**“King of Tokyo”**

**Final Project Proposal**

**CSCI24000 Fall 2022**

**Conner Shipley**

**Project Overview**

For my final project I’ve recreated the “King of Tokyo” boardgame in Java. In the game, 2-4 players choose a Kaiju-style monster (such as “Cyberkitty”, “Space Penguin”, or “MekaDragon”) to battle other monsters for control of Tokyo. There are two locations a monster can exist in: Tokyo or outside of Tokyo, every monster begins the game outside of Tokyo, and only one monster can occupy Tokyo at a time. Players receive 1 victory point when entering Tokyo, and 2 victory points when beginning their turn in Tokyo. If no player is occupying Tokyo at the beginning of a player’s turn, that player must move into Tokyo. Players start with 10 health and 0 victory points, and the goal is to smash all other players down to 0 health or be the first to acquire 20 victory points.

The first player rolls six dice with values of 1, 2, 3, Smash, Energy, or Heal three times, choosing which dice to keep in between each roll. If a player rolls 3 of a kind of any digit, they receive the face value of that digit as victory points plus an additional victory point for each additional die of that digit (i.e. rolling 4 threes would yield 3 victory points + 1 victory point = 4 total). Rolling a smash damages enemies in the opposite location of the player, so a player that rolls a smash in Tokyo would damage all players outside of Tokyo. If a player outside of Tokyo smashes the player in Tokyo, the player in Tokyo may yield the city, exiting it and forcing the player who attacked them to move into it. Rolling an energy grants the player +1 energy, which can be used to purchase power-up cards. Rolling a heal grants the player +1 health (up to a maximum of 10 health) and can only be used outside of Tokyo.

And that’s basically the game: it’s a series of dice rolls and resolution of the rolls. Each player must login or create an account at the beginning of each game, and player accounts are awarded wins, draws, and losses at the end of each game, then serialized into an external file. All player stats or a specific player’s stats can be viewed from the main menu.

Known issues include the card system, smash damage application, user input checks, and no account already exists check when creating a new account. The card system was going to add a layer of complexity that I wasn’t comfortable taking on in the amount of time given for this project, so I decided to nix it, which makes Energy effectively worthless. The game is already decently complex as is, and the cards would add new features like Poison and Fire DoT counters, increased max health or number of attacks, armor, damage reflection, cards to force the current King to yield, etc. (events that would trigger a series of cascading effects, essentially). I’d like to continue working on the project over winter break, and think I’ll try to implement a Card class and use operator overloading to “add” or “subtract” the cards to specific Monsters.

When a player outside of Tokyo attacks the King of Tokyo and the King yields, the player outside moves into Tokyo and then is allowed to continue applying Smash damage to the remaining players outside of Tokyo. In reality, the Smash damage should all be resolved at once, so only the King of Tokyo should take damage in that situation, then the player outside would move into Tokyo. Currently, the method performing this action is iterating through a list of all opponent players, comparing their location to the player’s current location, then applying smash damage as needed. This issue could be solved by assigning the value of the player’s location to a variable at the beginning of the function, then comparing the opponent’s location to this variable instead of the player’s current location.

User input is not always validated, particularly and most glaringly in the case of a player entering which dice they’d like to keep as a CSV. This is purely just due to my own laziness in not wanting to write a bunch of while loops or try-catch statements, but in the case of the CSV validation, I’ve brainstormed the following function that could be implemented in a while loop to catch bad input:

public boolean isValidCSV(String[] s){

try:

for (int i = 0; i < s.length; i++){

cast s[i] as int;

if (int s[i] is not between 0-7){

return false;

} // end if

} // end for

return true;

catch (cast type error):

return false;

} // end isValidCSV

The function would accept an array of Strings as input (so ideally would look like [“1”, “3”, “5”]), then iterate through the array, try to cast each String as an int, and check its value to make sure it’s between 0 and 7. If all elements pass, the function would return true; if an element causes a cast type error or is not between acceptable values, the function would return false.

