

Guide to Using the Graphical User Interface (GUI) Components

This document describes the GUI components I am providing. It explains what they do, and contains instructions for how to incorporate these components into your project. Remember, you are not obligated to use these components. You may design your own GUI if you wish.

Note: Throughout this document the words *node* and *square* mean the same thing. Both refer to a single square in the colony grid.

The Components

The components I am providing are contained in the [AntSimGUI.zip](#) file. The zip file contains 5 source files:

1. AntSimGUI
2. ColonyView
3. ColonyNodeView
4. SimulationEvent
5. SimulationEventListener

Add these classes to the other classes for your project. If you are using packages, you will need to add the package information to each of the source files.

The zip file also contains a folder of images that are required by the user interface components. You must copy this folder to the folder where your compiled class files are stored (e.g., your `classes` or `bin` directory) in order for the images to work.

Finally, the zip file also contains javadocs for the source files.

The Model-View-Controller Paradigm

When building an application it is useful to separate the application logic (the model) from the interface the user interacts with (the view). To enable the model and the view to be connected to each other, a controller is often used. For this project, there will be no controller, per se. The main point is to avoid having any code in your user interface that requires detailed internal knowledge of the model. The user interface should only have to rely on the public methods the model provides. The separation should be complete enough so that you can seamlessly swap out one user interface and replace it with another.

There are two modes in which model and view can communicate:

1. Pull Mode

In pull mode, the user interface must query the model for the current state. The data are *pulled* from the model, and displayed to the user. This mode is most useful when the state of the model does not change frequently, or the user is only interested in seeing periodic updates. Usually a button or key can be pressed, and the view makes the necessary requests and displays the result. The view can also periodically poll the model for state changes independently of user interaction. Depending on the polling interval (i.e., the time lapse between successive polls) some state changes may be missed. In order to use this mode, the view must have a reference to the model and must know what the public methods of the model are, in order to request the necessary data.

2. Push Mode

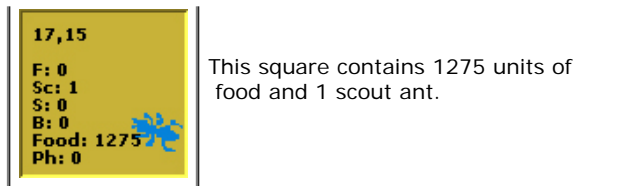
In push mode, the user interface is passive. It just sits there and waits for the model to tell it something has changed and tells it what to do to indicate the change to the user. The model *pushes* data to the view. This mode is useful when continuous updates are required. In this case it is not practical for the user to obtain the updates manually. In this mode, the view will always display all state changes, but it is possible for the model's state to change at a more rapid pace than the user can assimilate. For push mode the view does not need to know the public methods of the model, but the model must know the public methods of the view. Hence, the model must have a reference to the view.

The GUI I am providing operates using the push mode. Your model, therefore, will need to use the public methods of the GUI components to update the interface.

The ColonyNodeView Class

This class provides the view for an individual square in the colony. Most of the action occurs in this view. Every square, or node, in your colony will need to be associated with its own instance of the `ColonyNodeView` class. As the contents of a square change during the simulation, the square's associated `ColonyNodeView` instance should be informed. The following image illustrates one instance of the `ColonyNodeView` class:

	ID: 17,15 (x == 17, y == 15)
--	------------------------------



Several public methods are provided for this purpose. These methods are described below.

public ColonyNodeView()

This is the constructor for the class. Don't worry about creating and placing the components that display the ColonyNodeView's contents - these will automatically be created and laid out for you.

public void showNode()

Recall that until a scout ant has revealed the contents of a particular square, those contents should be hidden from view. When a scout ant opens up a new square, use this method to make the contents visible.

public void hideNode()

This method does the opposite of the `showNode` method. You should use this method when you first initialize your colony to hide the contents of all the squares except for the queen's square.

public void setID(String id)

This method sets the text for the ID that uniquely identifies each square. This text can be anything you want, but it should be small enough to fit within the square. As a suggestion, you might use the (row, column) or (x, y) coordinates that identify the square's location on the colony grid.

public void setQueen(boolean queenPresent)

Use this method to indicate a particular square contains the queen ant. The view for that square will have a different background color than the other squares in the colony.

public void setForagerCount(int numForagers)

