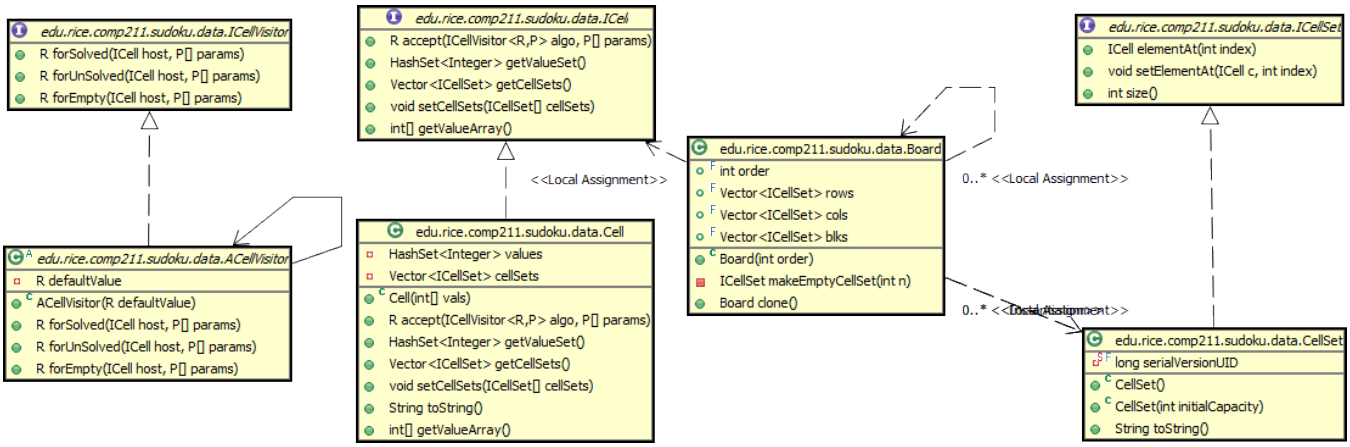
211hw11_details_S11

Comp211 HW11 Supporting Code Details

Here is a link to the Java 6 API documentation. Look here for information on any unfamiliar Java class, e.g. `HashSet<T>` or `Vector<T>`:

http://download.oracle.com/javase/6/docs/api/

Core Data Structures



The main data structures are represented by only a few interfaces:

- **`ICell`** -- a single square on the Sudoku board.
  - A cell has a set of integer values, which are accessible either as a `HashSet<Integer>` or as an array of integers.
  - A cell also has a collection (Vector<ICell>) of `ICellSets` to which it belongs. These are the cell's row, column and block, in that order.
    - Don't forget that the cell is also a member of every ICellSet that it references.
  - A cell can be in one of three logical states:
    - *Empty* - there are no values contained in the cell. This corresponds to the situation where the board is unsolvable because a cell has no possible value.
    - *Solved* - the cell contains exactly one value. <u>If and only if you know for sure that the cell is in this state</u>, then the single value can be accessed as `aCell.getValueArray()[0]`.
    - *Unsolved* - the cell contains multiple possible values. At most, a cell could contain `order*order` values.
  - `Cell` is a concrete implementation of `ICell`.
  - An `ICell` accepts an **`ICellVisitor`** which has cases for each of the 3 logical states, described above.
    - The `ACellVisitor` class is an abstract convenience class that provide a default return value for any cases that the developer does not which to override. There is no requirement to use this class, though it may simplify certain code.
- **`ICellSet`** -- A collection of `ICells` that represents a row, column or block.
  - For convenience sake, the cells are in a distinct order (i.e. left-to right for a row) and thus each cell is addressable by an index value.
  - An `ICellSet` is `Iterable` which means that it can be used in for-each loops, e.g.

