



DIPARTIMENTO
DI INFORMATICA
SAPIENZA
UNIVERSITÀ DI ROMA

The Battle of Sexes

Final Java Project

Applied Computer Science and Artificial Intelligence

Course

Programming 2

Professor

Pietro Cenciarelli

Contents

1. Introduction
2. The Project
 - 2.1 Idea Behind
 - 2.1.1 Characterization and genetic inheritance
 - 2.2 Division of tasks
 - 2.2.1 Men
 - 2.2.2 Women
 - 2.3 Multi-Thread handling
3. The Simulation
 - 3.1 TheBigBang
 - 3.2 StartSimulation
 - 3.3 WakeThreads
 - 3.3.1 MenThread
 - 3.3.2 WomenThread
 - 3.3.3 ChildrenThread
4. The Graphics
 - 4.1 The Page
 - 4.2 Implementation
5. Results and conclusion

1. Introduction

This project is inspired by Chapter 9 of *The Selfish Gene*, a 1976 book by Richard Dawkins. The chapter describes “the battle of the sexes”, where a model of a population is provided, featuring two male types, the faithful (F) and the philanderer (P), and two female types, the coy (C) and the fast (S), characterised as follows (the names are from the book):

- Faithful: they are willing to engage in a long courtship and participate in rearing their children.
- Philanderer: reckless men, they don't waste time in courting women: if not immediately accepted, they move away and try somewhere else; moreover, if accepted, they mate and then leave anyway, ignoring the destiny of their children.
- Coy: they accept a partner only after a long courtship.
- Fast: if they feel so, they don't mind copulating with whoever they like, even if just met.

The FPCS payoffs table

	F	P
C	$(a - b/2 - c, a - b/2 - c)$	$(0, 0)$
S	$(a - b/2, a - b/2)$	$(a - b, a)$

The three evolutionary payoffs involved in the battle of the sexes:

a: the evolutionary benefit for having a baby

b: the cost of parenting a child

c: the cost of courtship

Is then possible to represent the payoffs resulting from a woman X engaging with a man Y with a pair (x, y) , having x as the payoff of X and y as the payoff of Y.

The battle of the sexes can be formalised as in the following matrix, which may be used in defining the evolution rules.

The Assignment

Given an initial population, the simulator will iteratively apply the rules of evolution till, if ever, an evolutionary stable state is reached, in which case the percentages of the four types are returned as result.

2. The Project

The project is structured in 2 main threads, the main program and the graphics based on JavaFx.

First, we run the Main, the program creates and starts a thread that executes the main program where we can find the simulation of the society.

Immediately after it executes the graphic.

We chose to call the class where we can find the simulation through a thread because, by doing that we can execute the thread at the same time of graphic.


If, on the other hand, we had called the *simulation* first, then the graphic part would never have started because it would have to wait to find the stability (not so common).

Instead by doing in our way we avoid that because the program and the graphic are run simultaneously and so it is possible to update the graphic's data during the running time

```
public static void main(String[] args) {  
    Pop_Controller pop1 = new Pop_Controller();  
    pop1.start();  
    launch(args);  
}
```

