

# Tabella dei contenuti

Analisi del problema e studio dei paper - 11, 12 gennaio 2016 .....	2
Studio dell'hardware e del banco prova - 14 gennaio 2016 .....	3
Studio del collegamento tramite porta seriale-FTDI - 15 gennaio 2016.....	4
Invio di comandi al rotore tramite linea di comando (Putty) - 18 gennaio 2016 .....	4
Studio dell'ambiente Git per lo sviluppo in team, integrazione con Eclipse - 20, 21 gennaio 2016 ..	6
Studio della libreria rxtx e comunicazione seriale - 22 gennaio 2016 .....	10
Prototipo 1: semplice invio comandi seriale tramite Java - 26, 29 gennaio 2016.....	11
Studio della telemetria del motore e classe AutoQuadEsc32 - 1, 2, 4 febbraio 2016.....	14
Prototipo 1: cleanup del codice, miglioramenti nella redirectione output telemetria - 8 febbraio 2016 .....	16
Prototipo 2: set di istruzioni e prima interfaccia grafica - 9 febbraio 2016.....	17
Prototipo 3: GUI, routines e astrazione - 11, 15, 16 febbraio 2016.....	18
Prototipo 4: astrazione della Telemetria, parsing routines da file - 17 febbraio 2016 .....	23
Prototipo 4: presentazione grafica output: SimpleTelemetryView - 18 febbraio 2016 .....	25
Studio e testing della libreria JFreeChart per i nostri scopi - 19 febbraio 2016 .....	26
Prototipo 5: interfaccia grafica per la visualizzazione dei dati in tempo reale usando JFreeChart - 22 febbraio 2016.....	27
Prototipo 5: risolto problema di sincronizzazione tra thread - 24 febbraio 2016.....	28
Prototipo 5: conversione a progetto Maven per la gestione delle dipendenze - 25 febbraio 2016 ..	29
Miglioramento affidabilità dei timestamp delle misure - 26 febbraio 2016 .....	30
Esportazione dei dati in formato csv, analisi preliminare in Office - 29 febbraio 2016 .....	32
Aggiornamento di codice deprecato e piccoli miglioramenti - 1 marzo 2016.....	33
Testing del motore su diverse routines - 2 marzo 2016 .....	34
Progettazione del package Analyzer - 3, 4 marzo 2016.....	35
Interfaccia grafica per l'analisi - 7, 9 marzo 2016 .....	36
Implementazione Analyzer usando Apache Math commons - 11 marzo 2016.....	41
Cleanup codice, risoluzione bug, piccole migliorie - 14, 16 marzo 2016.....	41
Ultimazione documentazione, grafici uml - 17 marzo 2016.....	42
Readme per utilizzatori, relazione ai tutor - 18 marzo 2016 .....	42

## Analisi del problema e studio dei paper - 11, 12 gennaio 2016

Le pubblicazioni forniteci per lo studio del problema descrivono una procedura di analisi utilizzata per l'identificazione del modello matematico dei motori di un quadricottero, basata sui lavori svolti da Bangura et al. Ai fini dell'identificazione è necessario stimare per ogni motore i coefficienti caratteristici presenti in questa equazione:

$$v_a = K_e \omega + R_a i_a + L_a \frac{di_a}{dt}$$

In cui  $K_e$  è la costante delle armature del motore proporzionale al campo elettromagnetico generato,  $R_a$  è la resistenza del motore e  $L_a$  la sua induttanza. Al fine di semplificare la procedura per l'identificazione di questi parametri si considera il motore in steady state, operante cioè in condizioni di corrente costante, in questo modo la derivata della corrente è nulla ed  $L_a$  si semplifica dall'equazione. Nelle suddette condizioni il motore opera erogando una coppia e quindi un'accelerazione angolare costanti, legate tra loro dalla seguente relazione, in cui si introduce anche il parametro  $K_q$ , costante di coppia del motore:

$$\tau = K_q i_a = I \dot{\omega}$$

Tramite l'ESC è possibile eseguire una routine ad accelerazione, ottenendo istante per istante una stima della corrente circolante nel motore, della tensione applicata e degli rpm.

Tramite regressione lineare si valuta l'accelerazione angolare del motore, Unitamente al momento di inerzia di un disco applicato al motore, l'accelerazione è usata per stimare la coppia generata. Successivamente si stima la corrente media, supposta costante e la si usa per calcolare infine il parametro  $K_q$ .

$$K_q = \frac{\tau}{i_a} = \frac{I \dot{\omega}}{i_a}$$

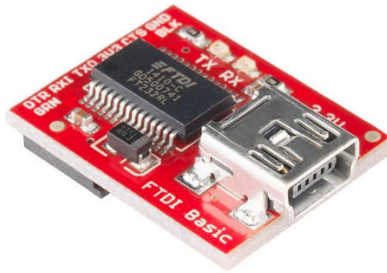
Inoltre, tramite regressione lineare tra gli rpm e la tensione del motore si ottengono  $K_e$  e  $R_a$ , rispettivamente come coefficiente angolare e ordinata all'origine divisa per la corrente media:

$$v_a = K_e \omega + R_a i_a$$

L'analisi qui descritta è condotta tramite un software appositamente sviluppato per gestire la comunicazione tramite seriale con un motore collegato a un ESC, la raccolta dei dati della telemetria e la loro analisi.

## Studio dell'hardware e del banco prova - 14 gennaio 2016

Il banco prova con sul quale metteremo in atto i nostri esperimenti è costituito da un supporto mobile sul quale è montato il rotore T-Motor MT2212 KV750, collegato a un Electronic Speed Controller Autoquad Esc32 v2, a sua volta connesso a un convertitore FTDI-Seriale SparkFun FTDI Basic Breakout 3.3v.



Incontrando per la prima volta questi componenti abbiamo investito del tempo a comprenderne il ruolo e il funzionamento.

### **Rotore T-Motor MT2212 KV750**

Il rotore T-Motor è un brushless motor elettrico progettato per l'utilizzo su quadricotteri e per questo motivo leggero e performante. È in grado di funzionare tra i 1000 e i 10000 rpm, alimentato da una tensione costante tra i 12 e i 15 Volt assorbendo corrente fino a 16 Ampere.

### **Autoquad Esc32 v2**

Per garantire il suo corretto funzionamento, il motore deve essere comandato tramite un Electronic Speed Controller. Questa categoria di dispositivi si occupa di generare le corrette forme d'onda da inviare al motore per ottenere la velocità desiderata, occupandosi anche della gestione della potenza elettrica. In questo modo tramite microcontrollori che operano a basse tensioni è possibile regolare motori di potenza. Inoltre un ESC si può occupare di misurare i parametri di telemetria del motore quali l'effettiva velocità, il consumo istantaneo di corrente e tensione.

### **Convertitore FTDI-Seriale SparkFun FTDI Basic Breakout 3.3v**

Al fine di comandare l'ESC tramite un computer questo viene connesso via seriale utilizzando un convertitore FTDI-Seriale. Per il corretto funzionamento della comunicazione sarà necessario installare e configurare sul computer i driver FTDI.

## Studio del collegamento tramite porta seriale-FTDI - 15 gennaio 2016

Su Windows è necessario installare i driver FTDI dal sito ufficiale SparkFun e configurarli, in modo tale che la periferica FTDI collegata tramite USB venga riconosciuta come porta seriale virtuale COM. In ambiente Linux invece questa configurazione è automatica e il device viene visualizzato come `/dev/ttyUSB0`.

