    // other attributes
    list<people> neighbours update: people at_distance neighbours_distance;
    bool is_happy <- false;

    reflex computing_similarity {
        float rate_similar <- 0.0;
        if (empty(neighbours)) {
            rate_similar <- 1.0;
        } else {
            int nb_neighbours <- length(neighbours);
            int nb_neighbours_sim <- neighbours count (each.color = color);
            rate_similar <- nb_neighbours_sim /nb_neighbours ;
        }
        is_happy <- rate_similar >= rate_similar_wanted;
    }
    //other reflex and aspect definition
}
```

*The **length** operator computes the number of elements in a list*

Segregation model 2:
Step 3. Compute similarity rate and happiness level for the people species

To do:

- define a reflex called computing_similarity for the people:
- if the neighbours is empty, set the rate_similar to 1.0
- Otherwise, **compute the number of neighbours, then the number of neighbours with the same colour as the agent**, then set the rate_similar to the number of similar neighbours divided by the number of neighbours
- Compute the happiness state of the agent (and store it in an attribute)

Compute of similarity	People

```
species people {
    // other attributes
    list<people> neighbours update: people at_distance neighbours_distance;
    bool is_happy <- false;

    reflex computing_similarity {
        float rate_similar <- 0.0;
        if (empty(neighbours)) {
            rate_similar <- 1.0;
        } else {
            int nb_neighbours <- length(neighbours);
            int nb_neighbours_sim <- neighbours count (each.color = color);
            rate_similar <- nb_neighbours_sim /nb_neighbours ;
        }
        is_happy <- rate_similar >= rate_similar_wanted;
    }
    //other reflex and aspect definition
}
```

*The **count** operator computes the number of elements in the first operand (list, species) that fulfill the second operand condition*

Note: for list operators, the keyword ***each*** represents each element of the list

Segregation model 2:
Step 3. Compute similarity rate and happiness level for the people species

To do:

- define a reflex called computing_similarity for the people:
- if the neighbours is empty, set the rate_similar to 1.0
- Otherwise, compute the number of neighbours, then the number of neighbours with the same colour as the agent, then set the rate_similar to the number of similar neighbours divided by the number of neighbours
- **Compute the happiness state of the agent (and store it in an attribute)**

Compute of similarity	People

```
species people {
    // other attributes
    list<people> neighbours update: people at_distance neighbours_distance;
    bool is_happy <- false;

