PlasterClash™

Taylor Burdette

1 Problem Description

Everyone knows the two things that CS 1331 TAs enjoy more than anything else are card games and garden gnomes. When they're not helping you learn Object-Oriented Programming, they're playing games against each other to keep their minds sharp and casting garden gnomes out of plaster to hone their creative skills.

Of course, for years, they've looked for a game that combines their two loves, but no one in the industry seems to understand their love for gnomes! That's where you come in. We at PlasterMaster Games want to design a gnome card game, and we need your help! We've organized a framework, but we need you to fit the parts together.

1.1 WAIT

Before you dash in and start coding away, please, please read through this document in its entirety. Especially be sure to read Solution and Tips for its major tips on how to approach this homework. Okay, keep reading.

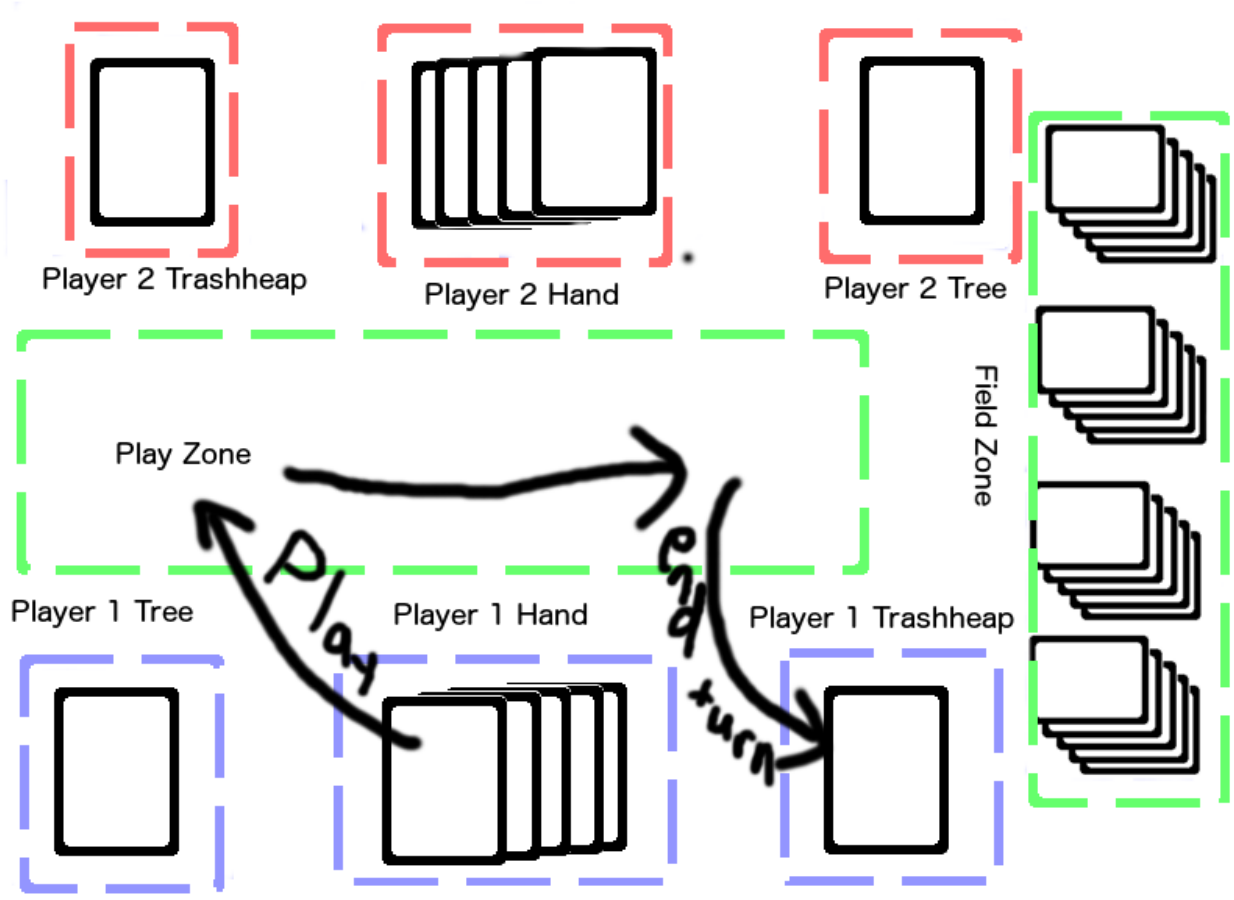
1.2 The Game

Here's our idea for PlasterCrash™:

