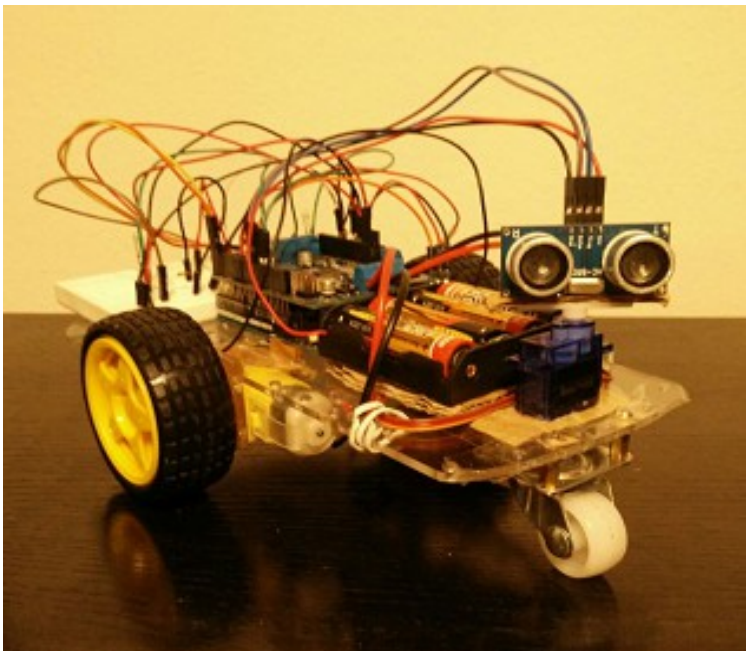
Relazione progetto Programmazione di Sistemi Embedded

Professore: Alessandro Ricci



A.A. 2015/2016
CFU: 6

Giannoni Federico 0000693741
Semprini Mattia 0000693115

Indice:

1. Definizione mediante linguaggio naturale:

Lo scopo di questo progetto è quello di realizzare un robot a due ruote in grado di muoversi, individuare ed evitare ostacoli statici sul suo percorso.
Oltre ad avere una logica di navigazione embedded, il robot dovrà anche fornire una funzionalità di pilotaggio manuale; più nello specifico, esso dovrà essere controllabile in remoto mediante un'apposita applicazione.
Per finire, il robot dovrà anche controllare un suo mini-impianto di illuminazione, in grado di azionarsi quando la luce dell'ambiente circostante scende sotto una determinata soglia e di lampeggiare durante la fase di ricalcolo del percorso a seguito dell'individuazione di un ostacolo.

1.1 Precisazioni:

Prima di partire con la scelta delle componenti hardware e con la parte di progettazione, è necessario fornire alcune precisazioni.

Innanzitutto, un obstacle avoiding robot è realizzabile in molti modi diversi. Tenendo in considerazione che si tratta della progettazione del nostro primo embedded system, abbiamo deciso di fare in modo che il comportamento del robot da realizzare non risulti essere troppo complicato. In altre parole, si vuole fare in modo che la sua logica sia rappresentabile mediante delle macchine a stati non troppo intricate.

Tuttavia, per non rendere il tutto eccessivamente banale e noioso, abbiamo deciso di fare in modo che il robot implementasse due logiche di orientamento distinte (benché entrambe relativamente semplici) intercambiabili in real-time.

In questo modo, il nostro robot potrà passare non solo da una modalità di autopilot ad una modalità di controllo manuale, ma anche da una modalità di autopilot “smart” ad una modalità di autopilot “lazy”.

Definiamo ora, in maniera sommaria, il comportamento che il robot dovrà assumere una volta completato, nella sua modalità di autopilot:

Innanzitutto, esso dovrà muoversi in avanti fino a quando non rileverà la presenza di un ostacolo. A questo punto, il robot dovrà fermarsi e “guardarsi intorno”, per trovare un passaggio libero. Se in modalità “smart”, il robot effettuerà uno scan completo del suo campo visivo per scegliere poi la via “più libera” (cioè quella con ostacoli il più lontano possibile). Se in modalità “lazy” invece, il robot si accontenterà della prima strada libera individuata.

2. Elenco componenti hardware:

Illustrate le feature di base che il nostro robot dovrà implementare, è ora possibile compilare una lista delle componenti di cui esso avrà bisogno per raggiungere l'obiettivo.

2.1 Piattaforma hardware:

Per quanto riguarda la piattaforma hardware, si trattava di prendere una decisione tra Arduino e Raspberry Pi.

Benché Raspberry offra un'allettante capacità di calcolo e il supporto di un SO (per non parlare della possibilità di utilizzare Java per la parte software, linguaggio con il quale abbiamo decisamente più confidenza), la nostra scelta è ricaduta su Arduino.

Il motivo è principalmente legato al fatto che i requisiti computazionali del nostro progetto non sono poi così elevati e l'obiettivo è raggiungibile anche senza il supporto di un SO (in sostanza, il trade-off costo maggiore per prestazioni migliori non risultava essere vantaggioso nel nostro caso). Inoltre, visto che la maggior parte del corso è stata incentrata sullo studio di Arduino, ci siamo sentiti più a nostro agio nello scegliere di lavorare su quest'ultimo.

Passiamo ora ad illustrare l'insieme di attuatori e sensori impiegati, partendo da questi ultimi.

2.2 Sensori:

Ovviamente, per realizzare un robot in grado di evitare ostacoli, sarà necessario un sensore in grado di individuarli, ovvero un sensore di prossimità.

Oltre a questo, per fare in modo che il mini-impianto di illuminazione funzioni come richiesto da specifica, abbiamo fatto uso di una foto-resistenza, che registrerà la luminosità dell'area circostante in modo da far sì che, se questa è troppo bassa, vengano attivate le luci.

2.3 Attuatori:

Per quanto riguarda gli attuatori invece, abbiamo fatto ricorso a due DC Motors delegati allo spostamento del robot, un servo sul quale abbiamo posizionato il sensore di prossimità, rendendolo dunque il "collo" del nostro robot (che ne farà uso per "guardarsi intorno") ed alcuni led come fonte di luce per il mini-impianto di illuminazione.

2.4 Componenti aggiuntive:

Per riuscire a controllare i DC Motors ad un più alto livello, abbiamo fatto uso di un motor shield (Adafruit Motorshield v2) e delle sue relative librerie.

3. Progettazione:

Come prima cosa, è stato fondamentale individuare i task di base che compongono il sistema. Basandosi sulle specifiche, siamo giunti alla conclusione di suddividere il comportamento complessivo del robot in quattro task, concorrenti e comunicanti, coordinati tra di loro da un opportuno scheduler cooperativo e un insieme di variabili globali.

3.1 Task:

Come già detto, i task sono quattro; l'elenco sottostante li illustra in ordine da quello con priorità più alta a quello con priorità più bassa:

1. Message parsing task → questo task si occuperà di controllare se ci sono messaggi in arrivo sulla seriale ed, eventualmente, di reagire ad essi;
2. Orientation task → questo task si occuperà di individuare eventuali ostacoli sul percorso del robot e di individuare una strada da prendere per evitarli, pilotando opportunamente il servo motore e il sensore di prossimità;
3. Navigation task → questo task sarà dedicato a far muovere il robot (ovvero a pilotare opportunamente i suoi motori);
4. Light control task → questo task avrà il solo scopo di controllare l'impianto di illuminazione del robot, accendendone le luci quando l'ambiente circostante è scarsamente illuminato e facendole lampeggiare quando il robot ha individuato un ostacolo e sta ricalcolando il percorso.

