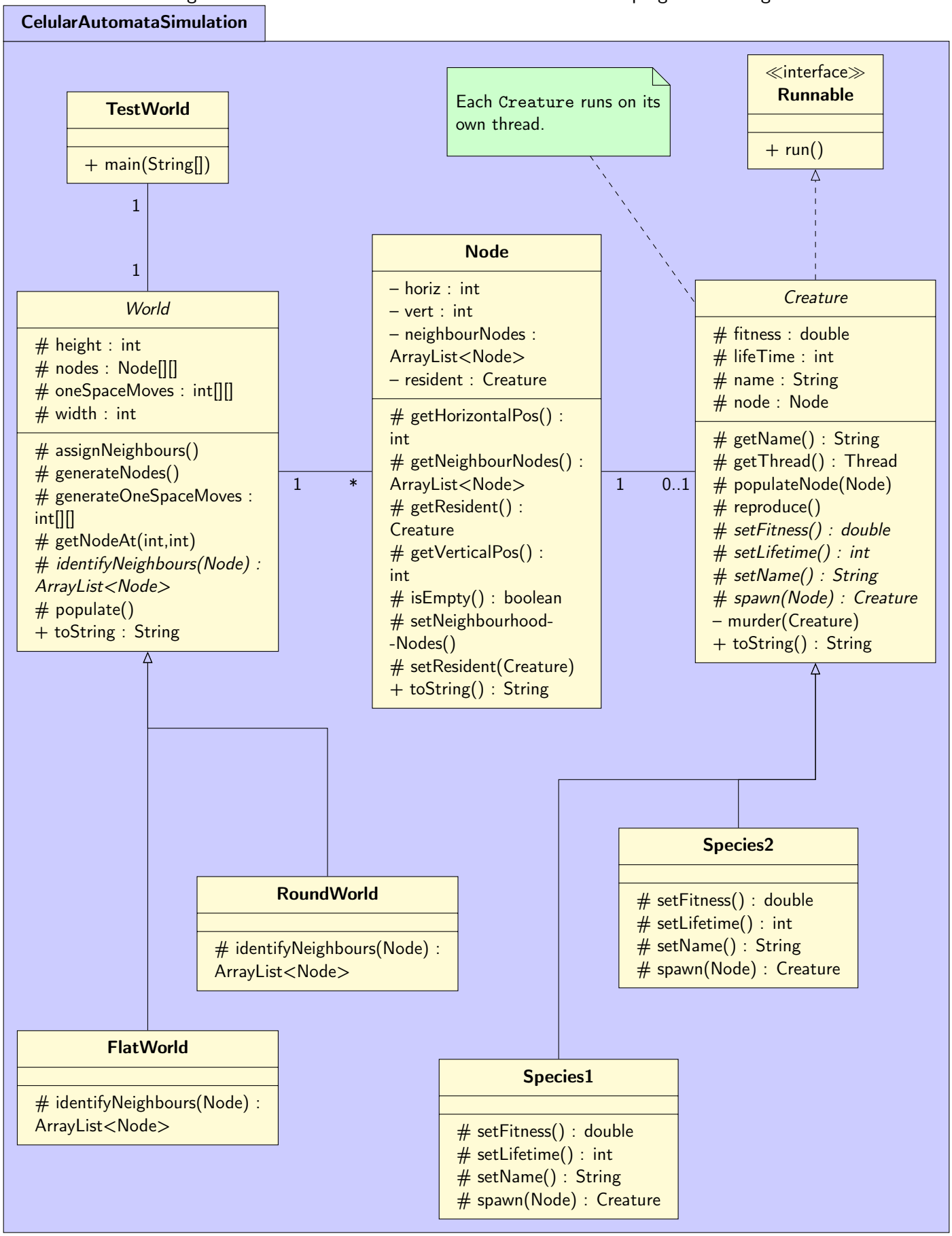
Advanced Programming
Assessed Exercise

Butler, Aidan
2281611B@student.gla.ac.uk

February 23, 2017

Contents

Figure 1: Celular automata simulation and visualization programme design



0.1 Celular automata simulation and visualization programme design

This programme is designed to implement and display a command line visualisation of a cellular automata simulation containing two self-reproducing types of agents, or creatures, with differing attributes which can exist, reproduce and destroy one another.

The programme begins when `TestWorld`'s `main` method generates a new `World` and begins visualising the state thereof periodically by printing a representation of the evolving world-state to the command line. The abstract `World` class can be instantiated as either a `RoundWorld` (wherein the edges of the `World` "wrap around") or a `FlatWorld` (wherein they do not). The differences between the two subclasses are determined by `World`'s abstract `identifyNeighbours()` method, which both classes must implement. The `World` contains an array of `Node` objects, each of which has an `ArrayList` of neighbouring `Node`s and either zero or one resident `Creature`s. The abstract `Creature` class has `fitness`, `lifeTime` and `name` attributes, which are set by abstract methods implemented in the `Species1` and `Species2` classes which inherit from it. A `Creature` will sleep for the duration of its `lifeTime` and if interrupted (`murdered`) will simply terminate. If uninterrupted, the `Creature` will `murder` other `Creature`s in the surrounding `Node`s and reproduce in them, depending on the result of randomised function's which take the `Creature`'s `fitness` attribute as an input.

The `World` constructor generates an initial population of `Creature`s by calling the `World.populate()` method. Each `Creature` is allocated its own thread. The `Creature`s are able to interact with one another by accessing a common heap containing the `Node`s. Thready-safety is provided by ensuring that `Creature`s lock `Node`s prior to making changes to them.

The use of abstract classes, such as `Creature`, containing abstract methods such as `Creature.spawn()` is designed to minimise code repetition and maximise class cohesion. As much code as possible is placed in abstract classes, meaning that changes can be carried out in inheriting classes in a uniform manner and by changing as few lines of code as possible. The use of threads is designed to allow each creature to function independently from the rest.

World-state visualisation is made possible by `TestWorld` calling the `World`'s `toString()` method, which in turn calls the `Node`'s `toString()` method, which in its turn calls `Creature`'s `toString()` method.