Report for assignment 4

Erik Dackebro & Zacharias Erlandsson & Dan Hemgren &
Kenneth Runnman & Pojan Shahrivar

2018-02-28

Project

Name: cs56-games-poker
URL: https://github.com/ucsb-cs56-projects/cs56-games-poker

Complexity

By running lizard on the cloned files we get the following functions as the top ten highest complexity. For five of the we did a CCN count manually and received the same numbers as lizard after some work.

Ranking	Function	NLOC	CCN	Manual CCN
1.	CompareHands::isStraightFlush	92	39	
2.	OpponentAI::takeTurn	72	23	
3.	CompareHands::twoPairTie	44	14	11
4.	CompareHands::pairTie	39	13	
5.	CompareHands::straightTie	45	12	10
6.	CompareHands::compareHands	32	12	12
7.	PokerSinglePlayer::showWinnerAlert	29	12	12
8.	PokerGameGUI::betButtonHandler	30	9	
9.	CompareHands::calculateValue	20	9	
10.	CompareHands::getMostCommonSuit	33	9	

For the manual complexity measurement, we did not arrive at the same values as the tool that we used (Lizard). This is hard to explain, we have however observed that lizard seems to
When it comes to lines of code (NLOC) and Cyclomatic Complexity we can see that a higher number of lines of code somewhat entails a higher CCN, as the two functions with the highest CCN (StraightFlush() and takeTurn()) also are a bit lengthier than the other functions. However, we can also see counterexamples to this in our table, as the length of getMostCommonSuit() surpasses showWinnerAlert() with 4 lines of code while still having a cyclomatic complexity that is 3 less.
The purpose of the most complex function by far is to determine if the current poker hand contains a straight flush. The purpose of the second most complex function is to construct a very simple AI that decides if the computer-controlled avatar is going to fold or bet for the current hand. We can see a pattern where the majority of the functions in the top-10 of the complexity count often represent a bit more complex logical operations as opposed to purely functional programming such as UI and handling user interaction.
The program does not contain any handling or throwing of exceptions, so we can't measure any potential effects on the cyclomatic complexity of the program.
The documentation of the ten most complex functions is existent. However it could be more detailed.the branches that exist are not all documented, but there is rather a short summary of what the function does.

Documentation of 10 functions with high complexity

CompareHands::getMostCommonSuit()
getMostCommonSuit is a method that iterates over an ArrayList of cards (representing a hand in a game of poker) and returns the most common suit in that hand (i.e. spades, diamond, hearts or or clubs). The method has a cyclomatic complexity of 9 which isn't overly much, this is because of the many if-statements that checks the suit of the cards in the ArrayList. The method's cyclomatic complexity could probably be reduced with the use of some data structure holding information about the hand and the most common suit throughout the game but it isn't the first method that'd need to be looked at in an eventual refactoring.
The different branches in the function are taken depending on which cards we iterate over. In theory, all the branches of the first for-loop could be traversed if the hand contains at least 5 cards and at least 1 hearts card, 1 spades card, 1 diamonds card, 1 clubs card and 1 'disallowed' card. For the four later branches, only one can be traversed per run.

CompareHands::straightTie()
straightTie() is a method that calculates the better straight of two straights. The method has a cyclomatic complexity of 12. This method's relatively high complexity owes to the fact of the many clauses in the predicates of the if-statements. When you look at the method however, it doesn't appear overly complex and although the method could probably be made more simple using a different data structure for the cards, the method with the current implementation of hands is hard to improve any further. One minor change could be to, instead of having two predicates with four clauses each, make use of two for-loops with a single if-statement inside each one. That could possibly lower the cyclomatic complexity of the method.
The different branches of the function that are traversed depend on the two hands being compared; e.g. for all the three return branches to be traversed there needs to be test cases that tests a winning hand for player 1, a winning hand for player 2, as well as a draw. The other branches are always traversed if there's a straight in both player 1 and player 2's cards.

