Introduction to Turtlekit: A Platform for Building Logo Based Multi-Agent Simulations with Madkit

Fabien Michel

Laboratoire d'Informatique, de Robotique
et de Microélectronique de Montpellier

Unité Mixte de Recherche UMR 5506
CNRS - Université Montpellier II

RAPPORT DE RECHERCHE

Introduction to *TurtleKit*: A Platform for Building Logo Based Multi-Agent Simulations with MadKit

Fabien MICHEL
`fmichel@lirmm.fr`

06/2002 RR LIRMM 02215

161 rue Ada – 34392 Montpellier Cedex 5 – France
Tél : (33) 4 67 41 85 85 – Fax : (33) 4 67 41 85 00

TurtleKit

A Platform for Building Logo Based Multi-Agent Simulations with MadKit

Fabien MICHEL

This document describes the TurtleKit simulation model, a reactive agent execution tool that runs on the "synchronous engine" of the MadKit platform. This document contains two chapters: the first presents the simulation model and the tools provided by the platform, the second shows several examples of simulations built withTurtleKit.

TABLE OF CONTENTS

Chapter I

The TurtleKit Platform

1. Introduction

1.1. Simulation model overview

The simulation engine of TurtleKit provides tools for modelling, using and exploiting *multi-agents* simulations based on agents who evolve in a discretized world and act on it.

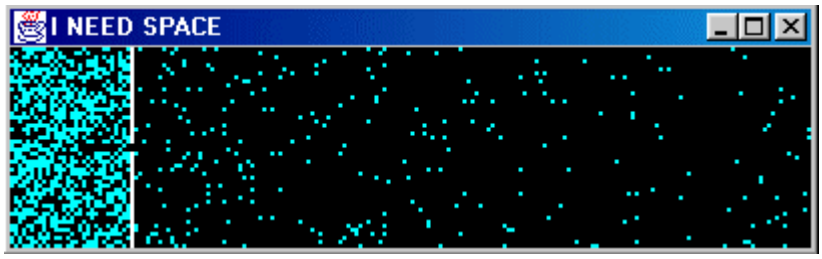

Figure 1: Gas simulation

It has been largely inspired by Logo based multi-agent platform such as the StarLogo system (http://education.mit.edu/starlogo/) or NetLogo (http://ccl.northwestern.edu/netlogo/).

The simulation model is a discrete 2D environment on which the so-called turtles (the simulated agents) evolve. The TurtleKit is a Java library with basic classes designed to quickly model, use and observe these kinds of simulation.

1.2. Motivations

Our primary goal when developing TurtleKit was not to do another turtle-based simulation engine. In fact, our main goal was to test the MadKit generic simulation module. This specialized engine is completely based on the synchronous engine (Watchers, Schedulers, Probes and Activators). From a simple test case, this library has grown to a full featured simulation model, so we now include it in the standard distribution. With TurtleKit we aim at bridging the gap between logo platforms like StarLogo or NetLogo, which are designed for newbie users, and simulation tools like Swarm (ww.swarm.org) that provide simulation libraries for advanced users using high level programming languages.

We see two parts in multi-agent simulation:

- Organize the agents' execution.
- Interpret or display the data produced by the simulation: system representations, database updates, agents' variable analysis...

Indeed, doing a *scheduler* job and a *watcher* job. These jobs requires to instantiate (explicitly or not) a multi-agents organization model which defines interactions between agents. As multi-agent organizations are the central concept of MadKit, building a MAS simulation simply consists in defining an appropriate Agent/Group/Role model.

1.3. "Mixing" Logo and MadKit

Although TurtleKit looks like a scaled-down StarLogo clone, the turtles, and all the agents of the simulation (launchers, observers, viewers) are actual MadKit agents written in Java. Knowing that the MadKit platform does not impose any constraint on the agent architecture, a turtle or an observer has an internal architecture that can be freely defined. Moreover, as MadKit messaging engine works using the Agent/Group/Role model, the agents of the simulation have the ability of communicating using messages with any other agent running in MadKit and not only with these of their own simulation. It is also possible to use all the usual functionalities of the platform and its tools like the *GroupObserver* system agent.

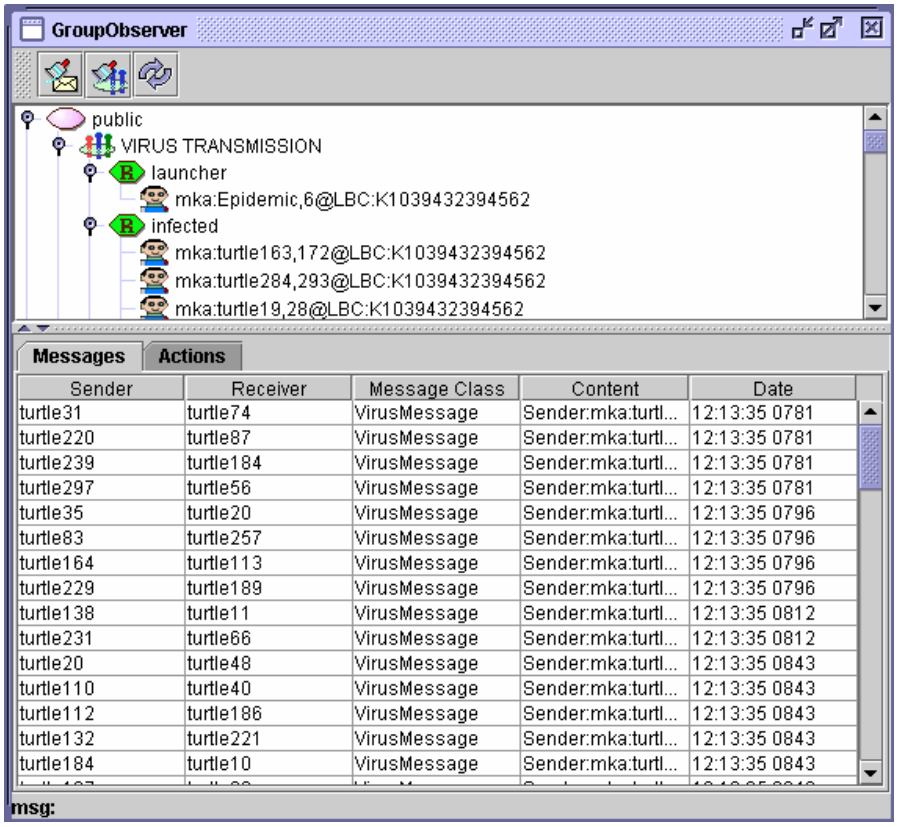

Figure 2: The GroupObserver agent of MadKit

2. The TurtleKit turtles

2.1. Logo programming approach

The Logo language, was designed in the late sixties at the Massachusetts Institute of Technology (MIT). The main motivation was to provide a learning tool that does not require programming skills while giving immediate graphical results. The principle is to manipulate a graphical animat, a turtle[1], typing simple commands to make it move and draw shapes on the screen. For instance the *forward 10* command makes the turtle moves forward 10 steps. Logo programs are usually collections of small procedures that define turtle behaviours which can be combined to achieve more complex behaviours. In multi-agent versions of Logo such as StarLogo, NetLogo and TurtleKit, a turtle lives on a 2D discrete world and is spatially located in a patch (the space unit). Turtles have the ability to locally interact with the environment, other turtles and patches, by changing environmental properties using turtle commands like `setPatchColor` for instance. In TurtleKit, a turtle can also leave an object on a patch (any java object).

2.2. Programming the turtles

In TurtleKit, defining a turtle consists in identifying its atomic behaviours, and coding them as Java methods. To setup the succession of these single behaviours, each method is required to return a String, which is the name of the next behaviour to be run at the next step. Inside each method, we can use tests, turtle commands, etc. The only requirement is to return this "next action" string. Thus, the global behaviour of a turtle is a small automaton, with transitions coded within the methods that return a string. For instance, a termite begins its life with the `searchForChip()` behaviour that will evolve into `findNewPile()` when a chip is found, and so on.

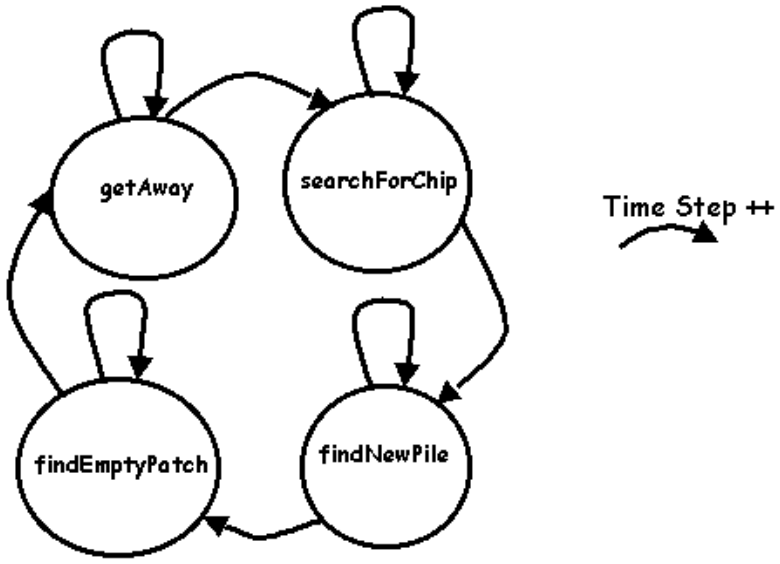

Figure 3: A Turtle's behaviour is defined as an automaton of atomic behaviours

[1] Originally a robotic creature that sat on the floor and could be directed to move around by typing commands at the computer.

