Programming 3

Programming Assignment: An Object-Oriented Version of Conway's Game of Life

Authors: David Rizo Valero, Pedro José Ponce de León Amador
Translation into English: Juan Antonio Pérez-Ortiz
Proofreading: Alicante University Language Services

University of Alicante, June 2014

These are the instructions for a programming assignment of the subject Programming 3 taught at University of Alicante in Spain. The objective of the assignment is to build an object-oriented version of Conway's game of life in Java. The assignment is divided into four sub-assignments.

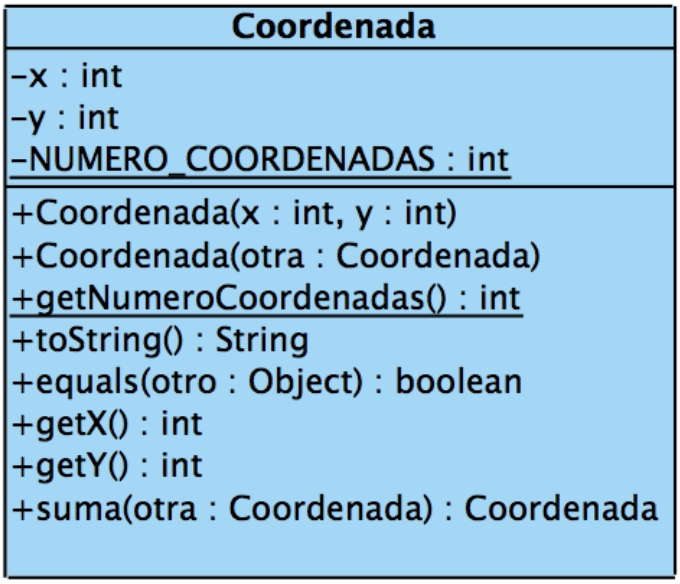
First Programming Assignment

Introduction

In this first assignment, you will implement a new Java class from an already existing class implemented in C++. This class, which is a part of the model you will be working on during the semester, is called Coordenada. It will be used to represent both the coordinates of a cell on the game board, and the dimensions of this board. The target of this assignment is to find the main differences between Java and C++.

The checkerboard in Conway's game of life can be represented as a two-dimensional matrix; therefore, in order to refer to a particular cell, a row (variable x) and a column (variable y) need to be known. This model will also be used to describe the size of the board: the variable x will represent the number of columns (width) and the variable y will represent the number of rows (height). Class *Cordenada* is shown in UML notation in figure 1.

Figure 1.1. UML class diagram.



The code in C++ available under directory **cpp** (and which you can download from here) includes member functions for the destructor and the assignment operator. As they do not have an equivalent in Java, they have to be ignored when writing the Java code. The == operator is equivalent to the 'equals' method in Java.

In Java, the main function has to be encapsulated within a class as a static method. Class *Main1C1314* (which you can download from here) contains the same operations also implemented in C++ in file main.cc; use *Main1C1314*it as the main class in your solution to the assignment.

Documenting your code

Source files must include all necessary comments in Javadoc format. These comments must be defined at least for:

- **Files:** annotation **@author** must include the name and ID number (DNI) of the authors.

- **Classes:** 3 lines describing the main purpose of the class.
- **Operations:** 1 line for trivial functions; 2 lines, input and output parameters, and dependent functions for the rest.
- **Attributes:** 1 line describing each attribute.

Package structure and directories

Package structure is implemented in Java at the filesystem level by conveniently using directories. This assignment has to be organised into two packages or directories:

- **modelo** will contain the file Coordenada.java
- **mains** will contain Main1C1314.java

All this directory structure should be compressed in a file **prog3-1-13-14.tgz**, no larger than 500 KB, by typing this in the root directory containing the source files:

tar czvf prog3-1-13-14.tgz modelo mains

Source files will contain the documentation in Javadoc format, but you do not need to include the HTML files generated by Javadoc in the compressed file.

Submission

Your programming assignment should be programmed in Java under the GNU/Linux operating system. It must compile without errors with the version of the JDK 1.6 compiler installed in lab computers. All the assignments must be done individually.

Minimal requirements for grading your assignment

- Your program must run without any errors.
- Unless otherwise stated, your program must not emit any kind of message or text through standard output or standard error.
- Source files will be compiled at the moment of evaluation.

- Names of **all the identifiers** provided in this document must be rigorously respected; you must adhere to their name, visibility and type.
- Your code must be conveniently documented and significant content has to be obtained after running the Javadoc tool.

Remarks

- Although not recommended, you may add to your classes as many private attributes and methods as you wish. Notice, however, that you must implement all the methods indicated in this document and make sure that they work as expected, even if they are never called in your implementation.

Marking

Testing your assignment will be done automatically, which means that your program must conform strictly to the input and output formats given in this document, as well as the public interfaces of all the classes: do not introduce changes neither in the method signatures nor in their behaviour. For instance, method ModeloCoordenadas(int,int) must accept two integer values as arguments and store them in the corresponding attributes.

More information on how the marking of the programming assignments will be carried out may be found in the course syllabus available through the Virtual Campus¡.

Software for plagiarism detection will be used by the instructors. Each student is responsible for being aware of what constitutes cheating and plagiarism and avoiding both. Penalties for handing in plagiarised work will range from at least a final mark of zero for the whole course in the current assessment period (*convocatoria*) to stronger disciplinary measures. Students who share their work for the purpose of cheating are subject to the same penalties as the student who commits the act of cheating.

Deadline

Work handed in late will not be accepted.

Second Programming Assignment

Introduction

The game of life, as originally proposed by Conway, will be implemented in this assignment. The design of the application has been made considering future extensions of the game, such as non-square cells, boards with more than two dimensions, or different rules.

Our model will meet the following criteria:

- boards with square cells and any size can be created (restriction to positive sizes will be implemented in later assignments);
- patterns of any size can be created by indicating which cells are alive and which ones are dead;
- patterns can be placed any time at arbitrary positions on the board;
- an unlimited number of patterns can be placed on the board (a record of all the patterns loaded needs to be maintained);
- the board is updated by following Conway's rules;
- it is possible to obtain a string representation (with ASCII characters only) of the current configuration of the board; this string can be used to print or save the board in a file.

As the complexity of the application will be raised in later assignments, you will need to develop a set of unit tests for each method you implement. Unit testing allows the programmer to refactor code later, and make sure the module still works correctly (e.g., in regression testing). The procedure is to write test cases for all the methods so that whenever a change causes a fault, it can be quickly identified and fixed. For this purpose, you will use the JUnit unit testing framework and benefit from its integration into the Eclipse development environment.

