Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación
IIC2233 — Programación Avanzada 2020-2

8 de Octubre de 2020
Actividad Sumativa

Actividad Sumativa 02

Threading

Entrega

- **Lugar:** En su repositorio privado de GitHub, en la **carpeta** `Actividades/AS02/`
- **Hora del *push*:** 16:50

Importante: Antes de comenzar, comprueba que Git este funcionando correctamente en tu repositorio privado. Para esto, **sube los archivos base de la actividad de inmediato** (*add*, *commit*, *push*). Se espera que en esta actividad (así como en las demás actividades y tareas) utilices Git a lo largo de **todo tu desarrollo** como una herramienta, no sólo como un método de entrega. Es por esto que recomendamos enfáticamente que vayas subiendo tus cambios constantemente (***push***), ya que **problemas de último minuto** relacionados con la entrega y Git **no serán considerados**.

Introducción

Es viernes por la noche, otro día de encierro, necesitas despejarte y descansar después de una larga semana estudiando Programación Avanzada (y otros ramos menos importantes). Decides que quieres jugar algún videojuego, por lo que abres tu carpeta con nombre *trabajos importantes para la universidad* y revisas los juegos que tienes. – ¿**DCCombate Naval**? Ya me tiene aburrido ¿**DCCriaturas**? Estoy muy grande para esas cosas.

No... necesitas algún juego con más interacción. Por lo que entras al prestigioso canal de Discord del ramo y preguntas por *misceláneo*:

– ¿Su Among Us?

Lógicamente, armas un grupo en menos de un minuto y te preparas para jugar, estás emocionado, hoy si que si ganarás todos los juegos y engañarás como una mente maestra. Expectante le das al botón jugar y, para tu sorpresa (o quizás para sorpresa de nadie), no puedes entrar ya que los servidores del juego están caídos.

Aburrido de la ineficiencia de los servidores de Among Us, has decidido utilizar tus conocimientos de *threading* para programar la nueva sensación, **DCCrewmates**.

Flujo del juego

El juego consiste en una nave con cierta cantidad de tripulantes, entre quienes existe un perverso impostor que busca evitar que los tripulantes arreglen la nave y hagan sus tareas (entre ellas, `git add`, `git commit`, `git push`). La misión de los tripulantes es arreglar la nave lo más rápido posible (lo cual consiguen realizando sus tareas asignadas). Por otro lado, el impostor buscará evitar que los tripulantes arreglen la nave, engañándolos, haciendo pensar a sus compañeros que está realizando tareas cuando en realidad está saboteando y asesinando a los tripulantes en cuanto tenga oportunidad.

El modulo principal de ejecución del programa es `dccrewmates.py`, el cual se encargará de instanciar los tripulantes y al impostor con un color aleatorio entre los disponibles.

Una vez iniciado el juego, los tripulantes se dedicarán a hacer tareas, mientras que el impostor intentará asesinar al grupo. El juego termina cuando los tripulantes logran cumplir todas sus tareas o cuando el impostor mata a todos los tripulantes. Al ocurrir una de estas dos situaciones, se evaluará si los tripulantes cumplieron todas sus tareas y se les otorgará la victoria o derrota dependiendo del resultado. Finalmente, se imprime en pantalla quién ganó y se revela el color del impostor.

Archivos

Deberás trabajar dentro de dos archivos:

- `dccrewmates.py`: Contiene la clase **DCCrewmates**, que **debes completar**. Este será el archivo principal que debes ejecutar.
- `tripulantes.py`: Este archivo contiene a la clase **Tripulante** y la clase **Impostor**, que deben ser **completadas**.

Los siguientes archivos **no deben ser modificados**

- `parametros.py`: Este archivo contiene los parámetros que se utilizarán durante el juego. Todos los parámetros mencionados **deben** ser utilizados durante el programa mediante importación. **Solo se deben modificar los valores de los parametros si deseas probar tu código.**
- `funciones.py`: Este archivo contiene funciones externas que serán de utilidad para la implementación de tu código.
- `sabotajes.csv`: Este archivo contiene los nombres de los sabotajes que puede realizar el impostor.
- `tareas.csv`: Este archivo contiene los nombres de las tareas que pueden realizar los tripulantes, o bien que puede fingir el impostor.

