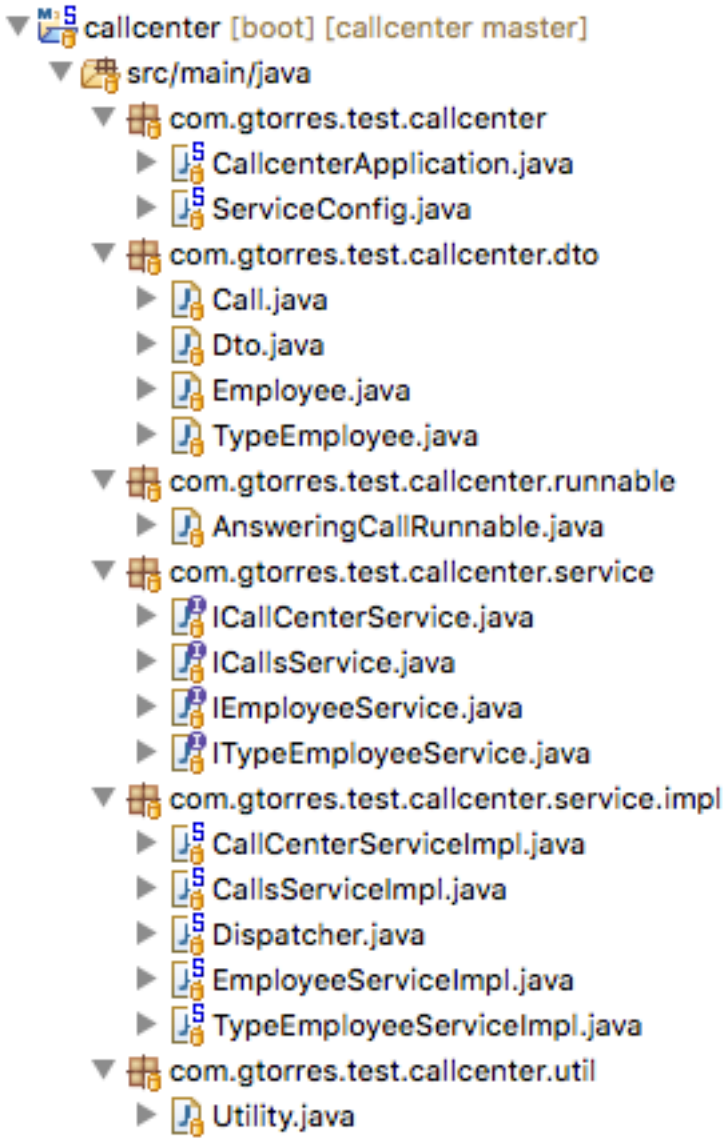
DESARROLLO EJERCICIO JAVA CALLCENTER



REQUERIMIENTO 1

Debe existir una clase Dispatcher encargada de manejar las llamadas, y debe contener el método dispatchCall para que las asigne a los empleados disponibles.

```
 4  package com.gtorres.test.callcenter.service.impl;
 5
 6  import org.springframework.beans.factory.annotation.Autowired;
10
11  /**
12   * @author gtorresoft
13   *
14   */
15  @Service
16  public class Dispatcher {
17
18      private ICallCenterService iCallCenterService;
19
20      /**
21       * @param iCallCenterService
22       */
23      @Autowired
24      public Dispatcher(ICallCenterService iCallCenterService) {
25          super();
26          this.iCallCenterService = iCallCenterService;
27      }
28
29      /**
30       * Recibe las llamadas y las asigna a un empleado
31       *
32       * @param call
33       *             llamada a ser asignada a un empleado
34       * @throws InterruptedException
35       */
36      public void dispatchCall(Call call) throws InterruptedException {
37          iCallCenterService.receivedCall(call);
38          iCallCenterService.assignCall(call);
39      }
40  }
```

REQUERIMIENTO 2 y 3

Requerimiento 2: El método dispatchCall puede invocarse por varios hilos al mismo tiempo.
Requerimiento 3: La clase Dispatcher debe tener la capacidad de poder procesar 10 llamadas al mismo tiempo (de modo concurrente).

