Grid Game Framework

Individual Software Design and Development
305-01
Chase Dreszer
December 3, 2015

Contents

1 Design Overview

This document specifies the structure of the Grid Game Framework and the relations among its components, as well as any design patterns that may have been utilized.

1.1 Grid Game Framework Package Structure

The grid game framework is separated into three packages; the *default* package which contains the main app, the *gridgameviews* package containing the implementation of the graphical user interface and console user interface, and the *gridgames* package containing implementation of key features and templates to create fully functional games.

The GUI and console user interfaces are separated from the rest of the framework to ensure that the embedding of user interfaces within the internal data representation is avoided.

Package	**Classes**
default	GridGameApp
gridgameviews	GridGameGUI GridGameConsole
gridgames	GridGame GameClock Preferences PlayingArea MyRendarable CellRenderer GridAction QuitAction HOFAction SizeAction HallOfFame<PlayerEntry> PlayerEntry

Table 1: Package Overview

1.2 Underlying Model

The *gridgames* package provides the structure for the underlying model for creating games, customizable grid game cells, customizable game menu actions, an ascending or descending game clock, the ability to read and manage preferences, and the option to enshrine a hall of fame.

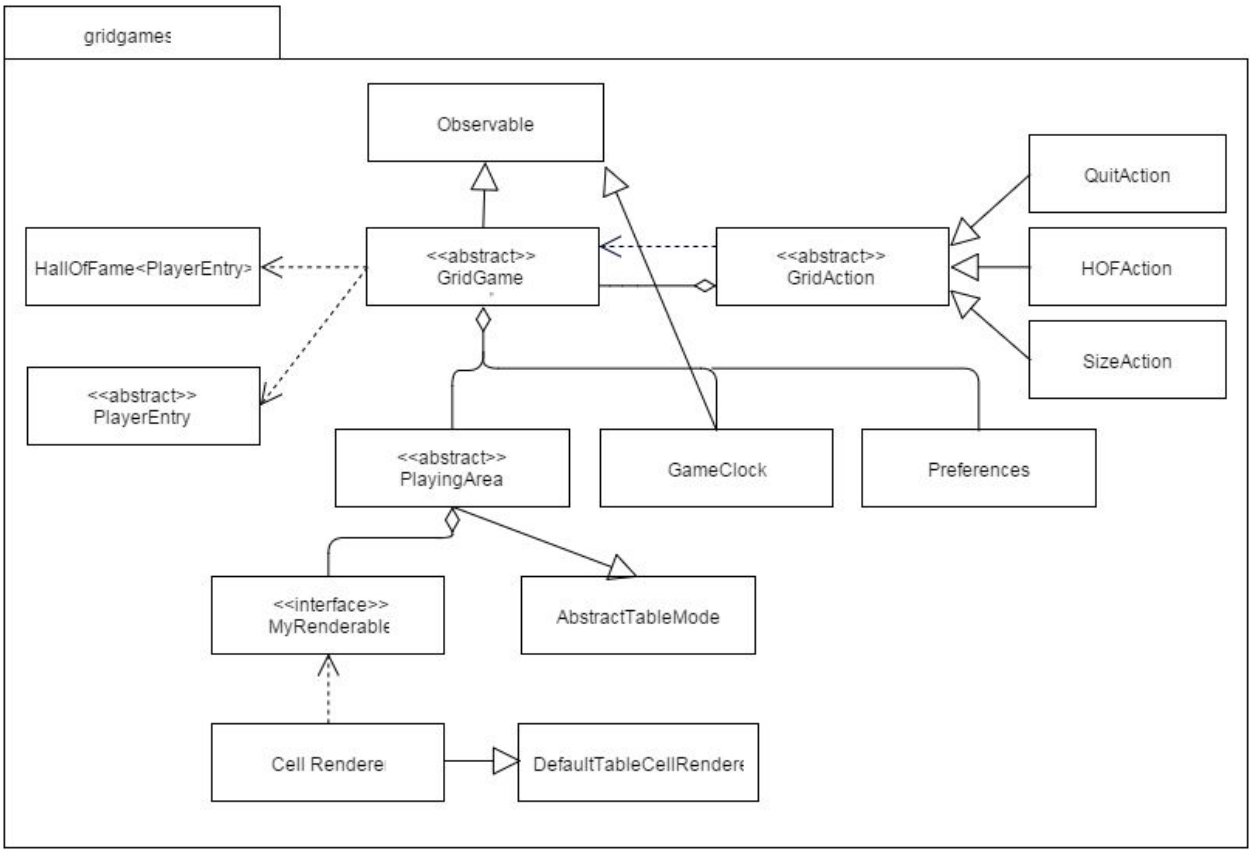

Figure 1: High-level UML digram of *gridgames* package

1.2.1 Grid Game

The abstract class GridGame extends observable and is the main communicator with the views and therefore contains all necessary components for a functional game. GridGame has an instance of preferences, a playing area, and a game clock, as well as factory methods to get a list of the game's menu actions and another to retrieve the hall of fame.

