Java Project: The Battle of the Sexes

Gianluca Monte
Francesco Di Monaco
Cecilia Iacometta

Spring 2022

1 Introduction

This project is based on the study of population and their relationships. Reminding that, "a population is a set of individuals; individuals are grouped into types."
What we know at starting point is that there exists population evolution according to "rules which determine how the individuals of each type thrive (grow in their number) or langiush (decrease) depending on the global state."
According to Chapter 9 of The Selfish Gene, a 1976 book by Richard Dawkins we are going to simulate its "the battle of sexes" in which we do have 4 types of people within a population:

- **F-faithfull men**: "they are willing to engage in a long courtship and participate in rearing their children;"
- **P-philanderers**: "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;"
- **C-coy women**: "they accept a partner only after a long courtship;"
- **S-fast women**: "if they feel so, they don't mind copulating with whoever they like, even if just met."

Moreover, we have to consider the "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"

But what is the battle of sexes? Here it is:

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

Figure 1: The battle of sexes

2 Problem analysis

As stated above in the Introduction the problem consists of the simulation of a population that is expected to end up with an evolutionary stable solution.
There are two ways in which we can define define what a stable solution is:

- We have an evolutionary stable solution when the ratios (faithfulMen/totalMen, coyWomen/totalWomen) start to differ less and less between iterations, until they almost completely stabilize.
- If we take a closer look to the paragraph (1), we learn that a population finds its equilibrium when:
  - Average gain of a coyWoman == Average gain of a fastWoman;
  - Average gain of a faithfulMan == Average gain of a philanderer;
  - The average gains are stable, i.e. they remain equal as time progresses.

The second definition, as we will explain, is key to our algorithm. Our entire project is based on what we call "convenience". At the beginning of a person's life, they will adopt the strategy that is more convenient to them at that moment. A man can grow to be a faithfulMan or a Philanderer, and a Woman can grow to be a coyWoman or a fastWoman. The convenience factor is based on comparing the average gain of each strategy, i.e. if the fastWoman's gain is greater than the coyWoman's, the females that are born in that time will grow to be coyWomen.

3 Program Architecture

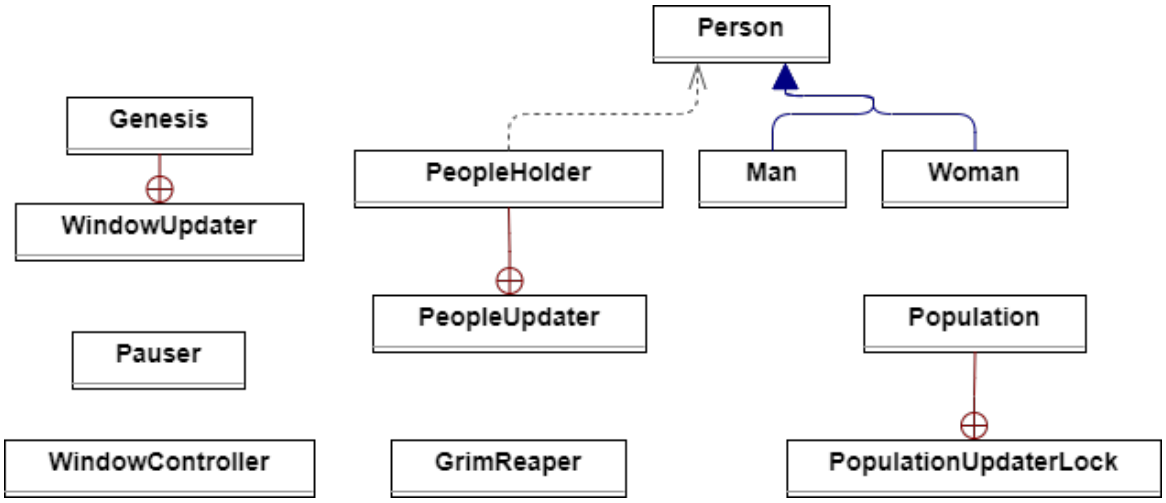

Figure 2: Program Architecture

When a new population is created via the User Interface, the **Population constructor** is invoked with the parameters entered by the user.
The Population constructor initializes the population with the specified number of people, adding them to the two (*one for each gender*) **PeopleHolder** objects. PeopleHolder objects are designed to **contain all the needed lists and data structures for a single gender** (ex. dead, alive).

```
public class PeopleHolder <S extends Person> {
    private final Population.PopulationUpdaterLock updaterPool;
    final ConcurrentLinkedQueue<Integer> dead = new ConcurrentLinkedQueue<>();
    final XYChart.Series<Number,Number> series = new XYChart.Series<>();
    final ArrayList<S> alive = new ArrayList<>();
    private boolean started = false;
    public ThreadLocalRandom tlr;
```