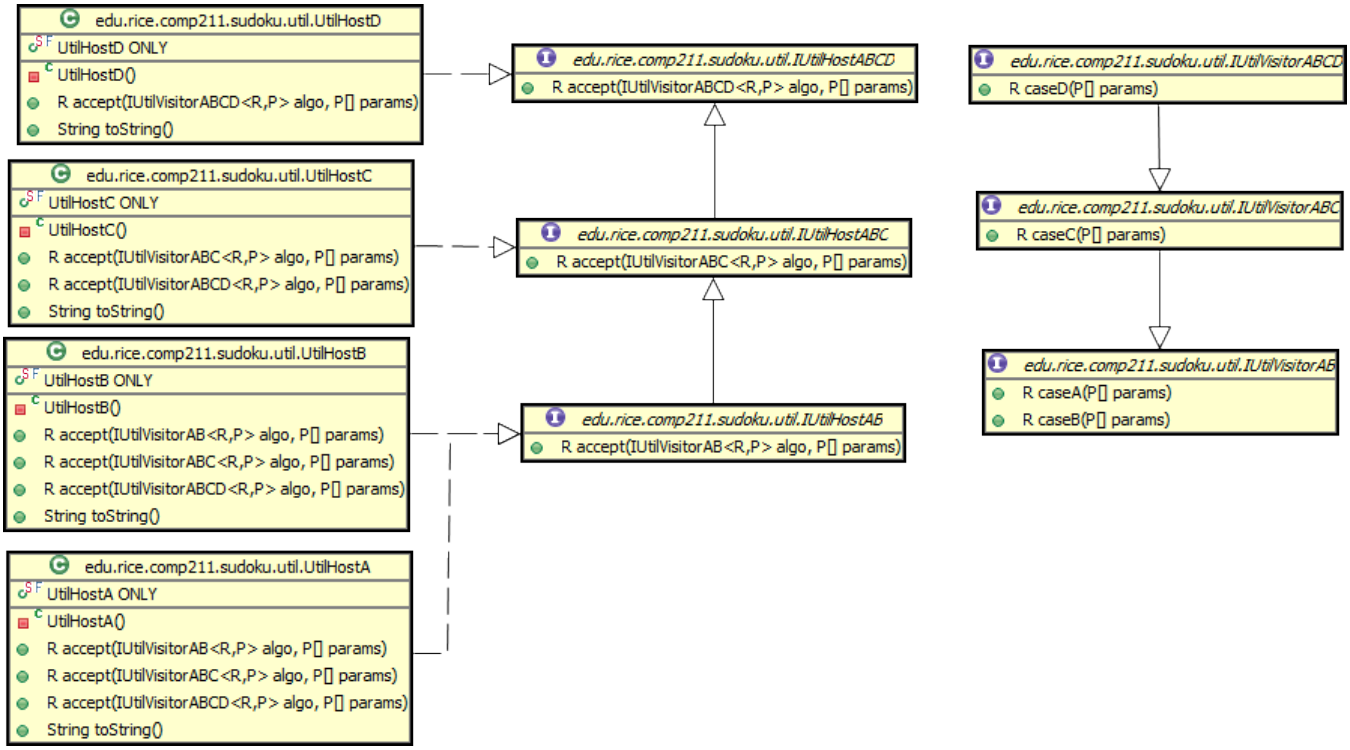
```
for(ICell c : aCellSet) {
    ...
}
```

- **`Board`** -- A set of 3 Vector<ICellSet>'s where each element of the vector is an `ICellSet`. These vectors represent the entire set of rows, columns or blocks in a game.
  - A `Board` has the ability to make a "deep" copy of itself, where everything, all the way down to the individual cells, is copied. This is the c`lone()` method.
  - The `Board` also has a utility method for generating empty cell sets, which is used during initialization.
  - It should be noted that since *every* cell appears *exactly once* in each of the `rows`, `cols`, and `blks` `Vectors` of a `Board`, *any* of those 3 vectors will server equally well as a means to iterate over all the cells of a `Board`

Utility Data Structures

Delegation-model programming and imperative (conditional and loop-based) programming have a fundamental incompatibility: Delegation works by having code on the <u>inside</u> of an object (i.e. a method of that object) perform the object-specific processes. On the other hand, imperative programming utilizes program structures (conditionals and loops) which are <u>outside</u> the object to perform object-specific processes. This can cause major architectural headaches when combining both styles in a program, which is what we are doing here. For instance, see the programming hints section on the previous wiki page on how to control loops and mutate data while in the middle of a delegation process.