    reflex computing_similarity {
        float rate_similar <- 0.0;
        if (empty(neighbours)) {
            rate_similar <- 1.0;
        } else {
            int nb_neighbours <- length(neighbours);
            int nb_neighbours_sim <- neighbours count (each.color = color);
            rate_similar <- nb_neighbours_sim /nb_neighbours ;
        }
        is_happy <- rate_similar >= rate_similar_wanted;
    }
    //other reflex and aspect definition
}
```

Segregation model 2:
Step 4. People moves when they are not happy

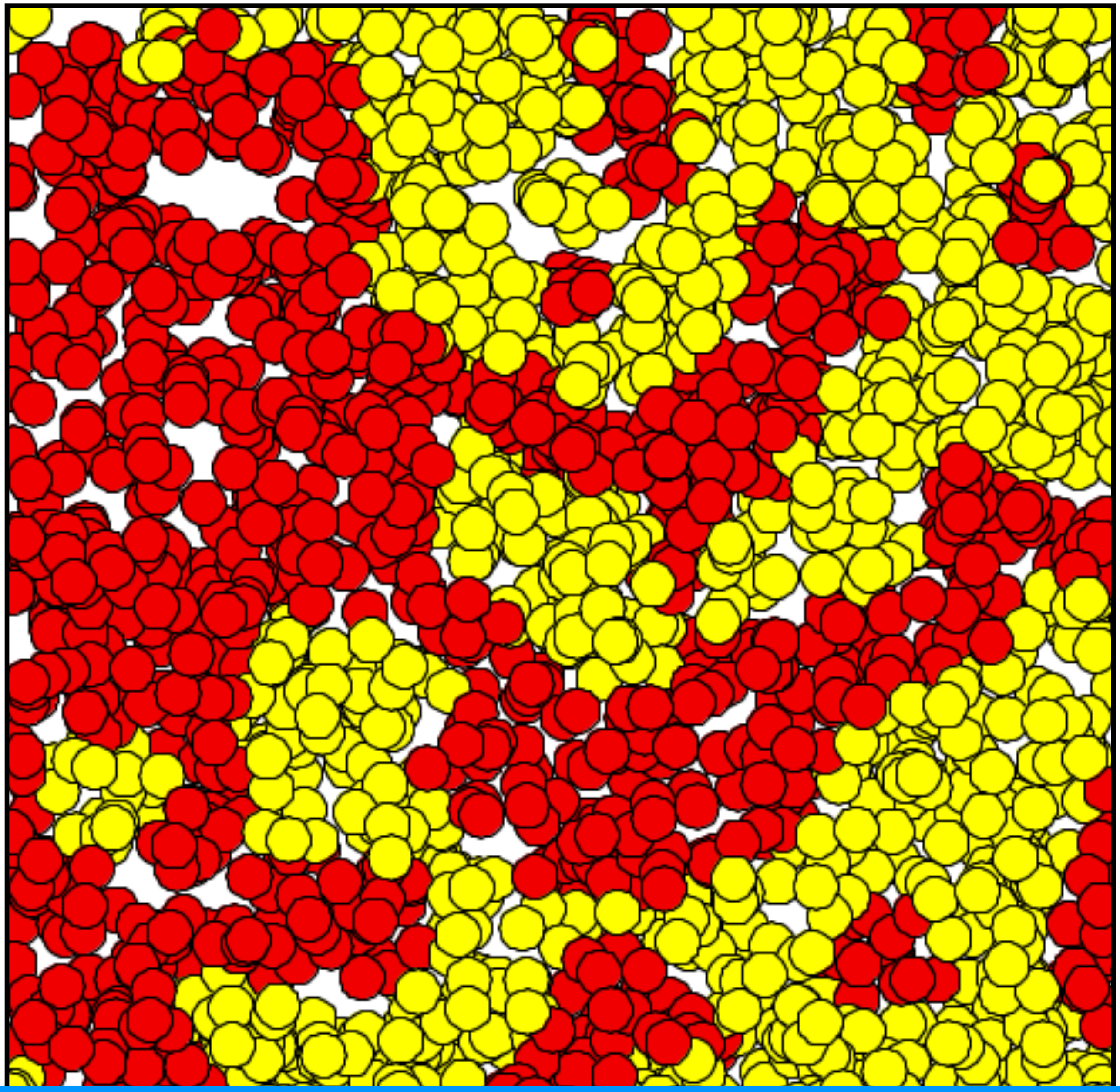
- **To do:**
  - activate the move reflex only if the agent is not happy (not is_happy)

- **Solution:**

Modify the move behavior	People

```
species people {

    reflex move when: not is_happy {
        location <- any_location_in(world.shape);
    }