PokerSinglePlayer::showWinnerAlert()
The purpose of this function is to, at the end of the game, show a message box with information about who the winner was (i.e. the player, the opponent or a tie). The buttons of the message box also determines whether the player wants to play again or quit. If the player or opponent has less than 5 chips remaining, the game does not allow a new game to be started. In my opinion, the part showing whether to play a new game or not and the handling of this should be refactored into its own function, as this is entirely unrelated to showing who the winner is. This would cut down the complexity of the function to 6 and create a new function with complexity 7. This would however make the code more modular, easier to test and the functions would hold more relevant code.

CompareHands::isFlush()
This functions purpose is to find out whether a hand is a flush or not (a flush is all 5 cards of the same color). It iterates over the hand and increments a counter based on the color of the card. After it have iterated over the players hand, it checks whether any of the counters is ≥ 5. If yes, it returns true and false otherwise. The function could be refactored to lower CC from the current 9. I do personally believe this would be unnecessary and make the code harder to understand, since everything is related to the functionality of finding out whether the hand is a flush or not is in this function.

CompareHands::compareHands()
This method compares the hands of the two players and returns an integer: 1 if player 1 is the winner, 0 if player 2 is the winner and 2 if it is an absolute tie. First, the values of each hand are grabbed. If either is winner, the method returns with either a 0 or 1. If both hands have the same value, the method checks the contents of the hand of player 1 and then delegates comparing the two hands to a separate method. So, if both player 1 and player 2 have three-of-a-kind, the initial value will be the same (as both have the same type of hand) after which the winner is decided by returning threeOfAKindTie(). The complexity of this function is simply due to having 12 if-checks with corresponding return statements; one 'if'/'else if' /'else'-statement with a switch-statement of 10 cases embedded in the else-clause. Needless to say, only one return statement can be reached per call, resulting in a complexity of 12 (which correponds to the CNN count).

CompareHands::calculateValueToString()
Here, the hand of the player is mapped explicitly to a string representation of the hand. For instance, if the player holds a hand of three-of-a-kind, the function returns "Three of a Kind". This is done with a simple 'if'/'if else'/'else'-statement holding nine checks in total with corresponding return statements (one for each type of hand). Each check is delegated to some other method, so isThreeOfAKind() checks whether the hand actually holds three-of-a-kind. This results in a complexity of 9.

CompareHands::twoPairTie
This function decides the winner in the situation that both players have a two pair. For example if one player has a hand of two A's and two 3's and a 2, and the opponent has a hand of two K's and two Q's and a J. The winner of this round would be the player. To calculate the winner the function first sorts the players hands and then go through the hand to identify the pairs. When the pairs is identified it compares both players better pair, and if it is equal it will compare their lower pair. When those two are equal the fifth card should decide if it is a win, loss or tie. The complexity for this function can be explained by three components which are the loops. 2 for loops and one while loop with several if-statements in them. The manually counted cyclomatic complexity is 11 which is lower than lizards CCN (14). This is due to that I count with the formula $CCN = decisions - exitpoints + 2$, while lizard counts $CCN = decision + 1$, where the +1 is because it counts all returns as one exit point. The flow graph for this function can be found below.

CompareHands::pairTie
The purpose of this function is to determine which player has the highest pair of cards. In the case where two players have the same type of pair, the highest kicker has to be determined.
The complexity of this function is partitioned into three parts, first the pairs have to be extracted, which due to the design of the card deck involves some looping and use of if-statements that introduce branching. Secondly we have branching where players might have a higher or lower pair or of equal value. Lastly the kicker card has to be determined, this further involves looping with if-statements that branch, much like extraction of pairs. Finally there is the case where both players have pairs and kickers of equal value; Most of the time, most of the code is not reached. As the probability of an event decreases more branching is needed to cover all events.