1. Each player will have a deck of cards that they draw from called "The Tree", and they'll throw "Keebles" at gnomes to lead them into their traps by playing them from their "Hand".
2. Played cards will be removed from the Player's hand. Most cards, but not all, will then be played to the "Play" zone located in the PlasterClash game object. (Gnomes are not; thus, they are removed from the game when played.)
3. Players can use their Keebles to buy better lures, special traps, or even capture Gnomes from "The Field". Those will be added to that player's Trashheap for future use. We've included a couple of example cards, but you should submit at least one custom card you made up. Feel free to have fun with it.
4. No matter what, you can always choose to use your Keebles to acquire "Garden Bread", a superconcentrated form of Keebles. Each Garden Bread is worth two Keebles.
5. Each gnome captured will alert the rest that the garden is being compromised, and a point will be added to the "Conspiracy Meter". When the Conspiracy Meter has 10 points in it and the current player ends their turn, the player with the most gnomes in their Tree will win!
6. Lastly, during their turn, a player can choose to shoot their Gnomes out of their PlasterBlaster™, trashing their Gnome to make the other player randomly discard one of their own cards. The Gnome is then **not** added to the playing zone, removing them from the game entirely. Shooting the Gnome is done by playing the card, similar to playing a Keeble or other card.

Players start with 9 Keebles and 1 Gnome in their Tree. This Gnome does NOT increment the Conspiracy Meter. Each player will draw five cards, and turns will alternate between players.

Below is an example of the game's playing area for two players. Your game will have four players. You can see what happens when you play a card, and what happens at the end of your turn as well.



1.3 The Turn Order

A basic turn will look like:

1. Play Stage: The player plays any Cards (Keebles, Garden Bread, Gnomes, etc) in their Hand if they wish. Depending on the card, some action may result.
2. Buy Stage: If the player played any Keebles or Garden Bread, they have temporary "keebles" which they can spend to acquire new cards. Any purchased cards go into the player's "Trashheap".
   For example, a player may play two Keeble cards and buy a Garden Bread Card - effectively combining two Keeble cards into one and saving space in a future hand. They don't lose the played Keebles or Garden Breads cards, they are still in the Play Zone.
3. The remainder of the player's hand is then discarded into the trash heap, along with the cards in the Play Zone.
4. The current player draws 5 cards from their "Tree". This is the player's new hand. If the Tree does not have enough cards to fill the hand, the contents of the trash heap are shuffled and transferred to the Tree.
5. Finally, play moves to the next player.