Class diagram

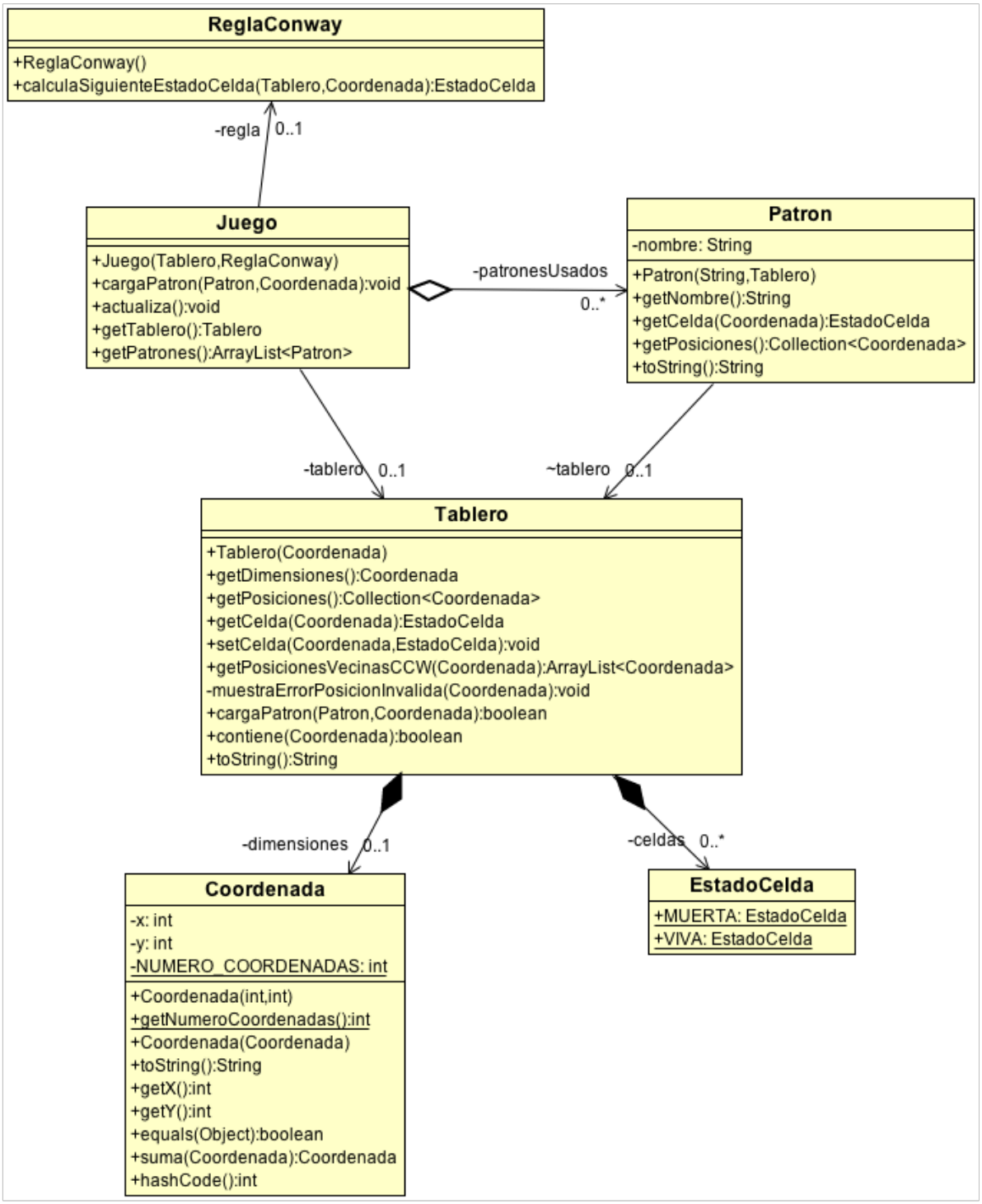


Figure 2.1. Class diagram.

Every new class will belong to package **modelo**, already created for the first assignment.

In this section, all the methods to be implemented are described. Attributes or relationships will not be covered as they are already shown in the UML diagram. Elements which do not change with regard to the previous assignment will not be explained again. *Setters* or *getters* which simply return the current value of a property, or set a property to a new value (after checking it is valid) are not commented either. In case parameter validation fails in a setter, the value to be assigned will be indicated in these instructions.

Coordenada

This class has the same methods and attributes as in the first assignment and a new method *hashCode* which yields a numerical value from a combination of the values of the attributes of the class. There are different ways to calculate this value, but for our purposes the method which automatically generates the Eclipse environment will suffice:

int hashCode ()
Returns the value 31(31+x)+y.

EstadoCelda

EstadoCelda is a Java enumerated type with values MUERTA (dead) and VIVA (alive).

Tablero

This class represents the matrix of cells used in the game of life. With the aim of having an internal representation which is independent from the type of board, thus easing its possible extension to more dimensions, do not use a two-dimensional array (a vector of vectors), but a data structure which, given a coordinate, returns the state of the referenced cell. Although you may use the data structure which you consider most appropriate, it is recommended to use a hash table (a data structure included in Java standard libraries), where for each board position the state of the corresponding cell will be stored. Therefore, if this structure is used to implement the composition, the private member *celdas* will be defined as:

```
private HashMap<Coordenada, EstadoCelda> celdas;
```

The class Coordenada will also be used to represent the size of the board (*x* will be used for the width and *y* for the height): a board with the size represented by coordinate (3.5) could be graphically represented as in figure 2:

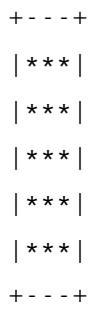


Figure 2.2. Example of a board.

Alive cells (in figure 2, all of them) are represented with an asterisk where position (0.0) is located in the upper left corner, and position (2.4) in the lower right corner.

Tablero(Coordenada dimensiones)
Stores the board size (*parameter dimensiones*) and initializes the cell map by inserting a dead cell (represented with the enumerated type EstadoCelda) for each position. Note that verifying whether the coordinate that represents the size is correct will not be done in this assignment.

Collection<Coordenada> getPosiciones() returns the set of positions for the board (see method keySet() in class HashMap if you are using this class in your implementation).

void muestraErrorPosicionInvalida(Coordenada c) prints a message through standard error (followed by an end of line character) indicating that the position is not within the board limits; the format and text for the message will be exactly as the one in the following example:

"Error: La celda (12,3024) no existe"

EstadoCelda getCelda(Coordenada posicion) returns the state of the cell corresponding to the given position. If such a cell does not exist, the same error message already shown for the method muestraErrorPosicionInvalida will be emitted through standard error, and null will be returned.

void setCelda(Coordenada posicion, EstadoCelda e) assigns the state *e* to the cell located at *posicion*, overwriting the previous state value (see method *put* in class HashMap if you are using this class in your implementation). If such a cell does not exist, the same error message already shown for the method muestraErrorPosicionInvalida will be emitted through standard error, and the control will be then returned to the caller.