1.2.2 Playing Area

The PlayingArea class is the underlying model for the two-dimensional playing area that is to be displayed and is composed of a two-dimensional array of customizable renderable cells. The JTable within the GUI automatically updates whenever the fireTableChanged() method is invoked because the PlayingArea extends AbstractTableModel. All cell modification is done by modifying the renderables in the playing area.

1.2.3 Customizable Grid Cells

The MyRenderable interface applies the strategy pattern, allowing for plug-in authors to implement their own concrete strategy to customize their renderable cells. Each renderable has a symbol and a renderable image map, which maps the appropriate symbol to the desired image. Renderables can either be displayed on the GUI as an image or text, or displayed on the console as its corresponding symbol.

1.2.4 Game Menu Actions

The abstract class GridAction extends Action, which allows for the user to specify label, icon, and accelerator for the action. GridAction also takes in a GridGame as a parameter in order to be able to manipulate its behavior when an action is performed. The framework provides grid actions for quitting, setting the hall of fame to be displayed, and for resizing the board.

This abstraction of grid game actions allows for the process of adding customizable menu items to both the GUI and console to be easy. The abstract method getMenuActions() within GridGame is a factory method that returns a list of GridActions that should be added to the game menu. By making the method abstract it allows plug-in authors the freedom to specify their own custom or provided menus.

1.2.5 Preferences

The preferences for the grid game are read through a .ini file provided by the plug-in author and are then translated into menu items, as well as mapped as key-value pairings. The getPreferenceValues() method in the Preferences class allows for plug-ins to access a map with key-value pairings of section to map of all options and corresponding values

within the section. The Preferences class also has a method getCurPreferences which return a map that shows for each section, which option is currently chosen. These maps together allow for the plug-in author to construct customized preference menu options. The plug-in author should check for updated preferences at each new game call. The getConsolePreferenceMenu() allows for the console to retreive the console menu with no extra work on its end.

1.2.6 Game Clock

The GameClock class is implemented so that it is given a constructor specifying whether or not the clock is ascending or descending and is given a specified start time. If the clock is ascending, the desired start time is ignored. The GameClock class extends observable and notifies all observers after each second interval.

1.2.7 Hall of Fame

The abstract method getHallOfFame() in GridGame is a factory method that retrieves the hall of fame the plugin is using from the specified file path.

The HallOfFame<PlayerEntry> class takes in a file path for which it should be saved, as well as the size of the hall of fame, and the number of players that should be displayed. The HallOfFame<PlayerEntry> class implements serializable so that it can be easily saved to persistent storage. The abstract class Player Entry, which the hall of fame takes in, implements comparable so that the plug-in author has the ability to customize how the hall of fame is sorted. The hall of fame class does not contain any repeats because it is composed of a TreeSet of type PlayerEntry, so if the player entry is determined to be the same as another (determined by plug-in author) it will not be added in.

Player entries are created and added into the hall of fame after the user is prompted to enter in their name. This process is done from within the GUI or console component.

1.3 Grid Game Views

Type	Components Involved
Observer Pattern	GridGame (observable) and GridGameGUI (observer) GridGame (observable) and GridGameConsole (observer) GameClock (observable) and GridGameGUI (observer)

The two grid game views, GUI and console interface, both follow the observer pattern and extend observable. Both views are passed a GridGame as a parameter and are notified when there are changes within the game and update their displays accordingly.

1.3.1 GUI Component

In addition to observing the GridGame class, the GUI component observes the GameClock within the GridGame class in order to update its timer when necessary.

The GUI utilizes layout managers (BoxLayout) in order to display the title bar, menu bar, status area, and two-dimensional table from top to bottom.

1.3.2 Console Interface Component

The console interface component uses the observer pattern similarly to the GUI, however it is not an observer for the game clock and thus does not update with every interval of the game clock.

1.4 Grid Game App

Reflection is used in order to create an instance of some class that extends GridGame at runtime. This allows for their to be any number of plug-ins rather than a fixed amount. Using Apache Commons CLI library the grid game app gets the command line and parses its values based on options.

2 HOW-TO

2.1 Introduction

The kind of games supported by this framework are "grid games" that are played on a square grid of cells. The framework will provide you a GUI with a menu bar, status area, and two-dimensional table, as well as a console interface. The framework also provies you with built in hall of fame functionality, customizable game and preference menus, as well as the ability to implement an elapsed time or countdown timer.

Writing a grid game plugin:

1. Create class that implements MyRenderable
2. Create class that extends PlayingArea
3. Create class that extends GridGame
4. Create class that extends PlayerEntry
5. (Optional) Create classes that extend GridAction

Constraints:

- You must name the package your plug-in is in the name of your game and the class that extends GridGame must follow the naming convention ___Game.

  ex.

  minesweeper , MinesweeperGame
  collapse , CollaspeGame

- To build the preference menu, you need to return a preference action list and therefore must know the names ahead of time.
  - And at each newGame() call there has to be checking and implementation of preference values.

2.2 Customizing Images and Text

When implementing the MyRenderable interface, you will be asked to implement the getRenderableImageMap(). In order to correctly implement this method you must provide path files to any images you may want to render (if none, don't worry about it) mapped to the corresponding symbol you wish that token to have.