Figure 3: PeopleHolder

PeopleHolder also starts a **"PeopleUpdater" Thread** to update the people in its "*alive*" list.

```
class PeopleUpdater extends Thread {
    PeopleUpdater () {
        super( name: "PeopleUpdater");
        tlr = ThreadLocalRandom.current();
    }

    public void run() {
        try {
            while (updaterPool.running) {
                for (int i = 0; i < alive.size(); i++) {
                    alive.get(i).update(i);
                }
                synchronized (updaterPool) {
                    updaterPool.finished++;
                    updaterPool.wait();
                }
            }
```

Figure 4: PeopleUpdater Thread

So, in the end, two PeopleUpdater are instantiated (<Man>, <Woman>), and obviously those **need to be synchronized** between each other, meaning that you want them to *start at the same time and restart only when the other has finished too.*
The Population Constructor also starts a **PopulationUpdaterLock Thread**, that serves the purpose of *synchronizing* the two between each iteration (1 year).

```
class PopulationUpdaterLock extends Thread {
    volatile int iterationDelay = 0;
    volatile boolean paused = false;
    volatile boolean running = true;
    volatile transient int finished = 0;
    private final ThreadLocalRandom tlr = ThreadLocalRandom.current();

    PopulationUpdaterLock () { super( name: "PopulationUpdaterLock"); }
```

Figure 5: PopulationUpdaterLock Thread

PopulationUpdaterLock acts like a "**synchronizer**" for the two *PeopleUpdater* threads, meaning that it starts the next iteration only if **both** are finished. It also acts as a "*Control Room*", meaning that it can **pause or stop** the whole population's execution.
The *PeopleUpdater* Thread updates every person in its "*alive*" list.
The **Person** object is designed in such a way as to **act differently depending on its current status**.
The Person class itself is abstract, and it contains the shared code between class **Man** and **Woman**.
Many methods are implemented by the Man and Woman classes, for example the **update()** method, on which the whole program depends.

```
public class Man extends Person {
    private static final BiConsumer<Man, Integer> single = Man::single;
    private static final BiConsumer<Man, Integer> old = Person::deathChance;
    private static final BiConsumer<Man, Integer> dead = (man, i) -> {};
    private static final BiConsumer<Man, Integer> young = Person::tooYoung;
    private BiConsumer<Man, Integer> statusFunc = young;

    void update(int i) {
        statusFunc.accept( t: this, i);
    }
```

Figure 6: Person

Man and Woman act as if they had a "state", meaning that the update Method changes dynamically based on what the person's current situation is (young, single...). Every method then checks if it's the case to change state to the next one, i.e. if the person is young enough, it should change state from young to single.

The lifespan of a Person is limited, and we found a way to make it as real as possible. To do this, we created a weighted random algorithm based on ISTAT data to compute the death age of a person.

```
public class GrimReaper {
    public static int[] deathArray = {0,0,0,0,0,5,10,15,15,20,20,25,25,25,
    public static Random seedOfDeath = new Random();

    public static int deathAge () {
        return GrimReaper.deathArray
                [seedOfDeath.nextInt(GrimReaper.deathArray.length)]
                + seedOfDeath.nextInt( bound: 5);
    }
```

Figure 7: GrimReaper

Finally, when a Population reaches equilibrium, the execution is stopped by the Stopper class.

```
boolean isStable(LinkedList<Float> list) {
    float sum = 0;
    float min = list.getFirst();
    float max = list.getFirst();
    for (Float f: list) {
        if (f > max) max = f;
        else if (f < min) min = f;
        sum += f;
    }
    return findMaxDeviation(max, min, average: sum / list.size()) < threshold;
}
```

Figure 8: isStable

4 Algorithm Development

4.1 Strategies Convenience

We based our entire project off of 2 key concepts:

- **Strategies**: An entity can adopt multiple different game strategies;
- **Convenience**: An entity can change its strategy based on what's more convenient to them.