CompareHands::isFullHouse
This function should return true if the cards form a full house and false in other cases. The complexity in the function is unnecessarily high as this is written by a somewhat novice programmer, excessive use of if statements. The branching in the function is due to the increase of a counter. The same functionality can be achieved with far fewer instructions and in linear time complexity.
To reduce complexity, counters of observed cards can be used. In the end, after iterating through the cards, we can check for the magic numbers 3 and 2, which form a full house.

PokerGameGui::betButtonHandler
This function handles input from the user when betting chips for the pot. It is first bounded by the an if-statement that make sure that the text string sent from the user is not empty. In that if-statements it checks for three cases: invalid bets, successful bets and if the bet exceeds the value of the player's chips. The outcome of the function is either that the playerprompt shows an error that the user have done or it shows how much the player has bet to the pot and bet is transactioned to the pot and the next turn of the game is started.

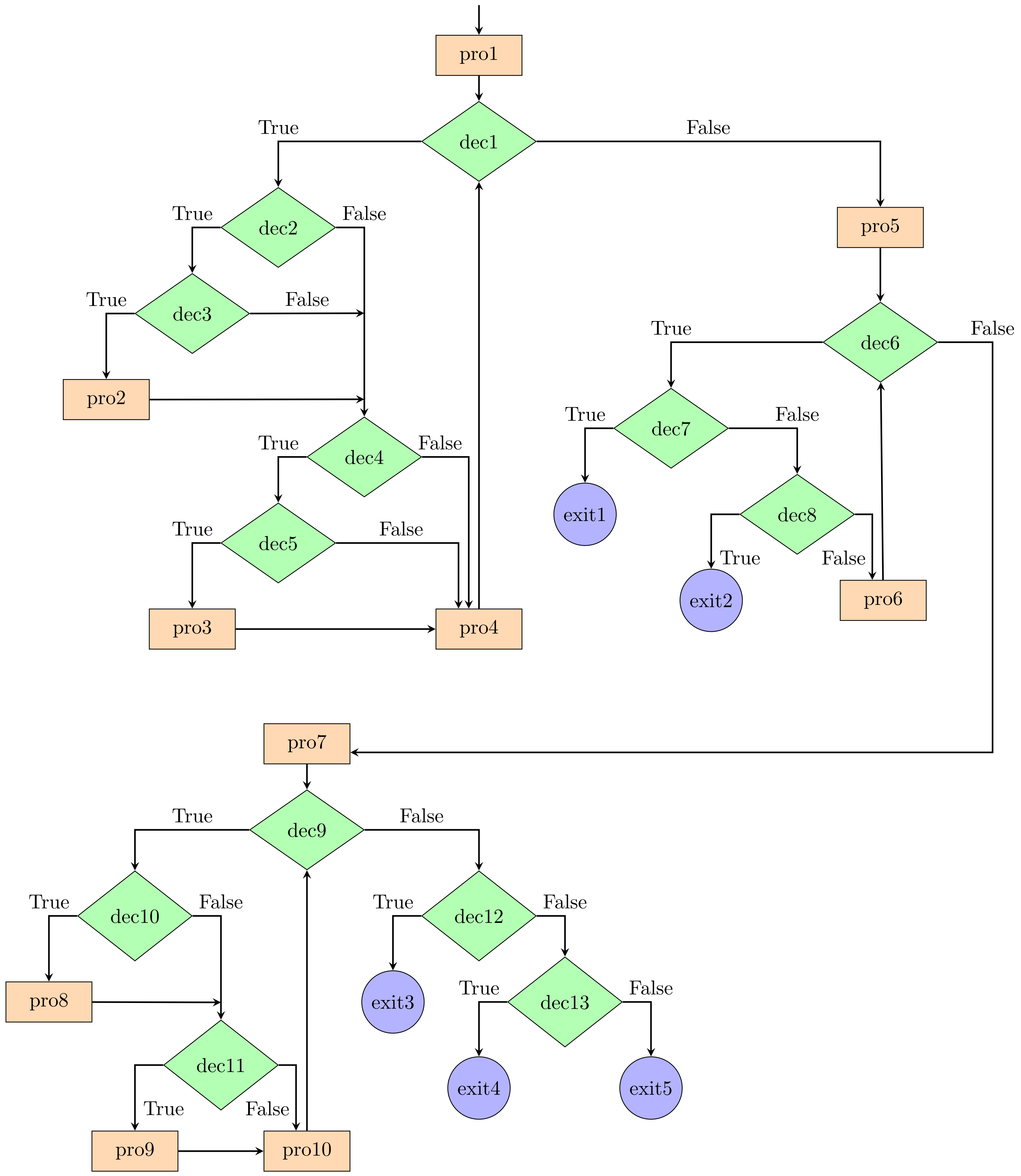

Figure 1: Flow graph for CompareHands::twoPairTie

Coverage

Tools

Lizard

The tool was easy to use and could specify specific files or entire folders and count CC on them. You could specify a certain language to only take these files, which facilitates when working with larger projects. There was also something similar to filter function, generating warnings for all functions with CC over user specified input.

Eclipse with Clover

Both eclipse and clover the whole team found some difficulty setting up. But whence it was up and running (and enabling it on the project), it preformed quite good and showed line both as a summary and in the editor.

DIY

In order to check the branch coverage of the chosen methods, we first manually identified the possible branches for each method, then added these to a hashmap of [branchID, {true,false}]-pairs. We then added markers in each method at each branch using TestBench.BranchReached(branchID) to specify that the branch was actually reached given that the particular segment was executed. After running our tests, we have a hashmap in which the value set can be used to calculate the percentage of coverage (via TestBench.AnalyzeCoverage()) based on the markers we placed in the code. The DIY coverage tracking can be found in the TestBench.java class, and TestBench.AnalyzeCoverage method call should be placed in the teardown of the test suite to ensure that it is run after all tests have been performed. As an example, branch markers in calculateValue yields the following code:

