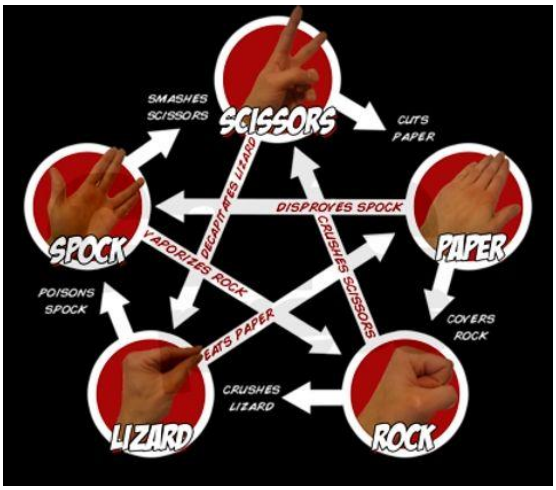
Universidad Simón Bolívar
Departamento de Computación
y Tecnología de la Información
CI-3661: Laboratorio de
Lenguajes de Programación

Prof. Fernando Torre Mora
Prof. Carlos Gomez

Proyecto 3 Septiembre – Diciembre 2018

Piedra, Papel, Tijeras, Lagarto, Spock

1 Introducción



"Piedra, Papel, Tijeras, Lagarto, Spock" es una popular extensión del juego de manos "Piedra, Papel o Tijeras" inventada por Sam Kass y Karen Bryla en algún momento de los años 90, y popularizada por la serie *The Big Bang Theory* en 2008. El juego consiste de cinco elementos que pueden ser formados con la mano:

- Piedra, representada por un puño cerrado
- Papel, representado por todos los dedos extendidos
- Tijera, representado por dos dedos extendidos
- Lagarto, representado juntando las puntas de los dedos
- Capitán Spock, representado separando los dedos medio y anular

En un round, los jugadores deben simultáneamente realizar una jugada (mostrar un elemento con la mano). Se decide el ganador del round de la siguiente manera:

- Si se muestra piedra
  - La piedra le gana a las tijeras (decimos que la piedra "aplasta" a las tijeras)
  - La piedra le gana al lagarto (decimos que la piedra "aplasta" la cabeza del lagarto)
  - La piedra pierde ante el papel (decimos que la piedra es "envuelta" por el papel)
  - La piedra pierde ante Spock (decimos que la piedra es "desintegrada" por el rayo desintegrador de Spock)
- Si se muestra papel
  - El papel le gana a Spock (decimos que el papel académico "refuta" la lógica de Spock)
  - El papel pierde ante las tijeras (decimos que el papel es "cortado" por las tijeras)
  - El papel pierde ante el lagarto (decimos que el papel es "comido" por el lagarto)
- Si se muestran Tijeras
  - Las tijeras le ganan al lagarto (decimos que las tijeras "decapitan" al lagarto)
  - Las tijeras pierden ante Spock (decimos que las tijeras son "aplastadas" por Spock)
- Si se muestra el Lagarto
  - El lagarto le gana a Spock (decimos que el lagarto "envenena" a Spock con la mordida)

2 Requerimientos del programa

Usted debe modelar el juego con una estructura específica usando Ruby aprovechando el polimorfismo de Ruby.

2.1 Las Jugadas

Utilizaremos la clase `Jugada` para representar la noción de la jugada ejecutada por un jugador, siendo necesario contar con las subclases `Piedra`, `Papel`, `Tijera`, `Lagarto` y `Spock`, para representar los elementos específicos. Es necesario implantar los métodos:

- `to_s` para mostrar el invocante como un `String`.
- `puntos(j)` que determine el resultado de la jugada entre el invocante y la jugada `j`, correspondiente al contrincante, que es recibido como argumento. El resultado de `puntos` debe ser una tupla que representa la ganancia en puntos resultado de la jugada:
  - el primer elemento de la tupla representa la ganancia del invocante,
  - mientras que el segundo elemento representa la ganancia del contrincante.

  Así, la tupla resultante debe ser [1,0], [0,1] o [0,0] dependiendo de los movimientos involucrados.

2.2 Las Estrategias

Cada jugador sería representado por un objeto de la clase `Estrategia`. La clase `Estrategia` permite generar la siguiente jugada del jugador, quizás aprovechando las jugadas anteriores propias, del rival, o ambas, como base de referencia. Toda estrategia debe proveer los métodos:

- `prox(m)` que genera la próxima `Jugada` usando como información adicional, si le conviene, la `Jugada j` suministrada como argumento. Note que este método debe retornar un objeto en alguna de las clases `Piedra`, `Papel`, `Tijera`, `Lagarto` ó `Spock` – es un error retornar un `String`.
- `to_s` para retornar el invocante como `String`. Siempre debe mostrar el nombre simbólico, pero debe estar especializada para mostrar los parámetros de configuración específicos de cada estrategia discutida más abajo.
- `reset` para llevar la estrategia a su estado inicial, cuando esto tenga sentido.

Usted debe implantar al menos las siguientes especializaciones de Strategy:

- `Manual`, que espera a que el usuario indique la siguiente jugada a jugar
- `Uniforme`, construida recibiendo una lista de movimientos posibles y seleccionando cada movimiento usando una distribución uniforme sobre los movimientos posibles, i.e.

  ```
  r = Uniforme.new([ :Piedra, :Papel, :Tijeras, :Lagarto, :Spock ])
  ```

  Al construir una instancia de esta estrategia, es necesario eliminar duplicados y verificar que haya al menos una estrategia en la lista, en caso contrario emitir una excepción describiendo el error.

