Departamento de TIC
Algoritmos y Programación II
Unidad 2 - Laboratorio 2
Sebastián Jaramillo Torres

Especificación de Requerimientos Funcionales

El programa debe estar en la capacidad de:

R1- Agregar un nuevo usuario con tipo de documento, número de documento, nombres, apellidos, teléfono y dirección. No se permitirá agregar usuarios con un número de documento repetido, tampoco se permitirá agregar un usuario que no tenga tipo de documento, numero de documento, nombre o apellido, los parámetros de teléfono y dirección si pueden ir vacíos.

R2- Asignar un turno ingresando el número de identificación de un usuario creado previamente. Este turno está conformado por una letra un número entre 00 y 99. El primer turno es el A00, el siguiente será A01 y así sucesivamente. Si el turno llega al último número de una letra (99), se cambiará al primer número de la siguiente letra y así seguirá (ejemplo. De B99 pasa a C00). Si el usuario tiene un turno pendiente entonces no podrá asignársele un nuevo turno hasta que su turno pendiente sea atendido. El turno a su vez tiene un tipo de turno y su duración será de acuerdo con la duración de su tipo, y se atenderán de acuerdo con su hora de inicio, el tiempo estándar de cambio de turno es de 15 segundos (0.25 minutos).

R3- Agregar nuevos tipos de turno, todos los tipos de turno tienen un nombre y una duración en minutos, por ejemplo, la duración puede ser de 2 minutos y medio (2.5), 5 minutos y cuarto (5.25), etc.

R4- Actualizar la hora del sistema de control de turnos de acuerdo con la hora del sistema de computo o ingresando nueva fecha y hora manualmente, esta hora solo se puede actualizar a una fecha u hora superior a la actual del sistema de control de turnos.

R5- Atender turnos hasta la fecha y hora actual del sistema de control de turnos. Por ejemplo, El turno A00 inicia a las 13:05, la hora actual del sistema es las 13:10, si se elige la opción de atender turnos este turno quedará atendido.

R6- Guardar el estado del programa tal cual está, se guardan sus clientes registrados, turnos registrados, tipos de turnos, hora del sistema, etc.

R7- Cargar un estado del programa desde el archivo en el cual se guardó en el anterior requerimiento.

R8- Generar un reporte con todos los turnos que ha tenido un cliente.

R9- Generar un reporte con todas las personas que han llegado a tener un turno dado por el usuario.

R10- Generar aleatoriamente un número de clientes indicados por el usuario.

R11- Generar de turnos de forma aleatoria y asignarlos a clientes registrados previamente. El usuario debe indicar cuantos días de turnos se generarán, la generación siempre empieza desde el día actual. El usuario también debe indicar el número de turnos por día que quiere generar.

R12- Mostrar una lista ordenada de los clientes. El usuario elige si la quiere ordenar por nombres y apellidos o por ID.

R13- Mostrar la fecha y hora del sistema de control de turnos en cada despliegue del menú.

R14- Buscar un turno por su nombre para verificar que existe.

REQUERIMIENTOS NO FUNCIONALES

RNF1- Para añadir un cliente se utiliza una búsqueda secuencial y excepciones propias.

RNF2- Para añadir un turno se utiliza búsqueda secuencial y excepciones propias, a su vez se utiliza el método de ordenamiento insertion para imprimir la lista de tipos de turnos en forma ordenada por nombres.

RNF3- Para añadir un tipo de turno se utiliza búsqueda secuencial y excepciones propias.

RNF4- Para actualizar las fechas me apoyo en la clase del API LocalDateTime.

RNF5- Para atender turnos comparo los tiempos de inicio de cada turno con el tiempo actual del sistema de control de turnos.

RNF6- Para aplicar persistencia uso la interfaz Serializable, ObjectOutputStream y guardo el modelo en un archivo en una ruta especifica.

RNF7- Para cargar un modelo guardado uso ObjectInputStream.

RNF8- Para generar un reporte de un cliente y escribirlo en un .txt uso BufferedWriter con FileWriter.

RNF9- Para generar un reporte de un turno y escribirlo en un .txt uso BufferedWriter y FileWriter.

RNF10- Para generar clientes de forma aleatoria el programa lee dos archivos de texto, uno con 100 nombres y otro con 100 apellidos, los agrega a un ArrayList y con Math.random se escoge el index de cada ArrayList.