Use this method to indicate how many forager ants are in a particular square. The result is displayed as the letter "F" followed by the number of forager ants present in the square.

public void setScoutCount(int numScouts)

Use this method to indicate how many scout ants are in a particular square. The result is displayed as the letters "Sc" followed by the number of scout ants present in the square.

public void setSoldierCount(int numSoldiers)

Use this method to indicate how many soldier ants are in a particular square. The result is displayed as the letter "S" followed by the number of soldier ants present in the square.

public void setBalaCount(int numBalas)

Use this method to indicate how many Bala ants are in a particular square. The result is displayed as the letter "B" followed by the number of Bala ants present in the square.

public void setFoodAmount(int food)

Use this method to indicate how many units of food are in a particular square. The result is displayed as the word "Food" followed by the number of food units present in the square.



public void setPheromoneLevel(int pheromone)

Use this method to indicate how many units of pheromone are in a particular square. The result is displayed as the letters "Ph" followed by the number of pheromone units present in the square.

Methods for showing and hiding ant icons

For each type of ant, you can display a colored ant icon in a square's view to indicate the presence of one or more ants of a particular type. For example, to show there are forager ants present, you would call the `showForagerIcon` method. If the number of forager ants falls to zero, you can hide this icon by calling the `hideForagerIcon` method. There is an analogous pair of methods for each of the different ant types. Using the ant icons makes it easier to follow where the ants in your colony are, and how they move. The table below shows the various ant icons. The icons themselves are located in an "images" folder in the AntSimGUI_2006.zip file. You need to copy this folder to the folder containing your compiled class files in order to use them.

	queen ant
	forager ant
	scout ant

	soldier ant
	Bala ant

Displaying the Pheromone Levels

The pheromone levels are displayed in two ways: 1) numerically, and 2) by the background color of the square. It can be tedious to identify where the pheromone trails are by looking at the numerical representation. It is much easier to do this by looking at the background colors of the squares. There are seven possible background colors a square can have depending on the concentration of the pheromone in the square. The possible colors and their meanings are shown below. Another advantage of using the different colors is that it makes it possible to distinguish heavily used trails from infrequently used trails.

Color	Pheromone Level
tan (or gray, if queen's square)	0
violet	$0 < \text{pheromone} \leq 200$
blue	$200 < \text{pheromone} \leq 400$
green	$400 < \text{pheromone} \leq 600$
yellow	$600 < \text{pheromone} \leq 800$
orange	$800 < \text{pheromone} \leq 1000$
red	$\text{pheromone} > 1000$

The ColonyView Class

The `ColonyView` class serves as the view for the entire colony. Basically, it is nothing more than a container for all the individual `ColonyNodeView` instances. The image below shows the `ColonyView` as it appeared at one point during an execution of the simulation.



There are only two methods in this class, both of which are described below.

```
public ColonyView(int colonyHeight, int colonyWidth)
```

This is the constructor for the class. You need to provide the height and width of your colony, in squares. Since the project requirements state the grid should be 27 squares by 27 squares, the colony height and width should both be 27.

```
public void addColonyNodeView(ColonyNodeView nodeView, int x, int y)
```

This method allows you to add a single ColonyNodeView instance to the ColonyView. You need to provide a reference to the ColonyNodeView instance being added, along with the x and y coordinates (in squares, not in screen pixels) of the square the ColonyNodeView instance represents. For example, to add the square that is at position (x,y) = (17,15) in the colony grid, x would be 17 and y would be 15.

The AntSimGUI Class

This class provides the main application window. It is based on Java's JFrame class. When you create an instance of AntSimGUI you will get a window that will be maximized to fill your screen. It contains two components: 1) a button control panel, and 2) the view of your ant colony.

The button control panel comes with a total of seven buttons, but only three of them are essential for running the simulation. The **Normal Setup** button should be implemented to initialize the simulation for a normal execution. You should click this button to set up the simulation before you run it. The four buttons, **Queen Test**, **Scout Test**, **Forager Test**, and **Soldier Test**, are provided as a way to initialize the simulation differently to test the individual ant types. Feel free to create other such buttons if you wish. You are not required to implement these buttons for the final project, but you may find them very useful during testing.

The **Run** button should be implemented to run the simulation continuously, non-stop. The **Step** button should be implemented to allow the user to step through the simulation one turn at a time. If you use this interface you are required to implement these two buttons.

