UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Manual del programador

PARA EL SOFTWARE "Desarrollo de un sistema de monitorización y control de un robot simulador de diabetes"

TRABAJO FINAL DE MÁSTER: MÁSTER EN AUTOMÁTICA E INFORMÁTICA INDUSTRIAL

Antonio Bengochea Carrasco
UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DIRECTOR: Juan Francisco Blanes Noguera

Tabla de contenido

1. Introducción

En este documento se proporcionará toda la información necesaria para seguir trabajando con el robot NAO, así como seguir ampliando la arquitectura software e implementado más funcionalidades a la misma. También se describirá el cliente realizado con todas sus clases para poder aumentar las capacidades del mismo.

1.1 Organización

En primer lugar, se comentarán las herramientas, bibliotecas y software de terceros necesario para poder desarrollar modules locales para el NAO.

Se explicará como realizar una instalación correcta del entorno de desarrollo, así como el IDE a utilizar bajo un sistema Linux. También se describirán los pasos necesarios para poder crear nuevos proyectos y los pasos para compilarlos y conseguir un .so que mover al robot.

Como se han desarrollado dos programas distintos con dos lenguajes de programación distintos, se explicará cada uno por separado indicando la estructura de ficheros, las clases que los componen indicando sus métodos y atributos y todo lo relevante para futuros desarrollos y ampliaciones.

1.2 A quien va dirigido

Este documento está dirigido a futuros programadores con conocimientos en los lenguajes C++ y Java. En particular va dirigido a todo el que busque ampliar la arquitectura software del servidor o implementar nuevas funcionalidades o escenarios, o al que tenga la intención de ampliar las capacidades del programa cliente.

2. Instalación del entorno y uso de NAOqi

Como se va a desarrollar un módulo local en el lenguaje C++, se necesita configurar un entorno de programación que disponga de todas las herramientas básicas para realizar una compilación cruzada.

El grupo Aldebaran para realizar desarrollos cruzados proporciona las herramientas para dos sistemas operativos, Mac Y Linux, en nuestro caso se usará el sistema operativo Linux, en concreto Ubuntu 14.04.5 LTS 32bits, en una máquina virtual.

Antes de realizar las instalaciones y configuraciones de las herramientas de compilación cruzada se tendrá que actualizar el sistema.

```
sudo apt-get install update
```

```
sudo apt-get install upgrade
```

2.1 Herramientas necesarias

Las siguientes herramientas serán necesarias para poder realizar un módulo local para el NOA, algunas de ellas se instalarán desde línea de ordenes otras será necesario descargarlas de la paginas oficiales.

- **GCC**
  Tiene que ser una versión superior a la 2.8.3.
- **CMAKE**
  Versión 4.4 o superior.
- **QT creator**
  IDE para programar.
- **Python 2**
  Usado por qibuild.
- **qiBuild**
  Esencial para programar el NAO, configurara el proyecto.
- **toolchain**
  Para realizar la compilación cruzada.
- **naoqi-SDK**
  Versión 2.1.4.13-linux.

2.2 Instalación

- **Instalación de CMAKE**

Antes de poder instalar CMAKE necesitaremos ejecutar el siguiente comando (si no, nos dará un error al intentar instalar CMAKE):

```
sudo apt-get install build-essential
```

Una vez se termine el comando tendremos que descargarnos CMAKE versión 3.7.2 y descomprimirlo en algún directorio auxiliar (una vez se realice la instalación de forma correcta se podrá borrar). Y navegar con la consola a la carpeta descomprimida, entonces ejecutaremos:

```
~/Desktop/cmake-3.7.2$ ./bootstrap
```

```
~/Desktop/cmake-3.7.2$ make -j4
```

```
~/Desktop/cmake-3.7.2$ sudo make install
```

Para comprobar que se instaló correctamente tenemos el comando:

```
torulo@ubuntu:~/Desktop/cmake-3.7.2$ cmake --version
cmake version 3.7.2

CMake suite maintained and supported by Kitware (kitware.com/cmake).
```

- **Instalación GCC**

Tendremos que usar el siguiente comando:

```
~/Desktop$ sudo apt-get install gcc
```

Una vez termine ejecutaremos el siguiente comando para ver si se instaló de forma correcta:

```
torulo@ubuntu:~/Desktop$ gcc --version
gcc (Ubuntu 4.8.4-2ubuntu1~14.04.3) 4.8.4
Copyright (C) 2013 Free Software Foundation, Inc.
```

- **Instalación Python**

Tendremos que usar el siguiente comando:

```
~/Desktop$ sudo apt-get install pythom
```

Para comprobar:

```
torulo@ubuntu:~/Desktop$ python --version
Python 2.7.6
```

- **Instalación QtCreator**

Tendremos que usar el siguiente comando:

```
~/Desktop$ sudo apt-get install qtcreator
```

Una vez termine podremos buscar el IDE en la barra de búsqueda del sistema.

- **Instalación qiBuild**

Tendremos que usar los siguientes comandos:

```
~/Desktop$ sudo apt-get install python-pip
```

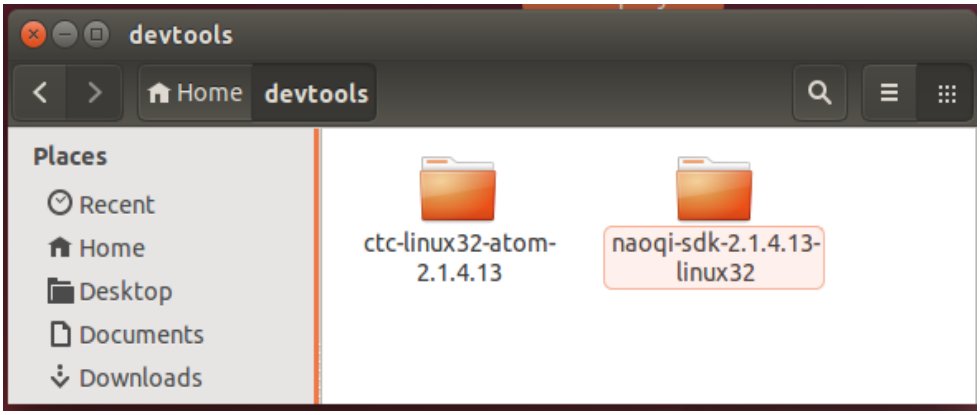
```
~/Desktop$ sudo pip install qibuild
```

```
~/Desktop$ echo export "PATH=$PATH:$HOME/.local/bin" >> ~/.profile
```

- **Instalación toolchain y naoqi-sdk**

Lo primero necesitaremos descargarnos el toolchain “ctc-linux32-atom-2.1.4.13” y sdk “naoqi-sdk-2.1.4.13-linux32.tar” de la web de Aldebaran.

Una vez los tengamos necesitaremos crear una carpeta que guardará todos los ficheros y proyectos. Crearemos una carpeta en /home/usuario/ llamada devtools y descomprimiremos en ella los ficheros descargados. Terminando con los ficheros que vemos en la ilustración dentro de la carpeta.




```
torulo@ubuntu:~$ cd devtools/
torulo@ubuntu:~/devtools$ ls
ctc-linux32-atom-2.1.4.13  naoqi-sdk-2.1.4.13-linux32
```

2.3 Configuración de qiBuild

Una vez tenemos nuestra carpeta devtools con las herramientas descomprimidas dentro ejecutaremos el siguiente comando, con el que le indicaremos a qibuild las herramientas a utilizar:

```
:~/devtools$ qibuild config --wizard
```

El cual nos dará una serie de opciones a configurar, indicaremos:

-choose generator: introduciremos 1

-IDE: introduciremos 2 y a continuación “y”.

Después tendremos que indicarle a qiBuild donde se encuentra las herramientas descargas, para ellos utilizaremos los dos siguientes comandos (donde se le indicara la ruta, tendremos que especificar nuestra ruta en la que se encuentra nuestra carpeta devtools y el fichero toolchain.xml):

```
torulo@ubuntu:~/devtools$ qitoolchain create toolchain /home/torulo/devtools/naoq
i-sdk-2.1.4.13-linux32/toolchain.xml
```

```
torulo@ubuntu:~/devtools$ qitoolchain create atom /home/torulo/devtools/ctc-linux
32-atom-2.1.4.13/toolchain.xml
```

Con las herramientas ya configuradas para ser usadas por qiBuild podemos proceder a configurar el directorio (la carpeta devtools), para ello utilizaremos los siguientes comandos:

```
:~/devtools$ qibuild init
```

```
~/devtools$ qibuild  add-config toolchain -t toolchain --default
```

```
:~/devtools$ qibuid add-config cross-atom -t atom
```

Con esto ya tendremos listo nuestro entorno.

2.4 Creación de un nuevo proyecto

Para poder crear nuestro nuevo proyecto, tendremos que navegar a nuestra carpeta devtools previamente configurada y utilizar el siguiente comando:

```
~/devtools$ qisrc create nombreProyecto
```

Este comando nos creara la estructura básica de archivos y para empezar a desarrollar nuestro modulo. Que estará constituida por los siguientes ficheros:

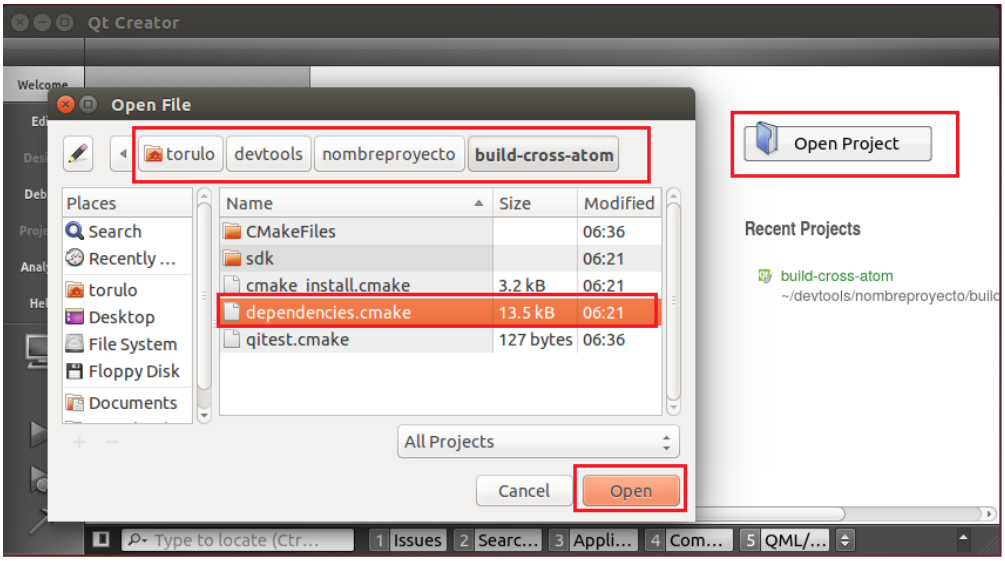
- Qiproject.xml: fichero que contendrá información del proyecto como el nombre, este fichero no se modificará.
- Main.cpp: este es el fichero principal del programa, en este fichero será donde se le indique si nuestro modulo será de ejecución local o remota.
- CmakeList.txt: contendrá toda la configuración de nuestro proyecto, se le indicaran las clases que componen el proyecto y su localización, las librearías a utilizar, el nombre para el fichero ".so", etc.
- Test.cpp: simple fichero de código de ejemplo el cual podremos borrar.

Una vez tenemos el proyecto creado navegamos dentro de él utilizando la herramienta "cd" de liana de comandos.

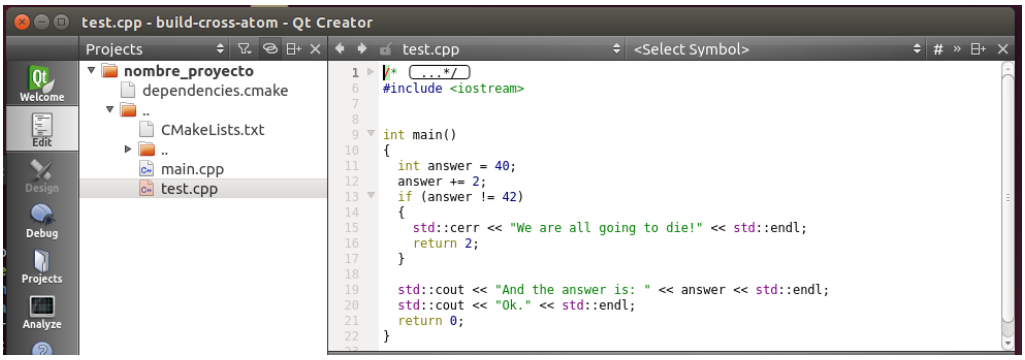
Para poder utilizar nuestro IDE para programar de una forma más cómoda, se necesitará ejecutar el siguiente comando que configurará las dependencias, en otro caso nos dará error al intentar abrir el proyecto desde QtCreator:

```
~/devtools/nombreproyecto$ qibuild configure -c cross-atom
```

Para poder abrir el proyecto con QtCreator, tendremos que abrir el programa y pulsar en el botón "Open Project", se nos abrirá una ventana en la cual tendremos que navegar a la carpeta que contiene nuestro proyecto y a la carpeta "build-cross-atomm", donde encontraremos un fichero con el nombre "dependencies-cmake" (solo encontraremos esta carpeta si hemos realizado el comando configure) al cual aremos doble clic y pulsaremos en el botón configure Project si nos lo pide.



El programa nos cargara todas las dependencia y ficheros que componen nuestro proyecto y ya podremos utilizar el IDE para programar.



Una vez queramos compilar y generar nuestro .so o nuestro binario tendremos que utilizar el siguiente comando:

```
~/devtools/nombreproyecto$ qibuild make -c cross-atom
```

Dependiendo de las opciones indicadas en el fichero CMAKE y el main.cpp nos generara unos ficheros u otros, los cuales se puede encontrar en:

- La subcarpeta *"bin":* se generará la aplicación en formato ejecutable para ejecutar de manera remota tras compilar sin emplear el compilador cruzado
- La subcarpeta *"lib":* se generará la aplicación en formato librería, lista para ser cargada en el robot y ejecutada de manera local.

Si queremos abrir un proyecto ya creado solo tendremos que moverlo a la carpeta devtools y realizar los pasos anteriores (comando configure y abrirlo en QtCreator).

3. Estructura física del proyecto

3.1 Ficheros del sistema

A la estructura de proyecto básico generado por qiBuild se le han ido añadiendo nuevas clases y ficheros, terminando con los siguientes ficheros que componen al sistema:

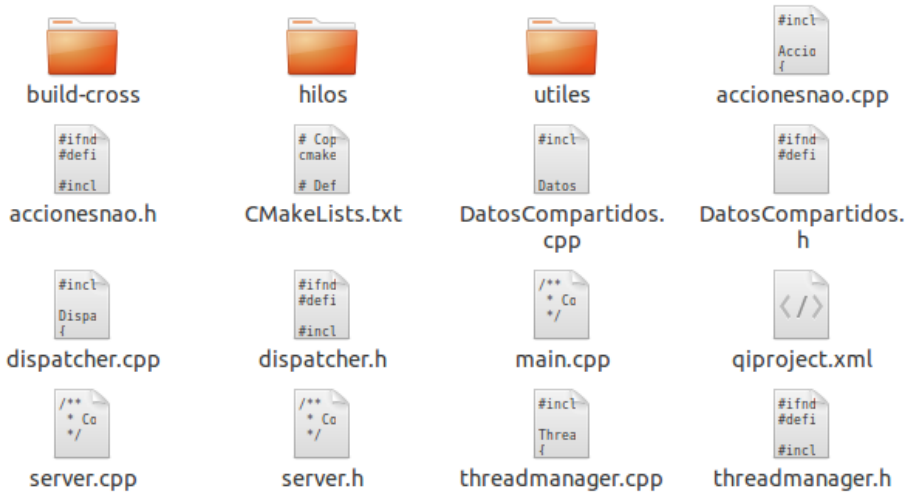
- Ficheros generados por qibuild y básicos:
  - main.cpp
  - qiproject.xml
  - CMakeList.txt
- Ficheros referentes a la librería Alglib:
  - **alglibinternal.cpp**
  - **alglibinternal.h**
  - **alglibmisc.cpp**
  - **alglibmisc.h**
  - **ap.cpp**
  - **ap.h**
  - **diffequations.h**
  - **diffequations.cpp**
  - **fasttransforms.cpp**
  - **fasttransforms.h**
  - **integration.cpp**
  - **integration.h**
  - **linalg.cpp**
  - **linalg.h**
  - **specialfunctions.cpp**
  - **specialfunctions.h**
  - **statistics.cpp**
  - **statistics.h**
  - **stdafx.h**
- **Ficheros añadidos durante el desarrollo**
  - server.cpp
  - server.h
  - threadmanager.cpp
  - threadmanager.h
  - DatosCompartidos.cpp
  - DatosCompartidos.h
  - accionesnao.cpp
  - accionesnao.h
  - dispatcher.cpp
  - dispatcher.h
  - runnable.cpp
  - runnable.h
  - interaccion.cpp
  - interaccion.h
  - escenario.cpp
  - escenario.h
  - tcpServer.cpp

- tcpServer.h
- tcpreciver.cpp
- tcpreciver.h
- tcpsender.cpp
- tcpsender.h
- simulador.cpp
- simulador.h

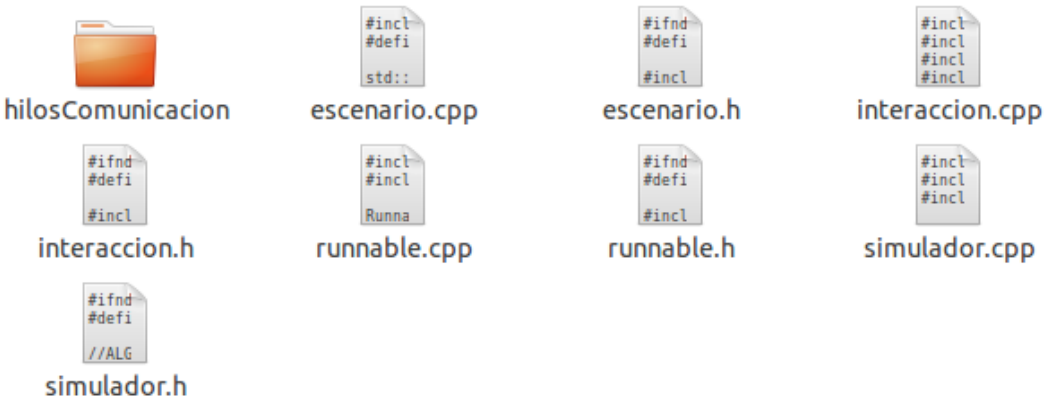
3.2 Organización en carpetas

Como podemos ver en el apartado anterior el número de ficheros que componen el proyecto es bastante elevado, por lo que se ha optado por organizarlas en subcarpetas con intención de ganar en claridad y simplicidad a la hora de modificar las clases y programar.