}
```

End of step 2



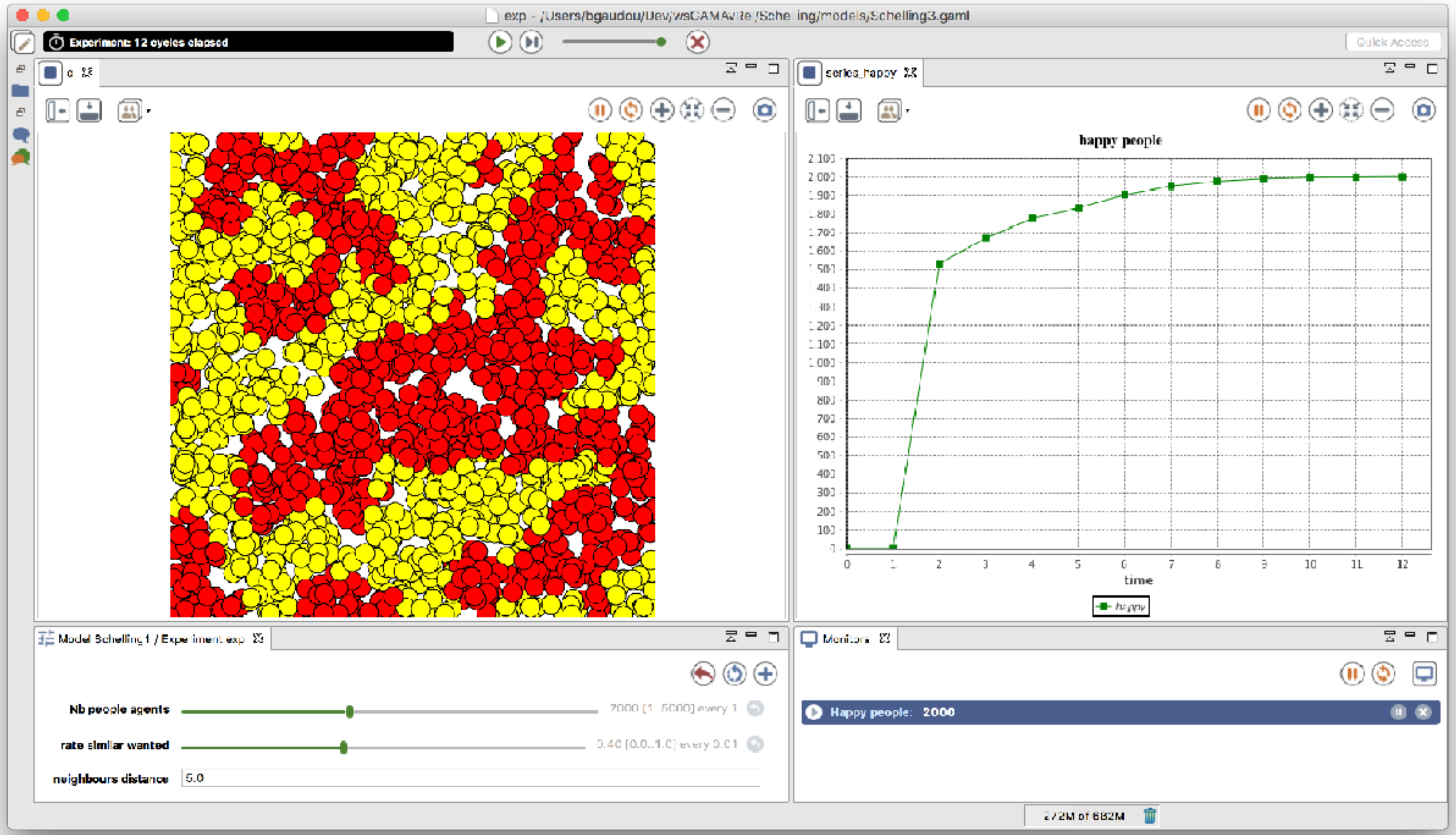
Run the model!

Try to change the number of people agents

Now it's time to define some new parameters and new outputs for the model !

Step 3: definition of new parameters and outputs

- **Objectives:**
  - ➡ Compute the total number of happy people and store it in a global variable
  - ➡ Definition of an ending condition (when all people are happy)
  - ➡ Definition of parameters
  - ➡ Definition of a new monitor to follow the number of happy people
  - ➡ Definition of a chart to follow the evolution of the number of happy people



Segregation model 3:
Step 1. Computation of the number of happy people

- **To do:**
  - define a global attribute called nb_happy_people:
  - **Type:** int;
  - **Value:** updated at each simulation step with the number of people agents that are happy

- **Solution :**

Compute the number of happy people	Global

```
global {
    // other attributes
    int nb_happy_people <- 0 update: people count each.is_happy ;
    //...
}
```

We are now able to know at every step the number of happy people
Next step: pause the simulation when all the people agents are happy!

For that we need to use the pause action of the world agent!

Segregation model 3:
Step 2. Stop the simulation

- **To do:**
  - define a global reflex called end_simulation:
  - It is activated only when everybody is happy (i.e. the number of happy people is equal to the number of people)
  - call the « pause » action of the world agents that pauses the simulation

- **Solution:**

Stop the simulation when everybody is happy	Global

```
global {
   //attributes and init

   reflex end_simulation when: nb_happy_people = length(people) {
      do pause;
   }
}
```

do is used to call an action
(here the built-in **pause** action)

The GAML corner: an action in GAML is a capability available to the agents of a species (*what they can do*)

- It is a block of **statements** that can be used and reused whenever needed. An action can accept arguments.

