COMPETENCIA DE VIDA ARTIFICIAL

Diseño Completo

INTRODUCCIÓN

Teniendo en cuenta que el desarrollo del código es en el lenguaje Java y en base a un Diseño Orientado a Objetos, se detallan a continuación los lineamientos generales de las clases que forman parte del programa de vida artificial. Existe una clase a partir de la cual el concursante deberá desarrollar el método propio de supervivencia de sus microorganismos. Hay otras clases de las cuales se crearán objetos que serán de los organizadores del encuentro y que el concursante no podrá modificar. Algunos de estos objetos estarán accesibles para que los microorganismos puedan obtener datos útiles en el momento de tomar decisiones estratégicas.

El siguiente diagrama simplificado de las clases muestra el modelo del *Competidor1* y del *Competidor2*, que son las clases de los microorganismos desarrollados por los competidores a partir de los cuales se realizarán las colonias. El método más importante es **move**. El alimento para los MO es administrada por la clase **Agar** y la determinación de las relaciones entre los microorganismos y su supervivencia las resuelve **Environment**.

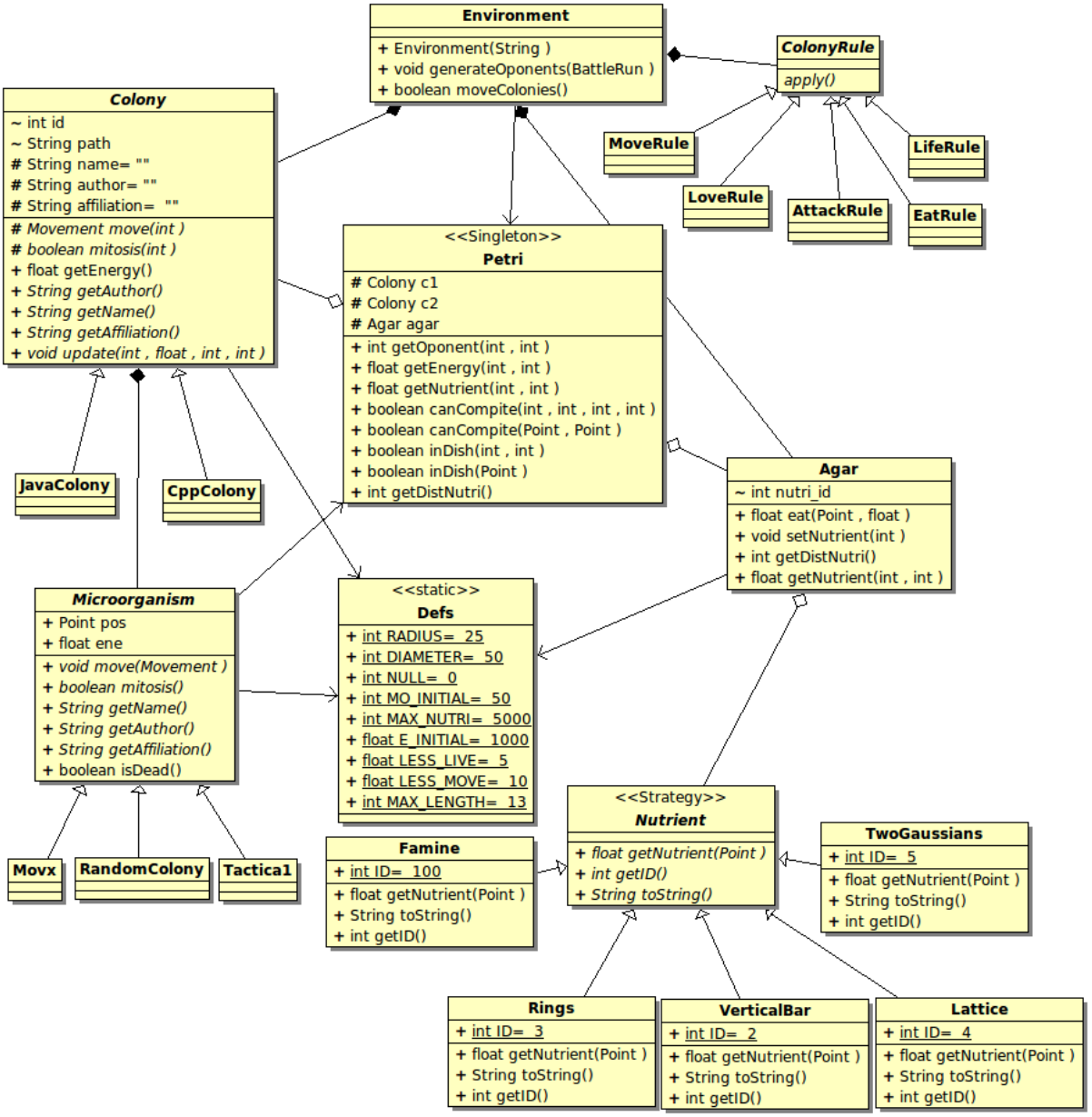

Figura 1: Diseño General

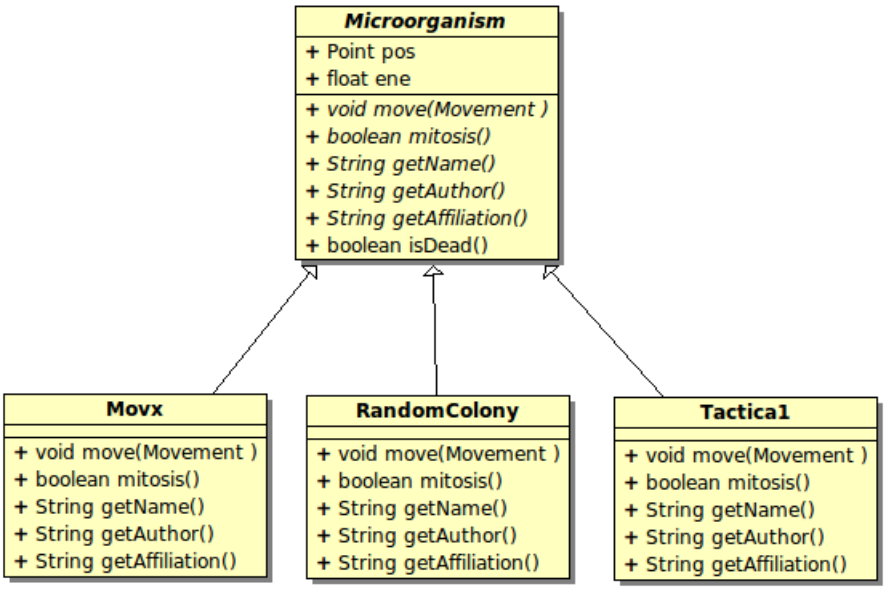

Figura 2: Clase Microorganism

La clase Microorganism

Esta clase declara los métodos básicos del comportamiento, de la cual deberá heredar el MO concursante.

En la clase descendiente que debe desarrollar el concursante, a través del método **move** le indicará a donde quiere moverse cuando sea su turno. Cuando la colonia invoca el método **mitosis** del microorganismo, este indica si se quiere dividir en dos. El método **update** es desde donde recibe el identificador (**id**) con el cual está compitiendo, sus coordenadas y energía antes de pedir que devuelva el movimiento deseado, es decir que estos atributos del MO le son informados. Este método **update** es implementado únicamente en la clase madre **Microorganism** por lo que no es necesario que el competidor lo redefina en su MO.

La clase Colony

Modela una colonia de microorganismos. Cada colonia de microorganismo puede estar implementa- da tanto en C/C++ como en Java. Inicialmente cada participante tiene, en su colonia, 50 microorganismos que luego pueden ir duplicándose (este proceso se llama **mitosis**) e incrementando el número de microorganismos, pero éstos también pueden morir.

La Clase Agar

Esta es una clase muy importante. El agar es el medio de cultivo del que se alimentan los MO y por ende es la clase que administra las distribuciones de nutrientes.

