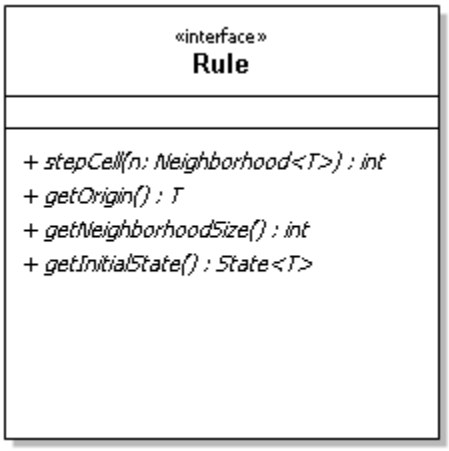
Design Rationale

Client Interfaces

Rule

We have three separate interfaces for the client to work with. The first is the Rule interface. This is the client's main interaction with the framework. The Rule interface defines four methods that are called by the Framework to determine how the automaton is to perform. Step Cell will take states for an area around the cell and provide a state for the current cell. This area can be of arbitrary size which was specifically designed to allow for much more powerful custom rules. The Rule interface also provides an initial state used to seed the universe.



Visualization

The Visualization interface provides only a single plugin. Given a state, it provides a Color object used to display the state on the grid. This was chosen to facilitate the implementation of a zoom feature. However, we not likely will be effecting this feature in the version of the framework that is to be turned in.

Viewport

Apart from the two major interfaces that we provide for the standard user of the framework, we will also be providing a special interface for a power user to implement custom visual representations of the cellular automaton. This Viewport acts as a painter that draws the grid and the cells given a state and a graphics object. Since the viewport is rather complex, the standard user will likely not have any interaction with the object. We will be providing a standard viewport that will be used by normal users and allow the power user to change viewports using a setViewport method.

Framework

Point

The framework itself allows for a user to provide any N-dimensional space for a rule set to work on. Given that, the abstract Point object holds information in an array of coordinates. We define a set of 1-Dimensional points and 2-Dimensional points for the users to use, though in theory, they could easily create their own N-dimensional point for a finite N. Points are the backbone of the framework. Everything in the framework depends on the dimensions of a point. Thus, an n dimensional point will create an n dimensional State (see below), n dimensional Neighborhood (also below), etc. We provide two implementations of points for 1-dimensional and 2-dimensional automata.

Neighborhood

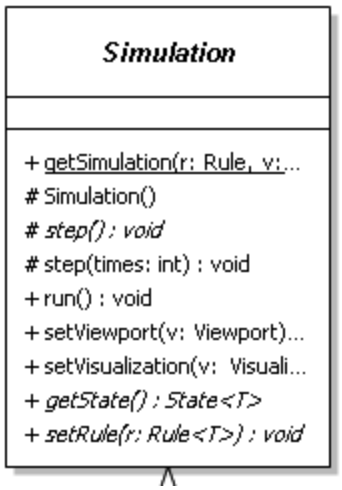
The Neighborhood interface is intended to provide the client with a method for interacting with the area around a cell. A neighborhood is centered on a point whose state is to be determined by the client's Rule. This is because a cell shouldn't know its position relative to everything else on the grid. For example, when determining the state of a point p in a 2-dimensional automata, a neighborhood would be provided to the client where (0,0) is the location of p in the neighborhood. A neighborhood is also iterable. This is simply to make the task of moving through the points in the neighborhood easier.

State

The State class is the most important class of the framework. It stores the state information of the automata. A HashMap of points to states is to be used so that we have a sparse data structure with very quick access times. The state is responsible for creating neighborhoods as well as performing a rule on itself. The state also provides the ability to define wrapping of its edges. If a point p extends off the grid, a state will either ignore the point or mod its coordinates (if edge wrapping is set for that dimension). This allows for very interesting properties such as either cylindrical or toroidal geometry in 2 dimensional spaces. Since a state's dimensions are determined by points, it is a generic type that is characterized by the point used to construct it.

Simulation

The simulation is the face of the framework. It is the object through which the user runs the framework. Upon calling the run method, the framework will provide a GUI used to interact with the program. The Simulation is also parameterized by a Point object to insure that the states and rules used are the same. Furthermore, we provide two default implementations (Simulation1D and Simulation2D) that are used to represent one and two dimensional automata. The class provides an easy way of obtaining one of these default simulations using the static getSimulation method. This was provided for the sole purpose of making the framework more user-friendly.



A simulation has methods that allow for the user to swap rules, visualizations, states, or even viewports at runtime. The only problem would be the number of allowable states from one rule to another. This will be rectified by setting any state above the maximum state k allowed by the new rule to k. This is possible since any the calculations in the program occur in a set of discrete disjoint time steps. So swapping any of these objects during runtime leads to well defined behavior.

Design Patterns

Factory Method:

A Simulation is a creator with the factory method being getSimulation. The concrete creators are the children of Simulation: either user created or one of Simulation1D and Simulation2D. The product would be another simulation of the same type as the child.

Template Method:

The step(int times) method in Simulation relies on the step() method defined in the subclasses. The step(int times) method will simply call the step method of the subclass multiple times.

Façade:

Points are rather complex objects. To simplify the interactions with the points, Point1D and Point2D provide simpler methods. They are thus facades to the Point class.