The final major known issue is that when creating a new account, there is no check to make sure the account name isn’t already taken. I’m not exactly sure what issue this would cause (I think it might actually be okay if the passwords are different), but my guess is that it would add an identical account to the end of the ArrayList storing all accounts, and if the two players ever tried to login and play at the same time again, they would both access and deserialize the same account (whichever one was created first and therefore comes first in the ArrayList), which may cause some sort of merge issue when the ArrayList is serialized again or would just keep adding new instances of the same account in the ArrayList every time this happened. This could be solved by creating a boolean findPlayer or checkAccountName function implemented in a while loop that would force the user to create a unique username.

The intended user is mostly me, possibly friends/family, and the problem I am trying to solve with this project is boredom/needing to come up with a good final project for CS240. Technologies I will use include serialization/file IO, classes, abstracts, interfaces, makefile and .jar files, and Java.

**Use Case Analysis**

* 1. King of Tokyo
     1. Main Menu
        1. Screen clears
        2. Menu options are printed to console
        3. User selects “View All Players”
           1. Screen clears
           2. Account ArrayList is deserialized from file

Successful

Account stats are printed to console

User presses any key to return to Main Menu (1.0.1.)

Unsuccessful

User is notified of failure to load accounts

User presses any key to return to Main Menu (1.0.1.)

* + - 1. User selects “Search Player Stats”
         1. Screen clears
         2. Account ArrayList is deserialized from file

Successful

User is asked to input account name

Account is searched for in ArrayList

Successful

Account stats are printed to console

User presses any key to return to Main Menu (1.0.1.)

Unsuccessful

User is notified account was not found

User presses any key to return to Main Menu (1.0.1.)

Unsuccessful

User is notified of failure to load accounts

User presses any key to return to Main Menu (1.0.1.)

* + - 1. User selects “Exit”
         1. Screen clears
         2. Program ends
      2. User selects “Start New Game”
         1. Screen clears
         2. User is asked to input number of players

Screen clears

First player is asked to “Login” or “Create New Account”

First player chooses “Login”

Screen clears

First player is asked to input account name or “EXIT” and password

First player enters account name and password

Account is searched for in ArrayList

Successful

First player is asked to choose their monster

First player presses any key to continue

Return to 1.0.1.6.2.1. and repeat with remaining players

Unsuccessful

First player is notified of failure

Return to 1.0.1.6.2.3.2.

First player enters EXIT and password

Goto 1.0.1.6.2.4.1.

First player chooses “Create New Account”

First player is asked to enter new account name and password

Return to 1.0.1.6.2.3.4.1.1.

* + - * 1. Screen clears
        2. Monsters roll for turn order and are notified of results
        3. First monster is asked to press any key to start turn.

Opponent information is shown to First monster

First monster’s information is displayed

Roll is displayed

First monster enters which dice they’d like to keep as CSV

If no dice selected

Reroll all dice

Return to 1.0.1.6.5.3. while n < 3

If some dice selected

Reroll dice that weren’t kept

Return to 1.0.1.6.5.3. while n < 3

If all dice selected or n is 3

End dice rolling phase

Goto 1.0.1.6.5.4.

Final roll is resolved

Results of round are displayed

Return to 1.0.1.6.5. and repeat with remaining monsters until one monster has won

Monster wins by Victory Points

Screen clears

Winning monster is displayed

Player accounts are serialized

Users are asked to press any key to continue and return to Main Menu (1.0.1.)

Monster wins by Elimination

Return to 1.01.6.5.7.1.

Draw (All Monsters Eliminated)

Screen clears

Display draw results

Return to 1.0.1.6.5.7.3.

**Data Design**

Main Menu

Game Players

Dice Monsters

The main data of the game is the dice. The best way to represent them is as a class with an int value and a HashTable that the value can be keyed into to generate a String, such as 3 produces “Smash”. Six instances of the Dice class will need to be aggregated as an array into the Game class, where they will be repeatedly rolled then read by a number of Player class instances.

Monsters are also an important data structure that deserve their own class, as they are temporarily owned by instances of the Player class and keep track of the Player’s performance during a game. They will need to store health, victory points, and location, and a single instance of the Monster class will be aggregated into each instance of the Player class during a game.