The actual code of the `findNewPile()` method is:

```
Public String findNewPile()
{
  if (getPatchColor() == Color.yellow)
      return("findEmptyPatch");
  else
   {
      wiggle();
      return("findNewPile");
   }
}
```

Of course, we can use additional method to code `findNewPile()`:

```
void wiggle()
 {
       fd(1);
       turnRight(Math.random()*45);
       turnLeft(Math.random()*45);
 }
```

Note that a TurtleKit turtle is a (direct or not) subclass of the abstract class `Turtle` (`turtlekit.kernel` package). You must have the `turtlekit.jar` (as well as `madkit.jar`) archive in your `CLASSPATH` to compile new projects. So you have to import at least this class to write the code of a new turtle class. All the basic turtle commands are defined in this `Turtle` class.

```
import turtlekit.kernel.Turtle

public class Termite extends Turtle
{...
```

2.3. The turtle Constructor

The constructor is very important: it is where who have to define the procedure that will be done *before* the first simulation step. This is done by calling the constructor of the super class Turtle with, and only with, a String argument witch is the name of the desired procedure.

```
public Termite()
{
 super("getAway");
}
```

Of course it is also where you can initialize your own variables but a very important point is that you *cannot use a turtle command* in the constructor.

2.4. The setup method

Setup is an empty method of the class Turtle that you can override to initialize the agent. It is where the turtle can setup commands that will be executed only at the beginning of the simulation. It is the place where you should initialize the turtle properties (color, localization, played roles, variable initializations, etc...).

```
public void setup()
{
 playRole("termite");
 setColor(Color.red);
 setHeading(West);
// = setHeading(180);
}
```

2.5. Default values

By default the colour of a turtle is red and its heading is East (0).

2.6. The turtle Api

These variables and methods are the core part of the model. The turtles use them to act on their world.

Field Summary	
`static int`	**`East`** default direction values: setHeading(East) ~ setHeading(0)
`static int`	**`North`** default direction values: setHeading(East) ~ setHeading(0)
`static int`	**`NorthEast`** default direction values: setHeading(East) ~ setHeading(0)
`static int`	**`NorthWest`** default direction values: setHeading(East) ~ setHeading(0)
`static int`	**`South`** default direction values: setHeading(East) ~ setHeading(0)

static int	**SouthEast** default direction values: setHeading(East) ~ setHeading(0)
static int	**SouthWest** default direction values: setHeading(East) ~ setHeading(0)
static int	**West** default direction values: setHeading(East) ~ setHeading(0)

Method Summary

void	**activate**() Madkit kernel usage
void	**bk**(int nb) turtle move backward
int	**countTurtlesAt**(int a, int b) return the number of turtles in the patch situated at (a,b) units away
int	**countTurtlesHere**()
int	**createTurtle**(Turtle t) create a turtle at the creator position (xcor,ycor) returns the ID of the new turtle
double	**distance**(double a, double b) returns the distance from the patch (a,b).
void	**dropMark**(java.lang.String markName, java.lang.Object theMark) Drop a mark on the patch
void	**dropMarkAt**(java.lang.String markName, java.lang.Object theMark, int a, int b)
int	**dx**() return the x-increment if the turtle were to take one step forward in its current heading.
int	**dy**() return the y-increment if the turtle were to take one step forward in its current heading.
void	**end**() Madkit kernel usage
void	**fd**(int nb) turtle move forward
java.awt.Color	**getColor**()

double	**getHeading**() return the current heading of the turtle
boolean	**getHidden**()
java.lang.Object	**getMark**(java.lang.String variableName) get a mark deposed on the patch
java.lang.Object	**getMarkAt**(java.lang.String variableName, int a, int b)
java.awt.Color	**getPatchColor**()
java.awt.Color	**getPatchColorAt**(int a, int b) get the color of the patch situated at (a,b) units away
double	**getPatchVariable**(java.lang.String variableName) return the value of the corresponding patch variable
double	**getPatchVariableAt**(java.lang.String variableName, int a, int b) return the value of the patch situated at (a,b) units away
Turtle	**getTurtleWithID**(int a) return the Turtle with the specified ID, null if not alive
int	**getWorldHeight**()
int	**getWorldWidth**()
void	**giveUpRole**(java.lang.String role) the turtle will no longer play the specified role
void	**home**() teleport the turtle to the center patch
void	**incrementPatchVariable**(java.lang.String variableName, double value) set the value of the corresponding patch variable
void	**incrementPatchVariableAt**(java.lang.String variableName, double value, int a, int b)
boolean	**isMarkPresent**(java.lang.String markName) test if the corresponding mark is present on the patch (true or false)
boolean	**isMarkPresentAt**(java.lang.String markName, int a, int b) test if the corresponding mark is present on the patch situated at (a,b) units away
boolean	**isPlayingRole**(java.lang.String role)
void	**moveTo**(int a, int b) teleport the turtle to patch (a,b).
int	**mySelf**() return the turtle ID

void	**playRole**(java.lang.String role) one way to identify a kind of turtle: give them a Role in the simulation.
void	**randomHeading**()
void	**setColor**(java.awt.Color c)
void	**setHeading**(double direction) set the turtle heading to the value of direction
void	**setHidden**(boolean b) if true, the turtle hides itself (not draw)
void	**setPatchColor**(java.awt.Color c)
void	**setPatchColorAt**(java.awt.Color c, int a, int b) set the color of the patch situated at (a,b) units away
void	**setup**()
void	**setX**(double a)
void	**setXY**(double a, double b)
void	**setY**(double b)
java.lang.String	**toString**()
double	**towards**(double a, double b) returns direction to the patch (a,b).
void	**turnLeft**(double a)
void	**turnRight**(double a)
Turtle[]	**turtlesAt**(int a, int b) return turtles who are on the patch situated at (a,b) units away
Turtle[]	**turtlesHere**() return other turtles on the current patch
int	**xcor**()
int	**ycor**()

3. The Launcher agent

3.1. Synopsis

The launcher agent is the special TurtleKit agent that customizes and launches simulation. This agent GUI allows you to control the simulation process. To build a TurtleKit simulation, you'll have to write a subclass of Launcher and override some methods: the constructor, the `initializePatchVariables` method and the `addSimulationAgents` method. The use of these methods is explained below. The Launcher class is located in the turtlekit.kernel package.

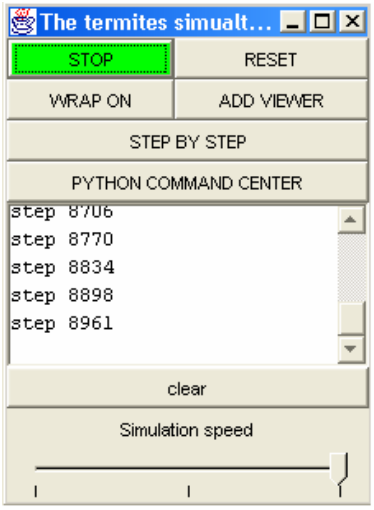

Figure 4: The simulation Launcher's default Gui

```
import turtlekit.kernel.Launcher

public class TermiteLauncher extends Launcher
{...
```

3.2. Constructor

In the constructor, you can modify different simulation properties like the world width, height or the default cell size display. You can also give the simulation a default name, which will be used to create the simulation group.

```
public Epidemic()
{
 setSimulationName("VIRUS TRANSMISSION");
 setWidth(60);
 setHeight(60);
 setCellSize(2);
}
```

If you're running the Launcher in the Desktop of, it will have its "property box". You may find here various options to control your simulation. We remind you that a property XXX is displayed when a getXXX() setXXX(...) couple of methods exists in the agent code (see section 3.4).

3.3. Launching simulation agents

The Launcher can launch three kinds of agents: turtles, observers and viewers. There is one special method for each kind. addTurtle, addObserver and addViewer.

```
public void addSimulationAgents()
{
  addViewer(3); // we choose a default viewer with a cell size of 3

  for (int i = 0; i < nbOfTermites; i++) //add the termites
    addTurtle(new Termite());

  // this method adds the PatchInitializer (an Observer) with no GUI
(false)
  addObserver(new PatchInitializer(nbOfTermites),false);
}
```

Only the Viewer has a default instantiation method. To add a turtle or an observer, you will have to define a subclass.

3.4. Add some patch variables

In some simulations (ants for example), you might need to define some patch variables in order to give turtles the capacity to mark their environment: by example the ants drops pheromone that other ants can smell and value the concentration. Moreover this pheromone might diffuse and evaporate itself.

To model this kind of simulation you will have to use the `PatchVariable` class to define this behavior. Once you have created a new `PatchVariable` object and set its properties with the methods of `PatchVariable`(`setEvapCoef`, `setDiffuseCoef` and `setDefaultValue`), add it to the simulation using `addPatchVariable` in the `initializePatchVariables` method of the Launcher:

```
protected void initializePatchVariables()
  {
       PatchVariable p = new PatchVariable("flavor");
       p.setDiffuseCoef(0.3153); //Optional
       p.setEvapCoef(0.025); //Optional
       p.setDefaultValue(32); //Optional
       addPatchVariable(p);
  }
```

Like any other MadKit agent in the Desktop gui of MadKit, a launcher has a properties window that allows simple customization from the user. Adding this kind of property simply consists in writing a `setPropertyand` `getProperty` couple of public methods.

```
int nbOfTermites=50;

public void setNbOfTermites(int n)
{
 nbOfTermites = n;
}

public int getNbOfTermites()
{
 return nbOfTermites;
}
```

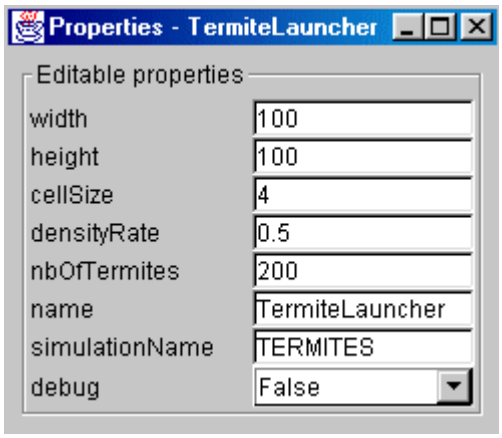

Figure 5: The property box of the Termite Launcher

In this example, these two methods permit to modify, directly in the properties Box, the `nbOfTermites` variable. Then it is possible to change this value before launch or reset the simulation. In this example the densityRate property was obtained by the same procedure.