Pop Controller is the Thread

Population is the class responsible of launching the simulation



```
public class Pop_Controller extends Thread{  
    @Override  
    public void run() {  
        Population pop = null;  
        try {  
            pop = new Population();  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
        pop.TheBigBang();  
        try {  
            pop.startSimulation();  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

The population class created to start the simulation has different fields:

- *Current_year*: increases every iteration to simulate the time flow
- *a, b, c*: the parameters of the simulation
- *men, women, children_list*: three ArrayList elements that contain what is indicated in their names and that are handled by three threads MenThread, WomenThread and ChildrenThread.

2.1 Idea Behind

The simulation is based on the creation of an initial group of people consisting of equal numbers of individual for each type, which are going to interact with each other laying the foundation for the growth of an exponentially bigger population.

Every person “born” is an object and they can be either of types **Coy**, **Fast**, which extend the superclass **Woman**, or **Faithful** or **Philanderer**, which extends the super class **Man**.

Both *Man* and *Woman* super classes implement the **IPerson** interface, which contains the methods *amlAlive* and *amlAdult*.

2.1.1 Characterization and genetic inheritance

In both *Man* and *Woman* super classes are defined common fields, such as *Sex*, *Attribute*, *Birth_year*, *Death_year*, *partner* and *maxChildren*.

The first difference is the field *Court_years* that is exclusive of the *Female* class and it is randomly chosen: for *Coy* type it goes from a minimum of 2 to a maximum of 8, while for *Fast* type the range is from 0 to 2. The aim of this field is to establish how many years two people must be engaged before copulating.

Characterization of people is achieved with four “gene” fields: *CoyGene* and *FastGene* for Females and *FaithGene* and *PhilGene* for Men. These fields are all initialized with the basic value of 50, and are supposed to increase or decrease as the population grows; every time a new person is born, its “genes” values are increased or decreased according to the payoffs resultant from the FPCS table: for example, the first girl born from a Fast Woman and a Faithful Man is going to have $CoyGene = 50 + \left(a - \frac{b}{2}\right)$ and

$$FastGene = 50 - \left(a - \frac{b}{2}\right).$$

Since a person is able to give birth to multiple children in its life and the costs of giving birth and parenting have an effect, also the “gene” values of the parents are modified with the same logic.

2.2 Division of tasks

Based of the different characteristics that men and females are given in The Selfish Gene, we have decided to divide the two different steps of a relationship between *Man* and *Woman* classes: men take care of the dating process, while women decide when and whether to give birth.

2.2.1 Dating

The **Dating** function is defined in the Man class and implemented differently in the two sub classes:

- Faithfull: a random woman is picked from the *women* list and, with a probability of 60% is determined if she's willing to start a relationship; if so, they became partners of each other and the *start_rel* field of the man is set to the current year.
- Philanderer: a random woman is picked from the women list and the function proceeds only if the man can still have babies and if her attribute is "Fast"; if she's not engaged there's 50% possibility that they're going to copulate and give birth, while if she has a partner there's 30% possibility that she's going to leave her partner for the new Philanderer man.

2.2.2 Labour

The **Labour** function is defined in the Woman class, and it takes as argument the parents. Firstly, it checks if both man's and woman's *maxChildren* field is greater than zero, and if so, they get decreased by one.

With a system of weights [...]

2.3 Multi-Thread Handling

The final project is based on the use of three threads, mainly because of the higher performance necessities.

The first prototype was running through the simulation with a while loop, containing all the processes and the methods needed to make it work. However, it was really expensive in terms of calculus speed: the program was really slow on finishing a cycle because it couldn't run processes in parallel, and moreover it was not realistic at all since it worked sequentially, creating the illusion of a "straight" life.

Thinking about numbers, the size of every array grows exponentially as time goes by. It means that, at the beginning of the simulation there were a bit of people to "analyse" and

to “let live”, but with big sizes, using the while loop version, every method must wait for the previous one to finish, sequentially going through big arrays element by element.

With the implementation of the threads, the arrays are still containing big amounts of people but, with the possibility of iterating through them at the same time and using different implementations, the running time of each cycle is significantly cut down.

Therefore, the program relies on three threads: one controls the *men* `ArrayList<Man>`, one controls the *women* `ArrayList<Women>` and one controls the generic `children_list ArrayList<Object>`.

This latter thread has been implemented when we realized that to have a smoother execution we needed to divide the persons who are still not 18 from the ones that can actually interact with others, and the main reason is that the children array is the one with the greatest growth with respect to the others.

Of course, so to implement this approach, we needed a way to make the threads share the same resources and work on the same values. We decided to opt for the use of Boolean values to synchronize the threads’ executions. Once the programme is launched, every thread gets the lock over the Population class, which contains arrays and values that are necessary for their iterations.

3. The Simulation

As soon as a new Population object is created, the user is prompted to enter the values of the three evolutionary payoffs ***a*** (the evolutionary benefit for having a baby), ***b*** (the cost of parenting a child) and ***c*** (the cost of courtship). Right after the function

TheBigBang is called, and it gives birth to a first group of people that, as soon as they turn 18, are going to interact to create subsequent generations.

Once the initial population is created, the simulation is launched with *StartSimulation* function.