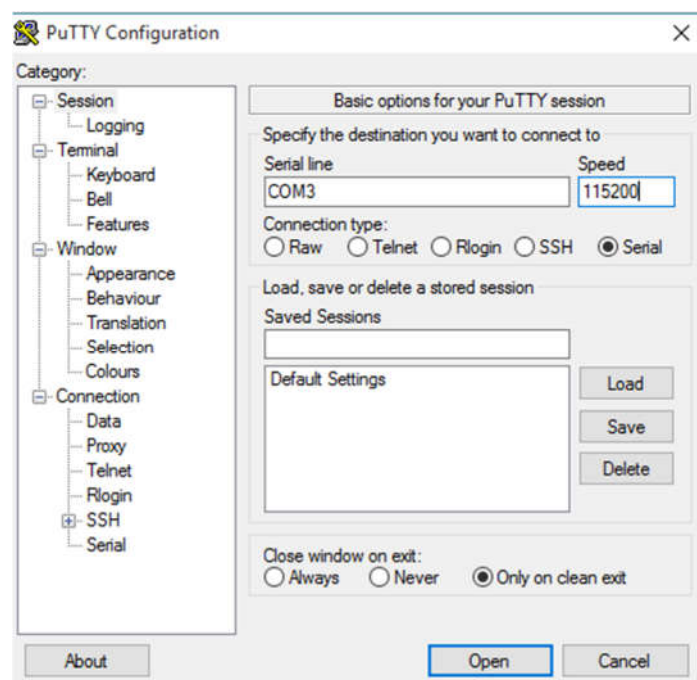
Inoltre in ambiente Linux sono disponibili versioni più aggiornate delle librerie `gnu.io` `rxtx` e `javax.comm`, finalizzate alla comunicazione via seriale da Java. Per questi motivi scegliamo di proseguire il progetto quasi esclusivamente su Ubuntu.

Dal datasheet dell'ESC ricaviamo i seguenti parametri di configurazione da utilizzare per la comunicazione via seriale:

- Baud rate 230400
- No bit di parità
- 1 bit di stop
- 8 bit di dati

## Invio di comandi al rotore tramite linea di comando (Putty) - 18 gennaio 2016

Per un primo approccio con la comunicazione con l'ESC sfruttiamo Putty, un software free ed open-source che, tra le altre cose, offre la funzionalità di emulatore di una console seriale. Per configurare la connessione con la porta seriale sfruttiamo i parametri sopra citati. Una volta aperta la console di putty, è possibile utilizzare tutti i comandi offerti dall'ESC.



L'AutoQuadESC32 v2 offre i seguenti comandi:

- arm: setta l'input mode su UART e arma il motore
- beep <frequency> <duration> : fa emettere un beep al motore (di durata e frequenza pari ai parametri passati)
- bootloader
- config [READ | WRITE | DEFAULT]: configura la modalità di ricezione comandi del motore
- disarm: disarma il motore
- duty: setta il duty cycle del motore
- help: mostra una lista dei comandi disponibili per l'ESC in uso
- input [PWM | UART | I2C | CAN]: seleziona l'input mode del motore, (default UART)
- mode [OPEN\_LOOP | RPM | THRUST | SERVO]: setta la modalità di funzionamento del motore
- pos <degrees>: seleziona l'angolo di partenza del motore
- rpm <rpm>: seleziona la velocità angolare espressa in giri al minuto per il motore
- set LIST | [ <PARAMETER> <VALUE>]: permette di settare una serie di parametri del motore tramite la lista di combinazioni nome-valore passata come argomento
- start: avvia il motore con una velocità angolare pari a 2000 rpm
- status: restituisce in output lo stato del motore
- stop: arresta il motore
- telemetry <Hz>: restituisce in output i dati della telemetria del motore con una frequenza pari a quella passata come argomento

Il nostro tutor del tirocinio ci informa che, per il particolare tipo di ESC in uso, esistono due modi per settare la velocità: o rpm, specificando quindi il numero di giri che il motore deve mantenere, oppure PWM (pulse wide modulation), ossia specificando la potenza costante da fornire al motore, che pertanto non implica che la velocità del motore rimanga costante.

Il range di PWM supportato dal nostro motore è circa 1050 - 1950.

Per collegare il motore all'alimentazione inizialmente utilizziamo 12 V per rimanere in sicurezza, anche se nominalmente dovrebbe funzionare a 14,5-15 V. Come da norme di sicurezza, è necessario collegare prima il cavo rosso (+) e dopo il cavo nero (-).

Osservando i risultati dell'output della telemetria, sfruttando una frequenza massima pari a 50 Hz, i dati variano molto velocemente e sembrano abbastanza accurati e significativi da poter fornire una base valida per una analisi statistica come da scopo del progetto.

## Studio dell'ambiente Git per lo sviluppo in team, integrazione con Eclipse - 20, 21 gennaio 2016

Approcciandoci per la prima volta allo sviluppo di un software in team ci troviamo davanti alla scelta delle metodologie e degli strumenti da utilizzare.

Dalle nostre ricerche online, l'opzione che appare migliore è utilizzare Git come strumento di version control e il servizio GitHub per la condivisione e la sincronizzazione.

Git è utilizzato principalmente per tenere traccia dell'avanzamento del progetto, permettendo inoltre di visualizzare lo stato di un file e i cambiamenti che ha subito in un qualsiasi momento dello sviluppo.

GitHub ci rende comodo la condivisione e il backup dei file, offrendo inoltre un'interfaccia web per visualizzare la storia del progetto.

Git è uno strumento molto ricco e dotato di funzionalità e opzioni articolate. La maggior parte delle guide suggerisce di imparare ad utilizzarlo da linea di comando, per meglio comprenderne le basi. Per questo motivo creiamo una repository temporanea su GitHub sulla quale fare delle prove da locale tramite linea di comando.

Per prima cosa cloniamo la repository:

```
git clone <url>
```

Dopodiché creiamo un file di prova per fare un semplice test di commit

```
echo "prova" > prova.txt
```

Il file attualmente esiste nella directory locale di uno dei nostri pc e non è ancora incluso nella lista dei file osservati da Git, per aggiungerlo:

```
git add prova.txt
```

Un'istantanea del file è ora stata aggiunta alla staging area, un luogo particolare di Git in cui tutti i file che si vogliono in seguito committare vengono preparati. Volendo è possibile ora continuare a lavorare sul file o creare altri file per poi aggiungerli alla staging area, al fine di questa prova invece procediamo subito al commit:

```
git commit
```

Tramite questo comando le modifiche introdotte ai file staged vengono rese permanenti nella storia locale della repository. Al commit vengono associati un timestamp e un hash in modo da renderlo univoco, inoltre per ogni commit è necessario scrivere un messaggio di commit molto utile per riassumere i cambiamenti effettuati al progetto senza necessariamente dover guardare le modifiche di ogni file.

A questo punto il commit è avvenuto solamente all'interno della repository locale, affinché venga propagato anche a quella remota su GitHub è necessario scrivere:

```
git push
```

Dall'altro computer è ora possibile ottenere i nuovi commit tramite il comando

```
git fetch
```

Sono stati importati nella repository locali i commit effettuati sull'altro pc, al fine di unirli al flusso locale è necessario un merge:

```
git merge
```

I comandi fetch e merge possono essere riassunti per semplicità in

```
git pull
```

Per il secondo computer è ora possibile accedere al file creato dall'altro e visualizzare l'elenco delle modifiche tramite:

```
git log
```

Fino a qui nulla di eccezionale, abbiamo ottenuto quello che avremmo potuto ottenere scambiandoci i file con una chiavetta USB, con il vantaggio che possiamo conservare una storia dei cambiamenti e che in ogni evenienza i nostri file sono al sicuro sui server di GitHub.

