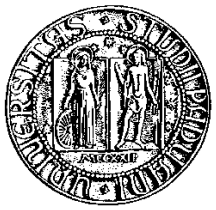

Università degli Studi di Padova

Facoltà di Ingegneria

Corso di Laurea in Ingeneria Informatica

tesi di laurea

Una guida per l'utilizzazione educativa del robot Mindstorms NXT con programmazione Java: la componente software

Relatore: Michele Moro

Laureando: Marco Rivello

ver.1.0 – 20 aprile 2010

Autore: Marco Rivello

Indice

Sommario

Nasce dalla collaborazione con il collegio vescovile Pio X l'esigenza di creare del materiale di supporto al corso di robotica sostenuto dai ragazzi delle superiori. Lo scopo di questa tesi è quella di realizzare della documentazione che possa esser d'aiuto all'approccio al robot Mindstorms NXT in Java da parte di tutti gli studenti delle superiori. Attualmente il robot e il firmware originario vengono sfruttati utilizzando il linguaggio nativo grafico NXT-G, che permette la programmazione del brick attraverso la combinazione di una sequenza di azioni atomiche già predefinite. Ovviamente l'utilizzo di questo linguaggio limita molto i risultati ottenibili dal robot e prorpio questo svantaggio ha fatto sorgere l'esigenza di ricercare qualcosa di più adatto per poter sfruttare appieno le potenzialità della macchina. La risposta a questo problema è l'utilizzo non più del firmware nativo con cui viene venduto NXT, ma Lejos, un sistema open source che permette la programmazione utilizzando il linguaggio Java. Ovviamente la complessità aumenta parecchio perchè è richiesta una buona conoscenza di Java, ma obiettivo di questo progetto è anche raggruppare tutti gli strumenti necessari che permettano il rapido apprendimento di questo linguaggio. La metodologia che viene utilizzata è quella di realizzare degli esperimenti di laboratorio, ripetibili dagli studenti in un secondo momento, per mostrare come utilizzare pienamente gli strumenti messi a disposizione da Lejos. Un ampio capitolo sarà dedicato allo sviluppo di software multi-threadig, argomento che solitamente viene trattato all'università, ma indispensabile nella programmazione dei robot. La complessità di questi concetti, come detto prima verrà affrontata mostrando degli esempi realizzati in laboratorio, ma soprattutto verranno affrontati e documentati in modo che gli studenti possano ricrearli durante le loro lezioni di robotica ed apprendere cosi i concetti basilari del multi-threading.

Capitolo 1

NXT Firmware Lejos

Lejos NXJ è un progetto open source sviluppato da José Solórzano ed implementa una Java Virtual Machine specifica per il brick NXT. É l'acronimo di Lego Java Operating Systems e va a sostituire il firmware originale dell'NXT. Una JVM non è altro che un software che permette al codice compilato, chiamato byte code, di essere interpretato ed eseguito sul sistema operativo sulla quale viene eseguita. Il punto di forza di Java è l'indipendenza del linguaggio dalla piattaforma utilizzata, infatti esisto molte versioni di JVM per diversi sistemi operativi. Lejos mette a disposizione una JVM scritta in C e che realizza il concetto di indipendenza, infatti è presente su altri due dispositivi: Lego Brick NXT e il Nintendo Gameboy Advance.

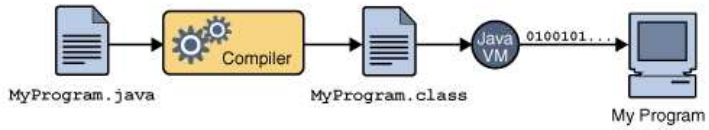

Figura 1.1: Dal sorgente all'eseguibile per la JVM

Oltre alla JVM, Lejos include anche molto altro:

- una libreria di classi che vanno a definire alcune applicazione per l'NXJ;
- gli strumenti necessari per la programmazione del firmware, il trasferimento di software e il debugging;
- le API necessarie per la programmazione lato PC, quindi tutti i modi di connessione al brick NXJ, via USB o Bluetooth oppure tramite il protocollo LCP (Lego Communications Protocol);
- le API complete di tutte le classi utilizzabili sul NXJ.

L'arrivo di questo nuovo sistema ha introdotto innumerevoli vantaggi rispetto alla semplice programmazione logo like NXT-G.

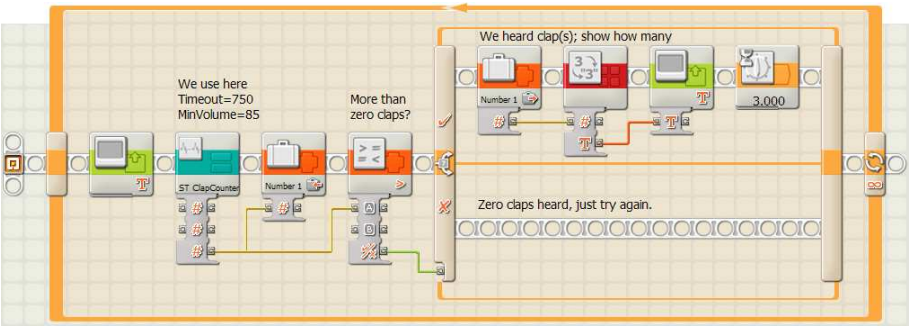

Figura 1.2: Programmazione NXT-G

L'utilizzo di java, come linguaggio di programmazione, permette di sfruttare la programmazione orientata agli oggetti, mettendo a disposizione ambienti di sviluppo diffusissimi come Eclipse e Netbeans.

Supporta il cross-platform, funziona sia su macchine Windows che Linux e Mac OS X. Nel capitolo 3 verrà spiegato passo passo come installarlo e configurare Windows e Mac OS X per riuscire a sviluppare software su questi due sistemi.

L'utilizzo di questo nuovo firmware rende molto più veloce la macchina e supporta completamente connessioni bluetooth, usb, I2C e protocolli RS485 riuscendo cosi a interfacciare il brick con il PC in entrambe le direzioni. Con l'ausilio di questi strumenti è possibile, sfruttando le API a disposizione, scrivere applicazioni di connessione tra altri dispositivi (PC, modulo GPS, ecc) e il modulo NXJ. Nelle nostre esperienze di laboratorio questo ritornerà molto utile per il monitoraggio dei vari sensori e per lo scambio di informazioni con il PC.

Implementa le più recenti funzionalità del linguaggio Java 1.6 con un'accurata gestione del motore tramite l'algoritmo di controllo PID, inoltre sfrutta degli algoritmi probabilistici per la gestione del robot, come il filtro di Kalman, noto per il monitoraggio di sistemi dinamici soggetti ad errori di misura. E' presente nelle API di Lejos un'apposita classe, 'KalmanFilter' che permette di implementare un filtro di Kalman, presente nel package lejos.util.*;

Molto importante è il supporto della programmazione multithreading riuscendo a far eseguire al robot più comandi contemporaneamente. Ad esempio si può pensare ad un thread che gestisce il movimento, ad uno che sfrutta il GPS ed un altro che usa il sensore ultrasuoni, mandati in esecuzione tutti insieme e gestiti dallo scheduler del brick, per ottenere un dato comportamento. La presenza dell'interfaccia Behaviour evita il problema dello spaghetti code, ovvero un groviglio mal definito di funzioni con scopi diversi. Grazie a questa astrazione è possibile definire comportamenti specifici, rendendo il codice molto più pulito ed efficiente.

Una delle principali migliorie, introdotta già dalla versione 0.5 è la presenza del garbage collector, che rende la programmazione più semplice, non dovendosi preoccupare degli oggetti generati e non più utilizzati. Infatti il compito principale del garbage collector è quello di eliminare gli oggetti non più referenziati, andando a lib-

erare memoria. Gestisce molti sensori di terzi, tra i quali quelli prodotti dalla Mindsensors e dalla HiTechnics. Permette l'utilizzo di operazioni matematiche in virgola mobile, di funzioni trigonometriche e di molte altre, permettendoci di elaborare dati per esperienze di laboratorio ad ampio spettro come quelle di fisica.

Capitolo 2

Installazione e utility LeJos

2.1 Installazione LeJos NXJ

- **Passo 1:** Prerequisito fondamentale, prima di iniziare a parlare dell'installazione di java, sono i driver USB. Se si è già installato sul Pc il software Lego Mindstorms, allora essi sono già presenti, altrimenti bisogna scaricarli dal sito Mindstorms ed installarli. Una volta completata l'installazione bisogna assicurarsi che il dispositivo venga riconosciuto correttamente e questo si può verificare andando sul pannello di controllo e controllando che sia presente il dispositivo Lego.

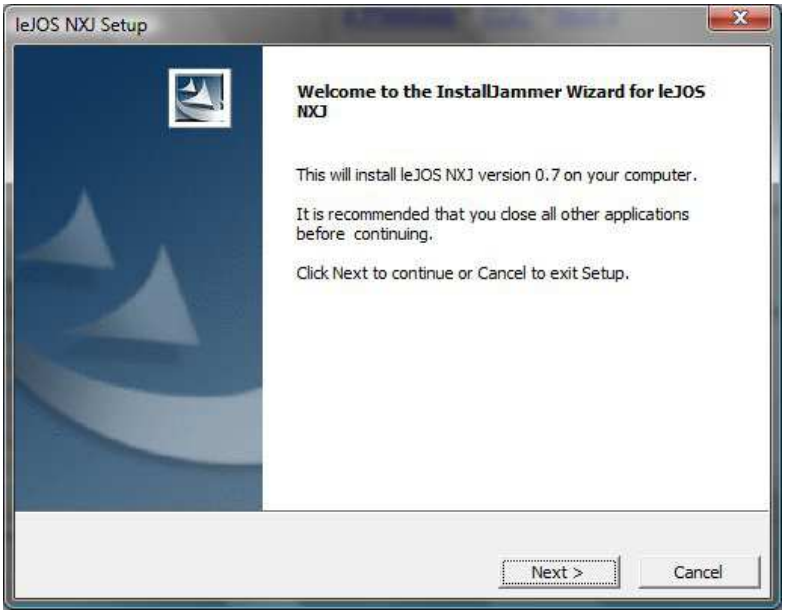

Figura 2.1: Installazione Lejos

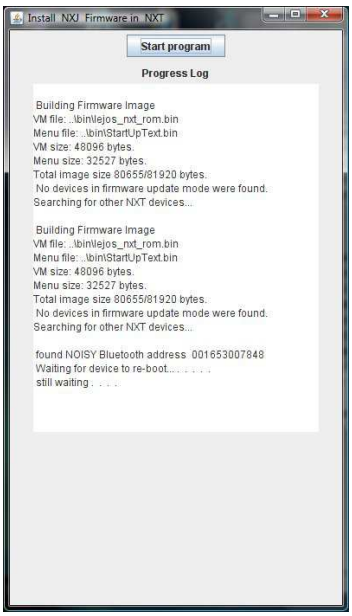

Figura 2.2: Aggiornamento Firmware

- **Passo 2:** Ora per poter scrivere e compilare i nostri programmi avremmo bisogno del Java Development Kit (JDK), il JRE non è sufficiente per far questo. Basta scaricarlo da http://java.sun.com e seguire la procedure d'installazione. LeJos NXJ è stato testato con la versione JDK 1.5 e 1.6, ma non dovrebbe dare alcun problema anche con versioni successive. Adesso avremo a disposizione i due comandi principali che ci interessano, javac per compilare e java per eseguire l'applicazione. Per poterli usare da qualsiasi cartella di lavoro è importante impostare il PATH di sistema con il percorso dove si è installato il JDK, specificando la cartella bin.

- **Passo 3:** Scaricare Lejos NXJ http://lejos.sourceforge.net/nxj-downloads.php dal sito di riferimento e seguire il wizard (vedi figura 2.1). Al termine della procedura d'installazione si aprirà una schermata che permetterà di aggiornare il firmware all'interno dell'NXT. Per eseguire l'aggiornamento si deve collegare l'NXT al Pc e accenderlo. Una volta riconosciuto sarà possibile procedere con l'update(vedi figura 2.2). Al termine dell'upgrade il software chiederà se si vuole eseguire la stessa operazione su un altro NXT, ovviamente se non si intende aggiornarne altri si può cliccare su No e uscire dalla procedura.

- **Passo 4:** Una volta completati correttamente i passi precedenti per poter raggiungere il nostro scopo è fondamentale impostare le variabili d'ambiente nel seguente modo.

NXJ_HOME	Percorso directory LeJos NXJ
JAVA_HOME	Percorso dove si ha installoto il JDK
PATH	Aggiungere la directory bin del JDK e di LeJos

Se la variabile NXJ_HOME e JAVA_HOME non esistono basta crearle come variabili di sistema.

- **Passo 5:** Per verificare la corretta installazione del software si può aprire il prompt dei comandi (Start->Esegui cmd) e digitare il comando set. In questo modo si otterrà una lista di variabili, nella quale si potrà verificare la presenza di quanto fatto al passo 4. Per impostare le variabili si deve andare sul Pannello di controllo, sistema e aprire la scheda avanzate. Qui si trova la sezione dedicata alle variabili d'ambiente(vedi figura 2.3).

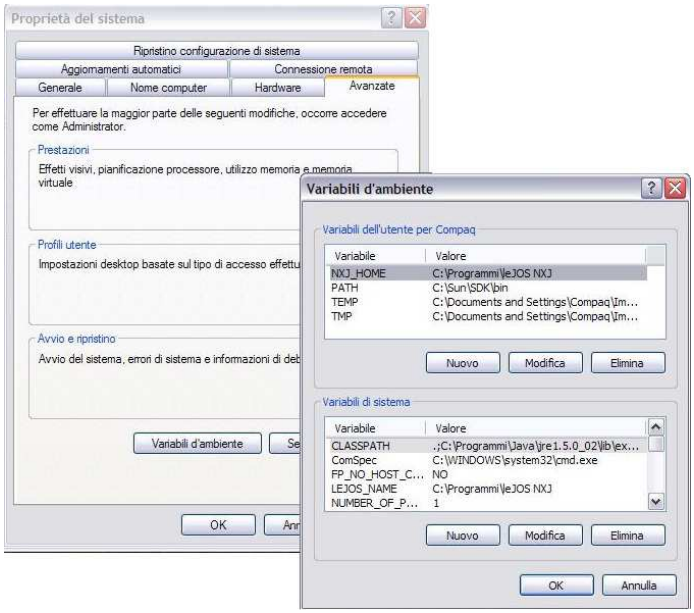

Figura 2.3: Variabili d'ambiente

2.2 Installazione e configurazione di eclipse

Per sviluppare software in ambiente IDE si consiglia l'utilizzo di eclipse scaricabile da http://www.eclipse.org/downloads/. L'unica cosa da selezionare durante l'installazione è il proprio workspace (vedi figura 2.4); si consiglia di indicare un percorso che non contenga spazi, questo per evitare problemi con il path. Terminata l'installazione si può avviare eclipse e creare un nuovo progetto java semplicemente con queste istruzioni: File => New => Java Project. L'importante è che il nome del nuovo progetto sia senza spazi. Adesso dobbiamo far in modo che il nostro progetto diventi un LeJos project e per far questo basta cliccare con il tasto destro sopra il progetto e

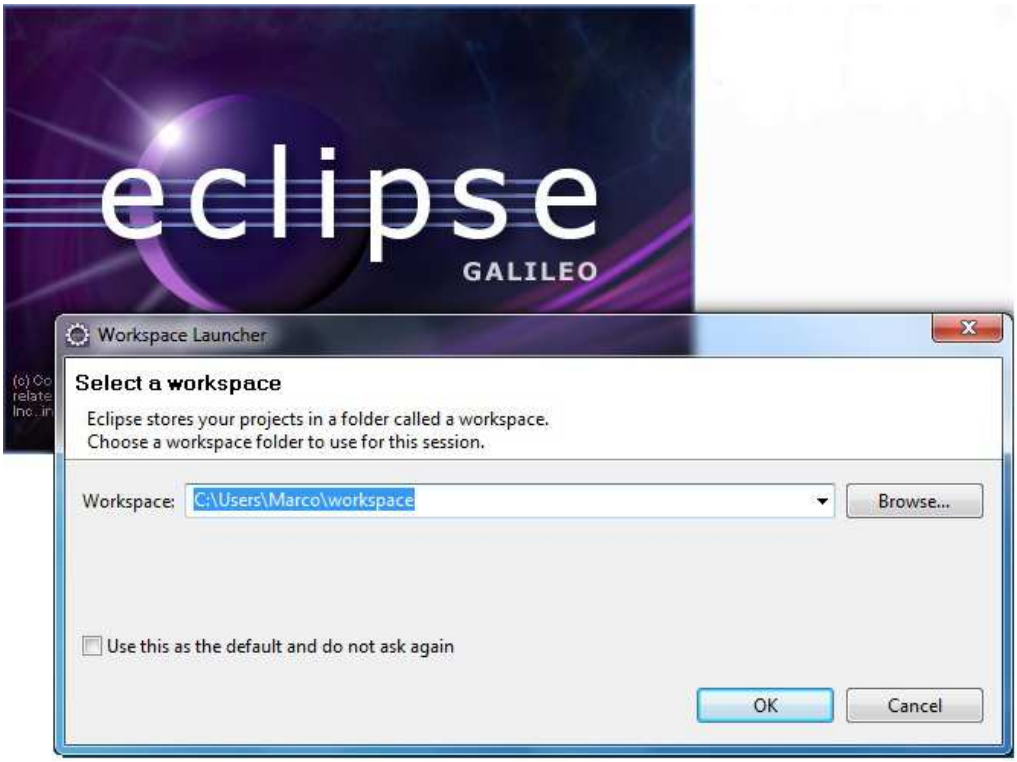

Figura 2.4: Workspace

selezionare Properties, selezionare Java Build Path sulla sinistra e cliccare su Libraries. Fatto questo bisogna cliccare su Add External Jars.., andare ad aprire la directory lib che si trova dove si ha installato lejos_nxj e selezionare il file classes.jar, aprendolo (vedi figura 2.5). Eseguite queste semplici istruzioni ora bisogna prepare il compilatore. Restando nelle proprietà del progetto ci si deve spostare su Java Compiler, spuntare Enable project specific settings e scegliere il livello 1.3, applicando le modifiche apportate (vedi figura 2.6). Ora si deve preparare Eclipse a trasferire il codice compilato all'interno dell'NXT, cliccando su Run => External Tools => Open External Tools Dialog... selezionandolo sulla sinistra, seguito da un click sull'icona New (vedi figura 2.7). Nominare lo strumento di download leJOS download, nel Main inserire il percorso di lejosdl.bat, che si troverà nella directory di installazione di lejos. Infine nello spazio per il Working Directory va inserito "${ project_loc }bin", mentre nella sezione Arguments va inserito "${java_type_name}". Come ultima configurazione di Eclipse va creato uno shortcut per scaricare velocemente i programmi all'interno del brick NXJ. Per far questo selezionare Organize Favorites... dall'icona verde Run e aggiungere leJOS download creato precedentemente. Ora siamo pronti per creare e trasferire del software all'interno dell'NXJ.