In Population class are two boolean fields representing what strategy is best for the players, that in our case are the two sexes.

```
boolean manConvenience = false;
boolean womanConvenience = false;
```

Figure 9: booleans

For instance, if manConvenience equals to True, the next Man instance will be of type Philanderer, otherwise it will be of type FaithfulMan.
Down below is the method we create for updating periodically the two convenience fields. It's called every time possible by the PopulationUpdaterLock Thread.

```
private void updateBirthValues () {
    womanConvenience =
            var1 * faithfulMen.get() < var2 * faithfulMen.get() + var3 * philanderers.get();
    manConvenience =
            var1 * coyWomen.get() + var2 * fastWomen.get() < a * fastWomen.get();
}
```

Figure 10: update Birth Values

The vars values are for optimization purposes, so that we don't evaluate the formulas each time.

```
var1 = a - (b/2) - c;
var2 = a - (b/2);
var3 = a - b;
```

Figure 11: vars values

4.2 Abstract Person class & Statuses

As we described before, Person is an abstract class that conceptually contains the core functions of a living person, and in this case it holds the shared code between Man and Woman.
More specifically, Person class contains all the aging related functions and statuses, including death.
Every person has a status property, which will change during the course of their lifespan and will influence their actions.

```
private BiConsumer<Man, Integer> statusFunc = young;
```

Figure 12: BiConsumer

A person can retain only one status at a time, choosing from:

- young;
- single;
- old;
- dead.

In our project, Statuses are both conceptual statuses and functions, we implemented them using the functional interface BiConsumer:

```
BiConsumer<Man, Integer> single = Man::single;
BiConsumer<Man, Integer> old = Person::deathChance;
BiConsumer<Man, Integer> dead = Person::nothing;
BiConsumer<Man, Integer> young = Person::tooYoung;
```

Figure 13: BiConsumers

BiConsumer is an interface that will let you execute whatever function is held within it by using the accept() method.
To get the full picture, down below is the update() function, which calls the method accept() and executes the function that corresponds to the current status of the person.

```
void update(int i) {
    statusFunc.accept( t: this, i);
}
```

Figure 14: update

4.3 Iteration Synchronization & Updater

One of our project's core concepts is that one iteration equals one year, and both men and women will need to have finished updating before the next year starts.
The updating of the two lists is done by the PeopleUpdater threads, which iterates through the lists of alive people and calls their update function.
As stated before, PeopleUpdaterLock is a Thread meant to synchronize the two PeopleUpdaters, making sure that the next iteration starts only when both have finished their task.
This is done by a wait()-notify() mechanism:

```
class PopulationUpdaterLock extends Thread {
    volatile boolean paused = false;
    volatile boolean running = true;
    volatile transient int finished = 0;

    public void run() {
        while (running) {
            updateBirthValues();
            if (finished == 2) {
                finished = 0;
                // if (paused) {synchronized (pauseLock) { pauseLock.wait();} }
                synchronized (this) {
                    notifyAll();
                }
            }
        }
    }
```

Figure 15: wait()-notify() mechanism

The finished counter refers to the number of PeopleUpdater that have completed the iteration. In this case, if the counter is equal to 2, it means that both have finished, and the program can resume iteration.

```
class PeopleUpdater extends Thread {
    public void run() {
        try {
            while (updaterLock.running) {
                for (int i = 0; i < alive.size(); i++)
                    alive.get(i).update(i);
                }
                synchronized (updaterLock) {
                    updaterLock.finished++;
                    updaterLock.wait();
                }
            }
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Figure 16: PeopleUpdater

Here you can see that the PeopleUpdater Thread does its iteration and then immediately calls wait(), after incrementing the finished counter.

4.4 Grim Reaper

We all gotta go at some point... but when? To avoid choosing that arbitrarily, we referenced the ISTAT data. Using a Python script, we created an array of numbers, representing death ages. The more occurrences of that number, the more likely it is to pass away at that age. The array is hard-coded into the class GrimReaper, and the death age is computed by getting a random index.

