Shawn S Hillyer
CS 162

10/02/2015

Lab 2 – A Friendly Game of WAR (With Dice)

Analysis of this project began with a review of the specifications. I printed and highlighted in yellow the key nouns and put a green box around as many of the obvious "actions" that would act on data that I saw in the specifications. This is attached as an additional exhibit with this write-up. I then used this to sort the requirements into each class as I went through the paragraph, writing down 3 main areas: Game class, Die classes, and overall program requirements. I noted during this process that the data being acted on in the third section very well could be part of the Game class since the required data is encapsulated there.

I also considered a more granular version of the abstractions for my own satisifaction, and I feel it would end up being more extensible. For example, I could invent new types of die and give the players more than one die, give players names, etc., by abstracting out a Player class. *However*, the specifications don't require we go to this level of complexity, so I present a solution based only on creating only three classes per the assignment specifications.

Requirements

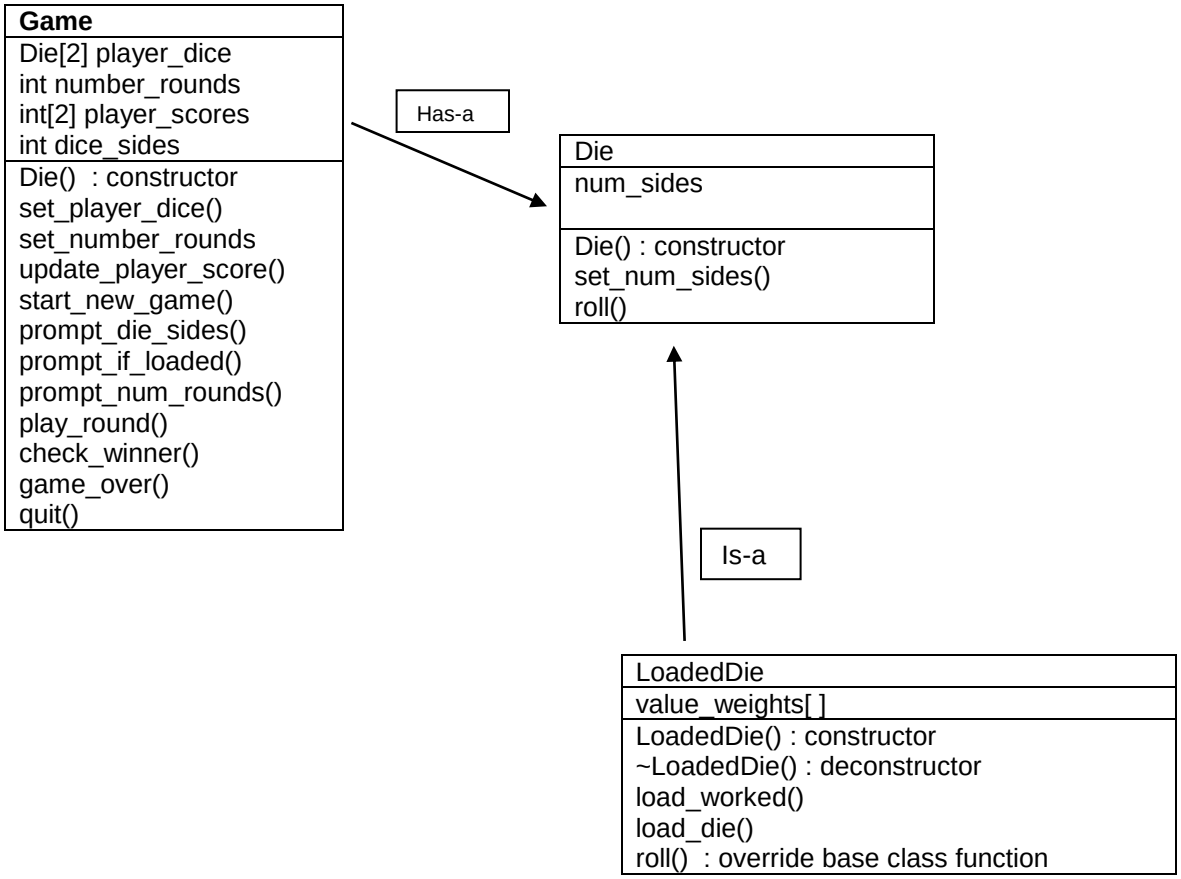
Specifications indicate this program must meet the requirements re-stated below. Note that some requirements are to have a specific class, others are for specific data members, and others are for certain overall program behavior.