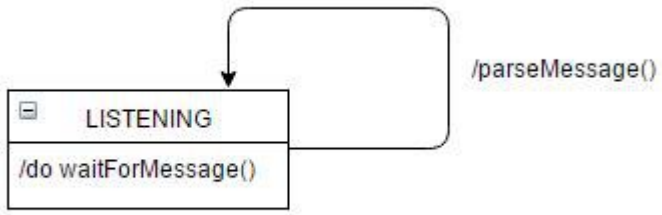
I task di orientamento e di navigazione comunicano tra loro tramite alcune variabili globali: detected, chosenDirection, found, ready.

La variabile detected è settata a true quando un ostacolo è stato individuato; la variabile chosenDirection serve invece a contenere la posizione in cui il servo motore si trovava quando ha individuato la nuova strada da prendere per evitare l'ostacolo; le variabili found e ready servono per coordinare i due task di orientamento e navigazione, in modo che quello di orientamento effettui sempre uno scan, prima che quello di navigazione attivi i motori.

Anche il task di controllo delle luci utilizza (in sola lettura) detected e found per implementare la funzionalità di blinking durante la ricerca di un percorso libero.

Vediamo ora nello specifico le macchine a stati di ciascun task:

3.1.1 Message Parsing Task:



Come possiamo notare, il task di message parsing consiste in una FSM con un unico stato che svolge continuamente l'attività di ascolto sulla seriale.

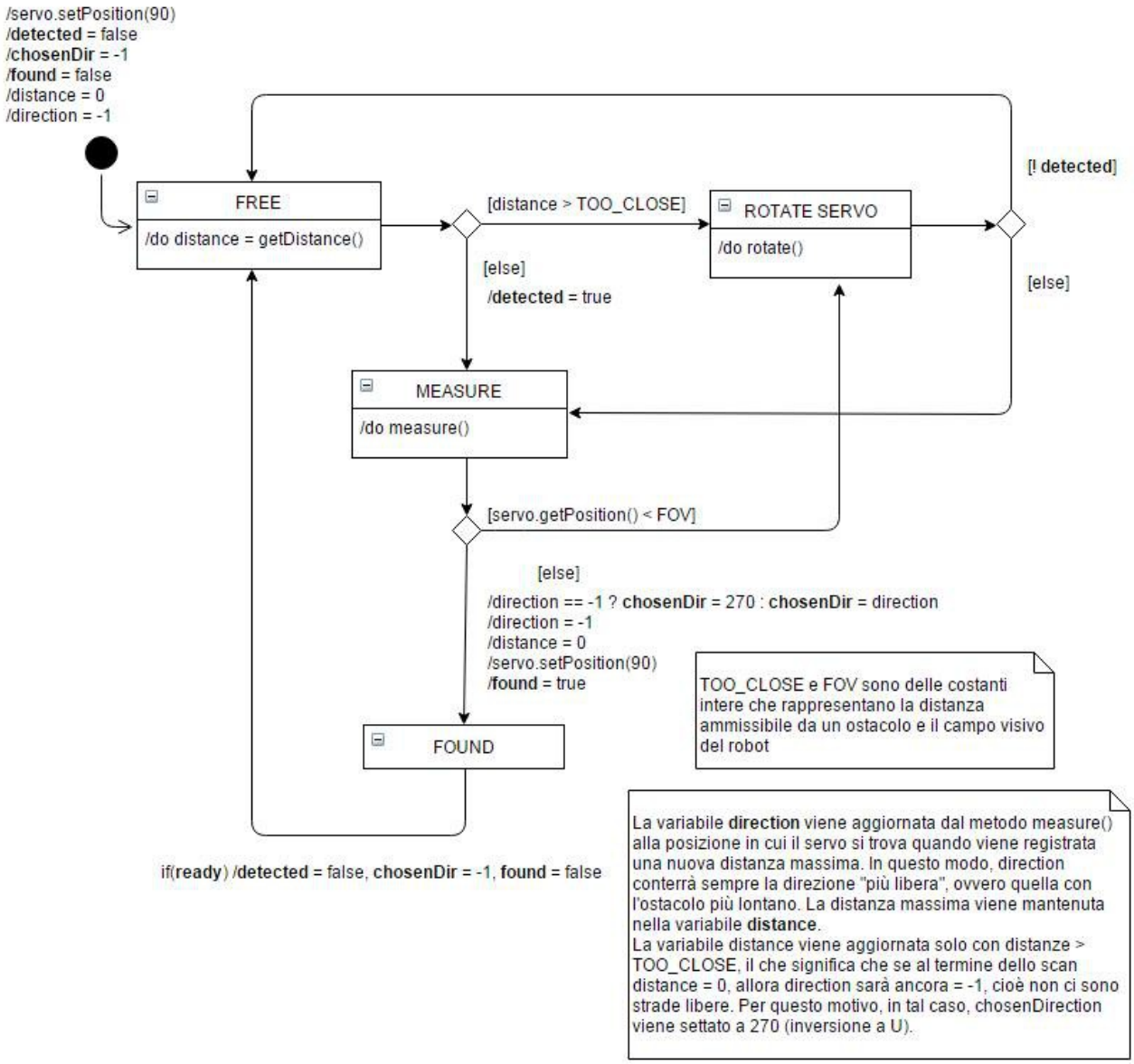
Quando qualcosa è disponibile al suo interno, lo recupera e ne effettua il parsing, reagendo opportunamente al messaggio.

3.1.2 Orientation Task:

Vediamo ora nello specifico il task di orientamento. Ricordiamo che le specifiche richiedono che il robot reagisca all'individuazione di un ostacolo in due maniere diverse, a seconda della logica di cui sta facendo uso in quel determinato momento.

Ciò significa che il task di orientamento avrà due macchine a stati, una per l'orientamento “smart” e una per l'orientamento “lazy”.

Vediamo innanzitutto la FSM relativa alla logica “smart”:



La FSM inizializza al suo avvio le variabili globali detected, found e chosenDirection rispettivamente a false, false e -1 ed entra nello stato FREE.

Lo stato FREE indica che non sono stati individuati ostacoli sul percorso attualmente preso dal robot. Notare che fino a quando non vengono individuati ostacoli, la variabile detected rimane settata a false e la FSM “rimbalza” dallo stato FREE allo stato ROTATE_SERVO, dove esegue una rotazione del servo motore di uno step predefinito. In altre parole, finché non vengono individuati ostacoli, il robot si “guarda intorno” (l'ampiezza del suo campo visivo sarà poi definita da codice).

Una volta individuato un ostacolo in una delle direzioni, la FSM passa allo stato di MEASURE, dopo aver settato a true la variabile globale detected.

La FSM “smart” nello stato di MEASURE esegue uno scan completo del campo visivo del robot, spostando il servo di uno step predefinito dopo ogni misurazione (rimbalzando sullo stato ROTATE_SERVO), fino ad aver scannerizzato l'intero campo visivo.