```
public int calculateValue(ArrayList<Card> player) {
    if(isStraightFlush(player)) {
     TestBench.BranchReached("calculateValue1");
        return 8;
    }else if(isFourOfAKind(player)) {
     TestBench.BranchReached("calculateValue2");
        return 7;
    }else if(isFullHouse(player)) {
     TestBench.BranchReached("calculateValue3");
        return 6;
    }else if(isFlush(player)) {
     TestBench.BranchReached("calculateValue4");
     return 5;
    }else if(isStraight(player)) {
     TestBench.BranchReached("calculateValue5");
        return 4;
    }else if(isThreeOfAKind(player)) {
     TestBench.BranchReached("calculateValue6");
        return 3;
    }else if(isTwoPair(player)) {
     TestBench.BranchReached("calculateValue7");
        return 2;
    }else if(isOnePair(player)) {
     TestBench.BranchReached("calculateValue8");
        return 1;
    } else {
     TestBench.BranchReached("calculateValue9");
        return 0;
    }
    }
```

Evaluation

https://github.com/dackebro/cs56-games-poker/blob/master/fork-cover.pdf
https://github.com/dackebro/cs56-games-poker/blob/master/original-cover.pdf

Refactoring

Refactor the top 5 most complex functions to lower complexity

OpponentAI::takeTurn()

I would move the following code to a helper function, and make that function return a touple, which takeTurn would set to the vaiables shouldBet and shouldCall, respectively.

```
    if (delegate.getStep() == PokerGame.Step.BLIND) {
            if (dValue >= 1) {
                shouldBet = true;
            }
        } else if (delegate.getStep() == PokerGame.Step.FLOP) {
            if (dValue >= 3) {
                shouldBet = true;
            }
            if ((dValue == 0 && delegate.getCurrentBet() >= 20)) {
                shouldCall = false;
            }
        } else if (delegate.getStep() == PokerGame.Step.TURN) {
            if (dValue >= 4) {
                shouldBet = true;
            }
            if ((dValue < 2 && delegate.getCurrentBet() > 20)) {
                shouldCall = false;
            }
        } else if (delegate.getStep() == PokerGame.Step.RIVER) {
            if (dValue >= 4) {
                shouldBet = true;
            }
            if ((dValue < 2 && bet > 20))
                shouldCall = false;
        }
```

This would lower the complexity from 23 to 9 and create a new function with complexity 15. This would lower complexity of the original function by 60.8%.