Ma cosa succederebbe tuttavia se decidessimo di modificare entrambi lo stesso file e rendere persistenti le nostre modifiche con un commit?

```
PC 1:
echo "pc1" >> prova.txt
echo "altro" > altro.txt
git commit -a
git push

PC 2:
echo "pc2" >> prova.txt
```

```
git commit -a  
git pull
```

Al momento del pull il PC 2 fa il fetch del commit del PC1, Git non è in grado di unificare automaticamente (merge fast-forward) le modifiche introdotte dai due commit al file prova.txt e presenta all'utente un conflict da risolvere. Dettagli sul conflict sono ottenibili tramite:

```
git status
```

All'interno del file prova.txt sono presenti entrambe le versioni e per risolvere il conflict è necessario modificarlo a mano preservando la prima, la seconda o entrambe, fatto ciò si indica a Git che il conflict è stato risolto ed è possibile effettuare un nuovo commit.

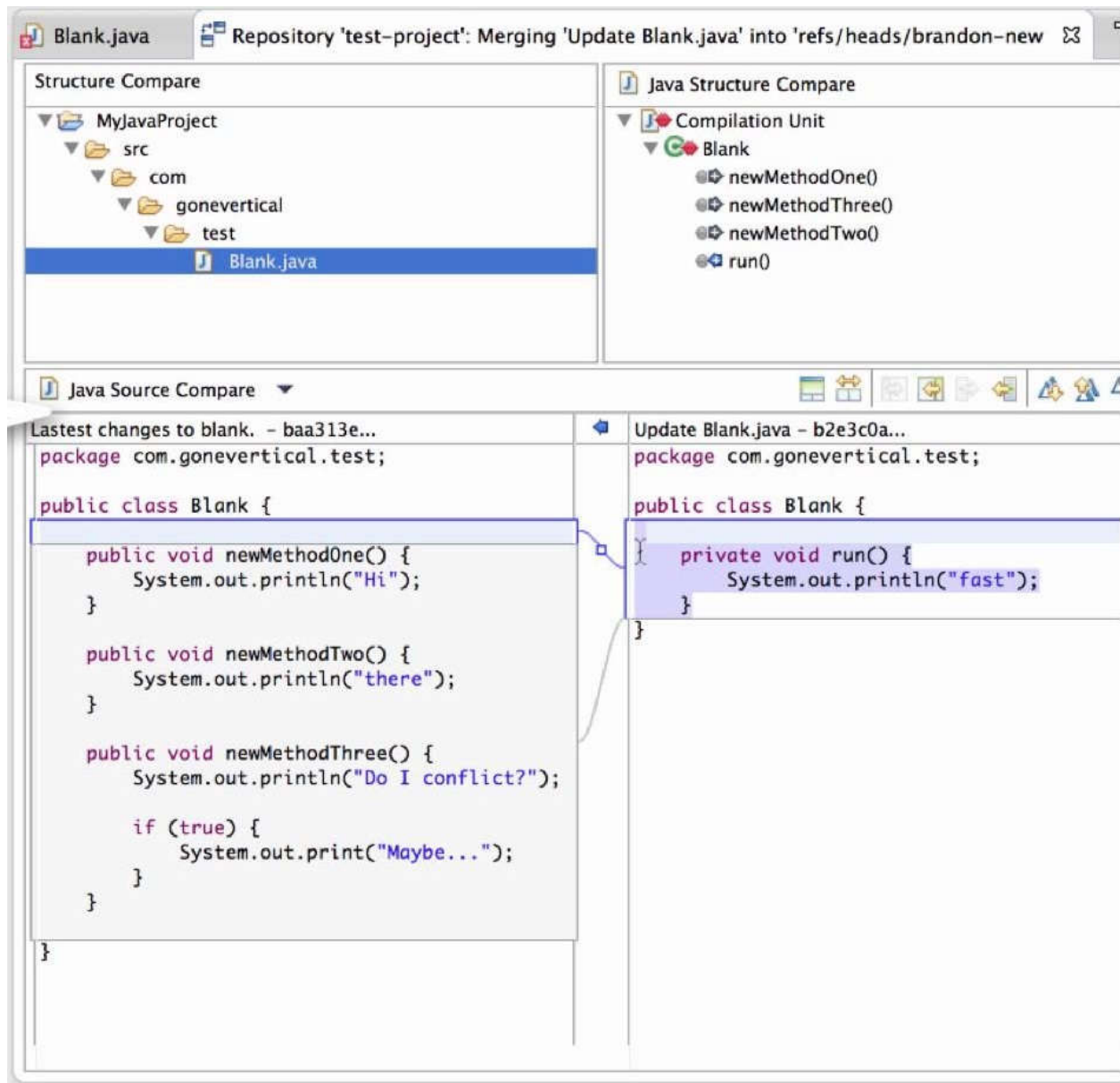
```
git add prova.txt  
git commit
```

Questo commit presenterà un messaggio di merge prodotto automaticamente da Git e una volta pushato su GitHub sarà visibile anche dal PC 1.

Questa funzionalità ci permette comodamente di lavorare in parallelo su parti diverse del progetto, senza perdere troppo tempo a integrare i due lavori nel caso di riscritture contemporanee.



L'IDE Eclipse offre al suo interno un'efficace integrazione con Git permettendo dall'interfaccia grafica di effettuare con facilità le operazioni viste finora come clonare una repository di GitHub importando i progetti Eclipse contenuti al suo interno, modificare usualmente i file, metterli in staging ed effettuare i commit. Nell'eventualità di conflitti durante un merge offre inoltre uno strumento visuale, molto più pratico per comparare le versioni e selezionare le modifiche da mantenere.



## Studio della libreria rxtx e comunicazione seriale - 22 gennaio 2016

Come già detto, scegliamo di rimanere in ambiente linux per comodità. In tale ambiente installiamo le già citate librerie javax.comm e gnu.io rxtx-java. Quest'ultima risulta essere più funzionale ed aggiornata, pertanto scegliamo di proseguire il progetto con essa. Per installazione ed uso della libreria eseguiamo i seguenti passi:

- Esecuzione del comando `sudo apt-get install librxjava`
- Aggiunta del jar della libreria al classpath del progetto
- da ubuntu 13.04 in poi per comunicare con le porte seriali bisogna essere nel gruppo dialout, pertanto eseguiamo il comando: `sudo adduser $USER dialout && sudo reboot`
- Importiamo nel progetto la javadoc per le api appena aggiunte per la comunicazione con le porte seriali

La libreria offre molte funzionalità per la comunicazione e l'uso di porte seriali, nel nostro uso andremo ad utilizzare le seguenti classi e le loro funzionalità:

- **gnu.io.CommPort**: classe astratta rappresentante una porta di comunicazione generica e il suo contratto d'uso. Offre metodi funzionali come `getInputStream()`, `getOutputStream()` per ottenere gli stream per la comunicazione con la porta, `getName()` per ottenere il nome della porta, `close()` per chiudere la comunicazione aperta con la porta.
- **gnu.io.SerialPort**: subclass astratta di CommPort che rappresenta una porta seriale; oltre i metodi ereditati offre metodi specifici per configurazione e lettura di parametri d'uso della porta come baud rate, bit di parità, bit di data e bit di stop (`setSerialPortParams(int,int,int,int)`). Inoltre offre la possibilità di registrare un listener (istanza di `SerialPortEventListener`) per la gestione degli eventi di comunicazione ad interrupt.
- **gnu.io.RXTXPort**: implementazione di SerialPort
- **gnu.io.CommPortIdentifier**: astrazione dell'identificativo di una porta di comunicazione, offre metodi come `open()` che restituisce un'istanza connessa di CommPort, o `getPortType()` per ottenere il tipo della porta rappresentata dall'identificativo.