```

public void TheBigBang(){
    for (int i = 0; i < 50; i++){
        Faithful faith = new Faithful(Current_year);
        Fast fast = new Fast(Current_year);
        Philanderer phil = new Philanderer(Current_year);
        Coy coy = new Coy(Current_year);

        men.add(faith); counter_F++; counter_M++;
        men.add(phil); counter_P++; counter_M++;
        women.add(fast); counter_S++; counter_W++;
        women.add(coy); counter_C++; counter_W++;
        counter_tot = counter_C + counter_F + counter_S + counter_P;
    }
}

```

The Big Bang

3.1 StartSimulation

In the *StartSimulation* function the three threads are initialized, giving them as arguments the respective list and the population object that is used as lock for synchronization.

It is also created an object of type **Stability** that , taking as arguments the three parameters *a*, *b*, *c*, is going to check at every iteration if the evolutionary stability is reached. If the stability isn't reached, the function **WakeThreads** is called; otherwise, the Boolean value *Population.end* is set to true and the program is stopped.

3.1.1 MenThread

In the MenThread class, the *run()* method iterates on every element of the *Men* list, and, after checking whether the element is alive and over 18, if he is not engaged, its *Dating()* function is called. If at the end of the function the man has a partner, he's added in a supportive *dating* ArrayList<>.

Once the entire list is iterated, the thread is locked to the Population object and its *done* value is set to *true*.

3.1.2 WomenThread

In the WomenThread class, the *run()* method iterates on every element of the *Women* list and, after checking whether the element is alive and over 18, it checks if she's ready to copulate. So, if a partner is set and if the courtship period has been long enough, her

function *Labour()* is called. Once the entire list is iterated, the thread is locked to the Population object and its *done* value is set to *true*.

3.1.3 ChildrenThread

