Informe Event IT Master. Diseño y funciones

Di Capua Valentino

Gianfelice Luciano

Manfredi Francisco

Lopez Emiliano

Pella Joaquín

Introducción:

El programa presentado a continuación fue creado con el propósito de seguir lo planteado en el enunciado de EventIT, siguiendo las reglas de la programación orientada a objetos (POO). De manera resumida, el sistema funciona en base a dos tipos de usuarios con distintas responsabilidades y atributos. El citizen y el admin. Ambos son capaces de realizar sus funciones ingresando a un menú general denominado MainMenu desde el cuál pueden acceder a submenús denominados menú\_administrador (admin), menú\_citizen(citizen), y menu\_amigos(citizen). Adicionalmente, se puede ingresar desde el MainMenu a un mapa con sensores marcando los distintos tipos de eventos.

Polimorfismo: Dado que las clases de usuarios presentan responsabilidades extremadamente distintas, no se han implementado casos de polimorfismo.

Encapsulamiento: Se restringieron las responsabilidades según el tipo de usuario, estas clases son completamente independientes del resto. Los menús solo sirven para acceder a estas clases sin violar el encapsulamiento.

Clases: Cada ente de la realidad en el programa (admin, citizen, zona, listadeciudadanos, eventos, etc.) ha sido tomado como objetos individuales. No se han utilizado setters y se ha mantenido la modificación de las instancias lo menos ocurrente posible; la clase listadeciudadanos, al igual que otras listas, sí son cambiadas pues en sus responsabilidadess, se actualizan según lo que sucede en el programa.

Los Archivos y clases:

**menu\_completo.py:** Contiene a la clase MainMenu la cual funciona de menú principal desde el cual se puede acceder al resto de los menús. Sus funciones son estáticas y no tiene inicializador, pues el menú es solo una interfaz para acceder a las funciones del programa y no recibe parámetros propios Entre sus funciones se encuentra:

* menu\_o(): Es el primer prompt que le surgirá a un usuario cuando quiera utilizar EventIt, permite acceder al registro y realizar el login como citizen, administrador o sensor. Además, permite salir de la interfaz (terminar el programa), esto se logra con los módulos time y os.
* menu\_login\_citizen(): un prompt dentro del menu principal, al cual se accede cuando se opta por ingresar como ciudadano. De aquí se puede acceder a las funciones register y login.
* register(): Permite a un usuario registrarse por primera vez. El register es exclusivo de ciudadanos, y pide toda la información necesaria para que el ciudadano pueda tener el perfil. Entre esta información necesaria se encuentra el CUIL, el cuál será utilizado en otras funciones, y la zona, la cuál se toma de la clase RegistrodeZonas que mantiene registro de las zonas disponibles en EventIt. Al crear el perfil, esta información es registrada en el archivo Database.csv, además de que se instancia una clase Citizen con la información del ciudadano en cuestión. Finalmente, esta nueva instancia es agregada a etlist, una instancia de la clase ListaDeCiudadanos que guarda registro de todas las instancias de ciudadanos. Cuando el registro es terminado, se vuelve al menú de login (es decir, se ejecuta la función menu\_login\_citizen() )
* login(): Otra responsabilidad del menu, los ciudadanos deben poder ingresar a su usuario mediante el menú. Si el login es realizado con éxito, el ciudadano entonces es “redireccionado” a su menú de ciudadano, menu\_citizen. Este es diferente del menu\_login\_citizen, pues de aquí el ciudadano puede acceder a sus distintas funciones.
* menu\_citizen(citizen): es el menú de ciudadano al que se accede cuando se realiza un login() correctamente, desde aquí el ciudadano puede acceder a las distintas responsabilidades que le corresponden a la clase. La función recibe un parámetro citizen, pues la información a la que se puede acceder desde aquí es diferente según la instancia de citizen. Solo es accesible si el citizen no está banneado, es decir si su parámetro CitizenBan es False. Desde aquí puede asistir a un evento, dejar de asistir a un evento, entrar a su menu de amigos y cambiar su zona. Desde el punto de vista de POO, es posible escribir a esta función como una clase separada, reduciendo la cohesión. Sin embargo, por temas de funcionalidad, complejidad al momento de eliminar esta función y crear la clase, se optó por dejarla como una función del menú principal.
* log\_adm(): Un login especial para administradores, es diferente del citizen, pues requiere menos información, pero se encuentra más limitado, pues no se pueden registrar nuevos admins (si se quiere cambiar el admin, debe agregarse al ABM de usuarios administradores). Si el login es realizado con éxito, se accede a la clase menu\_admin y sus respectivas responsabilidades.