```
action simple_action {
    write "simple action";
}
```

write statement displays a message in the console

- An action can return a result (statement return).

```
return_type action_name (var_type arg_name,...)
 {
  [statements]
  [return value;]
}
```

Action that returns a value

```
int   sum (int a <- 100, int b) {
    return a + b;
}
```

- Some actions are directly available (built-in action, i.e. primitive) for all agents (e.g. die action) or to specific agents (pause action of the world agents)

The GAML corner: an action in GAML is a capability available to the agents of a species (*what they can do*)

- It is a block of **statements** that can be used and reused whenever needed. An action can accept arguments.

```
action simple_action {
    write "simple action";
}
```

write statement displays a message in the console

- An action can return a result (statement return).

```
return_type action_name (var_type arg_name,...)
 {
  [statements]
  [return value;]
}
```

Action that returns a value

Definition 2 arguments :
the first one named «a», type integer and default value is «100»;
the second named «b», type integer

Type of the returned value

```
int sum (int a <- 100, int b) {
    return a + b;
}
```

Return a value, and finish the action

- Some actions are directly available (built-in action, i.e. primitive) for all agents (e.g. die action) or to specific agents (pause action of the world agents)

The GAML Corner: Different ways to call an action in GAML

- Call a action that does not return any value:

```
do action_name(v1,v2);
```

- Call an action that returns a value:

```
my_var <- self action_name(arg1:v1, arg2:v2);
```

- Examples:

```
do action_simple;

int d <- self add(10,100);

int d <- self add(b:100);
```

Segregation model 3:
Step 3. Parameter definition

- **To do:** define 3 parameters:
  - attribute: nb_people, legend: « nb of people »
  - attribute: rate_similar_wanted, legend: « rate similar wanted », min: 0.0, max: 1.0
  - attribute: neighbours_distance, legend: « neighbours distance », step: 1.0

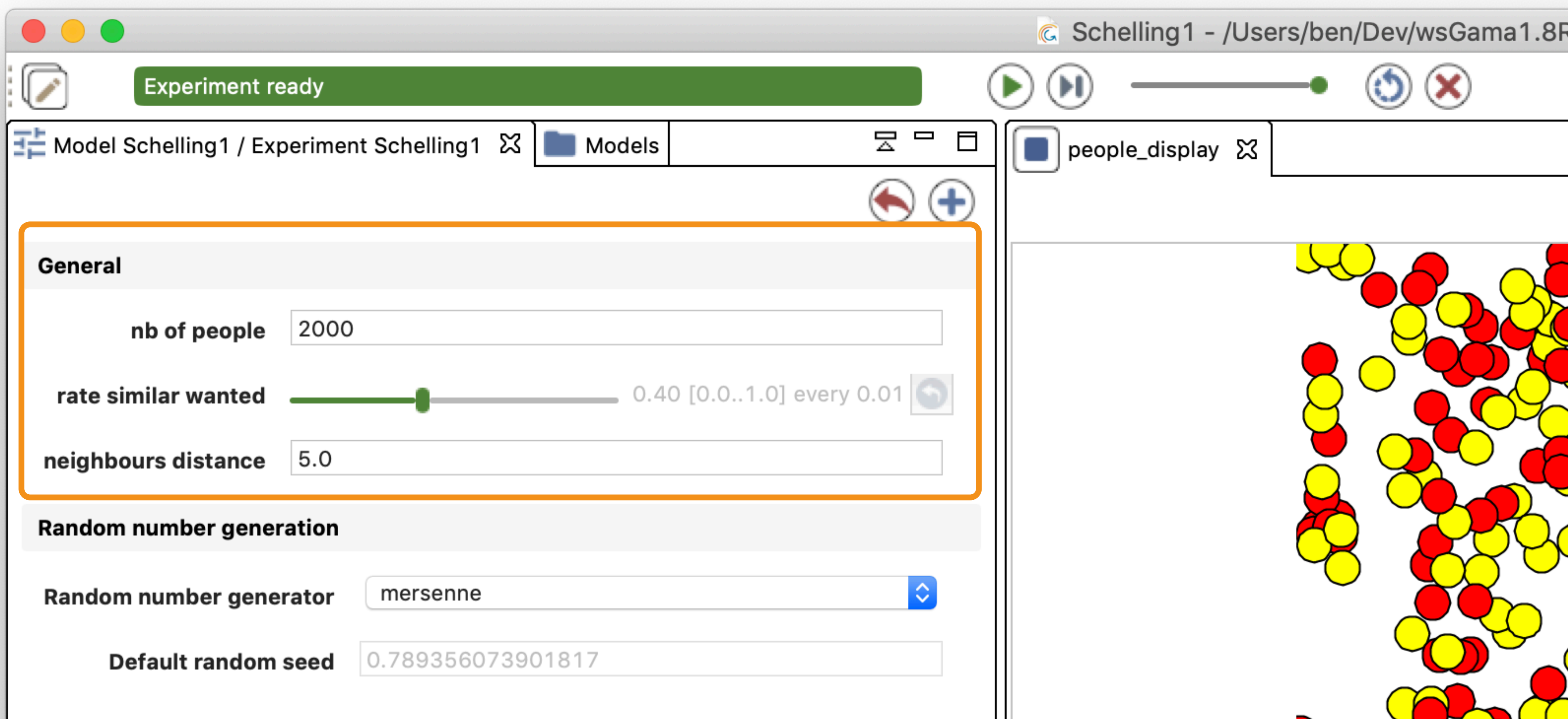
The GAML corner: parameter definition

- **Parameter (defined in the experiment block):**