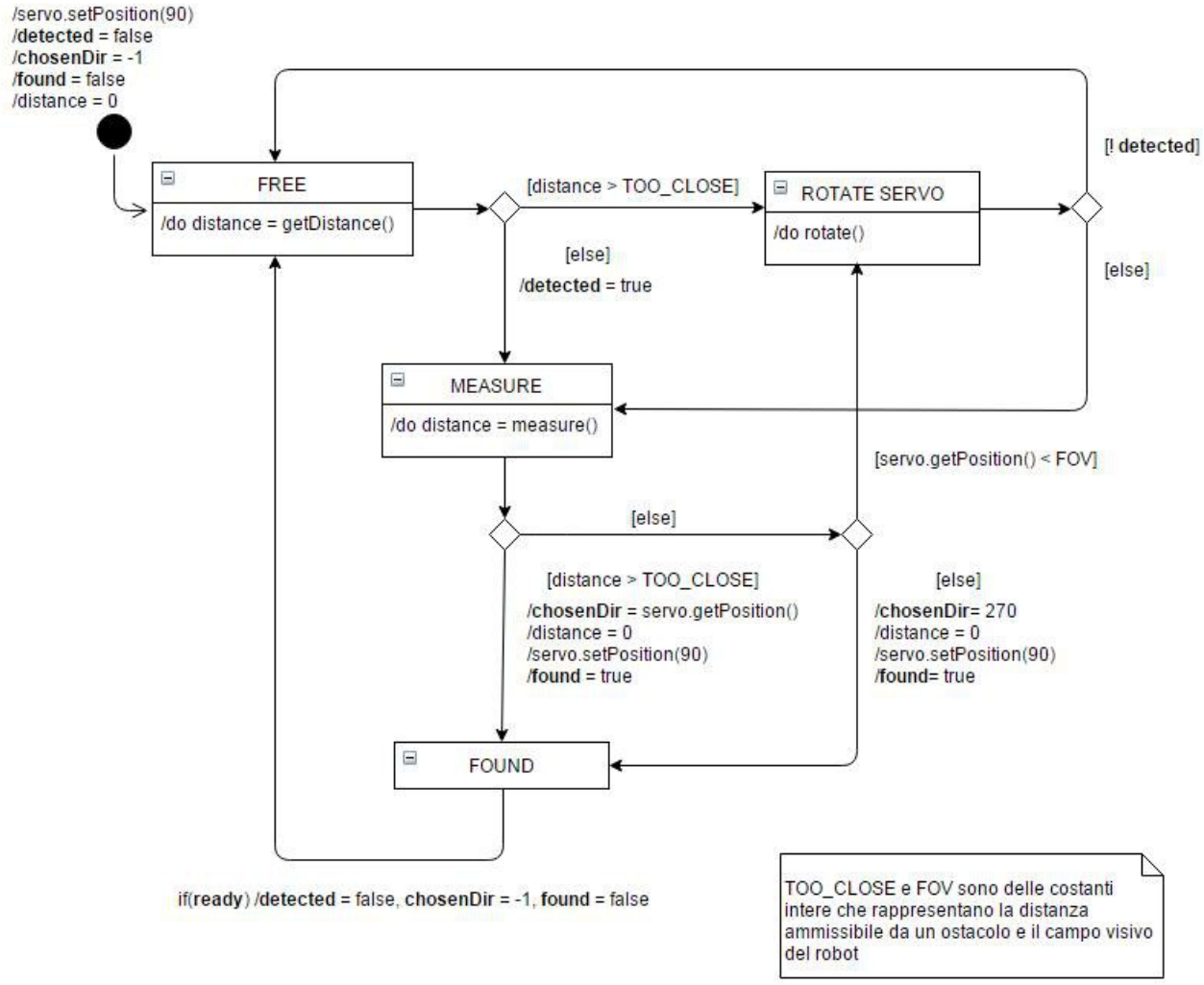
A questo punto, setterà la variabile globale chosenDirection impostando il suo valore a quello della direzione in cui ha rilevato l'ostacolo più lontano. Inoltre, verrà settata a true anche la variabile globale found, dopodiché la FSM passa nello stato di FOUND.

Notare che se in nessuna delle direzioni in cui è stata effettuata una misurazione è presente un cammino libero (cioè ci sono ostacoli troppo vicini in ogni percorso possibile), allora il valore di chosenDirection viene settato a 270.

Una volta nello stato FOUND, la FSM attende venga settata a true la variabile globale ready, prima di tornare nello stato NOT_DETECTED e re-inizializzare le variabili globali detected, found e chosenDirection.

nota: 270 rappresenta un'inversione ad U, questo è dovuto al fatto che per il servo motore, la posizione neutrale è data dal valore 90, il tutto a destra è dato dal valore 0, il tutto a sinistra è dato dal valore 180, quindi di conseguenza, se il servo potesse ruotare più di 180 gradi, la posizione 270 rappresenterebbe la direzione opposta a quella neutrale.

Passiamo ora al task di orientamento “lazy”:



Come possiamo notare, la FSM “lazy” ha un comportamento simile a quella “smart”, tant'è che entrambe presentano gli stessi stati.

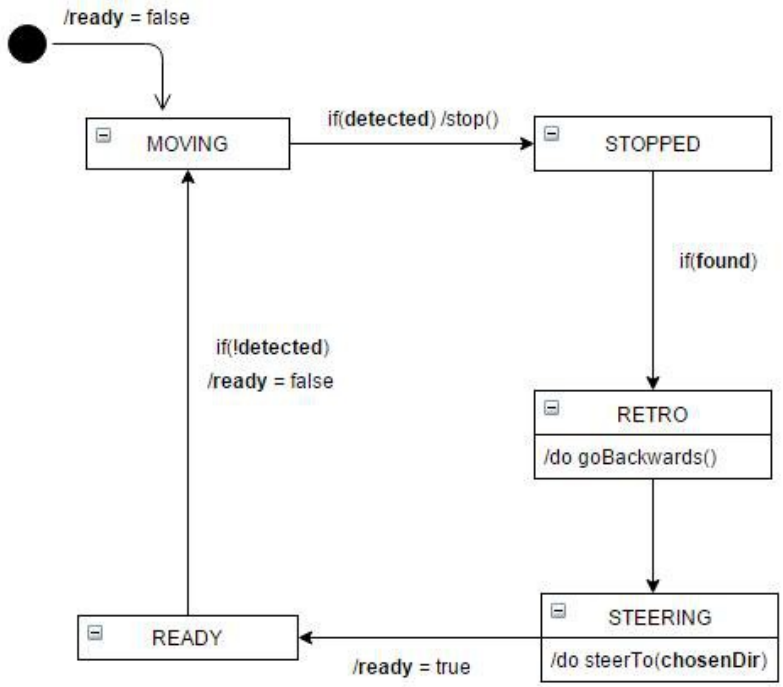
L'unica differenza sta nello stato MEASURE. Infatti, la FSM “lazy”, setterà chosenDirection e found non appena troverà un passaggio con un ostacolo lontano “abbastanza”, invece di valutare l'intero range di possibilità.

Il fatto che entrambe le macchine abbiano esattamente gli stessi stati, è un indicatore sul fatto che la differenza tra le due stia esclusivamente nella logica di valutazione di questi, o, in altre parole, nella loro strategia.

Questo segnale ci ha guidati verso l'utilizzo di uno Strategy pattern per realizzare le due logiche nella maniera più conveniente e verso l'organizzazione dell'intero sistema in un'architettura che permetta di implementare tale pattern in maniera pulita, ma questo verrà illustrato più avanti.

