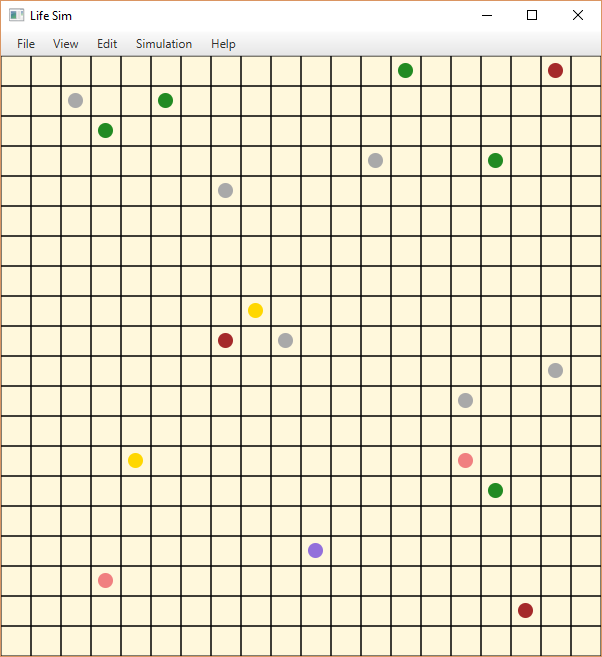
Artificial Life Simulator Report

By Mathew McLean

# Presenting the Application

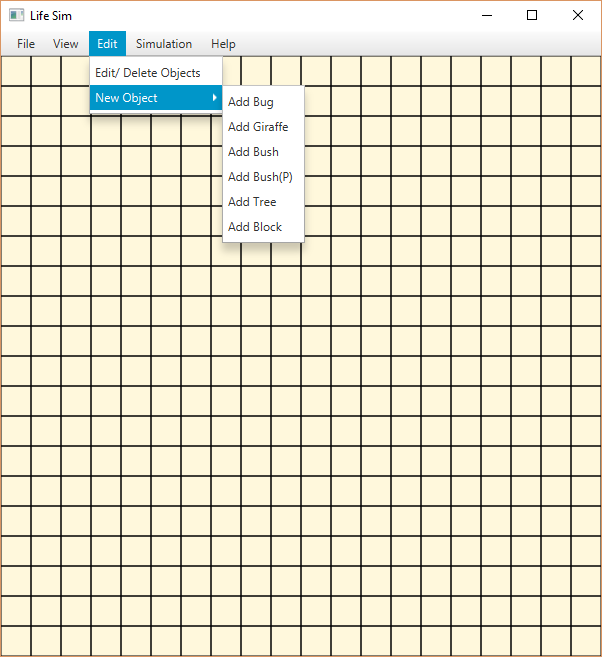
## Getting Started

The Artificial Life Simulator aims to process and visualise the behaviours of a variety of different artificial life forms on a 2D map. Whilst the behaviours of the lifeforms are pre-set, the user is able to save, load and run different configurations of animals, plants and obstacles to produce different maps and therefore different outcomes for the simulation.

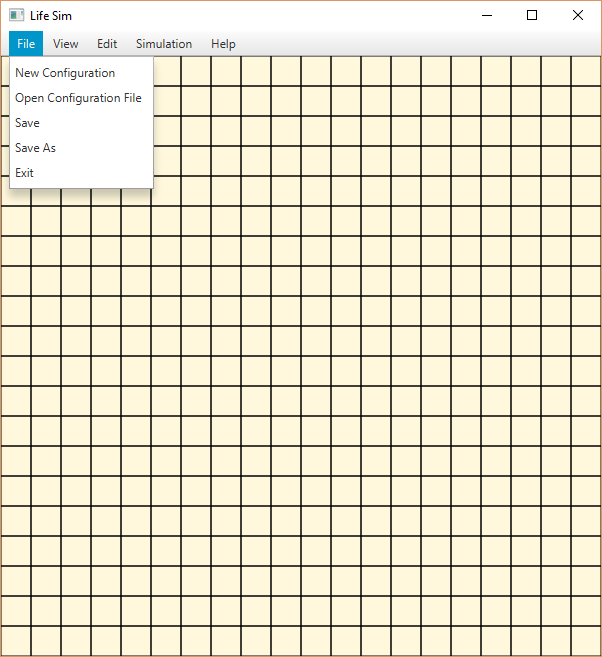
Once a configuration of lifeforms has been settled upon, the simulation can be run and the various types of plant and animal will interact with each other. For example bugs (light red) will search out and eat bushes (green) and poisoned bushes (purple). The poisoned bushes hurt the bugs over time and will eventually have their revenge and kill the bug who ate them. Each simulation is semi-random and the same simulation will sometimes play out differently.