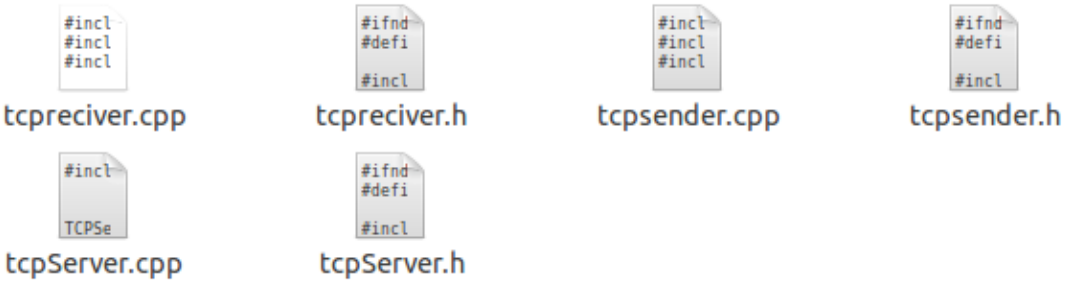
- Carpeta "Server": esta será la carpeta principal del proyecto, contendrá a las demás carpetas y algunos ficheros de configuración y algunas clases.



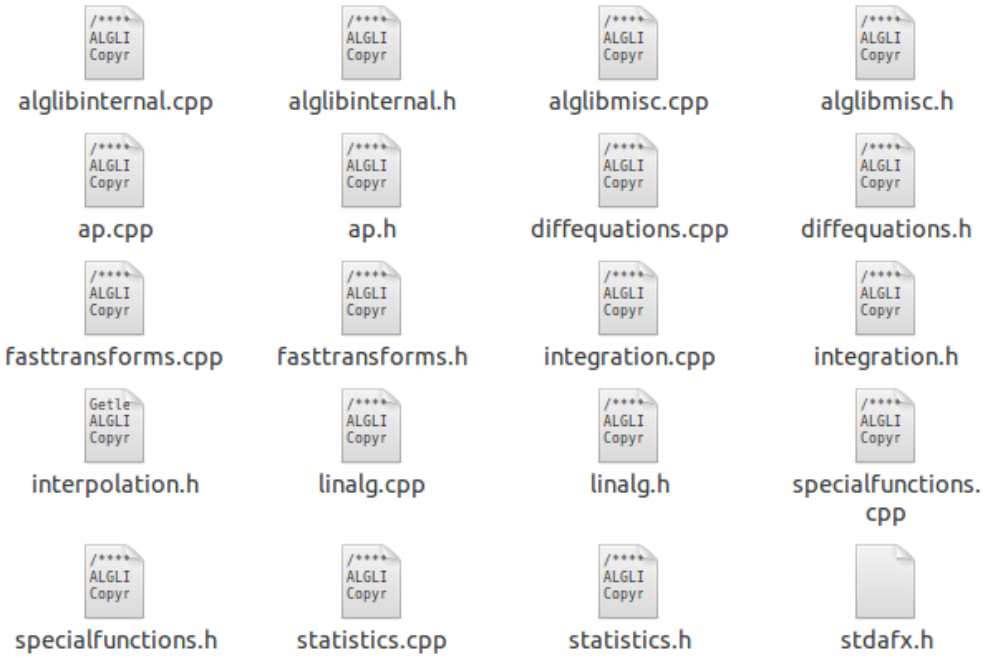
- Carpeta "build-cross": esta carpeta se genera automáticamente al realizar los comandos de "configure" y "make", puede que se genere con otro nombre.
- Carpeta "hilos": en esta carpeta se guardarán todos los ficheros relacionados con las clases encargadas de gestionar los hilos.



- Carpeta “hilosComunicacion”: se encontrará dentro de la carpeta hilos y contendrá todas las clases relacionadas con las comunicaciones vía internet.



- Carpeta “utiles”: en esta carpeta se tendrán todos los ficheros de la librearía Alglib.



4. Modulo Servidor

4.1 CMAKE

Como ya se comentó este fichero guardara información de configuración que necesitara qiBuild y NAOqi para compilar el proyecto. En él se encontrarán definidas todas las clases de las que se componen el proyecto (indicando tanto .cpp como .h), además se tendrá que indicar donde se encuentran en caso de no encontrarse en la carpetera raíz.

Sera necesario indicar que va a utilizarse qiBuild y las librerías de NAOqi a utilizar. También se le indicara con que nombre creara el módulo (.so) y en que lugar dentro de la carpeta "build-cross".

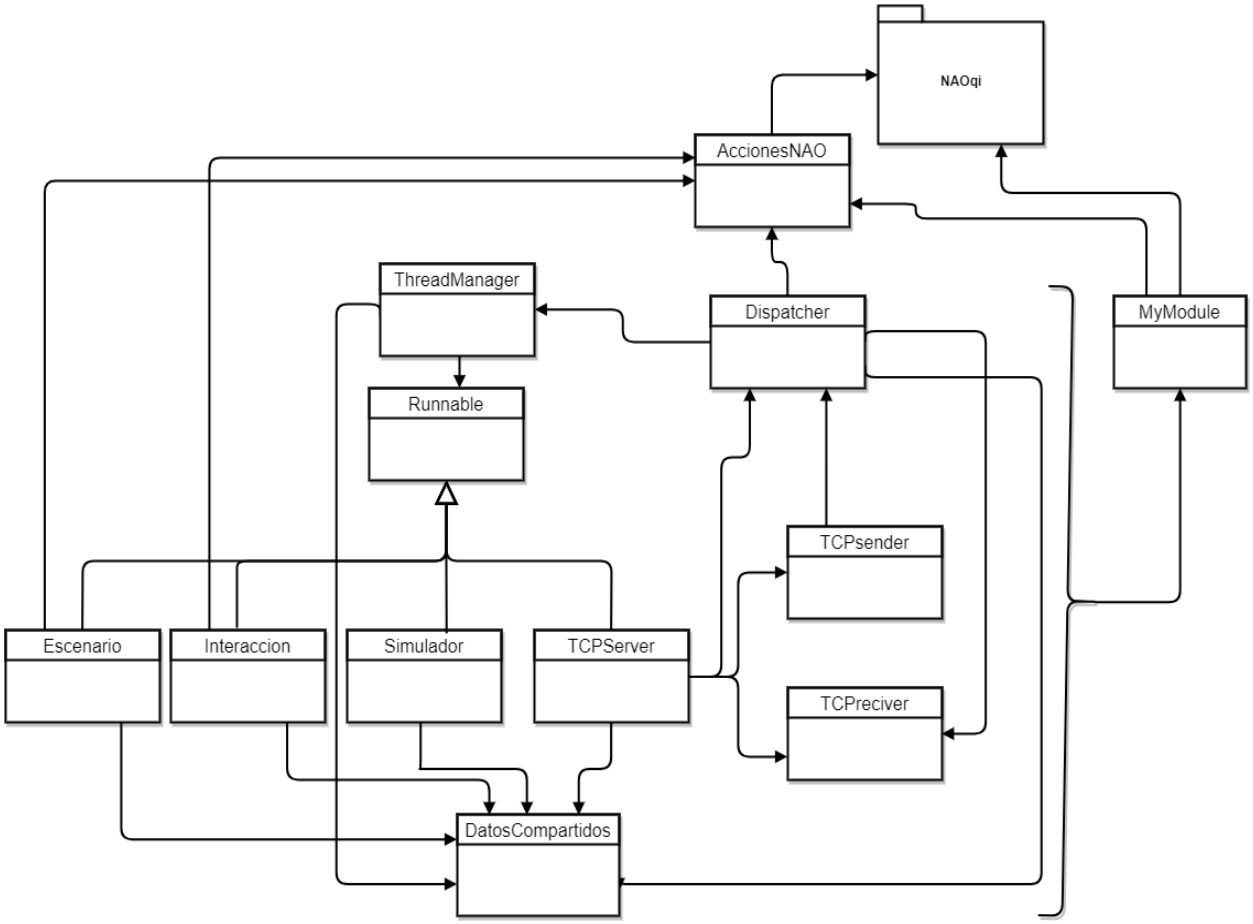
El fichero CMAKE de nuestro proyecto tendrá el siguiente aspecto:

```
# Copyright (C) 2011 Aldebaran Robotics
cmake_minimum_required(VERSION 2.6.4 FATAL_ERROR)

# Define the name of the project
project(server)

# This include enable you to use qibuild framework
find_package(qibuild)


# Create a list of source files
set(_srcs
  server.cpp
  server.h
  main.cpp

        threadmanager.cpp
        threadmanager.h
        DatosCompartidos.cpp
        DatosCompartidos.h
        accionesnao.cpp
        accionesnao.h
        dispatcher.cpp
        dispatcher.h

        hilos/runnable.cpp
        hilos/runnable.h
        hilos/interaccion.cpp
        hilos/interaccion.h
        hilos/escenario.cpp
        hilos/escenario.h

        hilos/hilosComunicacion/tcpServer.cpp
        hilos/hilosComunicacion/tcpServer.h
        hilos/hilosComunicacion/tcpreciver.cpp
        hilos/hilosComunicacion/tcpreciver.h
        hilos/hilosComunicacion/tcpsender.cpp
        hilos/hilosComunicacion/tcpsender.h

```

```
        hilos/simulador.cpp
        hilos/simulador.h
        utiles/alglibinternal.h
        utiles/alglibinternal.cpp
        utiles/alglibmisc.h
        utiles/alglibmisc.cpp
        utiles/ap.h
        utiles/ap.cpp
        utiles/diffequations.h
        utiles/diffequations.cpp
        utiles/fasttransforms.cpp
        utiles/fasttransforms.h
        utiles/integration.cpp
        utiles/integration.h
        utiles/linalg.h
        utiles/linalg.cpp
        utiles/specialfunctions.cpp
        utiles/specialfunctions.h
        utiles/statistics.h
        utiles/statistics.cpp
        utiles/stdafx.h
        utiles/interpolation.h
)


qi_create_lib(server SHARED ${_srcs} SUBFOLDER naoqi)

# Tell CMake that sayhelloworld depends on ALCOMMON and
# ALPROXIES.
# This will set the libraries to link sayhelloworld with,
# the include paths, and so on
qi_use_lib(server ALCOMMON ALPROXIES BOOST)
```

4.2 Clases

A continuación, veremos cada una de las clases del sistema centrándonos en los métodos y atributos que tienen, así como en sus particularidades. La interconexión y dependencias de las clases se puede ver en el siguiente diagrama de clases:



4.2.1 Main.cpp

Tendrá un código predefinido por NAOqi para la creación de módulos locales, en el cual se tendrá que modificar un par de líneas.

```
extern "C"
{
  ALCALL int _createModule(boost::shared_ptr<AL::ALBroker> pBroker)
  {
    // init broker with the main broker instance
    // from the parent executable
    AL::ALBrokerManager::setInstance(pBroker->fBrokerManager.lock());
    AL::ALBrokerManager::getInstance()->addBroker(pBroker);
    // create module instances
    AL::ALModule::createModule<Server>(pBroker, "Server");
    return 0;
  }

  ALCALL int _closeModule(  )
  {
    return 0;
  }
} // extern "C"
```

Tendremos que indicar el ID o nombre que tendrá nuestro modulo. Cuando se encienda el robot y se carguen todos los módulos NAOqi asociara este ID a nuestro modulo (lo necesitaremos cuando solicitemos al robot escuchar palabras ya tenemos que decir el ID del módulo al que tiene que avisar). Y tendremos que realizar el include de nuestra clase principal, en nuestro caso “server.h”.

```
#ifndef _WIN32
# include <signal.h>
#endif

#include <alcommon/albroker.h>
#include <alcommon/albrokermanager.h>
#include <alcommon/altoolsmain.h>

#include "server.h"
```

El resto del código no hay que modificarlo.

4.2.2 Server

```
class Server : public AL::ALModule
{
  public:
    Server(boost::shared_ptr<AL::ALBroker> pBroker, const std::string& pName);
    virtual ~Server();
    virtual void init();
    void onSpeechRecognized(const std::string& name,const AL::ALValue& val,const std::string& myName);

private:
    ThreadManager *tManager;
    DatosCompartidos *datosCompartidos;
    TCPServer *tcp;
    Simulador *simu;
    Interaccion *inte;
    Escenario *escenario;
    AccionesNAO *ac;
    Dispatcher *disp;
};
```

Esta clase será la que instancie el robot cuando al ponerlo en marcha se llame al módulo (será obligatorio que herede del método AL::ALModule y sobrescriba algunos métodos), será la clase de entrada de nuestro sistema. Siendo necesario realizar los includes de todas las clases que vayamos a utilizar.

Tendrá solo dos métodos principales:

- Init(): en este método estará toda la instanciación y creación de la estructura de nuestro modulo, el método es invocado automáticamente una vez se cree el módulo local, se añadirán al ThreadManager los hilos:
    - Simulador
    - Interaccion
    - Escenario
    - TCPServer

  De los cuales a TCPServer y al Simlador los pondrá en marcha.

```
27  void Server::init()
28  {
29      ac = new AccionesNAO();
30      int aux=ac->setParentBroker("Server",getParentBroker());
31      if(  aux == -2 ){
32          ac->decirFrase(std::string("broker mal"));
33          return ;
34      }
35
36      ac->decirFrase(std::string("Empezamos"));
37
38      tManager = new ThreadManager();
39      datosCompartidos = new DatosCompartidos();
40
41      //iniciamos datos que utilizaran algunos hilos y no se tiene claro donde cre
42      datosCompartidos->setData("EXACPALABRA",(double)0.4,false);
43
44      simu = new Simulador();
45      tcp = new TCPServer(6666);
46      disp = new Dispatcher();
47      inte = new Interaccion();
48      escenario = new Escenario();
49
50      //ac->decirFrase(std::string("clases bien"));
51
52      //proporcionamos a cada objetos el resto de objetos que neceista para funcio
53      if( inte->setAccionesNAO(ac) != 1 ){ ...}
57      if( inte->setDatosCompartidos(datosCompartidos) != 1 ){ ...}
61
62      if( escenario->setAccionesNAO(ac) != 1 ){ ...}
66      if( escenario->setDatosCompartidos(datosCompartidos) != 1 ){ ...}
70
71      if( disp->setAccionesNAO(ac) != 1 ){ ...}
75      if( disp->setThreadManager(tManager) != 1 ){ ...}
79      if( disp->setDatosCompartidos(datosCompartidos) != 1 ){
80          ac->decirFrase(std::string("datos dispatcher mal"));
81          return ;
82      }
```

```
83
84      if( tcp->setDatosCompartidos(datosCompartidos) != 1 ){ ...}
88      if( tcp->setDispatcher(disp) != 1 ){ ...}
92
93      if( simu->setDatosCompartidos(datosCompartidos) != 1 ){ ...}
97
98      if( tManager->setDatosCompartidos(datosCompartidos) != 1 ){ ...}
102
103     if( tcp->inicializarServidor() != 1 ){ ...}
107
108
109     if( tManager->addHilo("SIMULACION",simu) == -1)
110         ac->decirFrase(std::string("Error añadir simulacion"));
111     if( tManager->addHilo("TCP",tcp)  == -1)
112         ac->decirFrase(std::string("Error añadir tcp"));
113     if( tManager->addHiloExcluyente("INTERACCION",inte)  == -1)
114         ac->decirFrase(std::string("Error añadir interaccion"));
115     if( tManager->addHiloExcluyente("ESCENARIO",escenario)  == -1)
116         ac->decirFrase(std::string("Error añadir interaccion"));
117
118     sleep(5);
119     ac->decirFrase(std::string("Lanzamos hilos"));
120     sleep(5);
121
122     //SE ARRANCAN LOS HILOS
123     if( tManager->arrancar("SIMULACION") == -1)
124         ac->decirFrase(std::string("Error arrancar simulacion"));
125
126     sleep(5);
127     ac->decirFrase(std::string("Simulacion lanzada"));
128
129     if( tManager->arrancar("TCP") == -1)
130         ac->decirFrase(std::string("Error arrancar tcp"));
131
132     ac->setLedsOjosRed(false);ac->setLedsOjosGreen(true);ac->setLedsOjosBlue(false);
133     ac->decirFrase(std::string("hilos arrancados"));
134     printf("join tcp \n");
135     tcp->join();
136 }
```

- onSpeechRecognized(): en este método se indica que hacer cuando el robot nos avise de una palabra reconocida, en nuestro caso avisaremos a la clase AccionesNAO con la palabra y exactitud.

```
void Server::onSpeechRecognized(const std::string &name,
                                const ALValue &val, const std::string &myName)
{
    std::string palabra = (std::string)val[0];
    float exactitud = (float)val[1];

    ac->palabraReconocida(palabra,exactitud);
}
```

Los atributos que vemos en esta clase son las clases que dan funcionalidad al sistema, se instanciaran (una sola vez) y configuran (pasándole todas las clases que necesiten para funcionar) en la método init().