- `Sesgada`, construida recibiendo un mapa de movimientos posibles y sus probabilidades asociadas, de modo que cada jugada use una distribución sesgada de esa forma. Al construir una instancia de esta estrategia, es necesario eliminar duplicados y verificar que haya al menos una estrategia en el mapa; en caso contrario, emitir una excepción describiendo el error. Las probabilidades asociadas a cada tipo de movimiento serán números enteros, es decir:

```
b = Sesgada.new(
      { :Piedra => 2, :Papel => 5, :Tijeras => 4,
        :Lagarto => 3, :Spock => 1
      })
```

Resultando en probabilidades $\frac{2}{15}$, $\frac{1}{3}$, $\frac{4}{15}$, $\frac{1}{5}$ y $\frac{1}{15}$ respectivamente.

- `Copiar`, cuya primera jugada es definida al construirse, pero a partir de la segunda ronda siempre jugará lo mismo que jugó el contrincante en la ronda anterior.
- `Pensar`, cuya jugada depende de analizar las frecuencias de las jugadas hechas por el oponente hasta ahora. La estrategia debe recordar las jugadas previas del oponente, y luego decidir de la siguiente forma:
  - Sean `r`, `p`, `t`, `l` y `s` la cantidad de veces que el oponente ha jugado Piedra, Papel, Tijera, Lagarto y Spock, respectivamente.
  - Se genera un número entero al azar n entre 0 y $r + p + t + l + s - 1$
  - Se jugará
    - Piedra si $n \in [0, r)$
    - Papel si $n \in [r, r + p)$
    - Tijera si $n \in [r + p, r + p + t)$
    - Lagarto si $n \in [r + p + t, r + p + t + l)$
    - Spock si $n \in [r + p + t + l, r + p + t + l + s)$

Notará que varias de las estrategias requieren el uso de números al azar. La librería `Random` provista por Ruby tiene toda la infraestructura necesaria. Con el propósito de poder evaluar de manera semiautomática sus soluciones, es necesario que todos los generadores de números al azar tengan exactamente la misma semilla, para ello declare una constante de clase en `Estrategia` con el valor 42, a ser utilizada cada vez que necesite una semilla.

2.3 El Juego

Un juego ocurre entre dos jugadores, cada uno definido por su estrategia. La clase `Partida` debe construirse recibiendo un mapa con los nombres y estrategias de los jugadores

```
m = Partida.new( { :Deepthought => s1, :Multivac => s2 } )
```

donde `s1` y `s2` son instancias de alguna de las estrategias. Es necesario verificar que hay exactamente dos jugadores y que en efecto se trata de estrategias, generando excepciones descriptivas del problema cuando esas condiciones no se cumplan.

Tendremos interés en observar el desarrollo del juego de varias maneras:

- `rondas(n)`, con `n` un entero positivo, debe completar `n` rondas en el juego y producir un mapa indicando los puntos obtenidos por cada jugador y la cantidad de rondas jugadas.
- `alcanzar(n)`, con `n` un entero positivo, debe completar tantas rondas como sea necesario hasta que alguno de los jugadores alcance `n` puntos, produciendo un mapa indicando los puntos obtenidos por cada jugador y la cantidad de rondas jugadas.
- `reiniciar`, debe llevar el juego a su estado inicial.

Ha de ser posible continuar un juego en curso, es decir si se ejecutara

```
m.rondas(10) # Se ejecutan 10 rondas
m.rounds(20) # Se ejecutan 20 rondas adicionales
r = m.upto(100) # Se ejecutan las rondas hasta que alguno llegue a 100
{ :Multivac => 84, :Deepthought => 100, :Rounds => 238 }
```

el juego producirá los resultados de las primeras 10 rondas, a los cuales acumularía los resultados de las siguientes 20 rondas y por último continuaría hasta que algún jugador acumule 100 puntos.

3 Requerimientos de I/O

Se espera que desarrolle una sencilla interfaz con Ruby Shoes que muestre la forma de la mano correspondiente a la jugada en ese momento. Esto debe hacerse por medio de una imagen que se muestre al momento de la jugada. La interfaz debe contener botones que permitan seleccionar la siguiente jugada (Piedra, Papel, Tijeras, Lagarto, Spock) si la estrategia es `Manual`. Tome en cuenta que se puede especificar una partida de dos jugadores (`Manual` vs. `Manual`).

4 Requerimientos de la entrega

Debe entregar su código original de Ruby en un archivo comprimido en el Moodle de la materia en la sección marcada como "  Proyecto 3" antes del 24 de diciembre de 2018. Sólo deberá efectuar una entrega por grupo.

Debe entregar un informe que indique cómo correr su programa, y cualquier decisión de diseño relevante. El informe debe tener introducción y conclusión

Todos sus códigos deben estar debidamente documentados en RDoc.

5 Evaluación

El proyecto tiene una ponderación de 17 puntos. Se asignarán

- 8 puntos por código
  - 1 punto por la clase `Jugada`
  - 6 puntos por la clase `Estrategia`
    - 1 punto por cada estrategia
    - 1 punto por los demás métodos requeridos de la clase
  - 1 pt por la clase `Partida`
- 4 puntos por ejecución
  - 1 punto por poder jugar contra la computadora al especificar una estrategia
  - 1 punto por poder simular una partida entre dos estrategias de la computadora
  - 2 puntos por su interfaz gráfica
    - 1 punto por mostrar la forma de la mano
    - 1 punto por tener botones funcionales para completar la ronda utilizando la interfaz gráfica
- 3 puntos por documentación (1 pt por cada clase requerida)
- 2 puntos por su informe
  - 1 punto por sus instrucciones de cómo jugar
  - 1 punto por sus decisiones de diseño

Se asignará un punto adicional si la interfaz permite seleccionar la estrategia, y otro punto adicional si permite seleccionar la manera de desarrollar la partida.

El programa debe correr sin errores.