RNF11- Para generar turnos aleatorios se crea una lista de clientes que no tienen turnos y a estas personas se les asignan los turnos indicados por el usuario en sus respectivos dias.

RNF12- Para imprimir la lista ordenada de clientes se usan dos métodos de ordenamiento, bubble para ordenarlos por nombre y apellido, y selection para ordenarlos por ID.

RNF13- Para buscar si el turno existe se usa búsqueda binaria, retorna null si no existe.

RNF14- Para medir cuantos milisegundos se demora en ejecución cada opción se utiliza el System.currentTimeMillis.

Escenarios

Nombre	Clase	Escenario
setup1	ControlTest	Un objeto de la clase Control con todos sus array list vacios
setup2	ControlTest	Un objeto de la clase Control añadiéndole tres objetos de tipo Client con parámetros: (0)= typeId:"CC", id:"1068895", name:"Pedro", lastName:"Hernandez", phone:"3165577", address:"cra 23 #20-36"; (1)= typeId:"CC", id:"63883786" , name:"Sandra", lastName:"Gutierrez", phone:"", address:""; (2)= typeId:"CC", id:"10097733", name:"Alfonso", lastName:"Bustamante", phone:"", address:"". También se le añaden tres objetos tipo TurnType con parámetros: (0) = name:"SERVIR DESAYUNO",duration:20; (1) = name:"SERVIR ALMUERZO",duration:26; (2) = name:"SERVIR CENA",duration:30;
setup3	ControlTest	Un objeto de la clase Control añadiéndole cuatro objetos de tipo Client con parámetros: (0)= typeId:"CC", id:"1068895", name:"Pedro", lastName:"Hernandez", phone:"3165577", address:"cra 23 #20-36"; (1)= typeId:"CC", id:"63883786" , name:"Sandra", lastName:"Gutierrez", phone:"", address:""; (2)= typeId:"CC", id:"10097733", name:"Alfonso", lastName:"Bustamante", phone:"", address:"" (3)= typeId:"CC", id:"44695237", name:"Andres", lastName:"Lopez", phone:"", address:"" También se le añaden tres objetos tipo TurnType con parámetros: (0) = name:"SERVIR DESAYUNO",duration:20; (1) = name:"SERVIR ALMUERZO",duration:26; (2) = name:"SERVIR CENA",duration:30;
setup4	ControlTest	Un objeto de la clase Control añadiéndole un objeto de tipo Client con parámetros: (0)= typeId:"CC", id:"1068895", name:"Pedro", lastName:"Hernandez", phone:"3165577", address:"cra 23 #20-36". También se le añade un objeto tipo TurnType con parámetros: (0) = name:"SERVIR DESAYUNO",duration:20

Diseño de Casos de Prueba

Objetivo de la prueba: Verificar que el programa agrega un usuario correctamente cuando no hay usuarios registrados				
Clase	**Método**	**Escenario**	**Valores de entrada**	**Resultado**
Control	addClient	setup1	typeId:"CC", id:"1068895", name:"Pedro", lastName:"Hernandez", phone:"3165577", address:"cra 23 #20-36"	Se crea correctamente el usuario con los valores pasados por parámetro al constructor

Objetivo de la prueba: Verificar que el programa agrega un usuario correctamente cuando ya hay usuarios registrados				
Clase	**Método**	**Escenario**	**Valores de entrada**	**Resultado**
Control	addClient	setup2	typeId:"CC", id:"44695237", name:"Andres", lastName:"Lopez", phone:"", address:""	Se crea correctamente el usuario con los valores pasados por parámetro al constructor

Objetivo de la prueba: Verificar que si se intenta añadir un usuario que ya existe nos arroja una excepción				
Clase	**Método**	**Escenario**	**Valores de entrada**	**Resultado**
Control	addClient	setup3	typeId:"CC", id:"44695237", name:"Andres", lastName:"Lopez", phone:"", address:""	El programa debe arrojar la excepción **ExistingClientException**

Objetivo de la prueba: Verificar que el método buscar devuelve el objeto que estamos buscando de acuerdo con el id si el usuario ya existe				
Clase	**Método**	**Escenario**	**Valores de entrada**	**Resultado**
Control	addClient search	setup1	typeId:"CC", id:"6455443", name:"Gabriel", lastName:"Martinez", phone:"", address:"" id:"6455443"	El método buscar nos devuelve el objeto que buscamos de acuerdo con el id “6455443"