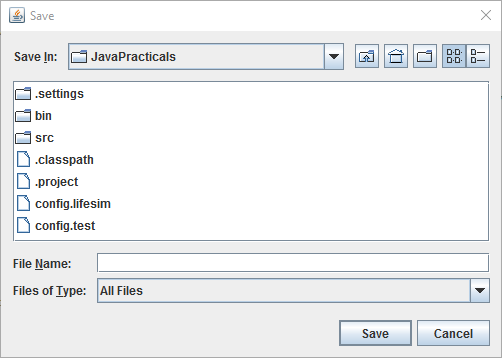
## Running a Simulation

Running a simulation is very simple. On start-up the game defaults to an extremely simple simulation that only contains a few obstacles and plants however, this sim isn’t very active so the first step is to add an animal and then run a basic sim.

To do this, first go to the Edit menu and scroll down to new object where you will see a list of the objects it is possible to add to the game. For now, select the first object ‘Add Bug’.

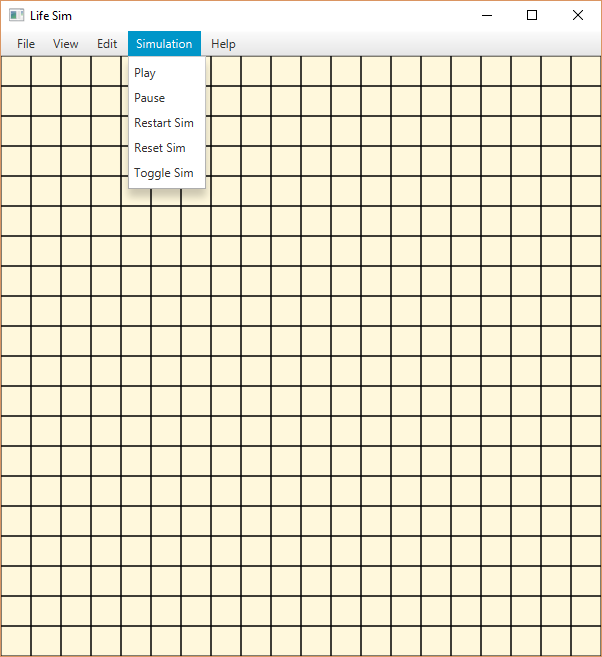
Once you have added a bug, go to the simulation menu and press play to start the game. To pause the game, go to the Simulation menu and press pause. A reference screenshot and explanations for other options in the simulation menu are available below in *Controlling the Simulation*.

Now that you have a running simulation, you might want to save your configuration so that you can run it again in the future. To do this, go to the File menu.

 In the File menu you will see several options, the relevant ones here are Save and Save As. Save will save the configuration to the current file location. In this case, since we haven’t loaded a game, the current file location is game’s default path which overwrites the config.lifesim file in the project directory. To save the file in a different location, select Save As and use the filebrowser to name and save the file in a place of your choosing.

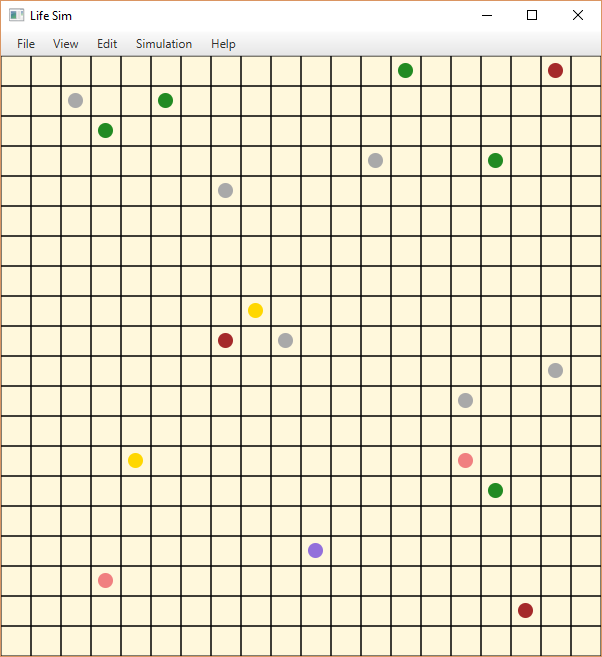
To reset your configuration, press New Configuration which will take return you to the default configuration. This new configuration can be altered and saved in just the same way as the previous one. Your old configuration can be accessed any time by selecting Open Configuration File and navigating to the location where you saved it. A saved configuration stores the starting locations and type of each object in the simulation.