There are four public method you need to use in this class. Each is described below.

```
public void initGUI(ColonyView colonyView)
```

This method is used to initialize the interface with the view of the entire ant colony, which is represented by the ColonyView class.

```
public void setTime(String time)
```

This method is used to set the current simulation time.

```
public void addSimulationEventListener(SimulationEventListener listener)
```

This method adds an event listener that will listen to events in the simulation. In this case, the events are the button click events for the buttons in the button control panel. You will need to decide which of your class(es) should be responsible for responding to the button clicks, and make that class(es) implement the `SimulationEventListener` interface, which is discussed later. This class should be one of your top-level classes.

```
public void removeSimulationEventListener(SimulationEventListener listener)
```

This method removes an event listener from the interface. You will probably have no need to use this method in your project. It is included simply as a matter of form (if you provide a way to add something, generally you also want to provide a way of removing it).

Using Events and Event Listeners

If the model has a reference to the view so that it can push information regarding state changes to the view, but the view does not have a reference to the model, how is it possible for the user to control the model via the view? The most common way to do this is to have the view broadcast important events, such as button clicks, and to have the model be a listener for those events. Whenever an event occurs, all the listeners are notified and can take whatever action is appropriate. In this way, the view maintains a shallow reference to the model. I call the reference shallow because even though the view has a reference to all of its event listeners, it only "knows" them as event listeners. The view does not have any knowledge of what class a particular listener is, or what its public methods are.

I have included a `SimulationEvent` class to represent simulation events, as well as an interface that can be implemented by any class that needs to be able to respond to a `SimulationEvent`. These are described below.

The `SimulationEvent` Class

This class represents an event that can happen in the simulation. For the purposes of the graphical interface, a `SimulationEvent` results whenever one of the buttons in the control panel is pressed. There are six types of events, one for each button. The public methods are described below.

```
public SimulationEvent(Object source, int eventType)
```

This is the constructor. You need to provide the `Object` on which the event occurred. You should be able to use the `this` keyword for the source. You also need to supply the type of event, which must be one of the six constant values in the class. Here is an example of how you would create a `SimulationEvent` for when the "Run" button is pressed:

```
SimulationEvent se = new SimulationEvent(this, SimulationEvent.RUN_EVENT);
```

```
public int getEventType()
```

This method returns the type of event. You will need to use this method in whichever class(es) you designate as `SimulationEvent` listeners (i.e., they implement the `SimulationEventListener` interface) in order to determine how to respond to the event.

The `SimulationEventListener` Interface

This interface must be implemented by whichever class(es) you designate as listeners for `SimulationEvents`. You should probably make one of your top-level classes the `SimulationEvent` listener. It will need to provide an implementation for the one method in the `SimulationEventListener` interface: `simulationEventOccurred(SimulationEvent simEvent)`. This method will contain the code for responding to the various events. You could implement the method using the following pattern:

```
public void simulationEventOccurred(SimulationEvent simEvent)
{
    if (simEvent.getEventType() == SimulationEvent.NORMAL_SETUP_EVENT)
    {
        // set up the simulation for normal operation
    }
    else if (simEvent.getEventType() == SimulationEvent.QUEEN_TEST_EVENT)
    {
        // set up simulation for testing the queen ant
    }
    else if (simEvent.getEventType() == SimulationEvent.Scout_TEST_EVENT)
    {
        // set up simulation for testing the scout ant
    }
    else if (simEvent.getEventType() == SimulationEvent.FORAGER_TEST_EVENT)
    {
        // set up simulation for testing the forager ant
    }
    else if (simEvent.getEventType() == SimulationEvent.SOLDIER_TEST_EVENT)
    {
        // set up simulation for testing the soldier ant
    }
    else if (simEvent.getEventType() == SimulationEvent.RUN_EVENT)
```

```

{
    // run the simulation continuously
}
else if (simEvent.getEventType() == SimulationEvent.STEP_EVENT)
{
    // run the next turn of the simulation
}
else
{
    // invalid event occurred - probably will never happen
}
}