The Player class will need to store a name, password, wins, draws, losses, and most/least played monsters, as well as an ArrayList of current opponents the Player is facing and the Player’s current Monster for each game. Players will need to be serializable, but the Monster and opponents should be transient and not stored between games.

I thought of the Game class as a temporary object aggregated by the Main Menu that is repeatedly born, takes in Players, produces results for the Players, stores the updated Players, then is thrown away. The Game class will need an array of six Dice, an ArrayList of current Players, and an ArrayList of all Players for logging in, storing new accounts, and saving results/new accounts.

Finally, the Main Menu class will drive the program and repeatedly generate new Game instances or allow access to the serialized ArrayList of all Players. It will need access to the same ArrayList of all Players that the Game class is accessing, and will need to reload the ArrayList each time Player stats are viewed, so that results can viewed before a game and after a game and the user will see updated Player stats without having to exit the program.

**Algorithm**

**ScreenHelper Abstract:**

* Methods
  + clearScreen – Clear the console screen
    - Input – None
    - Output – None
    - Steps –
      * Print command that clears console in Linux
  + pressAnyKey – Prompt the user to press any key to continue, then clear the screen
    - Input – None
    - Output – None
    - Steps –
      * Ask user to press any key to continue
      * Read input
      * Call clearScreen() method

**MainMenu Class (Extends ScreenHelper):**

* Attributes
  + game – an instance of the Game Class
  + accounts – ArrayList of all Player instances deserialized from external file
* Methods
  + main – Drive the program, constructs new instance of MainMenu class
    - Input – String[] args
    - Output – None
    - Steps –
      * Call MainMenu constructor and assign instance to variable
  + MainMenu – default constructor
    - Input – None
    - Output – None
    - Steps –
      * Call menu() method
  + menu – Print options the program can perform and ask user to choose one
    - Input – None
    - Output – None
    - Steps –
      * Call clearScreen() method
      * While true
      * Print available program actions
      * Ask user to select one
      * Read input
      * Compare input to available actions
      * Perform selected action
      * If user chooses exit, end while loop and clear screen
  + searchPlayer – Allow user to input a player’s account name and search for their win/loss/draw and most/least played monster statistics
    - Input – None
    - Output – None
    - Steps –
      * Call populateAccounts() method to update Player accounts ArrayList
      * Call clearScreen() method
      * Set boolean variable to false
      * Ask user for account name
      * Read user input
      * Loop through all player accounts and compare input to each one
      * If the account names match, show the player stats and set boolean to true
      * If boolean stays false, no matching player account has been found and user is provided feedback
      * Call pressAnyKey() method to continue back to main page
  + viewAllPlayers – Show the statistics of all player accounts
    - Input – None
    - Output – None
    - Steps –
      * Call populateAccounts() method to update Player accounts ArrayList
      * Call clearScreen() method
      * Loop through all accounts and show each one’s statistics
      * Call pressAnyKey() method to continue back to main page
  + populateAccounts – Deserialize Player accounts ArrayList stored in external file
    - Input – None
    - Output – None
    - Steps –
      * Try to read in the ArrayList from the external file and assign it to accounts attribute
      * If unsuccessful, print error message