The following utility data structures are designed to handle the situation where an operation has more than 2 or more possible outcomes. While not entirely optimal, these classes show how clever sub-classing can enable one to re-use classes in multiple situations.



- **`IUtilHostAB` -- `IUtilVisitorAB`** : A host-visitor set that represents a situation where there are two possible outcomes. There are two possible concrete hosts, `UtilHostA` and `UtilHostB` and thus the `IUtilVisitorAB` visitor has two cases corresponding to these two hosts.
- **`IUtilHostABC` -- `IUtilVisitorABC`** : A host-visitor set that represents a situation where there are three possible outcomes. There are three possible concrete hosts, `UtilHostA`, `UtilHostB` and `UtilHostC` and thus the `IUtilVisitorABC` visitor has three cases corresponding to these three hosts.
- **`IUtilHostABCD` -- `IUtilVisitorABCD`** : A host-visitor set that represents a situation where there are four possible outcomes. There are four possible concrete hosts, `UtilHostA`, `UtilHostB`, `UtilHostC` and `UtilHostD` and thus the `IUtilVisitorABCD` visitor has four cases corresponding to these three hosts.

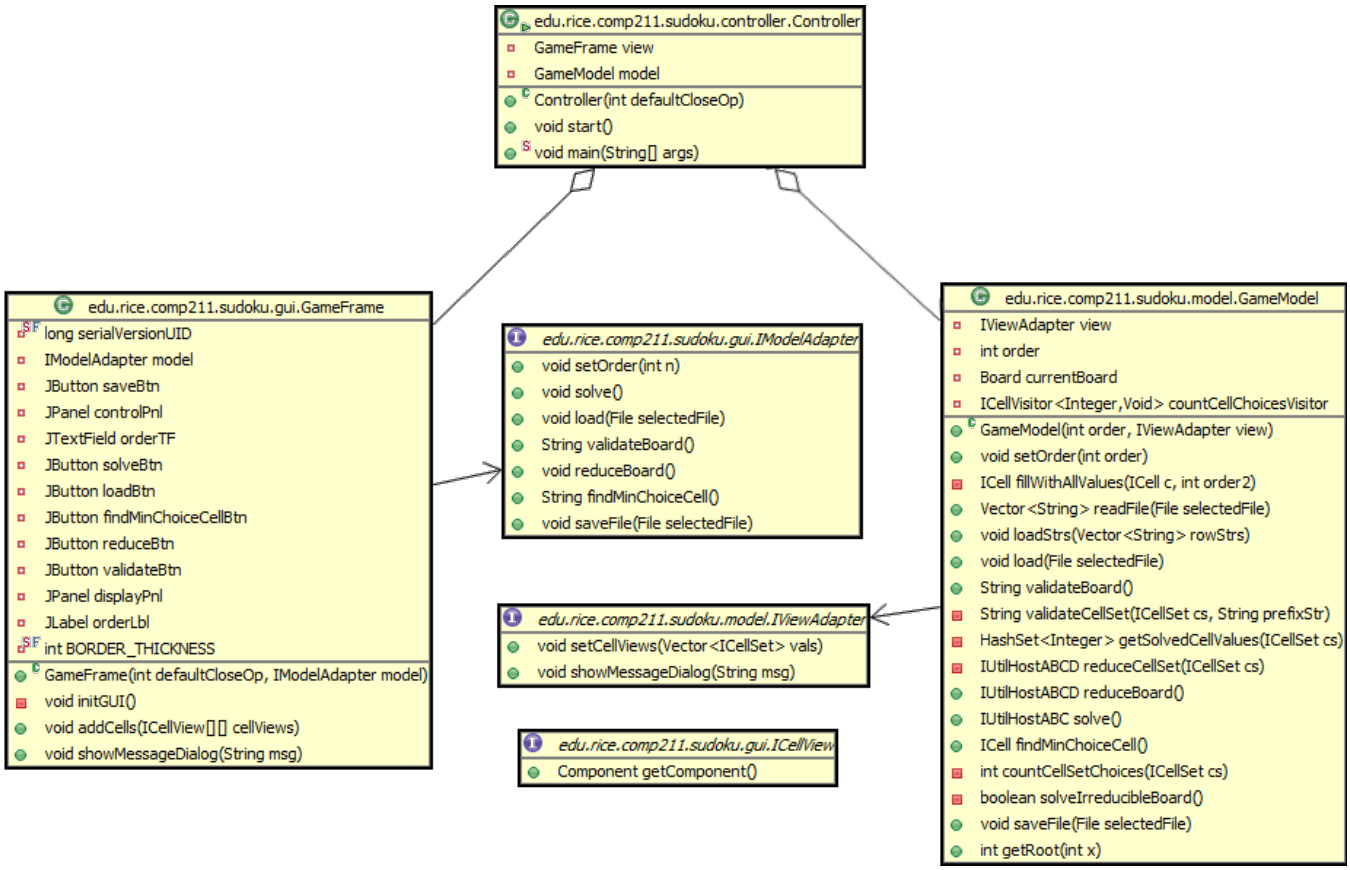
Typically, a method returns a value of type `IUtilHostAB/C/D` depending on how many possibilities there are. For instance, a single reduction pass on the Sudoku board would result in a value of IUtilHostABCD, corresponding to the 4 possible outcomes for the board: being still reducible, irreducible, solved or unsolvable.