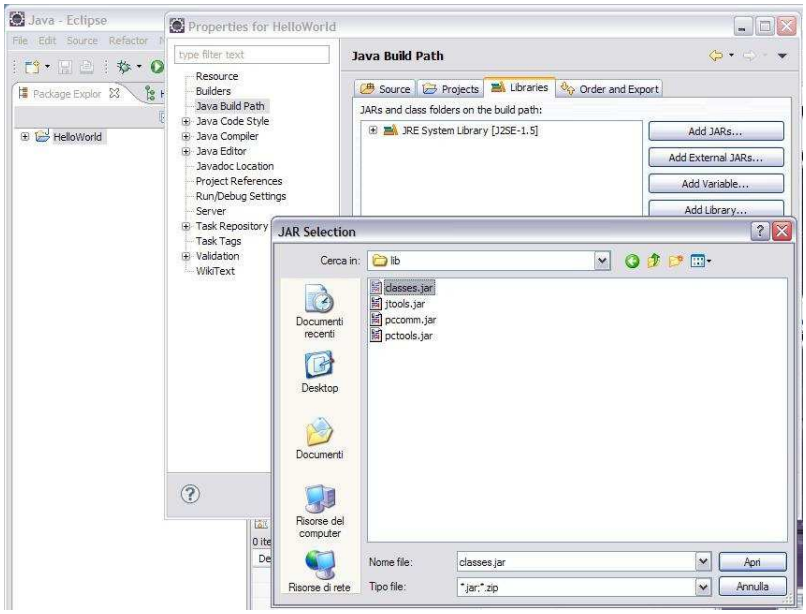

Figura 2.5: Trasformare un progetto in Lejos Project

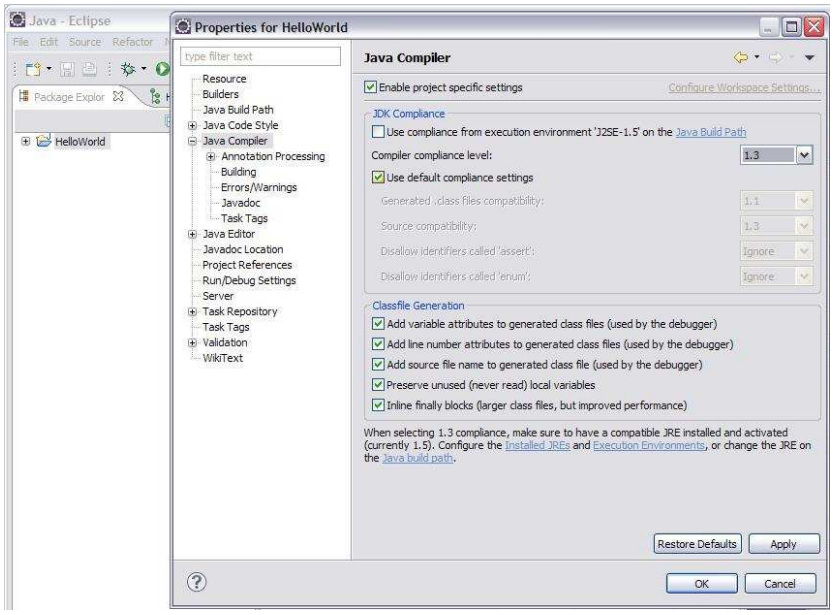

Figura 2.6: Abilitazione configurazione eclipse

2.3 Installazione e configurazione su Mac OSX

Vediamo ora in questa sezione come poter configurare gli strumenti Lejos e l'ambiente IDE su sistemi Mac OSX. Le procedure di installazione sono abbastanza più complesse

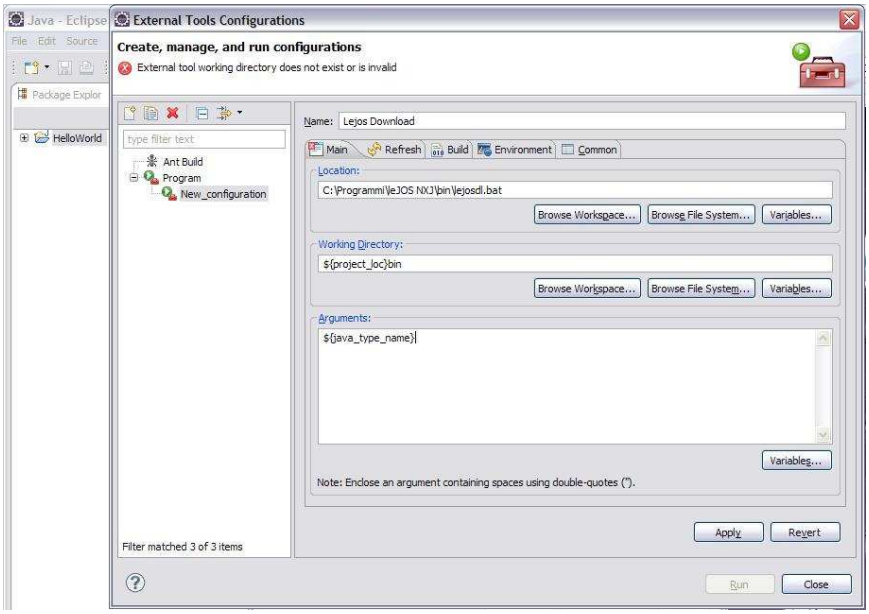

Figura 2.7: Procedura di trasferimento

rispetto ad un ambiente Windows, soprattutto perché bisogna andare a modificare dei file di sistema manualmente. Per eseguire Lejos su sistemi OSX prima di tutto bisogna aver installato il software standard della Lego, per poter cosi connettere NXT al Mac tramite usb. E' consigliabile utilizzare una versione del sistema operativo pari alla 10.5. Adesso che il driver è installato possiamo dedicarci al Java Development Kit (JDK). Per poter programmare in Java è necessario installare il software di sviluppo perché Java Runtime Environment (JRE) non è sufficiente. Il JDK dovrebbe esser già presente nel sistema OSX; eventualmente è possibile aggiornare il sistema, attraverso l'apposito meccanismo di aggiornamento o installarlo, scaricandolo dal sito http://java.sun.com/. Quando il JDK è stato correttamente installato si può procedere con il download del software Lejos. Come si vede dagli screenshot, il software lejos è stato posizionato nella cartella '/Applications', nulla vieta di sistemarlo in qualche altra cartella, ma a quel punto dovranno cambiare anche i riferimenti all'interno dei file di configurazione.

La parte difficile inizia ora con l'impostazione delle variabili d'ambiente. A differenza di Windows per OSX non esiste un pannello di controllo dal quale andare a modificare semplicemente il contenuto di queste variabili, ma si dovrà modificare il contenuto di due file di sistema. Per far questo dobbiamo eseguire l'applicazione terminale, che si trova tra le utility di sistema e posizionarci all'interno della 'home'(vedi figura 2.8).

Variabile	Valore
NXJ_HOME	La directory nella quale hai installato Lejos
JAVA_HOME	La directory nella quale hai installato JDK
PATH	Aggiungere la directory bin per il JDK e Lejos
DYLD_LIBRARY_PATH	Aggiungere la directory bin per il driver fantom

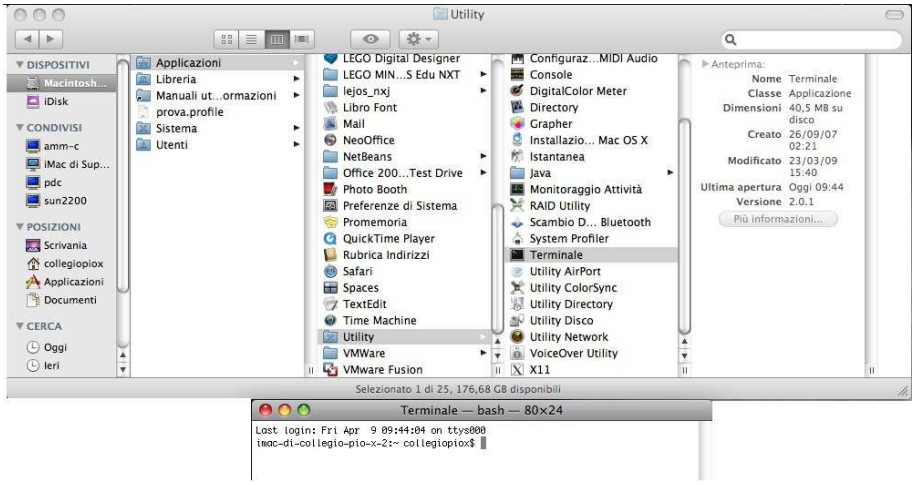

Figura 2.8: Terminale Mac OSX

Una volta raggiunta la directory ‘home’ dal terminale, si deve creare un file ‘.profile’ ed inserire le seguenti aggiunte alle variabili d’ambiente:

- export JAVA_HOME=/System/Library/Frameworks/JavaVM.framework/Versions /1.4/Home
- export NXJ_HOME=/Applications/lejos_nxj
- export DYLD_LIBRARY_PATH=$NXJ_HOME/bin
- export PATH=$PATH:$JAVA_HOME/bin:$NXJ_HOME/bin

Il comando da utilizzare è ‘sudo pico .profile’(vedi figura 2.9), il quale apre un editor creando il file indicato. L’aggiunta della parola chiave ‘sudo’, permette di acquisire i permessi di superuser per creare questo file, ovviamente richiede l’inserimento della password. Ora non resta che scrivere la precedente lista di comandi all’interno dell’editor che si è aperto(vedi figura 2.10), specificando gli esatti percorsi nei quali si trovano i file interessati, salvando e uscendo tramite la combinazione di tasti ctrl + Q. Creando questo file all’interno della ‘home’, le variabili d’ambiente verranno impostante solo per l’account in uso. Per rendere la modifica valida anche per gli altri account si deve modificare un file di sistema chiamato ‘profile’ e presente all’interno della directory ‘/etc’. Sempre dal terminale ci si deve spostare dalla ‘home’ a questa directory, tramite il comando ‘cd /etc’ e aprire il file ‘profile’ con l’editor pico e avendo acquisito

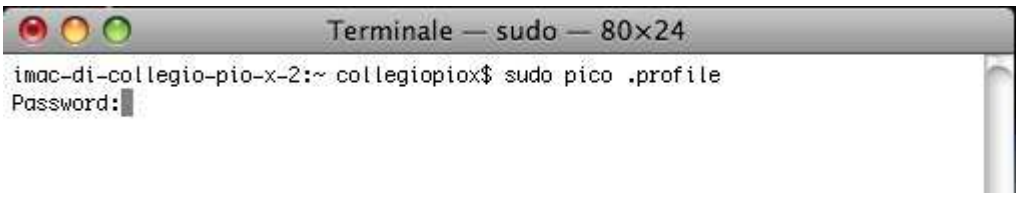

Figura 2.9: Creazione file .profile locale

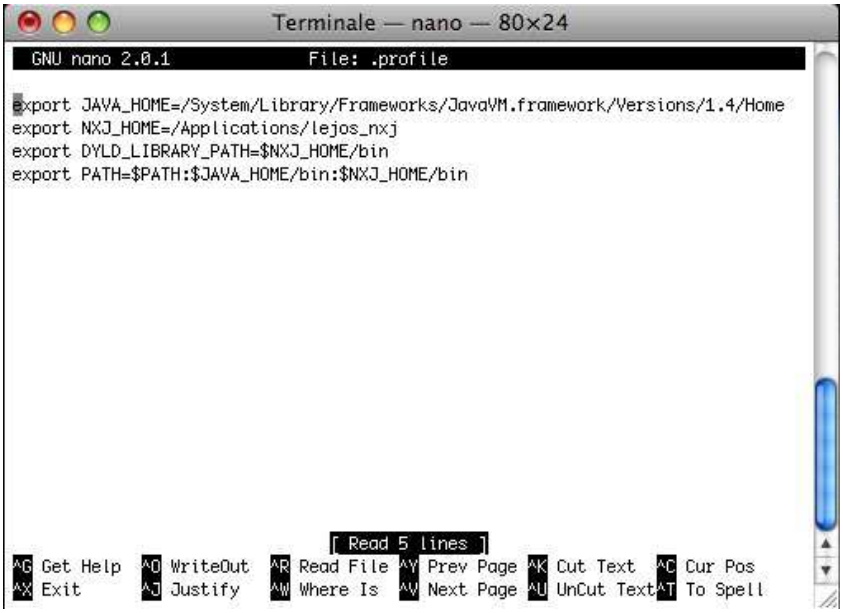

Figura 2.10: Impostazione variabili d'ambinete locali

i permessi necessari (vedi figura 2.11). Come prima quindi il comando necessario sarà: sudo pico profile. A differenza del caso precedente, in questo file va scritto il codice seguente, facendo attenzione a specificare i percorsi corretti(vedi figura 2.12).

```
# System-wide .profile for sh(1)

if [ -x /usr/libexec/path_helper ]; then
  eval `/usr/libexec/path_helper -s`
fi

if [ "${BASH-no}" != "no" ]; then
  [ -r /etc/bashrc ] && . /etc/bashrc
fi
## setloginpath added /usr/local/bin start at Fri Nov  2 18:44:47 EDT 2007
## Do not remove the previous line
if [ `whoami` != "root" ]
then
  export JAVA_HOME="/System/Library/Frameworks/JavaVM.framework/Versions
  /1.5.0/Home"
```

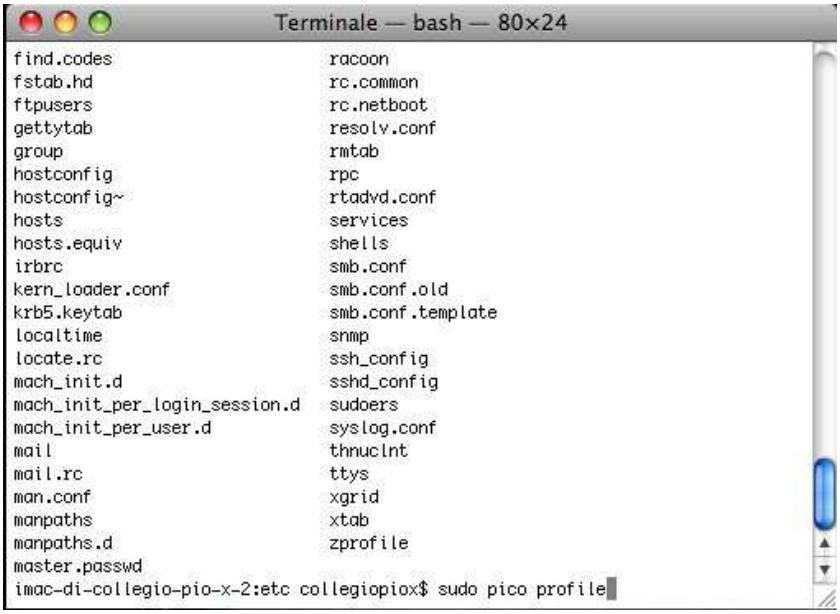

Figura 2.11: Configurazione variabili ambiente globali

```
  export NXJ_HOME="/Applications/lejos_nxj"
  export DYLD_LIBRARY_PATH="$NXJ_HOME/bin"
  PATH="$PATH:$NXJ_HOME/bin:$JAVA_HOME/bin"
  export PATH
fi
## Do not remove the next line
## setloginpath added /usr/local/bin end at Fri Nov  2 18:44:47 EDT 2007
```

Ora che le variabili d'ambiente sono correttamente impostate si può procedere con l'installazione di Eclipse. La configurazione dell'ambiente di sviluppo è praticamente uguale a quella di windows, tranne che per un piccolo particolare. In questo caso, al momento della creazione dello strumento di compilazione e download (leJOS download), si deve specificare il percorso dell'eseguibile 'nxj'. Come si vede dalla figura 3.13, nell'area dedicata alla 'Location' è stato inserito il percorso '/Applications /lejos_nxj/nxj'(vedi figura 2.13). Avendo portato a termine la precedente procedura d'installazione di Lejos, adesso si può procedere con l'aggiornamento del firmware del NXT. La procedura è equivalente al caso di windows, solo che non parte in automatico, ma è richiesta l'esecuzione da terminale del comando nxjflash. Questo comando va eseguito con NXT correttamente collegato alla porta USB e in modalità upgrade. Per selezionare questa modalità è sufficiente premere il tasto centrale del NXT. Eventualmente è possibile eseguire il comando nxjflashg per avere un interfaccia grafica. Il corretto aggiornamento del firmware verrà segnalato da dei messaggi sul terminale o sulla text area dell'interfaccia.