## Controlling the Simulation

To control the simulation, navigate to the simulation tab on the menu. From there, you will see a list of options.

* **Play**: Starts the simulation from the current point in time.
* **Pause**: Stops the simulation from updating but does not change any of the object’s variables so that the simulation can be resumed from the current point in time by pressing play again.
* **Restart** **Sim**: Returns all objects to their starting locations and statuses and the starts the simulation again.
* **Reset Sim**: Returns all objects to their starting locations and statuses but does not start the simulation running.
* **Toggle Sim**: Toggles the display of the simulation on or off. The simulation will continue to run in the background but grid on the screen will not be updated until the display is toggled on again.

## Animal Types

There are several different object types available in the artificial life sim, each with their own behaviours:

* **Bug**: Bugs (light red) search out and eat bushes and poisoned bushes.
* **Giraffe**: Giraffes (yellow) search out and eat trees.
* **Bush**: Bushes (green) are eaten by bugs.
* **Bush(P)**: Poisoned bushes (purple) are eaten by bugs, however they inflict a poisoned status on any who eat them, which deals damage over time until the target is dead.
* **Tree**: Trees (brown) are eaten by giraffes, once eaten they will regrow and reappear on the map in the same spot after a number of turns. At which point they can be eaten again and the cycle continues.
* **Block**: Blocks (grey) provide and obstacle to animal movements.

## Additional Features

There are additional features for the game that aren’t yet implemented. These include the ability to edit the dimensions of the world, such as the number of rows and columns present on the map, the ability to delete or edit lifeforms that have been added to the simulation already and the ability to display information on the map or on the status of the lifeforms represented in the simulation.

# Object Oriented Design

## Class List

Below is a list of the classes in the program, an explanation of their function and an analysis of their design and organisation.

### GridsCanvas

Grids Canvas is the main class for the project. It contains the gameloop, the functionality for all the menus and drawing to the screen.

GridsCanvas does too many different things for a single class. Each individual class should have a single responsibility whereas GridsCanvas has several. This violates the principle of encapsulation as the implementation of unrelated parts of the program are taking place in the same class which makes the program as a whole less modular and thus the code less reusable as parts of the code that should be replaceable such as the drawing functions or menus (which both should have been split off as separate classes) require a larger refactoring instead.

Gridscanvas also contains unnecessary static variable such as width and height which are only referred to at the creation of certain objects. These static variables remain in memory unnecessarily after they’ve been used. In the current implementation of the program where they cannot be changed, these variables should also be final. However, a useful future feature for the program would be the ability to adjust the board size which would require these variable to be changeable.

### PopUp

PopUp is a class designed to handle functionality that would require an extra window to appear in addition to the main game window. Examples would be a menu for editing existing objects or for displaying information about the simulation. However this class is unfinished and currently only used to support a popup containing information about the application and the author.

PopUp was intended to be a reusable class that displayed information as an extra popup window. From a design perspective, this would have been an efficient class as multiple functionalities could be added to the program whilst all displaying information in the same way. Examples include: a colour key to identify different types of lifeform, help information on the app, information on the current configuration.

### Configuration

Configuration handles the reading and writing of java properties files for the purposes of saving and loading.

Configuration is a fairly simple class. However, a continuous instance of it is created in GridsCanvas which is wasteful, given that it only needs to exist at save and load times.

### AWorld

AWorld manages the background sim. It contains lists of all objects in the game, as well as the positioning grid that’s used to determine the relative positioning of each object. It populates and updates this grid.

AWorld is also an offender for unnecessary static variables. None out of objectList, blockList, lifeList and Grid should have been static. Whilst being static gives these variables the necessary scope to be accessed by all the classes that need access to them, it also means that there can only be one instance of AWorld active at a time without the two conflicting with each other. This limits the extensibility of the program e.g. can’t run multiple simulations at once.

AWorld also has poor information hiding and encapsulation as there are too many methods that are public, some of which duplicate functionality, such as addBush and addObject (addBush should be deleted and replaced by addObject being called multiple times). Several of these methods, such as clearGrid should be private and called from within the class, given that it is almost always called alongside drawGrid.

### AObject

AObject is an abstract class that serves as the root class for all objects in the game. It contains the essential methods and information that are common to all animals, plants and blocks. This information relates to the starting and current positions of the object and detecting collisions.