La siguiente imagen muestra una porción de código que se repite en las pruebas unitarias en la clase **DispatcherTest.**

```
218         Dispatcher dispatcher = new Dispatcher(iCallCenterService);
219         List<UserRunnable> listUser = new ArrayList<>();
220         iCallCenterService.createEmployeesOperator("Op1", "Op2");
221         iCallCenterService.createEmployeesSupervisor("Sup1", "Sup2");
222         iCallCenterService.createEmployeesDirector("Dir1", "Dir2");
223         int numberThread = 10;
224         for (int i = 0; i < numberThread; i++) {
225             listUser.add(new UserRunnable(new Call(i + 1l), dispatcher));
226         }
227         final ExecutorService executorService = Executors.newFixedThreadPool(numberThread);
228         for (UserRunnable userRunnable : listUser) {
229             executorService.execute(userRunnable);
230         }
```

En la imagen anterior se evidencia la creación de una lista de 10 objetos **UserRunnable** que no son mas que 10 hilos para ejecutarse que simularan 10 llamadas concurrentes (Linea 224-226). Los hilos para su creación se les debe inyectar un objeto **Call** y uno **Dispatcher. Call** es la llamada que realizaría el Usuario y **Dispatcher** tiene un método **DispatcherCall** que se invocara concurremente por los hilos (Linea 225). En la línea 229 se evidencia la ejecución de los hilos concurrentemente. A continuación se muestra la clase **UserRunnable**

```
4  package com.gtorres.test.callcenter;
5
6  import com.gtorres.test.callcenter.dto.Call;
8
10 * @author gtorresoft
13 public class UserRunnable implements Runnable{
14     private Call call;
15     private Dispatcher dispatcher;
16
18      * @param call
20     public UserRunnable(Call call, Dispatcher dispatcher) {
21         super();
22         this.call = call;
23         this.dispatcher = dispatcher;
24     }
25
26     @Override
27     public void run() {
28         try {
29             this.dispatcher.dispatchCall(call);
30         } catch (InterruptedException e) {
31             // TODO Auto-generated catch block
32             e.printStackTrace();
33         }
34     }
35 }
```

REQUERIMIENTO 4

Cada llamada puede durar un tiempo aleatorio entre 5 y 10 segundos.

Se creo una clase Utility donde se crean métodos de utilidad general. En esta clase se crea el método **randomNumber** que retorna un numero aleatorio entre un rango determinado. A continuación se muestra el método.

```
39      public static int randomNumber(int desde, int hasta) {
40          Random rnd = new Random();
41          return rnd.nextInt((hasta - desde) + 1) + desde;
42      }
```

La asignación de este numero aleatorio se realiza en la ejecución de la atención de la llamada por un empleado. Para esto se creo una clase **AnsweringCallRunnable** que no es mas que un hilo que se abre para atender un llamada cuando un empleado esta disponible. A continuación la clase **AnsweringCallRunnable** en la línea 39 la asignación de este numero aleatorio**.**

```
4  package com.gtorres.test.callcenter.runnable;
5
6  import java.time.LocalTime;
12
14 * @author gtorresoft
17 public class AnsweringCallRunnable implements Runnable {
18     private Call call;
19     private Employee employee;
20     private IEmployeeService iEmployeeService;
21
23      * @param call
26     public AnsweringCallRunnable(Call call, Employee employee, IEmployeeService iEmployeeService) {
27         super();
28         this.call = call;
29         this.employee = employee;
30         this.iEmployeeService = iEmployeeService;
31     }
32
33     @Override
34     public void run() {
35         LocalTime justoAhora = LocalTime.now();
36         System.out.printf("Empleado %s (%s) Atendiendo la llamada %d a las %d horas con %d minutos y %d segundos\n",
37                 this.employee.getName(), this.employee.getTypeEmployee().getName(), this.call.getId(),
38                 justoAhora.getHour(), justoAhora.getMinute(), justoAhora.getSecond());
39         call.setCallTimeInSeconds(Utility.randomNumber(1, 10));
40         Utility.timeOutOfEmployee(call.getCallTimeInSeconds());
41         call.setTimeFinished(LocalTime.now());
42         System.out.printf("Empleado %s (%s) finalizo la llamada %d a las %d horas con %d minutos y %d segundos. Tiempo de la llamada %d\n",
43                 this.employee.getName(), this.employee.getTypeEmployee().getName(), call.getId(),
44                 call.getTimeFinished().getHour(), call.getTimeFinished().getMinute(),
45                 call.getTimeFinished().getSecond(), call.getCallTimeInSeconds());
46         this.iEmployeeService.vacateEmployee(employee);
47     }
48 }
```

REQUERIMIENTO 5

Debe tener un test unitario donde lleguen 10 llamadas.