The caller of such a method need only to simply delegate to the returned value by having the returned value accept the proper visitor. Suppose the method `my3OutcomeMethod` has three possible outcomes, so it returns an `IUtilHostABC` object. All we need to do is to delegate to the returned value of the method:

```
my3OutcomeMethod().accept(new IUtilVisitorABC() {...})
```

Model-View-Controller Architecture

A discussion of the details and implications of the Model-View-Controller Design Pattern are beyond the scope of this class, so here is but a brief overview:



The point of the MVC is to separate how a program looks to the user from what it does. In the above UML diagram, you can see that we have two separate packages: `edu.rice.comp211.sudoku.gui` and `edu.rice.comp211.sukodu.model` that hold the "view" and "model" components respectively.

The view, `GameFrame` communicates with the rest of the system via its `IModelAdapter` which is also in the `gui` package. The `ICellView` is used to communicate to the view, and so, it is also in the `gui` package. The `gui` package does **not** define what the `IModelAdapter` does! But the `GUIFrame` does not know this--so far as it can tell, it's entire world is defined by the `gui` package because there are no references to anything outside of the package.

Likewise, the model, `GameModel` communicates with the outside world via its `IViewAdapter`. The model does use the code in the `data` and `util` packages, but only to define data structures and provide utility capabilities. All communications to the view take place through the `IViewAdapter` which is in the `model` pacakge. Like the view, the model code has no idea what the implementation of `IViewAdapter` is.

The `view` and `model` packages are thus completely isolated and independent from each other. The job of the `Controller` class in the `edu.rice.comp211.sudoku.controller` package is to assemble the complete, operational application by joining the `view` and `model` together. It does so by creating implementations of both `IModelAdapter` and `IViewAdapter` that pass their method calls onto the `GameFrame` and `GameModel` respectively. This may involve simply passing the call along to the appropriate method on the receiver, for instance, `IModelAdapter.solve()` is implemented to simply call `GameModel.solve()`. But other methods may require more extensive translation, for instance, `IviewAdapter.setCellViews()` is implemented to take a `Vector<ICellSet>` object, convert it into a 2-dimensional array of `ICelllViews` and then make the call to `GameFrame.addCells()`.

Thus, the `Controller` is the class that contains the `main()` method that starts the application. The constructor of `Controller` instantiates the model, `GameModel`, the view, `GameFrame`, and the adapters that communicate between them, `IViewAdapter` and `IModelAdapter`. The method `Controller.start()` is then called to actually start the application.