**menuadministrador.py:** Contiene a la clase menu\_administrador el cual funciona como un menú exclusivo de administradores desde donde ellos como usuarios pueden acceder a sus responsabilidades. Sus funciones son estáticas y no tienen inicializador, al igual que MainMenu, pues es solo una interfaz y no recibe parámetros propios; es igual para cualquier admin. Entre sus funciones se encuentra:

* Bienvenido(): Funciona como un pseudo menú principal desde el cuál se accede a las distintas funciones de la clase, o volver al menú principal.
* BanCitizen(): Permite acceder a la responsabilidad exclusiva del admin de realizar bans sobre citizens (es decir, un estado donde no pueden realizar las funciones típicas de citizen) se introduce el CUIL del usuario al que se desea bannear. Luego de verificar que este usuario exista y esté registrado, es baneado y se agrega a la lista de baneados, elemento que se encuentra en la clase CitizenList. Aquí se rompe un poco el encapsulamiento, pues tanto la instancia de CitizenList (etlist) y el estado de ban (un parámetreo de citizen) son modificados por una función externa. Esto, sin embargo, es necesario para alterar el estado de ban, y para llevar registro de qué usuarios se encuentran en este estado.
* UnBanCitizen(): Permite acceder a la responsabilidad exclusiva del admin de remover el ban de un citizen. Luego de verificar que el usuario exista y esté banneado, se saca de la lista correspondiente y el estado de ban es removido. Aquí se rompe un poco el encapsulamiento, pero es necesario saber quién ya no se encuentra banneado.
* check\_revision\_list(): Permite al admin ver, acceder, a la lista de revisión y decidir qué va a hacer con los ciudadanos que se encuentran en ella. Cuando se accede a la misma, los usuarios presentes en la lista pasan a ser banneados. Consideramos el “bloqueo” y “ban” descriptos en el enunciado como el mismo estado para un citizen (donde no puede acceder a sus funciones normales). El admin, luego, puede decidir entre remover el ban o puede simplemente dejar el ban. En ambos casos el ciudadano es removido de la lista y se limpian las condiciones que lo colocaron en esa lista, de modo que no vuelva a aparecer cuando se ejecute un updaterevisionlist() (función propia de la clase RevisionList). El encapusulamiento es afectado en esta parte, pues se modifican tanto la instancia de RevisionList, defualt\_revision\_list, al igual que el citizen, pero esto es necesario para que el mismo no aparezca en la lista una vez que la revisión por parte del admin fue completada.
* addmin\_add\_event():Permite acceder a una de las responsabilidades del admin: crear eventos. Mediante inputs se introduce la información del evento y luego se accede a la función propia del admin.

Debido a un problema de importación circular, la clase descripta se encuentra dentro del archivo menu\_completo y no en su archivo separado como se planteó en el diseño

**menuamigos.py:** Contiene a la clase menu\_amigos el cual funciona como un menu para cada citizen y permite realizar acciones relacionadas con su lista de amigos. Sus funciones son estáticas y no tiene inicializador. Entre sus funciones están:

* friends\_menu(citizen): Funciona como un pseudo menú principal desde el cuál se accede a las distintas funciones de la clase, o se puede volver al menú principal. Permite acceder a las responsabilidades correspondientes al citizen con relación a sus amistades dentro de EventIT.
* Se pueden ver las solicitudes pendientes, para luego decidir si rechazarlas o aceptarlas
* Se puede enviar solicitud a un usuario deseado
* Se puede observar los eventos “compartidos” por contactos. Consideramos la asistencia a un evento como lo mismo que “reportarlo” o asistir. Solo los admins crean eventos, los ciudadanos pueden asistir a los mismos, lo cual consideramos lo mismo que “reportarlo” pues se vuelve visible a sus contactos mediante esta función.