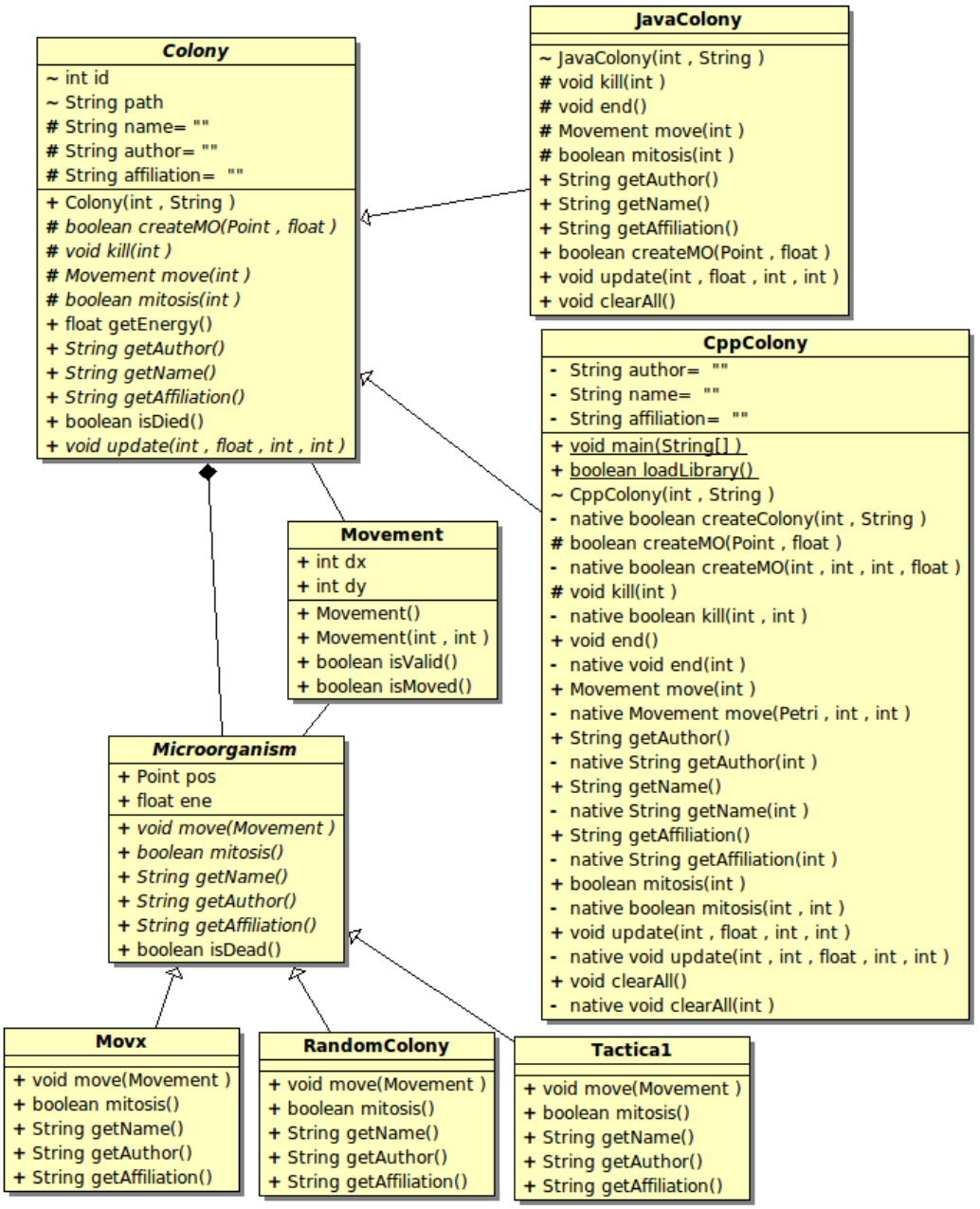

Figura 3: Clase Colony

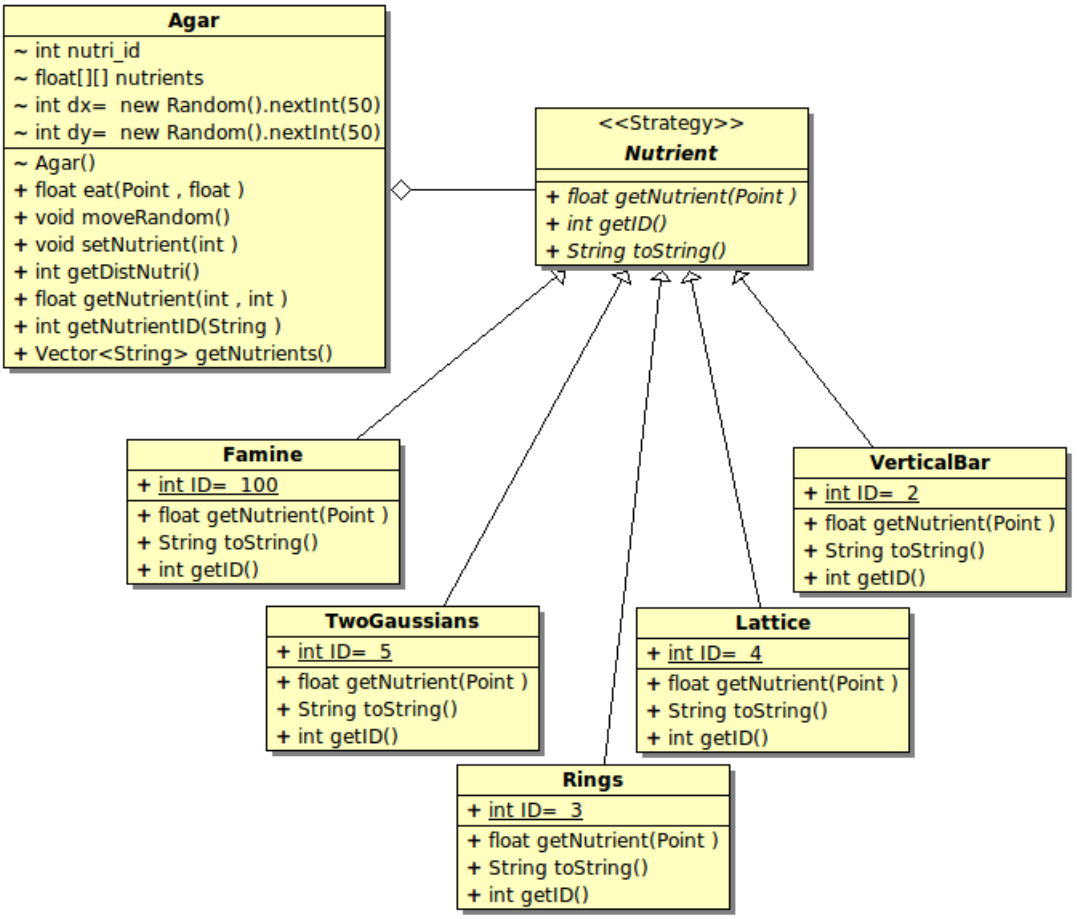

Figura 4: Clase Agar

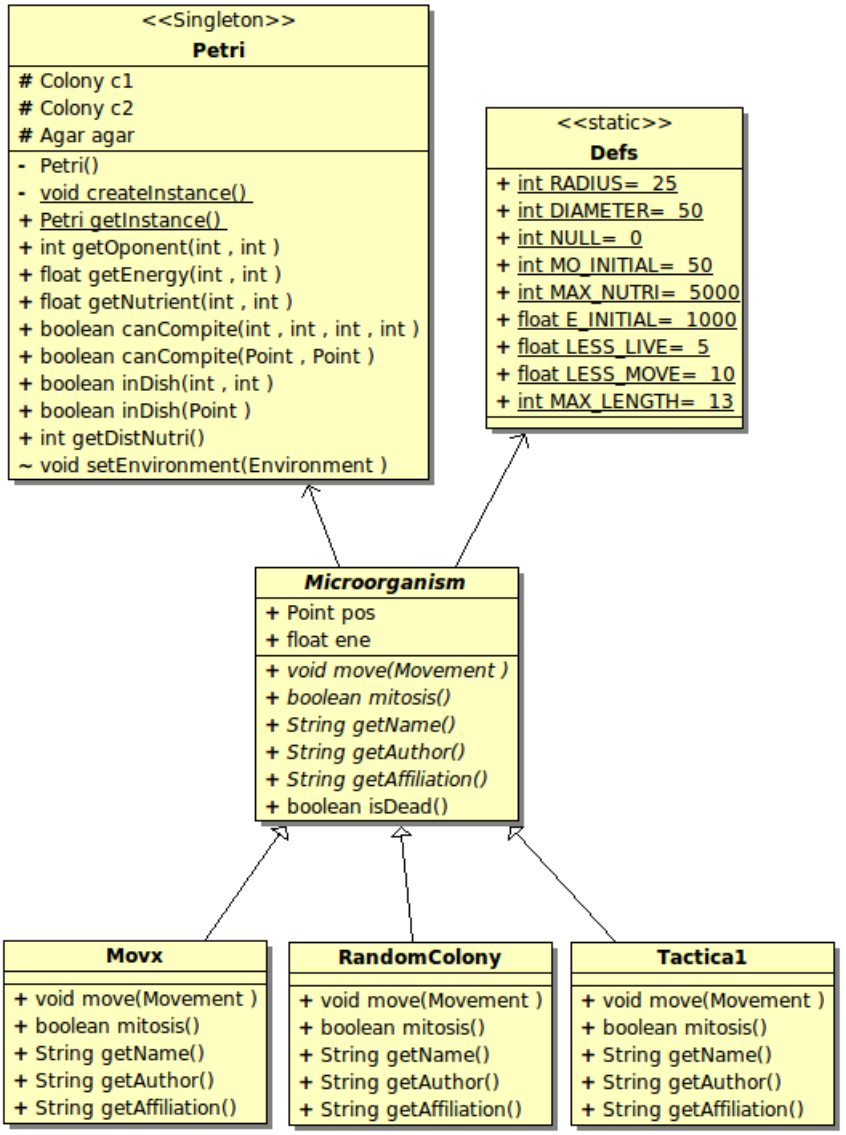

Figura 5: Clase Petri

La clase Petri

Esta es la clase más importante para el competidor por que es la encargada de brindar información a los microorganismos. Cada competidor puedo invocarla para obtener información necesaria para tomar decisiones estratégicas. Los métodos disponibles son:

int **getOponent**(int x, int y): retorna un identificador de la clase del MO que se encuentra en la posición (x,y). si la posición está libre se retorna 0.

float **getEnergy**(int x, int y): retorna la energía del MO que se encuentra en la posición (x,y). si la posición está libre se retorna 0.

float **getNutrient**(int x, int y):retorna la cantidad de nutrientes que hay en la posición (x, y).

int **getDistNutri**(): Este método retorna el tipo de distribución de nutrientes que se está utilizando en la competencia. Los identificadores que las representan son los siguientes:

1 : plano inclinado (Inclined Plane)
2 : barra vertical (Vertical Bar)
3 : anillo (Rings)
4 : grilla (Lattice)
