CIS*3260 F21 – [Individual] Assignment 1

Purpose

- Implement classes and methods in Ruby from a given (simple) type/interface design provided
- The system being designed models a player with a cup full of dice/coins;
  the cup is thrown and the results recorded

Main idea behind the system to be built

- Each "player" has a bag and a cup
- Dice and coins are created using a class method, and placed in a player's bag
- Dice/coins can be removed from a bag (using a "selection description" described below)
  and placed in a cup
- When moving dice/coins they are placed in a "hand" object, which functions as a stack/queue etc.
  (i.e. is iterable); the implementation is your choice
- Dice/coins are randomized by "throwing" the cup
  - This returns a "throw" object which holds the results rolls/flips of all the dice/coins in the cup
  - Note: the items remain in the cup
    - only the results are recorded (however you want to record them)
- The items in the cup should be emptied back into the player's bag
- Each player stores the result of each throw, and the player can be asked for the results and simple statistical summaries of the throws

How to understand the requirements below

You are coding "support" classes, not a program

- The classes described below is not a program with a "mainline"
- Rather they comprise a "class library" that can be used by a script or irb session to support a "game" with players that use coins and dice in order to play

How to read the class descriptions provided below

- You are not being given complete descriptions of the classes and methods to be implemented
- For example, no instance variables are specified
- In effect, you are being supplied with the class interface,
  not the complete implementational details
- This way you have complete flexibility to design the internals of the class as you see fit

Testing the class behaviour

- To prove to us that the classes work, you must create a script that utilize the classes and methods in the library you are creating
- This script is to be called 'testgames.rb'
- These tests should be based on your Use Cases that you will be creating and submitting
  - When creating your Use Cases, try to think of scenarios that naturally touch on as much of the classes and methods as they can, to minimize the number of Use Cases you need to write
- Technically, the Use Cases should be driving your testing, not the other way around
  - However, sometimes to thoroughly test a class, you might create a "use case" that is a bit artificial … just there to test out a part of the class that needs testing
  - This is fine as use cases are usually developed for a specific application, not a support library as we are doing here

The classes and methods to be implemented in Ruby

Randomizer Classes

Randomizer *an "abstract class" used for polymorphism and to store common methods*

- randomize() *both randomizes and returns the randomizer itself (for method chaining)*
- result() *returns the result of the randomization, or nil if never randomized*
- randomize_count() *returns the number of randomizations performed*
- reset() *sets the result to nil and number of randomizations performed to 0*

Coin *<inherits from Randomizer>*

- initialize(denomination:Enum) *constructor (i.e. Coin.new(denomination))*
- denomination() *returns the denomination of the coin (does not set it)*
- flip() *flips the coin and returns the "flipped" coin (for method chaining)*
- sideup() *returns :H or :T (the result of the last flip) or nil (if no flips yet done)*

Die *<inherits from Randomizer>*

- initialize(sides:Int, colour:Enum) *constructor (i.e. Die.new(sides, colour))*
- colour() *returns the colour of the die (does not set it)*
- sides() *returns the number of sides (does not set it)*
- roll() *randomizes and returns the "rolled" die (for method chaining)*
- sideup() *returns 1..sides or nil*

*Note1: Int and Enum are only indications of the arg types expected by the method.
They will not be present in your code as Ruby is Duck typed.
UML type notation has been used, with the type following the arg name, separated by a :*

*Note2: Enum is not actually a class; it is a duck-type-style interface, i.e. a protocol.
E.g. colour is consistently represented by symbols such as :red and :blue;
There is no class holding or enforcing these values
(although they can be error checked against by the methods that accept them, if you wish).
All enumeration protocols are listed in the "Enumerations" section later in the assignment.*

RandomizerContainer Classes

RandomizerContainer *an "abstract class" used for polymorphism and to store common methods*

- store(r:Randomizer) *stores a randomizer in the container*
- move_all(rc:RandomizerContainer) *get each randomizer in rc & store it in self*
- empty() *removes all members of the container*

Bag *<Inherits from RandomizerContainer>*

when store() or move_all() invoked, Bag makes sure that all randomizers added to the bag are reset

- select(description:Hash, amt=:all)
  selects items from Bag based on the description and returns a Hand object that is holding the items up to the number entered into amount (if you want all items, supply the symbol :all instead of a number)
- empty()
  empties all items from the Bag into a Hand, which is returned

