**Rubric for Assessing Dominoes Buildup in C++**

Name: John Matthew Sahagun

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 |  |  |
| **Dominoes** | Human starts with black double-six set | Set contains all and only 28 tiles | The set is shuffled |  |  |
|  | Computer starts with white double six-set | Set contains all and only 28 tiles | The set is shuffled |  |  |
| **Board** | Starts with six stacks for human | Rest of the tiles in boneyard for human |  | Starts with six stacks for computer | Rest of the tiles in boneyard for computer |
| **First player** | A domino is picked from each boneyard | Player with more pips on the tile plays first |  |  |  |
|  |  |  | If both players have same pips, tiles are returned to boneyards | Boneyards are shuffled | Process of selecting first player is repeated |
| **Human Player** | | | | | |
|  | Player picks a tile from own hand to place | Player places only one tile per turn |  |  |  |
| **Non-double**  **on**  **Double/**  **Non-Double** | Player can place non-double tile on own smaller tile | Player cannot place non-double tile on own larger tile |  | Player can place non-double tile on opponent’s smaller tile | Player cannot place non-double tile on opponent’s larger tile |
| **Double**  **On**  **Double** | Player can place double tile on own smaller double tile | Player cannot place double tile on own larger double tile |  | Player can place double tile on opponent’s smaller double tile | Player cannot place double tile on opponent’s larger double tile |
| **Double**  **On**  **Non-Double** | Player can place double tile on own smaller non-double tile | Player can place double tile on own larger non-double tile |  | Player can place double tile on opponent’s smaller non-double tile | Player can place double tile on opponent’s larger non-double tile |
| **Pass** | Player can pass if no tile can be placed | Player can pass only if no tile can be placed |  |  |  |
| **Computer Player** | | | | | |
|  | Player picks a tile from own hand to place | Player places only one tile per turn |  |  |  |
| **Non-double**  **on**  **Double/**  **Non-Double** | Player can place non-double tile on own smaller tile | Player does not place non-double tile on own larger tile |  | Player can place non-double tile on opponent’s smaller tile | Player does not place non-double tile on opponent’s larger tile |
| **Double**  **On**  **Double** | Player can place double tile on own smaller double tile | Player does not place double tile on own larger double tile |  | Player can place double tile on opponent’s smaller double tile | Player does not place double tile on opponent’s larger double tile |
| **Double**  **On**  **Non-Double** | Player can place double tile on own smaller non-double tile | Player can place double tile on own larger non-double tile |  | Player can place double tile on opponent’s smaller non-double tile | Player can place double tile on opponent’s larger non-double tile |
| **Pass** | Player can pass if no tile can be placed | Player passes only if no tile can be placed |  |  |  |
| **Playing the Game** | | | | | |
| **Hands** | First hand has 6 tiles | Second hand has 6 tiles | Third hand has 6 tiles | Fourth hand has 4 tiles |  |
| **Hand Completion** | Hand ends when all tiles are placed |  | Hand ends if neither player can play a tile |  |  |
| **Score** | Scores calculated when hand ends | Score is sum of all own tiles on stacks | Score subtracts any tiles that could not be played |  | Hand scores are added to round scores |
| **Round Completion** | Round ends when fourth hand ends |  |  |  |  |
|  | Winner of the round is announced | Winner is the player with the greatest score | The scores of both players are announced |  | If both scores are the same, the round is a draw |
| **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 rounds won | Announces the rounds won by both the players | If both players have won the same number of rounds, 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** | Computer stacks | Computer boneyard | Computer’s current hand | Computer’s score in the round | Computer’s rounds won |
|  | Human stack | Human boneyard | Human’s current hand | Human’s score in the round | Human’s rounds won |
|  | The next player |  |  |  |  |
| **Help Mode** | Has option to ask computer for a recommended move | Computer recommends the best tile to place | Computer recommends the stack on which to place the tile | Computer uses its own strategy to recommend the “best” move | Computer prints the rationale for its recommendation |
| **Computer’s Strategy** | | | | | |
| **Picking the tile to place** | loops through each stack and cross references each one with a domino in the player’s hand and checks if it is a legal move | | | | |
| **Picking the stack for a Non-Double tile** | loops through each stack and cross references each one with a domino in the player’s hand and checks for the first combination to be a legal non-double tile move | | | | |
| **Picking the stack for a Double tile** | loops through each stack and cross references each one with a domino in the player’s hand and checks for the first combination to be a legal double tile move | | | | |
| **Game features** | | | | | |
| **Validates input from human player** | Input of tile to place | Input of stack on which to place | Input of passing a turn | Asking for help from the computer | Input on whether to start a new round |
|  | Input on whether to start a game using a text file | Input on the text file from which to resume a game |  | Input on whether to suspend a game after a play | Input of the name of the file in which to save the game |
| **Output** | Stacks referred to with color and number, e.g., B1 | Tiles referred to with color and pip counts |  | Next player clearly identified |  |
|  | Score of both players clearly displayed | Scores correctly updated after each round |  | Rounds won clearly displayed for both players |  |
| **For Human player** | Stacks properly updated after each move | Hand properly updated after each move |  | Computer’s recommendation displayed in user-friendly format |  |
| **For Computer player** | Stacks properly updated after each move | Hand properly updated after each move |  | Computer’s move is described in user-friendly format | Computer’s strategy is explained |
| **Design** | | | | | |
| **Object-oriented design** | At least 7 classes are included (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 | Virtual 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) | | Display issues separated from problem logic (Model Vs View) | |
| **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 |
| ***Tile***  **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 |
| ***Hand***  **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 |
| ***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** | | | | | |
| **User’s Manual** | Includes 2 screen shots | Describes how to run the program | Includes bug report | Includes missing features report | Includes additional features report |
| **Technical Manual** | Includes description of classes | Includes description of data structures | Includes project log | Source and documentation are placed in a directory and the directory is zipped | |
| **Milestones uploaded?** | First: N/A | | Second: YES | | |