```

Firing Events

The `fireSimulationEvent` method (shown below) in the `AntSimGUI` class broadcasts simulation events to all listeners, which are stored in a `LinkedList`.

```

/**
 * fire a simulation event
 *
 * @param eventType the type of event that occurred (see the
 * SimulationEvent class for allowable types)
 */
private void fireSimulationEvent(int eventType)
{
    // create event
    SimulationEvent simEvent = new SimulationEvent(this, eventType);

    // inform all listeners
    for (Iterator itr = simulationEventListenerList.iterator(); itr.hasNext(); )
    {
        ((SimulationEventListener)itr.next()).simulationEventOccurred(simEvent);
    }
}

```

Making Visual Updates Smoother

When running the simulation continuously, you may find the screen will not update until after the simulation has ended, or many turns have elapsed. You won't see this problem when using the Step button to run the simulation one turn at a time.

The problem here lies in the fact that Java updates its Swing components on a different thread than it uses for the main application. Swing components that have changed are sent to a queue for repainting on the *Event-dispatching thread*, while the main method that drives an application is controlled by the *Main thread*. Thus, there is a problem with synchronization. The Main thread runs the program, but the GUI lags behind. So, when you press the Run button to run the simulation continuously, the Main thread runs the simulation, but you typically won't see the GUI updated until the program ends, or until many turns have passed. execution of that method is complete.

I'm including a solution to this problem below. However, if you find it to be too complex to integrate this into your project, you may leave it out. I will not deduct any points from your project, *provided I am able to see what is going on by using the Step button.*

Solution

One solution to this problem is to make the application be controlled by the Event-dispatching thread, in order to keep the states of the simulation and the GUI synchronized. An easy way to do this is to use the `javax.swing.Timer` class. This class was designed specifically to control `ActionEvents` that are fired at regular intervals. To use the `javax.swing.Timer` class, you will need to do the following:

1. Declare an attribute of type `javax.swing.Timer` in your `Simulation` class (or whatever class you create to control the simulation, hereafter referred to as `Simulation`).
2. Make sure you have a method in your `Simulation` class that will start the simulation. You can name this method whatever you want, but for this example I will use the name `start()`.
3. Make sure you have a method in your `Simulation` class that will run *a single turn only*, of the simulation. This is **not** the same method referred to in Step 2 above. (*This is critical.*)
4. Make sure your `Simulation` class implements the `ActionListener` interface (e.g.: `public class Simulation implements ActionListener`).
5. In your `Simulation` class, implement the `actionPerformed(ActionEvent e)` method, which will become required when you make your `Simulation` class implement `ActionListener`.
6. In your implementation of the `actionPerformed(ActionEvent e)` method, you will need to set up a control statement. If the queen dies, the simulation must end immediately, so invoke the `stop()` method of your `Timer` attribute. Otherwise, invoke the method you created to run a single turn of the simulation (from Step 3).
7. Now, to tie everything together: In your `start()` method, instantiate your `javax.swing.Timer` attribute. It has only one constructor: `Timer(int delay, ActionListener listener)`. The `delay` argument is the length of time, in milliseconds, between consecutive firings of `ActionEvents`. The `listener` argument is the object that will respond to the `ActionEvents` fired by the timer. When you instantiate your timer:

- a. Pass a reasonable value for delay. To have the turns of your simulation occur in one-second intervals, use a value of 1000. If your delay is too short, the simulation may run too quickly for you to keep track of what is going on; too large a value will make you wait too long between turns.
 - b. Pass a reference to your Simulation class as the value for listener. Note that if you have followed these instructions to this point, you will be instantiating your timer from within your Simulation class, so you would use the `this` keyword as your reference to your Simulation class.
8. To start the `Timer`, you must invoke the `Timer`'s `start()` method after you instantiate the `Timer`. You will not need to use any other methods in the `javax.swing.Timer` class to get this to work.
 9. If you have problems getting this solution to work, email me.

Final Thoughts

You should be able to incorporate these components into your project without having to modify them. Which class(es) will implement the `SimulationEventListener` interface will depend on how you design your project. You will need to have some sort of "Node" or "Square" class that models an individual square in the colony grid. Each instance of this class will need to have a reference to a corresponding `ColonyNodeView` object. You will also need a class that serves as a container for all the individual square objects. This class will have a reference to the `ColonyView` class. One of your top-level classes will need to have a reference to an instance of the `AntSimGUI` class. If you have difficulties getting these components to work with your project, let me know and I will do what I can to help you.