En la clase **DispatcherTest** se realizan test unitario para 2, 4, 5, 10 y 15 llamadas concurrentes donde se acierta que la atención de todas las llamadas hechas. A continuación el test de 10 llamadas concurrentes.

```
208    /**
209     * 10 llamadas concurrentes 2 Operadores, 2 Supervisores y 2 Directores
210     */
211    @Test
212    public void dispatcherCall_10llamadasConcurrentes() {
213        ICallsService iCallsService = new CallsServiceImpl(new ArrayList<>());
214        IEmployeeService iEmployeeService = new EmployeeServiceImpl(new ArrayList<>());
215        ExecutorService executorServiceEmployees = Executors.newFixedThreadPool(10);
216        ICallCenterService iCallCenterService = new CallCenterServiceImpl(iEmployeeService, iCallsService,
217                iTypeEmployeeService, executorServiceEmployees);
218        Dispatcher dispatcher = new Dispatcher(iCallCenterService);
219        List<UserRunnable> listUser = new ArrayList<>();
220        iCallCenterService.createEmployeesOperator("Op1", "Op2");
221        iCallCenterService.createEmployeesSupervisor("Sup1", "Sup2");
222        iCallCenterService.createEmployeesDirector("Dir1", "Dir2");
223        int numberThread = 10;
224        for (int i = 0; i < numberThread; i++) {
225            listUser.add(new UserRunnable(new Call(i + 1l), dispatcher));
226        }
227        final ExecutorService executorService = Executors.newFixedThreadPool(numberThread);
228        for (UserRunnable userRunnable : listUser) {
229            executorService.execute(userRunnable);
230        }
231        executorService.shutdown();
232        while (!executorService.isTerminated())
233            ;
234        while (!executorServiceEmployees.isTerminated()) {
235            if (iCallsService.findByTimeFinished().size() == iCallsService.findAll().size()) {
236                if (!executorServiceEmployees.isShutdown()) {
237                    executorServiceEmployees.shutdown();
238                }
239            }
240        }
241
242        iCallsService.findAll().forEach(call -> {
243            Assert.assertNotNull(call.getTimeReceived());
244            Assert.assertNotNull(call.getEmployeeAsigned());
245            Assert.assertNotNull(call.getTimeAsigned());
246            Assert.assertNotNull(call.getTimeFinished());
247            Assert.assertTrue(call.getCallTimeInSeconds() > 0);
248        });
249    }
```

Extras/Plus 1

Dar alguna solución sobre qué pasa con una llamada cuando no hay ningún empleado libre

En el método ***CallCenterServiceImpl.assignCall(Call call)*** se valida que exista empleados desocupados; Si no es el caso, se hace una espera de 1 segundo preguntado si existe un empleado disponible, hasta que exista un empleado disponible no se hace la asignación de la llamada entrante.

```
137     @Override
138     synchronized public void assignCall(Call call) {
139         Employee employeeAsigned = null;
140         while (!this.existVacantEmployee()) {
141             Utility.timeOutOfEmployee(1);
142         }
```

En la línea 141 se evidencia el llamada de un método ***Utility.timeOutOfEmployee*** que por un segundo duerme el hilo en ejecución. A continuación el método ***Utility.timeOutOfEmployee***

```
16      /**
17       * Tiempo de ocupacion del empleado
18       *
19       * @param segundos
20       *            tiempo en segundos
21       */
22      public static void timeOutOfEmployee(int segundos) {
23          try {
24              Thread.sleep(segundos * 1000);
25          } catch (InterruptedException ex) {
26              Thread.currentThread().interrupt();
27          }
28      }
29
```