3.1.3 Navigation Task:

Vediamo ora il comportamento della FSM relativa al task di navigazione:



La FSM del task inizializza la variabile globale ready ed entra nel suo stato MOVING, il che significa che il robot, inizialmente, si sta muovendo in avanti.

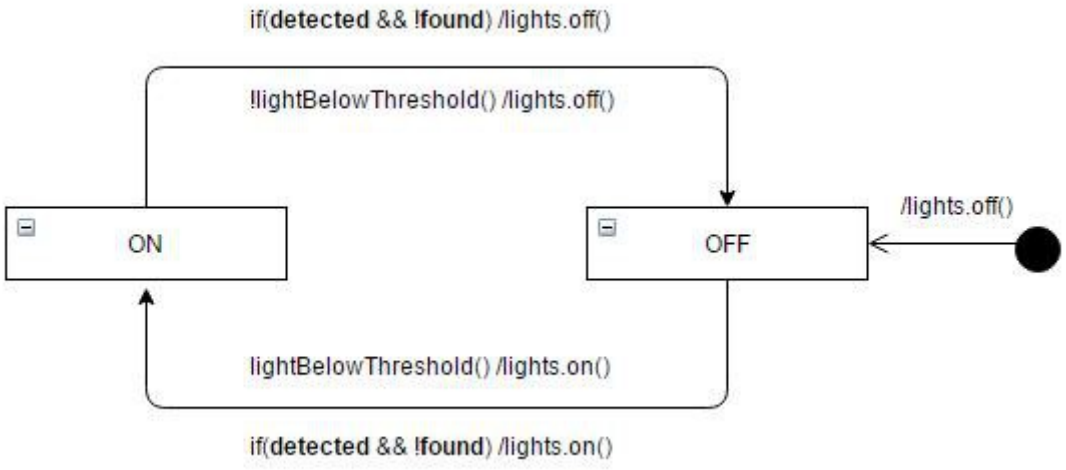
Quando il task di orientamento setta detected a true, ovvero quando viene rilevato un ostacolo, la FSM entra nello stato di STOPPED, fermando i motori, dove rimane fino a quando la fase di scan non termina, ovvero fino a quando il task di orientamento setta a true la variabile found. A questo punto, la FSM entra nello stato di RETRO. Questo stato fa muovere il robot all'indietro ad una velocità predefinita per un tick del task. Questa retromarcia ha il solo scopo di allontanare il robot dall'ostacolo tanto abbastanza da permettergli di girare verso la chosenDirection senza incastrarsi.

Nello stato di RETRO viene calcolato verso dove il robot deve girare e per quanti tick dovrà girare (la rotazione del robot verrà quindi calcolata sulla base di euristiche e sarà strettamente legata al periodo di esecuzione del task), dopodiché, viene attivato il motore opportuno e la FSM entra nello stato di STEERING, dove rimarrà per il numero di tick calcolato precedentemente.

Una volta trascorsi i tick, i motori vengono fermati e il robot entra nello stato di READY, settando a true la variabile ready per comunicare al task di orientamento che il robot si trova ora sulla strada libera.

3.1.4 Light Control Task:

Per finire, vediamo la FSM del task di controllo dell'impianto di illuminazione:



Anche questa macchina a stati, come quella del message parsing task, risulta essere piuttosto semplice. Si passa da uno stato OFF ad uno stato ON nel caso in cui la luce dell'ambiente circostante sia sotto una determinata soglia (con relativa accensione delle luci) e viceversa, da ON a OFF se la luce è sopra tale soglia (con relativo spegnimento delle luci).

Infine si “rimbalza” da ON a OFF e viceversa fino a quando è settata la variabile detected e non è settata la variabile found, cioè quando il robot è fermo dopo aver rilevato un ostacolo e sta effettuando uno scan dei vari percorsi disponibili, per trovare quello da prendere.

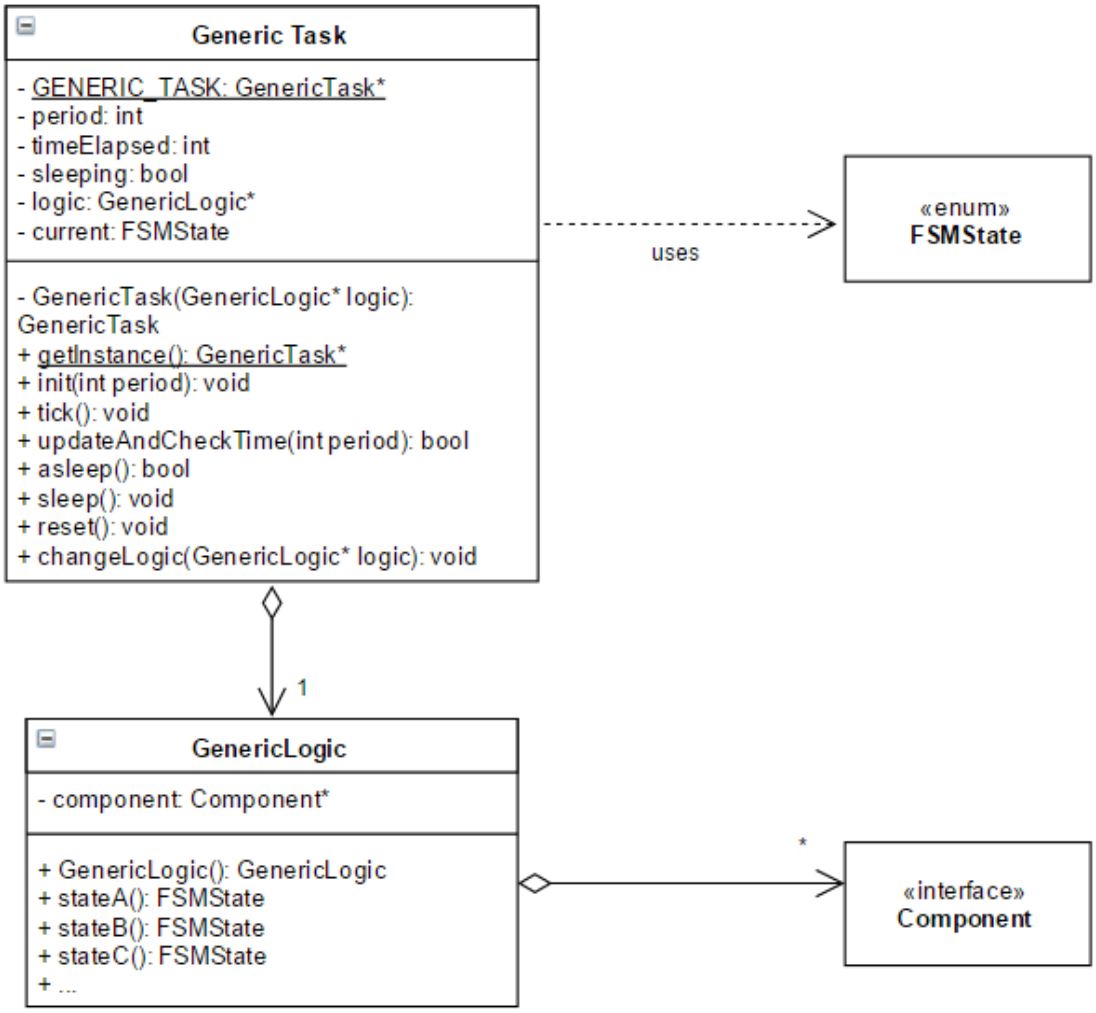
3.2 Architettura:

Abbiamo deciso di organizzare il lavoro in maniera modulare, costruendo un modulo per ogni task e separando il tutto in modo che ogni modulo sia costruibile e testabile singolarmente, così da semplificare il debugging, che risulta essere alquanto complesso su una piattaforma come Arduino.