**Game Class (Extends ScreenHelper):**

* Attributes
  + players – ArrayList of current Player objects of the game
  + dice – Array of six Dice objects
  + monsters – ArrayList of Strings that represent the available monsters a Player can choose from; used in selecting and assigning monsters to players
  + accounts - ArrayList of all Player instances deserialized from external file
* Methods
  + main – Unit test the Game Class
    - Input – String[] args
    - Output – None
    - Steps –
      * Create new instance of Game Class and assign to variable
      * Call match() method
  + Game – Default constructor, deserialize ArrayList of all accounts from external file for Player logins, fill list with available monsters for players to choose from, create the Dice for the game, initialize appropriate number of Player instances and allow them to login, populate each Player’s opponents, start the game
    - Input – None
    - Output – None
    - Steps –
      * Call populateAccounts() method
      * Call populateMonsters() method
      * Call populateDice() method
      * Call populatePlayers() method
      * Call populateOpponents() method
      * Call match() method to begin game
  + isTokyoEmpty – Check if any Player’s monster is occupying Tokyo
    - Input – None
    - Output – boolean
    - Steps –
      * Iterate through list of current players
      * If a player’s monster’s location is Tokyo, return true
      * Otherwise, return false
  + countAlive – Count the number of players left alive
    - Input – None
    - Output – an integer representing the number of alive players (monsters with health > 0)
    - Steps –
      * Initialize counter variable
      * Iterate through list of current players
      * If Player’s monster’s health is greater than 0, increment counter
      * Return counter
  + victoryByDeath – See if a specific Player instance has won the game by eliminating all other monsters
    - Input – Player instance
    - Output – boolean
    - Steps –
      * If the number of alive is 1 and this Player’s monster’s health is greater than 0, they have won the game, return true
      * Otherwise, return false
  + victoryByPoints – Check if a specific Player instance has won the game by reaching 20 Victory Points first
    - Input – Player instance
    - Output – boolean
    - Steps –
      * If the Player’s monster’s health is greater than 0 and their victory points are greater than 19, they have won the game, return true
      * Otherwise, return false
  + awardWinLoss – Award wins and losses to Player accounts as appropriate
    - Input – Player instance representing the Player that has won the game
    - Output – None
    - Steps –
      * Increment Player’s wins
      * Iterate through list of Player’s opponents
      * Increment opponents’ losses
  + awardDraw – Award a draw to all current Players in the event everyone is eliminated simultaneously
    - Input – None
    - Output – None
    - Steps –
      * Iterate through list of current Players
      * Increment each Player’s draws

**\*\*\*\*\*checkWinner and declareWinner probably need a little work; they’re functioning properly, but the logic is slightly messy and overlaps\*\*\*\*\***