Debido a un problema de importación circular, la clase descripta se encuentra dentro del archivo menu\_completo y no en su archivo separado como se planteó en el diseño

**usuarios.py:** Contiene la clase abstracta Usuario. Se planteó esta clase inicialmente para que haya polimorfismo entre el ciudadano y el admin, pero durante el desarrollo del programa, como se vio que sus responsabilidades y comportamientos eran tan diferentes, no se implementó, salvo para el inicializador de citizen.

**citizen.py:** Contiene a la clase Ciudadano, uno de los dos users. Presenta un inicializador con nombre, apellido, CUIL, número de teléfono (heredados de user) y zona correspondiente, una lista de eventos en los que está anotado, su estado de ban (la cual es falsa por defualt), su lista de amigos, sus solicitudes, y por quién fue rechazado.

* create\_citizen(): Función de clase que crea un citizen además de añadirlo al registro de ciudadanos.
* Init\_citizen\_creation(): Función de clase parecida a create\_citizen, pero en lugar de añadirse al registro, devuelve la clase creada.
* asistevent(): permite al ciudadano anotarse a un evento disponible
* unasistevent(): permite al ciudadano bajarse de un evento en el que ya se anotó
* enviar\_solicitud(friend\_cuil): permite enviar una solicitud de amistad al usuario que le corresponda el cuil pasado como parámetro. No respeta completamente el encapsulamiento pues la solicitud cambiará el parámetro de solicitudes del usuario receptor.
* ver\_solicitudes(): permite ver las solicitudes pendientes
* aceptar\_solicitudes(idem): permite aceptar una solicitud según el idem pasado como parámetro, el cuál corresponde a la posición de la solicitud en la lista de solicitudes
* rechazar\_solicitudes(idem): permite rechazar una solicitud según el idem pasado como parámetro, el cuál corresponde a la posición de la solicitud en la lista de solicitudes. Además, el ciudadano que realiza el rechazo es añadido a la lista de quién me rechazó del ciudadano rechazado
* change\_zone(): permite cambiar la zona en la que se encuentra el ciudadano. Permite moverse, y asistir eventos de otra zona, pero no cambia la zona con la que está registrado en el database.

**administrador.py:** Contiene a la clase administrador. No recibe parámetros, pues la información del admin ya se encuentra en el ABM de admins

* addevent(): accede a la clase EventCreator para crear un evento nuevo. Es responsabilidad del admin crear eventos
* bancitizen(citizen): Permite cambiar el estado de Ban del citizen pasado como parámetro a true, y lo añade al registro de citizens banneados
* unbancitizen(citizen): Permite cambiar el estado de Ban del citizen pasado como parámetro a false, y lo saca del registro de citizens banneados

**revisionlist.py:** Contiene a la clase RevisionList. Funciona como un listado de gente “en espera” para ser auditados por los administradores. Tiene un inicializador que crea una lista vacía para cada instancia. Además, se encuentra la instancia de esta clase default\_revision\_list, esta es utilizada como la lista de revisión principal del programa con la que interactúan otros objetos.

* update\_revision\_list(): accede a la lista de ciudadanos etlist, y revisa qué ciudadanos cumplen con los requisitos para estár en esta lista. Cuando el ciudano cumple los requisitos (en este caso haber sido rechazado 5 veces), se agrega a la lista de revisión.
* removecitizen(citizen): accede a la lista de revisión y remueve al citizen pasado como parámetro. No se implementó exception handling, pues esta función es accedida desde el menu\_de\_admin, luego de presentarle al admin los ciudadanos presentes en esta lista.
* getlist(): Devuelve la lista de revisión de la instancia.