ArrayList<Coordenada> getPosicionesVecinasCCW(Coordenada posicion) returns a vector of positions in anti-clockwise order comprising the neighbouring cells of the cell indicated in the parameter. The 8 neighbouring positions of a cell are those surrounding it as shown in figure 3. In the case of cells located at the board end, the number of neighbours will be lower. See figure 3 in order to identify the cell which will begin the traversal.

	neighbour 0	neighbour 7	neighbour 6	
	neighbour 1	**position**	neighbour 5	
	neighbour 2	neighbour 3	neighbour 4	

Figure 2.3. Cells traversed in anti-clockwise order.

boolean cargaPatron(Patron patron, Coordenada coordenadaInicial), given a pattern, copies the states of its cells to the board beginning at the position indicated in the second parameter. In order to calculate positions relative to the second parameter, you can use the method *suma* in class Coordenada. If the pattern does not fit in the board, an error will be shown using method muestraErrorPosicionInvalida (indicating the first coordinate which is out of the board), the board will remain unchanged and false will be returned; otherwise, true will be returned.

boolean contiene(Coordenada posicion) returns true only if the position belongs to the set of board cells already initialised in the constructor.

String toString() returns a string representation of the board as shown in figure 2. Alive cells are shown with an asterisk, and dead ones with a white space. Boundaries are represented with characters '+', '-' and '|'. All cells in figure 2 are alive. Figure 4 shows a board with two dead cells at coordinates (0.0) and (2,3).

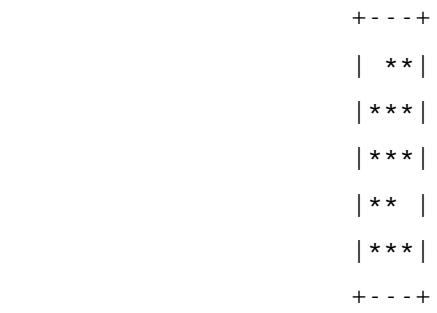


Figure 2.4. An example of a possible output emitted by System.out.println(tablero.toString()).

Patron

A pattern represents a set of alive cells. Boards will be used to implement patterns. Note: composition, and not inheritance, is used to represent this relationship because Tablero contains methods (for example, getPosicionesVecinasCCW) which a pattern cannot contain. Figure 5 shows an example of a well-known pattern.

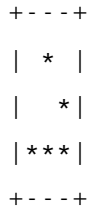


Figure 2.5. The glider is a pattern that travels across the board in Conway's Game of Life.

Patron(String nombre, Tablero tablero)
This method simply stores the parameter values in the corresponding attributes.

EstadoCelda getCelda(Coordenada posicion)
Collection<Coordenada> getPosiciones()
These two methods delegate their behaviour to the homonymous methods in Tablero.

String toString() returns the pattern name, followed by an end-of-line character, and followed by its string representation (using the same format used in Tablero).

ReglaConway

This class represents a rule which defines how the states of cells evolve during the game.

EstadoCelda calculaSiguienteEstadoCelda(Tablero tablero, Coordenada posicion) returns the new state for the given cell. The rules are:

- Any live cell with fewer than two live neighbours dies, as if caused by under-population.
- Any live cell with two or three live neighbours lives on to the next generation.
- Any live cell with more than three live neighbours dies, as if by overcrowding.
- Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction; otherwise, it remains dead.

Note that this method only returns the value for the next state without changing the state of the cell on the board. The next generation is created by applying the above rules simultaneously to every cell in the current board (births and deaths occur simultaneously), and performing this is a responsibility of class Juego.

Juego

This class orchestrates the rest of classes. By conveniently using the board specified in its constructor, the rule to use (only Conway's in this assignment), and the different patterns loaded onto the board, the class makes the cells evolve to the next generation after every call to actualiza().

Juego(Tablero tablero, ReglaConway regla) initialises instance attributes and stores the given parameters.

void cargaPatron(Patron p, Coordenada posicionInicial) tries to load the pattern and store it on *patronesUsados*. In case the pattern does not fit (see the boolean returned by Tablero.cargaPatron), a message followed by the end-of-line character will be emitted through standard error, such as the following one for a pattern called *Intermitente*:

Error cargando plantilla Intermitente en (23,45)

void actualiza() makes all cells evolve simultaneously using ReglaConway.calculaSiguienteEstadoCelda. This operation must be performed in two steps. In the first one, a temporary data structure will be used to store the state for the next generation for every cell; in the second step, the state for all the cells on the board will be changed accordingly.

Example

Main2C1314.java can be used as an example of a game: different patterns are created; then, they are tried to be placed them onto the board (the process will fail sometimes), and the game is executed for 5 generations (ticks). The result can be seen in this capture of the console output. Error messages are shown in red.

Unit testing

You should write your own JUnit tests for each of the implemented methods. As a starting point, you may consider reusing the tests included in the automatic evaluator of the first assignment.

Documentation

Your source files must include all the comments in Javadoc format as indicated in the first asignment. You do not need to include the HTML files in the compressed file generated by the Javadoc tool.

Package and directory structure

The same as in the first assignment. All the new classes belong to package **modelo**.

Evaluation

Evaluation criteria and methodology are the same than those used in the first assignment.

Submission

Your solution to this assignment must be programmed in Java under the GNU/Linux operating system. It must compile and run without errors with the version of the JDK 1.6 installed in lab computers.

The whole directory structure must be compressed and packaged in a file **prog3-2-13-14.tgz**, no larger than 500 kB; unit tests must NOT be included in this file. The main method to be included in package mains will be the one discussed in section Example above.

Third Programming Assignment
Game of life: exceptions and inheritance

Introduction

In this third assignment, you will add exception handling to the second assignment and extend its behaviour by means of inheritance.

As the complexity of the application will be rising in later assignments, you will need to develop a set of unit tests for each method you implement. Unit testing allows the programmer to refactor code later, and make sure the module still works correctly (e.g., in regression testing). The procedure is to write test cases for all the methods so that whenever a change causes a fault, it can be quickly identified and fixed. For this purpose, you will use the JUnit 4 unit testing framework and benefit from its integration into the Eclipse development environment.

Part 1: exceptions

In order to improve your implementation, console error messages will be replaced with exception handling. Consequently, Tablero.*muestraErrorPosicionInvalida* will be removed for this assignment, and the class hierarchy depicted in figure 1 will be used. Note that the set of exceptions include exceptions to be declared in the throws clause of different methods

(ExcepcionPosicionFueraTablero, ExcepcionCoordenadaIncorrecta and its derived classes), and runtime exceptions (ExcepcionEjecucion) which will be used to catch programming errors which should not happen; for instance, *ExcepcionArgumentosIncorrectoswill* be thrown when a parameter which is expected to be non-null is null.