1.4 The Design

1. **`Card`**: We'll start with the cards themselves.
   (a) Each card will have a name, a description, a block of flavor text, whether it counts as a Gnome for scoring, and a Keeble cost.
   (b) Each instance variable above will require accessor methods. `Card` attributes should be immutable.
   (c) Cards should be able to be played, and should have the default behavior of removing themselves from the current player's hand.
   (d) Cards should also have a method that moves the card into the playing zone. Not all cards will necessarily need this (i.e. Gnomes).
   (e) Card objects should act properly as elements of collections (including hashed collections).
   (f) Card objects are considered equal if they have the same name.
   (g) Cards should display their attributes nicely to the user.
   (h) Each Card should have more detail than has been mentioned above - i.e, it should not be possible to have a Card with no other specifics. (Hint: there's a single Java keyword that encapsulates this idea)
2. **`Zone`**: There are several Zones throughout the gameplay in which cards can reside. A zone will need a backing data structure to organize its cards.
   (a) Zones should have a backing list of cards.
   (b) A Zone is ordered - think of a Zone like a deck or stack of cards: the position of a card in a Zone matters.
   (c) Zones should have methods to display, add, and remove cards from their backing lists.
   (d) Zones should be able to randomize their contents, effectively "shuffling" the deck.
   (e) Zones should be iterable over their Card contents.
   (f) Zones should be able to easily transfer all their contents to another Zone.
   (g) A Zone should be able to count up its cards as well as specifically the number of Gnomes.
3. **`Field`**: A set of cards the player can purchase.
   (a) A Field should have a backing map that maps Cards to be purchased to the number remaining.
   (b) A Field's contents should be set at the start of the game. New cards should not be added mid-game.
   (c) When a card is purchased, its mapped value should decrease. If a card's mapped value is 0, there are no more left in the Field and no more can be purchased.
   (d) A Field should be able to return its card contents as an ordered list.
   (e) A card should not be shown in the list of purchasable cards if there are no more left.
4. **`Tree`**: A special `Zone` that comes with cards at the start.
   (a) A Tree should be a Zone that starts off with a Gnome Card and 9 Keeble Cards in it.
5. **`Gnome`**: A `Card` worth gnome/victory points.
   (a) A Gnome should be worth points at the end of the game.
   (b) A Gnome, when played, should cause every other player to randomly discard a card.
   (c) Gnomes, when played, are removed from the player's hand but not added the playing zone, removing them from the game entirely.

   Hint: When setting the Gnome's cost, choose a relatively low value - if you set it too high, it will either take forever for the player to accumulate enough Garden Bread to pay for it, or it will be impossible. (5 cards in the hand times 2 keebles per Garden Bread equals a maximum of 10 for Gnome's cost).

6. **`Custom Card`**: At least one custom `Card` that you will design and write.
   (a) The custom card can be named whatever you like and do whatever you like. Feel free to have fun!
   (b) The custom card, when played, should do something unique to affect the game state.

You have been given the following classes already. Do **NOT** modify any of these classes.

1. **`PlasterClash`**: The game driver itself.
2. **`Player`**: A player for the game.
3. **`Keeble`**: A Keeble Card that increases a player's keeble resources by one when played.
4. **`Garden Bread`**: A Garden Bread Card that is worth two keebles when played.

2 Solution and Tips

2.1 Tips

This homework provides general direction where specific implementation details can be found in the provided files. Be sure to generate the javadocs for the provided code (or just read the code directly) to look at all the provided methods.

Our code assumes you will name methods in a certain way. You'll find this out pretty quickly when you first try to compile and you get several errors. If you haven't already, learn how to read these errors. Navigate to the location of the error in the provided code. Read the javadocs for that method, and discern its purpose. From that and the error, infer what method you should write in your code. This isn't as hard as it sounds: it just means you may be naming a method "randomize" where we expect "shuffle".

In summary: READ THE JAVADOCS, and sometimes our provided code, too.

2.2 Sample Output

When you run PlasterClash, you should see an output like this:

```
========================================
           Player 0's Turn:
========================================
Cards remaining in Tree: 5
Total Gnomes: 1

-------------------------
      Play Stage
-------------------------
Hand:
0: Gnome (4)
1: Keeble (1)
2: Keeble (1)
3: Keeble (1)
4: Keeble (1)

Enter the card # you would like to play, "end" to end, or "all" to play all.
>
```

Then buying cards:

```
-------------------------
       Buy Stage
-------------------------
Number of keebles: 4
Available cards:
0: Gnome (4)
1: Garden Bread (2)

Enter what card you would like to buy, or "end" to end turn.
> 0

<<<<<<<<<<<<<<<<<<<<>>>>>>>>>>>>>>>>>>>>
| Gnome captured! Conspiracy meter:  1 |
<<<<<<<<<<<<<<<<<<<<>>>>>>>>>>>>>>>>>>>>

Discarding hand...
```

Then with a winner:

```
=========================
|    WINNER: Player 0   |
=========================

Number of gnomes: 6

Thanks for playing!
```

Do not modify the provided files at **all**. Those files will define the behavior of the other classes.