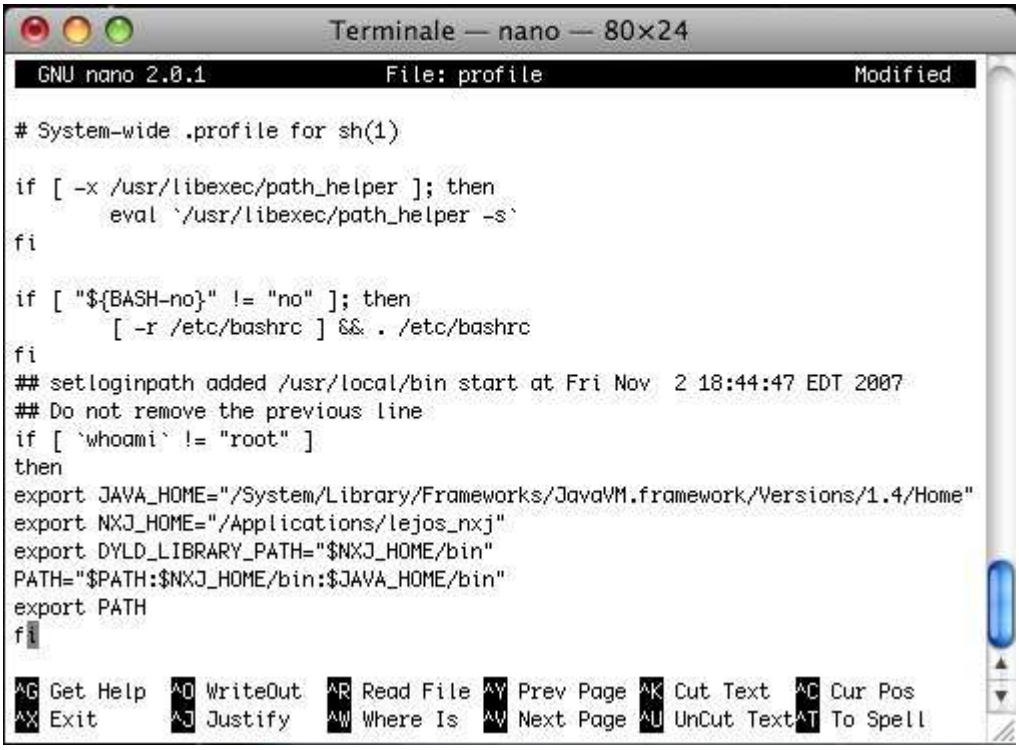

Figura 2.12: Modifica file profile

2.4 Strumenti LeJos NXJ

LeJos usa il compilatore standard di Java per compilare i programmi, ma ha bisogno di rimpiazzare le librerie standard con le sue contenute nel file classes.jar. Per questa ragione Lejos mette a disposizione il comando nxjc che non fa altro che impostare il percorso da dove vengono caricate le classi, con quello del classes.jar.I parametri usati con il comando javac rimangono gli stessi. I programmi LeJos sono diversi dai normali programmi Java perché non supportano il caricamento dinamico delle classi. Per ovviare a questo problema il linker riunisce in un file binario tutte le classi utilizzate creando un file .nxj, destinato in seguito ad essere caricato sull'NXJ. LeJos mette a disposizione i seguenti strumenti per compilare, eseguire e collegare le classi:

nxjflash aggiorna il firmaware LeJos;

nxjc compila i programmi Java per LeJos NXJ Esempio: nxjc <NomeClasse.java> Questo comando chiama il classico javac passandogli dei parametri: -bootclasspath: percorso di classes.jar -NomeClasse.java

nxjlink Collega la classe specificata con le altre referenziate nella stessa directory e con quelle presenti in classes.jar Esempio: nxjlink [-v|–verbose] [-g|–debug] [-a|–all] main-class -o <binary> L'opzione -v o -verbose crea una lista di classi e

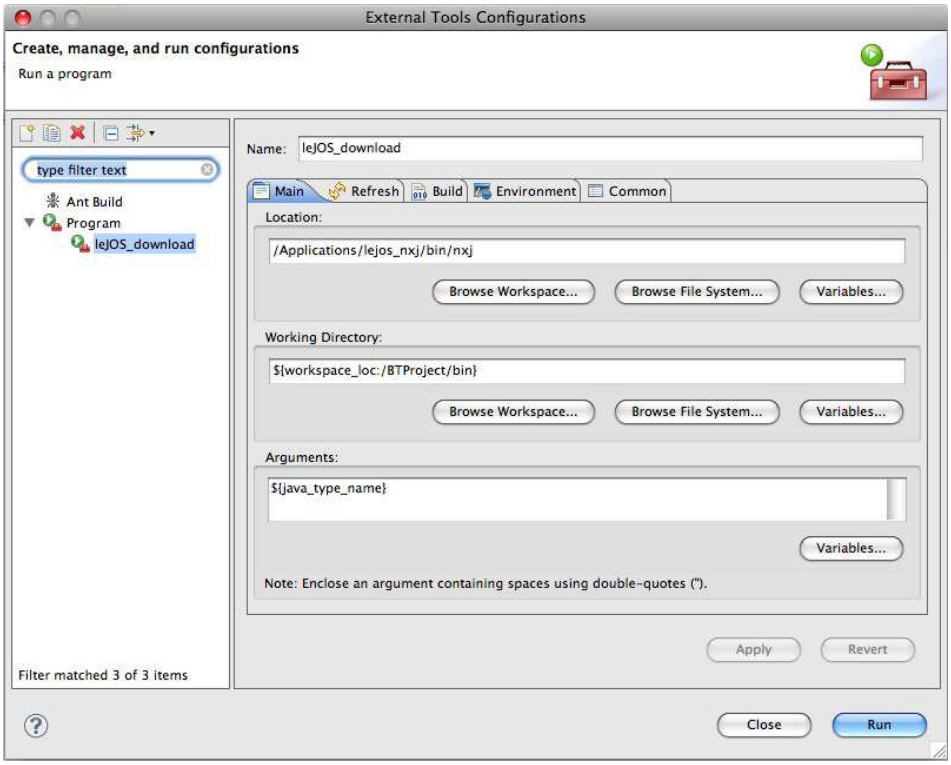

Figura 2.13: Configurazione Eclipse per Mac OSX

metodi contenuti nel file binario e li visualizza; L'opzione -g o -debug è un controllo di debug che viene incluso nel programma. Questo permette l'interruzione del programma quando è in esecuzione, con la sequenza di tasti INVIO+ESC e ritorna lo stack quando viene lanciata un eccezione non catturata. Il linker rimuove i metodi che non sono utilizzati. Specificare -a o -all per includerli tutti anche se non utilizzati. Usa l'opzione -h o -help per stampare le opzioni possibili.

nxjupload carica e eventualmente lancia il programma Esempio: nxjupload [-b|–bluetooth] [-u|–usb] [-d|–address address] [-n|–name name] [-r|–run] <binary> Di default prova il trasferimento via Bluetooth, altrimenti se è settato -bluetooth tenta direttamente il trasferimento wireless. Se è specificato il -usb prova solamente via usb. Quando si scegli di usare il bluetooth la ricerca del dispositivo, indicato dall'indirizzo –address, termina quando si stabilisce la connessione con l'elemento cercato per indirizzo. Il parametro -name limita la ricerca al solo NXT con quel nome. Se il nome non è specificato prova a connettersi a tutti gli NXT che trova, e l'upload viene fatto sul primo che si connette. Se il parametro –run è specificato, al termine del caricamento il programma verrà eseguito.

nxj crea il collegamento con le altre classi della directory e con quelle presenti in classes.jar e carica il programma nell'NXJ, eventualmente lanciandolo Esempio:

nxj [opzioni] <classe principale> Le opzioni che possono essere specificate sono quelle dell' nxjlink e dell'nxjupload.

nxjbrowse esplora i file dell'NXJ

nxjmonitor monitora e traccia sfruttando la connessione bluetooth

nxjconsole debugging via USB

nxjdataviewer visualizza in remoto i files Datalogger

nxjproxyserver proxy server per il Socket e connessioni SocketServer

emu-lejosrun emula un programma leJOS NXJ su Unix

Strumenti per usare le API PC:

- nxjpcc - compila uno o più file Java che usano le API PC Esempio: nxjpcc [javac-options] <File-Java>
- ncjpc - esegue il programma sul PC Esempio: nxjpc [java-options] <Classe-Principale>

Le opzioni possibili in questo caso sono quelle date dal comune compilatore Java.

Capitolo 3

Classi di utilizzo generico e primi programmi in Java

Si vedono in questo capitolo le classi principali di utilizzo generico che ci permetteranno di scrivere i primi semplici programmi in Java. Le classi che verranno affrontate sono:

- Battery
- Button
- Delay
- LCD

Queste classi permettono di gestire le funzionalità di base del NXT, quali i bottoni presenti sul brick, il display e la batteria. Prima di vedere come scrivere un applicazione per il robot è il caso di studiare le API, messe a dispozione da lejos e disponibili al relativo indirizzo http://lejos.sourceforge.net/nxt/nxj/api/index.html.

3.0.1 Battery

La classe Battery permette di sfruttare dei metodi per conoscere il livello di carica della batteria. Queste informazioni possono tornare molto utili nei nostri programmi per capire ad esempio quale può esser la potenza massima di un motore in un determinato momento. I metodi che mette a disposizione sono i seguenti:

- getVoltage() - ritorna il valore di tipo float che rappresenta i volt. E' un metodo statico quindi richiamabile tramite il nome della classe (Battery.getVoltage()).
- getVoltageMilliVolt() - ritorna un intero che rappresenta i millivolt. Anche questo è un metodo statico.

3.0.2 Button

Questa classe permette di utilizzare i bottoni presenti sul NXT. Definisce innanzitutto una serie di oggetti statici che identificano il singolo bottone.

- ENTER
- ESCAPE
- LEFT
- RIGHT

E un insieme di constanti intere che rappresentano l'identificativo di ogni pulsante:

- ID_ENTER
- ID_ESCAPE
- ID_LEFT
- ID_RIGHT

I metodi invece che mette a disposizione sono:

- getId() - ritorna l'intero identificativo del bottone. Può assumere i valori 1,2,4 o 8.
- isPressed() - ritorna il valore 'true' se il bottone è premuto, 'false' altrimenti.
- waitForPressAndRelease() - aspetta finché il bottone è rilasciato, poi ritorna il controllo.
- waitForPress(int timeout) - aspetta finché non viene premuto un bottone e rilasciato, ritorna l'identificativo del bottone premuto o 0 in caso scada il tempo. Il timeout rappresenta il tempo massimo di attesa in millisecondi.
- waitForPress() - è uguale al metodo precedente solo che è senza timeout.
- readButtons() - ritorna un intero rappresentante il bottone premuto, con i bit impostati nel seguente modo: 0x01 ENTER, 0x02 LEFT, 0x04 RIGHT, 0x08 ESCAPE. Ritrna 0 se non ne è premuto neanche uno.
- setKeyClickVolume(int vol) - imposta il volume associato al click del tasto.
- getKeyClickVolume() - ritorna come intero il valore corrente del volume associato al click.
- setKeyClickLength(int len) - imposta la lunghezza del click.
- getKeyClickLength() - ritorna come intero la lunghezza del click.

- setKeyClickTone(int key,int freq) - imposta la frquenza usata dal dal key passato come parametro. Per disabilitarlo impostarlo a 0.
- getKeyClickTone(int key) - ritorna un intero rappresentante la frequenza del particolare key.
- loadSettings() - Carica le impostazioni correnti associate a questa classe. Viene chaimata in automatico per inizializzare la classe e può esser richiamata per ricaricare alcune impostazioni.

I metodi waitForPressAndRelease(), waitForPress(int timeout) e waitForPress() oltre ad aspettare la pressione di un tasto, sopspendono il thread facendolo passare dallo stato Running a Ready (vedi capitolo multi-threading).

3.0.3 Delay

Questa classe mette a disposizione tre metodi statici per la sospensione del esecuzione del codice per la quantità di tempo passatagli come parametro. I metodi sono i seguenti:

- msDelay(long period) - aspetta per il numero specificato di millisecondi
- nsDelay(long period) - aspetta per il numero specificato di nanosecondi
- usDelay(long period) - aspetta per il numero specificato di microsecondi

3.0.4 LCD

Mediante l'utilizzo dei metodi mesi a disposizione di questa classe, si riesce ad accedere alle funzionalità del display. Vediamo ora i metodi principali che ci serviranno durante le prove di laboratorio:

- clear() - cancella il display;
- setPixel(int rgbColor,int x,int y) - disegna sullo schermo un pixel. rgbColor se è impostato a 0 indica bianco, se 1 indica il nero. x e y rappresentano le cordinate.
- getPixel(int x,int y) - ritorna un intero rappresentante il colore del pixel che si trova alle coordinate x,y. Come prima può assumere due valori, 0 se bianco, 1 se nero.
- drawString(String str,int x,int y,boolean invert) - stampa a schermo una stringa nella posizione passata (x,y). Se invert è uguale a 'true' la stringa viene visualizzata al contrario.
- drawChar(char c,int x,int y,boolean invert) - visualizza sullo schermo il carattere passato nella posizione (x,y). Se invert è ugulae a 'true' il carattere viene visualizzato al contratio.

- drawString(String str,int x,int y) - Visualizza una stringa alle coordinate (x,y);
- drawInt(int i,int x,int y) - Visualizza un intero alle coordinate (x,y);
- refresh() - aggiorna il display.

3.1 Sensori

Il robot NXT mette a disposizione 4 porte per collegare dei sensori. La classe SensorPort definisce al suo interno altrettante costanti di tipo SensorPort che rappresentano la porta alla quale si collega il sensore. Ogni qual volta si istanzia un nuovo oggetto di un sensore è essenziale passare al costruttore l'effettiva porta alla quale viene collegato. Le costanti che rappresentano le porte sono le seguenti:

- SensorPort.S1
- SensorPort.S2
- SensorPort.S3
- SensorPort.S4

3.1.1 Touch Sensor

La classe TouchSensor viene utilizzata per gestire il sensore di tocco. Al suo interno sono definiti il costrutture, che richiede come parametro la porta al quale è collegato e un metodo:

- TouchSensor(ADSensorPort port) - richiede come parametro la porta al quale è collegato;
- boolean isPressed() - ritorna il valore true se il sensore è premuto, altrimenti false.

3.1.2 Light Sensor

Un altro sensore disponibile nel kit di base del NXT è il sensore di luce. La classe LightSensor rappresenta l'astrazione delle sue caratteristiche e definisce i seguenti metodi:

- LightSensor(ADSensorPort port) - Costruttore che richiede la porta alla quale viene collegato il sensore;
- LightSensor(ADSensorPort port, boolean floodlight) - Costruttore che oltre alla porta richiede di specificare se viene utilizzato come sensore di riflessione.
- void calibrateHigh() - questo metodo viene chiamato quando sta leggendo valori alti, viene usato da readValue();

- void calibrateLow() - questo metodo viene chiamato quando sta leggendo valori bassi, viene usato da readValue();
- Colors.Color getFloodlight() - ritorna il colore letto dalla riflessione, compreso Color.NONE;
- int getHigh() - ritorna il valore normalizzato corrispondente al massimo readValue() possibile;
- int getLightValue() - ritorna il valore calibrato e normalizzato della luminosità della luce bianca, 0 indica l'oscurità e 100 un intensa luce;
- int getLow() - ritorna il valore corrispondente al minimo readValue() possibile;
- int getNormalizedLightValue() - ritorna il valore normalizzato di luce letto. La codifica va da 0 a 1023, ma solitamente il range va da 145(scuro) a 890(luce solare);
- boolean isFloodlightOn() - controlla se la riflessione è attiva e ritorna true, altrimenti false;
- int readNormalizedValue() - ritorna il valore letto normalizzato;
- int readValue() - ritorna il valore letto;
- void setFloodlight(boolean floodlight) - attiva o disattiva il LED. Un valore true lo accende.
- boolean setFloodlight(Colors.Color color) - viene utilizzato per accendere e spegnere la luce in base al colore. Se il sensore lavora con più colori si può controllare se il LED è acceso o spento. Se il colore non esiste non fa niente e ritorna false;
- void setHigh(int high) - imposta il massimo valore possibile che può ritornare readValue();
- void setLow(int low) - imposta il minimo valore possibile che può ritornare readValue().

I colori che sono codificati dalla classe Color sono:

- BLACK
- BLUE
- GREEN
- NONE
- RED

- WHITE
- YELLOW

Essendo rappresentati con una struttura particolare, possono essere richiamati in questo modo: Colors.Color BLACK.

3.1.3 Sound Sensor

La classe SoundSensor permette di accedere alle funzionalità del sensore di suono. Al suo interno sono definiti due costruttori e due metodi:

- SoundSensor(ADSensorPort port) - questo costruttore richiede come parametro la porta alla quale viene collegato il sensore;
- SoundSensor(ADSensorPort port, boolean dba) - oltre alla porta richiede di indicare la modalità che si vuole, DB o DBA. La modalità DBA fa ritornare dei valori compatibili con la percezione dell'orecchio umano. Un valore true attiva questa seconda modalità, false invece attiva la modalità DB;
- int readValue() - ritorna il valore letto corrente come una percentuale;
- void setDBA(boolean dba) - Un valore true passato come parametro imposta la modalità DBA.