- A Game class
  - Game tracks the type of die for each player
  - Game will have 2 players and 1 user
  - Game will track the number of rounds to play
  - Game will somehow maintain the score
  - Game will utilize a has-a (composition) relationship with the Die class(es) in some way
  - Game will play a round of the game which causes the die for each player to be rolled
  - During each round of the game, Game will determine which player won based on the value of their roll
- Die classes
  - The members of the Die classes which are in common will be encapsulated in a super-class ("parent class")
  - There will be an inheritance relationship between the Die classes
  - There will be a loaded die
  - Die must track its number of sides
  - Die must be able to update its number of sides to match user input
- Program as a whole
  - Collect the number of sides on the dice
  - Determine which player(s), if any, are using loaded die
  - Collect the number of rounds to play
  - *Program must print the winner (whoever won the most rounds)*
    - The four above items could be encapsulated into an appropriate class – probably the game class – because it works on certain data members of the Game class and simply acts on those data members (or uses them to display information)

Note that in Game, each element of Die player_die can be of type Die, the superclass, or LoadedDie, the inherited class. Its implied that the Game will set the Die in each element to the appropriate.

Game class has-a relationship is a simple array of 2 Die class objects. Because the inherited class can be assigned to a superclass variable, and there is only one public interface (namely, .roll()), this seems okay. I don't know if this is true in all programming languages, but it makes sense that it should be true (because any inherited object will have all of the same methods as its parent superclass(es)).

Here are the three classes prior to drawing the relationship lines and indicators.



The main program outline will be simple. The goal was to make all data and functionality fully encapsulated in objects. The main will simply instantiate the Game and game will call start_new_game(). Game will then call each of its functions to execute each step of the game and recursively call start_new_game() to re-initialize the data members during game_over(). If during game_over() the user decides to quit, the function will call quit, which will cause Game to fall out of scope and the program will terminate.

The Die class will be instantiated by the Game class, and the LoadedDie will as well, if appropriate. This is determined by the type of player_dice that is determined during prompt_if_loaded() for each player.

Class Hierarchy

Below is the class hierarchy:

Testing Plan – Draft

Incremental testing - I would test each class as I added functionality and data. I would start with the Die Class, then LoadedDie, and finally work through Game class

	Arguments	Expected Outcome	Actual Outcome
Die()	None	num_sides == 6	
Die()	-1	num_sides == 6	
Die()	6	num_sides == 6	
Die()	10	num_sides == 10	

	Arguments	Expected Outcome	Actual Outcome
Die()::set_num_sides()	-1	num_sides == 4	
Die()::set_num_sides()	4	num_sides == 4	
Die()::set_num_sides()	10	num_sides == 10	

**A Die cannot have less than 4 sides for our purposes

(no args)	Driver function	Expected Outcome
Die::roll()	report_min_max()	always returns value between 1 and num_sides
Die::roll()	report_distribution()	Should have even distribution over all possible values
Die::roll()	report_average()	Should return a value close to the median value

-Notes: report_min_max() would iterate large number of roll() calls (could easily do 100k+) and track min/max and report it

-report_distribution() would iterate large # of roll() calls and track the frequency of each value by incrementing a paralell array index (in which every element is initialized to zero). The index would be Die::.roll() - 1. When reporting the distribution, we expect to see a similar number of results for all values over a large spread (so in 600 rolls on a 6 sided dice, roughly 100 each), or sample_size/num_sides

The method of systematically analyzing the distribution is left as an exercise for a more detailed test plan

LoadedDie class

	Arguments	Expected Outcome	Actual Outcome
LoadedDie()	None	num_sides == 6	
LoadedDie()	-1	num_sides == 6	
LoadedDie()	6	num_sides == 6	
LoadedDie()	10	num_sides == 10	

These tests essentially test the constructor in the same way we tested the Die Constructor.