Abbiamo deciso di costruire una classe per ogni task. Questa classe definisce gli stati che la FSM relativa al task può assumere.
La valutazione ed eventuale transizione di stato verrà però delegata ad un oggetto “logica” specifico per quel task.
Questo oggetto avrà un metodo per ogni stato della FSM che si occuperà di valutare tale stato manipolando le opportune componenti e restituendo lo stato in cui la FSM verrà a trovarsi dopo tale valutazione.

Sarà quindi la logica ad avere accesso alle componenti e a manipolarle.

Mostriamo qui sotto uno schema esplicativo dell'architettura utilizzata (abbiamo deciso di applicare il Singleton pattern per quanto riguarda i vari task):



Inoltre abbiamo deciso di applicare il Singleton pattern per quello che riguarda il robot, dal momento che deve esisterne una sola istanza.

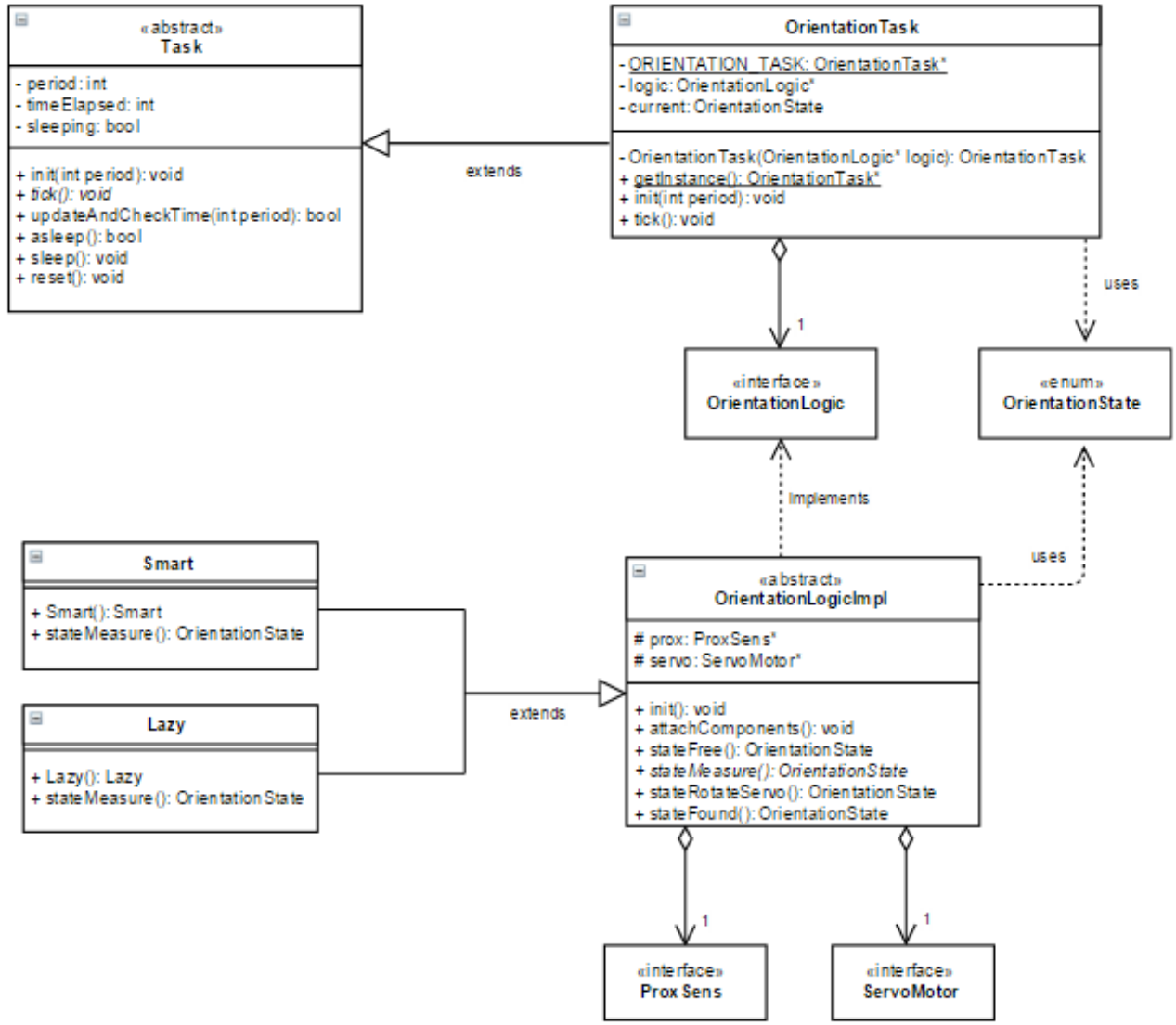
Un'architettura del genere ci ha permesso di costruire e testare singolarmente i vari task ed effettuare poi test di integrazione, unendoli tra di loro.

Inoltre ci ha anche permesso di risolvere il problema di cambio logica in maniera relativamente semplice, facendo uso di uno Strategy pattern.

Più nello specifico, abbiamo implementato una logica astratta per il task di orientamento, che definisce le implementazioni di tutte le valutazioni di stato in comune tra le FSM “lazy” e “smart”, ma ha un metodo astratto per la valutazione dello stato MEASURE, unico stato nel quale le FSM presentano differenze di logica.

A questo punto è stato sufficiente costruire due sotto-classi logica che implementino opportunamente il metodo di valutazione dello stato MEASURE per raggiungere l'obiettivo in maniera rapida e senza duplicare codice.

Qui sotto abbiamo un diagramma che mostra uno dei moduli nello specifico (quello di orientamento) e l'utilizzo dello Strategy pattern:



Notare come disaccorpare le logiche di valutazione di stato dai task ci permetta di utilizzare un solo task per il modulo relativo all'orientamento del robot.

Questo è dovuto al fatto che ogni task, all'interno del sistema, definisce solo gli stati di una FSM e sia la FSM di orientamento “smart” che quella “lazy” hanno esattamente gli stessi stati.

3.3 Organizzazione del cambio di logica in real-time:

Dopo aver definito l'architettura del sistema, abbiamo cominciato a pensare a quali sarebbero stati i principali problemi implementativi, in modo da organizzarci anticipatamente.

Il problema progettuale principale che ci siamo ritrovati ad affrontare riguardava l'organizzazione del cambio di logica in real-time.

Strutturando il lavoro come descritto nel paragrafo precedente, saremmo riusciti ad implementare le due logiche in maniera efficace, ma rimaneva comunque il problema di come effettuarne il cambio in real-time.

Abbiamo deciso innanzitutto di rendere il cambio di logica effettuabile solo tramite la ricezione di opportuni messaggi su seriale, ossia inviando specifici messaggi al robot tramite l'applicazione.

A questo punto, abbiamo individuato tutti i possibili casi di cambio di logica e abbiamo stilato una lista di passi relativi alla loro gestione.