Parte I. DCCrewmates

Esta clase será el centro del juego, la cual se encargará de instanciar, iniciar y verificar los *threads* de los tripulantes. La clase hereda de `Thread`, por lo cual, es un *thread* personalizado que se debe completar. Posee los siguientes atributos:

- `self.impostor`: Una instancia de la clase impostor.
- `self.tripulantes`: Una lista de instancias de la clase Tripulantes.
- `self.evento_termino`: Es una instancia de la clase `Event` que indicará cuando todos los *threads* de tripulantes hayan terminado.
- `self.evento_sabotaje`: Instancia de la clase `Event` que indica cuando hay un sabotaje en proceso.
- `self.diccionario_tareas`: Es un diccionario con todas las tareas que deben realizar los tripulantes durante el juego para ganar. El diccionario tendrá por llaves los nombres de las tareas y el valor para cada tarea será otro diccionario de la siguiente forma:

```
1       {
2           "lock": Lock(),
3           "realizado": False,
4           "nombre": "NOMBRE_TAREA"
5           }
```

  El objetivo de completar una tarea es llegar a cambiar su estado `"realizado"`, por lo cual, debes utilizar el `Lock` correspondiente para que solo pueda trabajar un tripulante a la vez sobre el diccionario de cada tarea.

Para poder ejecutar el programa se debe completar el siguiente método:

- `def run(self)`: Este método permitirá la ejecución del programa, por lo tanto, debes seguir estas instrucciones:
  - Se debe llamar al método `self.asignar_tripulantes()` para definir las instancias de los tripulantes e impostor que compondrán el juego. Estas instancias quedarán guardadas en el atributo `self.tripulantes` y `self.impostor` respectivamente.
  - Se debe iniciar los *threads* de todos tripulantes y del impostor, para luego esperar que todos los tripulantes terminen su ejecución.
  - Una vez que todos los tripulantes hayan acabado, se debe activar el evento `self.evento_termino`.
  - Ahora debes esperar a que el impostor termine su ejecución.
  - Luego utilizando el diccionario `self.diccionario_tareas` debes comprobar que todas las tareas hayan sido realizadas con su llave `"realizado"`. En caso de que se hayan realizado todas, se debe anunciar como ganadores a los tripulantes y en caso contrario anunciar al impostor.
  - En cualquier caso debes anunciar al final, cual era el color del impostor.

Parte II: Clase Tripulante

La clase Tripulante se encuentra en el archivo `tripulantes.py`. Esta clase se encargará de realizar las acciones de cada uno de los personajes.

Puedes probar su implementación ejecutando el archivo `tripulantes.py` una vez ya completado

Esta clase será un *thread* personalizado, por lo que se hereda de `Thread` e incluye los atributos:

- `self.color`: Un `str` que representa el color del tripulante.
- `self.tareas`: Una `lista` que contiene `strings` con las tareas por hacer del tripulante
- `self.diccionario_tareas`: Diccionario con la información de las tareas obtenido de la clase `DCCrewmates`.
- `self.esta_vivo`: Un `bool` que será `True` si es que el tripulante está vivo y `False` de lo contrario.
- `self.evento_sabotaje`: Es una instancia de `Event()` que indica si es que hay un sabotaje activo en la nave. Este evento será activado al iniciar un sabotaje y desactivado una vez que se haya solucionado. En Notas se dejó un consejo sobre como verificar si este elemento está activo.

Además deberás implementar los siguientes métodos:

- `def run(self)`: Contiene la lógica de cada `Tripulante`. El tripulante se mantiene funcionando siempre que se cumplan las siguientes condiciones de manera simultánea (ambas a la vez):
  - El tripulante se encuentra vivo
  - Al tripulante aún le quedan tareas por hacer

  Durante su ejecución, el tripulante intentará realizar tareas con `self.hacer_tarea()`, siempre que no haya un evento sabotaje en curso. Una vez completada la tarea, el tripulante tomará un descanso de `TIEMPO_ENTRE_TAREAS` antes de intentar completar sus tareas restantes.

  Cuando un sabotaje esté en curso el tripulante intentará arreglarlo con `PROB_ARREGLAR_SABOTAJE` de probabilidad. Si la probabilidad se cumple, se deberá ejecutar `self.arreglar_sabotaje()`; si fracasa deberá esperar `TIEMPO_ENTRE_TAREAS` para volver a intentarlo con la misma probabilidad.
- `def hacer_tarea(self)`: Se ejecutará cada vez que se encuentren las condiciones para que el tripulante realice una tarea. El tripulante elige una tarea al azar desde `self.tareas` y obtiene su información desde `self.diccionario_tareas`. Una tarea no puede ser realizada por más de un tripulante al mismo tiempo, por lo tanto, si una tarea está en ejecución por otro tripulante, el tripulante actual debe esperar hasta que la tarea sea liberada y tomarla durante su "realización" utilizando el mismo `Lock` de la tarea.