If you don't have any images simply set the isImage() method to always return false.
If the isImage() method returns false the rendered will print out the cell as a html tag instead, displaying the renderable symbol as text.

```
/**
 * Returns a map of all the renderable token's names
 * for the grid game mapped with the file path to their images, if anything.
 *
 * Key = piece name
 * Value = path to image
 *
 * @return map of piece names and images
 */
public Map<Character, String> getRenderableImageMap()
{
    Map<Character, String> tileImages = new HashMap<Character, String>();

    tileImages.put('-', pkg + folder + hidden);
    tileImages.put('*', pkg + folder + explodedBomb);
    tileImages.put('B', pkg + folder + bomb);
    tileImages.put('@', pkg + folder + flagged);

    return tileImages;
}
```

2.3 Customizing Game Menu and Creating Grid Actions

When extending the GridGame class, you will be asked to implement the getMenuActions() which allows you to customize the game menu actions and return a list of all actions in the game. The framework provides the HOFAction and the QuitAction, but in order to customize you own actions you need to extend GridAction.

Most of the trickier parts are done for you when you make a super call to the GridAction constructor. You should be passing your plugin class that extends GridGame to the GridAction as a parameter in order to be able to adjust the behavioral functionality of the game.

```
public class RestartAction extends GridAction
{
    //the game to restart
    private CollapseGame game;

    /**
     * The constructor for a GridAction allows for the author to assign a label
     * to the action, an icon for the action, and an accelerator for the action.
     *
     * @param label - the name of the action to appear in the menu
     * @param icon - the icon to appear next to the name if any
     * @param mnemonic - the keyboard shortcut for the action
     * @param game - the game to perform the action on
     */
    public RestartAction(String label, ImageIcon icon,
                         Integer mnemonic,
                         CollapseGame game)
    {
        super(label, icon, mnemonic);
        this.game = game;
    }

    /**
     * Performs a restart on the grid game.
     * @param event - the action event.
     */
    public void actionPerformed(ActionEvent event)
    {
        game.restartGame();
    }
}
```

After creating some GridActions, you can now implement the getMenuActions() method and have a fully functioning game menu for both the GUI and console.

```
/**
 * Returns a list of menu actions for the observable game.
 * @return list of menu actions
 */
public List<Action> getMenuActions()
{
    //Creates menu actions
    Action restart = new RestartAction("Restart", null, KeyEvent.VK_R, this);
    Action newGame = new NewGameAction("New Game", null, KeyEvent.VK_N, this);
    Action undo = new UndoAction("Undo", null, KeyEvent.VK_U, this);
    Action hof = new HOFAction("Hall", null, KeyEvent.VK_H, this);
    Action quit = new QuitAction("Quit", null, KeyEvent.VK_Q, this);

    List<Action> actions = new ArrayList<Action>();

    //adds menu actions
    actions.add(restart);
    actions.add(newGame);
    actions.add(undo);
    actions.add(hof);
    actions.add(quit);

    return actions;
}
```

2.4 Getting the Hall of Fame

When extending the GridGame class, you will be forced to implement the getHallOfFame() factory method which retrieves the hall of fame. Here is where you can specify where you want the hall of fame saved, the size of the hall of fame, and how many people should be displayed.

Due to the hall of fame persisting in memory, you can elect to just return a new instance of the hall of fame with the file path when the framework requests getHallOfFame().

Recommended implementation:

```
/**
 * Returns an instance of the HallOfFame for the current game.
 * @return HallOfFame (filename, size, how many to be displayed)
 */
public HallOfFame<PlayerEntry> getHallOfFame()
{
    return new HallOfFame<PlayerEntry>("collapse/HOF.txt", kSizeOfHOF, kSizeOfHOF);
}
```

2.5 Running Grid Game Application

In order to run your plugin, the first command line argument must be in the form:

"plugin.PluginGame"

ex. "collpase.CollapseGame", "minesweeper.MinesweeperGame"

Where the first part of the String is the package name and the second part is the name of the class that you wrote that extended GridGame.

All other command line arguments are structured in this manner:

```
Command line usage:  java CollapseApp -cg [-b boardfile] [-i file1] [-o file2]
 *  -b,--board      predefined board file.
 *  -c,--console        use console
 *  -g,--gui            use gui
 *  -i,--infile     file to use instead of Standard Input.
 *  -o,--outfile    file to use instead of Standard OutInput.
```

2.6 Application Data Files

You must provide a **preferences.ini file** that has all desired preferences within it formatted as shown below. Preferences file must be located in plug-in package and package must be the name of your game.

```
#Sample Collapse preferences.ini File

[Board Size]
small = 8
medium = 10
large = 12

[Skin]
Default = Default
Serenity = Serenity
NeoPet = NeoPet
Giraud = Giraud
```

3 Known Issues

- Have to use @SuppressWarning “unchecked” for the Hall of Fame serialization.
- Preferences customizability not fully implemented for console.
- HOF asks for input dialog regardless of whether or not the user will be entered, but doesn’t enter you if you don’t qualify.