LoadedDie::roll() tests should use the same three functions, report_min_max and report_distribution as above.

We expect different results though - min_max should still be between 1 and num_sides, but our distribution should indicate a shift towards the higher end that is detectable.

Furthermore, the simpler test report_average() could be used to verify a higher average over a large number of .roll() calls. Combined with the distribution/frequency information, we could decide of the loaded die is going to have a higehr value than the expected average

LoadedDie::load_worked accepts an int between 1 and num_sides for the test die expect the load to return true at least some of the time. Call this in a loop.

Note: If it doesn't return true a perceptibly to the user or not is irrelevant. Specifications don't indicate how well the loaded die must work

The probability it works is actually controlled by the value_weights data and load_die() method

load_die()	0	All weight_values will be 0
load_die()	50	All weight_values will be 100 in the bottom two thirds of the range and 80 in the top third
load_die()	100	Bottom two thirds will be 50 and top third will be 40
load_die()	110	All weight_values will be 100 in the bottom two thirds of the range and 80 in the top third

Game() class

Game() constructor

No args

Can instantiate object and initialize all data members

Note: This will be implicitly tested during every test but must work to conduct further tests. The game will take no arguments in the current implementation design

Next, test that you can prompt for the input from user and set the Game variables needed to properly initialize a new game.

	Arguments	**Expected Result**
prompt_die_sides()	3	Re-prompts for a die 4 sides or greater
prompt_die_sides()	4	Sets dice_sides to 4
prompt_die_sides()	10	Sets dice_sides to 10

	Arguments	Expected Result
prompt_num_rounds()	0	Re-prompts for rounds >= 1
prompt_num_rounds()	1	Sets num_rounds to 1
prompt_num_rounds()	10	Sets num_rounds to 10

Use a stub to pass to set_number_rounds_stub()

	Input	Expected Result
prompt_if_loaded()	Y, Y	Both die in array will be a LoadedDie
prompt_if_loaded()	Y, N	Player 1 die Loaded, player 2 Die
prompt_if_loaded()	N, Y	Player 1 die Die, player 2 Loaded
prompt_if_loaded()	N, N	both die regular Die
Use a stub to call set_player_dice() until that function is implemented		

	Arguments	Expected Result
set_player_dice()	true, true	Both die in array will be a LoadedDie
set_player_dice()	true, false	Player 1 die Loaded, player 2 Die
set_player_dice()	false, true	Player 1 die Die, player 2 Loaded
set_player_dice()	false, false	both die regular Die

Use a driver to pass in two bools, bool player_1_loaded and player_2_loaded and use same testing criteria

	Arguments	Expected Result
set_number_rounds()	0	Re-prompts for rounds >= 1
set_number_rounds()	1	Sets num_rounds to 1
set_number_rounds()	10	Sets num_rounds to 10

Use a driver to pass in arguments set_number_rounds_test()

arg1 -> player, arg2 -> score	**Arguments**	**Expected Result**
update_player_score()	-1,-1	Throw exception
update_player_score()	0,-1	Throw exception
update_player_score()	0,0	Score array not updated
update_player_score()	1,0	Score array not updated
update_player_score()	0,1	player_scores[0] incremented 1
update_player_score()	1,1	player_scores[1] incremented 1

start_new_game(): Can test this using stubs for each function before they are made, or after making other methods. This method will initialize the data members to default values and call the prompt_ functions, then call play_round() number_rounds times. Once it is done it calls game_over(), which will print the results.

play_round()

Calls .roll() on every element of the player_dice[] array and then calls check_winner, printing the return values and relevant labels for each function call. If check_winner() returns -1, no score is updated. Otherwise, calls update_player_score() using the return value of check_winner().

check_winner()

Should return 0 or 1 depending on which player won the round, or -1 if neither won (draw). Return value is passed to update_player_score().

Pass in the roll from each player, player 1 being the first argument and player 2 being the second. If neither player wins, returns -1.

game_over()

Prints the overall winner based on the higher score in the array player_scores[]. Prompts user if they wish to play again. Recursively calls start_new_game() (in which this is nested) to effectively create a loop and re-initialize the board.

quit()

Causes the game (program) to terminate successfully.