Riportiamo un estratto del nostro codice per esempio di utilizzo della libreria:

```
SerialPort connect(CommPortIdentifier selectedPortIdentifier)
    throws PortInUseException, UnsupportedOperationException
{
    SerialPort serialPort = null;
    // the method below returns an object of type SerialPort
    serialPort = (SerialPort) selectedPortIdentifier.open(
        "multi-rotor-auto-tuning", TIMEOUT);
    // parameters needed for AutoquadEsc32
    serialPort.setSerialPortParams(230400, SerialPort.DATABITS_8,
        SerialPort.STOPBITS_1,
        SerialPort.PARITY_NONE);
    System.out.println("Connection Established");
    return serialPort;
}
```

Nel codice abbiamo cablato i parametri di connessione specifici per l'AutoQuadEsc32. Per offrire maggiore flessibilità del software sarebbe meglio permettere all'utente di specificare i suoi parametri, ma per semplicità di utilizzo rimandiamo lo sviluppo di questa feature.

## Prototipo 1: semplice invio comandi seriale tramite Java - 26, 29 gennaio 2016

Prototipo 1: sviluppo di un semplice applicativo che emula putty e permette di inviare e ricevere comandi via seriale, interfaccia da linea di comando per scelta della porta e comunicazione, un ESC è modellato solo come due stream di comunicazione: input e output, risoluzione del problema di LF e CR

Come primo approccio Java-based al progetto cerchiamo di realizzare un applicativo in grado di emulare il comportamento di Putty. In particolare vogliamo avere ottenere un software in grado di inviare e ricevere comandi via seriale con una semplice interfaccia da linea di comando per la scelta della porta e la comunicazione.

In questo primo prototipo, l'intero progetto è realizzato in un'unica classe *Communicator* che comprende al suo interno tutti le funzionalità necessarie e l'entry point dell'applicativo. La classe risulta avere uno scheletro di questo tipo:

```
public class Communicator {
    // for containing the ports that will be found
    private static Enumeration<CommPortIdentifier> ports = null;
    // map the port names to CommPortIdentifiers
    private static HashMap<String, CommPortIdentifier> portMap;
    // this is the object that contains the opened port
    private static CommPortIdentifier;
    private static SerialPort serialPort;
```

```

// input and output streams for sending and receiving data
private static InputStream input = null;
private static OutputStream output = null;
private static ByteArrayOutputStream inputBuffer =
    new ByteArrayOutputStream();
// the timeout value for connecting with the port
final static int TIMEOUT = 2000;
private static BufferedReader console;

public static void main(String[] args) {
    scanPorts();
    selectPort();
    connect();
    new Thread(new Runnable() {...}).start();
    //lettura ed invio comandi
}

private static void close() {...}
private static void sendCommand(String string) {...}
private static void selectPort() {...}
private static void scanPorts() {...}
private static void connect() {...}
}

```

I metodi offerti dalla classe sono i seguenti:

- *public static void scanPorts();* con il quale vengono scannerizzate tutte le porte di comunicazione di cui è dotato il calcolatore, filtrate quelle di tipo seriale e aggiunte al field statico *ports*.
- *private static void selectPort();* con il quale, tramite interazione da riga di comando, viene selezionata la porta da utilizzare per la comunicazione e assegnata al field statico *serialPort*.
- *private static void connect();* con il quale avviene la connessione alla porta selezionata (l'implementazione del metodo è riportata nel capitolo precedente).
- *private static void close();* con il quale viene chiuso il canale di comunicazione con la porta seriale.
- *private static void sendCommand(String);* con il quale viene inviato un comando all'ESC tramite la porta seriale. Durante la realizzazione di questo metodo ci siamo scontrati con un problema imprevisto: i comandi inviati come semplice stringa di testo serializzata in byte non venivano recepiti ed eseguiti dall'ESC. Dopo ripetute prove ed analisi dell'output prodotto dall'ESC abbiamo capito che un comando ben formato necessita di essere terminato dai caratteri *line feed* e *carriage return* rappresentati dai codici UTF-8 13 e 10

rispettivamente. Riportiamo l'intera implementazione del metodo per completezza:

```
private static void sendCommand(String string) {
    try {
        output.write(string.trim().getBytes("UTF-8"));
        output.write(13); // LF
        output.write(10); // CR
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Per la ricezione dell'output della telemetria, scegliamo inizialmente di sfruttare le funzionalità offerte dalla classe `SerialPort` registrando un listener (istanza di *`SerialPortEventListener`*) in grado di gestire ad eventi la comunicazione. Dopo qualche tentativo con frequenza di telemetria variabile, ci rendiamo conto che la gestione ad interrupt di una porta seriale è adatta solamente nei casi in cui la quantità e la frequenza di input dei dati non sia elevata. Ipotizziamo infatti che venga generato un evento ogni volta che è presente un carattere sullo stream di input e che questo renda assolutamente lenta ed inefficiente la lettura. Per questo motivo scegliamo di dedicare un thread apposito alla lettura continua dello stream di input, fintanto che questo non viene chiuso. Compito del thread è quello di leggere e filtrare i dati e presentarli in output.

```
new Thread(new Runnable() {
    private int i=1;
    @Override
    public void run() {
        try {
            byte singleData;
            while ((singleData = (byte) input.read()) != -1) {
                inputBuffer.write(singleData);
                if (singleData == 10) {
                    ByteArrayInputStream bin =
new ByteArrayInputStream(inputBuffer.toByteArray());
                    BufferedReader reader =
new BufferedReader(new InputStreamReader(bin, "UTF-8"));
                    String line = reader.readLine();
                    if (line.startsWith("RPM")) {
                        StringTokenizer st = new StringTokenizer(line);
                        st.nextToken();
                        System.out.println(st.nextToken() + ",");
                    } else if (line.startsWith("MOTOR VOLTS")) {
                        StringTokenizer st = new StringTokenizer(line);
                        st.nextToken();
                        st.nextToken();
                        System.out.println(st.nextToken() + "\n");
                    }
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
});
```

```

        if(i++%50==0)
            System.out.println("lette triplete " + i);
    } else if (line.startsWith("AMPS AVG")) {
        StringTokenizer st = new StringTokenizer(line);
        st.nextToken();
        st.nextToken();
        System.out.println(st.nextToken() + ",");
    }
    inputBuffer.reset();
}
}
} catch (Exception e) {
    e.printStackTrace();
}
}
}).start();

```

Il thread realizzato in questo prototipo costruisce una stringa leggendo i dati dalla seriale. Per ogni linea completa ricevuta filtra quelle rappresentanti gli output della telemetria per RPM, MOTOR VOLTS e AMPS AVG.

Da questa classe unica, procediamo poi a estrarre la prima astrazione di un ESC, rappresentata da una semplice classe:

```

public class ElectronicSpeedController {
    private SerialPort port;
    private InputStream input;
    private OutputStream output;

    public ElectronicSpeedController(SerialPort port) {
        this.port = port;
        this.input = port.getInputStream();
        this.output = port.getOutputStream();
    }
}

```

## Studio della telemetria del motore e classe AutoQuadEsc32 - 1, 2, 4 febbraio 2016

Cominciamo ad analizzare nel dettaglio le funzionalità offerte dall'AutoQuad ESC 32 al fine di modellarlo come oggetto del mondo Java, fino ad ora infatti l'ESC non è stato altro che una façade per accedere agli stream di comunicazione della porta seriale. Vogliamo invece che la classe modelli un AutoQuad ESC 32 in termini di comandi e di telemetria prodotta in output.

Per quanto semplificata al massimo, questa classe offre i metodi di base per comandare un ESC e riceverne la telemetria tramite un thread separato.

Senza entrare nel dettaglio implementativo la classe realizzata presenta la seguente interfaccia:

```
public class ElectronicSpeedController {
    private InputStream input;
    private OutputStream output;
    private ReaderThread reader;
    private static HashSet<String> telemetryParameters =
new HashSet<>();
    public static String[] allParameters = {"INPUT MODE", "RUN
MODE", "ESC STATE", "PERCENT IDLE", "COMM PERIOD", "BAD
DETECTS", "FET DUTY", "RPM", "AMPS AVG", "AMPS MAX", "BAT
VOLTS", "MOTOR VOLTS", "DISARM CODE", "CAN NET ID"};

    public ElectronicSpeedController(SerialPort port) throws
IOException {...}

    public void sleep(long millis){...}
    public void setRPM(int rpm) {...}
    public void arm() {...}
    public void disarm() {...}
    public void start() {...}
    public void stop() {...}
    public void sendCommand(String command){...}

    public void startTelemetry(int frequency, PrintWriter writer)
    {...}
    public void stopTelemetry() {...}

    public void addTelemetryParameter(String parameter){...}
    public void addTelemetryParameters(String[] parameters){...}
    public void removeTelemetryParameter(String parameter){...}
    private class ReaderThread extends Thread {...}
}
```

I metodi vengono chiamati dal main uno di seguito all'altro e verifichiamo il corretto funzionamento della classe accertandoci che i movimenti del motore corrispondano alle istruzioni inviate.

La telemetria è gestita dal ReaderThread che filtra le linee ricevute tramite seriale in base alle stringhe indicate tramite addTelemetryParameter e removeTelemetryParameter. L'output della telemetria è indirizzato in forma non formattata (un valore per ogni linea) verso un print writer, rappresentato indipendentemente dalla console o da un file di testo semplice.

## Prototipo 1: cleanup del codice, miglioramenti nella redirezione output telemetria - 8 febbraio 2016

Dopo la prima implementazione della classe `ElectronicSpeedController` fatta durante la giornata precedente cominciamo a ragionare ora su alcune funzionalità che rendano più comoda la scrittura del codice e l'analisi della telemetria.

Come prima miglioria facciamo sì che i metodi utilizzati per inviare comandi all'ESC restituiscano l'ESC stesso, in modo tale da poterli invocare uno di seguito all'altro facendo chaining delle chiamate.

```
public ElectronicSpeedController sleep(long millis){...}
public ElectronicSpeedController setRPM(int rpm) {...}
public ElectronicSpeedController arm() {...}
public ElectronicSpeedController disarm() {...}
public ElectronicSpeedController start() {...}
public ElectronicSpeedController stop() {...}
public ElectronicSpeedController sendCommand(String command){..}
public ElectronicSpeedController startTelemetry(int frequency,
    PrintWriter writer){...}
public ElectronicSpeedController stopTelemetry() {...}
```

Così facendo il codice del Main risulta molto più fluente, pulito e leggibile:

### PRIMA

```
esc.arm();
esc.sleep(1000);
esc.startTelemetry(10,
writer);
esc.start();
esc.sleep(10000);
esc.startTelemetry(10, file);
esc.sleep(10000);
esc.stopTelemetry();
esc.stop();
esc.disarm();
```

### DOPO

```
esc.arm()
    .sleep(1000)
    .startTelemetry(10,writer)
    .start()
    .sleep(10000)
    .startTelemetry(10, file)
    .sleep(10000)
    .stopTelemetry()
    .stop()
    .disarm();
```

Modificando leggermente il codice del `ReaderThread` siamo in grado di ottenere un file formattato in csv. Per quanto semplice, questo formato ci permette di analizzare meglio i dati della telemetria e di esportarli verso una suite di più alto livello.

Avendo a disposizione LibreOffice su Ubuntu importiamo alcuni file generati da diversi set di istruzioni e proviamo ad analizzarli a mano. LibreOffice ci permette di effettuare



analisi statistiche preliminari sulla telemetria, disegnare i grafici dei dati ottenuti e farci un'idea di come vogliamo realizzare l'interfaccia grafica del programma.

## Prototipo 2: set di istruzioni e prima interfaccia grafica - 9 febbraio 2016

Passiamo ora alla realizzazione della prima semplice interfaccia grafica, che consiste in una semplice vista dotata di un pulsante e di una text area. Alla pressione del pulsante avviene l'esecuzione sul motore di una serie di istruzioni hard coded simile a quella precedentemente descritta. Nella text area invece viene mostrato l'output della telemetria del motore (se avviata nelle istruzioni). Questa primissima prova di interfaccia risulta poco funzionale in quanto la telemetria è piuttosto veloce e la text area non riesce a mostrare in real time i dati, e ciò risulta in una visualizzazione confusionaria ed inefficiente.

Per molte istruzioni è stato sufficiente far sì che l'invocazione dell'istruzione provocasse la scrittura sullo stream seriale dei caratteri corrispondenti al comando dell'ESC. Tuttavia, non essendo il metodo *accelerate* built-in nell'ESC, è stato necessario realizzarne un'implementazione sfruttando le primitive a disposizione.

```
private AutoQuadEsc32 accelerate(int from, int to, double pace)
{
    if (pace == 0 || from == to)
        throw new IllegalArgumentException("Cannot accelerate");

    if (pace < -400)
        System.out.println("WARNING: deceleration with a rate greater
                             than 400 rpm/s cannot be achieved");
    int deltaRpm = pace > 0 ? 1 : -1;
    long deltaT = Math.round(deltaRpm / pace * 1000);

    if (from < to && pace > 0) {
        setRPM(from);
        while (from <= to) {
            from += deltaRpm;
            setRPM(from);
            try {
                Thread.sleep(deltaT);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    } else if (from > to && pace < 0) {
        setRPM(from);
        while (from >= to) {
            from += deltaRpm;
            setRPM(from);
            try {
                Thread.sleep(deltaT);
```

```

    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
} else
    throw new IllegalArgumentException("from, to, pace not
compatible");
return this;
}

```

### Prototipo 3: GUI, routines e astrazione - 11, 15, 16 febbraio 2016

Scegliamo ora di reindirizzare il nostro progetto verso una struttura più adatta all'estensione e alla manutenibilità. Tramite l'astrazione dei singoli componenti vogliamo permettere un utilizzo più generale e personalizzabile del software:

- Architettura a model-view-controller per rendere semplice lo sviluppo di interfacce grafiche e modelli dei dati intercambiabili
- Astrazione dal modello di ESC utilizzato e dai particolari comandi a cui risponde
- Astrazione del set di istruzioni inviabili agli ESC
- Astrazione delle routines eseguibili dagli ESC intese come successione di istruzioni

Suddividiamo il progetto nei package:

- VIEW
- CONTROLLER
- ESC
- ROUTINE
- SERIALPORTS

All'interno del package ROUTINE è contenuta la parte del modello relativa all'astrazione dell'instruction set e delle routines. Le principali classi al suo interno sono:

**InstructionType:** il set di possibili istruzioni che un generico ESC deve poter eseguire, rappresentato da un enumerativo

```

public enum InstructionType {
    ARM, DISARM, SLEEP, START, STOP, SET_RPM, ACCELERATE,
    START_TELEMETRY, STOP_TELEMETRY;
}

```

Abbiamo cercato di individuare le più comuni istruzioni che è possibile inviare ad un ESC, basandoci sulla nostra attuale conoscenza limitata solamente al modello che ci è stato fornito e a qualche ricerca online. Alcune istruzioni probabilmente troveranno un corrispondente diretto nel set di istruzioni del proprio ESC, altre come ACCELERATE andranno gestite come istruzioni di più alto livello e scomposte in comandi più semplici.

**Instruction:** rappresenta un'istruzione generica da far eseguire a un'istanza di ESC. si compone di un tipo specifico (InstructionType) e una mappa tramite la quale è possibile associare ulteriori parametri là dove è necessario, ad esempio per impostare la velocità ad un certo valore.

Al fine di limitare la creazione di Instruction identiche sono state fornite istanze statiche delle istruzioni che non prevedono parametri