4. The TurtleKit simulation Observer

4.1. Synopsis

An Observer is a kind of MadKit watcher specially designed for the TurtleKit simulations. This agent is extensible, and may be specialized to send messages to other agents, have a specific interface, write into a database, ... The turlekit observer initialize the world and observe it (patches+turtles). It has specially the capacity to access turtles. As it gets these references it can examine their properties or make them do some commands. In this example the Observer displays the number of turtles that are infected just by observing the turtles that play the role "infected": an easy way to give a closer look to some subset of simulated entities. To build an observer you have to create a subclass of the `Observer` class of the turtlekit package and optionally override some key methods: `setup`, `initializeTurtleTables` and `watch`.

```
import turtlekit.kernel.Observer

public class VirusObserver extends Observer
{...
```

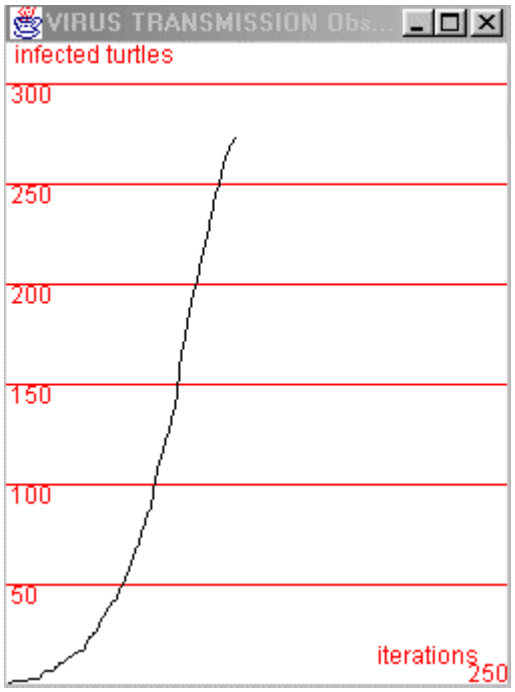

Figure 6: The virus observer displays customized model informations

4.2. The Observer class

A TurtleKit Observer has a direct access to patches using its public variables: `patchGrid` a Patch[][] array, `envWidth` and `envHeight` (the grid dimension). Moreover, it can access the turtles using arrays (Turtle[]) built through a role parameter. In order to create these arrays, you *must* initialize them in the `initializeTurtleTables` method using the `getTurtleWithRole(String role)` method that returns the corresponding Turtle array. So distinctions are possible as turtles use their `playRole` primitive. In the following example the `gasTable` is composed by every turtle who play the role of "gas" in the simulation. Note the arrays that have been setup through this mechanism are automatically updated by the simulation engine. So an array size always represents the number of alive turtles playing the corresponding role.

```
public class gasWatcher extends TurtleWatcher
{
  Turtle[ ] gasTable;

  public void initializeTurtleTables()
  {
    gasTable = getTurtleWithRole("gas");
  }
  ...
```

4.3. Initialization and observation: the `setup` and `watch` methods

To initialize the world (patch variables' values, colors...) you have to override the `setup` method. In the following example the PatchInitializer of the termites simulation makes the path color black (empty) or yellow (a chip).

```
public void setup()
{
 for(int i=0;i<envWidth;i++)
  for(int j=0;j<envHeight;j++)
    if (Math.random() < densityRate)
       patchGrid[i][j].setColor(Color.yellow);
    else
       patchGrid[i][j].setColor(Color.black);
}
```

To finally observe the world you have to override the `watch` method. This method will be executed at each step of the simulation engine.

```
public void watch()
{
 int cpt=0;
 for(int i = 0;i < turtleTable.length;i++)
 {
    if (turtleTable[i].xcor() > 10) cpt++;
 }
 println("there is "+cpt+" turtles on the right side (>10)");
}
```

4.4. The `initGUI` method

Like any other MadKit agent a TurtleWatcher may manage its own interface. By example in the virus simulation the Observer's GUI is a special component that plot data. For instance:

```
public void initGUI()
{
setGUIObject(plot = new SimplePlotPanel("infected
turtles",250,nbMax));
}
```

Override this method is not mandatory. If you do not and launch an Observer with a GUI (`addObserver(o,true)`), MadKit will give the agent a default interface where you can display text information using the `println(String s)` primitive.

5. The TurtleKit simulation Viewer

5.1. Synopsis

A TurtleKit Viewer is the simulation default display agent. It draws the patches and turtles. The default behaviour is to represent them with a coloured rectangle. To add a viewer to the simulation, you should use the `addViewer` method in the `addSimulationAgents` method of the Launcher. These methods allow you to add a default viewer or a special viewer (your own subclass), with a default or a customized cell size (the onscreen patch size).

```
public void addSimulationAgents //in the Launcher
{
 addViewer(3); //default with 3 for cell size
}
```

Figure 7: Simulation default viewer

5.2. How to create your own representation

It is possible to define your own representation of the patch or the turtle (independently). A Viewer uses two methods to draw the world: `paintTurtle`, and `paintPatch`. These methods can be overridden in order to obtain a different display effect. By example, It is possible to make a patch variable concentration visible by drawing patches according to the value of the corresponding variable. So to view different representations of the same simulation you have just to instantiate the Viewer class as many times as you want. Here are two examples of that:

```
public class FlavorViewer extends Viewer
{
 public void paintPatch(Graphics g, Patch p,int x,int y,int CellSize)
 {
   int a = ((int) p.getVariableValue("flavor"))%256;
   g.setColor(new Color(a,a,a));
   g.fillRect(x,y,CellSize,CellSize);
 }
}
```

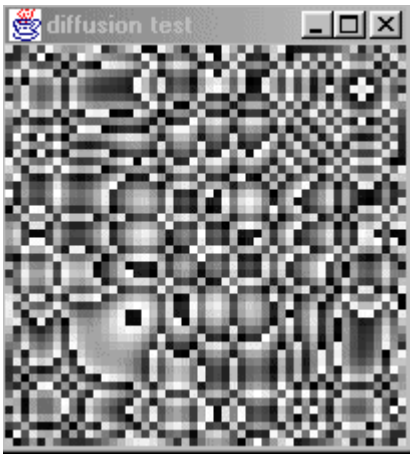

Figure 8: The flavour viewer is a special viewer that displays particular model's information

```
 public void paintTurtle(Graphics g,Turtle t,int x,int y,int cellSize)
 {
   g.setColor(t.getColor());
   g.fillOval(x,y,cellSize*3,cellSize*3);
 }
 public void paintPatch(Graphics g,Patch p,int x,int y,int cellSize)
 {
   g.setColor(p.getColor());
   g.fillOval(x,y,cellSize*3,cellSize*3);
 }
```

Figure 9: This viewer redefines the default shape of the turtles

5.3. Controlling the display with the property Box

In order to control the display process, each Viewer owns a properties Box where you can make changes. By example, you can stop the display by setting the `show` property to false. In the same way, you can choose to view the world state only some times by setting the `flash` value to true and the wanted `flashStepSize`. In this example if flash is settled to true the display will occur every ten simulation steps. All these options are made in order to accelerate the simulation. Indeed the simulation display seriously reduces the simulation speed.

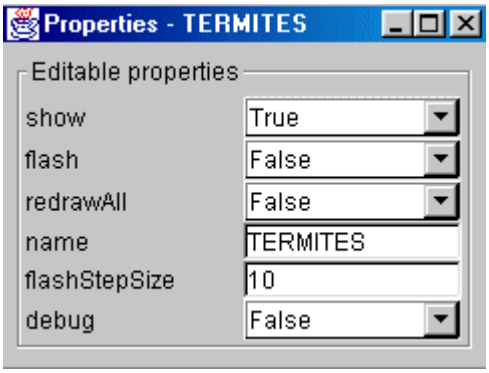

Figure 10: The viewer's property box

6. A step by step example

In this example we will see how to build the termite simulation example from scratch. This simulation is a copy of the corresponding StarLogo project.

This project is inspired by the behavior of termites gathering wood chips into piles. The termites follow a set of simple rules. Each termite starts wandering randomly. If it bumps into a wood chip, it picks the chip up, and continues to wander randomly. When it bumps into another wood chip, it finds a nearby empty space and puts its wood chip down. With these simple rules, the wood chips eventually end up in a single pile.

This example is a transcription of the original. The goal of the termite is to search for woodchips and place them into piles. The presence of a chip on a patch is modeled by the patch's current color: black (empty) or yellow (a chip). The termites are red.

6.1. The Termite class: Termite.java

The first thing to do is to create a new java file: Termite.java. This class is a sub class of turtlekit.kernel.Turtle:

```
public class Termite extends turtlekit.kernel.Turtle
```

Then we have to write the java methods that define the simulated termite behavior. To compute such a behavior, we have to identify the basic behavior of a termite. For instance, we could say: the first behavior of a termite is to search for woodchips. The termite will adopt this behavior until it will find a chip. Then its behavior will be to search for a pile where put the chip down.