* + checkWinner – Check if any current Player has won the game
    - Input – None
    - Output – boolean
    - Steps –
      * Iterate through list of current Players
      * If player has won the game by Victory Points, return true
      * If player has won the game by eliminations, return true
      * If all players have been eliminated, return true
      * Otherwise, return false
  + declareWinner – Declare which Player has won the game (or declare a draw)
    - Input – None
    - Output – boolean
    - Steps –
      * Iterate through list of current Players
      * If player has won the game by Victory Points, call clearScreen() method, award wins and losses, print feedback to console, increase each Player’s monster play count, serialize the plays, call pressAnyKey() to return to main menu, return true
      * If player has won the game by eliminations, call clearScreen() method, award wins and losses, print feedback to console, increase each Player’s monster play count, serialize the plays, call pressAnyKey() to return to main menu, return true
      * If all players have been eliminated simultaneously, call clearScreen() method, award draws, print feedback to console, increase each Player’s monster play count, serialize the plays, call pressAnyKey() to return to main menu, return true
      * Otherwise, return false
  + incMonsterCount – Increment each Player’s monster play count for whichever monster they chose this game
    - Input – None
    - Output – None
    - Steps –
      * Iterate through list of current Players
      * Increment the Player’s monster play count for their chosen monster
  + match – Play the game, consists of rounds which consist of Player turns
    - Input – None
    - Output – None
    - Steps –
      * While no Player has won
      * Play a round
      * When a Player has won, declare the winner
  + round – Play a round, a subset of a match and consisting of Player turns
    - Input – None
    - Output – None
    - Steps –
      * Iterate through list of current Players
      * If no one has won and the Player’s monster’s health is greater than 0, have the Player instance play their turn
  + turn – Player plays their turn, a subset of a round
    - Input – Player instance representing whichever Player’s turn it currently is
    - Output – None
    - Steps –
      * Print the Player who’s turn it is
      * Check if their monster is in Tokyo, award victory points if so
      * Show the Player’s monster’s and opponents’ monster’s stats
      * Allow Player to roll and choose dice
      * Resolve the dice
      * If no Player’s monster controls Tokyo, move this Player’s monster into Tokyo
      * Show monster stats again after turn has been resolved
  + startTurn – Notify the Player whose turn is about to begin and prompt them to press any key to continue
    - Input – Player instance representing Player whose turn it is
    - Output – None
    - Steps –
      * Print Player’s monster’s name whose turn it is
      * Call pressAnyKey() method
  + showTable – Show Player’s monster’s stats and opponents’ monster’s stats
    - Input – Player instance representing Player whose turn it is
    - Output – None
    - Steps –
      * Iterate through list of Player’s opponents
      * Show each opponent’s monster’s stats
      * Show player’s monster’s stats
  + numberOfPlayers – Ask users to enter number of players for the current game (2-4 players)
    - Input – None
    - Output – an integer with value 2-4 representing the number of players for the current game
    - Steps –
      * Ask user to input number of players
      * Read input
      * Check that input falls within valid range
      * If so, return number of players
      * Otherwise, force user to input number of players again
  + chooseMonster – Ask a Player to pick which monster they’d like to play as for the current game
    - Input – Player instance
    - Output – None
    - Steps –
      * Iterate through list of monster options
      * Print off each option
      * Ask user to select one
      * Read input
      * Assign selected monster to Player
      * Remove the monster as a choice from the list
  + playerMenu – Allow a user to login into a previously created Player account or create a new account
    - Input – an integer with value 1-4 representing the Player who is currently active (i.e. Player 1, Player 2, etc.)
    - Output – Player instance
    - Steps –
      * Print Player who’s active and their options
      * Prompt user to select option
      * Read user input
      * If they choose login, take them to the Player login screen
      * If they choose create new account, initialize new Player instance and add new Player to list of all accounts
      * Return the deserialized (login) or newly created Player instance
  + login – Ask user to enter their Player credentials and login into their account
    - Input – None
    - Output – Player instance, deserialized from external file or newly created