```
public class Instruction {
    public static final Instruction ARM =
        new Instruction(InstructionType.ARM, null);
    public static final Instruction DISARM =
        new Instruction(InstructionType.DISARM, null);
    public static final Instruction START =
        new Instruction(InstructionType.START, null);
    public static final Instruction STOP =
        new Instruction(InstructionType.STOP, null);
    public static final Instruction STOP_TELEMETRY =
        new Instruction(InstructionType.STOP_TELEMETRY, null);

    public final InstructionType type;
    public final Map<String, Object> parameters;
    public Instruction(InstructionType type,
        Map<String, Object> parameters){
        this.type = type;
        this.parameters = parameters;
    }
}
```

**Routine:** questa classe rappresenta un oggetto in grado di far eseguire una serie di Instruction a un generico ESC. Routine è sottoclasse di Thread e quindi il controller può avviarla nel momento in cui desidera l'utente. Nel suo metodo run() la routine chiamerà sull'ESC passatole il metodo executeInstruction() per ogni istruzione presente nella lista mantenuta al suo interno. Al momento le routines possono essere definite solamente da codice passando al costruttore di Routine una lista di istruzioni, per comodità di testing abbiamo inoltre scritto due routines con un elenco predefinito di istruzioni che impostano l'ESC a velocità costante o lo fanno accelerare. Prevediamo inoltre di rendere più user friendly la definizione di una routine tramite file di testo.

```

public class Routine extends Thread {

    public AbstractEsc esc;
    protected SerialPort serialPort;
    protected List<Instruction> instructions;

    public Routine(List<Instruction> instructions) {
        this.instructions = instructions;
    }

    public void setEsc(AbstractEsc esc) {
        this.esc = esc;
    }

    public void run() {
        if (instructions == null)
            throw new IllegalStateException("Routine has null
set of instructions");
        if (esc == null)
            throw new IllegalStateException("Routine has no ESC
attached");
        for (Instruction i : instructions)
            esc.executeInstruction(i);
    }

    public String toString(){
        return "Routine";
    }

    public PipedOutputStream getOutput() {
        return esc.getPipedOutput();
    }

}

```

All'interno del package ESC vengono modellate le gerarchie e le classi relative agli Electronic Speed Controller:

**AbstractEsc:** è la rappresentazione astratta di ciò che un ESC deve essere in grado di fare in termini di funzionalità e servizi offerti, in particolare eseguire un'istruzione, gestire l'output della propria telemetria in base ai parametri richiesti e disconnettersi in sicurezza

```

public abstract class AbstractEsc {
    protected InputStream input;
    protected OutputStream output;
    protected PipedOutputStream pipedOutput;
    private SerialPort port;
    protected static ArrayList<String> telemetryParameters;
}

```

```

    public AbstractEsc(SerialPort port) throws IOException {...}
    public PipedOutputStream getPipedOutput() {...}
    public abstract void executeInstruction
        (Instruction instruction);
    public abstract void setTelemetryParameters
        (List<String> params);
    public final AbstractEsc sleep(long millis){...}
    public void stopAndDisconnect(){
        executeInstruction(Instruction.STOP);
        executeInstruction(Instruction.STOP_TELEMETRY);
        executeInstruction(Instruction.DISARM);
        port.close();
    }
}

```

**AutoQuadEsc32:** la nostra implementazione di un generico ESC, realizzata nel contesto di questo progetto, contiene la logica per gestire l'invio dei comandi tramite seriale al fine di rispondere all'invocazione del metodo `executeInstruction`, per la lettura della telemetria si affida al `ReaderThread` precedentemente descritto

**EscLoader:** questa classe rappresenta un loader per le diverse implementazioni di ESC disponibili a runtime, permette sia di elencare le classi implementanti `AbstractEsc` che di crearne un'istanza. Questo approccio è stato scelto per garantire agli utenti futuri la possibilità di realizzare la propria implementazione di ESC e vederla riconosciuta automaticamente dal software senza doverlo ricompilare.

Per realizzare questo comportamento abbiamo imparato ad utilizzare le caratteristiche di Reflection di Java, necessarie per ispezionare il codice a runtime ed individuare le sottoclassi di `AbstractEsc`. La libreria `Reflections` sviluppata da Google si dimostra un valido strumento per sintetizzare il processo di ricerca in poche righe di codice.

Senza reflection, tipologie di ESC hardcoded e difficilmente modificabili:

```

public class EscLoader {
    private List<Class<? extends AbstractEsc>> escList;

    public EscLoader() {
        escList = new ArrayList<Class<? extends AbstractEsc>>();
        escList.add(AutoQuadEsc32.class);
    }
    public List<Class<? extends AbstractEsc>> getEscList() {
        return escList;
    }

    public AbstractEsc newInstanceOf(Class<? extends AbstractEsc>
        cls, SerialPort port) {...}
}

```

Con reflection, un utente può aggiungere a runtime la sua implementazione di AbstractEsc:

```
public static List<Class<? extends AbstractEsc>> getEscsList() {  
    Reflections reflections = new Reflections("esc");  
    Set<Class<? extends AbstractEsc>> subTypes =  
        reflections.getSubTypesOf(AbstractEsc.class);  
    return new ArrayList<>(subTypes);  
}
```

Il package VIEW contiene l'interfaccia grafica propria dell'applicativo, in particolare una finestra principale contenente i comandi per avviare delle routines e visualizzarne l'output. Questa configurazione grafica è solamente temporanea poiché prevediamo di creare una finestra separata per la visualizzazione della telemetria in un formato più comprensibile, possibilmente con grafici in real time.

**MainView:** la finestra principale dell'applicazione, contiene gli elementi per selezionare una routine tra quelle preconfigurate e lanciarla sull'Esc selezionato tramite EscSelectorGui. Utilizza componenti base del package Swing come le JComboBox e i JButton, facendo riferimento ai metodi del model (getRoutines) per caricare le Routines e del controller (startRoutine) per far partire la routine selezionata nella combobox. A lato è presente una JTextArea alla quale viene associato un thread che ha il compito di leggere i dati che vengono forniti dalla telemetria e scriverli sulla stessa.

Riportiamo ora il flusso dei dati di telemetria dall'ESC alla text area:

- Il metodo startTelemetry chiamato sull'ESC crea un ReaderThread il quale riceve i byte in uscita dall'esc per la telemetria, li trasforma in stringhe e valori numerici e li filtra in base ai parametri di telemetria desiderati
- In uscita dal ReaderThread i dati vengono scritti su di un PipedOutputStream che può essere acceduto tramite la classe Routine che al suo interno contiene un'istanza di AbstractEsc
- All'avvio della routine tramite il button Start, il Controller ricava dalla routine lo stream dei dati dal quale crea la sua controparte di input, il PipedInputStream, e lo setta sulla view
- Dal PipedInputStream la MainView ricava i dati da scrivere nella JTextArea.

All'interno del package CONTROLLER è contenuta la classe Controller il cui compito è far partire l'interfaccia di selezione dell'ESC, ricevere da esso un ESC connesso e far partire la MainView inizializzandone la grafica. È inoltre dotato dei già citati metodi startRoutine per avviare una routine e i componenti grafici ad essa associati.

Il package SERIALPORTS non partecipa direttamente all'architettura complessiva e non deve essere inquadrato in una visione a model-view-controller. Al suo interno si trovano le classi per la creazione di un semplice dialogo EscSelectorGUI che permette di selezionare il tipo di ESC in uso e la porta a cui connettersi sfruttando al suo interno le classi precedentemente scritte per l'interfaccia a linea di comando.

Alla pressione del pulsante di conferma crea la connessione con l'ESC e restituisce al controller da cui è stato chiamato l'ESC connesso. A seguito di ciò sarà il controller ad mostrare la GUI vera e propria dell'applicativo dalla quale da cui si possono lanciare e fermare routines.

#### Prototipo 4: astrazione della Telemetria, parsing routines da file - 17 febbraio 2016

La creazione di routines è possibile solamente tramite codice e la definizione di una nuova routine implica una ricompilazione del codice. Questa caratteristica è evidentemente indesiderata per un software che vuole essere utilizzabile anche da utenti meno esperti. Per questo motivo ci impegniamo a rendere possibile la definizione di una routine da file di testo semplice, il quale dovrà essere soggetto a parsing e quindi caricato come routine dall'applicazione.

Prima di questo tuttavia procediamo ad astrarre i parametri della telemetria come un enumerativo, ricordando che fino a questo momento li avevamo trattati come semplici stringhe e dunque dipendenti dallo specifico modello di ESC in uso. Ad ogni parametro della telemetria associamo inoltre una classe rappresentante il tipo di dato (intero, decimale, stringa) e una stringa che funga da identificativo nei file di testo delle routines.

Da notare come non sia corretto dal punto di vista della astrazione che ai TelemetryParameter sia associata la stringa specifica dell'AutoQuadEsc32. Sarebbe più corretto che ogni implementazione di un ESC mantenga al suo interno una tabella di conversione tra la sua telemetria interna e i telemetry parameter astratti. Ad esempio un altro modello di ESC potrebbe utilizzare la stringa "STATE" o addirittura i byte 0xAB per identificare il suo stato. Tuttavia, pur lasciando qui le stringhe, ogni implementazione di AbstractEsc può decidere di ignorarle e gestire la telemetria a modo suo filtrando in base al valore dell'enumerativo e non della stringa contenuta al suo interno.