I casi individuati sono tre:

1. caso di cambio da logica autopilot (qualsiasi) a manuale;
2. caso di cambio da logica manuale a autopilot (qualsiasi);
3. caso di cambio da logica autopilot (smart o lazy) all'altra logica auto;

Vediamo ora gli step per ciascun caso:

caso 1)

1. reset del robot (spegni i motori, le luci e porta il servo in posizione neutrale);
2. mette a dormire tutti i task fatta eccezione per quello di message parsing;

Abbiamo immaginato un task dormiente come un task che viene ignorato dallo scheduler e non esegue il proprio tick. In questo modo, quando si passa alla logica manuale, l'unico task ad essere eseguito è quello di message parsing, così che il robot si comporti a tutti gli effetti come uno slave.
Lo stato dormiente in un task, è stato manipolato non come un vero e proprio stato, ma come un semplice campo flag (questo per motivi di semplicità e di riutilizzo del codice).

caso 2)

1. reset del robot;
2. sveglia tutti i task dormienti;

Il risveglio dei task porterà anche ad una loro re-inizializzazione, in modo da resettare le variabili globali necessarie alla comunicazione tra task.

caso 3)

1. reset del robot;
2. cancellazione della vecchia logica di orientamento e attach di quella nuova al task;
3. reset dei task di navigazione e orientamento in modo che vengano resettate le variabili globali necessarie alla loro comunicazione;

A questo punto è ovvio che il task che si occuperà di effettuare il cambio di logica (cioè quello di message parsing), avrà bisogno di poter accedere agli altri task, per poterli resettare, mettere a dormire o svegliare.

Per questo motivo, abbiamo deciso di implementare prima i task necessari alla funzionalità di autopilot e solo successivamente il task di message parsing.

4. Dettagli implementativi:

In questa sezione illustreremo alcuni dettagli implementativi degni di nota.

Partiamo con l'illustrare l'implementazione della classe robot e del Singleton pattern.

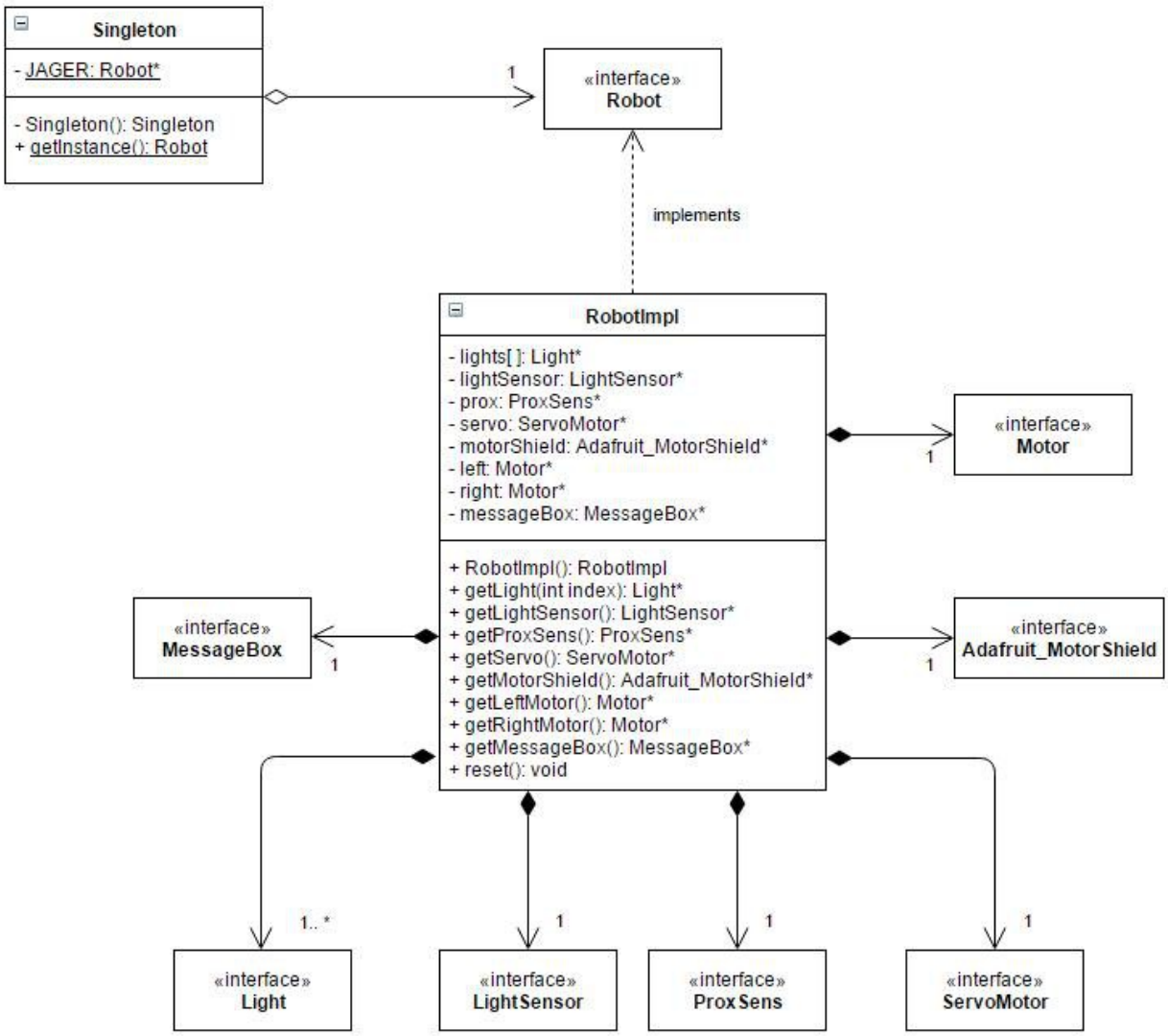
4.1 Robot e Singleton pattern:

Abbiamo deciso di fare in modo che il robot si “auto-costruisca”, ossia presenti un costruttore senza parametri che al suo interno istanzi già tutte le sue componenti.

In questo modo e grazie all'utilizzo del Singleton pattern, è stato possibile semplificare anche l'istanziamento delle logiche dei task, rendendo anch'esse in grado di auto-costruirsi mediante costruttore senza parametri recuperando il robot (e quindi le componenti che devono manipolare) tramite “getInstance()”.

Questo ha permesso una più facile gestione dei cambi di logica in real time, che richiedono distruzione e re-istanziamento di tali logiche.

Qua sotto è presente un diagramma delle classi relativo alla classe robot e all'utilizzo del Singleton pattern.



4.2 Scambio di messaggi:

Per quanto riguarda lo scambio di messaggi, abbiamo fatto uso di una classe Message, con un contenuto di tipo Content. Nel nostro caso, abbiamo reso Content un unsigned char tramite typedef (in questo modo se si vuole modificare il tipo del contenuto dei messaggi, è immediato farlo).

Abbiamo scelto un unsigned char perchè vista la limitata disponibilità di RAM offerta da Arduino, è stato necessario utilizzare solo il minimo quantitativo indispensabile di risorse per mappare tutti i possibili comandi. Benché lavorare con delle stringhe avrebbe dato la possibilità di organizzare meglio i vari messaggi, viste le risorse limitate abbiamo preferito non rischiare ed utilizzare invece un singolo byte (unsigned char) per codificare i vari comandi decifrabili dal robot.