OpponentAI::isStraightFlush()

There was a lot of code duplication present in this function which grossly inflated the complexity. For each suit of the deck, the hands were counted to see if a straight flush was present. This involved iterating through the hand in a for-loop and checking for a straigh flush. We suggest a refactoring in which the straight flush-check is delegated to a separate method. The resulting partial code is then:

```
    if(spadeCounter >= 5) {
         straightFlushCounter = straightFlushCounter(spades, spadeCounter);
        } else if (clubsCounter >= 5) {
         straightFlushCounter = straightFlushCounter(clubs, clubsCounter);
        } else if (heartCounter >= 5) {
         straightFlushCounter = straightFlushCounter(hearts, heartCounter);
        } else if (diamondCounter >= 5) {
         straightFlushCounter = straightFlushCounter(diamonds, diamondCounter);
        } else {
         return false;
    }
```

This refactors the original method of CNN 36 into a method of CNN 11 (isStraightFlush) and a method of CNN 8 (straightFlushCounter).

CompareHands::isStraightTie()

One minor change could be to, instead of having two predicates with four clauses each, make use of two for-loops with a single if-statement inside each one. That could possibly lower the cyclomatic complexity of the method.

CompareHands::twoPairTie()

A lot of the logic in twoPairTie() could be split into smaller, more manageable functions that could also be used throughout the program in other functions as well. For example; the pairTie()-method could very well be called inside twoPairTie with the pairs iteratively, returning an int with the difference in value between the 'pairs of pairs'. The total difference could then decide if a check of potential kickers is needed. This would vastly improve the CCN of this method as it contains a lot of functionality that is almost identical in other methods.

CompareHands::pairTie()

As in the method above, twoPairTie(), the logic concerning attaining the pairs from an ArrayList of cards could easily be swapped into another helper function that could also be used in this method. Other than that, there is yet again logic for checking for the highest kicker in the case of the hands having pairs of the same value. This could, as stated above, be put in a helper function which would reduce the cyclomatic complexity of not only this method but twoPairTie() (and maybe a few others).

Effort spent

Table 1: Time distribution

	Dan	Erik	Kenneth	Pojan	Zacharias
Plenary discussions/meetings	1	1	1	1	1
Discussions within parts of the group	1	1	1	1	1
Reading documentation	1	1	1	1	1
Configuration/building	2	1	1	2	2
Analyzing code/output	1	1	2	1	1
Writing documentation	4	5	4	6	4
Writing code	6	7	6	6	7
Running code/tests	4	3	2	2	3
Searching for potential repositories	2	2	4	2	2
Total	22	22	22	22	22

Overall experience

As with the last lab, a significant portion of our time was spent simply sifting through repos on Github. Again, we feel that this is somewhat detrimental to the learning experience of the course as much of the work relies on the quality of the repo one finds. Selecting from a precomposed pool of repos would not only cut down on the initial confusion, it would also enable more easily weighing the ”difficulty” of the repo given the exercise and invite to comparing solutions between groups more closely afterwards. For this lab, we intentionally chose a repo where the code was easier to grasp than the previous lab. This, we felt, would benefit the learning outcome of the exercise as we could put more actual focus on what we needed to do.

Learning about Lizard and Clover was exciting, and we felt that it was very satisfying to watch the coverage increase directly in the Eclipse IDE. We also found a number of bugs via written tests, which proved why unit testing can be so useful.