Importante que en el constructor de esta clase se realice el “bind” del método onSpeechRecognized(), si no esté modulo no podrá recibir la palabra reconocida.

```
Server::Server(boost::shared_ptr<ALBroker> broker, const std::string& name):
  ALModule(broker, name)
{
  /** Describe the module here. This will appear on the webpage*/
  setModuleDescription("A sever module.");

  functionName("onSpeechRecognized",getName(),"Called by ALMemory when a word is recognized.");
  BIND_METHOD(Server::onSpeechRecognized);
}
```

4.2.3 DatosCompartidos

Esta clase es la encarga da controlar el acceso seguro a datos comunes para todos los hilos del sistema. Los datos dentro de esta clase estarán asociados a una ID única.

Podrá gestionar 2 tipos de datos: los genéricos (INT, DOUBLE, STRING) y un dato tipo struct específico para el hilo simulador.

```
struct datosSimulador{
    double bolus;
    double cho;
    double ejercicioTiempo;
    double ejercicioIntesidad;
    double ejercicioDuracion;
    bool ejercicio;
};

#define INT 0
#define DOUBLE 1
#define STRING 2
```

Se realizarán unos defines que nos ayudarán a guardar el tipo de cada dato, para poder averiguarlo cuando se necesite y tratarlo de forma adecuada.

Como los datos dentro de esta clase pueden ser accedidos por varios hilos de forma simultánea necesitamos declarar mutex que controlen el acceso, crearemos dos: uno para los datos genéricos y otro para los datos específicos de la simulación.

```
private:
    //datos especificos para el hilo simulacion
    datosSimulador datosSim;
    boost::mutex *mutexDatosSim;
    boost::mutex *mutex;

    std::map<std::string,int>         datosInt;
    std::map<std::string,double>      datosDouble;
    std::map<std::string,std::string> datosString;

    std::map<std::string,int>         tipoDato;
    std::vector<std::string> datosEnviables;
```

Los datos genéricos estarán almacenos en estructuras de datos de tipo std::map, donde la clave será el ID y el valor el dato guardado. Como vemos tenemos tres atributos:

- datosInt
- datosDouble
- datosString

Necesitamos tener tres std::map distintos ya que no se pueden guardar diferentes tipos de datos en un std::map. Dependiendo del tipo del dato que se quiera guardar o leer se accederá a una u otra de estas variables.

Para los datos de simulación como son específicos solo hará falta crear un atributo (datosSim) del tipo datosSimulador (struct definido en la clase). Este dato tendrá un valor por defecto (todo a 0 y false) y cada vez que se realice una lectura se pondrá a default.

```
datosSimulador  DatosCompartidos::getDatosSimulacion()
{
    struct datosSimulador aux;
    mutexDatosSim->lock();
        aux = datosSim;
        datosSim.bolus=0;
        datosSim.cho=0;
        datosSim.ejercicio=false;
        datosSim.ejercicioTiempo=0;
        datosSim.ejercicioIntesidad=0;
        datosSim.ejercicioDuracion=0;
    mutexDatosSim->unlock();
    return aux;
}
```

También se creará un atributo tipoDato, que será otro std::map donde la clave es el ID del dato y el valor es un entero que nos indica el tipo de dato asociado el ID. Tendremos también un vector con las IDs de todos los datos que estén marcados para ser enviados al cliente.

Los métodos públicos de esta clase permitirán al resto de hilos crear, modificar, eliminar y obtener cualquier dato de forma correcto.

```
    bool isData(std::string id);
public:
    DatosCompartidos();

    std::vector<std::string> getDatosEnviables();
    int getTipoDato(std::string id);

    //funciones para dar de alta datos compartidos
     int setData(std::string id,int data,bool en);
     int setData(std::string id,double data,bool en);
     int setData(std::string id,std::string data,bool en);

     //funciones para modificar el contenido de algun dato compartido
     int modifyData(std::string id,int data);
     int modifyData(std::string id,double data);
     int modifyData(std::string id,std::string data);

     //funciones para leer el valor de un dato compartido
     int getData(std::string id, int& data);
     int getData(std::string id, double& data);
     int getData(std::string id, std::string& data);

     int deleteData(std::string id);

    //seter/geter datosSimulacion
    int setDatosSimulacion(double _bolus, double _cho,bool _ejercicio, double _ejTiempo,
                           double _ejIntesidad,double _ejDuracion);
    datosSimulador getDatosSimulacion();
```

Como vemos para las acciones de crear (setData), modificar (modifyData) y obtener (getData) tenemos tres métodos para cada uno. Nos permiten tratar de forma correcta cada dato y así almacenarlo donde toca (entre ellos se diferencian por el tipo de dato que aceptan).

El método setData será el que nos permita marcar los datos como enviables y cada vez que se añada un dato, se actualizará la variable tipoDato con el ID y tipo de dato añadido.

```
int DatosCompartidos::setData(std::string id, double data,bool env)
{
    int aux = -1;
    mutex->lock();
        if(!this->isData(id))
        {
            std::pair<std::map<std::string,double>::iterator,bool> ret;
            ret = datosDouble.insert(std::make_pair(id, data));
            if (ret.second==false) aux = -1; //el elemento ya existe
            tipoDato.insert(std::make_pair(id, DOUBLE));
            aux = 1;
            if(env == true)
                datosEnviables.push_back(id);
        }
    mutex->unlock();
    return aux;
}
```

El método getDatosEnviables(), nos dará la lista con las ID de todos los datos para enviar, pudiendo obtener el dato. Como no podemos saber de que tipo es el que este asociado a la ID que tenemos se proporciona el método getTipoDato() para poder averiguarlo.

4.2.4 AccionesNAO

Esta clase es la que se encargara de realizar todas las llamas al framework NAOqi para realizar acciones con el robot (habla, movimientos, leds, etc). Necesitaremos incluir todos los proxys que se vayan a utilizar.

```
#include <alcommon/almodule.h>
#include <alproxies/almemoryproxy.h>
#include <alproxies/alanimatedspeechproxy.h>    //proxy para hablar de manera animada
#include <alproxies/alrobotpostureproxy.h>      //proxy de posturas del robot
#include <alproxies/almotionproxy.h>            //proxy de movimiento
#include <alproxies/alautonomousmovesproxy.h>   //proxy de movimientos naturales del robot
#include <alproxies/altexttospeechproxy.h>      //proxy para hablar
#include <alproxies/alledsproxy.h>              //proxy para leds
#include <alproxies/alautonomouslifeproxy.h>            //proxy para leds
#include <alcommon/alproxy.h>
#include <alcommon/albroker.h>
#include <alproxies/albasicawarenessproxy.h>
```

Por cada proxy que necesitemos utilizar tendremos que crear un atributo para almacenarlo. También será necesario crear un atributo que almacenará el nombre del módulo padre.

```
private:
    // nombre del module al que subcribiremos para que el nao avise de palabra reconocida
    std::string nombreModuloPadre;
    boost::shared_ptr<AL::ALProxy> fAnimatedSpeech;
    boost::shared_ptr< AL::ALBroker > parentBroker;
    boost::shared_ptr<AL::ALAutonomousMovesProxy> automovesproxy;
    boost::shared_ptr<AL::ALRobotPostureProxy> postureProxy;
    boost::shared_ptr<AL::ALMotionProxy> motionProxy;
    boost::shared_ptr<AL::ALAutonomousLifeProxy> autonomousLifeProxy;
    boost::shared_ptr<AL::ALProxy> fSpeechRecognition;
    boost::shared_ptr<AL::ALMemoryProxy> fMemory;
    boost::shared_ptr<AL::ALBasicAwarenessProxy> fBasicAwareness;
    AL::ALLedsProxy ledsProxy;
    AL::ALTextToSpeechProxy fTextToSpeech;

    bool isWaitingWord;
    bool isTalking;
    bool isMoving;
    bool isThreadBlock; // variable que indica si un thread quiere uso exclusivo

    //variables que permiten a los hilos esperar por una palabra
    boost::condition_variable cond;
    boost::mutex mut;
    std::string palabra;
    float exact;

    boost::mutex mutHablar;
    boost::mutex mutexAcciones;
    boost::mutex mutBlock;
```

Además, se necesitarán crear una serie de atributos que nos ayuden con la gestión de la clase, para saber si el robot está hablando, moviéndose, escuchando o si se ha solicitado que la clase sea de uso exclusivo y solo la use un hilo (atributo que tendrá que mirar todo hilo que quiera realizar una acción puntual como TcpSender).

Como esta clase puede ser llamada por varios hilos de forma simultánea, se protegerá con varios mutex, uno para el habla, otro para las Acciones y otro para controlar la variable que indica el uso exclusivo.

En cuanto a los métodos ofrecidos:

```
    void configurarGruposLeds();
public:
    AccionesNAO();
    int setParentBroker(std::string nombreModuloNAO,
                        boost::shared_ptr< AL::ALBroker > pBroker);
    int getDCMCycleTime();
    //metodos para el reconocimiento de palabras
    int esperarPalabra(std::vector<std::string> wordlist,
                       int seconds, std::string *_palabra, float &exactitud);
    void palabraReconocida(std::string _palabra, float exactitud);
    void pararEsperaPalabra();

    int decirFrase(std::string frase);

    //leds
    void setLedsOjosRed(bool onoff);
    void setLedsOjosGreen(bool onoff);
    void setLedsOjosBlue(bool onoff);

    //acciones predefinidas
    int accionMedirGlucosa();
    int accionComer();
    int accionPinchate();
    int accionCorrer();
    int accionTumbarse();
    int accionSentarse();
    int accionLevantarse();
    int posicionParada();
    int accionChocarMano();

    void setThreadBock(bool flag);
    bool getThreadBlock();

    int accionSaludo(int num);
    int accionDespedida(int num);
    int accionQUETALESTAS(int num);
};
```

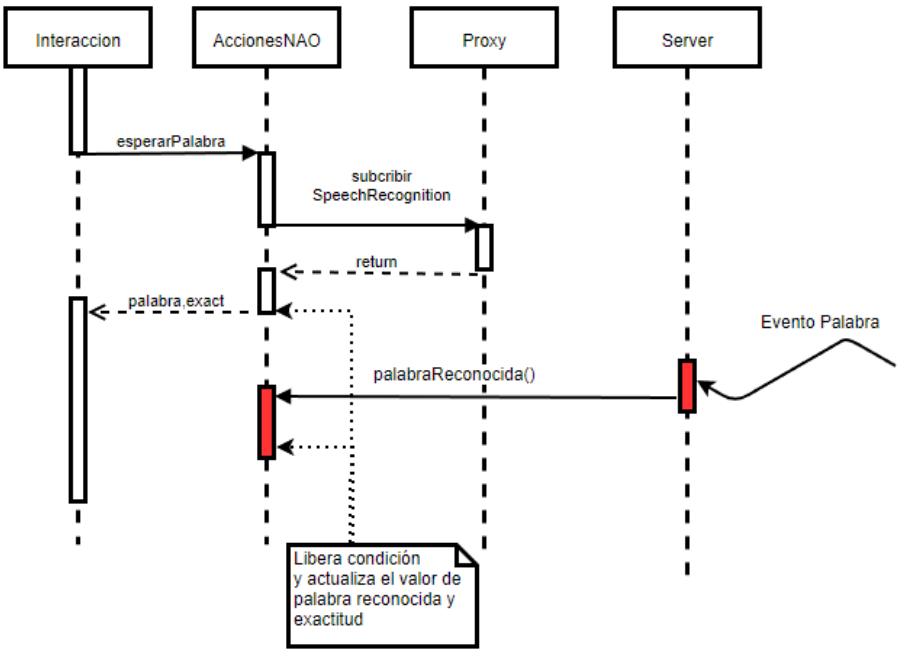
Contará con un único método privado que será configurarGruposLeds() y se encargara de realizar la configuración inicial de los leds que requiere el robot. El resto de los métodos serán públicos y accesibles por las clases.

EL método setParentBroker(), tendrá que ser llamado una vez se instancie la clase y antes de realizar cualquier otra acción. Este método se encargará de solicitar todos los proxys necesarios y hacer las configuraciones de los leds. Si esta función no es llamada el resto de funciones de acción nos darán error.

Los métodos setThreadBlock() y getThreadBlock(), serán los que nos permitan solicitar esta clase como excluyente y comprobar su estado.

La función esperarPalabra() pondrá al NAO en escucha de una serie de palabras (se le tienen que pasar como argumento). Este método será bloqueante, dejando a la espera a todo hilo que lo utilice hasta que se reconozca una palabra o pase un periodo de tiempo indicado. Para desbloquear al hilo que se encuentre esperando una palabra, la clase cuenta con dos métodos: palabraReconocida() que se le pasara la palabra y exactitud y será llamada por la clase Server

cuando el robot reconozca una palabra y pararEspera() que desbloqueara al hilo si necesitamos que se despierte antes del timeout o de reconocer palabra.



Para el manejo de los leds de los ojos la clase cuenta con tres métodos, los cuales aceptan un argumento y es el estado de los leds (apagados o encendidos). Los métodos son setLedsOjosRed(), setLedsOjosGreen(), setLedsOjosblue(). Los leds se pueden combinar para hacer colores distintos dependiendo los colores encendidos.

Solicitar al NAO decir una frase se realizara mediante el método decirFrase(), al cual se le pasará como argumento la frase que queremos que diga el NAO.

Los métodos con que nos permiten realizar movimientos predefinidos son (si no se tiene un parentBroquer o el robot ya está realizando alguna acción el método no realizara la petición):

- accionMedirGlucosa();
- accionComer();
- accionPinchate();
- accionCorrer();
- accionTumbarse();
- accionSentarse();
- accionLevantarse();
- posicionParada();
- accionChocarMano();
- accionSaludo(int num);
- accionDespedida(int num);
- accionQUETALESTAS(int num);

Los métodos que aceptan un argumento de tipo entero cambiaran el texto que dicen dependiendo del argumento pasado

4.2.5 Dispatcher

Esta clase ofrecerá los métodos necesarios para procesar los comandos entrantes al servidor, será utilizada por la clase TcpReciver. Necesitará que se le pasen las clases AccionesNAO, DatosCompartidos, TcpSender y ThreadManager para poder funcionar de forma correcta.

```
class Dispatcher
{
public:
    Dispatcher();
    int procesarComando(std::string comando);
    int setSender(TcpSender *_sender);
    int setThreadManager(ThreadManager *_th);
    int setAccionesNAO(AccionesNAO *_ac);
    int setDatosCompartidos(DatosCompartidos *_datos);

private:
    DatosCompartidos *datos;
    ThreadManager *th;
    TcpSender *sender;
    AccionesNAO *acNAO;

    int splitString(std::string comando,
                    std::vector<std::string> *splited,
                    char delimiter);
    int findChar(std::string comando, char delimiter);
    bool is_number(const std::string& s);

    std::string validarFormatoComando(std::string comando);
    int identificarTipoComando(std::string comando, std::string *arg);
    int ejecutarComando(int tipo, std::string arg);
    //metodos para procesar comandos
    int ejecutarComandoTHREAD(std::string arg);
    int ejecutarComandoDECIR(std::string arg);
    int ejecutarComandoSIMULAR(std::string arg);
    int ejecutarComandoMOVER(std::string arg);
    int ejecutarComandoLeds(std::string arg);
    int ejecutarComandoMODOSIMU(std::string arg);
    int ejecutarComandoEXACPALABRA(std::string arg);
};
```

La clase tendrá cuatro métodos que permitirán pasarle la clases que necesita, setSender(), setThreadManager(), setAccionesNAO() y setDatosCompartidos(). Estos métodos tendrán que ser llamados antes de que le clase se ponga a procesar comandos, ya utilizara las clases pasdas con ese propósito.