Sulla seriale vengono inviati/ricevuti i contenuti dei vari messaggi.
L'utilizzo di un unsigned char risulta essere vantaggioso anche in questo caso, perché la funzione messa disposizione da Arduino.h, Serial.read(), legge esattamente un byte dalla porta seriale ed un unsigned char è esattamente un byte (non c'è bisogno di ciclare o di stabilire un terminatore di messaggio).

Il protocollo di comunicazione utilizzato è quindi un semplice invio di byte.

Per quanto riguarda la classe MessageBox invece, essa è una componente del robot in grado di accedere alla seriale e wrappare ciò che si trova in essa in un opportuno Message, in essa mantenuto.

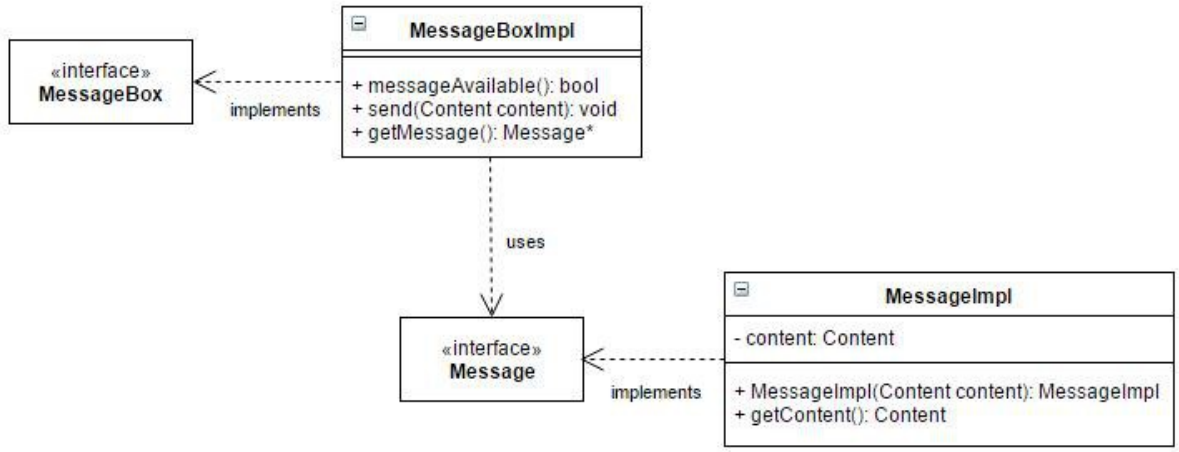
Quando il task message parsing richiede di parsare un messaggio, la MessageBox glielo fornisce (o meglio lo fornisce alla sua logica, che ha accesso al robot e alle sue componenti), se disponibile.

Nella versione corrente, la MessageBox può mantenere solo un messaggio al suo interno. Sarebbe possibile modificarla senza troppi sforzi per fare in modo che sia in grado di mantenere più messaggi, ma ciò sarebbe inutile per come il robot funziona al momento.

Infatti, il task di message parsing effettua ad ogni tick il controllo sulla seriale tramite la MessageBox e, se è presente un messaggio, ne esegue il parsing subito dopo (quindi la MessageBox dovrà conservare al massimo un messaggio prima che questo venga parsato e quindi distrutto).

Tuttavia, se si volesse fare in modo ad esempio, che il controllo sulla seriale, il wrapping del messaggio e il suo salvataggio nella MessageBox avvengano ogni volta che qualcosa giunge sulla seriale tramite interrupt, sarebbe necessario bufferizzare e fare in modo che la MessageBox sia in grado di conservare più messaggi (cosa che, come già detto, non risulterebbe difficile da fare per come il codice è stato organizzato).

Qui sotto riportiamo un diagramma delle classi relativo a ciò che riguarda lo scambio di messaggi.



4.3 Campo visivo del robot:

Abbiamo costruito il codice in modo che il comportamento del robot possa cambiare ulteriormente sostituendo il valore di alcune costanti.

Queste costanti sono definite nella classe AbstractOrientationLogic e sono SCAN_FOV, SCAN_STEP, PROBING_FOV.

La prima costante rappresenta il campo visivo del robot durante la fase di scan e non può essere maggiore di 180 (dal momento che il servo motore ha un range massimo di 180 gradi).

La seconda rappresenta invece lo step di rotazione durante la fase di scan per la ricerca di un percorso libero dopo che è stato individuato un ostacolo e serve a definire quanti possibili percorsi il robot valuterà (la costante dovrà essere un divisore di SCAN_FOV). In questo modo, ogni volta che incontrerà un ostacolo, il robot effettuerà (SCAN_FOV / SCAN_STEP) + 1 registrazioni, ciascuna su una direzione diversa. Ad es. se definiamo uno SCAN_FOV di 90, questo significa che il robot, durante la fase di scan, avrà un campo visivo frontale di 90 gradi. Definendo ora uno SCAN_STEP, ad esempio, di 45 gradi, questo significa che dopo aver effettuato una misurazione, il servo ruoterà di 45 gradi prima di effettuare la successiva. In altre parole avremo una registrazione sulla posizione 45 del servo, una sulla posizione 90 del servo ed una sulla posizione 135 del servo (90 gradi di campo visivo frontale, visto che la posizione 90 rappresenta la posizione frontale).

L'ultima costante infine, rappresenta il campo visivo del robot durante la fase di “probing”, cioè durante la fase di individuazione di ostacoli. La fase di probing avrà sempre tre step per garantire un funzionamento ottimale del robot.

4.4 Librerie utilizzate:

Nella fase di implementazione abbiamo fatto uso di due librerie diverse: AdafruitMotorShield v2 library, per pilotare il motor shield utilizzato e ServoTimer2, per pilotare il servo (spiegheremo poi il perché dell'utilizzo di ServoTimer2 anziché della tradizionale Servo library inclusa con Arduino IDE).

Per quanto riguarda la prima libreria (AdafruitMotorShield v2), abbiamo costruito una wrapper class che wrappa un oggetto di tipo Adafruit_DCMotor e che offre dei metodi per manipolare i motori DC in maniera più comoda e ad un più alto livello.

Anche per quel che riguarda ServoTimer2 abbiamo costruito una wrapper class per manipolare il servo motore ad un livello più alto.

5. Testing:

Come già menzionato, l'approccio progettuale modulare ci ha permesso di costruire e successivamente testare i vari moduli (task, logiche e componenti del robot) singolarmente ed effettuare poi dei test di integrazione, unendo i vari moduli tra di loro, fino a comporre e testare l'intero robot.

Specifichiamo anche che il testing del task di message parsing è stato effettuato prima della realizzazione dell'applicazione tramite serial monitor e, successivamente, tramite un'altra applicazione chiamata Bluetooth Terminal, che permette di connettersi ad un dispositivo bluetooth accoppiato col telefono ed inviargli (e anche ricevere) messaggi testuali.

Il poter ricevere i messaggi testuali che il robot stampava sulla seriale è risultato essere comodo per debuggare, visto che col modulo bluetooth attivo non era possibile farlo tramite il serial monitor messo a disposizione da Arduino IDE.

5.1 Problemi riscontrati:

Nei test individuali fatti su ogni modulo, non abbiamo riscontrato problemi particolari, fatta eccezione per quanto riguarda il modulo di orientamento.