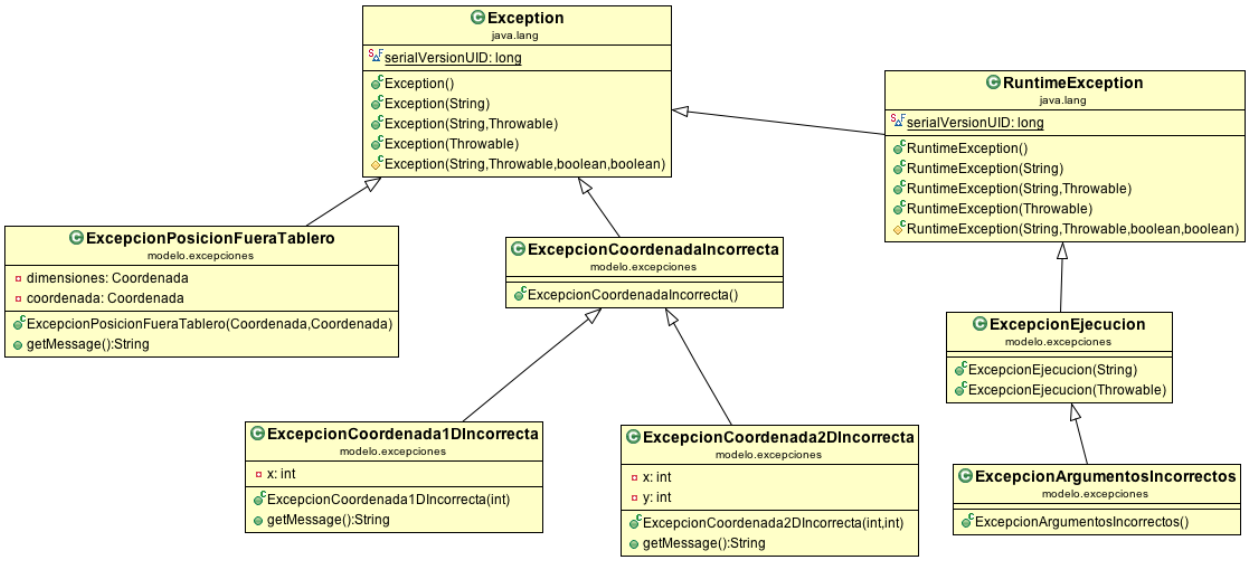


Figure 3.1. Class diagram for the exceptions.

Even though the UML diagram does not include them, public getters for all shown exceptions must be implemented. Besides, all strings returned in getMessage() must be decided by the student.

In order to add exception handling, you will have to make the following changes to your implementation of the second assignment:

Class Coordenada

- The constructor Coordenada(int x, int y) will throw ExcepcionCoordenadaIncorrecta when x or y are negative.
- The copy constructor will throw *ExcepcionArgumentosIncorrectos* when the parameter is null.

- The method *suma* will throw *ExcepcionArgumentosIncorrectos* when its parameter is *null*. In addition, this method has to declare ExcepcionCoordenadaIncorrecta in its throws clause as it uses the constructor of Coordenada which could throw this exception.

Class Tablero

- The constructor from an object of type Coordenada will throw *ExcepcionArgumentosIncorrectos* when the coordinate is null.
  - When initialising cells, the constructor in Coordenada is used; this constructor may throw exception ExcepcionCoordenadaIncorrecta. Since you are invoking, as a programmer, this constructor in a controlled way and you are supposed to only use correct coordinates, you have to catch this exception and re-throw it as ExcepcionEjecucion, setting the caught exception as parameter of the copy constructor:
    - In the initialization loop…

```
try {
celdas.put(new Coordenada(i, j), EstadoCelda.MUERTA);
} catch (ExcepcionCoordenadaIncorrecta e) {
throw new ExcepcionEjecucion(e);
}
```

  - Note that boards with size (0.0) can be constructed, but boards with at least one negative dimension cannot.
- *getCelda* will throw ExcepcionPosicionFueraTablero when the position whose state is to be obtained is out of the board. Besides, if the parameter of type Coordenada is null, exception *ExcepcionArgumentosIncorrectos* will be thrown.
- *contiene, setCelda* and *getPosicionesVecinasCCW* will perform the same exception handling for null arguments as *getCelda*. Moreover, since *getPosicionesVecinasCCW* has the responsibility to create coordinates which should not be incorrect, it will be necessary to catch ExcepcionCoordenadaIncorrecta and re-throw ExcepcionEjecucion as in the constructor.
- In methods *setCelda* and *getPosicionesVecinasCCW* an exception ExcepcionPosicionFueraTablero must be thrown whenever the parameter is outside the board.

- *cargaPatron* throws *ExcepcionArgumentosIncorrectos* when at least one of the parameters is null. Again, it will be necessary to catch ExcepcionCoordenadaIncorrecta and re-throw ExcepcionEjecucion. In the previous assignment the case in which a coordinate was outside the grid was controlled printing an error message. In this assignment, this message has to be changed by the launch of an ExcepcionPosicionFueraTablero. In this third assignment, cargaPatron does not return a Boolean value. When the method fails, an ExcepcionPosicionFueraTablero is thrown and it is not required to return any false value.
- *toString* does not throw any declared exception; those coming from other methods must be re-thrown as ExcepcionEjecución.

Class Patron

- Its methods throw *ExcepcionArgumentosIncorrectos*; getCelda also re.throws the exception thrown by Tablero.getCelda.

Class ReglaConway

- *calculaSiguienteEstadoCelda*, besides throwing *ExcepcionArgumentosIncorrectos* when a parameter is null, throws ExcepcionPosicionFueraTablero which may come from the methods getCelda or getPosicionesVecinasCCW.

Clase Juego

- The constructor Juego(Tablero, Regla) throws ExcepcionArgumentosIncorrectos whenever any of the parameters are null.
- cargaPatron(Patron, Coordenada) throws the exceptions given by Tablero.cargaPatron.
- actualiza() catchers ExcepcionPosicionFueraTablero to throw it as ExcepcionEjecucion

Mains

After finishing your implementation of the exceptions, you can test it by using the following mains adapted from those of the second assignment: Main1C1314_P3.java and Main2C1314_P3.java.

Unit testing

You should write your own JUnit tests for each of the implemented methods. As a starting point, you may consider reusing the tests included in the automatic evaluator of the previous assignments.

Documentation

Your source files must include all the comments in Javadoc format as indicated in the first asignment. You do not need to include the HTML files generated by the Javadoc tool in the compressed file.

Package and directory structure

The same as in the second assignment. The new classes for exceptions must be included in sub-package excepciones under package modelo.

Evaluation

Evaluation criteria and methodology are the same as those used in the first assignment.

Submission

Your solution to this assignment must be programmed in Java under the GNU/Linux operating system. It must compile and run without errors with the version of the JDK 1.6 installed in lab computers.

The whole directory structure must be compressed and packaged in a file **prog3-3-13-14.tgz**, no larger than 500 kB; unit tests must NOT be included in this file. The main method to be included in package mains will be the one discussed in section Example above.

Late work will not be accepted.

Part 2: extension by means of inheritance

The second part of the assignment is focused on using inheritance to extend the application, generalising the common behaviour and properties to embrace different varieties of the game. These varieties will include:

- adding a one-dimensional board and one new rule (commonly known as Rule 30) to determine the evolution of its cells;
- as a result of the introduction of one-dimensional boards, one-dimensional coordinates will be also added;
- introducing new specific class for bi-dimensional boards with square cells; although, non-square cells will not be used in this assignment, the resulting hierarchy will easily allow them in the future.