```
parameter legend var: var_name category: my_cat;
```

- **Allow to give the user the possibility to define the value of a global attribute**
- *legend: string* to display
- *var_name*: **reference to a global attribute**
- *category*: *string* (use to better organise the parameters) - optional

Example:



Segregation model 3:
Step 3. Parameter definition

- **To do:** define 3 parameters:
  - attribute: nb_people, legend: « nb of people »
  - attribute: rate_similar_wanted, legend: « rate similar wanted », min: 0.0, max: 1.0
  - attribute: neighbours_distance, legend: « neighbours distance », step: 1.0
- **<u>Hints:</u> nb_people has first to be defined first as a global variable, before becoming a parameter.**

Define a global variable for the number of people	Global

```
global {
    int nb_people <- 2000;
    float rate_similar_wanted <- 0.4;
    float neighbours_distance <- 5.0;

    int nb_happy_people <- 0 update: people count each.is_happy ;

    init {
        create people number: nb_people;
    }

    reflex end_simulation when: nb_happy_people = nb_people {
        do pause;
    }
}
```

Segregation model 3:
Step 3. Parameter definition

- **To do:** define 3 parameters:
  - attribute: nb_people, legend: « nb of people »
  - attribute: rate_similar_wanted, legend: « rate similar wanted », min: 0.0, max: 1.0
  - attribute: neighbours_distance, legend: « neighbours distance », step: 1.0

- **Hints:** nb_people has first to be defined first as a global variable, before becoming a parameter.

Define parameters	Experiment

```
experiment Schelling1 type: gui {
   parameter "nb of people" var: nb_people;
   parameter "rate similar wanted" var: rate_similar_wanted min: 0.0 max: 1.0;
   parameter "neighbours distance" var: neighbours_distance step: 1.0;

   output {
      display people_display {
         species people aspect: asp_circle;
      }
   }
}
```

Segregation model 3:
Step 3. Parameter definition

- **To do:** define 3 parameters:
  - attribute: nb_people, legend: « nb of people »
  - attribute: rate_similar_wanted, legend: « rate similar wanted », min: 0.0, max: 1.0
  - attribute: neighbours_distance, legend: « neighbours distance », step: 1.0

- **Hints:** nb_people has first to be defined first as a global variable, before becoming a parameter.

Define parameters	Global