Ofrecerá una serie de métodos privados (auxiliares) que facilitaran el tratamiento de cadenas de texto:

- splitString(): separara una String por el delimitador que se indique.
- findChar(): nos dirá en que posición de la cadena se encuentra el delimitador indicado.
- Is_number(): nos dirá si el String pasado es un número.

El método procesarComando() será el método público y principal que permitirá procesar una comando entrante, será llamado por la clase TCPreciver cuando reciba un nuevo comando.

```
int Dispatcher::procesarComando(std::string comando)
{
    if(th == NULL || sender == NULL || acNAO == NULL || datos == NULL)
        return -2;
    std::string argCom,comandoValidado;

    comandoValidado = validarFormatoComando(comando);
    if( !comandoValidado.empty() )
    {
        int tipoCom = identificarTipoComando(comandoValidado,&argCom);
        //tenemos comando identificado y argumentos
        ejecutarComando(tipoCom, argCom);
    }
}
```

Este método solo se ejecutará si se han pasado de forma correcta las 4 dependencias que necesita la clase. Con que una sola no sea correcta el método lanzará error.

En caso de que estén todas las dependencias bien resueltas, el método lo primero que realizaría es una comprobación del formato del comando que se quiere ejecutar, utilizara el método validarFormatoComado().

Con el formato validado de forma correcta se utilizara el método identificarTipoComando(), el cual nos dirá que comando tenemos que ejecutar.

Por último se utilizara el método ejecutarComado() pasándole el tipo de comando y los argumentos para realizar la ejecución del comando.

Para poder añadir nuevos comandos se deberán seguir los siguientes pasos (ejemplo comando PRUEBA):

1. Añadir el tipo de comando a la lista de DEFINES en el ".h".

```
#define DECIR 1
#define SIMULAR 2
#define MOVER 3
#define THREAD 4
#define LED 5
#define MODOSIMU 6
#define EXACPALABRA 7
#define PRUEBA 8
```

2. Añadir la búsqueda del tipo del comando en el método identificarTipoComando().

```
int Dispatcher::identificarTipoComando(std::string comando, std::string *arg)
{
    int tipo = -1;
    std::vector<std::string> split;

    //miramos palabra inicio comando no repetida
    splitString(comando,&split,':');

    if(split.size() != 2)
        return -2;

   if ( split.at(0).compare("DECIR") == 0 )
        tipo = DECIR;
   if ( split.at(0).compare("SIMULAR") == 0 )
        tipo = SIMULAR;
   if ( split.at(0).compare("MOVER") == 0 )
        tipo = MOVER;
   if ( split.at(0).compare("THREAD") == 0 )
        tipo = THREAD;
   if ( split.at(0).compare("LED") == 0 )
        tipo = LED;
   if ( split.at(0).compare("MODOSIMU") == 0 )
        tipo = MODOSIMU;
   if ( split.at(0).compare("EXACPALABRA") == 0 )
        tipo = EXACPALABRA;
   if ( split.at(0).compare("PRUEBA") == 0 )
        tipo = PRUEBA;

    arg->assign(split.at(1));
    return tipo;
}
```

3. Añadir en la función ejecutarComando() un nuevo “case” con el tipo del comando y la llamada a la función específica que ejecutara el comando pasándole los argumentos.

```
int Dispatcher::ejecutarComando(int tipo, std::string arg)
{
    switch (tipo) {
        case DECIR:
            ejecutarComandoDECIR(arg);
            break;
        case SIMULAR:
            ejecutarComandoSIMULAR(arg);
            break;
        case MOVER:
            ejecutarComandoMOVER(arg);
            break;
        case THREAD:
            ejecutarComandoTHREAD(arg);
            break;
        case LED:
            ejecutarComandoLeds(arg);
            break;
        case MODOSIMU:
            ejecutarComandoMODOSIMU(arg);
            break;
        case EXACPALABRA:
            ejecutarComandoEXACPALABRA(arg);
            break;
        case PRUEBA:
            ejecutarComandoPRUEBA(arg);
            break;
    }
    return -1;
}
```

4. Crear la función específica que ejecutara el nuevo comando

```
int Dispatcher::ejecutarComandoPRUEBA(std::string arg)
{
    //llamadas necesarias para que se ejecute el comando
}
```

4.2.6 Runnable

El propósito de esta clase es servir de interfaz común para todas las clases que quieran encapsular un hilo de ejecución y que puedan ser controladas por el ThreadManager. Como se puede ver tendrá 4 métodos virtuales que tendrán que sobrescribir las clases cuando hereden de esta clase (podrán sobrescribirlos en blanco).

```
#define PARADO 0
#define CORRIENDO 1
#define PAUSADO 2

class Runnable
{
    public:
        Runnable();
        virtual ~Runnable();

        int join();
        int getEstado();
        virtual void pausar() = 0;
        virtual void desPausar() = 0;
        virtual void pararThread() = 0;
        int start();
        void setEstadoHilo(int estado);

    private:
        int estado;
        boost::thread *hilo;
        virtual void run() = 0;
        boost::condition_variable cond;
        boost::mutex mut;
        bool pausa;

    protected:
        boost::thread* getHilo();
        int esperarCondicion();
        int liberarCondicion();

};
```

La clase tendrá un atributo boost::thread que será el encargado de proporcionarnos el hilo de ejecución. Contará con una variable “estado” que nos indicará el estado en el que se encuentra el hilo. Y además tendrá unas variables que nos facilitaran pausar el hilo: “cond”, “mut” y “pausa”, con la cuales podremos realizar una espera bloqueante sobre una condición.

Para lanzar el hilo, la clase contara con el método start(), que pondrá en marcha el hilo pasándole la función run() para que se ejecute en el hilo (que tendrá que ser sobrescrita en cada

clase). Este método será llamado siempre por la clase ThreadManager, que será la encargada de controlar el estado de los hilos.

```
int Runnable::start()
{
    if(estado != CORRIENDO && estado != PAUSADO && hilo == NULL)
    {
        hilo = new boost::thread(boost::bind(&Runnable::run,this));
        estado = CORRIENDO;
        return 1;
    }
    return -1;
}
```

Ofrecerá dos métodos esperarCondicion() y liberarCondicion() que facilitaran un punto donde pausar y despasar el hilo de ejecución, mediante el uso de variables condición. Cuando se esté sobrescribiendo el método run() se podrá llamar a esperarCondicion() donde se quiera parar el hilo.

```
int Runnable::esperarCondicion()
{
    pausa  =  true;
    estado = PAUSADO;
    boost::unique_lock<boost::mutex> lock(mut);
    while(pausa)
    {
        cond.wait(lock);
    }
}

int Runnable::liberarCondicion()
{
    boost::lock_guard<boost::mutex> lock(mut);
    estado = CORRIENDO;
    pausa=false;
    cond.notify_one();
}
```

Los métodos virtuales pausar(), desPausar() y pararThread() deberán usarse como indicadores (flags) de que se quiere cambiar el estado y dentro del código de ejecución del hilo llamar a esperarCondicion() o parar el hilo donde se pueda realizar una parada o pausa y siempre que se solicite parada con el método pararThread() o pausa con el método pausar().

El método setEstadoHilo() nos permite cambiar de forma manual el indicador de estado, se usara siempre que el mismo hilo necesite pausarse o pararse a él mismo, ya que no podrá realizar las llamadas utilizando la clase ThreadManager (las cuales actualizan el estado automáticamente).

El método join() proporciona un punto donde quedar a la esperar hasta que el hilo se acabe. Dentro de la función run() no podrá ser llamada esta función ya que el mismo hilo no se puede quedar a la espera de si mismo. Este método será utilizado por la clase ThreadManager para

parar hilos de forma correcta, por lo que en el método run tampoco se podrá utilizar la función de parada que nos ofrece el ThreadManager.

```
int Runnable::join()
{
    if(hilo != NULL)
    {
       hilo->join();
       estado = PARADO;
       delete hilo;
       hilo = NULL;
       return 1;
    }
    return -1;
}
```

4.2.7 ThreadManager

Se encargará del manejo de todos los hilos presentes en el sistema. Tratara a todos los hilos por igual utilizando los métodos que nos obliga a sobrescribir la clase Runnable y no se preocupara por la implementación particular de cada hilo.

```
class ThreadManager
{
private:
    DatosCompartidos *datosC;
    std::map<std::string,Runnable*>  hilos;
    std::map<std::string,Runnable*>  hilosExcluyentes; //solo puede esta
    std::string hiloExcluyenteCorriendo;

    boost::mutex mut;
    std::string estadoHilos;
public:
    ThreadManager();

    int setDatosCompartidos(DatosCompartidos * d);

    //funciones que nos permiten añadir hilos al sistema
    int addHilo(std::string id,Runnable *rclass);
    int addHiloExcluyente(std::string id,Runnable *rclass);

    //funciones que nos permiten arrancar y parar los hilos del sistema
    int arrancar(std::string id);
    int parar(std::string id);

    //hilos que nos permiten solicitar pausar o despausar a los hilos de
    int pausar(std::string id);
    int desPausar(std::string id);

    int restart(std::string id);
    bool isHilo(std::string id);
    void actualizarDatosEstadoHilos();

    std::string getEstadoHilos();
};
```

Dentro de la clase los hilos serán identificados mediante un ID (String) que será único para cada hilo. Como vemos la clase tendrá dos atributos “hilos” e “hilosExcluyentes”, los cuales serán de tipo std::map que nos permitirán almacenar pares con: valor el puntero a la clase que herede de Runnable y clave el ID único del hilo.

Dependiendo en donde este almacenado el hilo, se le considerará excluyente o no, de todos los hilos que estén almacenados en el atributo “hilosExcluyentes”, solo se podrá estar uno en ejecución a la vez. Mientras que los que estén en “hilos” podrán realizar ejecución simultánea.

Esta clase además tendrá un atributo “estadoHilos”, en el cual se almacenará el estado de los hilos, indicando la ID del hilo seguida de su estado. El dato será añadido a los datos compartidos marcado como enviable.

La clase contara con un método isHilo(), que se le pasara como argumento un ID y nos indicara si ya existe el ID en algunos de los map (“hilos” e “hilosExcluyentes”).

Para poder añadir hilos a la clase tenemos las funciones addHilo() y addHiloExcluyente(), a las cuales se les pasa el ID del hilo y un puntero a la clase Runnable. Si el ID pasado está ya en uso se lanzará error, en otro caso se añadirá el hilo a la lista que toque y se actualizará el estado de los hilos.

```
int ThreadManager::addHilo(std::string id,Runnable *rclass)
{
    boost::lock_guard<boost::mutex> lock(mut);
        if( isHilo(id) )
            return -2; //id ya existe en hilos o hilosExcluyentes
        std::pair<std::map<std::string,Runnable*>::iterator,bool> ret;
        ret = hilos.insert(std::make_pair(id, rclass));
        if (ret.second==false)
            return -1; //el elemento ya existe
        actualizarDatosEstadoHilos();
        return 1;
}

int ThreadManager::addHiloExcluyente(std::string id, Runnable *rclass)
{
    boost::lock_guard<boost::mutex> lock(mut);
        if( isHilo(id) )
            return -2; //id ya existe en hilos o hilosExcluyentes
        std::pair<std::map<std::string,Runnable*>::iterator,bool> ret;
        ret = hilosExcluyentes.insert(std::make_pair(id, rclass));
        if (ret.second==false)
            return -1; //el elemento ya existe
        actualizarDatosEstadoHilos();
        return 1;
}
```

Para poner en marcha un hilo se tendrá que utilizar el método arrancar() pasándole como argumento el ID del hilo deseado. Si el hilo que se quiere arrancar es excluyente y ya se encuentra un hilo excluyente en ejecución, no se encuentra el ID o ya está arrancado nos dará un error. En otro caso se arrancará el hilo y actualizará el estado de los hilos.

```
int ThreadManager::arrancar(std::string id)
{
    boost::lock_guard<boost::mutex> lock(mut);
        std::map<std::string,Runnable*>::iterator it;
        it = hilos.find(id);
        if (it != hilos.end())
        {
            if( it->second->getEstado() != CORRIENDO){
                it->second->start();
                actualizarDatosEstadoHilos();
                return  1;
            }
        }

        if(hiloExcluyenteCorriendo.empty())
        {
            it = hilosExcluyentes.find(id);
            if (it != hilosExcluyentes.end())
            {
                if( it->second->getEstado() != CORRIENDO){
                    it->second->start();
                    hiloExcluyenteCorriendo.assign(id);
                    actualizarDatosEstadoHilos();
                    return  1;
                }
            }
        } else {
            return -2;
        }

        return -1;
}
```

En caso de querer parar un hilo, se utilizará el método parar() indicándole la ID del hilo que se desea parar. Si el ID no se encuentra o el hilo ya está parado nos devolverá un error. En otro caso solicitará al hilo que se pare, esperará hasta que se pare y actualizará el estado de los hilos.

```
int ThreadManager::parar(std::string id)
{
    boost::lock_guard<boost::mutex> lock(mut);
        std::map<std::string,Runnable*>::iterator it;
        it = hilos.find(id);
        if (it != hilos.end())
        {
            if( it->second->getEstado() != PARADO){
                Runnable *aux = it->second;
                aux->pararThread();
                aux->join();
                actualizarDatosEstadoHilos();
                return 1;
            }

        }

        it = hilosExcluyentes.find(id);
        if (it != hilosExcluyentes.end())
        {
            if( it->second->getEstado() != PARADO){
                Runnable *aux = it->second;
                aux->pararThread();
                aux->join();
                hiloExcluyenteCorriendo.clear();
                actualizarDatosEstadoHilos();
                return  1;
            }
        }
        return -1;
}
```

Si se quiere pausar un hilo, se llamara al método pausar() indicándole el ID del hilo. Este método solicitara al hilo que se pause cuando pueda. Si el ID no existe o ya está parado el hilo nos lanzara un error. En otro caso realiza la petición de pausa y actualizará el estado de los hilos.

```
int ThreadManager::pausar(std::string id)
{
    boost::lock_guard<boost::mutex> lock(mut);
        std::map<std::string,Runnable*>::iterator it;
        it = hilos.find(id);
        if (it != hilos.end())
        {
            if( it->second->getEstado() != PARADO){
                Runnable *aux = it->second;
                aux->pausar();
                actualizarDatosEstadoHilos();
                return 1;
            }

        }

        it = hilosExcluyentes.find(id);
        if (it != hilosExcluyentes.end())
        {
            if( it->second->getEstado() != PARADO){
                Runnable *aux = it->second;
                aux->pausar();
                actualizarDatosEstadoHilos();
                return 1;
            }
        }
        return -1;
}
```

Para poder des-pausar un hilo, se tendrá que utilizar el método desPausar() pasándole el ID del hilo. Si el ID no existe o no está parado el hilo nos lanzara un error. En otro caso realiza la petición para des-pausar y actualizará el estado de los hilos.

```
int ThreadManager::desPausar(std::string id)
{
    boost::lock_guard<boost::mutex> lock(mut);
        std::map<std::string,Runnable*>::iterator it;
        it = hilos.find(id);
        if (it != hilos.end())
        {
            if( it->second->getEstado() == PARADO){
                Runnable *aux = it->second;
                aux->desPausar();
                actualizarDatosEstadoHilos();
                return 1;
            }

        }

        it = hilosExcluyentes.find(id);
        if (it != hilosExcluyentes.end())
        {
            if( it->second->getEstado() == PARADO){
                Runnable *aux = it->second;
                aux->desPausar();
                actualizarDatosEstadoHilos();
                return 1;
            }
        }
        return -1;
}
```

El método getEstadoHilos(), nos devolverá una cadena con todos los hilos almacenados en la clase y con su estado actual. Este método será llamado cada vez que se llame al método actualizarDatosEstadoHilos(), el cual guardara la cadena devuelto por getEstadosHilos() en los datos compartidos.