    - Steps –
      * Ask user to enter account name
      * Ask user to enter password
      * Read inputs
      * Iterate through list of all stored Player accounts
      * If inputs match an account, return that account
      * If player wishes to quit login, create new Player instance, add instance to accounts list, and return new Player
  + populateMonsters – Fill ArrayList monsters attribute with appropriate monster name Strings
    - Input – None
    - Output – None
    - Steps –
      * Add each monster’s name to monsters list
  + populateAccounts – Deserialize ArrayList of all stored Player accounts
    - Input – None
    - Output – None
    - Steps –
      * Try to deserialize ArrayList from external file
      * If successful, assign ArrayList to accounts attribute
      * If unsuccessful, print error message and assign new ArrayList to accounts attribute
  + populatePlayers – Populate the game with the number of players requested (2-4)
    - Input – None
    - Output – None
    - Steps –
      * Ask users to input number of players
      * Iterate through number of players
      * Each iteration, allow Player to login or create new account
      * Add Player to list of current Players for the game
      * Allow Player to choose their monster
      * Call pressAnyKey() method to continue
      * Have all players roll a twenty-sided die for turn order and sort the list of current Players accordingly
  + sortPlayers – Sort the ArrayList of current Players according to their twenty-sided die roll (descending order)
    - Input – None
    - Output – None
    - Steps –
      * Iterate through list of current Players
      * Print each Player’s roll
      * Call Collections.sort() method to sort the list of current Players
      * Print the final turn order
  + populateOpponents – Form a list to keep track of the opposing Players in the current game
    - Input – None
    - Output – None
    - Steps –
      * Iterate through list of current Players
      * Iterate through list of current Players again
      * If the two Player objects are not the same, they are different (opposing) Players
      * Add the opposing Player object to the Player’s list of opponents
  + populateDice – Fill the Dice array with 6 new instances of the Dice Class
    - Input – None
    - Output – None
    - Steps –
      * Iterate through dice array
      * Populate each index in the array with a new instance of the Dice Class
  + rollDice – Roll the Dice in the dice array to randomize their values
    - Input – None
    - Output – None
    - Steps –
      * Iterate through dice array
      * For each die, if it’s keep\_status is false (the Player doesn’t want to keep it), call its roll() method
  + showDice – Print the values of all Dice in the dice array
    - Input – None
    - Output – None
    - Steps –
      * Iterate through dice array
      * For each die, print its value
  + whichDice – Ask the Player which Dice they’d like to keep and which they’d like to reroll
    - Input – None
    - Output – String[] with each element representing the index of the die they’d like to keep (i.e. [“1”, “3”, “5”] will keep first, third, fifth Dice)
    - Steps –
      * Ask Player to enter which Dice they’d like to keep as a CSV (i.e. “1,3,5”)
      * Read input
      * Split into String array on the commas
      * Return String array
  + keepDice – Changes the keep\_status of the Dice the Player would like to keep (not reroll)
    - Input – String[] with each element representing the index of the die they’d like to keep (i.e. [“1”, “3”, “5”] will keep first, third, fifth Dice)
    - Output – boolean
    - Steps –
      * If the String array is blank, return true (reroll all Dice)
      * If the first element of the String array is “7”, return false (keep all Dice)
      * Otherwise, iterate through String array
      * Change the keep\_status of the selected Dice to true
      * Return true (reroll Dice without true keep\_status)
  + chooseDice – Allow Player to roll Dice up to 3 times, selecting which Dice they’d like to keep in between each roll
    - Input – None
    - Output – None
    - Steps –
      * Reset Dice
      * While the Player wishes to keep rolling
      * Roll the Dice
      * Show the Dice
      * Reset the Dice
      * Prompt Player to select which Dice they’d like to keep
      * Change the keep\_status of the selected Dice
      * Show the Final Roll
      * Reset the Dice
  + resetDice – Reset the keep\_statuses of all Dice to false (will be rerolled)
    - Input – None
    - Output – None
    - Steps –
      * Iterate through dice array
      * For each die, change its keep\_status to false
  + serialize – Store all Player accounts in an external file
    - Input – None
    - Output – None
    - Steps –
      * Try to write the ArrayList accounts attribute (representing a permanent store of all Player accounts ever created) to an external file
      * If an error occurs, print the error message