```
experiment Schelling1 type: gui {
    parameter "nb of people" var: nb_people;
    parameter "rate similar wanted" var: rate_similar_wanted min: 0.0 max: 1.0;
    parameter "

    output {
        display
            spec
        }
    }
}
```

The user can now modify the value of the global attributes through parameters

Try it !

(Do not forget to relaunch the simulation when needed)

Next step: definition of a monitor and a chart!

Segregation model 3:
Step 4. Monitor the number of happy people

- **To do:** define a monitor to follow the evolution of the number of happy people

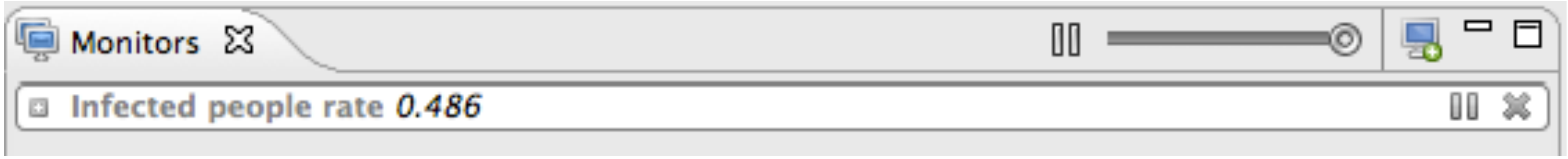
The GAML corner: monitor definition

- A monitor is an output allowing to display the current value of an expression
- The data to display have to be defined inside the output block:

```
monitor legend value: value
```

Example

```
experiment main_experiment type:gui{
    //...parameters
    output {
        monitor "Infected people rate" value: infected_rate;

    //...display
    }
}
```



Segregation model 3:
Step 4. Monitor the number of happy people

- **To do:** define a monitor to follow the evolution of the number of happy people

- **Answer:**

Define a monitor	Experiment

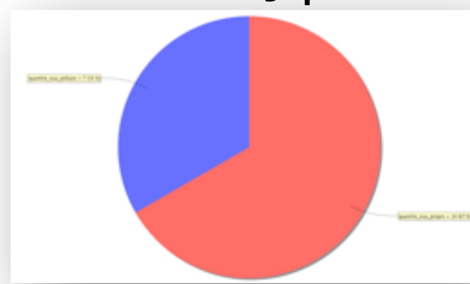
```
experiment Schelling1 type: gui {
    parameter "nb of people" var: nb_people;
    parameter "rate similar wanted" var: rate_similar_wanted min: 0.0 max: 1.0;
    parameter "neighbours distance" var: neighbours_distance step: 1.0;

    output {
        display people_display {
            species people aspect: asp_circle;
        }