4.2.8 TCPServer

En esta clase se encapsulará el hilo que se encargará de ofrecer conexión vía internet y gestionará un par de hilos creados con el mismo propósito. Este hilo heredara del método Runnable.

```
class TCPServer : public Runnable
{
public:
    TCPServer(int _puerto);
    void run();
    void pararThread();
    void pausar();
    void desPausar();
    int inicializarServidor();
    void mandarRespuesta(std::string resp);

    int setDatosCompartidos(DatosCompartidos * d);
    int setDispatcher(Dispatcher * dispa);
private:
    DatosCompartidos *datosC;
    Dispatcher *disp;
    int sockServidor; //sockect servidor
    socklen_t clilen;
    struct sockaddr_in serv_addr, cli_addr;

    TcpReciver *reciverCliente;
    TcpSender *senderCliente;
    boost::thread *hiloClienteSend;
    boost::thread *hiloClienteRecv;
    bool pararLoop;

    void joinHilos();
    int acceptConnection();
};
```

En el constructor de esta clase se creará un socket de tipo TCP/IP en el puerto que se le indique. Además, esta clase se encargará de gestionar dos hilos que se encargaran de: uno de estar pendiente de los comandos que envía el cliente y otro que estará enviando datos de forma periódica al cliente.

Se tienen dos atributos de tipo boost:thread los cuales nos darán soporte para crear los hilos:

- hiloCLienteSend: ejecutara un método ofrecido por la clase TcpSender.
- hiloCLienteRecv: ejecutara un método ofrecido por la clase TcpReciver.

Tanto la clase TcpReciver como TcpSender se instanciarán dentro del constructor de esta clase para que estén disponibles cuando sea necesario lanzar los hilos. Estas dos clases no heredaran de Runnable, ya que no se tiene necesidad de que sean gestionadas por el ThreadManager, si no que serán gestionadas por TCPServer.

Los método pararThread(), desPausar() y pausar() serán sobrescritos en blanco, ya que no se quiere que esta clase se pueda parar o pausar, una vez se añada al ThreadManager y se lance siempre estará ejecutándose.

El método que si tendrá que sobrescribir es el método run(), ya que es el que ejecutara el hilo.

```
void TCPServer::run()
{
    while(pararLoop)
    {
       int aux = acceptConnection();
       if(aux != -2 && aux != -1)
       {
           if(hiloClienteSend == NULL && senderCliente != NULL && reciverCliente != NULL)
           {

                hiloClienteSend = new boost::thread(boost::bind(&TcpSender::escribir,senderCliente));
                if(this->datosC != NULL)
                {
                    senderCliente->setDatosCompartidos(datosC);
                }
                sleep(0.5);
                hiloClienteRecv = new boost::thread(boost::bind(&TcpReciver::leer,reciverCliente));
                if(this->disp != NULL)
                {
                    reciverCliente->setDispatcher(disp);
                }
           }
       }
    }
    close(sockServidor);
}


int TCPServer::acceptConnection()
{
    int auxSocket = accept(sockServidor, (struct sockaddr *)&cli_addr, &clilen);

    if (auxSocket < 0) {
         return -1;
    }

    //en caso de tener ya un cliente rechazamos al nuevo
    if(senderCliente->isCliente() == true || reciverCliente->isCliente() == true)
    {
        close(auxSocket);
        return -2;
    }

    //liberamos los hilos del anterior cliente
    joinHilos();

    //indicamos cual es el socket del nuevo cliente
    senderCliente->setCliente(auxSocket);
    reciverCliente->setCliente(auxSocket);

    return auxSocket;
}
```

En este método se creará un bucle infinito, que estará en espera (las esperas no consumirán recursos, si no que bloquearan al hilo) por clientes, en cuanto se conecte un cliente se lanzaran los hilos TcpSender y TcpReciver los cuales se encargaran de la comunicación con el cliente, este cliente volverá a quedarse a la espera de clientes.

Si un cliente pide conexión cuando ya se está atendiendo a un cliente se le rechazará la conexión y se seguirá esperando más clientes.

4.2.9 TcpReciver

La clase encapsulara el comportamiento del hilo hiloClienteRecv, que será gestionado por la clase TCPServer, la cual le pasara un puntero a la clase Dispatcher para que pueda procesar los comandos entrantes.

```
#define BUFFER_SIZE 100

class TcpReciver
{
public:
    TcpReciver();
    ~TcpReciver();
    void leer();
    bool isCliente();
    void setCliente(int _sockCliente);
    void pararLoop();
    int setDispatcher(Dispatcher *dispa);
private:
    int leerComando(std::string *comando);
    bool pararLoopCliente;
    int sockCliente; //cliente conectado

    bool clienteConectado;
    Dispatcher *disp;
};
```

Cada vez que se tenga un cliente nuevo, se actualizará el atributo “sockCliente” mediante una llamada al método setCliente(), la llamada la realizara la clase TCPServer en cuanto se conecte un nuevo cliente.

El método que ejecutara el hilo será leer(), el cual estará en bucle infinito esperando y procesando comandos.

```
void TcpReciver::leer()
{
    std::string comando;

    this->pararLoopCliente = true;
    this->clienteConectado = true;

    while(pararLoopCliente)
    {
        comando.clear();
        int aux = this->leerComando(&comando);
        if(aux == -2)
            continue;
        if(aux == -1){
            close(sockCliente);
            this->clienteConectado = false;
            return;
        }
        disp->procesarComando(comando);
        char *buf;
        buf = new char[8640];
        recv(clienteConectado, buf, BUFFER_SIZE, MSG_DONTWAIT);
        delete(buf);

    }
    close(sockCliente);
    this->clienteConectado = false;
}
```

En cuanto se detecte que el cliente se ha desconectado, este hilo cerrara la conexión y terminara indicando en el atributo “clienteConectado” que ya no está gestionando ninguna conexión.

El método leerComando(), se encargará de estar escuchando del socket, y solo devolverá un comando cuando se exceda el buffer o encuentre el carácter de fin de comando “;”.

```
int TcpReciver::leerComando(std::string *comando)
{
    size_t buf_idx = 0;
    char chunk[BUFFER_SIZE];
    std::memset(chunk ,0 , BUFFER_SIZE);

    while(buf_idx < BUFFER_SIZE){

        if( recv(sockCliente , &chunk[buf_idx] , 1,0) < 1 ) {
            return -1; //cliente desconectado
        } else {

           if (';' == chunk[buf_idx]){
               comando->assign(chunk);
               return 1;
           }
           buf_idx++;
        }
    }
    return -2; //buffer comando lleno, lo desechamos

}
```

4.2.10 TcpSender

La clase encapsulara el comportamiento del hilo hiloClienteSend, que será gestionado por la clase TCPServer, la cual le pasara un puntero a la clase DatosCompartidos para que pueda obtener los datos a enviar.

```
class TcpSender
{
public:
    TcpSender();
    ~TcpSender();
    void escribir();
    bool isCliente();
    void setCliente(int _sockCliente);
    void pararLoop();

    void mandarRespuesta(std::string respuesta);

    int setDatosCompartidos(DatosCompartidos * d);
private:
    DatosCompartidos *datosC;
    bool pararLoopCliente;
    int sockCliente; //cliente conectado
    bool clienteConectado;

    int numDatosEnv;

};
```

Cada vez que se tenga un cliente nuevo, se actualizará el atributo “sockCliente” mediante una llamada al método setCliente(), la llamada la realizara la clase TCPServer en cuanto se conecte un nuevo cliente.

El método que ejecutara el hilo será escribir(), el cual estará en bucle infinito obteniendo los datos y enviándolos cada segundo, en caso de no tener datos enviables se enviara el comando solo con el prefijo “PER#” y el carácter de final de cadena ‘;’.

```
26  void TcpSender::escribir()
27  {
28      std::stringstream msg;
29      this->pararLoopCliente = true;
30      this->clienteConectado = true;
31      double auxD;
32      std::string auxS;
33      int auxI;
34      std::vector<std::string> auxDatos;
35
36      while(pararLoopCliente)
37      {
38
39          if( datosC != NULL)
40              auxDatos = datosC->getDatosEnviables();
41
42          msg.str("");
43          msg << "PER#";
44
45          for(int i = 0; i < auxDatos.size(); i++)
46          {
47              std::string id = auxDatos.at(i);
48              msg << id;
49              msg << "~";
50              if(datosC->getTipoDato(id) == INT){
51                  datosC->getData(id,auxI);
52                  msg << auxI;
53              }
```

```
            if(datosC->getTipoDato(id) == DOUBLE){
                datosC->getData(id,auxD);
                msg << auxD;
            }
            if(datosC->getTipoDato(id) == STRING){
                datosC->getData(id,auxS);
                msg << auxS;
            }
            msg << "/";
        }
        if(auxDatos.size() > 0)
            msg.seekp(-1, std::ios_base::end);

        msg << ";"; // caracter de fin de mensaje
        if(msg.str().empty())
            continue;
        if( send(sockCliente , msg.str().c_str(), std::strlen( msg.str().c_str() ) , 0) < 0)
        {
            this->clienteConectado = false;
            close(sockCliente);
            return;
        }
        boost::this_thread::sleep(boost::posix_time::milliseconds(1000));
    }
    this->clienteConectado = false;
    close(sockCliente);
}
```

En este método se obtendrán todos los datos marcados como enviables dentro de la clase DatosCompartidos y creará la cadena que se enviará al cliente. Como no sabemos cuál es el tipo de cada ID, necesitaremos ir viendo el tipo de cada dato para poder tratarlo de forma correcta.

En cuanto se detecte que el cliente se ha desconectado, este hilo cerrara la conexión y terminara indicando en el atributo “clienteConectado” que ya no está gestionando ninguna conexión.

Esta clase también proporcionara un método para poder enviar respuesta no periódicas, el método es mandarRespuseta().

```
void TcpSender::mandarRespuesta(std::string respuesta)
{
    std::stringstream msg;
    msg << "RES#";
    msg << respuesta;
    msg << ";";
    if( send(sockCliente , msg.str().c_str(), std::strlen( msg.str().c_str() ) , 0) < 0)
    {
        this->clienteConectado = false;
        close(sockCliente);
        return;
    }
}
```

Enviara un mensaje con el prefijo “RES#” seguido del mensaje que se le pase por argumento, este método será usado por la clase Dispatcher para enviar respuesta a los comandos recibidos cuando sea necesario.

4.2.11 Simulador

Esta será la clase que implemente el simulador de diabetes. Sera una clase que encapsule el comportamiento de un hilo que gestionará el ThreadManager por lo que la clase tendrá que heredar de Runnable.

```
32  class Simulador : public Runnable
33  {
34  private:
35      data_struct *d;
36      int modo;
37      bool flagPausar;
38      bool flagThread;
39
40      float bw;
41      float tmaxG;
42      float vg;
43      float egp0;
44      float f01;
45
46      //Entrada con valores iniciales
47      alglib::real_1d_array x;
48      alglib::real_1d_array t;
49
50      //  Variables internas simulador
51      alglib::real_2d_array xtbl;
52      alglib::real_1d_array ttbl;
53
54      //  BOLUS Y CHO
55      double bolus;
56      double cho;
57      double exercise[3];                 // [0] t
58      bool ejercicio;//indicacion de realizacion d
59
60      double fcorreccion;      //factor de correcc
61
62      int pasosSimulacion;
63      int marca;
64      int marcaLimite;
65      double tiempoUltimaSimu;
66      double tiempoIniPausado;
67      double tiempoTotalPausado;
```

```
68
69      double SimVolGlucosaTotal[121];
70
71      //funciones que utiliza el thread para realizar la
72      void simular();
73      void calcularGlucosa();
74      void calcularMarcaActual();
75      void actualizarEstadoInicial(int marca);
76      void run();
77      void ConfigurarSimulador();
78      DatosCompartidos *datosC;
79      datosSimulador datosSimu;
80
81  public:
82      Simulador();
83      ~Simulador();
84      void pararThread();
85      void setModo(int modo);
86      void pausar();
87      void desPausar();
88      int setDatosCompartidos(DatosCompartidos * d);
89  };
90
```

La clase sobrescribirá los métodos de pausar(), desPausar() y pararThread(), que permitirán al hilo pausarse y parar. Estos métodos nos permitirán marcar un flag para que el bucle principal sea consciente de que se ha solicitado parada o pausa. El método despausar() liberara la condición de parada para que el hilo se ponga en marcha.

```
void Simulador::pararThread()
{
    this->flagThread = false;
    if(flagPausar == true){
        flagPausar = false;
        this->liberarCondicion();
    }
}

void Simulador::pausar()
{
    flagPausar = true;
    //actualizamos el estado del hilo de forma inmediata,
    setEstadoHilo(PAUSADO);
}

void Simulador::desPausar()
{
    if(flagPausar == true){
        flagPausar = false;
        this->liberarCondicion();
    }
}
```

En el método run() se implementara toda la lógica del simulador de diabetes. Al inicio del hilo se realizarán las configuraciones que necesita el simulador para trabajar, mirará en DatosCompartidos el modo para el simulador y realizará la simulación inicial (siempre con los mismos valores).

```
void Simulador::run()
{

    try
    {

        int modoAux;
        if( datosC->getData("MODOSIMU",modoAux) == 1){
            this->modo = modoAux;
        } else {
            this->modo = 2;
        }
        this->ConfigurarSimulador();
        this->simular();
        this->calcularGlucosa();
        tiempoUltimaSimu = time(NULL);

        while (flagThread){

           //mirar pausa
           if(flagPausar == true){
               tiempoIniPausado=time(NULL);
               if( datosC != NULL)
                   datosC->modifyData("GLUCOSA",(double)-999);
               this->esperarCondicion();
               tiempoTotalPausado=tiempoTotalPausado+difftime(time(NULL),tiempoIniPausado);
           }

           //miramos si tenemos nuevos datos para realizar la simulacion
           if( datosC != NULL)
              this->datosSimu=datosC->getDatosSimulacion();

           if(datosSimu.bolus == 0 && datosSimu.cho == 0 && datosSimu.ejercicio == false){

               //obtenemos marca altual del simulador
               this->calcularMarcaActual();
               if( datosC != NULL)
                    datosC->modifyData("GLUCOSA",(double)SimVolGlucosaTotal[marca]);
               if (marca>=marcaLimite){  //Se ejecutará una simulacion sin entradas si pasan 5 minutos y
                   this->actualizarEstadoInicial(xtbl.rows()-1);
                   this->simular();
                   this->calcularGlucosa();
```

```
159
160                     tiempoUltimaSimu = time(NULL);
161                     this->tiempoIniPausado = 0;
162                     this->tiempoTotalPausado = 0;
163
164                 } else {
165                     sleep(1);
166                 }
167             } else {
168                 //valores de bolus, cho o ejercicio nuevos, actualizamos
169                 this->bolus = datosSimu.bolus;
170                 this->cho = datosSimu.cho;
171                 this->ejercicio = datosSimu.ejercicio;
172                 this->exercise[0] = datosSimu.ejercicioTiempo; //ejeTiempo
173                 this->exercise[1] = datosSimu.ejercicioIntesidad; //ejeIntensidad
174                 this->exercise[2] = datosSimu.ejercicioDuracion; //ejeDuracion
175
176                 this->actualizarEstadoInicial(this->marca);
177                 this->simular();
178                 this->calcularGlucosa();
179
180                 tiempoUltimaSimu = time(NULL);
181                 this->tiempoIniPausado = 0;
182                 this->tiempoTotalPausado = 0;
183
184             }
185
186
187         }
188         this->flagThread = true;
189         if( datosC != NULL)
190             datosC->modifyData("GLUCOSA",(double)-1000);
191     } catch(alglib::ap_error e) {
192         FILE *doc = fopen("/home/nao/naoqi/simuladorError.txt","w");
193         if (doc!=NULL){
194             fprintf(doc,"error :%s\n",e.msg.c_str());
195             fclose(doc);
196         }
197         return;
198     }
199 }
```

Una vez se ha realizado la simulación inicial, entrara en un bucle donde cada segundo se comprobará si se tiene que realizar una simulación personalizada, en caso negativo, el simulador actualizará el valor de la glucosa dentro de los DatosCompartidos y comprobara si han paso 5 minutos, en caso afirmativo realizara una nueva simulación con parámetros estándar.

Los métodos actualizarEstadoInicial(), simular(), calcularGlucosa() y calcularMarcaActual() se han creado para no tener que agrandar mucho el bucle principal del hilo, ya que se utilizan en varias ocasiones.

- actualizarEstadoInicial(): actualizara el estado inicial que necesita el simulador, como argumento se le tiene que pasar la marca en la que estamos si la nueva simulación es con parámetros nuevos, en otro caso se le pasara el numero de filas de la variable xtbl -1.
- simular(): realizara las operaciones matemáticas del simulador.
- calcularGlucosa(): calculara el valor de la glucosa para cada instante de la simulación.
- calcularMarcaActual(): calculara el punto por el que vamos en la simulación dentro de la ventana de 5 minutos.