Table 1. The first behaviour

```
// As a one time step procedure,
// it returns a String: the next time
behaviour

public String searchForChip()
{
    wiggle();

    if (getPatchColor() == Color.yellow)
    {
         setPatchColor(Color.black);
         randomHeading();
         fd(20);
         return("findNewPile");
    }
    else
         return ("searchForChip");
}
```

```
to searchForChip

{
 //call a defined procedure (see
below)
    do a random move;

    if (there is a chip on the
patch)
    {
        take the chip;

        don't stay in the same
place;

        findNewPile;
    }
else
        searchForChip;
}
```

The wiggle method just defines a sub procedure:

```
void wiggle()
 {
    fd(1);
    turnRight(Math.random()*45);
    turnLeft(Math.random()*45);
 }
```

We define the other behaviours: findNewPile, findEmptyPatch and getaway

Table 2. Other behaviours

Tableau 1: java code	**Tableau 2: Associated behaviour**

`public String findNewPile()`
`{`
`if(getPatchColor()==Color.yellow)`
`return("findEmptyPatch");`
`else`
`{`
`wiggle();`
`return("findNewPile");`
`}`
`}`
	`to findNewPile`
	`{`
	`if (there is a chip)`
	`findEmptyPatch;`
	`else`
	`{`
	`do a random move;`
	`findNewPile;`
	`}`
	`}`

`public String findEmptyPatch()`
`{`
`wiggle();`
`if(getPatchColor()== Color.black)`
`{`
`setPatchColor(Color.yellow);`
`return("getAway");`
`}`
`else`
`return("findEmptyPatch");`
`}`
	`to findEmptyPatch`
	`{`
	`do a random move;`
	`if (the patch is empty)`
	`{`
	`put down the chip;`
	`getAway; //go where there is no chip;`
	`}`
	`else`
	`findEmptyPatch;`
	`}`

`public String getAway()`
`{`
`if(getPatchColor()== Color.black)`
`return("searchForChip");`
`else`
`{`
`randomHeading();`
`fd(20);`
`return("getAway");`
`}`
`}`
	`to getAway //  from the pile`
	`{`
	`if (there is no chip)`
	`searchForChip;`
	`else`
	`{`
	`do a jump in a random direction;`
	`getAway;`
	`}`
	`}`

Now that we have defined the termite behavior, we have to define how the initialization will be done for this turtle. To do this we have to override the mandatory setup method (to make the termite initialize itself at the beginning of the simulation) and to write the Termite constructor where we choose the first behavior the termite will take.

```
public void setup()
      {
       playRole("termite");
       setColor(Color.red);
       randomHeading();
      }

public Termite()
      {
       super("searchForChip");
      }
```

The Termite is now completely defined. Now, we have to do the initialization of the patches for this simulation. To do this we will use a custom Observer.

6.2. The PatchInitializer class

For this simulation the patch initialization is simple: the patches have to be black or yellow given a density rate. So we need to build a special Observer and just override his setup method to make the patch initialization.

```
public class PatchInitializer extends turtlekit.kernel..Observer
   {
     float densityRate;

      public PatchInitializer(float density)
      {
       densityRate = density;
      }

      public void setup()
      {
        for(int i=0;i<envWidth;i++)
         for(int j=0;ji<envHeight;j++)
          if (Math.random() i< densityRate)
           patchGrid[i][j].setColor(Color.yellow);
         else
          patchGrid[i][j].setColor(Color.black);
      }
    }
```

It is in the setup method where we finally make the initialization of the patches. Note that we do not have overridden the watch method of the Observer. Indeed this Observer has no function during the simulation. With these two couples of methods, these variables will be added to the other default properties of a Launcher. So they will be directly accessible during the simulation in the Launcher's properties Box. So you can change the values before launching the simulation or doing a reset. Now we have to add the agents in the simulation engine by overriding the addSimulationAgents method of the TermiteLauncher:

```
public void addSimulationAgents()
      {
       //Add the termites with the addTurtle method.
       for (int i = 0; i i< nbOfTermites; i++)
       {
            Termite t = new Termite();
            addTurtle(t);
       }

//We choose to add a default viewer with a display cell size of 3 pixels.
       addViewer(3);

//Add the PatchInitializer with no GUI (false)
       addObserver(new
               PatchInitializer(densityRate), false);
       }
```

The simulation is now ready to run, but we can write a constructor for the TermiteLauncher in order to give this simulation our own default values (width, height, simulation name, wrap mode...).

```
public TermiteLauncher()
{
setSimulationName("TERMITES");
setWidth(150);
setHeight(110);
}
```

Now the only thing to do (after the compilation) is to launch the TermiteLauncher within MadKit !

6.3. Launch the simulation using MadKit

To finally use the simulation you have to launch the TermiteLauncher within Madkit. There is various way of doing that. You will find all you need in the MadKit User and Developer Guides.

7. Using the PythonTurtle interpreter

7.1. The pyTurtle command center

To be more flexible, turtlekit also provides the possibility of using the Python language to implement turtles or directly interact within a running simulation using simple commands. It is even possible to dynamically create procedures which can be associated with a button like in the original StarLogo.

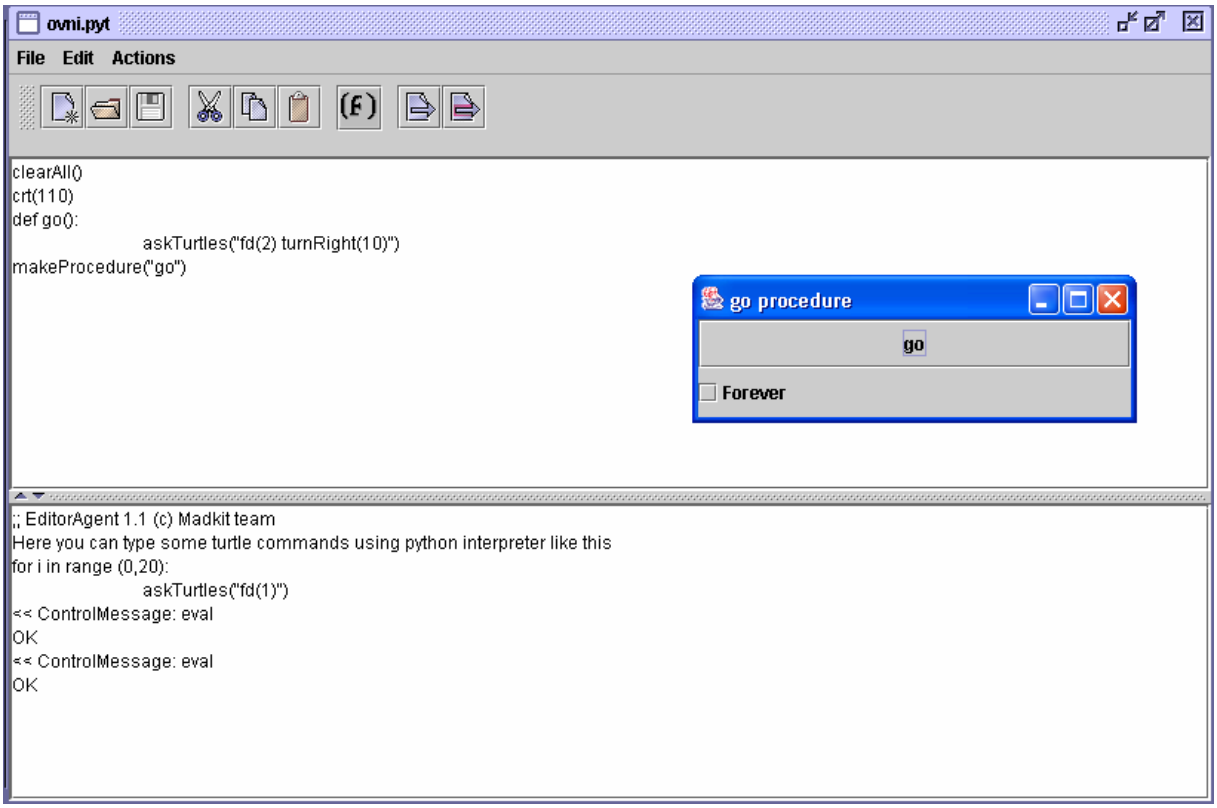

Figure 11: The PyTurtle command center

In this figure, first part of window is where you have to type the pyturtle code. Under this, the pyturtle interpreter shows evaluation messages.

In this example, the first line, `clearAll()`, will reset everything in the environment: delete every turtle and set the patches' colour to black. The second line, `crt(110)`, will create 110 turtles with random heading and colour. The three following lines define a procedure named go that will ask every turtle to do some actions with the *askTurtles* primitive. This primitive take a string as parameter which defines a list of regular turtle commands. Then the `makeProcedure` primitive will create a button associated with this procedure. These five lines exactly correspond to the simulation named OVNI (see chapter 2). Moreover it is possible to save predefined scripts which can be run in the interpreter. The *open* button opens a directory that contains several examples of such scripts.