Objetivo de la prueba: Verificar que el método buscar nos devuelve null al buscar un usuario cuando no existe				
Clase	**Método**	**Escenario**	**Valores de entrada**	**Resultado**
Control	search	setup2	id:"44695237"	**null**

Objetivo de la prueba: Verificar que el método buscar nos devuelve null al buscar un usuario cuando no existe usuarios en el programa				
Clase	**Método**	**Escenario**	**Valores de entrada**	**Resultado**
Control	search	setup1	id:"1068895"	**null**

Objetivo de la prueba: Verificar que el método buscar nos devuelve el objeto que estamos buscando de acuerdo con el id del usuario cuando el programa tiene varios usuarios existentes				
Clase	**Método**	**Escenario**	**Valores de entrada**	**Resultado**
Control	search	setup3	id:"1068895"	Objeto con el id "1068895"

Objetivo de la prueba: Verificar que el método registerTurn nos lanza una excepción cuando intentamos asignarle un turno a un usuario que ya tiene un turno pendiente				
Clase	**Método**	**Escenario**	**Valores de entrada**	**Resultado**
Control	registerTurn	setup4	id:"1068895" type: 0	El programa debe lanzar la excepción **ClientHasTurnException**

Objetivo de la prueba: Verificar que el método registerTurn le genera un turno nuevo a un usuario que no tiene ningún turno asignado				
Clase	**Método**	**Escenario**	**Valores de entrada**	**Resultado**
Control	registerTurn	setup3	id: "44695237" type: 1	El programa debe generarle un nuevo turno al usuario

Objetivo de la prueba: Verificar que el método registerTurn genere el turno A00 si es el primer turno que se genera y que el siguiente, al ser consecutivo sea A01				
Clase	**Método**	**Escenario**	**Valores de entrada**	**Resultado**
Control	registerTurn	setup3	id: "1068895" type: 1	Se le genera el turno A00 al usuario
Control	registerTurn	setup3	id: "63883786" type: 2	El turno generado para este usuario debe ser el A01

Objetivo de la prueba: Verificar que si el último turno generado es el D99 el siguiente debe ser E00				
Clase	**Método**	**Escenario**	**Valores de entrada**	**Resultado**
Control	setTurnToAssign registerTurn	setup3	turn: “D99” id: "1068895" type: 1	Se le genera el turno D99 al usuario
Control	registerTurn	setup3	id: "63883786" type: 0	Se le debe generar el turno E00 al usuario

Objetivo de la prueba: Verificar que si el último turno asignado es el Z99 el siguiente debe ser el A00				
Clase	**Método**	**Escenario**	**Valores de entrada**	**Resultado**
Control	setTurnToAssign registerTurn	setup3	turn: “Z99” id: "1068895" type: 0	Se le genera el turno Z99 al usuario
Control	registerTurn	setup3	id: "63883786" type: 1	Se le debe generar el turno A00 al usuario

Objetivo de la prueba: Verificar que el programa atiende correctamente un turno, también, si un turno no ha sido atendido su estado debe aparecer como pendiente y por último, consulta cuál es el siguiente turno				
Clase	**Método**	**Escenario**	**Valores de entrada**	**Resultado**
Control	registerTurn attendTurn getTurns getStatus	setup3	id: "1068895" type: 0	Devuelve el estado del turno como ***CALLED***
Control	registerTurn getTurns getStatus	setup3	id: "63883786" type: 1	Devuelve el estado del turno como ***PENDING***
Control	getTurnToAttend	setup3	*ninguno*	Devuelve el siguiente turno por atender

Objetivo de la prueba: Verificar que si se intenta registrar un turno a un usuario no existente devolverá una excepción				
Clase	**Método**	**Escenario**	**Valores de entrada**	**Resultado**
Control	registerTurn	setup1	id: "1068895" type: 0	Lanza la excepción **NoExistingClientException**

Objetivo de la prueba: Verificar que un turno que no ha sido asignado no puede ser llamado				
Clase	**Método**	**Escenario**	**Valores de entrada**	**Resultado**
Control	attendTurn	setup1	*ninguno*	Un mensaje que indica que el turno no ha sido atendido