5 : dos gausianas (Two Gaussians)
100: distrubucion uniforme (Famine)

Además, la clase Petri provee dos métodos para facilitar la programación de un MO:

boolean **canCompite**(int x1, int y1, int x2, int y2): retorna true si el MO en la posición (x1,x2) es oponente del MO en la posición (y1,y2).

boolean **inDish**(int x, int y): retorna true si la posición (x, y) está dentro del entorno, es decir, pertenece al círculo del centro (Defs.Radious, Defs.Radious) y radio Defs.Radious. **Defs** es una clase que define las constantes importantes para la competencia que se comenta a continuación.

int **getBattleId():** retorna un identificador que es único para cada batalla.

int **getLiveTime():** representa el tiempo de simulación en una batalla.

CONSTANTES

Para el desarrollo de la competencia se han definido constantes que pueden resultar muy útiles en el desarrollo de las estrategias.

Supongamos que se quiere dar lucha a otro MO que está en el lugar *x, y*; pero se quiere verificar que la energía de mi MO (que está en *pos.x, pos.y* y que tiene el nivel de energía *ene*), menos lo que perderá por moverse (dado por la contante *Defs.LESS_MOVE*), no implique llegar con una energía menor que la del oponente y por lo tanto ser vencido. El siguiente código contempla dicha situación:

```
if (Petri.inDish(x,y) && Petri.canCompite (pos.x,pos.y,x,y)){
    if (Petri.getEnergy(x, y) < ene-Defs.LESS_MOVE)
      {    … si voy a la posición x,y a competir…  }
```

Todas las constantes son las definidas en la siguiente clase:

```
#ifndef DEFS_H
#define DEFS_H
```

```
class Defs {
   public:
     // Radio del disco de Petri
          static const int RADIUS = 25;

    // Diámetro del disco de Petri
        static const int DIAMETER = 2 * RADIUS;

   // Número de MOs iniciales en cada colinia
        static const int MO_INITIAL = 50;

    // Máximo de nutrientes en cada distribución
        static const float MAX_NUTRI = 5000.0f;

    // Energía inicial de cada MO
         static const float E_INITIAL = 1000.0f;

    // Mínima energía para vivir
         static const float LESS_LIVE = 5.0f;

     // Mínima energía para poder moverse.
         static const float LESS_MOVE = 10.0f;

     // Longitud maxima de las  string en el entorno
         static const int MAX_LENGTH = 13;

};
#endif
```