7.2. The Termites simulation in *pyTurtle*

This code is contained in the *termites.pyt* source file. To ensure having the same execution procedure than the regular java turtles, it is here necessary to create a python class which define the Termite. This is mandatory because termites define four different behaviours which must be called in a specific order (see section 6 for details). So we have to create a variable, `whatToDoNext`, which define the next behaviour to activate when the turtle's turn comes.

```
from java.lang import Math
from turtlekit.kernel import Turtle
from java.awt import Color

class PythonTermite(Turtle):
	whatToDoNext=0

	def initWorld(self):
		self.setColor(Color.red)
		for x in range(0,120):
			for y in range(0,120):
				if Math.random() > 0.8:
					self.setXY(x,y)
					self.setPatchColor(Color.yellow)

	def init(self):
		self.setColor(Color.red)
		self.randomHeading()
		self.setX(Math.random()*120)
		self.setY(Math.random()*120)

	def wiggle(self):
		self.turnRight(Math.random()*45)
		self.turnLeft(Math.random()*45)
		self.fd(1)

	def getAway(self):
		if self.getPatchColor() == Color.black:
			self.whatToDoNext=1
		else:
			self.randomHeading()
			self.fd(20)
			self.whatToDoNext=0

	def searchForChip(self):
		self.wiggle()
		if self.getPatchColor() == Color.yellow:
			self.setPatchColor(Color.black)
			self.fd(20)
			self.whatToDoNext=2
		else:
			self.whatToDoNext=1

	def findNewPile(self):
		if self.getPatchColor() == Color.yellow:
			self.whatToDoNext=3
		else:
			self.wiggle()
			self.whatToDoNext=2
```

```
        def findEmptyPatch(self):
                self.wiggle();
                if self.getPatchColor() == Color.black:
                        self.setPatchColor(Color.yellow)
                        self.whatToDoNext=0
                else:
                        self.whatToDoNext=3

        def makeIt(self):
                if self.whatToDoNext == 0:
                        self.getAway()
                elif self.whatToDoNext == 1:
                        self.searchForChip()
                elif self.whatToDoNext == 2:
                        self.findNewPile()
                else:
                        self.findEmptyPatch()

def setupWorld():
        clearAll()
        addTurtle(PythonTermite())
        askTurtles("initWorld()") #faster than asking an anonymous turtle
to do it with askTurtles
        clearT()
        for i in range(0,125):
                addTurtle(PythonTermite())
        askTurtles("init()")

makeProcedure("setupWorld")

def doIt():
        askTurtles("makeIt()")

makeProcedure("doIt")
```

As termites are a python class, it is necessary to use the `addTurtle` command instead of the `crt` command. This command takes a subclass of Turtle as parameter and adds it to the system. Then, to initialize the system, we ask a single turtle to do it with the `setupWorld` procedure. This script will create two button procedures, `setupWorld` and `doIt`. To run the simulation, just press the setupWorld button, then press the doIt button after check its forever parameter (run the procedure forever).

7.3. Pyturtle language's possibilities

The main interest of this feature is to check java behaviours dynamically at runtime. Indeed it is possible to stop a java regular simulation to type some pyTurtle command to see how a particular behaviour is executed by the turtles. Furthermore, it is possible to add perturbations in a model at runtime by make the turtles doing a dynamically defined procedure.

This feature has been added recently and all its possibilities have not been explored yet. For examples it is possible to activate several *forever procedures* at the same time but such simulations have unknown properties that we do not analysis for now. We hope to give further explanations and examples of its use in a near future, stay tune.

Chapter II

The TurtleKit simulation Pack

1. Summary

This section includes some demonstration simulations that show several functionalities of the TurtleKit simulator.

This pack includes the following simulations:

- **Demo simulations.** Simple test demonstrations with several source codes: Walkers, Mosquitoes, Creation and Ovni.
- **Virus.** An epidemic simulation.
- **Gravity.** Turtles turn around others. No mathematic considerations here, just simple behaviours to obtain an interesting visual effect.
- **Gas simulation.** A transcription of a StarLogo project.
- **Soccer.** Turtles love playing soccer too.
- **Patch variable diffusions.** A simple simulation without turtles, to show diffusion in the environment of two patch variables.
- **Termites.** The transcription of the well known StarLogo project.
- **Conway's Game Of Life**. This shows how to use TurtleKit as a cellular automata platform

2. Demo simulations

2.1. Walkers

2.1.1. Synopsis

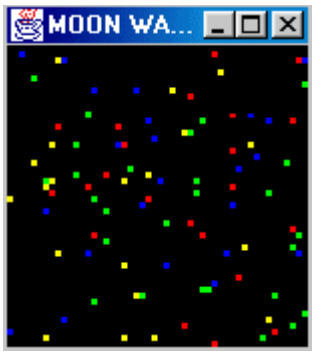

Figure 12: Walkers are just walking

Walkers is a little simulation where the turtle have two behaviours: walk forward or change its colour according to a new random heading. This simulation was made to test the good working of the heading primitives.

2.1.2. Code

```
package turtlekit.simulations.tests;

import turtlekit.kernel.Turtle;
import java.awt.Color;

/** the only thing is to walk and change color
  @author Fabien MICHEL
  @version 1.2 6/12/1999 */

 public class Walker extends Turtle
 {
      int count=10;

 public Walker(String s)
{super(s);}

public void setup()
{
      randomHeading();
      playRole("walker");
}

public String walk()
{
      fd(1);
      if (count < 0)
      {
            count = (int) (Math.random()*90);
             return("change");
      }
      else
      {
            count--;
            return ("walk");
      }
}

public String change()
{
      randomHeading();
      if (getHeading() > South) setColor(Color.red);
      else if (getHeading() > West) setColor(Color.blue);
      else if (getHeading() > North) setColor(Color.green);
      else setColor(Color.yellow);
      return("walk");
}
}
setHeading(towards(x,y));
```

2.2. Mosquitoes

2.2.1. Synopsis

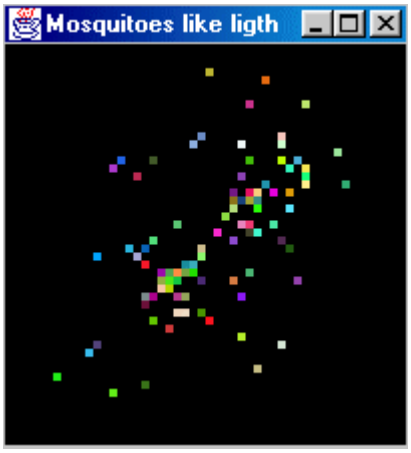

Figure 13: Mosquitoes fly around but always return to the light source

In Mosquitoes the turtles just "fly" in a random way during a countdown and then fall down to a light source represented by a yellow patch. Turtles known the localization (x,y) of this patch (as a parameter in the constructor). So they use the `towards` command to set their heading.

2.2.2. Code

```
package turtlekit.simulations.tests;
import turtlekit.kernel.Turtle;
import java.awt.Color;

/** a turtle for a turtle command test
    @author Fabien MICHEL
    @version 1.1 4/1/2000 */

public class Mosquito extends Turtle
{
    int sunX,sunY,cpt=50;
    public Mosquito(int a,int b)
    {
      super("move");
      sunX=a;
      sunY=b;
    }

    public String move()
    {
      if (Math.random() > 0.5) turnRight(15);
      else turnLeft(15);
      fd(1);
      cpt--;
      if (cpt < 0)
          {
            setHeading(towards(sunX,sunY));
            return ("fall");
          }
      else return ("move");
    }

    public String fall()
    {
      fd(1);
      if (distance(sunX,sunY) < 1)
          {
            cpt = (int) (Math.random()*100);
            return "move";
          }
      else return "fall";
    }

    public void setup()
    {
      setXY(sunX,sunY);
      setPatchColor(Color.yellow);
      moveTo(sunX+(int) (Math.random()*10),sunY+(int) (Math.random()*10));
      setHeading(Math.random()*360);
      setColor(new          Color((int)          (Math.random()*256),(int)
(Math.random()*256),(int) (Math.random()*256)));
    }
}
```

2.3. Creation

2.3.1. Synopsis

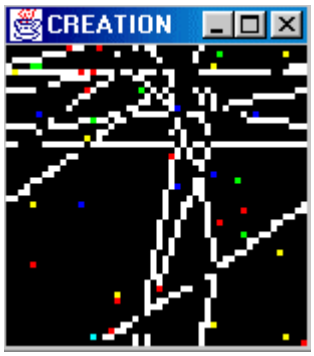

Figure 14: A turtle can create other turtles

In this simulation a turtle, a creator, tests the good working of the `createTurtle` command. When moving, a creator draws the patches in white. Then, when it crosses a patch that is already white, it creates a new turtle of a random kind using the `createTurtle` command.

```
createTurtle(new Walker());
```

At the beginning of the simulation, you can set the wanted number of creator in the launcher's properties box. Moreover an Observer, CreationObserver, displays the total number of turtles alive.

2.3.2. Code

```
/** Turtles who create other kind of turtle during simulation and dies
after a countdown

@author Fabien MICHEL
@version 1.1 6/12/1999 */

public class Creator extends Turtle
{
      public int life=50;

public Creator()
{
      super("ride");
}

public void setup()  { playRole("creator"); }

/**these behaviors have no means, just a test*/

public String ride()
```

```
{
      fd(1);
      if (getPatchColor()==Color.white)
      {
            if (Math.random()<0.99) createTurtle(new Walker("walk"));
            else
                  if (Math.random() < 0.95)
                        for (int i = 0; i < 50; i++)
                              createTurtle(new Ovni());
                  else
                        launchGravity();
      life--;
      setPatchColor(Color.black);
      return("erase");
      }
      else
      {
            setPatchColor(Color.white);
            return("ride");
      }
}

public String erase()
{
      if (life < 0) return null;    //the turtle dies

      turnLeft(Math.random()*50);
      turnRight(Math.random()*50);
      fd(1);
      if (getPatchColor()==Color.white)
      {
            setColor(Color.lightGray);
            setPatchColor(Color.black);
            return("erase");
      }
      else
      {
            setColor(Color.cyan);
            return("ride");
      }
}

/**launch turtles of the gravity simulation*/

public void launchGravity()
{
      Turtle[] ts = new Turtle[1];
      Turtle t1 = new BlackHole();
      ts[0] = t1;
      createTurtle(t1);
      for (int i = 0; i < 30; i++)
            createTurtle(new Star(ts,13));
}

}
createTurtle(new Walker());
```