3.1.4 Ultrasonic Sensor

Per creare degli oggetti che gestiscono il sensore ad ultrasuoni si deve utilizzare questa classe. Il costruttore è unico e richiede solamente la porta di connessione, mentre i metodi permettono di calibrare il sensore o di leggere i dati catturati:

- UltrasonicSensor(I2CPort port) - costruttore che richiede come parametro la porta di connessione;
- int capture() - imposta il sensore nella modalità capture. Questa modalità permette di capire se nelle vicinanze è presente un altro sensore ad ultrasuoni, che renderebbe i dati letti non attendibili. Ritona il valore 0 se è tutto apposto, un valore diverso altrimenti.
- int continuous() - passa alla modalità di ping e di cattura continua. Questa è la modalità di default del sensore, ritorna 0 se è tutto apposto, un valore diverso altrimenti.
- int getCalibrationData(byte[] data) - imposta 3 byte di calibrazione. Il primo byte contiene zero, il secondo un fattore di scala e il terzo un fattore di divisione. Ritorna 0 se è tutto apposto, un valore diverso altrimenti.

- byte getContinuousInterval() - ritorna l'intervallo (range da 1 a 15) della modalità continua. Ritorna -1 in caso di errore.

- int getData(int register,byte[] buf,int len) - esegue una lettura I2C e ritorna 0 in caso di successo, un valore diverso altrimenti.

- int getDistance() - ritorna la distanza da un oggetto o il valore 255 in caso non ce ne siano all'interno del range. Questo metodo deve attendere una piccola quantità di tempo per rendere valida la lettura del dato.

- int getDistances(int[] dist) - scrive all'interno del array 8 letture. Ritorna 0 in caso di successo, un valore diverso altrimenti. L'array può contenere il valore 255 nel caso in cui non ci siano oggetti a porata del sensore. Questo metodo può aspettare una piccola quantità di tempo prima di effettuare la letture, nel caso in cui i dati non siano ancora disponibili. Viene utilizzato nella modalità ping.

- byte getMode() - ritorna la modalità operativa corrente del sensore: 0 sensore spento, 1 singola acquisizione modo ping, 2 modalità continua, 3 acquisizione ad eventi. Ritorna -1 in caso si verifichi un errore.

- float getRange() - ritorna la distanza dell'oggetto più vicino.

- String getUnits() - ritorna una stringa che indica il tipo di unità utilizzata. Di default torna 10E-2m che indica i centimetri.

- int off() - disattiva il sensore. Non viene eseguito più nessun ping finché non viene resettato questo stato. Ritorna 0 in caso di successo, un valore diverso altrimenti.

- int ping() - invia un singolo ping.Il sensore può operare in due modi: continuo e ping. Se lavora in modalità continua, il ping viene inviato il più possibile e i dati possono esser letti appena sono pronti, dal metodo getDistance(). Quando invece lavora nella modalità ping, il ping è trasmesso solo quando si effettua una chiamata a questo metodo. Con un singolo ping si catturano un massimo di 8 valori. Questi valori possono esser letti grazie il metodo getDistance e passandogli un array. Il ping viene effettuato automaticamente se viene chiamato direttamente getDistance. Dal momento che si effettua la chiamata devono passare aprossivamente 20ms per avere i dati. Ritorna 0 in caso di successo, un valore diverso altrimenti.

- int reset() - effettua un soft reset del dispositivo. Ripristina la configurazione di default. Dopo questa chiamata il sensore lavora in modalità continua. Ritorna 0 in casso di successo, un valore diverso altrimenti.

- int sendData(int register,byte[] buf,int len) - esegue una transizione in scrittura di tipo I2C. Ritorna 0 in casso di successo, un valore diverso altrimenti.

- int setCalibrationData(byte[] data) - imposta 3 byte di calibrazione. Il primo byte contiene zero, il secondo un fattore di scala e il terzo un fattore di divisione. Ritorna 0 se è tutto apposto, un valore diverso altrimenti.
- int setContinuousInterval(byte interval - imposta l'intervallo di ping per la modalità continua. Ritorna 0 se è tutto apposto, un valore diverso altrimenti.

3.2 Scrivere il primo programma

3.2.1 Hello World

Le API viste in precedenza sono solo una piccola parte di quelle che mette a disposizione lejos, ma sono le prime che bisogna conoscere per riuscire ad accedere alle funzionalità di base del robot. Vediamo ora come realizzare un primo semplice programma Java andando ad utilizzare le conoscenze apprese ed eventualmente utilizzando la documentazione presente all'indirizzo http://lejos.sourceforge.net/nxt/nxj/api/index.html. Per i meno esperti di Java è consigliabile utilizzare un ambiente di sviluppo IDE quale Eclipse, che consente la realizzazione di progetti, segnala in modo più chiaro eventuali errori di scrittura del codice e infine, avendolo configurato come descritto nel capitolo 3, permette un semplice download degli applicativi all'interno del brick.

Il primo passo da compiere è quello di creare un nuovo progetto e di trasformarlo in un Lejos Project. Il programma che andiamo a creare sarà il classico Hello World, cioè dovrà visualizzare sul display del NXT la scritta di saluto. Per far questo dobbiamo aggiungere al progetto la classe principale, inserendola dal menu File=>New=>Class, chiamarla con il nome del nostro programma (HelloWorld) e includere il metodo main, spuntando l'opzione in fase di creazione della classe. Ora che abbiamo a disposizione la struttura generale, possiamo scrivere il codice all'interno del metodo main o eventualmente richiamarlo utilizzando delle funzioni di supporto. Per visualizzare sul display dei messaggi dobbiamo utilizzare i metodi messi a disposizione dalla classe LCD, ma prima bisogna importarla.

```
import lejos.nxt.LCD;
```

L'obiettivo è quello di visualizzare la scritta 'Hello World' in posizione (2,2) del display. Riguardando le API della classe LCD, il metodo che fa al caso nostro è drawString(String str,int x,int y), che è un metodo statico e quindi per richiamarlo non serve definire un nuovo oggetto, ma basta richiamarlo tramite il nome della classe.

```
public class HelloWorld {
 public static void main(String[] args) {
  LCD.drawString("HelloWorld",2,2);
  LCD.refresh();
   while(true) {}
   }
}
```

Il metodo refresh() viene invece chiamato per aggiornare la visualizzazione del display. Per dare il tempo all'utente di visualizzare la scritta si deve mettere in attesa il

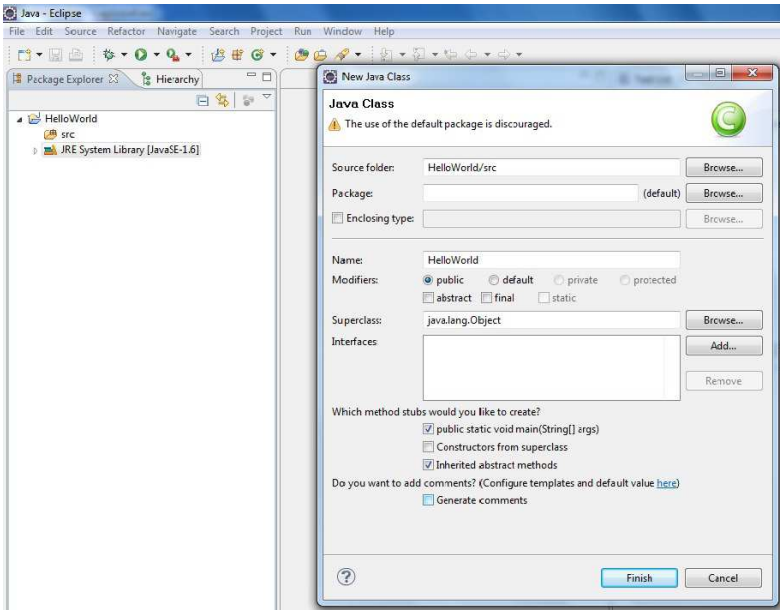

Figura 3.1: Creazione classe principale

programma, altrimenti terminerebbe immediatamente senza dare la possibilità di leggere il messaggio. Per porre rimedio a questo problema basta inserire un ciclo infinito, cioè con la condizione sempre verificata e l'esecuzione del programma resterà bloccata all'interno del ciclo while(true). Quest'ultima non è una buona soluzione perché per terminare il programma saremmo costretti a riavviare il robot mediante la pressione dei due tasti centrali. Una soluzione migliore e più elegante è quella di far attendere al programma una determinata quantità di tempo prima di farlo terminare. Controllando le API, la soluzione più adatta ce la fornisce la classe Delay.

```
import lejos.util.Delay;
public class HelloWorld {
 public static void main(String[] args) {
  LCD.drawString("HelloWorld",2,2);
  LCD.refresh();
   Delay.msDelay(5000);
   }
}
Vediamo ora come è possibile visualizzare il messagio
```

Grazie al metodo msDelay(5000), ottenuto importando la relativa classe, la scritta resterà visualizzata sul display per 5s, al termine dei quali il programma terminerà tornando il controllo a Lejos. In questo modo non siamo costretti a riavviare il brick. Adesso che il programma è pronto non resta altro che trasferirlo al NXJ utilizzando lo strumento di download creato seguendo le istruzioni del capitolo precedente. Ovviamente NXJ dovrà esser collegato correttamente al computer e acceso, altrimenti il trasferimento fallirà.

3.2.2 Muovere il robot

Adesso che abbiamo capito come scrivere una semplice applicazione, vediamo come è possibile sfruttare i metodi della classe Button per avviare il movimento del robot. Per muovere il brick dobbiamo importare la classe Motor, la quale mette a disposizione tre istanze della classe stessa, che permettono di chiamare i metodi per muovere il robot. I principali metodi da prendere in considerazione sono:

- forward() - fa ruotare il motore in avanti;
- backward() - fa ruotare il motore indietro;
- reverseDirection() - inverte la direzione del motore;
- setSpeed(int speed) - imposta la velocità, in gradi al secondo. Valore massimo 900 con 8 Volt.
- stop() - ferma il motore quasi istantaneamente.

Per maggiori informazioni sulla classe Motor, sul funzionamento di tutti i suoi metodi e per conoscere i limiti dei motori, si veda la componente hardware. Avendo già al suo interno tre istanze della classe per i tre motori collegabili al brick, non serve definire nuovi oggetti della classe Motor, ma i metodi sono richiamabili sfruttando gli oggetti già definiti. Ad esempio per impostare la velocità del motore collegato alla porta A e farlo ruotare in avanti è sufficiente scrivere la seguente istruzione:

```
Motor.A.setSpeed(360);
Motor.A.forward();
```

Vediamo ora il codice necessario per far muovere in avanti un robot che sfrutta la spinta di due motori. Vogliamo che il robot inizi il suo cammino solo dopo la pressione del tasto sinistro presente sul brick e che si fermi dopo la pressione del tasto d'uscita.

```
import lejos.nxt.Button;
import lejos.nxt.LCD;
import lejos.nxt.Motor;
public class Esempio2 {
 public static void main(String[] args) {
  LCD.drawString("Premi LEFT per avviare!!",0,0);
   while(!Button.LEFT.isPressed());
  LCD.drawString("Avvio robot!!",0,1);
   Motor.B.setSpeed(360);
   Motor.C.setSpeed(360);
   Motor.B.forward();
   Motor.C.forward();
  LCD.drawString("Premi Escape per terminare!",0,2);
   while(!Button.ESCAPE.isPressed());
   Motor.B.stop();
   Motor.C.stop();
 }
}
```

Il primo ciclo while blocca il proseguirsi del programma finché non viene premuto il tasto LEFT del brick. A quel punto il metodo isPressed() ritorna true, ma la condizione venendo negata causa l'uscita dal ciclo e l'avvio del robot. Alla fine il robot continua ad andare avanti finché non viene premuto il tasto ESCAPE. E' interessante notare che per l'utilizzo dei motori non è stato necessario definire assolutamente niente, ma semplicemente è stato sufficiente importare la relativa classe.

3.2.3 Utilizzo dei sensori

Ora vediamo come realizzare una semplice applicazione che definisca tre oggetti per tre tipi diversi di sensori e che stampi sullo schermo i valori catturati finché non venga richiesta la terminazione della visualizzazione, mediante la pressione di un tasto.

```
lejos.nxt.SoundSensor;
lejos.nxt.UltrasonicSensor;
lejos.nxt.LightSensor;

public class Esempio3 {
 public static void main (String[] args) {
  LightSensor light = new LightSensor(SensorPort.S3);
  SoundSensor sound = new SoundSensor(SensorPort.S2);
  UltrasonicSensor ultrasonic = new UltrasonicSensor(SensorPort.S1);
  LCD.drawString("Premi LEFT", 0, 0);
  LCD.drawString("per terminare", 0, 1);
  while (! Button.LEFT.isPressed()){
   LCD.drawInt("Light:" + light.readValue(), 3, 0, 2);
   LCD.drawString("Sound: " + sound.readValue(), 0, 3);
   LCD.drawString("Distance(cm): " + ultrasonic.getDistance(), 0, 4);
   Delay.msDelay(500);
  }
 }
}
```

Da come si vede dal codice, quando si vanno a creare i tre oggetti per i tre tipi di sensori, LightSensor, SoundSensor e UltrasonicSensor si devono collegare rispettivamente alla porta 3, 2 e 1 come stabilito dai valori passati ai costruttori.

Come ultimo esempio resta da vedere come si può utilizzare il TouchSensor in un semplice programma. Il codice che segue ha il compito di far andare avanti il robot finché il TouchSensor non rileva una pressione. In caso questa si verifichi, il robot dovrà fermarsi.

```
import lejos.nxt.*;

public class Esempio4 {
 public static void main (String[] args) {
  TouchSensor t = new TouchSensor(SensorPort.S1);
  LCD.drawString("Avvio robot!", 0, 0);
  Motor.B.setSpeed(360);
  Motor.C.setSpeed(360);
  Motor.B.forward();
  Motor.C.forward();
```

```
   while (!t.isPressed());
   Motor.B.stop();
   Motor.C.stop();
   LCD.drawString("Fine!!", 0, 1);
   Delay.msDelay(1000);
  }
}
```

L'unica novità in questo semplice programma è la creazione di un oggetto di tipo TouchSensor collegato alla porta 1 e l'utilizzo del suo metodo isPressed(). Il ciclo while presente, controlla continuamente se il sensore è premuto e non appena il metodo torna un valore true, viene negato ed esce dal ciclo.

3.2.4 Altri esempi

In questa sezione vengono proposti gli ultimi esempi utili. Vediamo nel primo esempio come è possibile visualizzare il livello della batteria.

```
import lejos.nxt.*;

public class Esempio5 {
 public static void main(String[] args) throws Exception {
  LCD.drawString("Livello Batteria: " + Battery.getVoltage(), 0, 0);
  Delay.msDelay(3000);
 }
}
```

Una soluzione alternativa alla tecnica utilizzata nei precedenti esempi per attendere la pressione di un tasto, consiste nel sfruttare il metodo waiForPressAndRelease().

```
import lejos.nxt.*;

public class Esempio6 {
 public static void main(String[] args) throws Exception {
  LCD.drawString("Attesa LEFT!!", 0, 0);
  Button.LEFT.waitForPressAndRelease();
  LCD.drawString("OK!", 0, 1);
  Delay.msDelay(3000);
 }
}
```

A differenza di prima, non serve inserire un ciclo while perché l'attesa è già incapsulata nel metodo waitForPressAndRelease().

Capitolo 4

Multithreading

In questo capitolo verrà presentata una panoramica della programmazione multithreading per NXT, una delle caratteristiche più interessanti. Il Multithreading dà la possibilità al programmatore di realizzare più Thread, cioè la suddivisione del codice in più flussi i quali verranno eseguiti 'concorrentemente'. In questo modo sarà possibile abbandonare il classico stile di programmazione sequenziale riuscendo a suddividere il programma in più parti e soprattutto a far eseguire la varie parti (Thread) in modo contemporaneo. Un esempio è la lettura di dati da più sensori collegati al brick, per far svolgere al robot delle azioni basate sull'utilizzazione di questi dati, ovviamente la lettura dovrà esser fatta in parallelo. Verrà fatto poi uno studio degli strumenti che ci mette a disposizione il Lejos per superare le problematiche dello 'spaghetti code', cioè codice molto complesso e incomprensibile, sfruttando l'uso dei Behavior.

4.1 Concetti generali

Prima di iniziare è fondamentale uno studio generale dei concetti di base del multithreading. Un classica applicazione scritta in modo normale non dà la possibilità di servire più richieste contemporaneamente, infatti sarebbe costretta a finire l'esecuzione di una richiesta prima di poterne servire un'altra. Scrivendo applicazioni che sfruttano il multi-threading è invece possibile suddividere il classico processo in più thread, ovvero suddividere il processo principale in diversi flussi di codice concorrenti. L'architettura hardware della macchina è fondamentale per il funzionamento di questo principio; infatti in una macchina multi-processore ogni thread viene eseguito su un processore diverso. Questo non è il caso del NXT che ha un unico processore ARM 7 e quindi delega il compito di parallelizzare l'esecuzione dei thread al sistema operativo Lejos. Ovviamente questa è solo una simulazione software dato che essendo unico il processore può eseguire una sola istruzione alla volta. In questi casi si parla di multi-threading a divisione di tempo e il cambio di esecuzione da un thread ad un altro avviene con sufficiente frequenza da non rendersi conto che in realtà ne viene eseguito solo uno alla volta.