Entorno de la Competencia

Se divide en dos paneles que permiten administrar las batallas y los torneos de forma sencilla. Además posee un menú con algunas opciones y configuraciones:

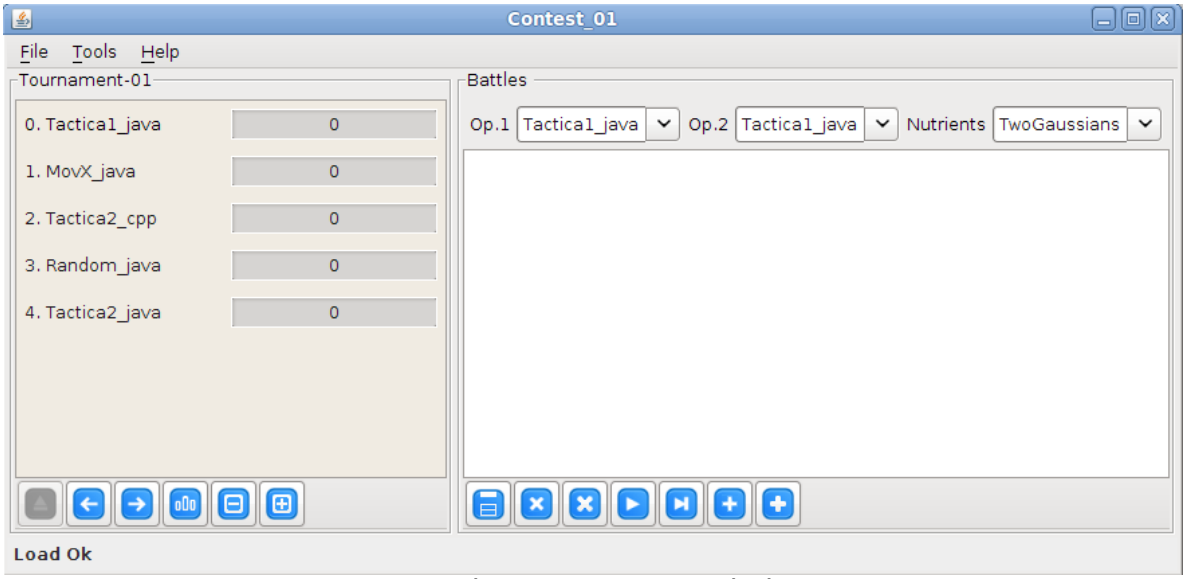

Figura 6: Entorno de la competencia

Panel de torneos: visualiza el nombre de cada colonia oponente y su respectiva energía acumulada en dicho torneo. En la parte inferior hay una lista de botones que permiten: agregar/eliminar una nueva colonia, navegar los torneos, visualizar el ranking, agregar/eliminar algún torneo.

Panel de batallas: permite crear y ejecutar batallas en el torneo actual. Se pueden seleccionar (usando las listas seleccionables que están en la parte superior del panel) los oponentes y distribución de nutrientes, y a continuación usar el botón “agregar batalla seleccionada” para agregar una batalla. También se puede usar el botón “agregar todo” para agregar las batallas que aún no se han agregado a la lista. Para correr las batallas hay dos opciones: **run** y **run all**, donde run corre la batalla seleccionada y run all las ejecuta secuencialmente a todas y cada una.

Menu: se divide en tres opciones (File, Tools y Help):

Menu File: administraciones básicas del contest.

Menu Tools: configuraciones(preferences) y reportes (report). En preferences se puede configurar el nombre del contest, el path del proyecto, el tiempo de pausa (en milisegundos) entre cada batalla, el modo en que corre la aplicación (programador o competencia), la distribución de nutrientes permitida en el contest y la interface gráfica que muestra las batallas.

Menu Help: ayuda e información del entorno.

Pasos para Comenzar

¿Qué hay que saber para comenzar?

Simplemente 2 cosas:

1. Cómo setear los modos del entorno

   Modo programador: está ideado para los programadores, ya que la aplicación no guarda la información respectiva a las batallas que se van realizando.
   Modo competencia: se usa solamente cuando se realizan los encuentros.

