Devon Harker
Josh Haskins
Vincent Tennant

Table of Contents

Plan

Our initial plan was to just have two people live in one house and have random events affect them. Next was that we wanted the ability for these people to have kids, these kids have kids and so on. We then decided to add in another family so that the simulation would be more complex and use more threading. You can sort of think of our simulation like a combination of the The Sims (customization), The Game of Life (events), and with threading.

Synchronization

Semaphore

We used the Java implementation of a semaphore from the Java.util.concurrent library to ensure that data for each house (House net worth etc.) was to be accessed mutually exclusive. The Java implementation of the semaphore is a counting semaphore. The semaphore is constructed with *n* number of tokens to be used. In our case we created two semaphores, one for each household (if two families are simulated). Both of the semaphores were created with only one token available. This ensured that whenever a person would go to grab a token from the semaphore they would either gain access to it and they'd enter their critical section, or they would be added to the list inside the semaphore to wait for a token to be freed. Once the token is released a person is released from the list and allowed to acquire the token and enter their critical section. This was a simple solution to ensure that all data in the house class is accessed by all the people in a mutually exclusive manner.

Cyclic Barrier

One issue we face was making sure that all people remained on the same cycle. to solve this problem we used the java implementation of a CyclicBarrier found in the java.util.concurrent library again. Unlike an ordinary barrier the CyclicBarrier works by setting *n* number of threads to wait for before repeating the cycle. When a thread calls the .await() function the barrier will halt it's execution until all *n* threads have declared that they're waiting. Once all of the threads using the barrier have called the .await() method, the barrier either lets all the threads pass seamlessly or executes a block of code inside the barrier itself while simultaneously letting the threads continue. This allowed us to implement each cycle with ease while at the same time letting us execute instruction in between cycles.

Important Traits

Events

Events are a major part of the simulation. They happen once per cycle and are randomly chosen. Some events have a higher chance of being picked over others. For example, Pneumonia has a 6.25% chance, while Apocalypse has less than a 1% chance of being picked. Each event also has a percent of people to be affected. Apocalypse has 100% and Pneumonia has 20%. Each event also causes damage to lifespan and money.

Houses

Houses are where people live. They also store all the money associated with that person and all people who live in that house. There are many different types of houses, ranging from a tent to a castle. Different types of houses cost different amounts of money, also they affect Feelings. Feelings will be explained below.

Jobs

Each person over 20 has a job. Once a child turns 20, their job is changed from null to a randomly generated job. There are many types of jobs. All the income values are based on actually averages of Vancouver based jobs. Jobs also affect the feelings of people.

Feelings

The feelings of a person are primarily dependent on five factors. The first of these factors is the type of job a person holds. In general, higher paying jobs are more stressful and as result they decrease happiness. The second factor in determining happiness is the type of house the person lives in. The more expensive the house, the happier the person living there is. The third factor is the current age of a person. Children are happier than adults, which are happier than senior citizens. The fourth factor is the status of close relatives. If a person's children or parents have died, the person is less happy. The fifth and final factor in determining happiness is the ratio of income to expenses. This can be broken down into two parts; the income of an individual compared to their expenses, and the amount of money contained in the family's bank balance.

Possibilities

Some things we would like to add to our program are; The ability to choose where the location of the house is (ie Canada, US, China, etc.). These different locations would cause the Events to affect the houses and people in different degrees. Also adding the ability to have more than two families to the simulation would be nice, theoretically right now we could implement this, but the GUI would not display the houses after the first two.

Problems

GUI Refresh Rate

We initially had an issue with the GUI being redraw so fast that it kept on freezing, so we limited the redraw rate. This solved our problem.

Rebuilding the Cyclicbarrier

Cyclic barriers, a class in Java's concurrent library, are created with a fixed size. We wanted our barrier size to be equal to the number of currently living people, meaning that the barrier had to be resized every time the number of people alive was changed. Recreating the barrier while also preventing people from using the old barrier was somewhat difficult, though we are happy with the final implementation.

Effects in a Balanced Manner

The values for random events, jobs, etc. are quite difficult to balance and unbalanced values had noticeable effects on the results of our simulations. An example of these effects was when our simulations were ending prematurely as a result of every one dieing from random events. Balancing these values in such a way that they were both noticeable but not overpowering is a task that may be impossible to accomplish, though we have made our best efforts to achieve balance..

Hang ups on massive influxes of new threads

Throughout our testing, we noticed a trend where our program would often (but not consistently) freeze when there were large influxes of new people. The reason this cause of this hangup was never accurately determined, though we suspect it has to do with recreating the cyclic barrier. At this point, we attempt to mask the problem by making each thread sleep after passing the barrier as this seemed to decrease the frequency of hangups; the longer the delay, the less frequent the hangups.

