Game Package

Henry Z. Lo

June 30, 2014

1 Game Usage

This document refers to the game package. Be sure to reference the code as you read the document.

1.1 Game class

Central to the game package is the `Game` abstract class, from which all other games are derived. The `Game` must contain the following information:

- Whose turn it is (`whoseTurn()` method)
- The current valid moves (`getMoves()`)
- What results after a move is made (`make(Move m)`)
- Whether the game is over (`isGameOver()`)

Using a game first consists of initializing it. Then, until the game is over, we find out whose turn it is, present that player with all possible moves, and make one of those moves.

As an example, see `cs310.TestGame.java`. This program:

1. Starts a game of Nim, which consists of three piles, from which a player can take sticks.
2. Calls `getMoves()` to display all possible moves.
3. Makes each of those moves, displaying the results.

`TestGame` does not really play a game, but shows the effect of each move at the beginning. In order to play a game, we must repeatedly call get and make moves.

1.2 Move class

Each `Game` must define a `Move`. `Move` is an abstract class, and is conceptually attached to an implementation of `Game`. Moves are the only way to change the game state.

`Move` is an intentionally barebone abstract class. Subclasses to `Move` generally contain some information, so that when the `Move` is made, the `Game` knows what to do with it. For example, a `TictacMove` for the tic-tac-toe game consists of a `spot`; when the move is made, the corresponding spot is made on the board.

1.3 Player class

Notice that the notion of a player is completely separate from the game. The `Player` abstract class exists when we want separate entities to make their own moves. Players do the following:

1. Takes in a `Game` object, which has the current state of a game being played.
2. Returns a `Move` object, using its `findBest` function.

The `PlayOneGame` shows an example of how `Player`s can be used with a `Game`. The game is simply initialized; then, depending on whose turn it is, the appropriate player makes its best move until the game is over. This is all done in the `go` function.

Note that the only way to evaluate moves is to make them. Thus, a player can hypothesize about future moves by copying the current game, and making moves based on that. This allows the player to see outcomes without actually changing the game, and allows the player to backtrack after evaluating ways the game can end up.

1.4 How to actually play a game

We can play games using the `PlayGame` class as follows:

```
java PlayGame Easy
java PlayGame Nim human human
java PlayGame Tictactoe human cs310.Backtrack
```

The third argument is the game type. The fourth and fifth arguments are the players we want to play the game; if not specified, they will default to human and backtrack, which you will need to implement in an assignment.

2 Factories

2.1 Game factory

Once we have a type of `Game` instantiated, we can playing it using just the methods in the `Game` abstract class. Polymorphism calls the appropriate method implementation based on the actual class.

But notice that we give a string `gameName`, and need to use that to instantiate `Game` that we want. There are many games in the package. One possible way is to use a large switch case or if/else block to instantiate the correct `Game` in the `PlayGame` class. For example:

```
Game game;
if (gameName.equals("Nim"))
  game = new Nim();
else if (gameName.equals("Tictactoe"))
  game = new Tictactoe();
else if (gameName.equals("Easy"))
  game = new Easy();
...
```

This can get large fast. Alternatively, we can use the `Class` library to obtain the class from `className`, then instantiate it.

```
Class c = Class.forName(className);
Game game = c.newInstance();
```

This is cleaner and more elegant. Of course, one should take care to only instantiate classes which are games.

In the `playGame` method of the `PlayGame` class, we can see that class instantiation is handled by the `GameFactory` class:

```
GameFactory.create(gameName);
```

Looking in the `GameFactory` class, we can see that all it does is make sure that `gameName` refers to a valid game, find the correct game, and instantiate it. We call `GameFactory` a *factory* because its sole purpose is to instantiate *other* classes (contrast this with a constructor).

2.2 Player factory

Just as we turn command line arguments into `Game` objects using the `GameFactory`, we can instantiate the correct type of `Player` with the `PlayerFactory`. We can see this happen in the `playGame` again, though it only uses `PlayerFactory` to instantiate the computer player.

This may seem like overkill since we only have one type of computer player (`Backtrack`, which you will implement). But it would be nice for future work; if you create other computer players in the future, they will work in `PlayGame` with no modification to any code. The user just needs to pass in the `playerString` for your new computer player class.

We could probably have had the `PlayerFactory` also create human players.

3 Observer Pattern

Notice that when we play a game, we can see a textual representation of it (try `java PlayGame Easy`). Note that in the actual game class, there is no reference

to a GUI, e.g. no print line statements in the `make` method. There is a clean separation between the `Game` and its text representation.

This separation is achieved by using observers. The GUI is an observer for the game – when the game changes, it notifies the GUI, which then prints out the current game state . Notice:

- In `PlayOneGame`, we call `makeObservable` rather than `make`.
- The only difference with `makeObservable` is that it notifies observers.
- `Game` implements `Observable`