```
public enum TelemetryParameter {  
    INPUT_MODE("INPUT MODE", String.class),  
    RUN_MODE("RUN MODE", String.class),  
    ESC_STATE("ESC STATE", String.class),  
    PERCENT_IDLE("PERCENT IDLE", Double.class),
```

```

    COMM_PERIOD("COMM PERIOD", Double.class),
    BAD_DETECTS("BAD DETECTS", Integer.class),
    FET_DUTY("FET DUTY", Double.class),
    RPM("RPM", Double.class),
    AMPS_AVG("AMPS AVG", Double.class),
    AMPS_MAX("AMPS MAX", Double.class),
    BAT_VOLTS("BAT VOLTS", Double.class),
    MOTOR_VOLTS("MOTOR VOLTS", Double.class),
    DISARM_CODE("DISARM CODE", Integer.class),
    CAN_NET_ID("CAN NET ID", Integer.class);

    public static TelemetryParameter parse(String string) {...}
}

```

Come ulteriore step preliminare applichiamo il pattern factory alla classe Instruction, che prima gestiva in modo diverso la creazione di istruzioni che necessitano parametri (es ACCELERATE) e l'ottenimento di istruzioni che non necessitano parametri (es. ARM) e soprattutto permetteva la creazione duplicata di istruzioni che non necessitano parametri. Nascondiamo inoltre l'utilizzo di una mappa parametro-valore all'interno per meglio presentare all'esterno i metodi.

```

public class Instruction {
    ...
    private static final Instruction STOP =
        new Instruction(InstructionType.STOP, null);

    ...
    public static final Instruction newAcceleration
    (int from, int to, double pace) {
        Map<String, Object> map = new HashMap<String, Object>();
        map.put("from", from);
        map.put("to", to);
        map.put("pace", pace);
        return new Instruction(InstructionType.ACCELERATE, map);
    }

    public static final Instruction newStop() {
        return STOP;
    }
}

```

Passiamo dunque alla definizione di files di routine, in formato di testo semplice, con estensione .rou e con la seguente struttura:

- la prima riga contiene il nome della routine
- la seconda riga contiene una lista di parametri da utilizzare nella telemetry sotto forma di stringhe separate da virgole(vedere la classe TelemetryParameter per le specifiche stringhe).



- le righe successive devono contenere una valida istruzione per l'ESC, così come specificate nella classe Instruction. Se l'istruzione necessita parametri aggiuntivi questi vanno specificati dopo il nome dell'istruzione e un ':', ogni parametro separato da uno spazio. Le linee vuote e quelle che iniziano per '#' vengono ignorate.

Un esempio di file di routine valido è il seguente:

```
Rapid acceleration from 2000 to 6000 rpm
RPM, AMPS AVG, MOTOR VOLTS

arm
start
rpm: 2000
sleep: 10000
telemetry: 50
sleep: 1000

# accelerate at 1200 rpm/s
accelerate: 2000 6000 1200
accelerate: 6000 1000 -400

sleep: 5000
stop telemetry
stop
disarm
```

Il parsing di un file in questo formato non presenta grandi difficoltà, se non molta pazienza per scrivere il codice e verificarne il funzionamento. Il parser di routines accetta in input un elenco di directory dal quale caricare tutti i file .rou realizzandone istanze di routines in seguito ottenibili tramite un altro metodo del parser. In caso di fallimento durante il parsing di un file viene sollevata una `FileFormatException` definita appositamente.

Alla classe `Routine` sono stati apportati alcuni cambiamenti come conseguenza di questa nuova feature. Innanzitutto passiamo dall'estendere `Thread` ad implementare `Runnable`, in questo modo una volta creata una routine è possibile eseguirla più volte associandola a un `Thread`, diversamente da quando era essa stessa un `Thread` e dunque monouso. Inoltre al costruttore si aggiungono il nome e la lista dei `TelemetryParameter`.

Prototipo 4: presentazione grafica output: `SimpleTelemetryView` - 18 febbraio 2016

Il passo successivo consiste in un miglioramento sostanziale della presentazione dei dati. Alla pressione del tasto `START` presente nella `MainView` viene invocato il metodo

*startRoutine(Routine routine)* con il quale viene avviata la routine e viene contestualmente generata una nuova view, rappresentata dalla classe *SimpleTelemetryView*, per la presentazione dell'output. Questa vista è composta da una tabella generata dinamicamente:

- Nella prima colonna troviamo delle label che rappresentano i nomi dei parametri di interesse ottenuti dalla routine. In particolare si tratta di *TextField* non editabili riempiti con il nome dei *TelemetryParameter* ottenuti dal metodo *getParameters()* della routine.
- Nella seconda colonna invece vi sono dei *TextField* non editabili nei quali viene mostrata (e coerentemente aggiornata) la misurazione del corrispondente parametro ottenuta dalla telemetria.