Da quanto detto finora sono stati introdotti due concetti molto simili tra loro, ovvero il concetto di processo e di thread. La traduzione letterale del nome completo thread of execution è filo d'esecuzione e sta proprio ad indicare l'unità atomica nella quale un processo si può suddividere. Un processo è sempre formato da almeno un thread, quello principale che si avvia dopo il caricamento e può generarne molti altri che verranno eseguiti concorrentemente. La differenza principale tra questi due elementi sta nel fatto che più processi sono indipendenti mentre più thread condividono informazioni di stato, memoria e altre risorse. Un altro concetto importante che è appena stato introdotto è quello di risorsa. Una risorsa non è altro che una rappresentazione di qualsiasi elemento/entità che serve ad un processo per portare a termine la propria attività. Può essere ad esempio un area di memoria, dati prodotti da un sensore, la CPU o nel caso del NXT un motore. Le risorse possono essere condivisibili quando sono utilizzabili da più di un processo nello stesso tempo o non condivisibili quando possono essere utilizzate da uno solo alla volta, ovvero in mutua esclusione. Possono essere consumabili se non è possibile un riutilizzo, sottraibili se è possibile sottrarre la risorsa al processo che la sta utilizzando (preemption) o non sottraibili se la risorsa può essere rilasciata solo dal processo che la sta utilizzando. L'utilizzazione di risorse non condivisibili da più thread fa sì che la programmazione multi-threading diventi molto complessa, proprio perché si deve fare molta attenzione a chi sta usando una risorsa prima di occuparla, utilizzando meccanismi di sincronizzazione. Ad esempio un thread che sta utilizzando una variabile potrebbe vedersi modificare prima del termine della sua esecuzione, il valore da un altro thread causando cosi degli errori nel risultato finale. Questo è il classico problema di mutua esclusione e si risolve utilizzando complessi meccanismi di sincronizzazione messi a disposizione dal linguaggio, che vietano a più thread di lavorare contemporaneamente sulle stesse risorse non condivisibili .

4.1.1 Stallo

Una conseguenza critica alla quale si va incontro a seguito di una cattiva politica di allocazione delle risorse è lo stallo, ovvero la non possibilità da parte di uno o più processi di raggiungere lo stato finale e questo accade quando si utilizzano risorse non condivisibili. Un esempio di stallo è il seguente: due processi P1 e P2 richiedono l'utilizzo di due risorse non condivisibili R1 e R2 in ordine temporale inverso, cioè il processo P1 acquisisce prima R1 e P2, R2. In un secondo momento per terminare la propria esecuzione il processo P1 ha bisogno anche dell'allocazione di R2, mentre P2 di R1. Non essendo le risorse condivisibili e non venendo rilasciate dai processi che le detengono, attendono indefinitamente la liberazione della risorsa di cui hanno bisogno. Questa è classica situazione di stallo che non permette al sistema di evolvere verso lo stato finale creando un blocco critico. Le condizioni necessarie per il verificarsi di un blocco critico sono:

- Mutua esclusione: le risorse non possono essere utilizzate contemporaneamente;
- Allocazione parziale: un processo richiede le risorse con il proseguire della sua evoluzione e non tutte quante all'inizio;

- Non sottraibilità delle risorse: solo il possessore della risorsa può cederla e quindi non può essere sottratta da nessun altro (no preemption);
- Attesa circolare: in presenza di n processi, con n maggiore di uno, quando il processo P1 attende la liberazione di una risorsa posseduta da P2 e P2 ne attende una posseduta da P3 e Pn attende quella di P1, si è in presenza di un attesa circolare dato che nessuna rilascia la risorsa posseduta.

Un sistema di processi che presenta tutte e quattro le precedenti condizioni è soggetto a finire in stallo,ma essendo solo necessarie e non sufficienti non è detto che il sistema si blocchi. Rimuovendone anche solo una di esse eviterebbe tale rischio.

4.1.2 Stati di un processo

Ovviamente come detto prima possono essere effettivamente in esecuzione contemporanea solo un numero massimo di thread pari al numero dei processori disponibili, uno nel caso del NXT, tutti gli altri dovranno esser gestiti da delle apposite code di sospensione attraverso uno scheduler. Da questo si deduce che un processo avrà a disposizione la CPU per un breve periodo di tempo prima di cederla, in questo modo la vita di un thread passa attraverso più stati.

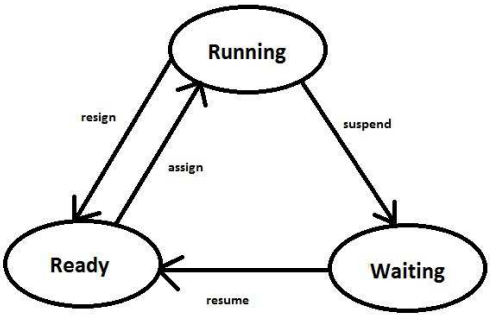

Figura 4.1: Stati di un processo

Gli stati in cui un processo può trovarsi sono i seguenti:

- Running se è in esecuzione;
- Ready se è pronto a procedere e quindi aspetta solo l'allocazione di un processore;

- Waiting se non può procedere, ad esempio in attesa di un trasferimento dati.

Come si vede dalla figura ad un processo in esecuzione, quindi che si trova nello stato Running, può essere tolto il processore dallo scheduler (transizione resign), ad esempio perché ha terminato il tempo che gli era stato concesso, ma non essendo terminato viene posto nella coda che rappresenta lo stato Ready. Il processo resterà in questo stato finché non toccherà di nuovo a lui riprendere il controllo della CPU (transizione assign). Dallo stato Running si può invece passare a quello waiting quando il processo non può proseguire con la sua esecuzione, ad esempio quando sta attendendo un trasferimento dati da una periferica o ha trovato un blocco sull'acquisizione di una risorsa (transizione suspend). Quando il trasferimento dati viene terminato o la risorsa si libera sarà compito di un altro processo risvegliare quello nella coda di waiting portandolo nello stato ready (transizione resume).

4.1.3 Scheduler NXJ

Lo scheduler del NXJ è una componente software del firmware Lejos che ha il compito di gestire l'utilizzo della CPU da parte dei vari processi. Esistono diversi algoritmi di scheduling, ma quello utilizzato dal robot NXJ è uno scheduling real-time con inversione di priorità. La differenza principale tra algoritmi real-time e quelli normali è la garanzia che devono dare nella risposta a eventi esterni entro tempi stabiliti. In questi sistemi quindi è di fondamentale importanza il rispetto delle scadenze temporali, anche dette deadline, cioè il massimo ritardo che può avere un processo per non compromettere la funzionalità del sistema. Le scadenze temporali, deadline, possono essere di due tipi: hard-realtime quando non viene tollerato un eventuale ritardo e soft-realtime quando è a volte tollerabile un ritardo. Lo scheduler manda in esecuzione i thread a più alta priorità bloccando di fatto quelli con priorità più bassa, mentre i thread aventi stessa priorità vengono gestiti tramite timesliced, ovvero la CPU viene assegnata ad un thread per una quantità di tempo stabilita, al termine viene assegnato ad un altro thread. L'inversione di priorità si manifesta ad esempio quando un processo A con priorità alta cerca di acquisire una risorsa mutuamente esclusiva già in possesso di un processo B a bassa priorità ed è quindi costretto ad aspettare che il processo B la liberi. Se a questo punto arriva un processo C a media priorità a cui non interessa la risorsa contesa da B e A, ma avendo priorità maggiore di B, passerebbe avanti a B causando un prolungato ritardo ad A, dato che questo non può procedere perché la risorsa mutuamente esclusiva è ancora in possesso di B sospeso. In questo caso non si rispetterebbe la priorità dei thread causando ritardi non prevedibili. Una possibile soluzione a questo problema è l'eredità della priorità, che consiste nel cedere la priorità del thread che attende la liberazione della risorsa occupata da uno a priorità più bassa, finché questo non rilascia la zona o la risorsa mutuamente esclusiva. In questo modo si riesce ad evitare l'inversione di priorità. Nel caso specifico del NXJ viene mandato in esecuzione sempre il thread a priorità più alta, in presenza di thread con stessa priorità si gestiscono tramite timesliced, garantendo che non avvenga un inversione di priorità.

4.1.4 Primitive wait&signal

Prima di introdurre i meccanismi di sincronizzazione che mette a disposizione Java, si devono introdurre dei concetti fondamentali che stanno alla base di un sistema multi-threading. Da quello che abbiamo capito, in un sistema mono-processore per poter evolvere più processi insieme, la CPU dovrà esser assegnata ad ogni processo per un periodo di tempo, o comunque ci dovrà esser alla base una politica di gestione dell'assegnazione della CPU. Questo fa sì che un processo che sta utilizzando la CPU possa esser interrotto in qualsiasi momento, anche quando sta eseguendo delle istruzioni che per sicurezza non potrebbero esser interrotte. La sincronizzazione tra più processi, riguarda proprio questa problematica. La soluzione che si presenta è quella di rimuovere la possibilità di interrompere il processo nello stato running, quando sta eseguendo una sezione di codice che per sicurezza non deve esser interrotto. Quindi un processo che vuole eseguire del codice in mutua esclusione dovrà possedere questo meccanismo detto lock, che gli permette di rimuovere eventuali interruzioni. E' ovvio che al termine della sezione eseguita in mutua esclusione, il processo dovrà riabilitare le interruzione tramite un unlock.

4.1.5 Multithreading in Java

Qualsiasi programma viene eseguito, la JVM gli crea un thread con il compito di lanciare il metodo main(). A sua volta questo thread creato in automatico può generarne degli altri sfruttando le potenzialità messe a disposizione dalla classe Thread. Per definire un nuovo thread è obbligatorio implementare il metodo run() presente nell'interfacia Runnable, in seguito il creatore dovrà avviarlo chiamando il metodo start(). Per definire un nuovo thread ci sono due strade che si possono percorrere. La prima consiste nell'ereditare i metodi della classe Thread che a sua volta implementa l'interfaccia Runnable.

```
public class NuovoThread extends Thread {
 public void run() {
 //Qui va il codice da far eseguire al nuovo thread
}
 public static void main(String[] args) {
  NuovoThread nt = new NuovoThread();
  nt.start();
 }
}
```

Come si vede nel precedente esempio si ereditano i metodi da Thread e si deve ridefinire il metodo run(), scrivendo all'interno il codice principale del nuovo thread. Il thread inizierà a concorrere per l'uso della CPU subito dopo il suo avvio (nt.start()). Al termine della procedura run(), terminerà.

La seconda tecnica che si può utilizzare per definire un thread consiste nell'implementare l'interfaccia Runnable. Questa è utile quando la nostra classe deve ereditare già delle caratteristiche da un'altra classe che non sia Thread e come noto Java non permette l'ereditarietà multipla.

```
public class NuovoThread extends MyClass implements Runnable {
 public void run() {
  //Qui va il codice da far eseguire al nuovo thread
 }
 public static void main(String[] args) {
  NuovoThread nt = new NuovoThread();
  Thread t = new Thread(nt);
  t.start();
 }
}
```

Come si può vedere dall'esempio precedente la classe NuovoThread eredita dei metodi da MyClass e per poter definire un nuovo thread si è costretti ad implementare l'interfaccia Runnable. Come prima il codice principale va scritto all'interno del metodo run(). La differenza sta nel fatto che ora bisogna creare un nuovo oggetto di tipo Thread e passargli al costruttore l'oggetto creato tramite la nostra classe. Ora siamo in grado di lanciare il metodo start() per avviare il nostro thread.

4.1.6 Metodi ereditati da Thread

Ereditando la classe Thread non si ereditano solo il metodo run() e start(), ma ci sono anche molti altri metodi interessanti che permettono di modificare lo stato di un thread. Riprendendo la figura precedente rappresentante gli stati di un processo, si riesce a far passare un thread in esecuzione dallo stato Running a quello Ready o Waiting e viceversa, utilizzando gli strumenti acquisiti. Quando un thread si trova sospeso nello stato Waiting non compete per l'acquisizione della CPU e le motivazioni per cui si può trovare in questo stato sono le seguenti:

- attesa del completamento di un operazione di input output;
- attesa di un periodo ti tempo, Thread.sleep();
- attesa di una segnalazione di risveglio, dopo che il thread si è messo in wait();
- sospensione causata da un altro thread, Thread.suspend().

Un thread può esser risvegliato per le seguenti motivazioni:

- completamento dell'operazione di input output;
- termine del tempo di attesa;
- arrivo della segnalazione di risveglio, notify();
- riattivazione da parte di un thread, metodo resume().

4.1.7 Monitor di Java

Il monitor è un costrutto che risolve il problema della mutua esclusione su variabili condivise. Ogni processo può tranquillamente accedere a tutte le variabile private che utilizza esclusivamente lui. Il problema sorge quando deve lavorare con variabili o procedure condivise tra più thread. E' chiaro che se più processi lavorassero contemporaneamente su una variabile non condivisibile, nascerebbero sicuramente dei problemi di inconsistenza dei dati. Il monitor non è altro che un sistema che associa ad un insieme di variabile condivise, delle procedure, le quali sono le uniche che possono accedere a queste variabili sensibili. Se qualche processo cerca di accedere alle variabili condivise mentre sono in uso già da un altro, il monitor lo bloccherebbe in una coda di attesa. Quando il thread che le sta utilizzando rilascia le variabili o la sezione critica, verranno sbloccati gli altri processi in attesa, i quali concorreranno per l'acquisizione del monitor. Questo costrutto realizza appieno il concetto di mutua esclusione e inoltre permette al processo di sospendersi all'interno della procedura di monitor nel qual caso non sia verificata una condizione per lui fondamentale. Riassumendo quanto detto lo scopo di un monitor è mettere a disposizione un insieme di procedure per dare l'accesso alle risorse in mutua esclusione e il funzionamento rispecchia la seguente logica:

- le procedure di monitor vengono eseguite in mutua esclusione e tutti i processi che ne richiedono l'uso, mentre è già assegnato ad un altro, devono attendere il completamento delle operazioni da parte di quest'ultima;inoltre regolano anche l'ordine di accesso alle risorse;

- se un processo che ha acquisito il monitor non può portare a termine la procedura, dopo una verifica dello stato delle variabili locali, può sospendersi in una coda, utilizzando la procedura di wait(). A questo punto il monitor viene rilasciato e viene dato l'accesso ad un altro processo che può modificare le variabili condivise e probabilmente rimuovere la causa che ha bloccato l'esecuzione del precedente processo, risvegliandolo tramite la procedura di signal().

Java mette a disposizione una versione propria di monitor. Attraverso l'uso della parola chiave synchronized, da aggiungere al prototipo di un metodo, si rende un metodo procedura di monitor, quindi eseguita in mutua esclusione. Tutti i metodi che non presentano la precedente parola chiave nella loro firma, possono esser eseguiti contemporaneamente da più thread.

```
public synchronized void deposita() {}
```

Il vincolo di mutua esclusione è imposto assegnando ad ogni OGGETTO un lock che un thread che richiama un metodo sincronizzato cerca di acquisire. Se il lock dell'oggetto è disponibile, il thread ne prenderà possesso e potrà lavorare con le risorse sensibili, altrimenti dovrà mettersi in attesa che il processo che lo detiene lo rilasci. Quando il processo riesce a conquistare il lock, come detto in precedenza non è detto che possa procedere con l'esecuzione della procedura, questo perché potrebbe trovarsi bloccato da determinati stati delle variabili locali. Quindi dovrà eseguire dei controlli tramite

delle strutture quali if o cicli while, per verificare la possibilità di procedere nell'esecuzione del codice. Nel caso in cui queste condizioni non si verifichino dovrà rilasciare il lock e mettersi in attesa sfruttando la primitiva di sospensione wait(). il richiamo di questo metodo fa si che il thread passi dallo stato ready a quello di waiting. Rilasciando il lock si dà la possibilità ad un altro processo bloccato sulla chiamata dei metodi sincronizzati, di accedere alle risorse e di effettuare un eventuale modifica di variabili da cui può dipendere l'attesa di altri thread. Quando si crede che la modifica dello stato di alcune variabili sia importante per alcuni processi bloccati, deve corrispondere una chiamata di risveglio notify(). Questa chiamata ha il compito di risvegliare uno scelto a caso tra i processi bloccati che avevano precedentemente eseguito il wait(). Esiste anche la versione notifyAll() che a differenza del notify(), risveglia tutti i thread bloccati in wait. I thread risvegliati dovranno controllare nuovamente la condizione sulla quale si erano precedentemente bloccati e in caso non sia cambiato niente dovranno sospendersi nuovamente chiamando il wait(). Per questo motivo il codice di controllo di queste condizioni si realizza nel seguente modo:

```
while(!condizione_attesa) wait();
```

In questo modo ad ogni risveglio controlla la condizione, e solo in caso affermativo, continua con l'esecuzione della procedura. Una cosa molto importante da sottolineare è che al risveglio il processo non possiede il lock, quindi dovrà competere nuovamente, senza particolari privilegi, per riacquistarlo.

Il classico esempio di sincronizzazione è quello del Produttore-Consumatore. Questo problema non è altro che una generalizzazione di tutti quei casi in cui c'è un processo che ha il compito di generare dati e uno che invece li deve consumare. Il produttore dovrà generare informazioni finché ha a disposizione dei contenitori vuoti per immagazzinarle, mentre il compito del consumatore è quello di prelevare l'informazione e di rendere nuovamente disponibile il contenitore al produttore.

```
public class ProduttoreConsumatore {
 private boolean infdip = false;
 private boolean contdisp = true;
 private int contenitore;

 public synchronized void deposita(int inf) {
  while(!contdisp) {
   try{
    wait();
   }catch(InterruptedException e){}
  }
  contdisp = false;
  infdisp = true;
  contenitore = inf;
  notify();
 }

 public synchronized int preleva() {
  while(!infdisp) {
  try{
```

```
      wait();
    }catch(InterruptedException e){}
   }
   infdisp = true;
   contdisp = true;
   int inf = contenitore;
   notify();
   return inf;
  }
}
```

Leggendo il codice precedente si può notare la presenza della parola chiave synchronized sui due metodi preleva e deposita; questo fa sì che i metodi chiamati da una stessa istanza della classe, vengano eseguiti in mutua esclusione.