2. Cómo hacer correr los MOs:

   Sólo hay que copiar el código fuente en la carpeta Contest-[año]/MOs del proyecto y correr la aplicación.

¿Qué se debe tener en cuenta a la hora de codificar un MO?

La clase abstracta Microorganism es la que declara los métodos necesarios para definir el comportamiento de una colonia, por lo tanto, el código del concursante deberá heredar de la misma.
A continuación se muestra la implementación de la clase Microorganism codificada en Java:

```
public abstract class  Microorganism  {
// Posicion absoluta del Microorganismo en el entorno.
protected Position pos;

// Energia actual  del microorganismo.
protected float  ene;

// identificador del microorganismo
protected int id;

// Permite a un   Microorganismo moverse a una posicion relativa  de su posicion actual  (pos).
public  abstract  void move(Movement  mov);

// Permite a un   microorganismo duplicarse.
public  abstract  boolean mitosis();

// Nombre   de la  colonia de Microorganismos.
public abstract String  getName();

// Autor del  codigo.
public abstract String getAuthor();

// Filiacion del autor del codigo!!
```

```
public abstract String  getAffiliation();
}
```

Ahora, la implementación en C/C++:

```
class  Microorganism{
protected:
int id;
float  ene;     // energia del microorganismo.
Position pos; // posicion absoluta  del microorganismo  dentro del petri.

public:
/*
*   Invoca a un   Microorganismo para  que se  mueva.
*   Movement  mov:  movimiento relativo  a la posicion (pos)
*   del microorganismo.
*/
virtual void  move(Movement &mov)  {
  mov.dx  =  mov.dy =  0;
};

/*
*   retorna true si el microorganismo va a realizar
*   el  proceso de mitosis(si  se va a reproducir).
*/
virtual  bool mitosis(){
  return  false;
};

/*
*   Retorna el  nombre de  la  colonia de microorganismos.
*/
virtual string getName() const  {
  return  "Nombre: no   sabe";
};

/*
*   Retorna el  nombre del  autor  del  codigo.
*/
virtual string getAuthor()  const {
  return "Author:  no   sabe";
};

/*
*   Retorna la  universidad a la  que pertenece el  microorganismo.
*/
virtual string getAffiliation() const  {
   return "Aff:  no   sabe";
};
};
```

EJEMPLO SIMPLE DE UN MO

Supongamos que queremos crear un MO cuya táctica es moverse en forma aleatoria. Para lograr nuestro objetivo, heredamos de la clase Microorganismo, como indicamos en los párrafos anteriores, e implementamos los métodos abstractos.

Implementación en Java:

```
//archivo  RandomColony.java.
import  java.util.Random;
import  lib.Microorganism;
import lib.Movement;


public class  RandomColony  extends  Microorganism{

  public  void move(Movement  mov)   {
    // Desplazamiento aleatorio!!
    mov.dx  =  new  Random().nextInt(3)-1;
    mov.dy  =  new  Random().nextInt(3)-1;
  }


  public  boolean mitosis()  {
    // Nunca se  reproduce  !!
    return  false;
  }


  public String  getName() {
    return  "Random";
  }


  public String  getAuthor()  {
    return "Author";
  }


  public String getAffiliation()  {
    return "Universidad XX – Facultad YY";
  }
}
```

Implementación en C/C++:

```
//archivo  RandomColony.h.

#ifndef RANDOM_COLONY_H
#define RANDOM_COLONY_H


#include<cstdlib>
#include  <iostream>
using  namespace std;
#include  "Microorganism.h"
#include "Petri.h"


class Movx: public  Microorganism{

  public:

    Movx(){
      srand(time(NULL));
    }

    void  move(Movement &mov)  {
      mov.dx  =  (rand()  %  3) -1;
      mov.dy  =  (rand()  %  3) -1;
    };


    bool mitosis(){
      return  false;
    };

    string  getName(){
      return  "Movx_cpp";
    };

    string  getAuthor(){
      return "Author";
    };

    string getAffiliation(){
      return  "UTN-FRSF";
    };

};

#endif
```

EJEMPLO DE UN MO MÁS AVANZADO

La clase Petri declara los métodos necesarios para que un MO pueda obtener información necesaria para tomar decisiones. Los métodos que posee esta clase son los siguientes:

public **Class getOponent**(int x, int y): Retorna la clase del MO que se encuentra en la posición (x,y). si la posición está libre se retorna -1.