4.2.12 Interaccion

Esta clase heredara de Runnable y encapsulara al hilo que se encargara de realizar una interacción básica con el usuario.

```
class Interaccion : public Runnable
{
public:
    Interaccion();
    ~Interaccion();
    void pararThread();
    void pausar();
    void desPausar();

    int setAccionesNAO(AccionesNAO *_ac);
    int setDatosCompartidos(DatosCompartidos *_datos);
private:
    AccionesNAO *ac;
    DatosCompartidos *datos;
    std::string ultimaPalabra;
    std::string ultimaPostura;
    int contador;
    void actualizarContador();

    std::vector<std::string> wordlist;
    void populateWordList();

    double *glucosa;
    double mirarGlucosa();
    void guardarGlucosa(double glu);
    std::string doubleToString(double aux);

    double exact;
    void run();
    bool pararLoop;
};
```

A este hilo se le permitirá parar y arrancar pero no pausar, por lo que los métodos pausar() y desPausar() se sobrescribirán en blanco. En cambio el método paraThread() se sobrescribirá para que indique al bucle principal que necesita pararse.

En el método run(), se creara un bucle el cual estará mirando la glucosa actual del simulador y dependiendo del valor el robot dirá una frase u otra (en este hilo no se realizan peticiones al simulador), después se pondrá a la espera de una serie de frases y palabras.

```
void Interaccion::populateWordList()
{
    wordlist.push_back("hola");
    wordlist.push_back("adios");
    wordlist.push_back("como te llamas");
    wordlist.push_back("que tal estás");
    wordlist.push_back("sientate");
    wordlist.push_back("levantate");
    wordlist.push_back("choca el puño");
    wordlist.push_back("salta");
    wordlist.push_back("tumbate");
    wordlist.push_back("dime tu glucosa");
}
```

Dependiendo de la frase que se le diga al robot, este realizara una serie de acciones u otras. Estará dentro de este bucle hasta que se le ordene parar.

```
85  void Interaccion::run()
86  {
87      if(datos == NULL || ac == NULL)
88      {
89          setEstadoHilo(PARADO);
90          return;
91      }
92      this->pararLoop = true;
93      std::string palabraRec;
94      int respEspera;
95      float exac;
96
97      populateWordList();
98      datos->setData("INTERACCION",std::string("-"),true);
99
100     ac->setThreadBock(true);
101     ac->accionLevantarse();
102     while(pararLoop)
103     {
104         double auxExac;
105         datos->getData("EXACPALABRA",auxExac);
106         this->exact =auxExac;
107
108         int probability = rand() % 10 + 1;
109
110         std::stringstream msg;
111         msg << "exac:";
112         msg << exact;
113         msg << "prob:";
114         msg << probability;
115         datos->modifyData("INTERACCION",(std::string)msg.str());
116         mirarGlucosa();
117
118         if(glucosa[0] > 75 && glucosa[0] < 150){ ...
125         } else if( glucosa[0] < 75 ){ ...
132         } else if( glucosa[0] >= 300 ){ ...}
139
```

```
140         if(ac != NULL)
141             respEspera = ac->esperarPalabra(wordlist,10,&palabraRec,exac);
142         switch(respEspera)
143         {
144             case -1: break; //despertado de forma manual
145             case -2:break;
146             case 1://palabra reconicida
147                 if(exac < exact){ ...}
151                 if( palabraRec.compare("hola") == 0 ){ ...}
156
157                 if( palabraRec.compare("adios") == 0 ){ ...}
162
163                 if( palabraRec.compare("como te llamas") == 0 ){ ...}
168
169                 if( palabraRec.compare("que tal estas") == 0 ){ ...}
174
175                 if( palabraRec.compare("sientate") == 0 ){ ...}
201
202                 if( palabraRec.compare("levantate") == 0 ){ ...}
217
218                 if( palabraRec.compare("choca el puño") == 0 ){ ...}
223
224                 if( palabraRec.compare("salta") == 0 ){ ...}
232
233                 if( palabraRec.compare("tumbate") == 0 ){ ...}
259                 if(palabraRec.compare("dime tu glucosa") == 0 && exac > 0.45){ ...}
263                 break;
264         }
265
266     }
267     ac->posicionParada();
268     ac->setThreadBock(false);
269     if(datos != NULL)
270         datos->deleteData("INTERACCION");
271
272 }
```

Al inicio del hilo se realizan las configuraciones necesarias y se indica en la clase AccionesNAO que se requiere el uso exclusivo de la clase.

4.2.13 Escenario

Esta clase heredara de Runnable y encapsulara al hilo que se encargara de realizar el escenario principal, en el que el usuario interactuara con el robot y con el simulador de diabetes.

```
class Escenario : public Runnable
{
private:
    AccionesNAO *acNAO;
    DatosCompartidos *datos;
    bool pararLoop;
    double *glucosa;
    int fase;
    double exactitud;

    //variables control para la fase2
    std::vector<std::string> getWordlistFase2();
    void fase2();
    int numHambre;
    int numEjercicio;
    int estadotaller;
    std::string ultimaPalabra;

    //variables control para la fase3
    std::vector<std::string> getWordlistFase3();
    int tiempoUltimaPeticionSimu;
    bool iniFase3;
    void fase3();
    int estado;
    int contador;
    void actualizarContador();
    int numerorandom;
    std::string ultimaPostura;

    bool pausa;
    double mirarGlucosa();
    void guardarGlucosa(double glu);
    void run();
public:
    Escenario();
    ~Escenario();
    void pararThread();
    void pausar();
    void desPausar();

    int setAccionesNAO(AccionesNAO *_ac);
    int setDatosCompartidos(DatosCompartidos *_datos);

};
```

A este hilo se le permitirá parar y arrancar pero no pausar, por lo que los métodos pausar() y desPausar() se sobrescribirán en blanco. En cambio el método paraThread() se sobrescribirá para que indique al bucle principal que necesita pararse.

En este hilo se tendrán dos fases, las cuales cambiarán las diferentes interacciones y respuestas entre el robot y el usuario, es un hilo con funcionalidad parecida al de Interaccion, salvo que en este se realizaran peticiones al simulador de diabetes y se actuara en torno a los valores que nos devuelven.

Pero el esquema general del bucle principal es el mismo.

1. Se mira el valor de glucosa y almacena
2. Dependiendo del valor se realiza una acción u otra
3. Se espera una frase
4. El robot responde con una serie de acciones y frases

```
71  void Escenario::run()
72  {
73      if(datos == NULL || acNAO == NULL)
74      {
75          setEstadoHilo(PARADO);
76          return;
77      }
78      acNAO->setThreadBock(true);
79      acNAO->accionLevantarse();
80      std::stringstream msg;
81      msg << -1;
82      datos->setData("ESCENARIO",msg.str(),true);
83
84      this->ultimaPalabra.assign("default");
85      this->ultimaPostura.assign("default");
86      this->pararLoop = true;
87      this->fase = 2;
88      this->numHambre = 0;
89      this->numEjercicio = 1;
90      this->estadotaller = 1;
91      this->estado = 1;
92      this->contador=1;
93      this->iniFase3 = true;
94
95      while(pararLoop)
96      {
97          mirarGlucosa();
98
99          double aux;
100         datos->getData("EXACPALABRA",aux);
101         this->exactitud =aux;
102
103         if (estado==1) {
104             //Calcular numero aleatorio entre 1 y 2
105             //  - SI numerorandom = 1: el robot dará una rec
106             //  - SI numerorandom = 2: el robot dará una rec
107             //srand(time(0)); //genera semilla basada en el
108             numerorandom=1+(rand() % 2);
109         }
110
111         switch(fase)
112         {
113             case 2:
114                 fase2();
115                 break;
116             case 3:
117                 fase3();
118                 break;
119         }
120
121     }
122
123     acNAO->posicionParada();
124     acNAO->setThreadBock(false);
125     if(datos != NULL)
126         datos->deleteData("ESCENARIO");
127 }
```

Al inicio del bucle se configuran una serie de variables que definen el estado de cada una de las fases, se modificaran durante la ejecución del hilo y lo llevaran por las diferentes respuestas y acciones que puede dar el robot.

4.3 Configuración del módulo en el NAO

En este apartado veremos cómo se tiene que añadir nuestro modulo al robot y las acciones y pasos necesarios para que al encendido del NAO cargue nuestro módulo de forma correcta.

4.3.1 Carga del modulo

Para que NAOqui pueda lanzar nuestro modulo, necesitaremos indicarle donde se encuentra alojado. Tendremos que abrir el fichero '/home/nao/naoqi/preferences/autoload.ini' donde tendremos que indicar la ruta en la que se encuentra nuestro .so en el apartado [user].

En nuestro caso la ruta para nuestro modulo seria '/home/nao/naoqi/libserver.so', si nuestro modulo no se encuentra en esta ruta entonces la cambiaremos.

Para realizar estas operaciones necesitaremos acceder al NAO mediante la red utilizando la herramienta ssh.

4.3.2 Encendido del NAO

Si ya tenemos nuestro modulo correctamente cargado en el robot, una vez se encienda se empezarán a cargar todos los módulos entre ellos el nuestro. Si todo funciona de manera correcta el robot nos tendrá que decir las siguientes frases:

1. Empezamos.
2. Lanzamos hilos.
3. Simulación lanzada.
4. Hilos arrancados.

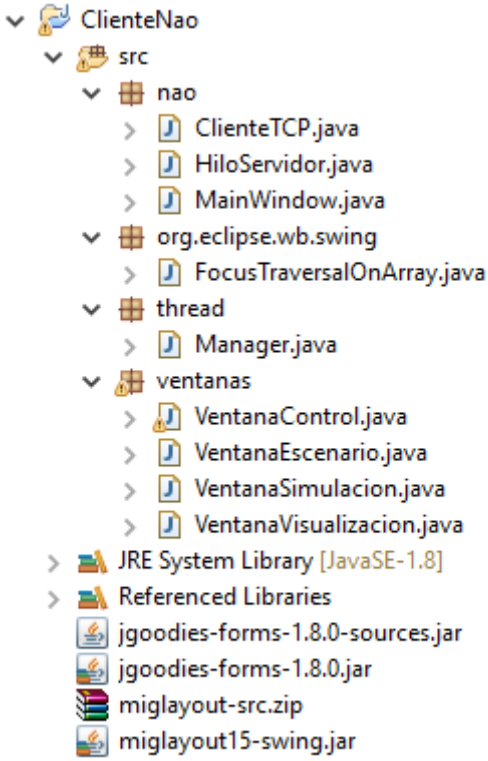
Una vez el robot diga la última frase ya tendremos el servidor totalmente desplegado y esperado por conexiones. Los hilos que estarán corriendo serán el de Simulador y el de TCPServer.

5. Cliente NAO

5.1 Proyecto de Eclipse

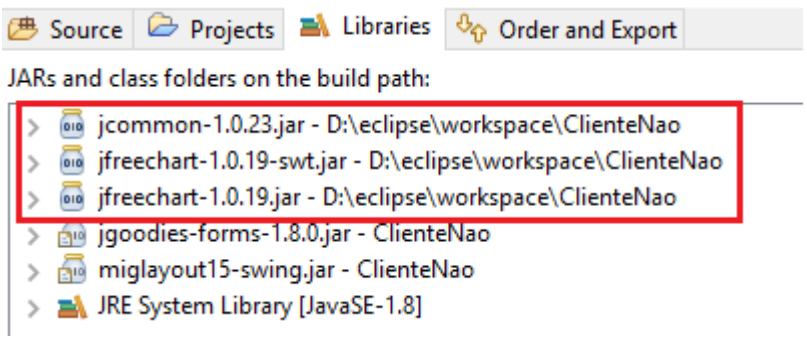
El programa que actuará como cliente para nuestro servidor alojado en el robot estará desarrollado en el lenguaje de programación Java y se ha realizado utilizando el IDE Eclipse, bajo el sistema operativo Windows 10.

La estructura de ficheros que componen el proyecto es la siguiente:



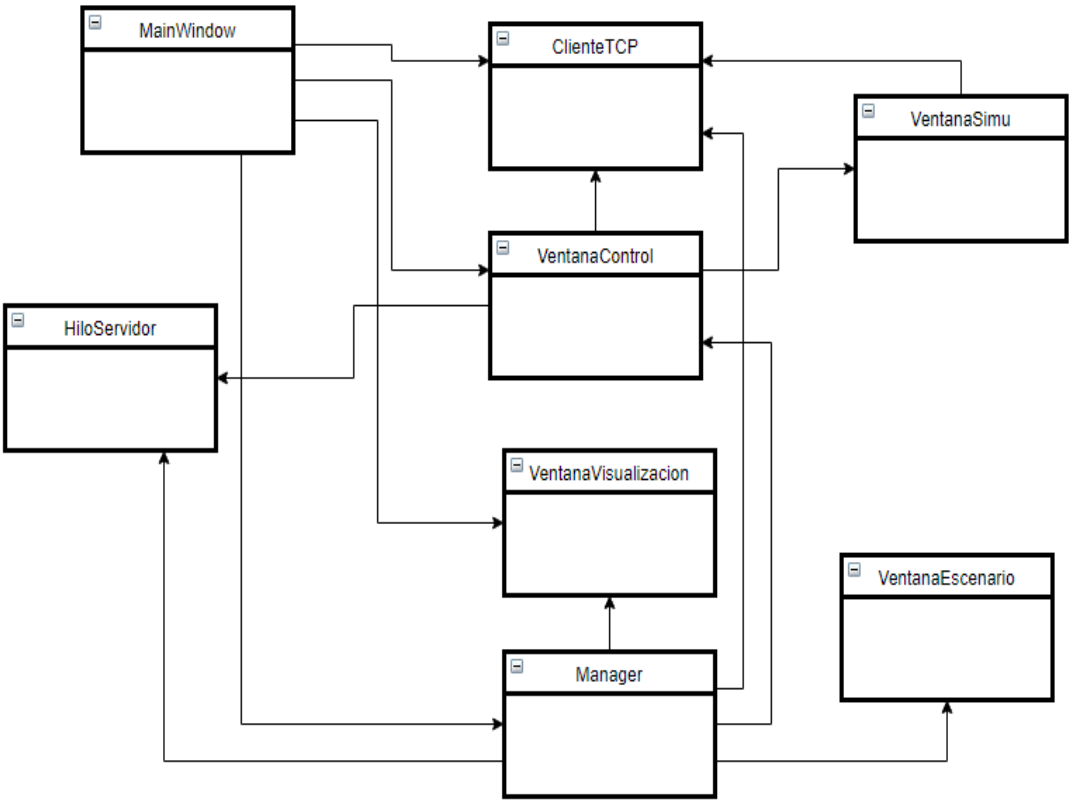
Para mayor facilidad a la hora de desarrollar las interfaces graficas de usuario se ha instalado un módulo de Eclipse llamado WindowBuilder.

Como el programa utilizara una librería de terceros llamada JFreeChart, tendremos que asegurarnos de añadirla al path de nuestro proyecto.



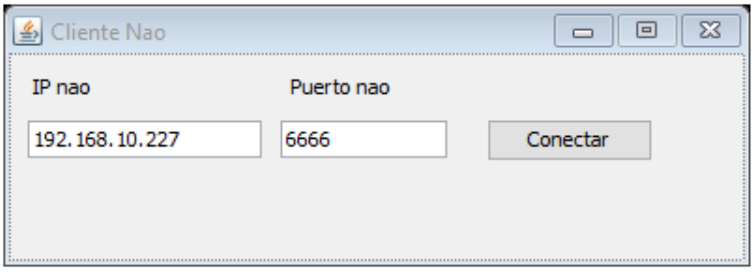
5.2 Clases

A continuación, veremos cada una de las clases del sistema centrándonos en los métodos y atributos que tienen, así como en las interfaces graficas que gestionan y sus particularidades. La interconexión y dependencias de las clases se puede ver en el siguiente diagrama de clases:



5.2.1 MainWindow

Esta clase será la entrada de nuestro cliente, gestionará la primera ventana que vera el usuario que nos permitirá solicitar conexión con el servidor.



Como esta clase gestiona una interfaz gráfica en su constructor se inicilizaran y confoguraran todos los elementos que la componen.