AObject is at the top of a long chain of inheritance that reduces a lot of code repetition. Use of the protected modifier allows subclasses to access the relevant methods whilst stopping unnecessary access from outside. This is effective information hiding. The existence of AObject also allows for a singular list of everything that can be placed on the grid, which is very useful for handling a large variety of objects.

### ALife – extends AObject

ALife is an abstract class that serves as a common class for all lifeforms, both plant and animal. It defines the methods and variables that are essential for all living creatures. This is to allow for a single list of living creatures with common methods that can be iterated over in the gameloop.

ALife allows differentiation between objects which are alive or inanimate which is efficient for reducing the number of superfluous calls.

### ABlock – extends AObject

ABlock defines the characteristics of a block. Most methods inherited from AObject.

### AAnimal – extends ALife

AAnimal is an abstract class that serves as a common class for all animals. It defines further methods from ALife that differentiate animals from other life. These include methods for searching for targets, determining whether a target is food and a method for processing eating other lifeforms.

### ABug – extends AAnimal

ABug defines the characteristics of a bug that differentiate it from other animals. The most important change is isFood function which defines what a bug is looking to eat (bushes).

### AGiraffe – extends AAnimal

AGiraffe defines the characteristics of a giraffe that differentiate it from other animals. AGiraffe contains two additions, it has a larger smell range than average and is looking to eat trees.

### APlant – extends ALife

APlant is an abstract class that defines the common characteristics of all plants.

### ABush – extends APlant

ABush defines the characteristics of a bush that differentiate it from other plants. Bushes are very generic plants.

Since, ABush doesn’t deviate much from APlant, the two classes should have been combined to be the superclass for all plants which could be extended by ABushP and ATree without having to alter either class.

### ABushP – extends APlant

ABushP defines the characteristics of a poisoned bush that differentiate it from other plants. Poisoned bushes differ in that the poisoned variable for them starts as true which affects animals that eat them. Additionally ABushP has a different reset method so that when restarting a configuration the bush starts as poisoned whilst other lifeforms do not.

### ATree – extends APlant

ATree defines the characteristics of a tree that differentiate it from other plants. When dead trees increase a counter which, after a period of time resurrects them and they become a viable food source again whereas other plants simply die out.

## Design analysis

The Artificial Life Simulator was developed by continuously iterating on and expanding more simplistic programs to form a more complex whole, this had a large impact on the design and structure of the final project. Below is a discussion of some of the previous iterations of the project, how those iterations evolved into the current version and the ways in which they impacted upon the project.

### Random moving bug, takes details from user

This was the first stage of the project, meaning that ABug is the oldest class in the application. Most of the code from this point of the project has since become obsolete. The random movement function is still in use for solving moments of indecision where more targeted options aren’t available.

### Pathing, searching for food

The essential logic from the 2D grid and the pathing algorithm are both still in use in the final version of the Life Simulator. Given that this was the point where I begin to really start to use legacy code several of the design flaws in the final product can be traced back to this point in the development.

AWorld was used as the main for the project at this point, which was a mistake I would repeat later by associating the main with the visual part of the program. Since AWorld was the main, it was less of an issue that the grid and list variables were static as there can only be one instance of the main.

### JOptionPane based interface

This was the point where the major refactoring of my code occurred as I had to move the main from AWorld to a different class due to the amount of tasks that AWorld was beginning to perform that were unrelated to the simulation of the game world.

### Graphical Interface

The graphical interface was just added in place of the JOptionPane interface. This finally split the grid that the simulation was running on and the graphical representation.

# Discussion and Conclusion

## Overall Design

Overall, the Artificial Life Simulator application has some serious design flaws, particularly to do with the processing of the interface and the background sim. GridsCanvas in particular. These issues were largely caused by attempts to take short-cuts using variables with larger scopes than necessary to give access to share information between several different classes. Ultimately however, this was costly as it postponed the points where I would have to reorganise my code, rather than circumventing them. This meant that invested further and further into dead ends where it was becoming very difficult to test my code due to it being heavily interdependent and the scale of the re-writes that would be required just grew.

The project suffered several set-backs due to short sighted design choices where I didn’t bear in mind the direction in which the project would have to change in order to accommodate its final size. In hindsight it was necessary to be more careful to abstract my code earlier on. I had to rewrite a lot of classes with each iteration of the program because they very interdependent and so I had to rewrite the way they interfaced with each other.

Ideally I would have separated out my main so that it was simply calling other classes to do the various tasks required by the program. This could have allowed me to do things like re-use the same interface code, both for my main window and extra ones by feeding in size variables when I created them.