Eclipse Crashing

When we were about halfway through coding the project using Eclipse to manage our files, then being synced with Dropbox. We were also using Eclipse on multiple OS's, Ubuntu, Mac, and Windows, so this could have contributed to the error. What happened was that Eclipse would open with half of the files missing and if they were re added, and Eclipse was relaunched, they would once again be missing. So we moved to NetBeans, also using a Project folder structure, and half not had any issues. One of the bonuses was that NetBeans has a Thread view far superior to that of Eclipse.

Mac JComboBox

We encountered this strange error when clicking multiple times in a JComboBox, but while only on a Mac. On Windows and Ubuntu there is no problem. Unfortunately this problem is not always reproducible. The program does run completely fine after the error has been thrown. The error is:

```
Java.awt.IllegalComponentStateException: component must be showing on the screen to
determine its location
       at Java.awt.Component.getLocationOnScreen_NoTreeLock(Component.Java:2044)
       at Java.awt.Component.getLocationOnScreen(Component.Java:2018)
       at sun.lwawt.macosx.CAccessibility$22.call(CAccessibility.Java:390)
       at sun.lwawt.macosx.CAccessibility$22.call(CAccessibility.Java:388)
       at sun.lwawt.macosx.LWCToolkit$CallableWrapper.run(LWCToolkit.Java:527)
       at Java.awt.event.InvocationEvent.dispatch(InvocationEvent.Java:241)
       at sun.lwawt.macosx.LWCToolkit$CPeerEvent.dispatch(LWCToolkit.Java:684)
       at Java.awt.EventQueue.dispatchEventImpl(EventQueue.Java:727)
       at Java.awt.EventQueue.access$200(EventQueue.Java:103)
       at Java.awt.EventQueue$3.run(EventQueue.Java:688)
       at Java.awt.EventQueue$3.run(EventQueue.Java:686)
       at Java.security.AccessController.doPrivileged(Native Method)
       at
Java.security.ProtectionDomain$1.doIntersectionPrivilege(ProtectionDomain.Java:76)
       at Java.awt.EventQueue.dispatchEvent(EventQueue.Java:697)
       at
Java.awt.EventDispatchThread.pumpOneEventForFilters(EventDispatchThread.Java:242)
       at Java.awt.EventDispatchThread.pumpEventsForFilter(EventDispatchThread.Java:161)
       at
Java.awt.EventDispatchThread.pumpEventsForHierarchy(EventDispatchThread.Java:150)
       at Java.awt.EventDispatchThread.pumpEvents(EventDispatchThread.Java:146)
       at Java.awt.EventDispatchThread.pumpEvents(EventDispatchThread.Java:138)
       at Java.awt.EventDispatchThread.run(EventDispatchThread.Java:91)
```

We have some some research into the issue. On the NetBeans website is has been submitted as an error many times, while the developers have said that it has been fixed. See https://netbeans.org/bugzilla/show_bug.cgi?id=119409 and https://netbeans.org/bugzilla/show_bug.cgi?id=208961. While on the Sun Microsystems website is has been assigned this status "it is not feasible to fix it without breaking existing applications, closed as will not fix", see http://bugs.sun.com/view_bug.do?bug_id=6877806 for more details on the error. Since it does not actually break the program and is only reproducible randomly on the Mac and that Sun is unable to fix the issue, we have no way of properly fixing the issue.

Learning Outcomes

Thread Synchronization

We learned practical ways to achieve concurrency with the use of classes in Java's concurrent library, such as semaphores and cyclic barriers. This was done by using the concurrent library and figuring out how exactly it worked first hand.

GUI Builders

We learned how to use simple GUI builders for Java applications, including the fact that they really aren't that hard to use.

Double Check Everything; Even Basic Logic

In an early build of our project, we had a bug where random events would affect an inverted number of people; for example, the 'polio' event would affect (and kill) 95% of a population instead of the 5% that was intended. The cause of this was using the wrong comparator in an if statement that checked to see if a person should be affected by an event. We learned that even 'simple' logic in a program should be double checked because it can have perverse effects on the rest of the program.

Planning Ahead is Very Important

Even though we began to plan and work on the project immediately after it was assigned, we had barely enough time to finish the project. If we had not planned ahead when we did, it is likely that our project would have not been finished.

Initial Plans Are Not Always Followed Through

Our initial plans had many aspects that were not included in the final project, such as a 'location' for the simulation to take place. We were having difficulties deciding on what exactly these aspects would be used for and, since we could not come to decision, decided to not include them. From this, we learned that it is not worthwhile to include every single element that is contained in the initial draft of a project as it is a poor use of resources.

Using the Program

At the command line, in the folder with the .jar of our project, use the command: 'java -jar lifeSimulator.jar'. This will execute the main class of the project, allowing the GUI to be used for further user interaction.
The program creates a 'lifesim.log' that contains information about finished simulations. This file is output to the same directory as the jar file.

NOTE: Running the project inside of an IDE, such as NetBeans, may result in the images not displaying correctly.

NOTE2: To run this project Java 1.7 is required. This is because of some GUI related elements only present this version.