All'interno del metodo deposita() viene controllato se è disponibile il contenitore, in caso negativo il thread si mette in attesa finché il processo consumatore non liberi il contenitore e non lo risvegli.

Il processo che chiama il metodo preleva, invece, dovrà mettersi in attesa se non è disponibile nessuna informazione. Sarà compito del produttore risvegliare il consumatore non appena produce l'informazione.

4.2 Differenze tra una programmazione lineare e multi-threading

La classica programmazione strutturata, come vedremo nell'esempio che è stato realizzato in laboratorio, non permette di sfruttare appieno le potenzialità del robot NXT. Questo perché crea un flusso unico di istruzioni che vengono eseguite in sequenza, non permettendo l'esecuzione 'simultanea' di più procedure. Come si può immaginare questo è un grandissimo limite, soprattutto per la programmazione di robot che hanno invece il compito di eseguire più task contemporaneamente, elaborando dati che provengo da diversi sensori.

4.2.1 Programmazione sequenziale

L'esperimento proposto è quello di far seguire all'NXT una linea nera su uno sfondo bianco, utilizzando il sensore di luce. Per andare a mostrare i limiti di un programma che si basa su un unico flusso, verrano collegati altri due sensori; quello a ultrasuoni per controllare che non ci siano ostacoli lungo il percorso e il sensore di suono per controllare se ci sono rumori più alti di una certa soglia in decibel prefissata. In caso ci sia un ostacolo lungo il percorso o si presenti un rumore più alto della soglia, il robot si deve fermare e il programma terminerà.

Prima di tutto dobbiamo importare le classi che ci permettono di istanziare gli oggetti che controllano i sensori, tramite il package lejos.nxt.*.

In questa versione del programma possiamo scrivere tutto il codice all'interno del metodo main(), senza dover definire altre classi al di fuori della principale. La primissima cosa da fare è creare gli oggetti che gestiscono i sensori, inizializzandoli.

```
LightSensor light = new LightSensor(SensorPort.S3);
SoundSensor sound = new SoundSensor(SensorPort.S2);
UltrasonicSensor ultrasonic = new UltrasonicSensor(SensorPort.S1);
```

I sensori devono ovviamante esser collegati ad una porta nota e per far questo si passa al costruttore della rispettiva classe il numero della porta al quale è collegato. La classe SensorPort definisce quattro oggetti statici, accessibili attraverso il nome della classe:

- SensorPort.S1
- SensorPort.S2
- SensorPort.S3
- SensorPort.S4

Dopo aver creato gli oggetti è importante impostare il sensore di suono in modalità DBA, in questo modo si adatta la sensibilità del sensore a quella dell'udito umano e il sensore di luce come sensore di riflessione.

```
sound.setDBA(false);
light.setFloodlight(true);
```

I valori ritornati dal sensore di luce ovviamente non son precisissimi, quindi dovremmo risolvere considerando dei valori entro certi range. Si deve quindi impostare una soglia oltre la quale si passa dal colore nero al bianco.

```
final int blackWhiteThreshold = 45;
```

Tramite prove effettuate con il sensore nelle condizioni ambientali che avevamo, si è constatato che i valori rappresentanti il nero si aggiravano attorno al 35, mentre per il bianco 55. Un'ottima soluzione per separare i due colori è prendere la media dei valori ed impostarla come soglia. In questo modo si riesce a far capire al robot se si trova sopra la linea nera o se l'ha persa.

Il codice di controllo che permette al NXT di seguire la linea è inserito all'interno di un ciclo. La condizione d'uscita da questo loop è una condizione multipla, cioè dipende dalla pressione del tasto escape, dal verificarsi di un rumore superiore ai 50 dB o dalla presenza di un ostacolo ad una distanza inferiore ai 25 cm. Da qui si vede il limite di una programmazione sequenziale perché se si dovesse verificare una delle condizioni d'uscita all'interno del ciclo, questa andrebbe persa ed il robot continuerebbe tranquillamente il suo percorso. Solo nel caso del verificarsi di una delle condizioni esattamente nel momento in cui viene fatto il controllo, all'interno del while, verrebbe catturata e causerebbe la terminazione del programma. In qualsiasi altro punto si verifichi sarebbe inutile.

```
while (! Button.ESCAPE.isPressed() && !(sound.readValue() > 50)
&& !(ultrasonic.getDistance() < 25)){
```

Nel caso non si presenti nessuno di questi eventi, il robot dovrà seguire la linea nera o in caso l'abbia persa, dovrà cercarla nelle vicinanze. Qualsiasi valore inferiore ritornato dal sensore di luce minore della soglia viene considerato come nero, quindi NXT dovrà continuare ad andare avanti.

```
if(light.readValue() < blackWhiteThreshold) {
 //imposta la velocità del motore a 360 gradi/s
 Motor.B.setSpeed(360);
 Motor.C.setSpeed(360);
 Motor.B.forward();
 Motor.C.forward();
}
```

In caso il valore letto sia superiore alla soglia, si dovrà cercare la linea nera nelle vicinanze. Dagli esperimenti fatti è risultato che impostando una velocità di 180 gradi/s il robot impiega 16 ms per ruotare di pochi gradi, circa 40. La ricerca dovrà esser fatta alternativamente una volta a destra e una a sinistra, questo per aumentare la probabilità di ritrovare prima la linea nera. Utilizzando questa tecnica di ricerca il tempo dovrà raddoppiare ogni volta per fare in modo che il robot ruotando ritorni alla posizione attuale e in più gli resti tempo per esplorare una nuova zona. Raggiunti gli 8 s non ha più senso incrementare il tempo, visto che a quel punto il robot avrebbe già compiuto un giro completo.

```
direzione = true;
//unità di tempo necessaria a far ruotare il robot di alcuni gradi
time = 16;
//Cerca la linea nera finchè non la trova
while(light.readValue() > blackWhiteThreshold) {
 //Diminuisci la velocità a 180 gradi/s
 Motor.B.setSpeed(180);
 Motor.C.setSpeed(180);
 if(direzione) {
 //se il tempo è inferiore ad 1s cerco la linea nera a destra
 if(time < 1000) Motor.B.stop();
 else Motor.B.backward(); //altrimenti ruoto il robot
 Motor.C.forward();
 }
 else {
 //se il tempo è inferiore ad 1s cerco la linea a sinistra
 if(time < 1000) Motor.C.stop();
 else Motor.C.backward(); //altrimenti ruoto il robot
 Motor.B.forward();
 }
stop = System.currentTimeMillis() + time;
direzione = !direzione; //inverto la direzione
time = time * 2; //il tempo di ricerca viene raddoppiato
//con questo tempo il robot esegue una rotazione completa
if(time > 8000) time = 8000;
//il robot continua a ruotare finchè
//non viene trovata la linea nera o scade il tempo a dispozione
while(light.readValue() > blackWhiteThreshold
&& System.currentTimeMillis() < stop);
```

L'ultimo ciclo while serve per permettere al robot di ruotare per il tempo stabilito, esce o perché si è ritrovata la linea nera o perché è scaduto il tempo di ricerca.

4.2.2 Programmazione Multi-threading

In questa sezione verrà riproposto l'esperimento precedente utilizzando però questa volta il multi-threading, cosi come è stato affrontato nella sezione dei concetti generali. Saranno realizzate tre classi, una per il sensore di suono, una per l'ultrasuoni e l'ultima che dovrà realizzare la funzione segui linea. Ovviamente per migliorare il risultato finale, rispetto la precedente prova, ogni classe dovrà esser associata ad un thread.

La prima classe che vediamo è quella che utilizza il sensore di suono per verificare il livello di rumore. Il metodo run(), eseguito al momento della chiamata start(), contiene un ciclo infinito in modo tale da controllare continuamente il livello di rumore. In caso fosse riscontrato un livello superiore ai 40 dB, il robot verrebbe fermato e il programma terminerebbe.

```
private class Sound extends Thread {
 public void run() {
  SoundSensor sound = new SoundSensor(SensorPort.S2);
 sound.setDBA(true);
 while(true) {
  if(sound.readValue() > 40) {
   Motor.B.stop();
  Motor.C.stop();
  System.exit(1);
  }
 }
 }
}
```

La seconda classe deve utilizzare il sensore ad ultrasuoni per verificare l'eventuale presenza di oggetti che ostacolano il percorso. Anche in questo caso il metodo run() dove contenere un ciclo infinito per verificare in continuo la presenza di oggetti ad una distanza inferiore ai 25 cm.

```
private class Ultra extends Thread {
 public void run() {
  UltrasonicSensor ultrasonic = new UltrasonicSensor(SensorPort.S1);
  while(true) {
   if(ultrasonic.getDistance() < 25) {
    Motor.B.stop();
    Motor.C.stop();
    System.exit(1);
   }
   try
   {
    Thread.sleep(50);
   }catch(InterruptedException ex) {}
 }
 }
}
```

L'ultima classe che rimane da sviluppare è quella che permette al robot di seguire la linea. Il codice all'interno del metodo run() è praticamente lo stesso dell'esercizio della sezione precedente. L'unica differenza sta nell'aggiunta nell'ultimo ciclo while e alla

fine, dell'istruzione Delay.msDelay(1), che richiama al suo interno un Thread.sleep(). Come detto nel capitolo precedente questa istruzione fa passare il thread dallo stato Running a quello di Waiting, dando modo ad altri thread pronti di prendere il controllo della CPU. Senza queste istruzioni il thread che fa seguire al robot la linea, monopolizzerebbe l'uso della CPU, non dando la possibilità agli altri thread di effettuare continuamente i loro controlli.

```
private class Light extends Thread {
 public void run()
 {
 LightSensor light = new LightSensor(SensorPort.S3);
 final int blackWhiteThreshold = 45;
 light.setFloodlight(true);
LCD.drawString("Premi LEFT per iniziare", 0, 2);
 while (! Button.LEFT.isPressed());
LCD.drawString("Premi ESCAPE per uscire", 0, 2);
 boolean direzione = true;
 long time, stop;
 int count = 0;
 while (! Button.ESCAPE.isPressed()){
  if(light.readValue() < blackWhiteThreshold) {
  Motor.B.setSpeed(360);
  Motor.C.setSpeed(360);
  Motor.B.forward();
  Motor.C.forward();
 }
 else {
  direzione = true;
  time = 16;
  while(light.readValue() > blackWhiteThreshold) {
   Motor.B.setSpeed(180);
  Motor.C.setSpeed(180);
  if(direzione) {
   if(time < 1000) Motor.B.stop();
   else Motor.B.backward();
   Motor.C.forward();
  }
  else {
   if(time < 1000) Motor.C.stop();
   else Motor.C.backward();
   Motor.B.forward();
  }
  stop = System.currentTimeMillis() + time;
  direzione = !direzione;
  time = time * 2;
  if(time > 8000) time = 8000;
   while(light.readValue() > blackWhiteThreshold
  && System.currentTimeMillis() < stop) Delay.msDelay(1);
   }
  }
  Delay.msDelay(1);
  }
 }
```

```
}
```

Adesso che abbiamo a disposizione tutte le classi non resta altro che creare gli oggetti e far partire i thread con il metodo start().

```
public SeguiLineaTh() {
 l = new Light();
 s = new Sound();
 u = new Ultra();
 l.start();
 s.start();
 u.start();
}

public static void main(String[] args) {
 LineFollowerTh lf = new LineFollowerTh();
}
```

Caricando sul robot il precedente software e questa nuova versione, si riesce immediatamente ad apprezzare le migliorie apportate dal multi-threading. Ora a differenza del caso con controllo sequenziale dei sensori, il verificarsi di un evento di stop causerebbe l'immediata risposta del robot.

4.3 Programmazione Behavior

La traduzione letterale della parola Bahavior è 'comportamento' e questo rende perfettamente l'idea di quello che si vuole realizzare utilizzando questa tecnica di programmazione. Lo scopo dei behavior è proprio quello di porre rimedio alla confusionaria programmazione strutturata, ovvero sequenze lunghissime di if-then-else (vedi figura 4.2) che rendono il codice pesante e di difficile comprensione da parte di altri, andando ad incapsulare ogni singolo comportamento che il robot deve avere. La suddivisione in comportamenti rende molto più leggibile il codice da parte di altri programmatori, richiedendo però uno sforzo maggiore in fase di pianificazione, questo perché deve esser chiaro fin da subito la suddivisione dei vari behavior. Grazie all'incapsulamento e alla preventiva pianificazione è possibile modificare in un secondo momento il codice, aggiungendo o rimuovendo dei behavior senza compromettere il funzionamento del robot. Per utilizzare questa tecnica si devono realizzare i metodi dell'interfaccia Behavior, importando il package lejos.robotics.subsumption. L'interfaccia è stata realizzata in modo molto generico per riuscire ad adattarsi alle esigenze della maggior parte dei programmatori. Una volta creati i vari Behavior e stabilità la gerarchia di priorità, si devono dare in gestione alla classe Arbitrator, la quale ha il compito di mandare in esecuzione il behavior pronto a priorità più alta. I metodi dell'interfaccia sono i seguenti:

- boolean takeControl() - questo metodo deve ritornare il valore true quando il bahavior diventa attivo. Per esempio quando il sensore di suono torna un valore superiore ad un certo livello di decibel.

- void action() - questo metodo inizia l'esecuzione quando il behavior diventa attivo, cioè quando takeControl() restituisce il valore true. Ad esempio quando il livello di rumore è superiore a 50 dB il robot si ferma.

- void suppress() - il codice di questo metodo deve terminare immediatamente quello del metodo action() e può essere anche usato per aggiornare lo stato di alcuni dati prima che il behavior sia completato.

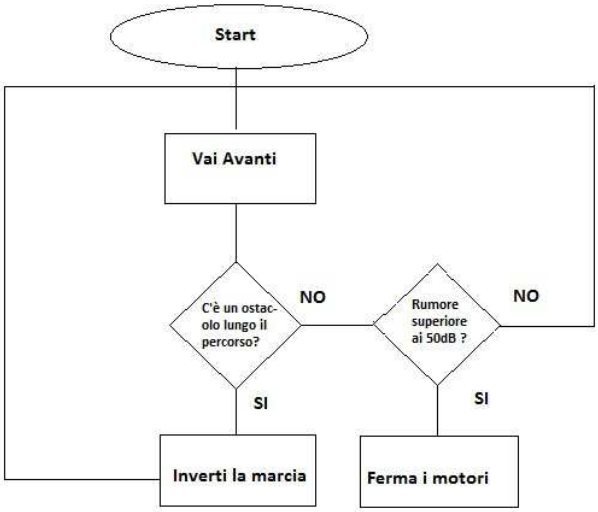

Figura 4.2: Esempio di programmazione strutturata

Questi sono i tre semplicissimi metodi che si devono realizzare per definire un nuovo Behavior. Come vedremmo nell'esperimento successivo il robot dovrà realizzare tre comportamenti diversi e quindi si è costretti a definire tre classi, ognuna delle quali realizza l'interfaccia appena spiegata. Una volta creati i behavior li si devono passare al costrutto di Arbitrator attraverso un array. La classe Arbitrator, che si importa dal package precedente, contiene il costruttore per realizzare il regolatore e un metodo per avviare la funzione di controllo:

- public Arbitrator(Behavior [] behaviors) - crea un oggetto Arbitrator che ha il compito di regolare i behavior che diventano attivi. Richiede come ingresso un array di Behavior e la priorità aumenta con il crescere dell'indice dell'array.

- public void start() - fa partire il sistema di regolazione.

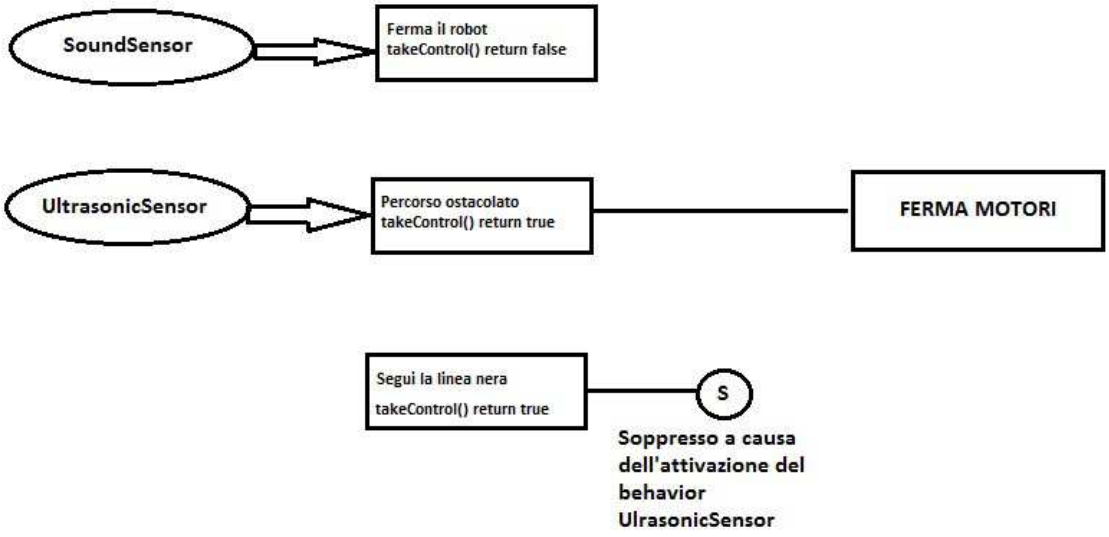

Figura 4.3: Esempio regolazione Behavior

4.3.1 Esperimento di laboratorio utilizzando i Behavior

Ora verrà riproposto l'esperimento visto in precedenza utilizzando però questa volta i behavior. Come potremmo vedere il codice subirà una netta divisione nei tre diversi comportamenti che il robot deve tenere. In questo modo si semplifica molto la lettura del codice andando ad evidenziare i vari comportamenti. Prima di tutto, come detto in precedenza, si devono pianificare i vari behavior, stabilendo l'ordine delle priorità. Il robot ha lo stesso compito dell'esperimento precedente, ovvero dovrà seguire una linea nera disegnata su uno sfondo bianco, finché o il sensore ad ultrasuoni non troverà un ostacolo lungo il percorso o il sensore di suono non misurerà un rumore superiore ad una certa soglia di decibel (si considera un rumore superiore come un comando di stop). Per avere un controllo continuo dei dati tornati dai sensori, dovremmo dare ai Behavior che gestiscono il sensore di suono e quello ad ultrasuoni, una priorità maggiore, riuscendo cosi ad ottenere una risposta tempestiva da parte del robot in caso si verifichi una situazione di stop. Se avessimo messo il behavior che ha il compito di seguire la linea come più prioritario, la risposta ad eventuale segnale di stop non sarebbe immediata, andando a finire in una situazione pericolosa per il robot stesso. Per mostrare le potenzialità di questa tecnica di programmazione si vogliono realizzare tre behavior, uno per far seguire al robot la linea nera, uno per controllare il sensore di suono e l'ultimo per controllare eventuali ostacoli. Prima di tutto bisogna realizzare le tre classi, ognuna delle quali dovrà implementare l'interfaccia Behavior e quindi sviluppare i suoi metodi.

