CA Explorer Architecture

For developers and the curious.
CA Explorer Design

The following is for anyone that might be interested in developing additional tools for the Cellular Automaton Explorer.

And it is for anyone who is curious about the design decisions made for the Cellular Automaton Explorer.

This material is not part of the class lectures.

But it might be useful if you are developing code for final projects, etc.
CA Explorer Flexibility

- The Cellular Automaton Explorer was designed as a research and teaching tool with flexibility and "ease of enhancements" as a primary goal.

- Most cellular automata are coded as cells with binary states and updates based on the values of nearest neighbors on one- and two-dimensional square grids. However, some very interesting simulations use vectors of binary numbers arranged on more complex grids (e.g., the Lattice Gas on a hexagonal grid). Some simulations, such as the Lattice Boltzmann, use real-valued vectors.

- While the last example may not be a true cellular automaton (depending on whether or not you are an integer-valued purist), the Cellular Automaton Explorer's design makes it relatively easy to add new lattice geometries, new rules, new neighborhoods, new display graphics, and new types of states.
Flexible Features 1

- **Arbitrary State Values**
  - Each cell in the Cellular Automaton Explorer is capable of storing arbitrary objects of any type. If it can be coded as a Java object, then it may be used. The most common state values are obviously numbers and arrays of numbers. When creating rules, the user can choose arbitrary states.

- **Arbitrary Rules**
  - The rules may be any function of the cell and its neighbors (and the neighbors can also be arbitrary). The rules are not restricted to the values of the current generation, and may access any number of previous generations. Among other possibilities, this makes finite-difference and finite-element simulations possible.
  - The rules can be added dynamically by the user. Write a little Java, drop it into a folder, and voila' the Cellular Automaton Explorer automatically incorporates your new rule. See the previous presentation on adding your own rules, titled “Write your own CA”.
Flexible Features 2

- **Arbitrary Lattices**
  - The most popular one and two-dimensional geometries are available – square, hexagonal, and triangular. The design makes other geometries relatively simple to add. The number and location of neighbors is easily changed for each geometry. (Currently, the user can add new geometries by adding classes to the `cellularAutomata.lattice` package, though new additions are not loaded dynamically and require a recompile. Dynamic loading is scheduled for a later release.)

- **Arbitrary Graphics**
  - Each simulation may choose arbitrary graphics. Using the “More Properties” feature, the graphics may even be changed on the fly. The code uses the popular model-view-controller design pattern to make graphics easily interchangeable.

- **Portability**
  - The Cellular Automaton Explorer will run on any platform that supports Java (all the major operating systems).
CA Explorer Speed

- **The price of flexibility is speed.** The Cellular Automaton Explorer can handle arbitrary lattice geometries with arbitrary rules and arbitrary state values. As such, there is no way to optimize the code for any particular simulation. Any optimization of one simulation would be inapplicable to others. Still, when possible, every effort was made to enhance speed by using simple graphics, decent data structures, and reasonable software engineering design patterns.

- **Many of the simulations will execute faster than the graphics can update.** By far, in any simulation, the biggest chunk of time goes to the display of graphics. If you have "the need for speed", then consider selecting the "Properties" option that updates the graphics only at the end of the simulation.

- By the way, some fantastic and wickedly fast "Game of Life" simulations are available elsewhere as free downloads, and these are worth a peek. These simulations have been optimized to ignore regions with no activity and to focus on bit manipulations in active regions. These wonderful enhancements are possible because the "Life" rule can ensure that the cell states are binary numbers, and that inactive regions won't suddenly change values.
Basic Flow of CA Explorer Code
(part 1)

- A description of the Cellular Automaton Explorer's flow may be useful to developers or anyone writing their own rule.

- **At Start Up**
  1. Reads a property file to determine the default user parameters such as the last used rule, geometry, grid size, initial state, etc. (CAPropertyReader).
  2. A factory (CAFFactory) instantiates a suite of compatible classes for the lattice, rule, and cell states. A separate Cell is instantiated for each grid site, and initial state values are assigned to each cell (CAStateInitializer). Each cell is stored by the lattice in an Iterator – no guarantees are made about the order of the elements in the iterator. The lattice finds the neighbors to each cell and stores them in a hash table (keyed by the cell).
  3. Graphics are instantiated (package cellularAutomata.graphics.).
  4. Listeners are instantiated by a variety of classes to await input from the user.
Basic Flow of CA Explorer Code (part 2)

- **After Start**

1. By listening for property changes, the CAController (and CAControllerListener) executes the basic actions initiated by the user. The "start" button in particular begins the CA simulation.
2. A simulation is started in a separate thread which loops until interrupted.
3. The loop increments from generation to each generation. At each generation, all of the cells are "simultaneously" updated and given a new value for the next generation.
4. In particular, the CAController loops over each cell (in the iterator) and calls the cell's `updateState()` method.
5. `updateState()` asks the lattice for the neighbors of the cell and then calls the selected Rule (e.g., Life). The rule updates the value of the cell (based on the values of its neighbors), storing the result in an ArrayList. Each cell has an ArrayList that keeps track of the current generation's value as well as previous generation's values.
6. At each generation, after each cell is updated the graphics are updated (including the display that indicates the current generation).
7. If a time delay has been set, then the program waits the specified time before continuing.
Basic Flow of CA Explorer Code (part 3)

- **When Properties are Changed**
  1. If the user selects a different rule, lattice, etc., then the Explorer re-instantiates everything.
  2. New properties are set.
  3. The `CAFactory` is called.
  4. See steps 2 through 4 of "At Start Up".
     - For graphics, only the simulation graphics are re-instantiated (not the menus and frame).