Hand *<Inherits from RandomizerContainer>*

- next()
  removes and returns the last objected added
  if no objects stored, return nil
- empty()
  returns nil (items are "dropped on the ground") i.e. the pointers to the contained objects are lost (and the objects will be garbage collected by the system)

Cup *<Inherits from RandomizerContainer>*

- throw()
  each randomizer in the cup is rolled or flipped
  virtual copies of all randomizers are stored in a newly created Throw object, which is returned
- load(hand:Hand)
  enters each randomizer from a Hand (synonym of move_all())
- empty()
  returns a Hand object to be returned to the bag, and leaves the cup empty

Note: All RandomizerContainer objects are created empty. Therefore they are just called with 'new' e.g. Cup.new

High Level Classes

Throw

- initialize(cup: Cup)
  constructor (i.e. Throw.new(cup))
- description(description:Hash)
  Stores a description in the Throw from which the "randomizer" objects in the Throw can be selected when computing the results, tally or sum
- results()
  Returns an array containing the "side-up" values of the randomizers recorded in the Throw.
  Only include the values from randomizers that match the description stored in the Throw.
  If description() has not yet been called, return the results from all randomizers
- tally()
  Counts the items in the Throw that match the description and returns the value
  If description() has not yet been called, count all randomizers
- sum()
  totals the value of the randomizer items in the Throw that match the description, where the value equals the number that is "up" (for coins, :H = 1 and :T = 0), and returns the value
  If description() has not yet been called, total the values across all randomizers

Player

- initialize(name:String)
  constructor (i.e. Player.new("Gandalf the Grey"))
- name()
  returns the name of the player (does not set it)
- store(item:Randomizer)
  stores the item in the player's bag
- move_all(rc:RandomizerContainer)
  gets each item in rc and stores it in the player's bag
- load(description:Hash = { })
  loads items from the player's bag to the player's cup based on the description
- throw()
  throws the (previously loaded) cup, and replaces the items in the cup to the bag
  the throw is both returned and stored internally
- clear()
  clears all stored throws
- tally(description:Hash = { })
  sets the description, and calls tally() on each stored throw and returns each of the values within a single array
- sum(description:Hash = { })
  sets the description, and calls sum() on each stored throw and returns the combined values as an array
- results(description:Hash = { }, throw:Int = 0)
  sets the description and returns the values as an array from given Throw,
  where the last Throw is "throw=0", the throw before is "throw=1", etc.
  If a throw is requested that doesn't exist (too far back in time and never occurred), return nil

Enumerations

- enum used for Randomizer items = :coin, :die
- enum used for coin denominations = 0.05, 0.10, 0.25, 1, 2
- enum used for die colours = :red, :green, :blue, :yellow, :black, :white
- enum used for coin sides = :H, :T

Description examples

- { item: :die, sides: 4, colour: :yellow, up: 4 }
- { item: :die, colour: :red }
- { colour: :red } ← selects same as above; Coins do not have colour, so can't be :red
- { item: :die, sides: 6 }
- { item: :die }
- { item: :coin, denomination: 0.25 }
- { item: :coin, up: :H }
- { item: :coin }
- { } ← selects everything

Note: an "impossible" description just means that you are not selecting any of the objects

Examples:, all of the following will not select any items

{ item: :coin, sides: 6 }	*as a coin can only have 2 sides,*
{ item: :die, sides:6, up: 4 } in a bag	*as 'up' there always equals nil*
{ item: nil }	*as nil is not a :coin or :die*

Grading

- [6 pts] The ruby code i.e. how much functionality does your written code try to encapsulate
- Create and run tests
  - [6 pts] Create use cases to base your tests on
    - i.e. reverse engineer use cases based on the type design provided
  - [12 pts] Execute your test cases and report the results
    - i.e. for each test
      - state purpose of test,
      - state expected output
      - report actual output
      - report pass/fail
    - Your tests, cumulatively, must test at least each method of each class
- [6 pts] Passes our tests
- [5 pts] OO Style
- [1 pt] Ruby Style

Submit in Courselink

- Each class is kept in its own file (ClassName.rb)
- Provide a script A1_Classes.rb that loads all your classes by successively calling
  ```
  require ClassName.rb
  ```
  on the successive class files; one ‘require’ per line
- The script demonstrating that your classes are working properly, should be called testgames.rb
  You should also require A1_Classes.rb at the beginning of the script
- Provide a .pdf or .txt file describing your use cases for the above testing script
- Provide a file that has the results of your tests (see grading),
  called mytest.pdf or mytest.txt
- Archive all files together (e.g. as a .zip file)