2.4. OVNI

2.4.1. Synopsis

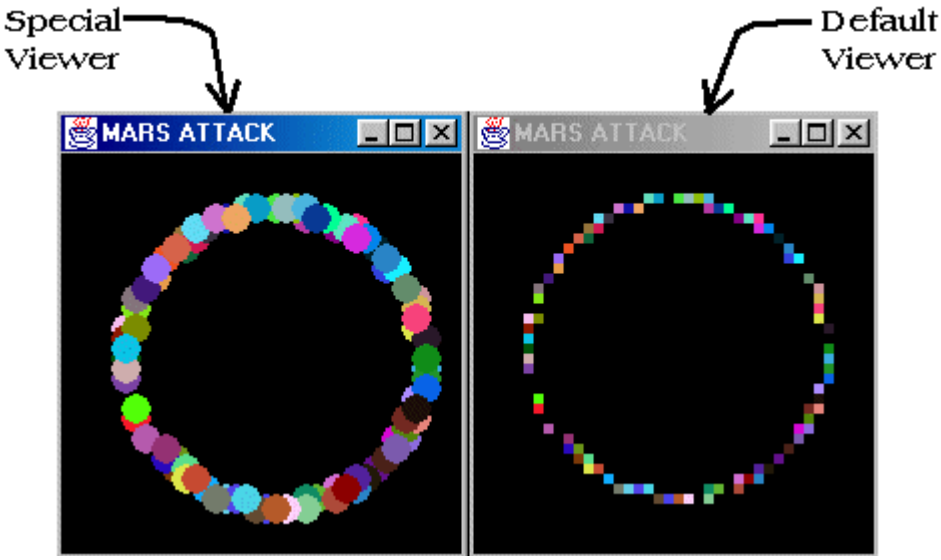

Figure 15: Multiple world interpretations

The Ovni (french for UFO) simulation was made only to test the display of multiple representations at the same time. To do this we have written a SpecialViewer (extends Viewer) and overridden its `paintTurtle` method in order to obtain that the turtles was shown like disks. So it is possible to create your own representation of a patch or a turtle (you can use a gif for a turtle).

```
public class SpecialViewer extends Viewer
{
    public void paintTurtle(Graphics g,Turtle t,int x,int y,int cellSize)
    {
                g.setColor(t.getColor());
                g.fillOval(x,y,cellSize*3,cellSize*3);
        }

    public void paintPatch(Graphics g,Patch p,int x,int y,int cellSize)
    {
                g.setColor(p.getColor());
                g.fillRect(x,y,cellSize*3,cellSize*3);
        }

}
```

We have also overridden the `paintPatch` method in order to avoid that the only draw of the turtles drops some residue on the floor. This dues to the optimizations made in the default display engine (using the redrawAll function (slower) is a solution too).

3. Virus

3.1. Synopsis

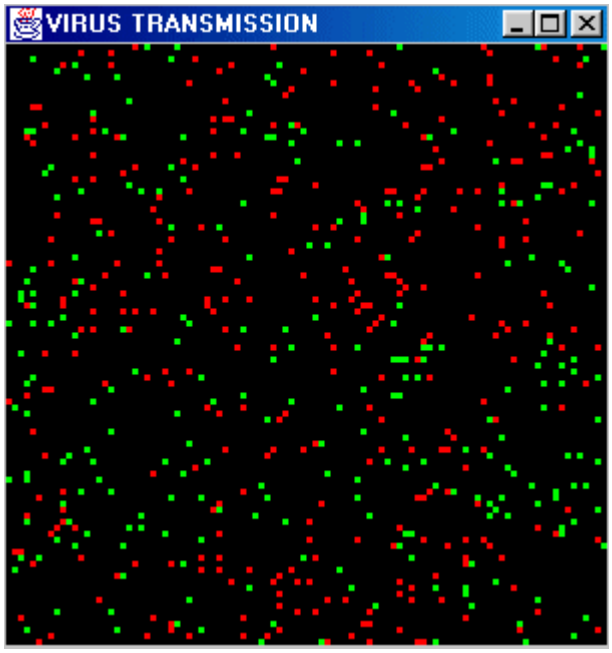

Figure 16: The green turtles are contaminated by the red ones

In this simulation, we want to simulate the transmission of a virus in a population and observe it. To do this, we have created a kind of turtle (Virus class) that can take two behaviors: infected or non infected (red and green). The all turtles just walk around and the red ones contaminate the others when cross them on a patch.

3.2. Virus transmission simulation

We have created two classes of Virus to simulate two ways to transmit the virus. The first way is to use real MadKit agent messages: when a red turtle cross turtles that are green, it gets their AgentAddress and sends them a VirusMessage.

So if a sane turtle has its mailbox not empty, it changes of behavior and becomes a red turtle. (Just like an email virus: we can imagine another version of this simulation where a turtle might be capable of being careful when reading its mailbox.

```
public String red()
  {
        wiggle();
        Turtle[] ts = turtlesHere(); //returns the other turtles that are
on the same patch
        if (ts != null)
                for (int i=0; i < ts.length;i++)
                        if (ts[i].getColor() == Color.green)
                                sendMessage(ts[i].getAddress(),new
VirusMessage());
        return("red");
 }
}

public String green()
  {
  if (nextMessage() != null) //check mailbox
  {
         setColor(Color.red);
         playRole("infected");
         return ("red");
  }
  else
  {
         wiggle();
         return ("green");
  }
 }
```

The second way (Virus2) consists in directly interact with the other turtles by changing their color to simulate the transmission. So a sane turtle always check its color and adopts the corresponding behavior: red or green.

```
public String red()
  {
        wiggle();
        Turtle[] ts = turtlesHere();
        if (ts != null)
                for (int i=0; i < ts.length;i++)
                        if (ts[i].getColor() == Color.green)
                                ts[i].setColor(Color.red);
        return("red");
 }

public String green()
  {
  if (getColor() == Color.red)
  {
         playRole("infected");
         return ("red");
  }
  else
  {
         wiggle();
         return ("green");
  }
 }
```

In each case, an infected turtle plays the role "infected" in order to be distinguished by the observer.

3.3. Observing the simulation

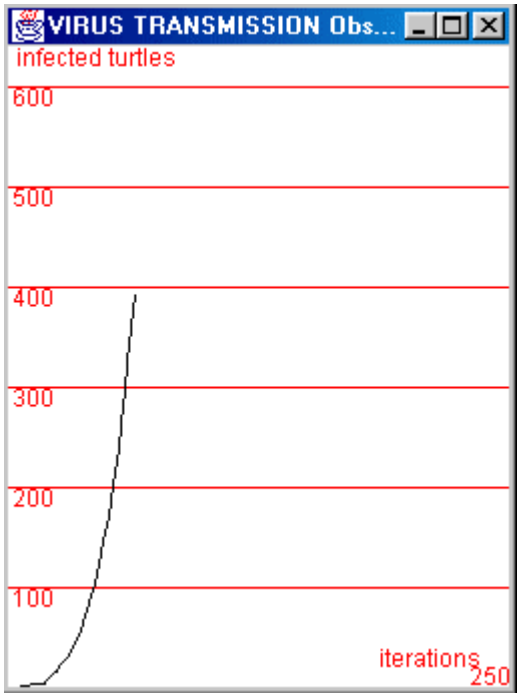

Figure 17:displaying simulation's results in a special Observer

The Observer of this simulation, the VirusWatcher, just creates a TurtleTable with the role infected and displays the corresponding number using a special GUI (a madkit.lib.simulation.SimplePlotPanel) suited to draw mathematic curves.

```
/**   initialize   the   variable   infectedTurtles   (a   Turtle[])   using
getTurtleWithRole*/
public void initializeTurtleTables()
{
    infectedTurtles = getTurtleWithRole("infected");
}
/**this method overrides watch in the class Observer. So it will be invoked
for each simulation step*/
 step public void watch()
{
        plot.addPoint(infectedTurtles.length); //plot is the GUI
}
```

3.4. About the code

A Turtle is a real MadKit agent: it owns all the possibilities of any `AbstractAgent` of MadKit. Specially, interesting points are the usage of group, roles and `AgentAddress` concepts to structure the application logics and identify the agents.

4. Gravity

4.1. Synopsis

Figure 18: Turtles are a part of the universe

In this simulation we have computed two kinds of turtle with different behaviours. One class called `BlackHole` and another one called `Star`. The starts turn around the nearest black hole. If a star is too near from two black holes, it is just no longer under influence. Here, the simulation of the gravity is far from having relation with real physic.

4.2. About the code

The turltes' code is based on the usage of the following turtle commands: `towards`, `distance`, `xcor` and `ycor`. See source files for details.

5. Gas simulation

5.1. Synopsis

This simulation mimics the StarLogo team's Molecules project.

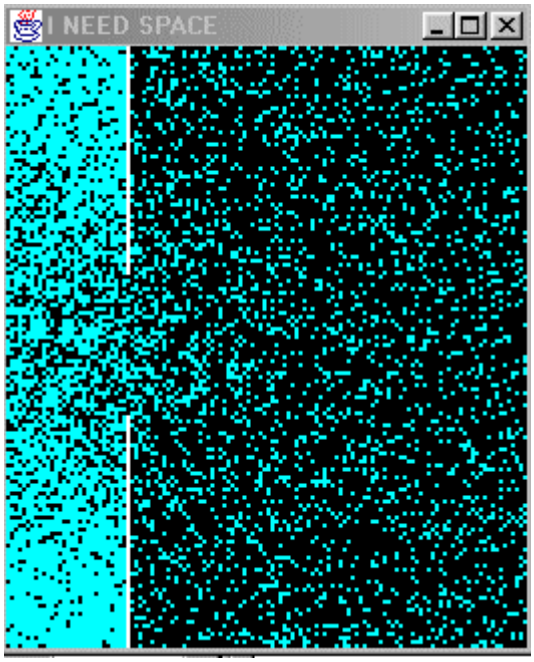

Figure 19: PV = nRT

This project presents a (highly simplified) model of gas molecules in a box. The box is divided into two sections with a small hole between the two sections.