- **At Exit**
  - When the user exits the program, `CAShutDown` saves the properties to a file and calls `System.exit(0)` to ensure that resources such as graphics and unfinished threads are released.
The following classes are at the foundation of the Cellular Automaton Explorer.

- The presentation is divided into packages. Developers can use this page as a complement to the API
  - anyone else will want to rapidly flee to more interesting material.

- Not all classes are listed. Only foundational classes.

- **package cellularAutomata**
  - *Cell*
    - Stores the current state and previous states.
  - *CAFactory*
    - Creates sets of related lattices, states, and graphics.
  - *CACController*
    - Loops over the generations and updates the state of each cell.
package cellularAutomata.cellState.model

- **CellState**
  - Abstract base class for all cell states. May use any `Object` as a state, though integers are most common. Composes an `Object` or primitive type as the state. Also composes a `CellStateView` (see below). All other state classes extend this class (e.g., `NCellState`, `IntegerVectorState`, etc.).

- **CellStateFactory**
  - Creates cell states by requesting an appropriate state from the selected Rule. See `getCompatibleCellState()` in any Rule class.
CA Explorer Class Structure 3

- **package cellularAutomata.cellState.view**
  - *CellStateView*
    - Abstract base class for all cell state views. These views specify the way that a particular cell state will be displayed (see above). All other state views extend this class (e.g., *NCellStateView*, *IntegerVectorDefaultView*, etc.).

- **package cellularAutomata.graphics**
  - *CAFrame*
    - The outermost graphics frame.
  - *CAMenuBar, CAToolBar*
    - Bars added to the frame. Have their own Listeners.
  - *StartPanel, StatusPanel, PropertiesPanel*
    - Panels added to the frame.
  - *AllPanelListener*
    - Listens for events occurring on the StartPanel, StatusPanel, and PropertiesPanel.
package cellularAutomata.lattice

Lattice

Abstract base class for all lattices. All implementations (SquareLattice, HexagonalLattice, etc.) are the "model" part of the model-view-controller design pattern. Any new Lattice class is automatically loaded into the Explorer and added to the drop-down properties list. In other words, any new Lattice is self-contained code, and no other updates are necessary to any other code elsewhere.

package cellularAutomata.lattice.view

LatticePanel

Abstract base class for all lattice views. Extends JPanel and is the basis for the cellular automata display. All implementations (SquareLatticePanel, HexagonalLatticePanel, etc.) are the "view" part of the model-view-controller design pattern. Implements paintComponent() which calls the abstract method draw(). All child classes implement draw() by writing on a Graphics object that is passed in from paintComponent().

TwoDimensionalLatticePanel, OneDimensionalLatticePanel

Abstract class extending the LatticePanel. Implements typical graphics for lattices that can be represented by one- or two-dimensional arrays. Queries each cell on the lattice for its CellState. The state supplies its CellStateView. These classes then draw that view in the correct location. Updates to off-screen graphics (BufferedImage) for speed.

SquareLatticePanel, HexagonalLatticePanel, etc.

Implementations of the TwoDimensionalLatticePanel. Provides the panel with specific information about the geometry of the lattice (such as a cell's width, height, and coordinates).
CA Explorer Class Structure 5

- **package cellularAutomata.reflection**
  - A collection of utilities that handle reflection used to load rules, lattices, and analyses.
  - *ReflectionTool*
    - Dynamically loads analyses, rules, and lattices.
      - A bit of a pain. Causes roadblocks in debugging, but that’s the double-edged sword of reflection.
  - *AnalysisHash, RuleHash, LatticeHash*
    - Keeps track of which analyses, rules, and lattices have been loaded. The hash table keys are descriptions of the class (for example, from the *Rule* method `getDisplayName()`). The hash returns a value that is the class name. In this way, a user can select a description of the rule, then this gets a class name which is then sent to the *ReflectionTool* (by the *CAFFactory*) for instantiation.
package cellularAutomata.rules

- Rule
  - Abstract base class for all rules. Has an abstract method that takes two parameters (a cell and its neighbors) and returns a new state value for the cell.

- Life, LatticeGas, MajorityRules, etc.
  - Implements the BinaryRuleTemplate, IntegerRuleTemplate, etc. (see below). The cell's new value (state) is calculated from its state and the states of its neighbors. Any new Rule classes are automatically loaded by reflection and are self-contained code. In other words, no other changes are necessary to other code anywhere else.

package cellularAutomata.rules.templates

- BinaryRuleTemplate, IntegerRuleTemplate, etc.
  - Convenience classes that make it easier to build new rules. See the previous presentation on adding your own rules, titled “Write your own CA”.
CA Explorer Class Structure 6

- **package cellularAutomata.storage**
  - `FileStorage`
    - Abstract base class for all classes that save data to a file.
  - `OneDimensionalFileStorage`, `TwoDimensionalFileStorage`
    - Implements the `FileStorage`. Stores the CA data for later retrieval.

- **package cellularAutomata.util**
  - Commonly used utility classes including those that deal with reflection for dynamic loading of new rules and lattices.

- **package userRules**
  - A location for rules that are added by the user. These classes are automatically loaded by reflection and are self-contained code. In other words, when the user adds a new rule, no other changes are necessary to other code anywhere else.
package cellularAutomata.analysis

- Analysis
  - Base class for all analysis tools.
- MusicAnalysis, PopulationAnalysis
  - Turns CA into music and population statistics.

package userAnalysis

- A location for analyses that are added by the user. These classes are automatically loaded by reflection and are self-contained code. In other words, when the user adds a new analysis, no other changes are necessary to other code anywhere else.