Figure 2 shows the class diagram with the complete class hierarchy. Note that Juego and Patron barely change.

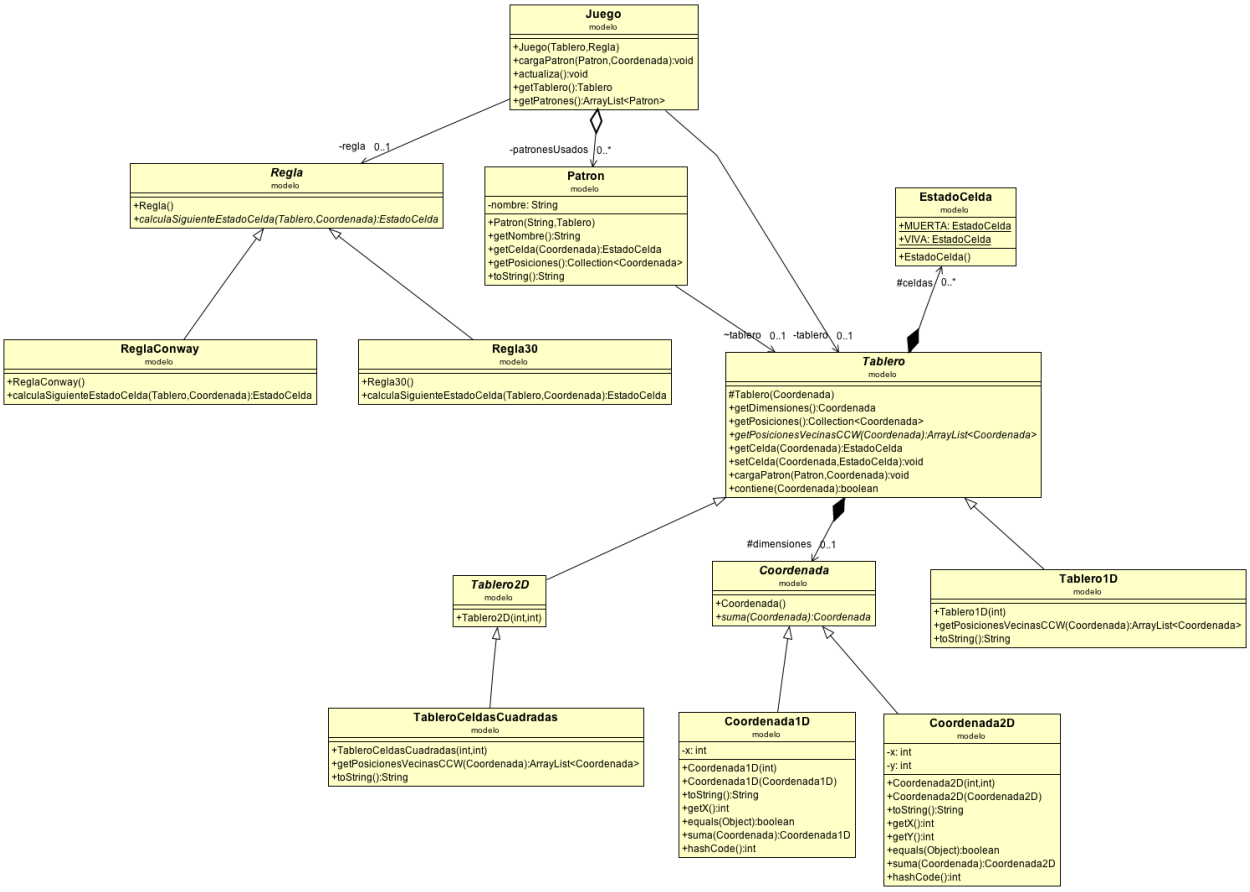


Figure 3.2: class diagram.

The description for each class follows. It is recommended that you follow the order set forth herein when implementing (and testing) your classes. When these instructions state that you have to reuse the code for the second assignment, it refers to the code for the second assignment with the addition of exceptions as described in the first part of this assignment. The fact that you need to verify that parameters are not null will not be mentioned either, since it has been already described in the first part.

EstadoCelda

This enumerated type does not change.

Coordenada, Coordenada1D y Coordenada2D

Since different types of coordinates will exist and we do not want Patron or Juego to continuously use control structures such as *if* or *switch* to determine the type of coordinate involved (either one or two dimensional), a new base class Coordenada will be created. Rename the existing class Coordenada to Coordenada2D and declare it as a child class of the new abstract class Coordenada.

Note that the method **suma** returns an object of type Coordenada in class Coordenada, but an object of type Coordenada2D in class Coordenada2D; as will be studied in class, this is known as *covariance*.

The implementation of class Coordenada1D is practically the same as the one of the old Coordenada, but restricted to component x.

The constructor Coordenada1D launches an exception ExcepcionCoordenada1DIncorrecta in the case the actual value of the x parameter is negative. Similarly, the constructor Coordenada2D launches an exception ExcepcionCoordenada2DIncorrecta in the case the actual value of any of the x parameter or y parameter are negative.

Tablero

Responsibilities for the old class Tablero are now divided among different classes. The new class Tablero is a generalisation over the old one. Only a small set of changes and some code relocation will need to be taken into account.

Note that now the relationships with Coordenada and EstadoCelda are implemented with protected attributes (they were originally private) so that they can be accessed from the child classes of Tablero.

Tablero(Coordenada dimensiones)

Its responsibility is creating (but not filling in) the cell structure and storing the board size. Size will need to be stored with this code:

this.dimensiones = dimensiones

This contradicts what has been studied in the theoretical sessions for the implementation of composition relationships. However, since Coordenada is an abstract class, thissomething like this is not allowed.dimensiones = new Coordenada(dimensiones). For this assignment, the creation of the object will be a responsibility of the constructors of the derived classes; have a look, for example, at the beginning of the constructor of Tablero2D:

```
public Tablero2D(int ancho, int alto) throws ExcepcionCoordenadaIncorrecta {
        super(new Coordenada2D(ancho, alto));
```

getDimensiones, getPosiciones, cargaPatron, getCelda, setCelda, cargaPatron, contiene

These methods will remain unchanged.

Tablero2D

public Tablero2D(int ancho, int alto)

This method invokes the constructor in the parent class, and then creates the cells of the board and initialises them to dead, as in the previous assignment. The class has no additional responsibilities.

TableroCeldasCuadradas

This class represents a bi-dimensional board whose cell shape is known and, consequently, neighbour cells can be obtained.

As an equivalent of the board in the second assignment can now be found in this class, you can download new versions of the old main methods: Main1C1314_P3b.java and Main2C1314_P3b.java.

The constructor of this class simply calls the constructor in the super class.

ArrayList<Coordenada> getPosicionesVecinasCCW (Coordenada posicion)

The code in the old getPosicionesVecinasCCW can be reused after adding the necessary castings to Coordenada2D.

String toString()

This code is similar to the old one.

