Rubric for Assessing Canoga in Java/Android

Name: _Elijah Campbell_____

Email: ____ecampbe3@ramapo.edu_____

Carefully **highlight all** the items that **work correctly**. Incorrect entries may be penalized. Not all the entries may be used for grading.

Setup					
Players	One player is Human	One player is computer	Players alternate		
Board	Human is asked to pick the size of rows when each new round starts	Size can be 9-11	Game works for n=9	Game works for n=10	Game works for n=11
	Starts with a row for human player	Human row has squares 1-n		Starts with a row for computer player	Computer row has squares 1-n
First player	Dice are thrown for both players	Player with more pips plays first		If both players have same pips, dice are thrown again	Process is repeated till first player is determined
Human Player					
Playing	Has option to throw 1 die if 7-n are covered	Has option to throw 1 die only if 7-n are covered	Will throw both dice otherwise		Player can cover or uncover, but not both
Turn	Can repeatedly throw dice		Cannot stop if move is available	Must stop when no move is available	
Covering	Can cover own uncovered square	Cannot cover own covered square		Sum of covered squares = sum of pips	Can cover 1, 2, 3 or 4 squares
Uncovering	Can uncover opponent's covered square	Cannot uncover opponent's uncovered square		Sum of uncovered squares = sum of pips	Can uncover 1, 2, 3 or 4 squares
Computer Player					
Playing	Executes option to throw 1 die if 7-n are covered	Executes option to throw 1 die only if 7-n are covered	Will throw both dice otherwise		Will cover or uncover, but not both
Turn	Repeatedly throws dice		Will not stop if move is available	Will stop when no move is available	
Covering	Covers own uncovered square	Will not cover own covered square		Sum of covered squares = sum of pips	Can cover 1, 2, 3 or 4 squares
Uncovering	Uncovers opponent's covered square	Will not uncover opponent's uncovered square		Sum of uncovered squares = sum of pips	Can uncover 1, 2, 3 or 4 squares

Playing the Game					
Round Completion	Ends when human covers all own squares	If so, human is declared the winner		Ends when human uncovers all computer squares	If so, human is declared the winner
	Ends when computer covers all own squares	If so, computer is declared the winner		Ends when computer uncovers all human squares	If so, computer is declared the winner
Score	If round won by covering, score is sum of opponent's **un**covered squares			If round won by uncovering, score is sum of own covered squares	
	Round score is computed correctly	Round score is announced	Round score is added correctly to the winner's score	Updated tournament scores of both players is announced	
Handicap	If winner of round is also first player of round, advantage goes to opponent			If winner of round is second player of round, winner keeps advantage for next round	
	Winner with advantage starts with one square covered		Covered square is the sum of digits of winning score of previous round		
	Advantage square cannot be uncovered by human till computer completes one turn		Advantage square will not be uncovered by computer until human completes one turn		
Tournament Control	At the end of a round, asks human whether another round should be played	If yes, another round is started			
	If no, announces the winner of the tournament	Winner is the player with the most points scored	Announces the score of both players	If both players have the same score, the tournament is a draw	Program exits after announcing winner of the tournament
Implementation Features					
Serialization	Provides option to stop game after each turn	Game is saved into text file	Correct format used for text file	Game state correctly saved	Game quits upon serialization
	Provides option to resume game from text file	Prompts for name of text file			
Correctly Restores	Human's squares	Human's score		Computer's squares	Computer's score
	The first player		The next player		
Dice throw	Can manually input a sequence of dice throws	Can play game for both players using manually input dice throws		Can automatically generate a sequence of dice throws	
Help Mode	Has option to ask computer for			Computer uses its own strategy	Computer prints the rationale for its

	a recommended move			to recommend the “best” move	recommendations in a context-sensitive
	Recommends whether to cover or uncover		Recommends player’s squares to be covered		Recommends opponent’s squares to be uncovered
Computer’s Strategy					
Whether to cover or uncover	*If one choice wins the game instantly, do that* *If covering is possible but uncovering isn't, cover* *If uncovering is possible but covering isn't, uncover* *If both are possible, rely on strategy:* *- Count uncovered squares on either side, choose option based on likelihood of winning* *- If player has less than or equal to half of their squares left to cover, choose cover* *- If opponent has less than half of their squares left to uncover, choose to uncover. Otherwise, cover*				
Covering its squares	*If one choice wins the game instantly, do that* *For covering, pick the move with the fewest squares and highest value (bigger squares better)*				
Uncovering opponent’s squares	*If one choice wins the game instantly, do that* *For uncovering, pick the move with the most squares (more squares better)*				
Throwing one or two dice when 7-n covered	*If the number of squares on the player’s side of the board add up to 6 or less, we throw one dice. If they add up to more than 6, we throw two.*				
Game features					