The `Controller` is the only class that knows which view and which model are used for any given application.

Key Methods of `GameFrame`

- `initGUI` - intializes the GUI components.
- `addCells` - displays the given cells on the screen. Used to update the view with the latest state of the board.
- `showMessage` - a utility method to display the given string as a pop-up dialog box.

Key Methods of `IModelAdapter`

Most method are self-evident. See the Javadocs.

- `validate` -- performs a validation of the board, returning a string that describes the current state of the board.
- `findMinChoiceCell` performs a search of the board to find the cell whose choices could be tested and that would lead to the fastest solution of the board. The heuristic used to choose the cell is up to the student. The returned string is the `toString()` of the chosen cell.

Key Methods of `GameModel`

There are a number of utility methods to perform useful self-evident tasks--see the Javadocs.

- **`currentBoard`** - a field which references the board that is currently being solved.
- **`loadStrs`** -- takes a vector of strings, which are the rows of a puzzle, and translates the strings into the cells of a new current board. This method is compatible with the puzzle generation utilities, but is not yet hooked up to them.
- **`validateBoard`** - performs a validation check on the current board and returns a string describing the current state of the board. This method calls `validateCellSet` on every cell set of the rows, columns and blocks of the board. *Examine this method carefully to get ideas on how to process the board!*
    - Important Note: This method should NOT BE USED as part of the solving process!! It was designed to give textual feedback to the user and not to be part of a solving algorithm. You *will be marked down* for using this method and attempting to parse it return string to determine the state of the board. In fact, it is less effecient to separately check the validity of the board during the reduction and solving process -- the board reduction algorithm (`reduceBoard()`) should be able to determine the status of the board as a byproduct of its actions, thus negating the need for a separate validation operation.
- **`getSolvedCellValues`** -- gets a `HashSet<Integer>` containing all the values from the solved cells in a cell set. This is used when reducing a cell.
- **`countCellChoices`** -- a utility method that will return the number of choices in all the unsolved cells in a given cell set
- **`reduceCellSet`** -- [*Student implemented*] perform a single reduction pass on every cell in a given cell set.
    - Start by having `reduceBoard` simply call this method on a specific cell set.
- **`reduceBoard`** -- [*Student implemented*] perform a single reduction pass on every cell in the board.
- **`solve`** -- [*Student implemented*] perform reduction passes on the board until the board is either shown to be solved or unsolvable. This method should be able to handle boards that are reducible, irreducible, or unsolvable.
    - The method is defined as also allowing a return status of "irreducible", though in the final implementation, that should never occur. This return value is allowed so that you can do a partial implementation of the solve algorithm *before* you implement `solveIrreducibleBoard`.
- **`findMinChoiceCell`** -- [*Student implemented*] Use some heuristic to find the best choice for a cell whose values represent the fastest path to a solution. This method is used when an irreducible board is encountered and the solving process needs to iterate through the choices of a given cell to test for possible solutions. If a board is solvable, on *any* cell, at least one value choice in an unsolved cell will always lead towards a solution. Other choices may lead to unsolvable boards.
- **`solveIrreducibleBoard`** -- [*Student implemented*] Given an irreducible board (a board that doesn't change when a reduction pass is performed), looks for a solution by testing the choices of a chosen cell. Note that a particular value choice in a cell could lead to an irreducible board, so this is fundamentally a recursive process!

Key Methods of `IViewAdapter`

- `setCellViews` -- takes a vector of cell sets that define the blocks of a puzzle (`Board.blks`) and displays the board on the screen.
- `showMessageDialog` -- utility method to show a string on the screen as pop-up dialog box. This is useful for showing status results.

Testing the GameModel

For testing purposes, a `GameModel` can be instantiated using an implementation of `IViewAdapter` whose method a no-ops. The methods of the `GameModel` can then be called.