In the ChildrenThread class, the *run()* method iterates on every element of the *children_list* list and, if the child has reached 18 years, is added to the right list depending on its gender.

At the end its *done* value is set to *true*.

3.2 WakeThreads

The *wakeThread()* function starts only if all the threads have their *done* field set to *true*. This is because of the necessity to update every list with new people and to do so the threads need to be sleeping.

If the initial condition is satisfied, the *Current_year* value is increased by one.

In the function we delete all people in the supportive lists *graveyard*, we add every over-18 child to *men* and *women* lists and remove from the list *men* all the element who are dating, adding the ones who have been left from their partner.

Finally, all the threads *done* values are set to *false* and the method ***notifyAll()*** is called, so that another year can pass.

4. The Graphics

We desired that our project had not only a functional usage, but also that was good looking and data easy to analyse, indeed we opted for a graphic interface which starts running at the same time of the main program.

4.1 The Page

The graphics of the program is realized with JavaFx

In the displayed page it is possible to visualize the population growing and distribution thanks to a pie chart and the two line charts, however at the same time it is possible to know exactly the data needed reading the real-time logs.

The graphics section is run in parallel with the actual program, in particular a specific thread is responsible for it.

The layout is created with a system of v-boxes and h-boxes, objects belonging to the JavaFx library which divide equally the space between the objects inserted in.

There is a total number of 4 objects:

1. Population Growth Line Chart
2. Population Members Growth Line Chart
3. Population Members Subdivision Pie Chart
4. Information Logs

4.2 Implementation

The line charts are objects of the class `LineChart<String, Number>` which uses the cartesian plan made up of 2 additional objects: `CategoryAxis` and `NumberAxis`.