```
public class GrimReaper {
    public static int[] deathArray = {0,0,0,0,0,5,10,15,15,20,20,25,25,25,
    public static Random seedOfDeath = new Random();

    public static int deathAge () {
        return GrimReaper.deathArray
                [seedOfDeath.nextInt(GrimReaper.deathArray.length)]
                + seedOfDeath.nextInt( bound: 5);
    }
```

Figure 17: Grim Reaper detail

4.5 Couple Making

To match men and women, we choose an interval of the smallest of the two lists (to avoid going out of bounds), and then match all the people in that list with their respective index from the other list. This is particularly useful because the size of said subset can be changed by the user even during execution, to limit the number of newborns and reduce memory usage.

```
private void randomInterval() {
    int sizeMan = menHolder.alive.size();
    int sizeWoman = womenHolder.alive.size();
    int lower = Math.min(sizeWoman, sizeMan);
    int span = (int) ((lower/2) * copulatingRatio);
    max = tlr.nextInt(span, bound: lower/2);
    min = max - span;
}
```

Figure 18: randomInterval

4.6 Equivalent exchange

To make sure we make the least possible operations on the lists, we found it useful to "exchange" the dead with the newborns.
When a person dies, their index is added to a queue.
When a new person is born, we have two possible cases:

- **There is a dead person in the queue**: we just replace the dead with the newborn.
- **The queue is empty**: we append the newborn to the end of the list (which doesn't cause a ConcurrentModificationException because we don't use a foreach).

```
void newSoul (S soul) {
    Integer passedSoul = dead.poll();
    if (passedSoul == null) {
        alive.add(soul);
    } else {
        alive.set(passedSoul, soul);
    }
}
```

Figure 19: newSoul

5 Conclusion

To display the data we obtain and the information regarding the execution, we designed a GUI that allows the user to create new populations with custom data, pause them, cycle through the created populations, and destroy them. The user is given various inputs to set many variables of the execution on the fly, such as refresh delay, iteration delay, changing a,b,c, setting the size of the "reproducing" people interval (refer to the Couple Making section for more information). The GUI also displays data such as percentages, iteration time, processed people per second, and memory usage.

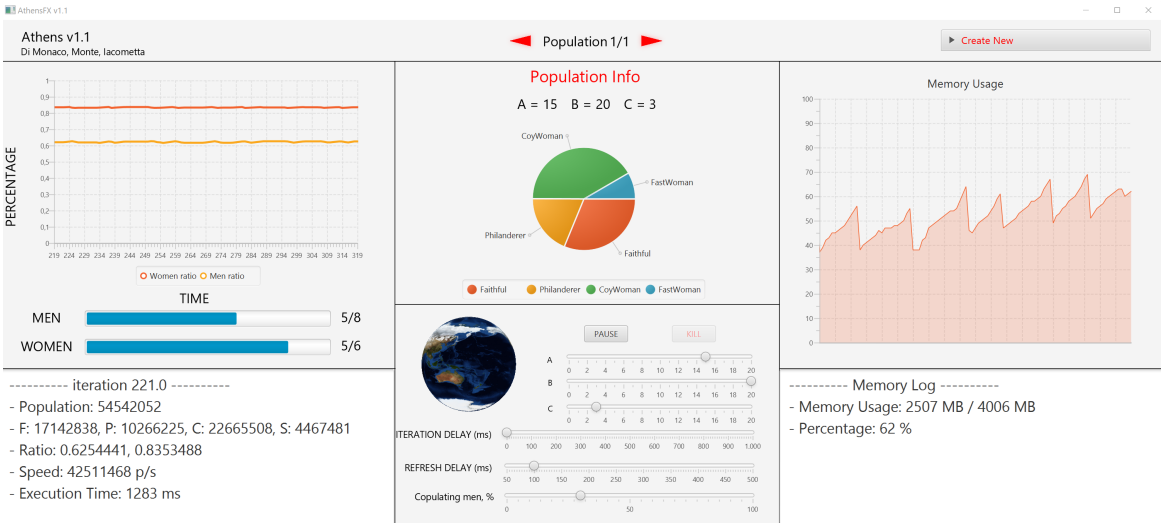

Figure 20: Graphic interface