Contamos con los cuadros de texto para introducir la dirección IP y puerto donde se encuentra el servidor y un único botón que nos permite solicitar la conexión, en cuanto se pulse el botón conectar se ejecutara el método conectar() de la clase.

```
88      private void conectar(){
89          int puerto = 0;
90          String ip = null;
91          try {
92              puerto = Integer.parseInt(textPuerto.getText());
93              ip = textIP.getText();
94          } catch (NumberFormatException e1) {
95              mostarPanelError("El puerto tiene que ser un entero");
96              return;
97          }
98
99          if (puerto != 0 && ip != null && ClienteTCP.validateIP(ip) == true) {
100             cliente = new ClienteTCP(ip, puerto);
101             int aux = cliente.conectar();
102             if (aux != -1 && aux != -2) {
103
104                 // conexion con nao creada creamos ventanas y manager
105                 vv = new VentanaVisualizacion();
106                 vc = new VentanaControl(cliente,vv);
107
108                 manager = new Manager(vc, vv, cliente);
109                 manager.start();
110
111                 vc.setVisible(true);
112                 vv.setVisible(true);
113
114                 frmClienteNao.dispose();
115             } else {
116                 if(aux == -2){
117                     mostarPanelError("Conexion rechazada");
118                 }else {
119                     mostarPanelError("Error al conectar con NAO (no respuesta server)");
120                 }
121                 return;
122             }
123
124         } else {
125             mostarPanelError("Error al introducir la ip o el puerto");
126             return;
127         }
128     }
```

Lo primero que hará el método conectar() es realizar una comprobación del formato de la IPy puerto, si no son correctos nos devolverá un mensaje de error. Si el puerto y la IP tienen los formatos correctos se instanciará la clase ClienteTCP que nos permitirá solicitar la conexión.

Si el servidor nos acepta la conexión (en otro caso se mostrará un mensaje de error) se instanciarán las ventanas principales de la aplicación: VentanaVisualizacion y VentanaControl pasándoles a cada una los parámetros que necesite.

Una vez creadas las ventanas instanciará la clase Manager que ejecutará el hilo que estará a la espera de los mensajes recibidos por el servidor y sabrá cómo tratarlos. Con las ventanas y el manager funcionando pondrá visibles las ventanas y ocultará la que ofrece esta clase.

5.2.2 ClienteTCP

Esta clase ofrecerá una serie de métodos que facilitaran la comunicación con el servidor. En su constructor tenemos que indicarles la IP y puerto al servidor que queremos conectarnos.

```
public ClienteTCP(String _ipServer, int _puertoServer) {
    this.ipServer = _ipServer;
    this.puertoServer = _puertoServer;
}
```

Para realizar la conexión ofrece el método conectar(), creara un socket y solicitara la conexión a la IP y puerto indicadas en el constructor de la clase. Además, como el cliente nos puede rechazar la conexión realizaremos una primera lectura utilizando el método read() para comprobar que no nos rechazó la conexión.

```
public int conectar(){

    try {
        clientSocket = new Socket(ipServer, puertoServer);
        String respuesta = read();
        if(respuesta.equals("error"))
            return -2;
        return 1;
    } catch (ConnectException e) {
        return -1;
    } catch (IOException e) {
        return -1;
    }

}
```

El método read() nos permite bloquearnos hasta recibir un mensaje. El método estará leyendo del socket hasta que encuentre el carácter de final de comando ";", se llene el buffer o el servidor se desconecte o nos cancele la conexión.

```
public String read(){
    BufferedReader br = null;
    String total = "";
    try {
        br = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
        while ( total.endsWith(";") == false)
        {

            int c = br.read();
            if(c == -1) {
                System.out.println(total);
                return "error";
            }
            total += (char)c;
        }
        //System.out.println(total);
    } catch (IOException e) {
        e.printStackTrace();
        return null;
    }
    return total;
}
```

Para el envió de datos al servidor la clase ofrece el método send(), el cual toma como argumentos una cadena de texto que será enviada al servidor.

```
public void send(String msg){
    DataOutputStream outToClient;
    try {
        outToClient = new DataOutputStream(clientSocket.getOutputStream());
        outToClient.writeBytes(msg);
    } catch (IOException e) {
        e.printStackTrace();
    }

}
```

5.2.3 HiloServidor

Esta clase será un contenedor de información, guardará la información relevante de cada hilo que se encuentra en el servidor. Se sobrescribirá las funciones equals() para que nos permita compararlas entre ellas fácilmente y el método toString() para que cuando se añada esta clase a una lista (JList) se muestre de la forma que queramos.

```
3  public class HiloServidor {
4
5      private String nombre;
6      private int estado;
7      private boolean excluyente;
8
9
10     public HiloServidor(String nombre){
11         this.nombre = nombre;
12         this.excluyente = false;
13     }
14     public boolean isExcluyente() {
15         return excluyente;
16     }
17     public void setExcluyente(boolean excluyente) {
18         this.excluyente = excluyente;
19     }
20     public String getNombre() {
21         return nombre;
22     }
23
24     public int getEstado() {
25         return estado;
26     }
27     public void setEstado(int estado) {
28         this.estado = estado;
29     }
30     @Override
31     public boolean equals(Object v) {
32
33         HiloServidor aux = (HiloServidor) v;
34
35         if(this.nombre.equals(aux.getNombre()))
36             return true;
37
38         return false;
39     }
40
41     @Override
42     public String toString() {
43         if(excluyente){
44             return " + "+nombre;
45         } else {
46             return "   "+nombre;
47         }
48     }
49 }
```

5.2.4 Manager

Esta clase encapsulará el hilo que estará a la espera de los mensajes que envía el servidor.

```
private VentanaControl vControl;
private VentanaVisualizacion vVisualizacion;
private ClienteTCP cliente;
private VentanaEscenario vEscenario;

private Vector<HiloServidor> hilosNao = new Vector<HiloServidor>();
private boolean parar;

public Manager(VentanaControl vControl, VentanaVisualizacion vVisualizacion, ClienteTCP tcp) {
    this.vControl = vControl;
    this.vVisualizacion = vVisualizacion;
    this.cliente = tcp;
    this.parar = true;
    vEscenario = new VentanaEscenario();
}
```

Necesitará que se le pasen las clases VentanaControl, VentanaVisualizacion y ClienteTCP, para que pueda quedarse a la espera de mensajes y proveer a las ventanas de la información relevante que llegue del servidor.

Esta clase tendrá un atributo de tipo Vector de HiloServidor, en el que guardará los hilos en el servidor y su información cada vez que los mande el servidor.

Se sobrescribirá el método run() el cual será el que ejecute el hilo.

```
30      @Override
31      public void run() {
32          String comando;
33
34          while(parar){
35              comando = cliente.read();
36              System.out.println(comando);
37              if(comando == null)
38                  return;
39              vControl.addStringRecivido(comando);
40              comando = comando.substring(0, comando.length()-1);
41              String[] parts1 = comando.split("#");
42              if(parts1.length != 2)
43                  continue;
44
45              if (parts1[0].compareTo("PER") == 0) {
46                  String[] parts = parts1[1].split("/");
47
48                  for (int i = 0; i < parts.length; i++) {
49                      String[] partes = parts[i].split("~");
50                      if(partes.length != 2)
51                          continue;
52                      if (partes[0].compareTo("GLUCOSA") == 0) {
53                          vVisualizacion.addDotGraf(Float.parseFloat(partes[1]));
54                      }
55
56                      if (partes[0].compareTo("ESTADOHILOS") == 0) {
57                              procesarHilosServidor(partes[1]);
58                      }
59                      if (partes[0].compareTo("ESCENARIO") == 0) {
60                          vEscenario.addDatos(partes[1]);
61                      }
62                  }
63
64              }
65
66              if(parts1[0].compareTo("RES") == 0){
67                  vControl.addTextPane(parts1[1]);
68              }
69
70
71          }
72
73      }
```

El hilo estará en un bucle infinito esperando y procesando los mensajes que lleguen. Pueden llegar dos tipos de mensajes, los periódicos con prefijo “PER” y las respuestas con prefijo “RES”. Dependiendo el prefijo tendrá que realizar unas acciones u otras, pero en general decodificara la información que envía el servidor y la proveerá a las ventanas que la necesiten.

Como dentro de los datos periódicos viene información de forma dinámica, tendrá que mirar uno por uno los datos que han llegado en el mensaje y tratarlos dependiendo de su ID.

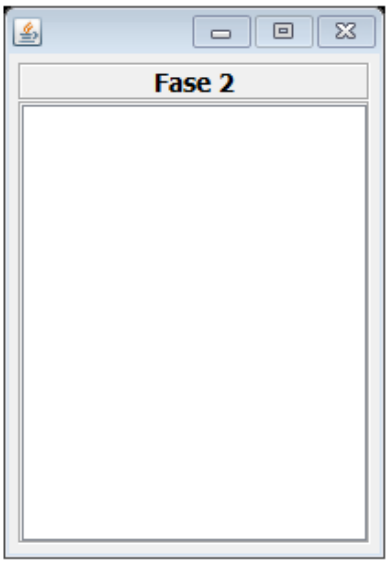
El servidor siempre nos mandara en los datos periódicos el estado de los hilos, por lo que se tiene la función procesarHilosServidor() que nos ayuda a tratarlos.

```
75      private void procesarHilosServidor(String hilos){
76          boolean update = false;
77
78          //separamos hilos excluyentes de los no excluyentes
79          String[] hilosTipo = hilos.split("%");
80
81
82          String[] parts = hilosTipo[0].split(",");
83          for (int i = 0; i< parts.length; i++) {
84              String[] split = parts[i].split(":");
85              HiloServidor aux = new HiloServidor(split[0]);
86              aux.setEstado(Integer.parseInt(split[1]));
87
88              //mirar si es hilo nuevo o no, si ya existe actualizar estado si es necesario
89              if(hilosNao.contains(aux)){
90                  if( hilosNao.get(hilosNao.indexOf(aux)).getEstado() != aux.getEstado() ){
91                      hilosNao.get(hilosNao.indexOf(aux)).setEstado(aux.getEstado());
92                      update = true;
93                  }
94              } else {
95                  hilosNao.add(aux);
96                  update = true;
97              }
98          }
99
100         if(hilosTipo.length == 2){
101             String[] parts1 = hilosTipo[1].split(",");
102             for (int i = 0; i< parts1.length; i++) {
103                 String[] split = parts1[i].split(":");
104                 HiloServidor aux1 = new HiloServidor(split[0]);
105                 aux1.setEstado(Integer.parseInt(split[1]));
106                 aux1.setExcluyente(true);
107                 //mirar si es hilo nuevo o no, si ya existe actualizar estado si es necesario
108                 if(hilosNao.contains(aux1)){
109                     if( hilosNao.get(hilosNao.indexOf(aux1)).getEstado() != aux1.getEstado() ){
110                         hilosNao.get(hilosNao.indexOf(aux1)).setEstado(aux1.getEstado());
111                         update = true;
112                     }
113                 } else {
114                     hilosNao.add(aux1);
115                     update = true;
116                 }
117             }
118         }
119
120         if(update == true)
121             vControl.addEstadoHilos(hilosNao);
122     }
```

Este método mirará uno a uno los hilos dentro del servidor, los añadirá a nuestro Vector de hilos en caso de que no estén o actualizara su estado en caso de que ya estén. Si se han añadido hilos nuevos o actualizado el estado de alguno actualizara la información de la VentanaControl.

5.2.5 VentanaEscenario

Esta clase encapsulara el control de una pequeña ventana que se mostrara siempre que este activa la fase dos en el servidor y nos llegue a los datos periódicos el dato con ID “ESCENARIO”.



En su constructor se instanciará y configuraran todos los elementos que componen la interfaz gráfica de usuario, que en este caso son solo dos: un JLabel y un JTextArea.

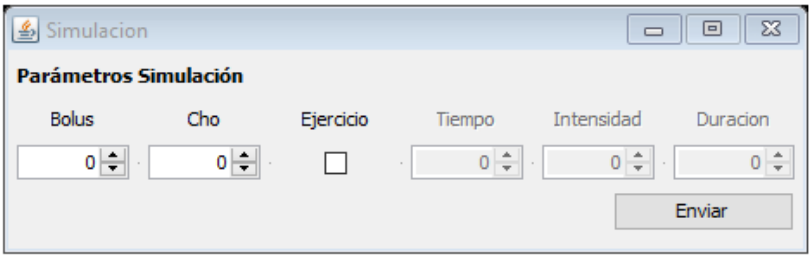
La clase ofrecerá el método addDatos(), al cual le pasara el Manager los datos que vengan con la ID “ESCENARIO”.

```
74⊖    public void addDatos(String datos){
75         String[] parts = datos.split(",");
76         textArea.setText("");
77         if(Integer.parseInt(parts[0]) == 2 && parts.length == 5){
78             if(getEscenarioFlag() == false){
79                 setVisible(true);
80                 setEscenarioFLag(true);
81             }
82             lblFase.setText("Fase 2");
83             textArea.append("EstadoTaller: "+parts[1]+"\n");
84             textArea.append("NumHambre: "+parts[2]+"\n");
85             textArea.append("NumEjer: "+parts[3]+"\n");
86             textArea.append("Ultima palabra: "+parts[4]+"\n");
87         } else {
88             auxEscenario = false;
89             setVisible(false);
90             dispose();
91         }
92     }
```

El método los mostrara en formato texto dentro del JtextArea y mostrara la ventana al usuario.

5.2.6 VentanaSimulacion

Esta clase ofrecerá una ventana que permitirá seleccionar de forma cómoda valores particulares para solicitar una petición de simulación al servidor.



La clase necesitara que se le pase en el constructor a la clase ClienteTCP para que pueda realizar el envió de comandos al servidor, además también se le pasara el JTextArea de la ventanaControl para que informe de la petición del comando.

```
47    public VentanaSimulacion(ClienteTCP _tcp,JTextArea textPane) {
48        this.tcp = _tcp;
49        this.textControl =textPane;
```

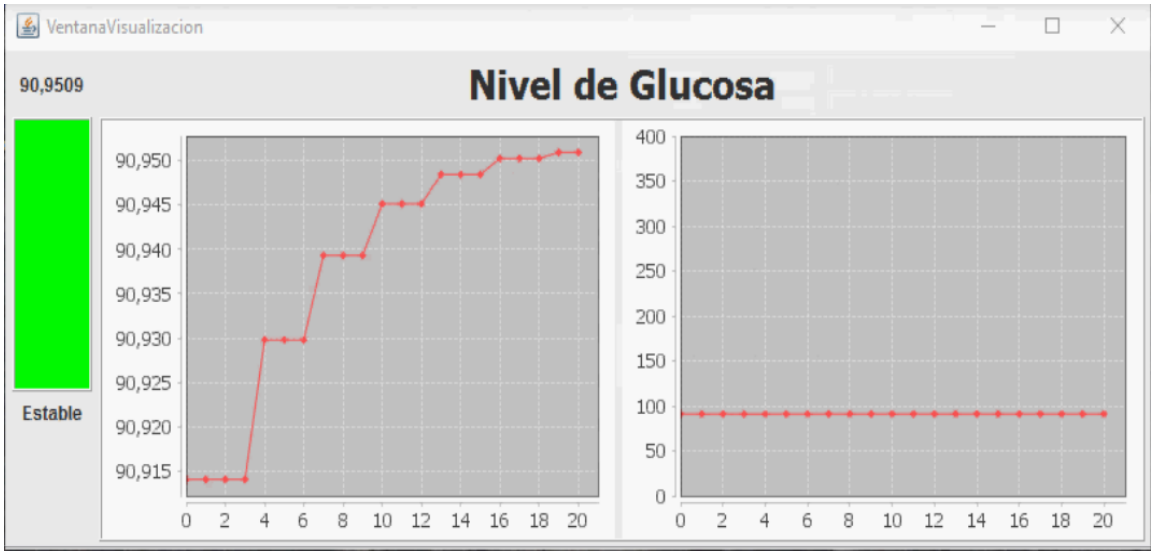
En el constructor se configurarán todos los elementos que componen esta interfaz y se añadirá el código para que cuando se pulse el botón “Enviar” se mande el comando al servidor.

```
botonEnviar = new JButton("       Enviar       ");
botonEnviar.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {

        String aux = "NAO#SIMULAR:";

        double bolus = (double) spinnerBolus.getValue();
        aux = aux + Double.toString(bolus) + ",";
        double cho = (double) spinnerCho.getValue();
        aux = aux + Double.toString(cho) + ",";
        boolean ejer = chckEjercicio.isSelected();
        if(ejer){
            aux = aux + "true" + ",";
            double ejerT = (double) spinnerTiempo.getValue();
            double ejerD = (double) spinnerDuracion.getValue();
            double ejerI = (double) spinnerIntensidad.getValue();
            aux = aux + Double.toString(ejerT) + ",";
            aux = aux + Double.toString(ejerI) + ",";
            aux = aux + Double.toString(ejerD) + ";";
        } else {
            aux =aux+ "false,0,0,0;";
        }
        textControl.append(aux +"\n");
        tcp.send(aux);
    }
});
```

Como vemos al pulsar en el botón, se obtendrán todos los valores dentro de los spinners y con ellos se formará el comando que será de tipo “SIMULAR”, se añadirá el comando al área de texto de la ventanaControl y se enviará al servidor.

5.2.7 VentanaVisualizacion

Esta clase encapsulara en comportamiento de la interfaz gráfica de usuario destinada a mostrar los valores de glucosa de maneras distintas. Sera una de las dos ventanas principales del programa.