The “growing lines” are instead objects of the class `XYChart.Series<String, Number>` and they represent the current number of alive people and deaths in a specific year in the first graph, and the number of people alive belonging to a specific type in the second.

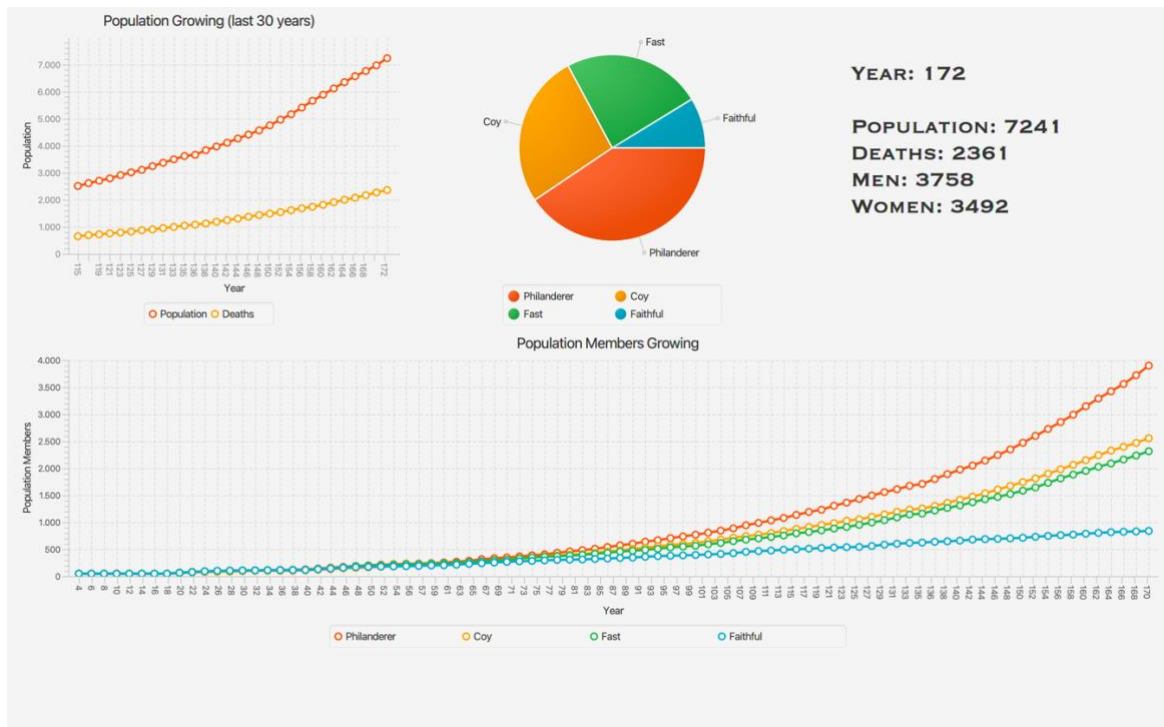
The pie chart is an object of the class `PieChart`, and its data, the four types of people, belong to the class `Piechart.Data`.

In the logs is possible to read more precisely the information you need, it is explicated the current number of alive people, deaths, men and women in the current year. Also they are an object, in particular of the class `Label` which gives the possibility to write some text inside and visualize it on the screen.

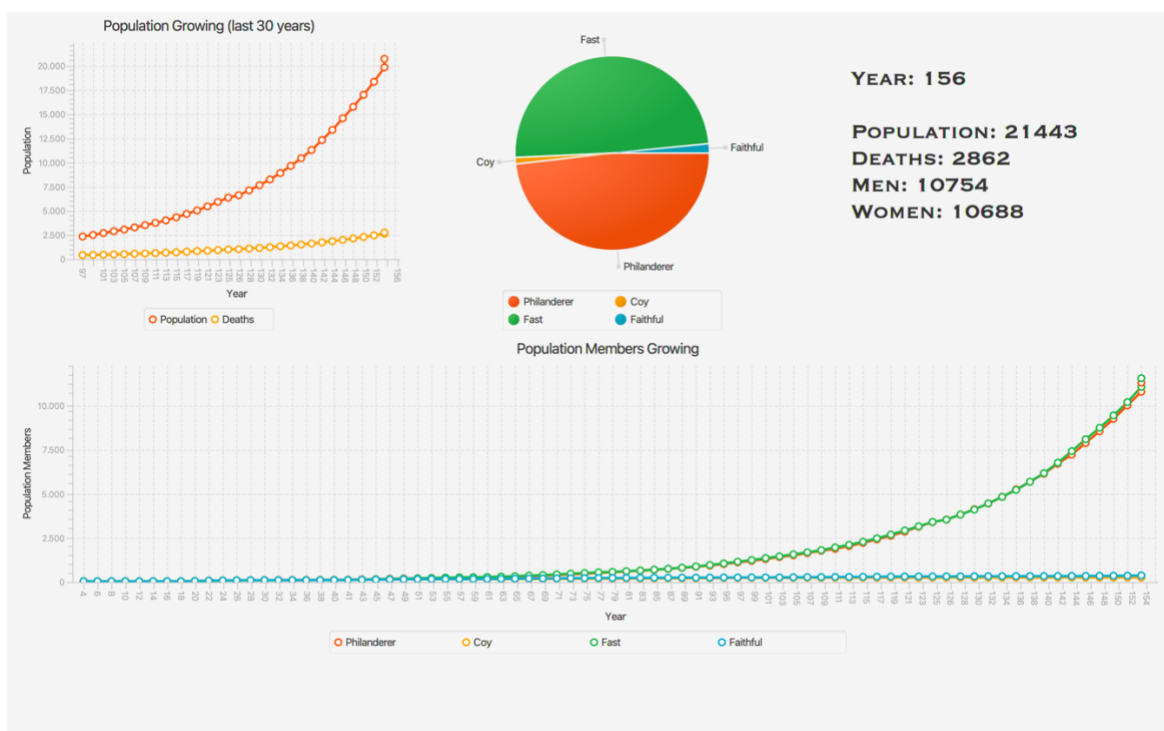
The display is updated every second thanks to the `scheduledExecutorService`, able to run a block of code at a constant rate.

Results

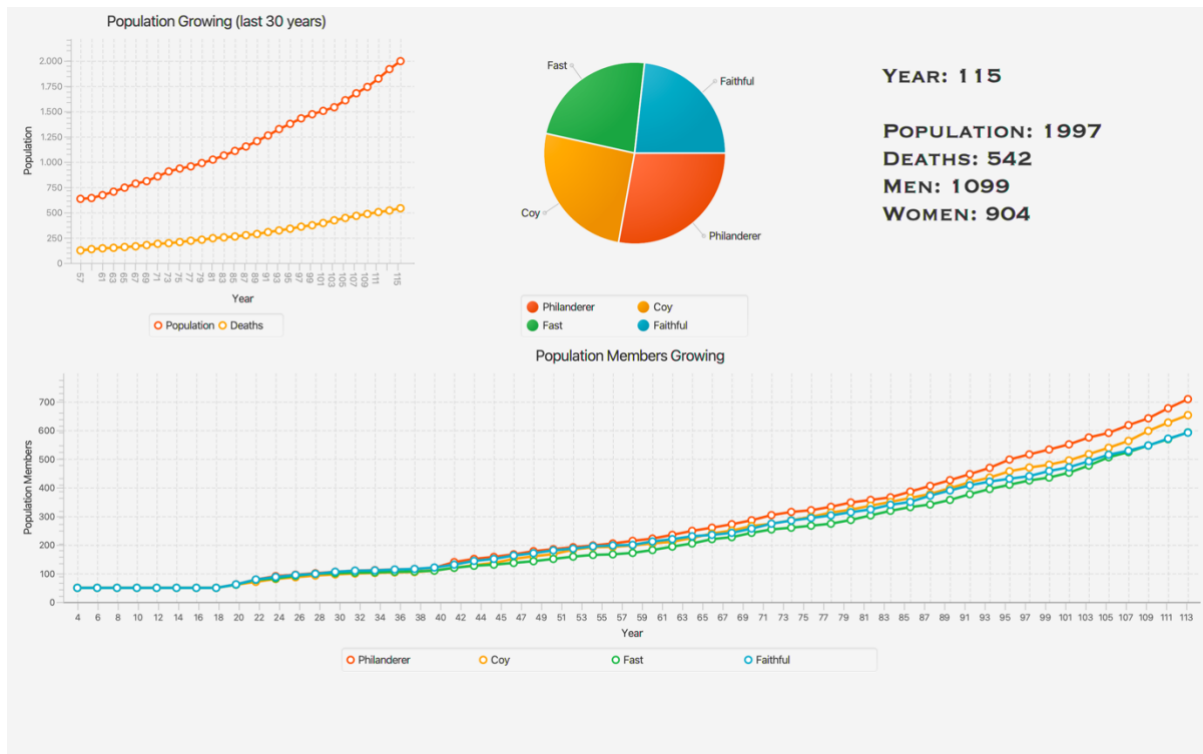
In order to elaborate the results of the project we firstly run the simulation using the parameters established in “The Selfish Gene” and then trying to change them checking the differences.



Run with the original parameters ($a=15$, $b=20$, $c=3$)



Run with custom parameters ($a=30$, $b=5$, $c=1$)



Run with custom parameters ($a=10, b=5, c=30$)

Conclusion

Observing the graphs of the results it is immediately possible to see how the change of parameters affect them, differently by the first simulation (the one with the original parameters) in the second one, in which the parameter “a” is much higher than the others, so the generated society has a high benefit of having a baby despite of the little cost of parenting and courtship, the society promotes Fast and Philanderer type.

On the other hand, in the third simulation rising the cost of courtship the 4 lines become more “compact”, then the distance of the y coordinates of the 4 member becomes shorter.

Project “The Battle of Sexes” by:

- Massaroni Flavio
- Salinetti Andrea
- Salvadori Gabriele
- Scappatura Leonardo
- Ugolini Lorenzo