It is absolutely vital that your submission compiles with the provided source files. Non-compiling solutions will be given a zero; no exceptions.

3 Javadocs

For this assignment you will be commenting your code with Javadocs. Javadocs are a clean and useful way to document your code's functionality. For more information on what Javadocs are and why they are awesome, the online documentation for them is very detailed and helpful.

You can generate the javadocs for your code using the command below, which will put all the files into a folder called javadoc:

```
$ javadoc *.java -d javadoc
```

The relevant tags that you need to have are `@author`, `@version`, `@param`, and `@return`. Here is an example of a properly Javadoc'd class:

```
import java.util.Scanner;

/**
 * This class represents a Dog object.
 * @author George P. Burdell
 * @version 13.31
 */
public class Dog {

    /**
     * Creates an awesome dog (NOT a dawg!)
     */
    public Dog(){
        ...
    }
```

```
    /**
     * This method takes in two ints and returns their sum
     * @param a first number
     * @param b second number
     * @return sum of a and b
     */
    public int add(int a, int b){
        ...
    }
}
```

Take note of a few things:

1. Javadocs are begun with `/**` and ended with `*/`.
2. Every class you write must be Javadoc'd and the `@author` and `@version` tag included. The comments for a class start with a brief description of the role of the class in your program.
3. Every non-private method you write must be Javadoc'd and the `@param` tag included for every method parameter. The format for an `@param` tag is `@param <name of parameter as written in method header> <description of parameter>`. If the method has a non-void return type, include the `@return` tag which should have a simple description of what the method returns, semantically.

3.1 Javadoc and Checkstyle

You can use the Checkstyle jar mentioned in the following section to test your javadocs for completeness. Simply add -j to the checkstyle command, like this:

```
$ java -jar checkstyle-6.2.1.jar -j *.java
Audit done. Errors (potential points off):
0
```

4 Checkstyle

You must run checkstyle on your submission. The checkstyle cap for this assignment is **100** points.
Review the Style Guide and download the Checkstyle jar. Run Checkstyle on your code like so:

```
$ java -jar checkstyle-6.2.1.jar *.java
Audit done. Errors (potential points off):
0
```

The message above means there were no Checkstyle errors. If you had any errors, they would show up above this message, and the number at the end would be the points we would take off.

The Java source files we provide contain no Checkstyle errors. For this assignment, there will be a maximum of **100** points lost due to Checkstyle errors (1 point per error). In future homeworks we will be increasing this cap, so get into the habit of fixing these style errors early!

5 Turn-in Procedure

Submit all of the Java source files you modified and resources your program requires to run to T-Square. Do not submit any compiled bytecode (`.class` files) or the Checkstyle jar file. When you're ready, double-check that you have submitted and not just saved a draft.

Please remember to run your code through Checkstyle!

Verify the Success of Your Submission to T-Square
Practice safe submission! Verify that your HW files were truly submitted correctly, the upload was successful, and that the files compile and run. It is solely your responsibility to turn in your homework and practice this safe submission safeguard.

1. After uploading the files to T-Square you should receive an email from T-Square listing the names of the files that were uploaded and received. If you do not get the confirmation email almost immediately, something is wrong with your HW submission and/or your email. Even receiving the email does not guarantee that you turned in exactly what you intended.
2. After submitting the files to T-Square, return to the Assignment menu option and this homework. It should show the submitted files.
3. Download copies of your submitted files from the T-Square Assignment page placing them in a new folder.
4. Recompile and test those exact files.
5. This helps guard against a few things.
   (a) It helps insure that you turn in the correct files.
   (b) It helps you realize if you omit a file or files. [1] (If you do discover that you omitted a file, submit all of your files again, not just the missing one.)
   (c) Helps find last minute causes of files not compiling and/or running.

[1] Missing files will not be given any credit, and non-compiling homework solutions will receive few to zero points. Also recall that late homework will not be accepted regardless of excuse. Treat the due date with respect. The real due date is midnight. Do not wait until the last minute!