**listadeCuidadanos.py:** Contiene la clase CitizenList que se inicializa con dos listas vacías, una de ciudadanos y otra de ciudadanos banneados (el ciudadano que está banneado no es removido de la lista de ciudadanos); además, al inicializarse, compara la información que tiene con el Database, y si encuentra un ciudadano que figura como registrado, pero no se encuentra en la lista, lo añade, así nos aseguramos que todos los ciudadanos registrados se encuentren en esta lista. Al interactuar únicamente con csv, no se viola el encapsulamiento, aunque también causa que todas las instancias de CitizenList contengan la misma información inicialmente, lo cuál puede considerarse una desventaja de este diseño. También se encuentra la instancia etlist, la cual es usada como un registro default de ciudadanos, el cuál es modificado por las funciones de otras clases en el código.

* addCitizen(citizen): añade al citizen pasado como parámetro a la lista de ciudadanos. Es responsabilidad de este tipo de registro de ciudadanos, añadir ciudadanos una vez que hayan creado un usuario en EventIt.
* removeCitizen(citizen): remueve al citizen pasado como parámetro de la lista de ciudadanos. Es una responsabilidad de este tipo de registro de ciudadanos, poder remover ciudadanos si se necesita. No ha sido implementada en el código, pues en el enunciado nunca es descripta una situación donde se deba remover ciudadanos. Aún así, consideramos que esta es una responsabilidad fundamental para un registro.
* addBannedCitizen(citizen): simplemente añade al citizen pasado como parámetro a la lista de citizens baneados. No cambia nada más, por lo que conserva perfectamente el encapsulamiento
* removeBannedCitizen(citizen): remueve al citizen pasado como parámetro de la lista de citizens baneados
* getcl(): devuelve la lista de ciudadanos.

**listadezonas.py**: Contiene a la clase registroDeZonas. No tiene inicializador y sus funciones son estáticas pues solo tiene una responsabilidad:

* listadodezonas(eventos): Toma la lista eventos, y devuelve las zonas registradas en una manera fácil de ver y trabajar

**Evento.py:** Contiene la clase EventType. Este documento recibió una serie de cambios a medida que se programó y se modificó el diseño. Originalmente se planteaba tanto una clase evento que contenía zona y número de asistentes; y una clase EventType que contiene nombre y descripción para distintos tipos de eventos. Finalmente, decidimos que el mejor diseño sería tenerlo en la misma, considerando un tipo de evento como un objeto con nombre, descripción, cantidad de personas y una localización. Bajo esta definición, el concepto evento “completo” sería una instancia de EvenType al que le corresponde un marcador en el mapa. El objeto es el EventType, y en su implementación recibe un marcador de zona.

* getDescription(): Devuelve un string con información básica sobre el evento
* getLocation(): Devuelve un string con las coordenadas del evento

**listaDeEventos.py:** Contiene a la clase eventList, cuyas responsabilidades son registrar EventTypes cuando el admin cree uno y asignarle un marcador en el mapa. Su inicializador se encarga de acceder a los csv de zonas y marcadores para añadir marcadores a las zonas según los eventos que se encuentran registrados. También contiene la instancia eventos, la cuál funciona como una lista de eventos default a la cual acceden otras clases en sus funciones. Sus responsabilidades no se acoplan, sino que en el proceso de creación, el admin manda a crear un evento dando la información del mismo como parámetros; EventList accede con esta información a EvenType, crea una instancia mediante la función eventCreator, la guarda y revisa que los datos registrados sean consistentes con las zonas presentadas (es decir, no se puede registrar un evento que este fuera de las zonas que EventList conoce mediante zonas.csv). Cabe comentar, que eventList tiene responsabilidades adicionales cuya correspondencia es discutible; la responsabilidad de registrar instancias de eventos y fijarse si están correctamente localizados es esencial, pero el proceso de creación y el añadido de marcadores podría atribuirse a una segunda clase con un nombre como “EventCreator”. Por razones de tiempo y porque se complejizarían los imports, se ha decidido dejarla como parte de la lista, pero aclaramos que se ha considerado un diseño más acorde al POO que no hemos llegado a implementar. Este cambio podría realizarse en una futura revisión.