  Para completar una tarea, se debe determinar un tiempo total de ejecución aleatorio dentro del rango `TIEMPO_TAREAS` y distribuirlo en 5 iteraciones hasta completarlo. Entre cada iteración se debe imprimir el progreso de avance con la función `print_progreso()` (Ver sección Funciones).

  El tripulante puede ser asesinado por el impostor entre cada iteración de la tarea, por lo que el *thread* debe revisar en cada iteración si el tripulante sigue vivo antes de continuar.

  Una vez completadas las iteraciones y transcurrido el tiempo total, la tarea se deberá remover de la lista `self.tareas` y marcar el valor de `"realizado"` del diccionario de la tarea de este tripulante como `True`.

  Recordar que se complete o no, la tarea se debe liberar para que otro jugador pueda realizarla.
- `def arreglar_sabotaje(self)`: este método se llama cuando un tripulante decide ir a arreglar un sabotaje. Para reparar el sabotaje se debe escoger un tiempo aleatorio dentro del rango `TIEMPO_SABOTAJE` y al igual que en el método anterior, se debe completar en 4 iteraciones, donde en cada una se imprima el progreso con la función `print_progreso()` (Ver sección Funciones). Entre cada iteración

también debes verificar que el tripulante siga vivo. Por último, si al acabar el progreso de desactivación el tripulante sigue vivo, entonces se deberá reparar el sabotaje desactivando el evento `self.evento_sabotaje`.

Se debe imprimir en consola al comienzo del método que el tripulante (color) ha comenzado a reparar el sabotaje y en caso de lograr repararlo, imprimir que ha terminado, para ello puedes utilizar la función `print_anuncio()` (Ver sección Funciones).

Parte III: Clase Impostor

Esta clase será un *thread* personalizado, que hereda de `Tripulante`. Además de los atributos de los tripulante, recibe una lista con las instancias de todos los tripulantes que están participando sin incluir al impostor mismo, y un evento que indica el término del juego.

Posee los atributos:

- `self.tripulantes`: Una `list` con todas las instancias de la clase `Tripulante`.
- `self.evento_termino`: Una instancia de `Event` que indicará el término del juego.
- `self.sabotajes`: Una lista de *strings* con los posibles sabotajes.

En la clase Impostor tendrás que implementar los siguientes métodos:

- `def run(self)`: Contiene la lógica del `Impostor`. Se debe mantener activo mientras queden tripulantes vivos o el evento `self.evento_termino` no haya sido activado. Mientras el *thread* esté activo, el impostor debe elegir una acción mediante la función `elegir_accion_impostor()`de `funciones.py`, y actuar según se indica. Los valores posibles son:
  - `'Matar'`: debe llamar al método `self.matar_jugador()`.
  - `'Sabotear'`: debe llamar al método `self.sabotear()`.
  - `'Esconderse'`: si el impostor está en apuros, se esconderá en los ductos de ventilación. Para ello debes pausar el `thread` por `TIEMPO_ESCONDITE` segundos.

  Luego de realizar una acción, el impostor debe esperar un tiempo de `TIEMPO_ENTRE_ACCIONES` segundos antes de volver a realizar otra.
- `def matar_jugador()`: deberá seleccionar un Tripulante al azar, modificar su atributo `esta_vivo` y eliminarlo de la lista `self.tripulantes`. Una vez hecho esto, debe imprimir un mensaje en la consola avisando que se eliminó un jugador, junto con su color y la cantidad de tripulantes vivos restantes. Para ello puedes utilizar la función `print_anuncio()` (Ver sección Funciones).
- `def sabotear()`: este método permite al impostor iniciar un sabotaje, siempre y cuando no esté otro en curso, lo cual se debe verificar por medio del evento `self.evento_sabotaje`. Cuando se inicia el sabotaje, se deberá:
  - Obtener al azar el nombre de un sabotaje desde la lista `self.sabotajes`.
  - Definir un tiempo entero aleatorio que esté en el rango dado por `TIEMPO_SABOTAJE`.
  - Crear e iniciar un `Timer` que, después del tiempo definido, ejecute `self.terminar_sabotaje()`.
  - Activar el evento `self.evento_sabotaje` e imprimir un mensaje avisando que se inició un sabotaje, utilizando la función `print_sabotaje(nombre_sabotaje)` (Ver sección Funciones).