En esta ventana se utilizarán elementos de la librería JFreeChart, más en concretos se utilizarán las clases XYSeries, XYSeriesCollection, JFreeChart y ChartPanel. Que nos permitirán la creación de dos gráficos de puntos que se irán actualizando con los valores que se encuentre dentro de nuestra serie XY (clase XYSeries) y que podremos actualizar de forma dinámica.

```
33 public class VentanaVisualizacion extends JFrame {
34
35     private static final long serialVersionUID = 1L;
36     //---------Paneles---------
37     private JPanel panelPrincipal;
38     private JPanel panelCentral;
39
40     private XYSeries datosGLucosa;
41     private  XYSeriesCollection data;
42     private JFreeChart chart1;
43     private ChartPanel chartPanel;
44     private JLabel lbl1;
45
46     private int i = 0;
47     private JLabel lblNivelDeGlucosa;
48     private JLabel labelGlucosa;
49     private JPanel panel_1;
50
51     private ChartPanel chartPanel_1;
52     private JFreeChart chart1_1;
53     private JLabel lblGluText;
54
```

En el constructor de la clase se realizan todas las configuración e instanciaciones de los elementos gráficos que componen la interfaz.

Para actualizar los elementos gráficos la clase ofrecerá una serie de métodos, que serán llamados por el Manager cada vez que se obtenga del servidor un nuevo valor de glucosa.

El método principal para actualizar la interfaz es addDotGraft(), este método recibirá como paramento la glucosa que envía el servidor, mirara su valor y actualizara la vista.

```
public void addDotGraf(float glucosa){
    try{
        if(glucosa != -999 && glucosa != -1000) {
            this.datosGLucosa.add(i, glucosa);
            i = i+1;
            setEstadoLabelGlucosa(glucosa);
        }
        if( glucosa == -1000)
            resetChart();
    } catch (Exception e) {
    }
}
```

Siempre que el valor de glucosa que llegue y tenga un valor disiento a -999 y -1000 el valor será válido y se podrá añadir al conjunto de datos (como es un conjunto XY necesita dos valores, la x será la posición en la que llego el dato la cual se actualiza con la llegada de cada uno) a la vez que se actualiza el resto de los elementos mediante el método setEstadoLabelGlucosa().

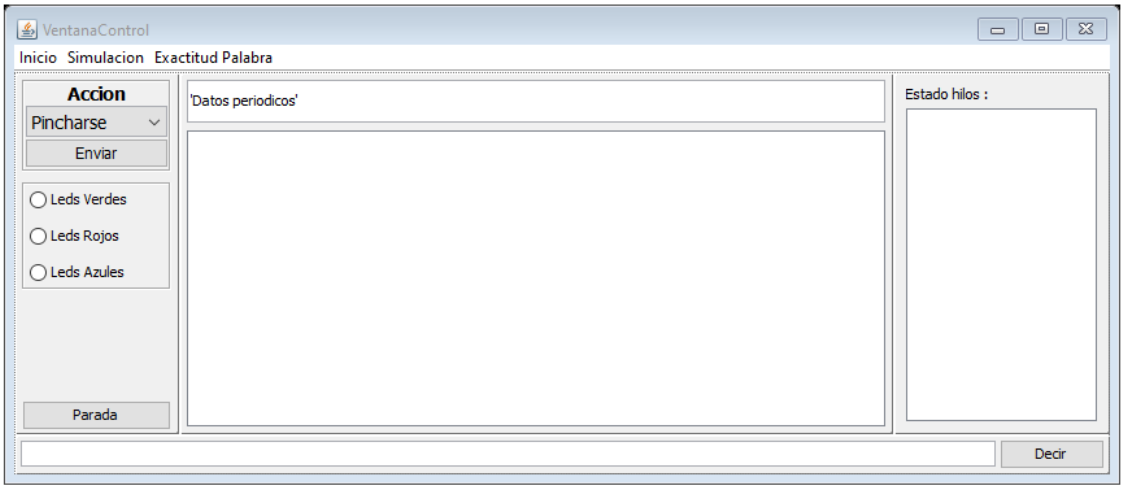
Si el valor que llega es -999 y -1000 es porque el simulador está parado (-1000) en cuyo caso se resetearan los gráficos o esta pausado (-999) en cuyo caso no se actualizara el grafico y se dejara como esta.

El método setEstadoLabelGlucosa() actualizara los Jlabel que muestran el valor de glucosa en formato texto y formato color.

```
public void setEstadoLabelGlucosa(double glucosa){

    lbl1.setBackground(new Color(0, 255, 0, 255));

    if(glucosa > 175){
        lbl1.setBackground(new Color(255, 0, 0, 255));
        lblGluText.setText("Alta");
    }
    if( glucosa > 75 && glucosa < 175){
        lbl1.setBackground(new Color(0, 255, 0, 255));
        lblGluText.setText("Estable");
    }
    if(glucosa < 75){
        lbl1.setBackground(new Color(0, 0, 255, 255));
        lblGluText.setText("Baja");
    }

    DecimalFormat df = new DecimalFormat("#.0000");
    String angleFormated = df.format(glucosa);
    labelGlucosa.setText(angleFormated);
}
```

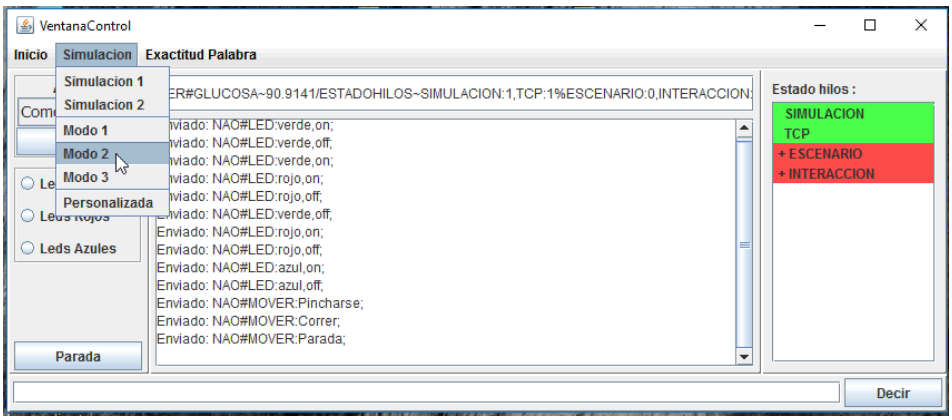
5.2.8 VentanaControl

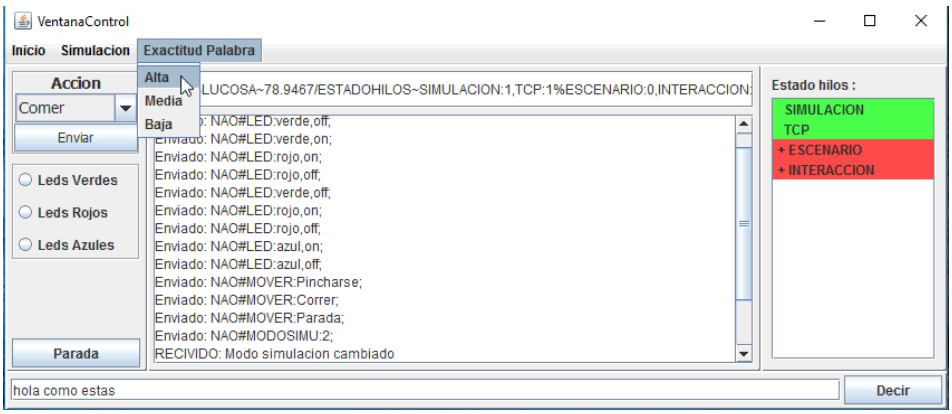
Esta clase encapsulara en comportamiento de la interfaz gráfica de usuario destinada a ofrecer controles para manejar el estado del robot durante una sesión de uso. Sera una de las dos ventanas principales del programa.



La clase necesitara tener acceso a la clase ClienteTCP para poder realizar los envíos de comandos al servidor. En el constructor se inicializarán y configuran todos los elementos que componen la interfaz.

La clase tendrá una barra de menú con diferentes botones.





Los cuales tendran implementado en su condigo el envio de los diferentes comandos, de tipo “SIMULAR” con simulaciones predefinidas, de tipo “MODOSIMU” para cambiar el modo de la simulacion o de tipo “EXACPALABRA” para cambiar el umbral para que NAO acepte una palabra.

```
mntmSimulacion_1 = new JMenuItem("Simulacion 1");
mntmSimulacion_1.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent arg0) {
        String aux = "NAO#SIMULAR:0,0,true,0,50,60;";
        tcp.send(aux);
        textPane.append("Enviado: "+ aux +"\n");
    }
});
mnSimulacion.add(mntmSimulacion_1);


  menuItemModo1 = new JMenuItem("Modo 1");
  menuItemModo1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent arg0) {
          String aux = "NAO#MODOSIMU:1;";
          tcp.send(aux);
          tcp.send("NAO#THREAD:SIMULACION,4;");
          textPane.append("Enviado: "+ aux +"\n");
      }
  });
  mnSimulacion.add(menuItemModo1);


   mntmAlta = new JMenuItem("Alta");
   mntmAlta.addActionListener(new ActionListener() {
       public void actionPerformed(ActionEvent arg0) {
           String aux = "NAO#EXACPALABRA:0.7;";
           tcp.send(aux);
           textPane.append("Enviado: "+ aux +"\n");
       }
   });
   menuExat.add(mntmAlta);
```

Ademas tendra un boton para abrir la VentanaSimulacion y poder enviar comandos de tipo “SIMULAR” con valores personalizados.

```
mntmNewMenuItem_1 = new JMenuItem("Personalizada");
mntmNewMenuItem_1.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent arg0) {
        vSimulacion.setVisible(true);
    }
});
mnSimulacion.add(mntmNewMenuItem_1);
```

Tendrá una botonera a la izquierda la cual ofrecerá botones para enviar comandos de tipo “MOVER” y de tipo “LED”.

Al pulsar el botón “Enviar” de este panel, se enviará un comando de tipo “MOVER” con la acción que se tenga seleccionada en el JComboBox.

```
btnNewButton_4.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent arg0) {
        String ac = (String) comboBox.getSelectedItem();
        String aux = "NAO#MOVER:"+ac+";";
        tcp.send(aux);
        textPane.append("Enviado: "+ aux +"\n");
    }
});
```

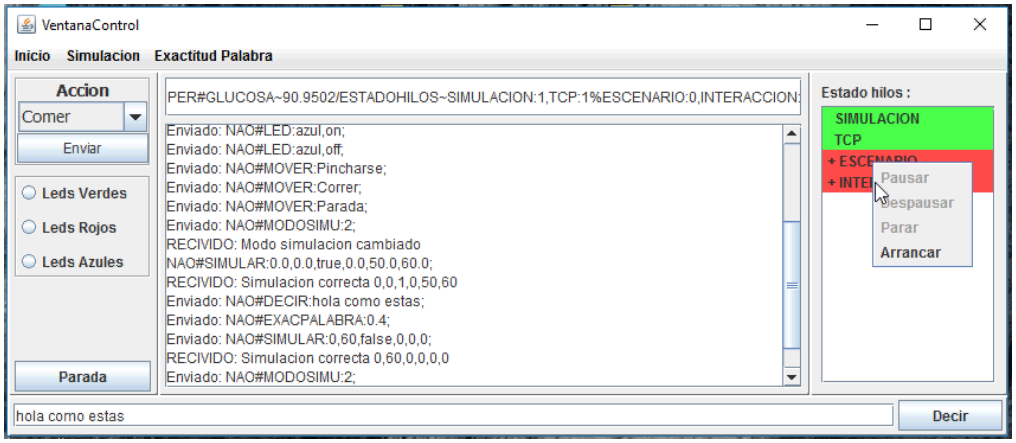
Cuando se pulse en uno de los Checkbox de los leds, se enviará un comando de tipo “LED”, indicando si apagar o no los leds del color seleccionado (dependiendo de si se pulsa o despulsa).

```
rdbtnLedsRojos.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent arg0) {
        String aux;
        if (rdbtnLedsRojos.isSelected()) {
            aux = "NAO#LED:rojo,on;";
        } else {
            aux = "NAO#LED:rojo,off;";
        }
        tcp.send(aux);
        textPane.append("Enviado: "+ aux +"\n");
    }
});
```

Y tendra un boton exclusivo para mandar al robot a una posicion de seguridad “Parada”.

```
btnParada.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        String aux = "NAO#MOVER:Parada;";
        tcp.send(aux);
        textPane.append("Enviado: "+ aux +"\n");
    }
});
```

En la parte derecha podemos ver un JList el cual nos mostrara los hilos en el servidor y se tendrá habilitado el clic derecho (al ser pulsado desplegará un menu que nos permitirá enviar comandos de tipo “THREAD”).



Cada vez que se pulse clic derecho encima del nombre de algún hilo, se seleccionara ese hilo y se mostrara un menú desplegable que nos dejara enviar los comandos asociados al control de hilos.

```
list.addMouseListener(new MouseAdapter() {
    @Override
    public void mouseClicked(MouseEvent arg0) {
        int aux = list.locationToIndex(new Point(arg0.getX(), arg0.getY()));
        if(aux == -1)
            return;
        list.setSelectedIndex(aux);
        index = list.getSelectedIndex();
        HiloServidor hilo = list.getSelectedValue();
        if(hilo.getEstado() == 1){ //corriendo
            pausar.setEnabled(true);
            despausar.setEnabled(false);
            parar.setEnabled(true);
            arrancar.setEnabled(false);
        }
        if(hilo.getEstado() == 0){ //parado
            pausar.setEnabled(false);
            despausar.setEnabled(false);
            parar.setEnabled(false);
            arrancar.setEnabled(true);
        }
        if(hilo.getEstado() == 2){ //pausado
            pausar.setEnabled(false);
            despausar.setEnabled(true);
            parar.setEnabled(true);
            arrancar.setEnabled(false);
        }
        if(SwingUtilities.isRightMouseButton(arg0) && index == list.locationToIndex(arg0.getPoint())){
            if(!list.isSelectionEmpty()){
                pop.show(list, arg0.getX(), arg0.getY());
            }
        }
    }
});
```

En el menú desplegable tendremos cuatro botones: pausar, despausar, parar y arrancar. Una vez pulsemos en uno de ellos se mandará un comando de tipo “THREAD” con el nombre del hilo seleccionado y la acción que indica el botón.

```
588     private void addPopup(){
589         pop.add(pausar);
590         pop.add(despausar);
591         pop.add(parar);
592         pop.add(arrancar);
593
594         pausar.addActionListener(new ActionListener() {
595             @Override
596             public void actionPerformed(ActionEvent e) {
597                 HiloServidor hilo =list.getSelectedValue();
598                 if(hilo!= null ){
599                     String aux = "NAO#THREAD:"+hilo.getNombre()+",2;";
600                     tcp.send(aux);
601                 }
602             }
603         });
604
605         despausar.addActionListener(new ActionListener() {
606             @Override
607             public void actionPerformed(ActionEvent e) {
608                 HiloServidor hilo =list.getSelectedValue();
609                 if(hilo!= null ){
610                     String aux = "NAO#THREAD:"+hilo.getNombre()+",3;";
611                     tcp.send(aux);
612                 }
613             }
614         });
615
```

```
616         parar.addActionListener(new ActionListener() {
617             @Override
618             public void actionPerformed(ActionEvent e) {
619                 HiloServidor hilo =list.getSelectedValue();
620                 if(hilo!= null ){
621                 String aux = "NAO#THREAD:"+hilo.getNombre()+",0;";
622                 tcp.send(aux);
623                 }
624             }
625         });
626 
627         arrancar.addActionListener(new ActionListener() {
628             @Override
629             public void actionPerformed(ActionEvent e) {
630                 HiloServidor hilo =list.getSelectedValue();
631                 if(hilo!= null ){
632                 String aux = "NAO#THREAD:"+hilo.getNombre()+",1;";
633                 tcp.send(aux);
634                 }
635             }
636         });
637     }
```

Como los hilos dentro del servidor cambian su estado de forma dinámica esta lista se tendrá que ir actualizando, para ello se tiene el método addEstadoHilos().

```
public void addEstadoHilos(Vector<HiloServidor> estadoHilos) {
    modeloHilosNao.clear();
    for(int i=0; i< estadoHilos.size(); i++){
        modeloHilosNao.addElement(estadoHilos.get(i));
    }
}
```

En la parte inferior tenemos el boton “Decir”, el cual cogera el texto de JTextArea que tiene al lado y lo mandara al NAO para que lo diga.

```
buttonDecir = new JButton("    Decir    ");
buttonDecir.setAlignmentX(Component.CENTER_ALIGNMENT);
buttonDecir.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent arg0) {
        String aux = "NAO#DECIR:"+textoDecir.getText()+";";
        tcp.send(aux);
        textPane.append("Enviado: "+ aux +"\n");
    }
});
```