5.2. Using the simulation

You can choose the number of molecule and it is also possible to choose the size of the hole (in the properties box of the Launcher, `GasExperiment`).

5.3. About the code

For this simulation, we just have created a kind of Turtle `Gas` with a single behavior: look for free space. A special Observer `GasObserver` observes and displays the number of turtles that are on the right side of the wall.

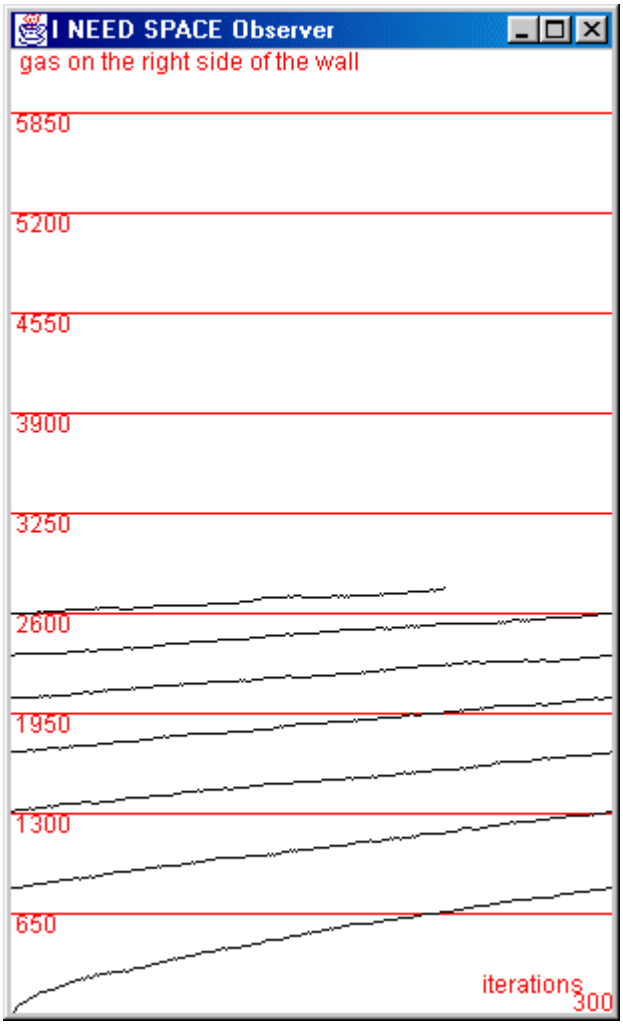

Figure 20: curve will stabilize when molecules will be proportionally distributed

6. Soccer

6.1. Synopsis

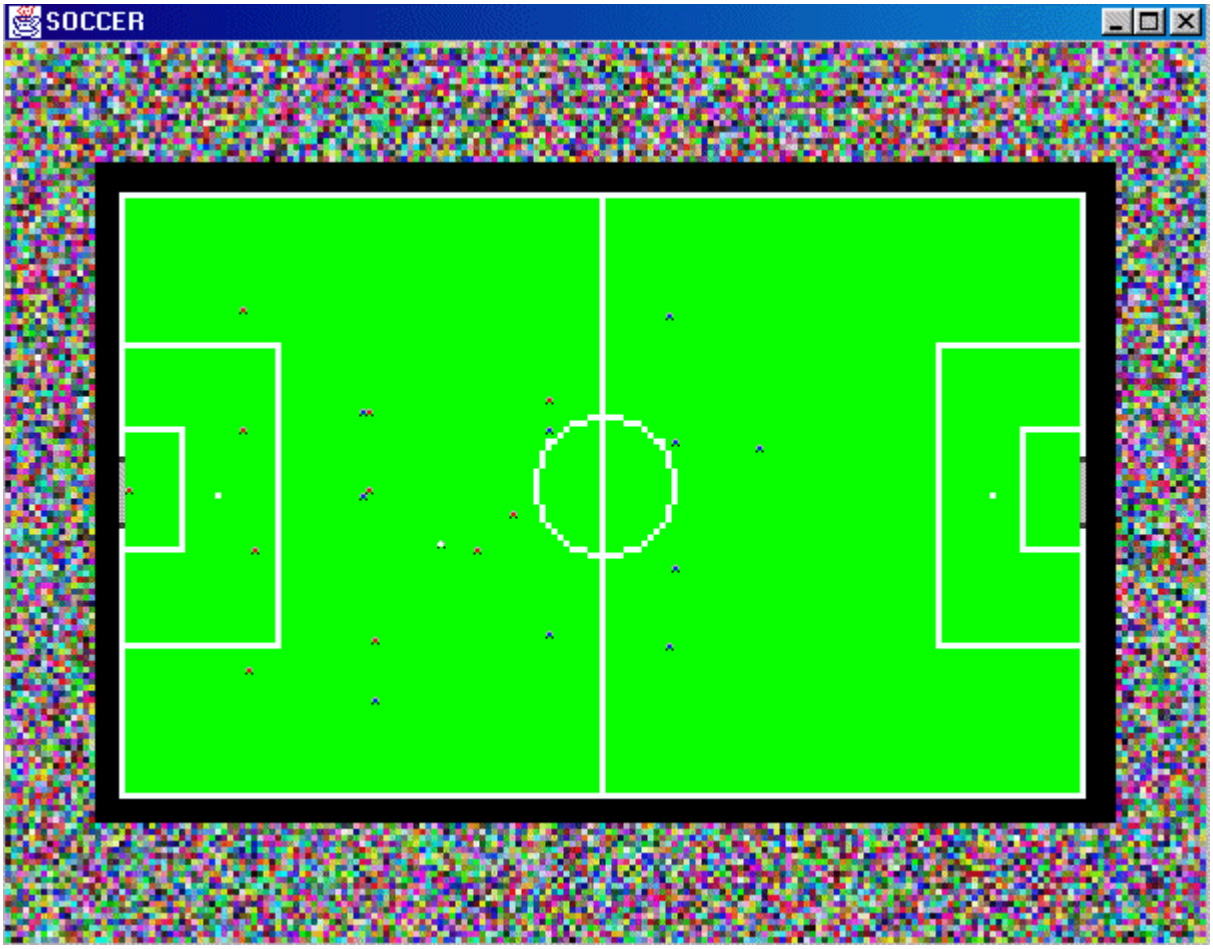

Figure 21: Turtles like soccer too!

This simulation is the most complex one and is not achieved. For now, the turtles' "internal AI" is very poor. But we will try to increase it :).

6.2. About the code

Although the code is not very explicit, You can try to modify the behaviour of the players by modifying the following classes: `Player`, the super class of all soccer players, `RedPlayer` and `BluePlayer` that define particular behaviour for each team. The ball is also a turtle `Ball`. The turtles are created in such a way that the red team is composed by the turtles 0 to 10 (blue 11 to 21). This is done by adding the turtles in the right order in the `addSimulationAgents` of the Launcher (`Soccer`). A turtle's ID corresponds to the number of invocations of the `addTurtle` method made by the Launcher.

7. Patch variable diffusions

7.1. Synopsis

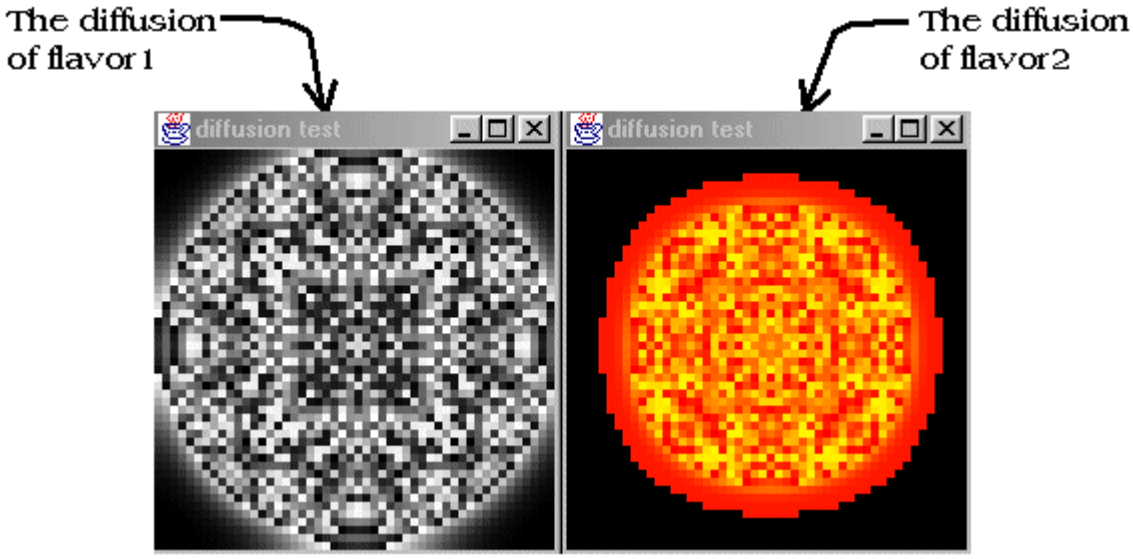

Figure 22: Two patch variables views at the same time

This simulation was made in order to test the diffusion of the patch variables in the environment. At the beginning of the simulation the values of the center patch's variables are settled to a big number.