Vediamo ora come realizzare i tre metodi per quanto riguarda il Behavior che gestisce il sensore di suono. Il metodo action() dovrà intervenire nel caso in cui venga riscontrato un valore di suono superiore ai 40 dB, venendo eseguito a seguito di un valore true tornato dal metodo takeControl().

```
public void action() {
 Motor.B.stop();
 Motor.C.stop();
 System.exit(1);
}
```

Il metodo suppress invece in questo caso resterà vuoto.

```
public void suppress() {}
```

Infine il metodo takeControl() avrà il compito di indicare ad Arbitrator, l'attivazioe del behavior.

```
public boolean takeControl() {
 if(sound.readValue() > 40)
  return true;
 return false;
}
```

L'intera classe sarà quindi realizzata come segue:

```
import lejos.robotics.subsumption.*;
public class SoundB implements Behavior {
 //costruisce l'oggetto di tipo
 //SoundSensor e lo inizializza
 public SoundB() {
  sound = new SoundSensor(SensorPort.S2);
  sound.setDBA(true);
 }
 //ritorna il valore true per richiedere
 //l'attivazione ad arbitrator
 public boolean takeControl() {
  if(sound.readValue() > 40)
   return true;
  return false;
 }

 public void suppress() {}
 //ferma i motori e termina il programma se
 //takeControl() ritorna true
 public void action() {
  Motor.B.stop();
  Motor.C.stop();
  System.exit(1);
 }
}
```

Ora che abbiamo capito come realizzare completamente un behavior possiamo passare a realizzare gli ultimi due. Sviluppiamo ora la classe che dovrà controllare i valori ritornati dal sensore ad ultrasuoni e bloccare il robot in caso di un oggetto che possa

ostacolare il cammino del NXT. Il metodo action() e suppress() saranno esattemento uguali al caso precedente, cambia invece il metodo responsabile dell'attivazione del Behavior:

```
public boolean takeControl() {
 if(ultrasonic.getDistance() < 25)
  return true;
 return false;
}
```

In caso il sensore ad ultrasuoni rilevi un ostacolo ad una distanza inferiore ai 25 cm nella direzione del robot, il metodo takeControl() ritorna il valore true, dando la possibilità ad Arbitrator di eseguire il metodo action() della suddetta classe. Il codice completo è il seguente ed anche in questo caso ovviamente si deve implementare l'interfaccia Behavior.

```
public class UltraB implements Behavior {
 //costruisce l'oggetto di tipo UltrasonicSensor
 public UltraB() {
  ultrasonic = new UltrasonicSensor(SensorPort.S1);
 }
 //ritorna il valore true per richiedere
 //l'attivazione ad arbitrator
 public boolean takeControl() {
  if(ultrasonic.getDistance() < 25)
   return true;
  return false;
 }

 public void suppress() {}
 //ferma i motori e termina il programma
 public void action() {
  Motor.B.stop();
  Motor.C.stop();
  System.exit(1);
 }
}
```

Infine l'ultimo Behavior che si deve sviluppare è quello che fa seguire la linea nera al robot sfruttando il sensore di luce.

Il metodo takeControl() in questo caso dovrà tornare sempre true, visto che deve sempre seguire la linea nera ed è compito degli altri Behavior eventualmente bloccarlo.

```
public boolean takeControl() {
 return true;
}
```

In questo caso il metodo suppress() a differenza dei precedenti è di fondamentale importanza, in quanto dovrà modificare lo stato della variabile suppressed nel caso in cui il metodo action() venga interrotto da parte di un behavior più prioritario. Questa soluzione serve a far in modo che una volta impartito l'ordine di stop da un altro behavior, quando questo riprende il controllo, non riattivi il robot.

```
public void suppress() {
 suppressed = true;
}
```

Il metodo action() sostanzialmente ripresenta il codice dell'esempio con il multi-threading puro. La differenza sta nel controllo che è stato aggiunto utilizzando il valore della variabile suppressed. Nel caso in cui il metodo action() sia stato precedentemente interrotto da un behavior a priorità maggiore e in un secondo momento ripreso, il controllo su questa variabile non gli permette di far riprendere al robot la sua attività, ma fa terminare completamente il metodo action() terminado il programma.

Se si rimuove questo controllo, anche in caso di una segnalazione di stop da parte degli altri behavior non sarebbe sufficente a bloccare il robot, in quanto essendo sempre attivo riprenderebbe prima o poi il controllo.

```
public void action() {
 boolean direzione = true;
 long time, stop;
 int count = 0;
 suppressed = false;
 if(light.readValue() < blackWhiteThreshold) {
 Motor.B.setSpeed(360);
 Motor.C.setSpeed(360);
 Motor.B.forward();
 Motor.C.forward();
 }
 else {
 count = 0;
 direzione = true;
 time = 16;
 while(light.readValue() > blackWhiteThreshold
 && !suppressed) {
  Motor.B.setSpeed(180);
  Motor.C.setSpeed(180);
  if(direzione) {
  if(time < 1000) Motor.B.stop();
  else Motor.B.backward();
  Motor.C.forward();
  }
  else {
  if(time < 1000) Motor.C.stop();
  else Motor.C.backward();
  Motor.B.forward();
  }
  stop = System.currentTimeMillis() + time;
  direzione = !direzione;
  time = time * 2;
  if(time > 8000) time = 8000;
  //esce se riteova la linea nera o ha raggiunto il
  //timpo limite di ricerca o è stato
  //precedentemente interrotto, altrimenti mette il
  //thread nella lista Ready tramite la
  //chiamata Thred.yield()
```

```
   while(light.readValue() > blackWhiteThreshold
   && System.currentTimeMillis() < stop && !suppressed)
    Thread.yield();
  }
  }
}
```

L'istruzione Thread.yield() che è stata aggiunta serve per mandare il processo dallo stato Running a quello Ready, per dar modo agli altri processi di effettuare eventuali controlli. La rimozione di questa istruzione non permetterebbe al robot di fermarsi immediatamente quando è alla ricerca della linea nera smarrita.

Ora che abbiamo tutti e tre i behavior definiti non resta altro che creare i relativi oggetti e inserirli in un array di Behavior, tenedo conto che quello con l'indice maggiore avrà priorità più alta.

```
public LFBehavior() {
 //creo tre oggetti Behavior, uno perogni sensore
 Behavior b1 = new LightB();
 Behavior b2 = new SoundB();
 Behavior b3 = new UltraB();
 //il behavior con indice del array più alto ha
 //priorità più alta
 Behavior [] bArray = {b1, b2, b3};
 //inizializzo arbitrator con il vettore di Behavior
 Arbitrator arby = new Arbitrator(bArray);
 //questa chiamata fa partire il controllo dei
 //Behaviors in un ciclo "senza fine"
 arby.start();
}
```

Si crea un oggetto di tipo Arbitrator passandogli al costruttore l'array di Behavior e si fa partire il regolatore tramite la chiamata del metodo start().

Il codice che abbiamo visto rappresenta la classica struttura che deve avere un programma che si prefigge l'obiettivo di sfruttare la programmazione a Behavior. Le due classi che realizzano il comportamento che induce il robot a fermarsi, volendo potevano esser fuse in una unica, modificando il metodo takeControl(), ma è stato volutamente realizzato in questo modo per mostrare come l'arbitrator gestisce più behavior.

Capitolo 5

Bluetooth

Lejos fornisce gli strumenti necessari per instaurare connessioni bluetooth tra NXT e un pc, tra due NXT, tra un NXT e un cellulare e infine tra un NXT e un qualsiasi dispositivo bluetooth remoto. Obiettivo di questo capitolo è sfruttare i metodi messi a disposizione dalla classe Bluetooth per riuscire a connettere due brick. Prima di iniziare a vedere il codice è doveroso effettuare uno studio della classe, come è strutturata, i metodi sviluppati e come riuscire a utilizzare questi strumenti.

5.1 Classe Bluetooth

In questa sezione verranno presentati i soli metodi ritenuti utili per stabilire una connessione e per far comunicare i due dispositivi. La classe Bluetooth è cosi formata:

- BTConnection waitForConnection() - questo metodo mette in attesa il ricevitore (slave). Quando un master si aggancia ritorna un oggetto che rappresenta la connessione.
- BTConnection waitForConnection(int timeout,int mode) - questo metodo a differenza del precedente permette di impostare un timeout massimo di attesa. Inoltre richiede di selezionare la modalità: NXTConnection.RAW, NXTConnection.LCP , NXTConnection.PACKET
- RemoteDevice getKnownDevice(String fName) - restituisce un oggetto rappresentante il dispositivo al quale ci si vuole connettere. fName indica il nome del dispositivo.
- byte[] getDeviceAddr() - restituisce l'indirizzo del dispositivo. Questo valore si usa per connettersi al dispositivo.
- BTConnection connect(BTRemoteDevice remoteDevice) - si connette al dispositivo remoto passato come parametro. Ritorna un oggetto BTConnection o null.

- public static BTConnection connect(String target,int mode,byte[] pin) - si connette al nome o all'indirizzo specificato, richiede di indicare una delle precedenti modalità e il pin utilizzato per questa connessione.
- BTConnection connect(String target,int mode) - come il precedente metodo, solo che non richiede di specificare alcun pin.

5.2 Stabilire una connessione

Il primo passo è creare una nuova connessione bluetooth tra i due dispositivi interessati. Una connessione di questo tipo è sempre formata da due parti: un master, che cerca dei dispositivi in ascolto e uno o più slave che sono in attesa di ricevere delle richieste di connessione. Lo slave si mette in attesa di un master, mentre il master dovrà connettersi al dispositivo scelto in attesa di connessione. Una volta che la connessione è stabilità, entrambi le parti potranno generare un flusso di dati di input/output.

5.2.1 Come creare e gestire un flusso di dati

Quando si è creata una connessione, per trasferire dei dati è necessario aprire un flusso in lettura o in scrittura. Un flusso, detto anche stream, non è altro che un canale virtuale che collega i dispositivi che hanno stabilito la connessione e permette di scrivere e leggere dati. La classe NXTConnection implementa la seguente interfaccia che permette di aprire degli strem:

- InputStream openInputStream() throws IOException;
- OutputStream openOutputStream() throws IOException;
- DataInputStream openDataInputStream() throws IOException;
- DataOutputStream openDataOutputStream() throws IOException;

Aprire uno stream quando si è stabilita una connessione è semplicissimo:

```
DataOutputStream oStream = con.openDataOutputStream();
DataInputStream iStream = con.openDataInputStream();
```

Ovviamente 'con' è un oggetto di tipo NXTConnection. Una volta aperto il flusso non resta altro che andare a leggere o scrivere dei dati. I metodi utilizzabili possedendo uno stream sono:

- int read(byte b[]) throws IOException
- int read(byte b[], int off, int len)throws IOException
- boolean readBoolean() throws IOException
- byte readByte() throws IOException

- short readShort() throws IOException
- readInt() throws IOException
- char readChar() throws IOException
- float readFloat() throws IOException
- String readLine() throws IOException

Per leggere un valore intero sul canale le istruzioni sono le seguenti:

```
int letto;
try {
  letto = iStream.readInt();
} catch (IOException e ) {System.out.println(" Errore: " + e);}
```

Questi metodi che leggono dati da degli stream lanciano delle eccezioni che vanno gestite. Inoltre, una cosa da tenere in considerazione è che tutti questi metodi sono bloccanti, cioè bloccano l'esecuzione del programma finché non finiscono la lettura. Se il programma nel frattempo deve effettuare altre operazioni, queste devono esser scritte in un altro thread.

I metodi che consentono di scrivere su uno stream sono invece:

- void write(byte b[], int off, int len) throws IOException
- void writeBoolean(boolean v) throws IOException
- void writeByte(int v) throws IOException
- void writeShort(int v) throws IOException
- void writeChar(int v) throws IOException
- void writeInt(int v) throws IOException
- void writeFloat(float v) throws IOException;
- void writeChars (String value) throws IOException

Per scrivere un valore intero sul canale le istruzioni sono le seguenti:

```
try {
  oStream.writeInt(7);
} catch (IOException e ) {System.out.println(" Errore: " + e);}
```

5.2.2 Slave

Vediamo ora come mettere un dispositivo in attesa di una connessione (slave). Il codice Java da scrivere è il seguente:

```
import lejos.nxt.*;
import lejos.nxt.comm.*;
import java.io.*;

public class Slave {
  public static void main(String[] args) {
    LCD.drawString("In attesa di una", 0, 0);
    LCD.drawString("connessione BT", 0, 1);
    NXTConnection conn = Bluetooth.waitForConnection();
    DataInputStream iStream = conn.openDataInputStream();
    DataOutputStream oStream = conn.openDataOutputStream();
    int letto = 0;
    try {
      letto = iStream.readInt();
    }catch (IOException e ) {
      System.out.println(" Errore: " + e);
    }
    LCD.drawString("Dato Letto: ", 0, 2);
    LCD.drawInt(letto , 0, 3);
    try {
      oStream.writeInt(-letto);
    }catch (IOException e ) {
      System.out.println(" Errore: " + e);
    }
  }
}
```

Con questo semplice codice si riesce a mettere in attesa un brick di una connessione. Appena un master si connette cerca di leggere un valore intero dal flusso di dati. Questa ultima parte è solo un esempio di come può avvenire uno scambio di dati dopo una connessione.

5.2.3 Master

Per inizializzare una connessione BT tra due NXT, bisogna prima di tutto associare i due dispositivi. Per far questo basta effettuare una ricerca dei dispositivi attivi tramite il menu Lejos. Verificare che il bluetooth sia attivo (Power on) e che sia visibile, effettuare una ricerca dei dispositivi in ascolto e associare il dispositivo slave a quello master. Ora che i due dispositivi sono associati, si può creare un oggetto RemoteDevice e verificare la presenza del nome del dispositivo al quale si vuole connettersi.

```
RemoteDevice r = new Bluetooth.getKnownDevice(name);
if(r == null) {
  LCD.drawString("Nessun dispositivo", 0 , 0);
  LCD.drawString("trovato", 0, 1);
  Button.waitForPress();
  System.exit(1);
```

```
}

BTConnection con = Bluetooth.connect(r);
if(con == null) {
  LCD.drawString("Connesione fallita", 0, 0);
  Button.waitForPress();
  System.exit(1);
}
```

Altrimenti se non si vuole connettersi al dispositivo utilizzando il nome, si può richiamare la funzione getDeviceAddr() e avere una lista dei possibili indirizzi (ritorna un array di byte).