Tablero1D

Tablero1D(int ancho)

This method invokes the constructor in the parent class and then creates the cells of the board and initialises them to the dead state.

ArrayList<Coordenada> getPosicionesVecinasCCW(Coordenada posicion)

Neighbour cells of position x comprise positions **(x-1)** and **(x+1)**, strictly following this order. The position must be checked so that it is not off the board.

String toString()

A single row is printed in this case. The character '|' is used to delimit the start and end of the board; '*' is used for the alive cells and a space for the dead ones. The line ends with an end of line character '\n':

```
|***  * * *|
```

Patron

This class remains exactly the same. Now it can be understood why the first design included a composition with Tablero instead of inheritance.

Juego

This class remains exactly the same.

Regla

An abstract class which only contains the declaration of the method in the UML diagram.

ReglaConway

This class inherits from Regla. The code in calculaSiguienteEstadoCelda is the same as in the first assignment.

Regla30

This represents the rule used for computing the new state in one-dimensional boards. See http://en.wikipedia.org/wiki/Rule_30. Let ABC be three adjacent cells, where B is the cell whose new state needs to be computed; if 1 means a live cell and 0 a dead one, the new state of the cell will be dead when the pattern is 111, 110, 101, 000. Border cells with a single neighbour will always be dead cells.

Here you are a new main (Main3C1314.java) which uses Tablero1D with this rule Regla30 and which is supposed to generate the output in the file salida_p3_regla30.txt.

Note: in this assignment, you do not have to check whether castings are performed with the right type: if a method expects an object of class Coordenada2D it will never receive an object of class Coordenada1D and vice versa (for example, Tablero1D.getPosicionesVecinasCCW() will always receive an object of class Coordenada1D) in the evaluation tests.

Exceptions

This a list of all the methods in the application throwing exceptions. Remember that all of them need to be added to the 'throws' declaration of the corresponding method, unless they derive from RuntimeException. Also, remember that exceptions thrown by a method implemented in a child class need to be the same as those thrown in the abstract methods in the parent class.

CLASS	METHOD	THROWN EXCEPTIONS
Coordenada	*suma(Coordenada)*	ExcepcionArgumentosIncorrectos

		ExcepcionCoordenadaIncorrecta
Coordenada2D	Coordenada2D(int,int)	ExcepcionCoordenadaIncorrecta
	Coordenada2D(Coordenada)	ExcepcionArgumentosIncorrectos
Coordenada1D	Coordenada1D(int,int)	ExcepcionCoordenadaIncorrecta
	Coordenada1D(Coordenada)	ExcepcionArgumentosIncorrectos
Tablero	Tablero(Coordenada)	ExcepcionArgumentosIncorrectos
		ExcepcionEjecucion
	getCelda(Coordenada)	ExcepcionPosicionFueraTablero
		ExcepcionArgumentosIncorrectos
	contiene(Coordenada)	ExcepcionArgumentosIncorrectos
	setCelda(Coordenada,EstadoCelda)	ExcepcionArgumentosIncorrectos
		ExcepcionPosicionFueraTablero
	cargaPatron(Patron, Coordenada)	ExcepcionPosicionFueraTablero
		ExcepcionArgumentosIncorrectos
		ExcepcionEjecucion
	getPosicionesVecinasCCW(Coordenada)	ExcepcionArgumentosIncorrectos
		ExcepcionPosicionFueraTablero
		ExcepcionEjecucion
Tablero2D	Tablero2D(int,int)	ExcepcionCoordenadaIncorrecta
		ExcepcionEjecucion
TableroCeldasCuadradas	TableroCeldasCuadradas(int,int)	ExcepcionCoordenadaIncorrecta
		ExcepcionEjecucion
	toString()	ExcepcionEjecucion
Tablero1D	Tablero1D(int)	ExcepcionCoordenadaIncorrecta
		ExcepcionEjecucion
	toString()	ExcepcionEjecucion

Patron	Patron(String,Tablero)	ExcepcionArgumentosIncorrectos
	getCelda(Coordenada)	ExcepcionPosicionFueraTablero
		ExcepcionArgumentosIncorrectos
Regla	*calculaSiguienteEstadoCelda(Tablero,Coordenada)*	ExcepcionArgumentosIncorrectos
		ExcepcionPosicionFueraTablero
Juego	Juego(Tablero, Regla)	ExcepcionArgumentosIncorrectos
	cargaPatron(Patron,Coordenada)	ExcepcionPosicionFueraTablero
		ExcepcionArgumentosIncorrectos
		ExcepcionEjecucion
	actualiza()	ExcepcionEjecucion

Fourth Programming Assignment

Game of life: interfaces

Introduction

In this assignment, you will use Java interfaces. Generally, when an application needs to interact with external resources such as databases or files, interfaces are created to guarantee the expected functionality; then, these interfaces will be implemented in order to get the real functionality.

This assignment introduces two interfaces: IParserTableros, which will be used to create boards from strings, and IGeneradorFichero, which will be used to print boards in different formats to files.

As a complement to these interfaces, and with the aim of instantiating objects of those classes implementing them, a new class Factory will be created. Similarly, a new class ParserTableros will be responsible of instantiating a board of the correct type according to a string parameter.

New error situations may arise and, consequently, two new exceptions, ExcepcionGeneracion and ExcepcionLectura, will be added to the model.

Text file generation from previous assignments will be reused here. Besides, it will be possible to generate GIF files as well as animated GIF files showing the evolution of the game. For this, your code will use the third-party library GIF4J (its JAR file can be downloaded from here). As we do not want you to mess around with all the details in the library, you will have at your disposal two classes, ImageGIF and ImageGIFAnimado, which use GIF4J in order to print squares to GIF files.

A third interface, Imprimible, will be used to make sure that the boards to be printed are able to generate a string representing themselves. This new interface will be implemented by Tablero1D and TableroCeldasCuadradas.

Finally, new mains will be provided in order to show the expected behaviour of this assignment.