public float **getEnergy**(int x, int y): Retorna la energía del MO que se encuentra en la posición (x,y). si la posición está libre se retorna 0.

public float **getNutrient**(int x, int y): Retorna la cantidad de nutrientes que hay en la posición (x, y). Además, la clase Petri provee dos métodos para facilitar la programación de un MO:

public boolean **canCompite**(): retorna true si el MO en la posición (x1,x2) es oponente del MO en la posición (y1,y2).

public boolean **inDish**(): retorna true si el MO en la posición (x,y) esta dentro del entorno, es decir, si pertenece al círculo con centro en (Defs.Radious, Defs.Radious) y radio Defs.Radious. **Defs** es una clase de utilidad que define las constantes importantes para la competencia, por ejemplo: cantidad de energía inicial de cada MO(E_INITIAL), radio del entorno (RADIOUS), cantidad de MOs que se crean al inicio de la competencia (MO_INITIAL), entre otras.

Ahora utilizando la información que nos provee la Clase Petri, podemos crear un MO que realice movimientos más avanzados. En los siguientes ejemplos trataremos de implementar una estrategia que ataca si hay un enemigo en una posición adyacente, en caso contrario se mueve a una posición en donde hay más nutriente.

Implementación en Java:

```
package lib.MOs;

import  java.awt.Point;
import  lib.Defs;
import  lib.Microorganism;
import lib.Movement; import lib.Petri;


public class  Tactica2  extends Microorganism{

  public  void move(Movement  mov){

    Point p  =  new  Point();
    for(int i =  -1; i <=  1; i++){
      for(int j =  -1; j <=  1; j++){
```

```
        try{
          p.x =  pos.x+i;
          p.y =  pos.y+j;
          if(Petri.getInstance().inDish(p)  &&
            Petri.getInstance().canCompite(pos, p)){
            if(Petri.getInstance().getEnergy(p.x,  p.y) <  ene-Defs.LESS_MOVE){
              mov.dx  =  i; mov.dy  =  j;
              return;
            }

          }

        }  catch(NullPointerException  ex){ }

      }
    }

    for(int i =  -1; i <=  1; i++){
      for(int j =  -1; j <=  1; j++){

        try{
          p.x =  pos.x+i; p.y =  pos.y+j; if(Petri.getInstance().inDish(p)){
            if(Petri.getInstance().getNutrient(pos.x,  pos.y)*1.4 <
              Petri.getInstance().getNutrient(p.x,p.y)){
              mov.dx  =  i; mov.dy  =  j;
              return;
            }
          }

        }  catch(NullPointerException  ex){}

      }
    }
  }


  public  boolean mitosis()  {
    return  false;
  }


  public String  getName() {
    return "Tactica2_java";
  }



  public String  getAuthor()  {
    return "Tactica2";
  }
```

```
  public String getAffiliation()  {
    return  "UTN-FRSF";
  }

}
```

Implementación en C/C++:

```
#ifndef MOVX_H
#define MOVX_H


#include<cstdlib>
#include  <iostream>
using  namespace std;
#include "Defs.h"
#include  "Microorganism.h"


class Movx: public  Microorganism{

  public:

    Movx(){}

    void  move(Movement &mov)  {

      Position p;

      for(int i =  -1; i <=  1; i++){
        for(int j =  -1; j <=  1; j++){
          p.x =  pos.x+i;
          p.y =  pos.y+j;
          if(petri.inDish(p) &&   petri.canCompite(pos, p)){
            if(petri.getEnergy(p.x,  p.y) <  ene-Defs::LESS_MOVE){
              mov.dx  =  i; mov.dy  =  j; return;
            }
          }
        }
      }



      for(int i =  -1; i <=  1; i++){
        for(int j =  -1; j <=  1;  j++){
          p.x =  pos.x+i; p.y  =  pos.y+j;
```

```
            if(petri.inDish(p)){
              if(petri.getNutrient(pos.x, pos.y)*1.4 <  petri.getNutrient(p.x,p.y)){

                mov.dx  =  i;
                mov.dy  =  j;
                return;

              }
            }
          }
        }
      };


      bool mitosis(){
        return  false;
      };


      string  getName(){
        return "Tactica2_cpp";
      };


      string  getAuthor(){
        return "Author";
      };


      string  getAffiliation(){
        return  "UTN-FRSF";
      };

};


#endif
```