Il codice completo per effettuare un test della connessione bluetooth è il seguente:

```
import lejos.nxt.*;
import lejos.nxt.comm.*;
import java.io.*;
import javax.bluetooth.*;

public class BTTest {
  public static void main(String[] args) {
    String name = "NXT";  //inserire il nome del brick NXT
    LCD.drawString("Connessione" , 0, 0);
    LCD.drawString("in corso..." , 0, 1);
    RemoteDevice r = Bluetooth.getKnownDevice(name);
    if(r == null) {
      LCD.drawString("Nessun dispositivo", 0 , 0);
      LCD.drawString("trovato", 0, 1);
      Button.waitForPress();
      System.exit(1);
    }

    BTConnection con = Bluetooth.connect(r);
    if(con == null) {
      LCD.drawString("Connesione fallita", 0, 0);
      Button.waitForPress();
      System.exit(1);
    }

    LCD.clear();
    LCD.drawString("Connesso!!", 0, 0);
    //ora che la connessione è instaurata possimo
    // provare a scambiare dei dati
    DataOutputStream oStream = con.openDataOutputStream();
    DataInputStream iStream = con.openDataInputStream();
    int scrivi = 7;
    int letto = 0;
    try {
      oStream.writeInt(scrivi);
    }catch (IOException e ) {
      System.out.println(" Errore: " + e);
    }
    try {
      letto = iStream.readInt();
```

```
    }catch (IOException e ) {
      System.out.println(" Errore: " + e);
    }
    LCD.clear();
    LCD.drawString("Ricevuto: ", 0, 0);
    LCD.drawInt(letto , 0, 1);
    try{
      LCD.drawString("Chiusura..", 0, 0);
      iStream.close();
      oStream.close();
    }catch(IOException ex) {
      LCD.drawString("Errore chiusura!", 0, 0);
    }
    LCD.drawString("Finito!", 3, 0);
    Button.waitForPress();
  }
}
```

5.2.4 Come controllare un NXT da un altro usando il bluetooth

Lejos mette a disposizione la classe RemoteNXT che permette di controllare un altro NXT remoto, avente lo stesso firmware Lejos o il firmware originale della Lego, utilizzando la connessione bluetooth e il protocollo LEGO Comunication (LCP). Per collegarsi al NXT remoto si utilizza il costruttore della classe:

```
public RemoteNXT(String name) throws IOException
```

Il nome del dispositivo remoto al quale ci si vuole connettere deve già esser associato, facendo una ricerca dei dispositivi bluetooth dal menu di Lejos. Il costruttore crea una connessione con il dispositivo associato che ha il nome passato come parametro e crea anche le istanze necessarie per gestire i motori e i sensori da remoto. Vengono messi a disposizione anche una serie di metodi per conoscere informazioni di base del modulo al quale ci si è connessi:

- public String getBrickName()
- public String getBluetoothAddress()
- public int getFlashMemory()
- public String getFirmwareVersion()
- public String getProtocolVersion()

Gli oggetti che rappresentano i sensori remoti, quando vengono creati, devono specificare la porta alla quale sono collegati. Le porte per i sensori sono sempre le solite: S1, S2, S3 e S4. Si dovrà però specificare che sono le porte del brick remoto e non quelle locali.

```
try {
  LCD.drawString("Connessione",0,0);
  LCD.drawString("in corso...",0,1);
   nxt = new RemoteNXT("Lidio");
  LCD.clear();
  LCD.drawString("Connesso", 0, 0);
}catch (IOException ex) {
  LCD.clear();
  LCD.drawString("Connessione fallita!", 0, 0);
   Button.waitForPress();
   System.exit(1);
}
LightSensor light = new LightSensor(nxt.S1);
LCD.drawString("Luce: " + light.readValue(), 0, 0);
```

Come si vede dal codice si crea normalmente l'oggetto che rappresenta il sensore e si specifica la porta remota tramite l'oggetto 'RemoteNXT', selezionandola tramite l'istruzione: 'nxt.S1'. Vengono creati in locale anche tre oggetti per controllare i motori remoti (A, B e C).

```
nxt.A.setSpeed(180);
nxt.A.forward();
nxt.B.setSpeed(180);
nxt.B.backward();
```

Con gli strumenti messi a disposizione dalla classe 'RemoteNXT' e con delle dovute modifiche al codice, sarebbe possibile riproporre l'esperienza dei capitoli precedenti utilizzando due NXT, uno che segue il percorso e l'altro che attiva il robot.

Conclusioni

Nel corso di questa tesi sono stati presentati gli argomenti principali ritenuti indispensabili per chi vuole iniziare a programmare il robot Mindstorms NXT in Java, pensando soprattutto agli studenti delle scuole superiori. Proprio per questo motivo si è cercato di semplificare il più possibile gli argomenti trattati, puntando principalmente ad introdurre il complesso meccanismo di gestione della tecnologia multi-threading. Si è ritenuto utile affrontare questi argomenti tramite la descrizione di varie esperienze di laboratorio, realizzate in modo tale da esser ripetibili da parte degli studenti in un secondo momento. Infine è stato introdotto la tecnica di programmazione Behavior, fondamentale per la realizzazione di applicazioni per i robot, anche in questo caso affrontando un esperienza di laboratorio. Obiettivo di questo materiale è quello si esser una valida guida per il corso di robotica tenuto presso l'istituto Pio X e porre soluzione ai principali problemi che possono nascere affrontando argomenti complessi come quelli presentati.

Appendice A

Codice completo: Programmazione sequenziale

Viene riportato qui di seguito l'intero codice Java della prima esperienza di laboratorio.

```
import lejos.nxt.*;
/*
 * Il compito di questa classe è far seguire al robot NXT
 * una linea nera sfruttando il sensore di luce
 * e controllare eventuali ostacoli lungo il percorso tramite
 * l'ultrasuoni. Si può anche comandre lo stop
 * al robot emettendo un suono superiore ai 50dB.
 * In questa versione il controllo dei sensori avviene sequenzialmente
     per dimostrare gli scarsi risultati
 * che si ottengono con questatecnica di programmazione.
 */
public class SeguiLineaUS {

  public static void main (String[] aArg) throws Exception
  {
    LightSensor light = new LightSensor(SensorPort.S3); //Creo un
        nuovo oggetto di tipo LightSensor collegato alla porta S3
    SoundSensor sound = new SoundSensor(SensorPort.S2); //Creo un
        nuovo oggetto di tipo SoundSensor collegato alla porta S2
    UltrasonicSensor ultrasonic = new UltrasonicSensor(SensorPort.S1);
         //Creo un nuovo oggetto di tipo UltrasonicSensor collegato
        alla porta S1
    //setto il sensore di suono in modalità DBA, è una modalità che
        corregge il valore di decibel.
    //adattando la sensibilità del sensore a quella dell'udito umano
    sound.setDBA(false);
    final int blackWhiteThreshold = 45; //valore a metà tra il nero e
        il bianco

    //imposta la modalità che permette al sensore di catturare la
        riflessione
    light.setFloodlight(true);
    LCD.drawString("Light %: ", 0, 0);
```

```
26
    // Visualizza sul dispaly la percentuale di luce, di suono e la
        distanza finchè non viene premuto il tasto LEFT
28  LCD.drawString("Premi LEFT per iniziare", 0, 3);
    while (! Button.LEFT.isPressed()){
30    LCD.drawInt(light.readValue(), 3, 9, 0);
      LCD.drawString("Sound %: " + sound.readValue(), 0, 1);
32    LCD.drawString("Distance (cm): " + ultrasonic.getDistance(), 0,
          2);
      Thread.sleep(500);
34  }
    //Segue la linea finchè non viene premuto il tasto ESCAPE o il
        valore del sensore di sono non è maggiore di 50
36  LCD.drawString("Premi ESCAPE per fermare il robot", 0, 3);
    boolean direzione = true;
38  long time, stop;
    int count = 0;
40  while (! Button.ESCAPE.isPressed() && !(sound.readValue() > 50)){
      //Se il lungo il percorso trova un oggetto ad una distanza
          inferiore ai 25cm ferma il robot
42    if(ultrasonic.getDistance() > 25) {
        if(light.readValue() < blackWhiteThreshold) { //Se il robot si
            trova sulla linea nera procede in avanti
44        //imposta la velocità del motore a 360 gradi/s
          Motor.B.setSpeed(360);
46        Motor.C.setSpeed(360);
          Motor.B.forward();
48        Motor.C.forward();
        }
50      else {
          count = 0;
52        direzione = true;
          time = 16; //unità di tempo necessaria a far ruotare il
              robot di alcuni gradi
54        //Cerca la linea nera finchè non la trova
          while(light.readValue() > blackWhiteThreshold) {
56          //Diminuisci la velocità a 180 gradi/s
            Motor.B.setSpeed(180);
58          Motor.C.setSpeed(180);
            if(direzione) {
60            if(time < 1000) Motor.B.stop(); //se il tempo è
                  inferiore ad 1s cerco la linea nera a destra
              else Motor.B.backward(); //altrimenti ruoto il robot
62            Motor.C.forward();
            }
64          else {
              if(time < 1000) Motor.C.stop(); //se il tempo è
                  inferiore ad 1s cerco la linea a sinistra
66            else Motor.C.backward(); //altrimenti ruoto il robot
              Motor.B.forward();
68          }
            stop = System.currentTimeMillis() + time;
70          direzione = !direzione; //inverto la direzione
```

```
                time = time * 2;  //il tempo di ricerca viene raddoppiato
72              //count++;
                if(time > 8000) time = 8000; //con questo tempo il robot
                    esegue una rotazione completa
74              //il robot continua a ruotare finchè non viene trovata la
                    linea nera o scade il tempo a dispozione
                while(light.readValue() > blackWhiteThreshold && System.
                    currentTimeMillis() < stop);
76            }
            }
78        }
          else{
80          Motor.B.stop();
            Motor.C.stop();
82        }
        }
84
    }
86  }
```

Appendice B

Codice completo: Programmazione Multi-threading

Viene riportato di seguito l'intero codice della seconda esperienza di laboratorio.

```
import lejos.nxt.*;
import lejos.util.*;

/*
 * In questa versione dell'inseguitore di linea vengono creati 3
    Thread per la gestione del robot.
 * Light ha il compito di far seguire la linea nera al NXT, Ultra
    controlla che non ci siano ostacoli
 * lungo il percorso, infine Sound controlla eventuali suoni superiori
     ai 50 dB per fermare il robot.
 * Scopo di questa classe  mostrare come il multi-threading migliori l
    'efficenza del robot a differenza
 * del controllo sequenziale dei sensori fatto in precedenza.
 */
public class SeguiLineaTh {
  Light l;
  Sound s;
  Ultra u;
  //il costruttore della classe ha il compito di crearmi gli oggetti
      che gestiscono i sensori e di lanciare
  //i thread corrispondenti
  public SeguiLineaTh() {
    l = new Light();
    s = new Sound();
    u = new Ultra();
    l.start();
    s.start();
    u.start();
  }

  public static void main(String[] args) {
    //crea un oggetto della classe esterna con il solo scopo di
        permettere al costruttore di lanciare i thread
```

```
28      LineFollowerTh lf = new LineFollowerTh();
    }
30
    //Questa classe permette al robot di seguire la linea nera
32  private class Light extends Thread { //eredità le caratteristiche di
        Thread
      //mtodo run eseguito al momento della chiamata l.start()
34    public void run()
      {
36      //crea un oggetto di tipo LightSensor collegato alla porta S3
        LightSensor light = new LightSensor(SensorPort.S3);
38      final int blackWhiteThreshold = 45; //valore a metà tra il nero
            e il bianco

40      //imposta la modalità che permette al sensore di catturare la
            riflessione
        light.setFloodlight(true);
42      LCD.drawString("Premi LEFT per iniziare", 0, 2);
        while (! Button.LEFT.isPressed());
44      //Segue la linea finchè non viene premuto il bottone ESCAPE
        LCD.drawString("Premi ESCAPE per uscire", 0, 2);
46      boolean direzione = true;
        long time, stop;
48      while (! Button.ESCAPE.isPressed()){
          if(light.readValue() < blackWhiteThreshold) {
50          Motor.B.setSpeed(360);
            Motor.C.setSpeed(360);
52          Motor.B.forward();
            Motor.C.forward();
54        }
          else {
56          direzione = true;
            time = 16;
58          while(light.readValue() > blackWhiteThreshold) {
              Motor.B.setSpeed(180);
60            Motor.C.setSpeed(180);
              if(direzione) {
62              if(time < 1000) Motor.B.stop();
                else Motor.B.backward();
64              Motor.C.forward();
              }
66            else {
                if(time < 1000) Motor.C.stop();
68              else Motor.C.backward();
                Motor.B.forward();
70            }
              stop = System.currentTimeMillis() + time;
72            direzione = !direzione;
              time = time * 2;
74            if(time > 8000) time = 8000;
              //Il compito di msDelay è quello di mandare il thread
                  nella coda waiting per dare la possibilià
```

```
76              //agli altri thread di andare in esecuzione ed effettuare
                    gli opportuni controlli
                while(light.readValue() > blackWhiteThreshold && System.
                    currentTimeMillis() < stop) Delay.msDelay(1);
78            }
            }
80          Delay.msDelay(1);
        }
82    }
  }
84
  //Questa classe rappresenta il controllo che deve effettuare il
      sensore di suono
86  private class Sound extends Thread { //eridità le caratteristiche di
       Thread
    //mtodo run eseguito al momento della chiamata s.start()
88    public void run() {
      SoundSensor sound = new SoundSensor(SensorPort.S2);
90    sound.setDBA(true);
      //si crea un ciclo infinito per controllare continuamente il
          valore del sensore di suono
92    while(true) {
        if(sound.readValue() > 40) { //se > 40 blocca i motori e
            termina il programma
94        Motor.B.stop();
          Motor.C.stop();
96        System.exit(1);
        }
98    }
    }
100 }

102 //Questa classe rappresenta il controllo che deve efettuar il
      sensore ad ultrasuoni
  private class Ultra extends Thread { //eredità le caratteristiche di
       Thread
104   //mtodo run eseguito al momento della chiamata l.start()
    public void run() {
106   UltrasonicSensor ultrasonic = new UltrasonicSensor(SensorPort.S1
          );
      //si crea un ciclo infinito per controllare continuamente il
          valore del sensore ad ultrasuoni
108   while(true) {
        if(ultrasonic.getDistance() < 25) { //se c'è un oggetto più
            vicino di 25 cm blocca i motori e termina il programma
110       Motor.B.stop();
          Motor.C.stop();
112       System.exit(1);
        }
114     try
        {
116       Thread.sleep(50);
        }catch(InterruptedException ex) {}
```

```
118             }
            }
120     }
}
```

Appendice C

Codice completo: Programmazione Behavior

Viene qui riportato il codice completo della terza esperienza di laboratorio.

```
/*
 * In quest'ultima versione dell'inseguitore di linea si sfruttano l'uso di Behavior per
 * realizzare il comportamento dei vari sensori. In questo modo si ottiene lo stesso risultato
 * della versione multi-threading solo che la complessità del codice diminuisce permettendo
 * una più facile comprensione.
 */
import lejos.nxt.*;
import lejos.robotics.subsumption.*;
import lejos.util.*;

public class LFBehavior {
  public LightSensor light;
  public SoundSensor sound;
  public UltrasonicSensor ultrasonic;
  final int blackWhiteThreshold = 45;
  boolean direzione = true;
  boolean _suppressed = false;
  long time, stop;

  public LFBehavior() {
    //creo tre oggetti Behavior, uno perogni sensore
    Behavior b1 = new LightB();
    Behavior b2 = new SoundB();
    Behavior b3 = new UltraB();
    //il behavior con indice del array più alto ha priorità più alta
    Behavior [] bArray = {b1, b2, b3};
    //inizializzo arbitrator con il vettore di Behavior
    Arbitrator arby = new Arbitrator(bArray);
    //questa chiamata fa partire il controllo dei Behaviors in un ciclo "senza fine"
```

```
    arby.start();
31  }

33  //questa clsse implementa l'interfaccia Behavior e ne sviluppa i
      metodi rappresentando il comortamento che deve avere
    public class LightB implements Behavior {
35    public LightB() {
        light = new LightSensor(SensorPort.S3);
37    }
      //ritorna sempre true, questo fa capire ad arbitrator che è sempre
          disponibile ad eseguire il metodo action
39    public boolean takeControl() {
        return true;
41    }
      //quando il metodo action viene interrotto viene eseguito suppres,
          che ha il compito di settare
43    //una variabile per ricordarsi della precedente interruzione
      public void suppress() {
45      _suppressed = true;
      }
47    //relizza il comportamento dell'inseguitore di linea, termina la
        sua esecuzione se era stato precedentemente interrotto
      public void action() {
49      boolean direzione = true;
        long time, stop;
51      int count = 0;
        _suppressed = false;
53      if(light.readValue() < blackWhiteThreshold) {
          Motor.B.setSpeed(360);
55        Motor.C.setSpeed(360);
          Motor.B.forward();
57        Motor.C.forward();
        }
59      else {
          count = 0;
61        direzione = true;
          time = 16;
63        while(light.readValue() > blackWhiteThreshold && !_suppressed)
              {
            Motor.B.setSpeed(180);
65          Motor.C.setSpeed(180);
            if(direzione) {
67            if(time < 1000) Motor.B.stop();
              else Motor.B.backward();
69            Motor.C.forward();
            }
71          else {
              if(time < 1000) Motor.C.stop();
73            else Motor.C.backward();
              Motor.B.forward();
75          }
            stop = System.currentTimeMillis() + time;
77          direzione = !direzione;
```

```
            time = time * 2;
79          if(time > 8000) time = 8000;
            //esce se riteova la linea nera o ha raggiunto il timpo
                limite di ricerca o è stato
81          //precedentemente interrotto, altrimenti mette il thread
                nella lista Ready tramite la
            //chiamata Thred.yield()
83          while(light.readValue() > blackWhiteThreshold && System.
                currentTimeMillis() < stop && !_suppressed) Thread.yield
                (); //Delay.msDelay(1);
          }
85      }
        //Delay.msDelay(1);
87    }
  }
89
  //implementa l'nterfaccia Behavior e sviluppa il comportamento che
      deve avere il sensore di suono
91  public class SoundB implements Behavior {
      //costruisce l'oggetto di tipo SoundSensor e lo inizializza
93    public SoundB() {
        sound = new SoundSensor(SensorPort.S2);
95      sound.setDBA(true);
      }
97    //ritorna il valore true per richiedere l'attivazione ad
          arbitrator
      public boolean takeControl() {
99      if(sound.readValue() > 40)
          return true;
101     return false;
      }
103
      public void suppress() {}
105   //ferma i motori e termina il programma se takeControl() ritorna
          true
      public void action() {
107     Motor.B.stop();
        Motor.C.stop();
109     System.exit(1);
      }
111 }

113 //implenta l'interfaccia Behavior e sviluppa il comportamento che
      deve avere il sensore ad ultrasuoni
    public class UltraB implements Behavior {
115   //costruisce l'oggetto di tipo UltrasonicSensor
      public UltraB() {
117     ultrasonic = new UltrasonicSensor(SensorPort.S1);
      }
119   //ritorna il valore true per richiedere l'attivazione ad
          arbitrator
      public boolean takeControl() {
121     if(ultrasonic.getDistance() < 25)
```

```
          return true;
123       return false;
        }
125
        public void suppress() {
127     }
        //ferma i motori e termina il programma
129     public void action() {
          Motor.B.stop();
131       Motor.C.stop();
          System.exit(1);
133     }
      }
135
      public static void main(String[] args) {
137     LFBehavior lfb = new LFBehavior();
      }
139 }
```

Elenco delle figure