CSC207H: Assignment 1

The Overall Task

This assignment is about a two-person dice game called *Flip* explained on Cheapass Games. You will write two independent Java classes that each implement the interface `GameState` that we have provided for you. Once your code is complete, you can play your game using the text-based user interface in `FlipRunner.java`, along with one of your two `GameState` implementations.

The Starter Code

Your two `GameState` implementations are called `BagGameState` and `SetGameState`. We have started each of them for you and added them to the `a1` directory of your repository. The interface and runner classes mentioned above are also in this directory. **You must not change either `GameState.java` or `FlipRunner.java`.** We will use our original versions to compile and test your completed code.

To understand the rest of this handout, you will first have to go look up the game rules and play until you understand them.

What is a `GameState`?

The *state* of a game is all the information that would be needed to continue the game from that point forward. It doesn't usually have to include all the details of past turns, just the current situation.

We need to know the state of the game when a player's turn begins, and in Flip that point in time needs to be thought through carefully. Remember that there are three actions players may perform: *flip*, *play* and *take*. If player A chooses to flip a die, then obviously player B's turn begins as soon as A has finished flipping. However, if A chooses to "instruct" B to play one of B's dice into the middle, then even though the official game description sounds as if B's turn begins with the play, B actually has no choice about it, and it makes sense to consider the play as part of A's move. The take that may follow the play, however, is B's choice, so B's move begins with the take. (And if B has no dice left, and cannot take any, then A wins the game.)

And what *is* the game state in Flip? Obviously it must include the dice belonging to each player and the dice in the middle. However, because of the stalemate rule, the state also has to include the information about which of those dice have been flipped since the player last did a play. In our discussions, we will call the dice that the player is not currently allowed to flip *unflippable*. The dice that can legally be flipped are called *flippable*.

Now consider what happens after player A chooses to play. When B's turn begins, a take is possible, but only up to the point where the sum of the taken dice is less than the die just played. Therefore, B needs to know — by examining the game state — which die was just played, so the state must include that information.

The game state also must show which player plays next. For simplicity, we choose to designate the player about to move or in the process of moving as player 1 (the other being player 2). At the end of the move, the `GameState` object's `switchPlayers` method swaps the identity of the players. This simplifies the logic for most of the methods and also simplifies the description of the state.

The `GameState` class also includes checking methods to be called before a move. For example, the `checkFlip` method checks if player 1 is allowed to make the specified flip.

Representing the game state as a string

Because we have already written unit tests that examine the current `GameState` by using its string representation, your implementation must provide a `toString` that builds the representation in this order.

- the flippable dice of player 1 (if any) in increasing order
- a plus sign (+)
- the unflippable dice of player 1 (if any) in increasing order
- a bar (|)
- the dice in the centre (not including the one that was just played) in increasing order
- a plus sign (+)
- the die that was just played into the centre on the last turn (if any)
- a bar (|)
- the flippable dice of player 2 (if any) in increasing order
- a plus sign (+)
- the unflippable dice of player 2 (if any) in increasing order

For example, suppose the string representation of the state is
`45+6|144+4|446+`
Then player 1 (who is about to make a move) has flippable dice 4 and 5, and an unflippable 6. Player 2 has two flippable 4s and a flippable 6, and no unflippable dice. In the middle, we see that a 4 has just been played, and that explains why player 2 has no unflippable dice. In the middle are also a 1 and two 4s, so if player 1 chooses, he or she may take the 1 from the middle before doing a play or flip.

Two Different Class Designs

Both `SetGameState` and `BagGameState` implement the same public interface but they use a very different internal data structure. `SetGameState` will model the dice for each player using a Java `Set` of `Die` objects. The `Die` class is provided in your repository and you must not change it.

The `BagGameState` does not model each die with its own object. Instead it models a collection of dice by storing only the number of dice with each face value. The class gets its name **`Bag`**`GameState` because we sometimes talk about having a bag of objects that are all effectively the same. All you need to know to model the bag is how many objects are in the bag. Since each player has flippable and unflippable dice, each player will have two bags. To model a bag, we have provided the class `Dice` in your repository. We might have called the class `BagOfDice` but that name just seemed too cumbersome.

Testing Your Code

You will also find three test files in your repository. `GameStateTest` is the abstract parent class that contains the JUnit test cases that you will use for both your implementations. `BagGameStateTest` and `SetGameStateTest` are the concrete child classes that instantiate the fixtures to be tested and inherit the shared test cases.

`GameStateTest` contains 24 JUnit tests that only partially test a `SetGameState` implementation. At the very least you should use the two child classes to test your code, but that alone is not enough to be sure that your implementations are correct. In particular, some of the interesting test cases for the program have been intentionally left out. We have 10-15 more test cases that we will add to the file when we use it to test your submission. *HINT:* As you learn the Flip rules, write new test cases for your local copy of `TestGameState` and use them to test your code as your develop it.

Flip Rules

The rules posted at Cheapass Games are the definitive Flip rules for this assignment. If you are confused about a rule, you might be able to find a test-case that addresses your confusion. If after reading the rules through carefully (at least three times), you still think there is an ambiguity, you should post a message to the course bulletin board or visit your instructors' office hours. In your message, you should outline the two or more interpretations of the abiguity. For example you could say, "When the game is in such and such a state, and player 1 does x, the possible outcomes could be either y or z and both seem legal according to the game instructions and the starter code."

Your Final Submisssion

Submit your assignment by committing your completed `SetGameState.java` and `BagGameState.java` to your repository. Each of these classes, and all the others we have provided, should be inside a package `flip`. This package must be inside the `a1` directory of your repository. You will find advice on working with packages in the Hints and tips page.

It is fine to have extra files in `a1`, and we don't mind if you accidentally change the other Java files in your repository. Just make sure your solution works with the original starter code (including the JUnit test files) because that is what we will be using in our testing.

Before you consider yourself finished this assignment, you should check that you have submitted everything correctly and that it all works on (`cslinux`. To do this, you want to check out a fresh copy of your repository. (Of course you already have a copy of your repository checked out somewhere, but even if that somewhere is also on your school account, you want to check out a fresh copy.) Log in on the school machine, either by going to the lab itself, or by connecting with an ssh program such as PuTTy. Use the Unix command-line tools inside a terminal window to make a completely new directory using `mkdir`. Change to the new directory (with `cd`), and do a checkout of your repository:

```
svn co https://cslinux.utm.utoronto.ca/svn/csc207h/YOUR_UTORid
```

Navigate to the subdirectory containing the `flip` package source code and confirm that your files are inside. Next compile your code, move to the parent directory, and run your code from the command-line. To include the JUnit files in your compilation you will need to specify the classpath option to java as follows.

```
javac -cp ".:/usr/share/java/junit.jar" flip/*.java
```

You can also run the JUnit tests from the command-line with the following commands.

```
java -cp ".:/usr/share/java/junit.jar" junit.textui.TestRunner BagGameStateTest
java -cp ".:/usr/share/java/junit.jar" junit.textui.TestRunner SetGameStateTest
```

Some Advice About Style and Design

As well as marking your code for correctness, we will be looking at style and design. The expectations for good Java style include following the standard naming conventions and using a consistent style for braces, indentation and other whitespace. We are not requiring full Javadoc comments, and we have provided external method comments in the interface, but we will be marking your internal comments. Most of the design decisions are made for you already, but you still have the freedom to use private helper methods as appropriate.

Last change: $Date: 2008-09-25 14:32:52 -0400 (Thu, 25 Sep 2008) $