Validates input from human player	Input on whether to cover or uncover	Input of squares to be covered	Input of squares to be uncovered	Input on whether to throw one die or two	Asking for help from the computer
	Input on whether to start a new round		Input on whether to start a round using a text file	Input on whether to suspend a round after a turn	
Output	Squares are labeled with numbers	Human and Computer rows clearly marked	Both rows read left to right		Handicap, if any, clearly displayed
	First player clearly displayed	Next player clearly identified		Scores of both players clearly displayed	Scores correctly updated after each round
For Human player	Tossed die/dice clearly displayed	Squares properly updated after each move		Computer's recommendation displayed in user-friendly format	
For Computer player	Tossed die/dice clearly displayed	Squares properly updated after each move		Computer's move is described in user-friendly format	Computer's strategy is explained in a context-sensitive manner
Graphical User Interface					
GUI for input:					
	Input on whether to cover or uncover	Input of squares to be covered	Input of squares to be uncovered	Input on whether to throw one die or two	Asking for help from the computer
	Input on whether to start a new round		Input on whether to start a round using a text file	Input on whether to suspend a round after a turn	
GUI for Output:	The dice	The board			
	Computer's recommendation to human player	Description of each move made by the computer player			
GUI for Game State:	Score of each player	Winner/loser of a round			Winner/loser of the tournament
GUI Quality	Minimizes text input	Precludes invalid inputs			
	Interface is functional – only necessary widgets displayed	Interface is logical – widgets displayed only at the correct time	Interface is intuitive – user always knows what options are available next		Interface is aesthetically designed
Design					
MVC design	Model and view are separate classes	All game logic handled by the model	All game state saved in the model only		

	Model has no reference to the view, but view has reference to the model	View updates itself based on the model		Controller updates the model only, not the view	Controller refreshes the view when necessary
Object-oriented design	Appropriate classes are included (such as listed below)	Each class is complete – self-contains all the necessary functionality	Inheritance is used for player classes: computer and human inherit from a base class	Overridden functions used for player classes	
Code Design – Data flow	Data: Only independent variables saved, dependent variables saved sparingly, only for efficiency	Data is *not* saved redundantly, no potential fidelity problems in data storage	Data is encapsulated – access to data is controlled	Changes to data always validated	
Code Design – Control flow	Overall design is hierarchical and evident in main()	Code for repeated execution separated from code for single execution (e.g., of round, tournament)			
Code Reuse	Code properly factored out of if-else, loops	Functions defined for any code executed more than once	Each function in charge of only one logical task		
Implementation					
***Board* Class**	All data members are private	Constructor initializes *all* data members	Selectors are const, don't break encapsulation	Mutators validate input, don't break encapsulation	Destructor releases resources
***BoardView* Class**	All data members are private	Constructor initializes *all* data members	Selectors are const, don't break encapsulation	Mutators validate input, don't break encapsulation	Destructor releases resources
***Player* Class**	All data members are private	Constructor initializes *all* data members	Selectors are const, don't break encapsulation	Mutators validate input, don't break encapsulation	Destructor releases resources
***Human* Class**	All data members are private	Constructor initializes *all* data members	Selectors are const, don't break encapsulation	Mutators validate input, don't break encapsulation	Destructor releases resources
***Computer* Class**	All data members are private	Constructor initializes *all* data members	Selectors are const, don't break encapsulation	Mutators validate input, don't break encapsulation	Destructor releases resources
***Game/Round* Class**	All data members are private	Constructor initializes *all* data members	Selectors are const, don't break encapsulation	Mutators validate input, don't break encapsulation	Destructor releases resources

***Tournament* Class**	All data members are private	Constructor initializes *all* data members	Selectors are const, don't break encapsulation	Mutators validate input, don't break encapsulation	Destructor releases resources
Identifiers	All classes have names corresponding to nouns in the problem description	All client functions have names corresponding to verbs in the problem description	Any abbreviations in the names are readable		
Coding style	No global variables used	Symbolic constants are used whenever possible	All literal constants are explained at *each* occurrence	Principle of least privilege used for parameter passing	
Courtesy Programming					
Listing	Code is indented properly		Client functions listed in the order in which they are first called	Classes are listed from basic to composite and derived	Each class listed in the following order: public, protected and private
Documentation	Every function has a complete header	Within each function, code is properly commented – steps in the algorithm are listed	Comments in the code describe semantics, not syntax	Comments in the code do not have spelling/ grammatical errors.	
Submission - Manual					
Screen shots of:	First player of the round being determined	Computer's move being explained	Computer providing help		Winner of the tournament being announced
Includes:	Bug report	Missing features report		Project log	Help from Generative AI
	Description of classes	Description of data structures		Source and documentation are placed in a directory with your full name and the directory is zipped	
Milestones uploaded?	First: Yes Third: Yes		Second: Yes Fourth: Yes		

Demonstration:

Case	1	2	3	4	5	6	7	8
1								*

2						*	*	*
3					*	*	*	*
4								*
5				*	*	*	*	*
New Game								

Do not delete these pages