7.2. Initializing the simulation

You can set the diffusion and evaporation coefficients and the value of the center patch of each patch variable (flavor and flavor2) in the Launcher's properties box (`Diffusion`). So you will obtain different effects during the simulation.

7.3. About the code

This little demo shows how to initialize and use patch variables: the `Flavor` class. These jobs have to be done in the `Launcher` code in the `initializePatchVariables` like in the following example.

```
    protected void initializePatchVariables()
    {
        PatchVariable a = new PatchVariable("flavor");
        a.setDiffuseCoef(valueDiff);
        a.setEvapCoef(valueEvap);
        addPatchVariable(a);
        PatchVariable b = new PatchVariable("flavor2");
        b.setDiffuseCoef(value2Diff);
        b.setEvapCoef(value2Evap);
        addPatchVariable(b);
    }
```

To view these two diffusions independently, we have made two specials viewers classes: `FlavorViewer` and `FlavorViewer2`. In each Viewer we have just overridden the `paintPatch` method in order to make the flavor's concentration visible on the screen.

```
public class FlavorViewer extends Viewer
{
    public void paintPatch(Graphics g, Patch p,int x,int y,int CellSize)
    {
        int a = ((int) p.getFlavorValue("flavor"))%256;
        g.setColor(new Color(a,a,a));
        g.fillRect(x,y,CellSize,CellSize);
    }
}
```

The way the flavor's intensity is interpreted as a specific color has no specific meaning.

Moreover we have used an `Observer`, `GridInitializer`, to setup the center patch at the beginning of the simulation overriding the `setup`. Note that in this simulation there is no turtle.

```
public class GridInitializer extends Observer
{
double val,val2;

 public GridInitializer (double v, double v2)
        {
         val = v;
         val2 = v2;
        }

public void setup()
{
patchGrid[(int)   (envWidth/2)][(int)(envHeight/2)].setFlavorValue("flavor",
val);
patchGrid[(int)  (envWidth/2)][(int)(envHeight/2)].setFlavorValue("flavor2",
val2);
}
}
```

8. Termites

8.1. Synopsis

This simulation mimics the well known StarLogo team's project named Termites.

This project is inspired by the behaviour of termites gathering wood chips into piles. The termites follow a set of simple rules. Each termite starts wandering randomly. If it bumps into a wood chip, it picks the chip up, and continues to wander randomly. When it bumps into another wood chip, it finds a nearby empty space and puts its wood chip down. With these simple rules, the wood chips eventually end up in a single pile.

This example is a transcription of the original. The presence of a chip on a patch is modeled by the patch's current color: black (empty) or yellow (a chip). The termites are red.

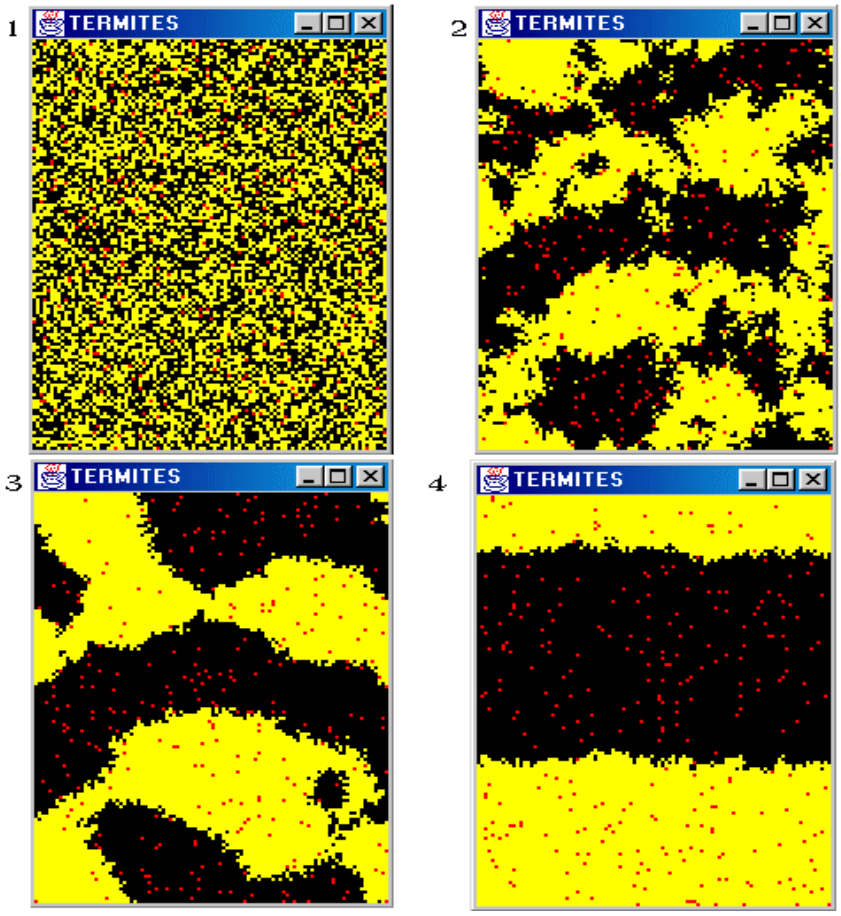

Figure 23: Termites: an illustration of emergence

8.2. About the code

This simulation and its computation are explained in details in the section *A step by step example* in the chapter I.

9. Conway's Game of Life

9.1. Synopsis

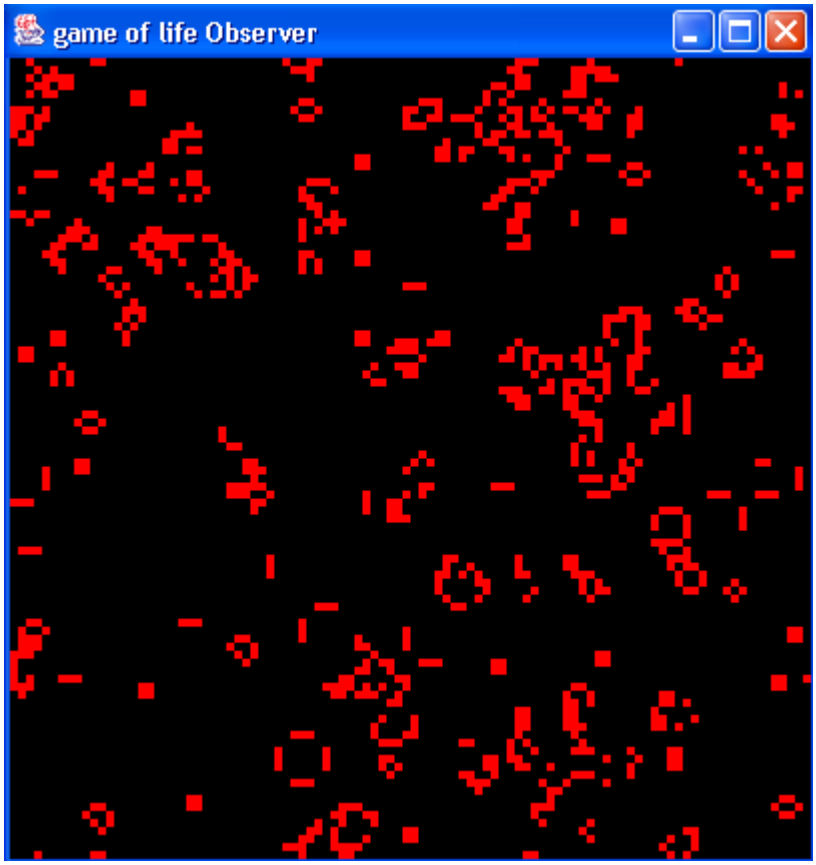

Figure 24 : The Conway's game of life

This simulation implements the Conway's game of life. This simulation is not a regular one in the sens that it does not involve any turtle. Here, the TurtleKit is used as a cellular automata platform.

9.2. Using an observer as a cellular automata programmer

To use the TurtleKit as a cellular automaton, it is possible to use an observer that controls the patches' evolution during the simulation. This operation is done using the patchGrid variable of the observer. The observer simply updates the patches' properties once per turn considering the vinicity of each patch.

9.3. About the code

First, the observer setup the simulation to give patches a starting value (alive or dead).

```
    public void setup()
    {
    gridBuffer=new byte[envWidth][envHeight];
        for(int i=0;i<envWidth;i++)
                for(int j=0;j<envHeight;j++)
                        patchGrid[i][j].setPatchVariable("lifeValue",0);

        for(int i=0;i<envWidth;i++)
                for(int j=0;j<envHeight;j++)
                        if(Math.random()<percentage)

        patchGrid[i][j].setPatchVariable("lifeValue",1);
                        else
        patchGrid[i][j].setPatchVariable("lifeValue",0);
    }
```

Then the *watch* method is implemented in the following way to create the cellular automaton's evolution.

```
    public void watch()
    {
        for(int i=0;i<envWidth;i++)
                for(int j=0;j<envHeight;j++)
                {
                        /*Patch[]*/                neighbors                =
patchGrid[i][j].getNeighbors();
                        byte alive=0;
                        for(int k=0;k<neighbors.length;k++)
                        {
                alive+=(byte)neighbors[k].getVariableValue("lifeValue");
                        }

        gridBuffer[i][j]=(byte)patchGrid[i][j].getVariableValue("lifeValue");
                        if ( gridBuffer[i][j]==1 && (alive < 2 || alive >3))
                        {
                                gridBuffer[i][j]=0;
                        }
                        else if (alive == 3)
                                gridBuffer[i][j]=1;
                }
        for(int i=0;i<envWidth;i++)
                for(int j=0;j<envHeight;j++)

        patchGrid[i][j].setPatchVariable("lifeValue",gridBuffer[i][j]);
}
```