        monitor "nb of happy people" value: nb_happy_people;
    }
}
```

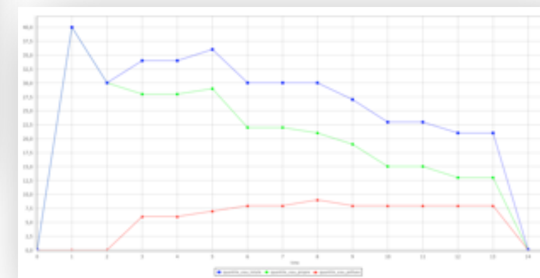
Segregation model 3:
Step 5. Plotting the number of happy people

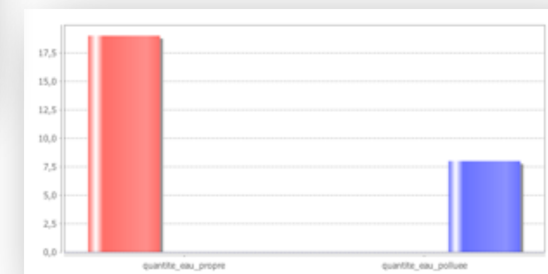
- **To do:** define a chart in a new display called *display_chart* to follow the evolution of the number of happy people.
  - chart name: « evolution of the number of happy people », type: series
  - data: "nb of happy people", value: nb_happy_people, color: green

The GAML Corner: chart definition (in experiment block)

- GAMA allows to display several type of charts :
  - Pie
  - Series
  - Histogram
  - XY chart







- A chart is a layer in a display:

  `chart legend type: chart_type`

- The data to display have to be defined inside the chart block:

  `data legend value: value color: colour`

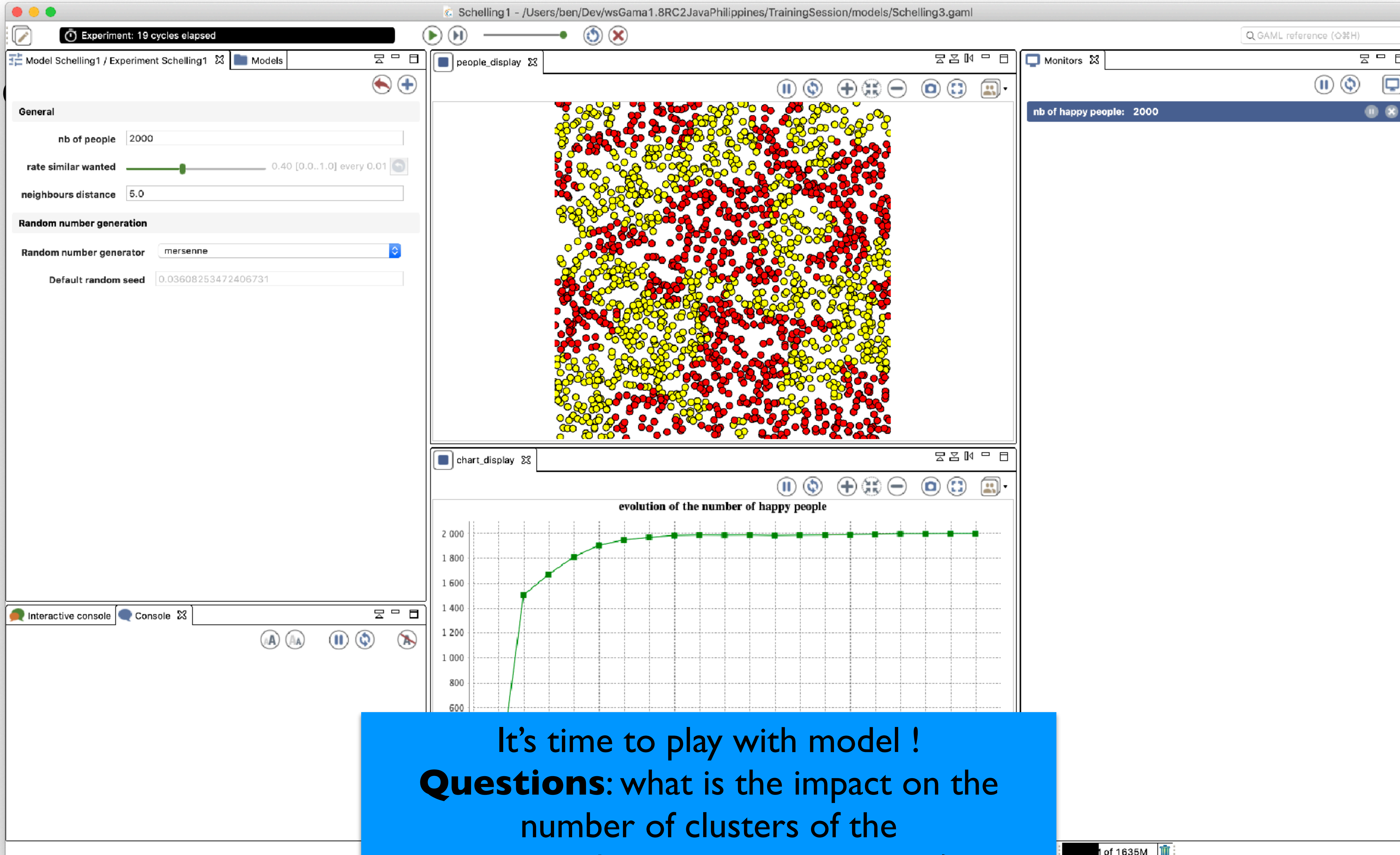
Segregation model 3:
Step 5. Plotting the number of happy people

- **To do:** define a chart in a new display called *display_chart* to follow the evolution of the number of happy people.
  - chart name: « evolution of the number of happy people », type: series
  - data: "nb of happy people", value: nb_happy_people, color: green
- **Answer:**

Define a chart	Experiment