Class diagram

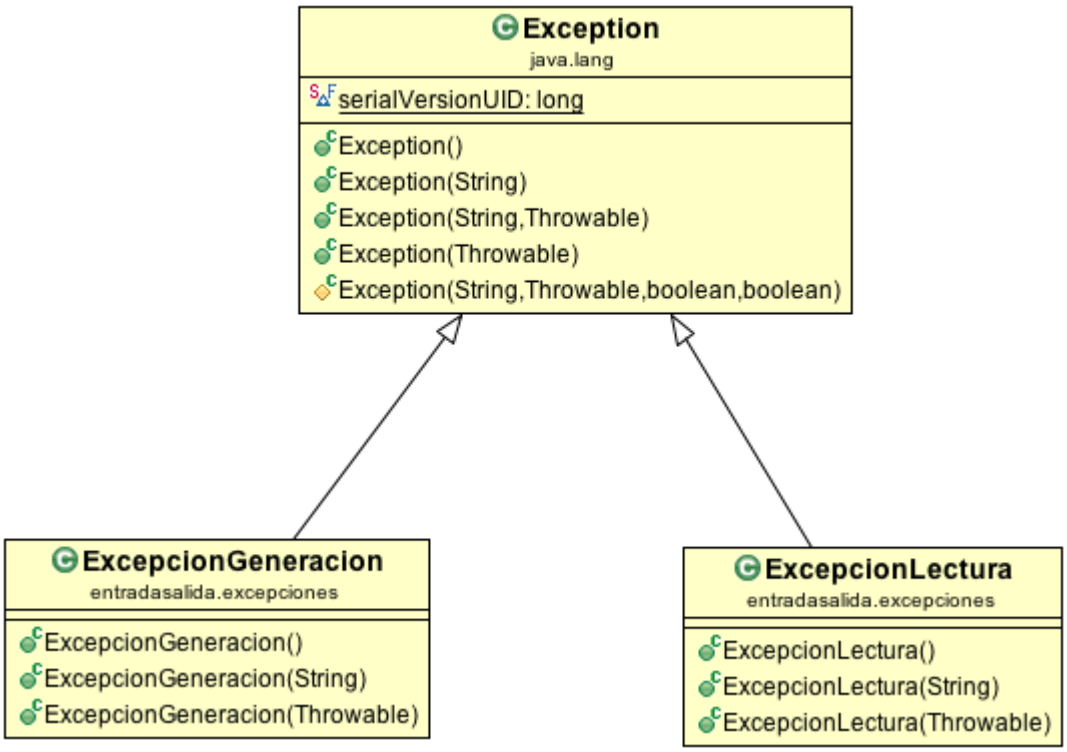


Figure 4.1. Class diagram for the new exceptions.

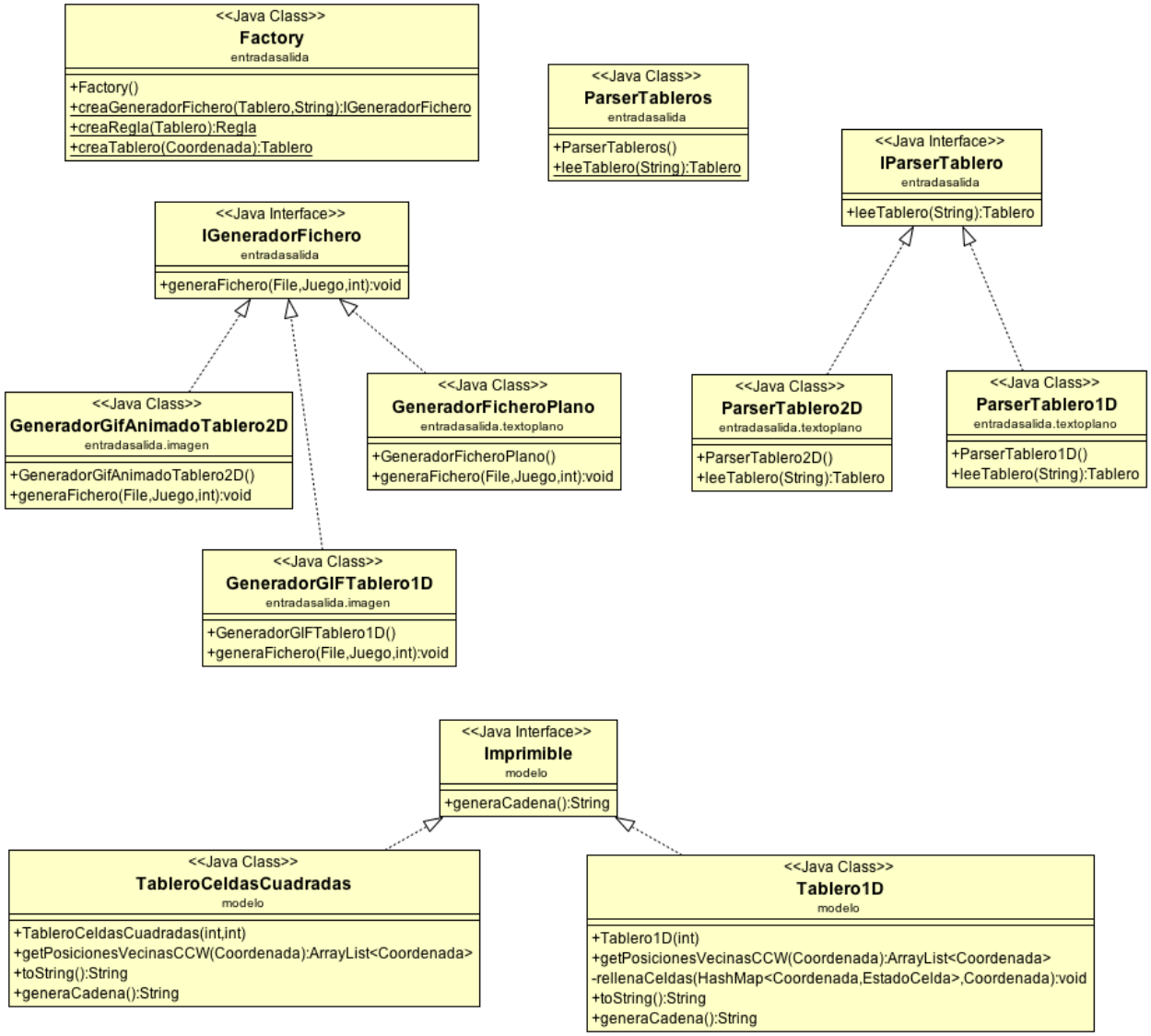


Figure 4.2. Class diagram for the fourth assignment.

Interfaces of the classes

Tablero1D and TableroCeldasCuadradas

The two classes must implement the interface Imprimible. Consequently, they must implement the method String generaCadena(). This method simply returns the same string as the already existing toString().

entradasalida.textoplano.ParserTablero1D

Default constructor can be empty.

Tablero leeTablero(String cadena)

It returns an instance of Tablero1D created from a string containing spaces for dead cells and asterisks for live cells. The size of the board will be the length of the parameter string. For example, if the input is “** **”, a board with 5 cells will be created.

This method throws:

- ExcepcionArgumentosIncorrectos when the parameter is null.
- ExcepcionLectura when the string is empty or any of its characters is neither a space nor an asterisk. The message used to create the exception object is not relevant.
- In the rest of error situations, as the error will be due to bad programming and not to issues which cannot be controlled, ExcepcionEjecucion will be thrown.

entradasalida.textoplano.ParserTablero2D

Default constructor can be empty.

Tablero leeTablero(String cadena)

It returns an instance of Tablero2D created from a string of lines separated by ‘\n’. Each of these lines represents a board row and, as with one-dimensional boards, will contain spaces for dead cells and asterisks for live cells. The number of columns in each row must be the same.