- Test game files can be made to load the `GameModel` with well-defined boards for testing.
- `GameModel` can both save and load games at any stage of reduction, so well-defined testing scenarios can be created where the board is in an exactly defined state, i.e. where every cell's value(s) are known precisely.
- The supplied code includes some examples of how you can instantiate a `GameModel` object, load puzzle boards with it and test it against other boards.

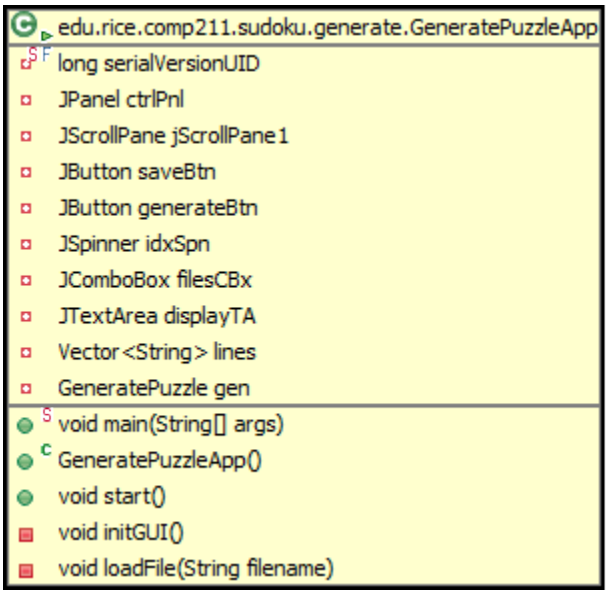
Puzzle Generation Utilities

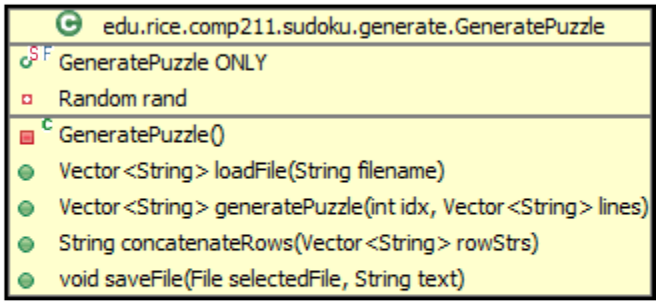
The supplied code also contains a self-contained package, `edu.rice.comp211.sudoku.generate`, that is used to read Sudoku puzzles from text files containing a total of over 50,000 solvable puzzles. These puzzles come from www.printable-sudoku-puzzles.com but are a different format than the Comp211 Sudoku solver uses. These utilities, which are all contained in the `GeneratePuzzle` class, can read those data files, pick either a specific puzzle from them, or a random puzzle, display it and save the individual puzzle in a format that the Comp211 Sudoku solver can read.

`GeneratePuzzleApp` is a a simple GUI interface to `GeneratePuzzle` that allows the user to easily perform conversions. Note that the higher numbered data files (located in `data\puzzle-src`) have more difficult puzzles. Each data file contains about 10,000 puzzles numbered from 1-10000 (approx.). An index of zero means to choose a random puzzle from the data file.

{[GeneratePuzzleApp]} has its own `main()` method and thus can be run as a separate, stand-alone application apart from the Sudoko solver.

See the Javadocs for more detailed information on the individual methods.


edu.rice.comp211.sudoku.generate.GeneratePuzzleApp
long serialVersionUID
JPanel ctrlPnl
JScrollPane jScrollPane1
JButton saveBtn
JButton generateBtn
JSpinner idxSpn
JComboBox filesCBx
JTextArea displayTA
Vector<String> lines
GeneratePuzzle gen
void main(String[] args)
GeneratePuzzleApp()
void start()
void initGUI()
void loadFile(String filename)


edu.rice.comp211.sudoku.generate.GeneratePuzzle
GeneratePuzzle ONLY
Random rand
GeneratePuzzle()
Vector<String> loadFile(String filename)
Vector<String> generatePuzzle(int idx, Vector<String> lines)
String concatenateRows(Vector<String> rowStrs)
void saveFile(File selectedFile, String text)