```
experiment main_xp type: gui {
   // parameter definition

   output {
      // display monitor
      // map display definition
      display chart {
         chart "evolution of the number of happy people" type: series{
            data "nb of happy people" value: nb_happy_people color: #green;
         }
      }
   }
}
```

End of step 3

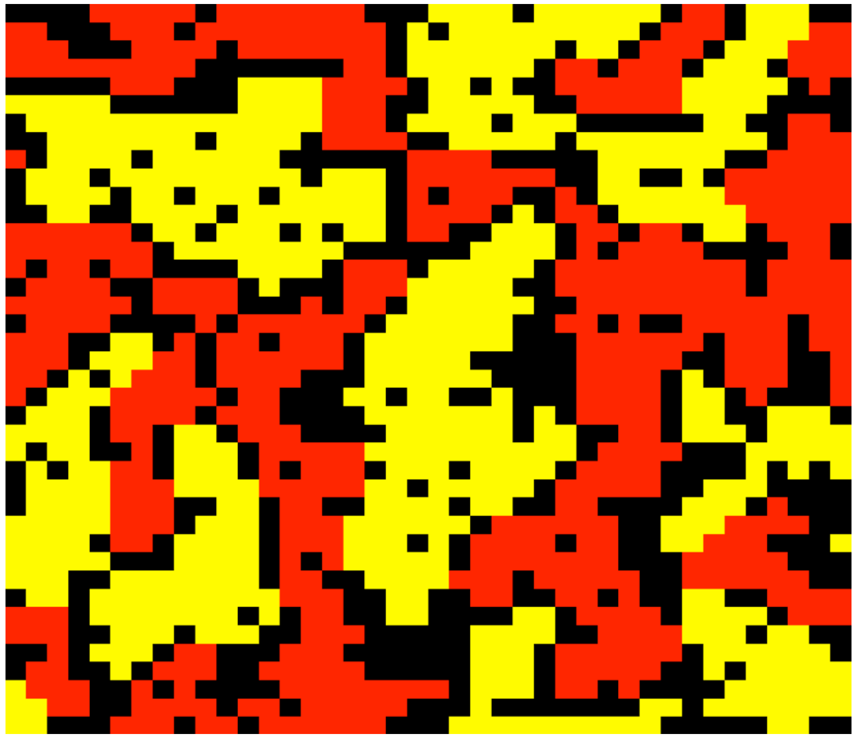


It's time to play with model !

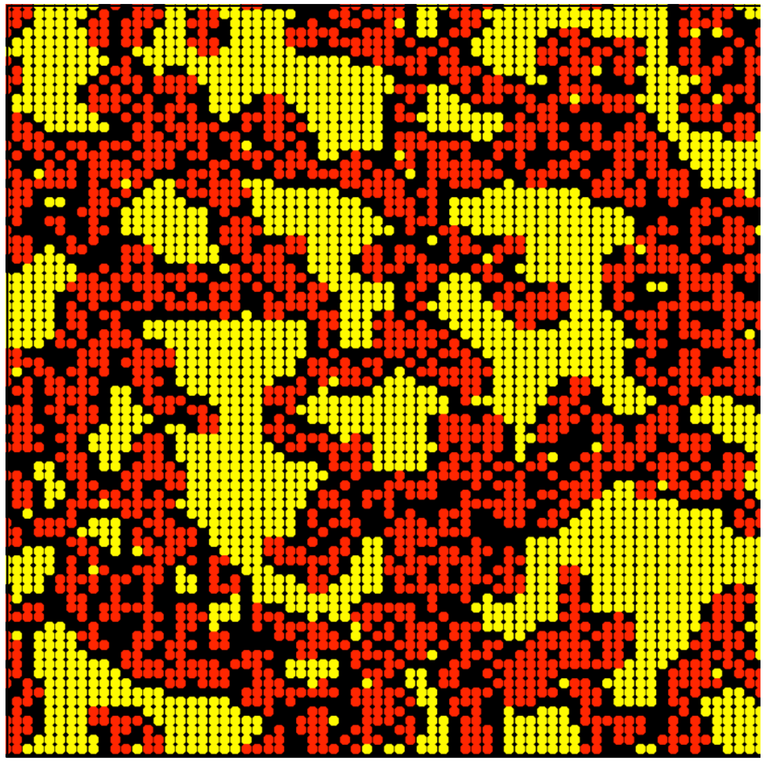
Questions: what is the impact on the number of clusters of the *rate_similarity_wanter* parameter?

Same question for the *neighbours_distance* parameter

Other implementations of the model are possible!



Cellular automata



Agents and grid

Agents and GIS