Shawn S Hillyer

CS 162 – Lab 3

Date: 10/12/2015

Implementing Inheritance

Readme.txt

There are several make targets:

default: 'make' will create the DiceWar program / game.

original: 'make original' will create the 'original' testing client made by the other student using the inheritance based class hierarchy

mysim: 'make mysim' will create the DiceSimulator program I created for Lab1 using the inheritance based class hierarchy. No changes were made to DiceSimulator.cpp code for this assignment.

Getting loadedDie and implement.cpp to work with no other mods

I first decided to test the class and make sure it compiled and performed as expected without any modifications. I suspected that the algorithm in their loaded die method was not performing the way the user expected, but I wasn't sure.

First, I modified my makefile to use the implement.cpp file that was provided and set up a new target which used that as the entry point instead of my original main() definition. The first problem was that they named their classes differently, as well, so I decided to rename their files using the standard I always see (CapitalCase instead of CamelCase). I also changed the #includes in their implementation to match the updated file names.

The next problem I ran into with the implementation file is that they defined the Die class using a setDie method that did not exist in my original die class. Since I don't have access to the original Die class, I'm not sure what their method actually did. I'm assuming this was supposed to be a set_num_sides method.

Also, all of the methods in the LoadedDie files provided used a different coding style than the class guidelines.

So I decided to take a second to just rename all of the methods to be consistent using a simple find-and-replace search. Also they named variables using capital case in some places and all lower cases in others. It was basically a bloody mess.

I trashed the method getSides because it was never used in any of the code. I also cleaned up the LoadedDie calls by deleting the scope resolution in front of the calls to the methods inside the class because the function calls were all on the instance of the class. For example, they called LoadedDie::getsides inside of the LoadedDie class. Instead, I just replaced it with a direct data member access. For example:

```
int result = (rand() % LoadedDie::getsides())+1,
```

became

```
int result = (rand() % num_sides) + 1,
```

This was almost enough to fix the program. However, for some reason the creator of the implement.cpp made their set_num_sides (original called setLoadedDie etc.) public calls. I felt that the die should only call this function from within the constructor in my Die class, so it was incompatible. So I opted to just change the calls to call the constructor on the die after called by the user.

Finally I was able to run their implement program. The problem with the program is that it has no loops, so every time I Run the program, I have to type in the file name again, manually select the number of sides, and manually compare the results of the two rolls. I tested a few basic values though to see the differences using their program just to prove that their class worked in their own implementation.

Results:

Using a 6 sided die on 100 rolls: Fair Die average roll 3.7, Loaded Die was 3.87

Using a 6 sided die on 1000 rolls: Fair die average roll 3.477, Loaded die was 3.973.

Using a 10-sided die on 10,000 rolls: Fair die average 5.5338, Loaded Die 6.4797

Using a 50-sided die on 50,000 rolls: Fair die average 25.5665, Loaded Die 31.529

I then used my ”make” default target, which was currently building my DiceSimulator.cpp output, and the test program compiled perfectly with the changes I had already made so that the LoadedDie class was compatible with what I had been using. My auto_test() runs 1000 samples on a 100-sided die and gets this result each time:

- Total of all Loaded Die rolls: 63635
- Average of all Loaded Die rolls: 63.635
- The Fair Die total: 49657
- Fair die - Average of 1000 rolls: 49.657
- The loaded die total: 63635
- Loaded die - Average of 1000 rolls: 63.635
- The loaded die averages a roll 13.978 higher than the fair die.

My biggest problem with the loaded die is that it never rolls a 1 on any dice because it always returns the roll incremented by 1 if the die rolls a 1. Otherwise, it seems to return a result, although it is very, very minimally better than the fair die.

I opted to get my game working properly before going on to fix the loaded die itself. I first went to work making the Loaded Die inherit from the Die.

Implementing Inheritance

Converting the re-styled LoadedDie to use inheritance was easy. I pulled up my header files for each class and deleted the duplicated data member, num_sides. Of course, I told LoadedDie to inherit from Die.

I converted the constructor to work with my simple Die constructor next. This actually just involved deleting the first of the two constructors written by the original author of this LoadedDie class and added the Base class constructor designation, and finally giving it a default value of 6 to match the Die class. I also deleted the set_num_sides method because it did precisely the same thing as in the original Die class. Now the sides would be set just by using the Base class constructor only.

All this left was the roll method and the constructor. I deleted reference to the private specifiers leaving a simple header consisting of just the constructor and roll() declaration and the definitions (implementation) file with just that same methods. Also, I changed the data member access to protected to give access to the data members to any inherited class (in this case, LoadedDie only).

Before making further changes, I saved this work and ran “make clean” and remade the implementation and the DiceSimulator versions of the test client for the Die classes. Everything was still working fine, so I decided to fix the “skew” of the loaded die to be more pronounced.

I manually tested Die of 6, 10, and 20 sides. There is only about a 10% increase in the average value, so I decided to just bump this up a bit by making it a 50% chance that a roll of less than half the max would just tip over and hit a max. This means that the die can still get a value in the bottom range, but about 1/4th of the time they roll a max value. Based on the original code, this seemed to be the original intent of the author, so I kept the spirit of their algorithm.

To test my fully implemented game, I needed to test a few use cases:

3 different n-sided die (including smallest I allow, 4) with 100 samples for each configuration of Loaded Die (Neither player having a loaded, both, and one or the other). I expected roughly 50/50 win ratio for the cases where they both have a loaded die or both had a fair die. And I expected a win ratio much higher for the case of a player 1 OR player 2 having a loaded die.

I decided to test 3 different n-sided die on 100 samples: 4 sided, 8 sided die, 50 sided. I did a test for all 4 cases of loaded configurations on all dice sides mentioned. The results are on the next page.

Overall I’m extremely satisfied that the Die classes work and that the correct roll is being used for each of the two Die stored in the arrays. I also ran a test by modifying the LoadedDie to always return a highest value possible and used that to confirm which Die was really being rolled for each player.

Sides	**Rounds**	**P1 Loaded**	**P2 Loaded**	**Result**	**P1:P2 score Ratio**
4	100	n	n	Player One scored 31 points! Player Two scored 32 points! The victor is... Player Two!	97%

4	100	y	n	Player One scored 53 points! Player Two scored 28 points! The victor is... Player One!	189%
4	100	n	y	Player One scored 23 points! Player Two scored 54 points! The victor is... Player Two!	43%
4	100	y	y	Player One scored 41 points! Player Two scored 30 points! The victor is... Player One!	137%
8	100	n	n	Player One scored 46 points! Player Two scored 40 points! The victor is... Player One!	115%
8	100	y	n	Player One scored 58 points! Player Two scored 29 points! The victor is... Player One!	200%
8	100	n	y	Player One scored 21 points! Player Two scored 61 points! The victor is... Player Two!	34%
8	100	y	y	Player One scored 39 points! Player Two scored 43 points! The victor is... Player Two!	91%
50	100	n	n	Player One scored 44 points! Player Two scored 52 points! The victor is... Player Two!	85%
50	100	y	n	Player One scored 66 points! Player Two scored 31 points! The victor is... Player One!	213%
50	100	n	y	Player One scored 36 points! Player Two scored 63 points! The victor is... Player Two!	57%
50	100	y	y	Player One scored 44 points! Player Two scored 48 points! The victor is... Player Two!	92%