Extras/Plus 2

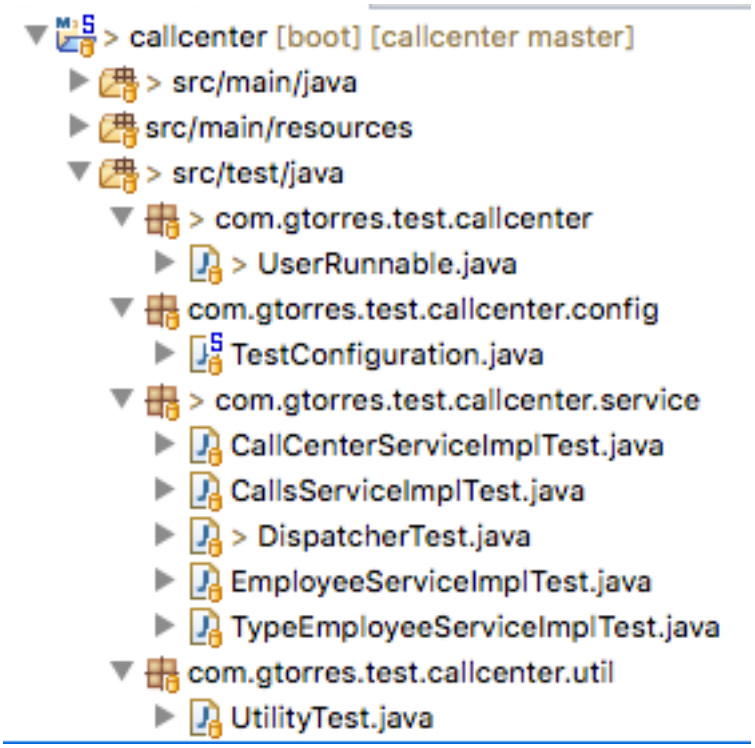
Dar alguna solución sobre qué pasa con una llamada cuando entran más de 10 llamadas concurrentes.

```
208    /**
209     * 10 llamadas concurrentes 2 Operadores, 2 Supervisores y 2 Directores
210     */
211    @Test
212    public void dispatcherCall_10llamadasConcurrentes() {
213        ICallsService iCallsService = new CallsServiceImpl(new ArrayList<>());
214        IEmployeeService iEmployeeService = new EmployeeServiceImpl(new ArrayList<>());
215        ExecutorService executorServiceEmployees = Executors.newFixedThreadPool(6);
216        ICallCenterService iCallCenterService = new CallCenterServiceImpl(iEmployeeService, iCallsService,
217                iTypeEmployeeService, executorServiceEmployees);
218        Dispatcher dispatcher = new Dispatcher(iCallCenterService);
219        List<UserRunnable> listUser = new ArrayList<>();
220        iCallCenterService.createEmployeesOperator("Op1", "Op2");
221        iCallCenterService.createEmployeesSupervisor("Sup1", "Sup2");
222        iCallCenterService.createEmployeesDirector("Dir1", "Dir2");
```

El uso la interface **ExecutorService** ayuda a crear un pool de hilos controlando la cantidad de estos a ejecutar concurrentemente. Si existen todos los hilos del pool en ejecución, este encola los hilos hasta que algún hilo del pool termine su ejecución. En los test unitarios en la línea 215 se evidencia la creación de un pool de máximo 6 hilos a ejecutar cocurrentemente. 6 hilos correspondientes a los 6 empleados que atenderán las llamadas. En las líneas 220-222 se evidencia la creación de los empleados 2 Operadores, 2 Supervisores y 2 Directores.

Extras/Plus 3

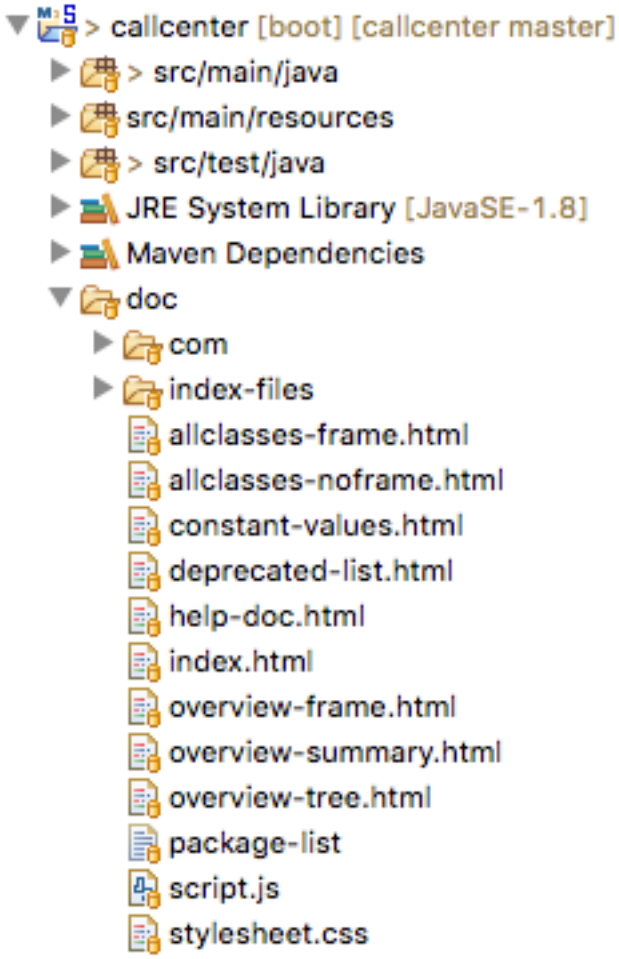
Agregar los tests unitarios que se crean convenientes.



Se hicieron las pruebas unitarios cumpliendo con un cubrimiento de mas de un 90% ya que es muy importante testear todos los métodos. Los test se ubican en la carpeta propia de Maven **src/test/java.**

Extras/Plus 4

Agregar documentación de código



La documentación del código puede evidenciarse dentro de la carpeta del proyecto en una carpeta llamada **doc.** Ver imagen anterior.