**Player Class (Implements Serializable, Comparable<Player>):**

* Attributes
  + name – String representing the account/Player name
  + password – String representing the password of the account
  + TRANSIENT monster – an instance of the Monster Class that the Player controls for the span of a game; effectively tracks the Player’s performance for the game
  + TRANSIENT opps – ArrayList of opposing Player objects for the current game
  + wins – integer representing the total number of wins associated with the account
  + draws – integer representing the total number of draws associated with the account
  + losses –integer representing the total number of losses associated with the account
  + monster\_count – HashTable with String, Integer key-value pairs representing the number of times the account has played a game as each monster
* Methods
  + main – Unit test the Player Class
    - Input – String[] args
    - Output – None
    - Steps –
      * Create new instance of Player Class and assign to variable
      * Call showPlayer() method
      * Call showStats() method
  + Player – Default constructor, set account name and password, set W/D/L to 0, establish monster\_count HashTable for tracking monster plays
    - Input – None
    - Output – None
    - Steps –
      * Call accountName() method to set name attribute
      * Call accountPassword() method to set password attribute
      * Set wins, draws, and losses to 0
      * Create monster\_count HashTable
      * Populate monster\_count HashTable (call populateMonsterCounter() method)
  + compareTo – Overwrite compareTo method inherited from Comparable Interface so that Players can be sorted with built-in Java method
    - Input – opponent, an instance of the Player Class
    - Output – int -1, 0, or 1, representing less than, equal to, greater than respectively
    - Steps –
      * If Player’s monster’s d\_twenty is equal to opponent’s monster’s d\_twenty, return 0
      * Return 1 if greater than
      * Return -1 if less than
  + populateMonsterCounter – Fill HashTable monster\_count attribute with relevant key-value pairs
    - Input – None
    - Output – None
    - Steps –
      * Add key-value String int pairs consisting of a monster name and 0 (representing 0 plays as the monsters)
  + leastPlayedMonster – Find which monster the Player has played as least
    - Input – None
    - Output – String, the name of the monster and the number of plays
    - Steps –
      * Set monster variable equal to none
      * Set count variable equal to zero
      * Iterate through monster\_count HashTable
      * If monster variable equals none or current monster’s play count is less than the count variable, set monster variable to current monster and count to current monster’s play count
      * Return monster and count variables as String
  + mostPlayedMonster – Find which monster the Player has played as most
    - Input – None
    - Output – String, the name of the monster and the number of plays
    - Steps –
      * Set monster variable equal to none
      * Set count variable equal to zero
      * Iterate through monster\_count HashTable
      * If monster variable equals none or current monster’s play count is greater than the count variable, set monster variable to current monster and count to current monster’s play count
      * Return monster and count variables as String
  + accountName – Set account name for Player instance
    - Input – None
    - Output – None
    - Steps –
      * Ask user to input account name
      * Read input
      * Assign input to name attribute
  + accountPassword – Set password for Player instance
    - Input – None
    - Output – None
    - Steps –
      * Ask user to input password
      * Read input
      * Assign input to password attribute
  + showStats – Print account statistic such as W/D/L and most/least played monsters (for use outside of games)
    - Input – None
    - Output – None
    - Steps –
      * Print Player attributes to console
  + showPlayer – Print Player account name + current Monster statistics such as health, energy, victory points, location, etc. (for use inside of games)
    - Input – None
    - Output – None
    - Steps –
      * Print Player and Monster attributes to console
  + resolveDice – Take actions based on the dice the Player has rolled
    - Input – Dice[], an array of six Dice objects passed from Game object
    - Output – None
    - Steps –
      * Iterate through Dice
      * If a die’s keep status is true, do nothing
      * If a die’s value is a digit, call resolveDigits() method to check for 3 of a kinds and award victory points
      * If a die’s value is energy, increase Player’s monster’s energy attribute by 1
      * If a die’s value is heal, call resolveHeal() method to heal monster
      * If a die’s value is smash, call resolveSmash() method to damage opponents
      * If smash damage was applied, check if an opponent is the King of Tokyo
      * If so, allow them to yield Tokyo and have Player’s monster enter Tokyo
  + resolveDigits – Count the number of integer Dice values rolled and award victory points
    - Input – Dice[], an array of six Dice objects passed from Game object
    - Output – None
    - Steps –
      * Initialize counter variables for 1’s, 2’s, and 3’s
      * Iterate through Dice
      * If die value is 1, 2, or 3, increment the respective counter and change the die’s keep\_status to true
      * Call scoreDigits() method for each counter
  + scoreDigits – Award victory points based on the number of integer Dice values rolled
    - Input – a counter, representing the number of a specific digit that has been rolled, and a factor, representing the value of that digit (i.e. count\_ones, 1)
    - Output – None
    - Steps –
      * If the counter is greater than or equal to 3, award victory points for 3 of a kind + 1 victory point for each additional die of that value
      * Print feedback to Players about victory points scored
  + resolveHeal – Check Player’s monster’s location and current health and increment health as appropriate
    - Input – None
    - Output – None
    - Steps –
      * If monster’s health is less than 10 and monster is outside of Tokyo, increment monster’s health
      * Otherwise, notify Player that heal cannot be applied
  + resolveSmash – Count the number of Smash Dice rolled and apply the damage to opponents as appropriate
    - Input – Dice[], an array of six Dice objects pass from the Game object
    - Output – Player instance representing the Player who’s monster controls Tokyo, used to allow Player to yield
    - Steps –
      * Initialize counter variable
      * Iterate through Dice
      * If die value is Smash, increment counter and change die’s keep\_status to true
      * Call applySmash() method
  + applySmash – Apply Smash damage to opponent monsters as appropriate
    - Input – integer count, representing the number of Smash dice rolled
    - Output – Player instance representing the Player who’s monster controls Tokyo, used to allow Player to yield
    - Steps –
      * Iterate through ArrayList of opponent Players
      * If opponent’s monster’s location is not equal to the player’s and the opponent’s monster’s health is greater than 0, attack the monster and apply damage
      * If opponent’s monster’s location is Tokyo and they die from the previous attack, change their monster’s location to “Deceased” and move Player’s monster into Tokyo
      * Return Player instance that controls Tokyo (this or opponent Player)