I problemi riguardavano principalmente la componentistica che il modulo ingloba, ossia il servo motore e il sensore di prossimità.

5.1.1 Problemi col sensore di prossimità:

Per quanto riguarda il sensore di prossimità, vista la sua elevata imprecisione, capitava spesso che gli impulsi mandati dal trigger, venissero persi e il sensore registrasse quindi come distanza, la distanza massima (una volta andato in timeout il pulseIn).

Questo ovviamente, avrebbe portato ad uno scorretto comportamento del robot ed ha richiesto un intervento. Abbiamo innanzitutto, limitato il timeout del pulseIn, dandogli un valore più basso in microsecondi. Questo è stato fatto per fare sì che il task che effettua la misurazione, avesse un WCET accettabile.

In altre parole abbiamo definito una costante di timeout di alcuni ms. Trascorsi questi ms dopo aver lanciato l'impulso, se il sensore non ha ancora registrato una distanza, registra uno 0.

Questo 0 può stare a significare o che il segnale di trigger è andato perduto (misurazione fallita), oppure che in quel periodo non ha fatto in tempo a tornare indietro ed essere registrato.

Questo limita la distanza a cui il nostro sensore “può vedere”, ma evita anche che il tentativo di misurazione occupi un numero di ms eccessivo, che porterebbe al dover tarare i periodi con dei valori alti abbastanza da rendere il tutto meno reattivo.

A questo punto però, sorge un altro problema. Cosa fare delle misurazioni che riportano il valore 0? Queste possono essere o degli errori o delle misurazioni fallite a causa della troppa lontananza dall'ostacolo (ovvero dal fatto che il sensore non riesce a “vedere” l'ostacolo nel tempo specificato).
Per risolvere questo ulteriore problema, abbiamo deciso di definire un'altra costante, che esprime un numero massimo di tentativi di misurazione della distanza in una determinata direzione.

Quando il sensore cerca di misurare la distanza in una determinata direzione, effettuerà un numero prestabilito di tentativi.

Verrà presa come misurazione valida, la prima a non dare 0; tuttavia, se tutti i tentativi di misurazione danno una distanza di 0, allora è probabile che questo sia dovuto non ad un insieme di errori consecutivi, ma al fatto che l'ostacolo è troppo distante per essere registrato dal sensore entro il periodo prestabilito.

Quindi, sapendo che in tale direzione c'è un ostacolo lontano abbastanza da non essere rilevato, verrà presa come distanza dall'ostacolo un valore massimo.

5.1.2 Problemi col servo motore:

Un ulteriore problema riguardava invece il pilotaggio del servo motore.

La libreria Servo.h infatti, fornita assieme ad Arduino IDE e che fornisce le primitive necessarie a pilotare il servo motore presente nel modulo di orientamento, utilizza il timer 1 di Arduino, lo stesso timer utilizzato anche da un altro file: Timer.h (utilizzato dallo scheduler per settare la propria frequenza di interrupt).

Questo ci ha costretti a dover utilizzare una seconda libreria: ServoTimer2.
La libreria tuttavia, presentava delle caratteristiche diverse rispetto a quella classica, prima fra tutte, il fatto che mancasse una primitiva in grado di settare la posizione del servo ad un valore espresso in gradi.

L'unica funzione offerta dalla libreria prendeva come parametro il numero di microsecondi dell'impulso che sarebbe poi stato usato per far muovere il motore.

A questo punto, avendo il bisogno di lavorare con delle posizioni precise (ossia espresse chiaramente in gradi), ci siamo trovati costretti a modificare alcune costanti di ampiezza degli impulsi nella libreria in modo che coincidessero con quelli della libreria Servo.h; dopodiché, abbiamo proseguito implementando anche nella nuova libreria, una funzione che modificasse la posizione del servo, portandolo a quella desiderata, basandoci sull'implementazione della stessa funzione offerta dalla libreria Servo.h.

6. Realizzazione dell'applicazione:

Una volta progettato ed implementato il robot e tutte le sue funzionalità di autopiloting e dopo aver ultimato le funzionalità di message parsing, ci siamo dedicati alla realizzazione di una piccola applicazione per controllarlo in remoto. Come protocollo di comunicazione, abbiamo scelto di utilizzare bluetooth, vista l'elevata difficoltà nell'utilizzare una scheda di rete con Arduino.

L'applicazione è stata realizzata utilizzando Android e, viste le sue limitate funzionalità, abbiamo deciso di impostarla su un'unica activity.

L'activity fa uso di due classi, una delegata alla connessione bluetooth verso il server (ossia il robot) ed una delegata invece alla gestione degli input.

La classe delegata alla connessione bluetooth è in grado di recuperare l'adapter e di selezionare come server un device specificato (tale device dovrà essere paired con il telefono) e connettersi ad esso tramite un opportuno AsyncTask.

La connessione è poi gestita da un ConnectionManager (per il quale è stato usato il pattern Singleton) che si occupa di gestire la socket di connessione verso il server, sia per quanto riguarda le operazioni di scrittura che di lettura.

La classe delegata alla gestione degli input invece, si limita semplicemente a reagire agli input utente e di comunicarli al robot nella maniera opportuna, sfruttando il ConnectionManager.

7. Conclusioni e possibili migliorie:

Nella realizzazione di questo progetto ci siamo impegnati nel cercare di rendere il codice il quanto più chiaro e facilmente modificabile possibile.
Abbiamo anche tentato di evitare di duplicare codice il quanto più possibile.

Ovviamente tutto ciò è stato possibile solo entro determinati limiti, visto che spesso siamo stati costretti a giungere a compromessi viste le scarse capacità di computazione della piattaforma hardware utilizzata.

Basti pensare all'utilizzo di singoli byte per il mapping dei messaggi.
Se avessimo lavorato con una piattaforma più performante avremo potuto utilizzare delle stringhe come contenuto ed utilizzare dei determinati pattern di identificazione dei messaggi (il che avrebbe reso il codice molto più elegante e leggibile e ci avrebbe permesso di realizzare un sistema di comunicazione più complesso).

Oltre a questo avremo potuto implementare delle funzionalità extra, quale ad esempio la modifica del campo visivo in real-time, o l'utilizzo di un giroscopio anziché di euristiche per avere una maggior precisione nella rotazione del robot, o altre cose ancora (ad esempio tutto ciò che riguarda l'applicazione, dal codice alla user experience, dal momento che essa è ancora solo poco più di un prototipo).

In ogni caso, abbiamo cercato di strutturare il codice in modo che tutte queste modifiche ed eventuali altre, siano implementabili senza troppa fatica, o senza dover riprogettare l'intero sistema (ad esempio, per quanto riguarda il cambio del campo visivo in real-time, basterebbe trasformare le costanti descritte nel paragrafo 4.3 in campi, in modo da renderli modificabili a run-time ed aggiungere infine dei messaggi che codifichino la richiesta di modifica del campo visivo).

7.1 Commenti finali:

Nello scrivere questa relazione abbiamo tentato di essere il più esaustivi possibile senza risultare però tediosi nella descrizione del nostro lavoro.

Ci siamo quindi limitati ad illustrare solo in parte l'insieme delle cose che abbiamo realizzato. Suggeriamo per questo, per comprendere a pieno il funzionamento dell'unità e le scelte fatte, di andare a vedere il codice, che è stato opportunamente commentato.