- `def terminar_sabotaje()`: este método será llamado al acabar el tiempo para reparar el sabotaje, y verifica si los tripulantes lograron arreglarlo. En el caso de que el sabotaje **no** haya sido arreglado, entonces se deberá matar a todos los tripulantes que aun no hayan sido eliminados. Finalmente, se deberá llamar a la función `print_explosion()`.

Funciones

Funciones que **debes** utilizar en tu código, ya sea en **Impostor** o **Tripulante** en cada caso. **No reimplementes las funciones y no uses funciones propias para lograr lo mismo**:

- `def elegir_accion_impostor()`: retorna un `string`, en base a probabilidades, que podrían ser: `'Matar'`, `'Sabotear'` o `'Esconderse'`. Lo debes utilizar en la clase **Impostor**.
- `def print_progreso(color, actividad, progreso)`: Recibe un **string** de un color, un **string** de la actividad que está realizando y un **int** del progreso que corresponde a un porcentaje entre 0 y 100, dependiendo del número de iteración.

```
    for iteracion in range(5):
        print_progreso(self.color, f"Realizando {nombre_tarea}", 25 * iteracion)
```

```
Rojo | Realizando Arreglar los escudos |
Rojo | Realizando Arreglar los escudos |■
Rojo | Realizando Arreglar los escudos |■■
Rojo | Realizando Arreglar los escudos |■■■
Rojo | Realizando Arreglar los escudos |■■■■
```

- `def print_anuncio(color, anuncio)`: Recibe un **string** del color y un **string** del anuncio a mostrar. Puede utilizarse para sucesos importantes, como arreglar un sabotaje o una muerte de un tripulante.

```
    print_anuncio("Café", "Ha comenzado Deslizar tarjeta ID")
    print_anuncio("Negro", "Ha muerto")
    print_anuncio("Moraro", "esta arreglando el sabotaje")
```

```
================== ¡Café Ha comenzado Deslizar tarjeta ID! ==================
============================== ¡Negro ha muerto! ==============================
==================== ¡Morado esta arreglando el sabotaje! ====================
```

- `def print_sabotaje(sabotaje)`: Recibe un **string** de un sabotaje, y se debe implementar cuando el **Impostor** realize un sabotaje.

```
    print_sabotaje("Arreglar el oxígeno")
```

```
********************************************************************************
*****************************>> SONIDO DE ALERTA <<*****************************
                    SABOTAJE EN CURSO: Arreglar el oxígeno
********************************************************************************
```

- `def print_explosion()`: imprime en consola la bomba de fin de juego :O Se debe utilizar cuando no se logra arreglar el sabotaje.

Ya utilizadas y no necesitas volver a utilizar:

- `def cargar_tareas()`: esta función es la encargada de cargar el archivo `tareas.csv`. Retorna una **lista**, con los nombres de cada tarea como un **string** de tipo `'nombre tarea'`. Se utiliza en el `__init__` de **DCCrewmates**.
- `def cargar_sabotajes()`: esta función es la encargada de cargar el archivo `sabotajes.csv`. Retorna una **lista** de todos los sabotajes, cada uno como **string**. Se utiliza en el `__init__` de **Impostor**.

Notas

- Recuerda utilizar `Locks` para los casos en que debas verificar que no existan conflictos cuando más de un *thread* desea acceder a una sección crítica.
- La función `sleep()` del módulo `time` recibe como atributo el tiempo **en segundos** durante los cuales el *thread* actual se detendrá.
- `is_set()` es un método de la clase `Event`, que retorna `True` si el evento está activo (si ya se hizo `Event.set()`). En caso contrario, retorna `False`.

Requerimientos

- (2.50 pts) Clase Tripulante
  - (1.25 pts) Define correctamente el método `run`.
  - (0.75 pts) Define correctamente el método `hacer_tarea`.
  - (0.50 pts) Define correctamente el método `arreglar_sabotaje`.
- (2.25 pts) Clase Impostor
  - (1.25 pts) Define correctamente el método `run`.
  - (0.25 pts) Define correctamente el método `matar_jugador`.
  - (0.50 pts) Define correctamente el método `sabotear`.
  - (0.25 pts) Define correctamente el método `terminar_sabotaje`.
- (1.25 pts) Clase DCCrewmates
  - (0.50 pts) Inicia correctamente los threads.
  - (0.50 pts) Utiliza correctamente el método `join`.
  - (0.25 pts) Define correctamente el ganador.