**Monster Class:**

* Attributes
  + name – String representing the name of the Monster; Monsters are functionally identical, the name is chosen by the Player and is essentially a skin
  + health – an integer less than or equal to 10 representing the Monster’s current health
  + victory\_points – an integer less than or equal to 20 (actually, can probably go up to 27 in a single turn before the game would end) representing the Monster’s current Victory Points
  + energy – an integer representing the Monster’s current amount of Energy; currently meaningless without the Card System active
  + location – String representing the location of the Monster, can be “Outside”, “Tokyo”, or “Deceased”
  + d\_twenty – an instance of the TwentyDice Class that is rolled for turn order at the beginning of the game
* Methods
  + main – Unit test the Monster Class
    - Input – String[] args
    - Output – None
    - Steps –
      * Create new instance of Monster Class and assign to variable
      * Call showMonster() method
  + Monster – Default constructor
    - Input – None
    - Output – None
    - Steps –
      * None
  + showMonster – Show the current stats of the Monster, such as name, location, health, energy, and victory points
    - Input – None
    - Output – None
    - Steps –
      * Print Monster attributes to console
  + checkHealth – Check that the monster’s health is above zero
    - Input – None
    - Output – boolean
    - Steps –
      * If monster’s health is greater than 0, return true
      * Otherwise, change monster’s location to “Deceased”
      * Return false
  + checkVictoryPoints – Check if a Player has won the game through victory points
    - Input – None
    - Output – boolean
    - Steps –
      * If monster’s victory points are greater than 19, return true
      * Otherwise, return false
  + inTokyo – Check if a Player starts their turn in Tokyo
    - Input – None
    - Output – None
    - Steps –
      * If monster’s location is Tokyo, print feedback to Players
      * Award monster 2 victory points
  + enterTokyo – Monster enters Tokyo
    - Input – None
    - Output – None
    - Steps –
      * Print feedback to Players that monster has entered Tokyo
      * Change monster’s location to Tokyo
      * Award monster 1 victory point
  + yieldTokyo – Monster yields Tokyo
    - Input – None
    - Output – boolean
    - Steps –
      * If monster’s location is Tokyo, ask if they’d like to yield
      * If yes, print feedback to Players and change monster location to “Outside”
      * Return true
      * Otherwise, return false

**AbstractDice Abstract:**

* Attributes
  + value – an integer between 0 and the number of sides minus 1 used to store the value of the current roll
  + keep\_status – boolean, used for keeping particular dice while rerolling others
  + dict – a HashTable that can use the die value as a key and return a String that represents it
  + sides – an integer representing the number of sides the die has
* Methods
  + ABSTRACT populateDict – Fill the HashTable dict attribute with integer, String key-value pairs that are specific to the dice type
  + roll – Assign a random value to the die
    - Input – None
    - Output – None
    - Steps –
      * Get a random integer between 0 and the number of sides of the die minus 1
      * Assign the random integer to the value attribute
  + getValue – Return the String value of the die
    - Input – None
    - Output – None
    - Steps –
      * Use the integer value of the die as a key in the HashTable
      * Return the corresponding String value

**Dice Class (Extends AbstractDice):**

* Methods
  + main – Unit test the Dice Class
    - Input – String[] args
    - Output – None
    - Steps –
      * Create new instance of Dice Class and assign to variable
      * Roll the die
      * Output the value
  + Dice – Default constructor, define number of sides the die has and populate its HashTable
    - Input – None
    - Output – None
    - Steps –
      * Assign integer to sides attribute (define number of sides the die has)
      * Call populateDict() method to fill HashTable with appropriate key-value pairs
  + populateDict – Overwrite method inherited from AbstractDice Class
    - Input – None
    - Output – None
    - Steps –
      * Assign an integer key to a String value for each side of the die

**TwentyDice Class (Extends AbstractDice):**

* Methods
  + main – Unit test the TwentyDice Class
    - Input – String[] args
    - Output – None
    - Steps –
      * Create new instance of TwentyDice Class and assign to variable
      * Roll the die
      * Output the value
  + Dice – Default constructor, define number of sides the die has and populate its HashTable
    - Input – None
    - Output – None
    - Steps –
      * Assign integer to sides attribute (define number of sides the die has)
      * Call populateDict() method to fill HashTable with appropriate key-value pairs
  + populateDict – Overwrite method inherited from AbstractDice Class
    - Input – None
    - Output – None
    - Steps –
      * Assign an integer key to a String value for each side of the die

**UML**