**new\_world.py:** Contiene a la clase Mapa, cuya única responsabilidad es mostrar el mapa creado mediante el módulo folium. Por estas razones, no tiene inicializador y su única función es estática.

* show\_map(): Permite visualizar el mapa según las coordenadas establecidas previamente. Se plantea que el mapa se centre en las zonas registradas. Se crea y ejecuta un archivo en html llamado index.html mediante el módulo folium, el cuál presenta la información y subprogramas necesarios para el correcto funcionamiento del mapa.

**Estadísticas.py:** Contiene a la clase Stats\_Board, la cual tiene la responsabilidad de realizar cálculos relacionados a las estadísticas del programa en cuestión. Sus funciones son:

* show\_p\_per\_zone: utilizando el modulo de Python matplotlib, genera un grafico que muestra a las personas por zona.
* showMaxZone(num): Devuelve los tres eventos mas concurridos de la zona a la que le corresponde el índice pasado como parámetros

Otros archivos:

**Database.csv:** Archivo separado por comas, el cuál contiene el CUIL, contraseña, PhoneNumber, nombre, apellido, edad y zona de un usuario registrado. Es actualizado cada vez que se registra un nuevo ciudadano.

**Base\_Adm.csv:** Base de datos que funciona como un ABM de usuarios administradores, contiene sus nombres de usuario y contraseñas. No puede ser modificada mediante el código, si se quiere añadir un nuevo administrador se debe reescribir manualmente este archivo

**marcadores.csv:** Base de datos que guarda la zona, nombre, descripción y coordenadas de un evento, al abrir el mapa, mediante la clase Mapa, se accede a esta database y se colocan los marcadores con esta información en las coordenadas dadas.

**zonas.csv:** Base de datos que contiene las zonas registradas, con su nombre, además de latitudes y longitudes extremas para marcar el área donde comienza y termina la zona. Utilizado para todas las funciones que requieran conocer las zonas disponibles, como cuando se quiere crear un evento, o cuando se registra un citizen.

**databaseANSES.csv** (pendiente implementación): Base de datos de la ANSES (ficticia) la cual contiene ciudadanos registrados por CUIL y número de teléfono. Se utiliza únicamente en el registro de un usuario para confirmar que la información introducida coincide con la persona conocida por la ANSES. En nuestro diseño, como consideramos que el usuario puede elegir su zona y moverse, la tabla de estadística no accede a esta database para obtener información, sino que trabaja con Database.csv. Esto puede ser considerado un error en el diseño.

**Index.html:** Archivo tipo HTML creado por el módulo folium de Python para poder crear un mapa, contiene información como el tipo de mapa a visualizar, colores, diseño, tamaño, etc.

Sobre el unitesting:

Intentamos implementar el unitesting a medida que realizamos el código; sin embargo, debido a distintos cambios de diseño realizados a lo largo del proyecto, se ha tenido que reescribir múltiples veces el testeo e incluso se han cambiado lo testeado. En la primer entrega puede observarse un unitest de funciones básicas de citizen y admin. En la entrega final, tenemos una clase nueva que es una copia de MainMenu, una de las partes más importantes de nuestro código, denominada FakeMainMenu. Se han hecho otros Fakes, al igual que FakeConsole, para poder simular inputs en la consola para realizar el testing. Entendemos que esta metodología no es la óptima, debido a que hay una cantidad de líneas de código excsevivas, pero pudimos visualizar efectivamente el comportamiento de nuestros ménus, parte esencial del diseño presentado.

Finalmente:

Hemos optado por un diseño óptimo, pensando en una interfaz de una aplicación para los usuarios. La parte esencial del código son los menus, que conectan con el resto de las clases y sus funciones respectivas. Se ha intentado mantener las responsabilidades de cada clase, lo más encapsuladas posible, para así poder acceder a las mismas desde los menús sin mayor cambio. El paradigma orientado a objetos fue efectivo en este objetivo.