For example, for the input “*****\n** **\n*****”, a board of 3 rows and 5 columns will be created. If we printed the output of toString() upon this board, we would get:

```
+-----+
|*****|
|** **|
|*****|
+-----+
```

This method throws:

- ExcepcionArgumentosIncorrectos when the parameter is null.
- ExcepcionLectura when the string is empty or any of its characters is neither a space nor an asterisk. This exception will also be thrown when there are at least two rows with a different number of columns. The message used to create the exception object is not relevant.
- In the rest of error situations, as the error will be due to bad programming and not to issues which cannot be controlled, ExcepcionEjecucion will be thrown.

The last line may optionally end with '\n'.

entradasalida.ParserTableros

Default constructor can be empty.

Tablero leeTablero(String cadena)
After checking whether the string has one line or more than one line, it delegates board reading on ParserTablero1D or ParserTablero2D, respectively. This method throws ExcepcionLectura when the input is empty or ExcepcionArgumentosIncorrectos when the parameter is null. It also re-throws the exceptions thrown by ParserTablero1D and ParserTablero2D.

entradasalida.textoplano.GeneradorFicheroPlano

Default constructor can be empty.

void generaFichero(File file, Juego juego, int iteraciones)
This method plays (juego.actualiza()) the game for the number of ticks indicated by the parameter iteraciones. After each update, it writes to the file indicated in the first parameter the result of calling the method generaCadena. In order to ensure that the board has this method implemented, you must check that the board in the game passed as the second parameter implements the interface Imprimible. In case it does not, ExcepcionGeneracion will be thrown. In case file or juego are null, ExcepcionArgumentosIncorrectos will be thrown. If iteraciones is not positive and greater than zero, ExcepcionGeneracion will be thrown (the message will have to indicate that the number of iterations is not correct, but the concrete message is not relevant).

The standard Java class PrintWriter can be used to write to a text file. In case something goes wrong, methods in this class throw exceptions (for instance, FileNotFoundException), which have to be caught and re-thrown as ExcepcionGeneracion. The pseudo-code for the algorithm is:

```
for i=0 to iteraciones-1 do
     juego.actualiza()
     add to the file the string obtained from Tablero.generarCadena (it must
implement Imprimible)
end for
```

Notice how the same class is used to print all the different types of boards. Two examples of valid outputs can be found in files regla30.txt and juego2D.txt, which are generated by the main methods provided (output files).

entradasalida.imagen.GeneradorGIFTablero1D

Default constructor can be empty.

void generaFichero(File file, Juego juego, int iteraciones)
In case file or juego are null, ExcepcionArgumentosIncorrectos will be thrown. If iteraciones is not positive and greater than zero, ExcepcionGeneracion will be thrown (the message will have to indicate that the number of iterations is not correct, but the specific message is not relevant).

This method creates an object ImagenGIF with the same width as the game board and with height equal to the number of iterations in the parameter. Then, after each iteration of the game, live cells are printed using ImagenGIF.pintaCuadrado. This method gets as parameter a coordinate (i,j) where:
- i: the x corresponding to the cell coordinate on the board
- j: the current iteration (starting from zero).

The pseudo-code for the algorithm is:

```
gif = create GIF Image with the same height and width as the board
for y=0 to iteraciones-1 do
     for x = 0 to board width -1 do
```

```
          if cell at position (x) is alive then
               gif.pintaCuadrado(x, y)
          end if
     end for
     juego.actualiza()
end for
save gif in file
```

Any ExcepcionPosicionFueraTablero or ExcepcionCoordenadaIncorrecta must be caught and re-thrown as ExcepcionEjecucion.

entradasalida.imagen.GeneradorGifAnimadoTablero2D

Default constructor can be empty.

void generaFichero(File file, Juego juego, int iteraciones)
In case file or juego are null, ExcepcionArgumentosIncorrectos will be thrown. If iteraciones is not positive and greater than zero, ExcepcionGeneracion will be thrown (the message will have to indicate that the number of iterations is not correct, but the concrete message is not relevant).

This method will be responsible of generating an animated GIF, which is a sequence of GIFs (frames) separated by a given time delay. The animated GIF will consist of a frame for each iteration. The pseudo-code for the algorithm is:

```
gifAnimado = create Animated GIF using a delay of iteraciones ms
for i=0 to iteraciones-1 do
     fotograma = create GIF Image with the same width and height as the board
     for x = 0 to board width-1 do
          for y = 0 to board height-1 hacer
               if cell at position (x, y) is alive then
                    fotograma.pintaCuadrado(x,y)
               end if
          end for
     end for
```

```
    add frame to the animated GIF by using addFotograma
    actualiza()
end for
save file with guardaFichero
```

This method re-throws the exceptions ExcepcionGeneracion coming from other methods. Any ExcepcionPosicionFueraTablero or ExcepcionCoordenadaIncorrecta must be caught and re-thrown as ExcepcionEjecucion.

entradasalida.Factory

Default constructor can be empty. All the methods will throw ExcepcionArgumentosIncorrectos if a parameter is null.

IGeneradorFichero creaGeneradorFichero(Tablero tablero, String extension)
If the extension is “txt”, a GeneradorFicheroPlano will be created; this can be used with all the types of boards. If the extension is “gif”, an instance of GeneradorGIFTablero1D will be created if the given board is Tablero1D or GeneradorGifAnimadoTablero2D if it is Tablero2D; if the board belongs to a different type, ExcepcionEjecucion will be thrown. If the extension is not “txt” or “gif”, ExcepcionGeneracion will be thrown. Exceptions coming from constructors will be re-thrown directly.

Regla creaRegla(Tablero tablero)
It creates an object of type Regla30 for one-dimensional boards and an object of type ReglaConway for two-dimensional boards; if the board belongs to a different type, ExcepcionEjecucion will be thrown.

Tablero creaTablero(Coordenada dimensiones)
It creates an object of type Tablero1D for one-dimensional coordinates and an object of type TableroCeldasCuadradas for two-dimensional coordinates; if the coordinate belongs to a different type, ExcepcionEjecucion will be thrown. A casting between dimensions may be needed. This method re-throws ExcepcionCoordenadaIncorrecta thrown by board constructors.

Mains

A set of example mains are provided. The most important one is Main4C1314, which shows how the use of interfaces allows for generalising what otherwise would be duplicated behaviours.

Included files:

Mains and support classes for GIF generation can be found in this file. Sample outputs emitted by those mains can be found in this file.

Unit testing

You should write your own JUnit tests for each of the implemented methods.

Documentation

Your source files must include all the comments in Javadoc format as indicated in the first assignment. You do not need to include the HTML files generated by the Javadoc tool in the compressed file.

Package and directory structure

Follow the UML diagrams and the description of the interfaces.

Evaluation

Evaluation criteria and methodology are the same as those used in the first assignment.

Submission

Your solution to this assignment must be programmed in Java under the GNU/Linux operating system. It must compile and run without errors with the version of the JDK 1.6 installed in lab computers.

The whole directory structure must be compressed and packaged in a file **prog3-4-13-14.tgz**, no larger than 500 kB; unit tests must NOT be included in this file. The main method to be included in package mains will be the one discussed in section Example above. Late work will not be accepted.