```
private void initGraphics(){
    this.setLayout(new GridLayout(parameters.size(), 2));
    for(TelemetryParameter p : parameters){
        this.add(new JTextField(p.name));
        JTextField field = new JTextField();
        binding.put(p, field);
        this.add(field);
    }
    setSize(640, 480);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setVisible(true);
}
```

All'istanziamento di un oggetto *SimpleTelemetryView*, passando come parametro una routine, viene creato l'end point di input del *PipedOutputStream* ottenuto dalla routine, viene invocato il metodo *initGraphics()* e viene lanciato un thread *Updater* il quale ha il compito di mantenere aggiornati i risultati presentati dalla vista.

Questo thread riceve dal *PipedInputStream* precedentemente connesso i dati letti dall'ESC, sotto forma di coppie di valori *<TelemetryParameter, value>* e ad ogni nuova lettura provvede ad aggiornare il text field corrispondente con la nuova misurazione.

## Studio e testing della libreria *JFreeChart* per i nostri scopi - 19 febbraio 2016

Volendo realizzare una presentazione dei dati sotto forma di grafici iniziamo a cercare e studiare le possibili librerie in grado di fornirci gli strumenti necessari.

La più semplice, utilizzata ed aggiornata risulta essere la libreria *JFreeChart*. Studiandola e provandone per la prima volta le funzionalità ne identifichiamo oggetti e interfacce utili ai nostri scopi:

- ***org.jfree.data.xy.XYSeries***: rappresenta una sequenza di zero o più dati nella forma (x,y). Di default gli item nella serie vengono ordinati in ordine crescente rispetto al valore di x e sono permessi valori duplicati di x. I valori di y possono essere null per rappresentare dati mancanti.
- ***org.jfree.data.xy.XYSeriesCollection***: rappresenta una collezione di XYSeries che può essere utilizzata come dataset in un grafico.
- ***org.jfree.chart.JFreeChart***: un grafico implementato usando le API 2D di Java. Supporta grafici a barre, a linee, a torta e plot xy. Coordina diversi oggetti per poter disegnare correttamente un grafico con Java 2D: una lista (o un solo valore) di oggetti *Title*, un oggetto *Plot*, e un *Dataset*.
- ***org.jfree.chart.ChartPanel***: un componente Swing GUI per la visualizzazione di un oggetto JFreeChart in grafica. Il pannello si registra presso l'oggetto JFreeChart per ricevere la notifica eventi di eventuali cambiamenti nell'oggetto chart e ridisegna contestualmente la grafica.
- ***org.jfree.chart.ChartFactory***: factory per la creazione di certi tipi standard di JFreeChart.
- ***org.jfree.chart.plot.XYPlot***: una classe generica per plottare dati nella forma di coppie (x,y). Può usare qualsiasi classe che implementi XYDataset (come XYSeries). Fa uso di un oggetto XYItemRenderer per disegnare ogni punto sul grafico. Usando diversi tipi di renderers si possono produrre diversi tipi di grafico. La factory ChartFactory possiede metodi statici per creare chart preconfigurate con XYPlot.
- ***org.jfree.chart.renderer.xy.XYLineAndShapeRenderer***: classe che implementa XYItemRenderer. Connette i punti dell'oggetto XYPlot associato disegnando linee e può renderizzare i punti con delle forme.

## Prototipo 5: interfaccia grafica per la visualizzazione dei dati in tempo reale usando JFreeChart - 22 febbraio 2016

Una volta compreso l'uso basilare della libreria JFreeChart passiamo all'uso di essa nel nostro progetto. In particolare desideriamo che similmente a quanto avveniva per SimpleTelemetryView vengano generati dei grafici in funzione del tempo per ogni TelemetryParameter di interesse della routine.

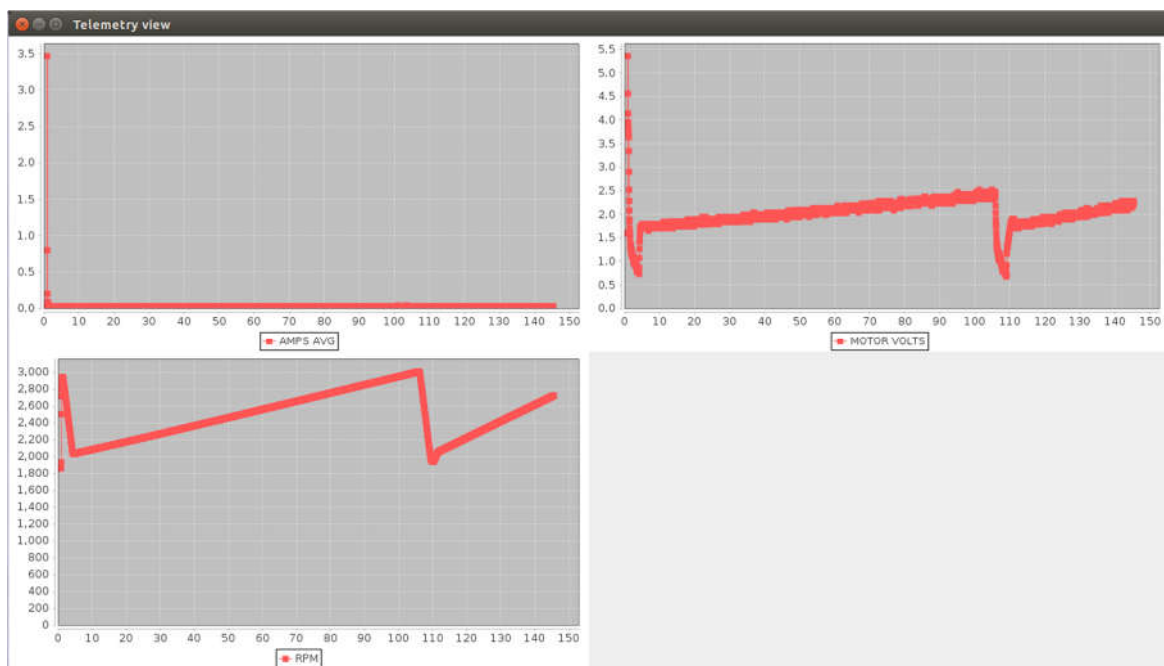
Per ogni TelemetryParameter viene istanziata una XYSeries che si occuperà di mantenere i dataset necessari per la costruzione dei grafici.

Alla generazione della view viene settato un layout a griglia (GridLayout) con abbastanza spazi per contenere un grafico per ogni TelemetryParameter. Ogni foro della griglia viene quindi riempito con un oggetto JFreeChart default istanziato dalla factory ChartFactory, associato alla XYSeries corrispondente. Per migliorare la resa

grafica si specializza l'XYItemRenderer associato all'XYPlot delle JFreeChart come XYLineAndShapeRenderer.

Per l'aggiornamento dei dati riutilizziamo lo stesso thread Updater già descritto nel caso di SimpleTelemetryView, con il compito di ricevere le nuove misurazione e di aggiungere i dati nelle XYSeries dei grafici mantenendo i disegni aggiornati dinamicamente.

Per ottenere la dipendenza dal tempo, ad ogni nuovo arrivo di dato viene associato ad esso un timestamp ottenuto dalla JVM tramite *System.currentTimeMillis()*.



## Prototipo 5: risolto problema di sincronizzazione tra thread - 24 febbraio 2016

Facendo delle prove d'uso del software abbiamo riscontrato un problema nella sincronizzazione tra i diversi thread in gioco. In particolare vi era un problema nella sincronizzazione della istanziazione e connessione degli end point del PipedStream utilizzato per comunicare tra la View ed AutoQuadEsc32.

Per risolvere questo problema abbiamo utilizzato le primitive di sincronizzazione di basso livello offerte dal linguaggio Java ***wait()*** e ***notify()***. Tramite l'uso di questi strumenti abbiamo eliminato le corse critiche che talvolta si verificavano nella connessione del PipedStream.

Abbiamo valutato la possibilità di implementazione di uno strumento di sincronizzazione più avanzato come un monitor per rendere totalmente thread-safe

il software, ma non l'abbiamo ritenuto necessario non avendo riscontrato ulteriori problemi.

## Prototipo 5: conversione a progetto Maven per la gestione delle dipendenze - 25 febbraio 2016

Il progetto comincia a diventare grande in termini di complessità e librerie di terze parti utilizzate. Per questo motivo decidiamo di affidarci a Maven per la gestione delle dipendenze. Come primo approccio a Maven ci documentiamo su quali caratteristiche offra e quali strumenti siano necessari per utilizzarlo.

Maven, principalmente, è uno strumento completo per la gestione di progetti software Java, in termini di compilazione del codice, distribuzione, documentazione e collaborazione del team di sviluppo. Secondo la definizione ufficiale, si tratta di un tentativo di applicare pattern ben collaudati all'infrastruttura del build dei progetti.

Maven è quindi contemporaneamente un insieme di standard, una struttura di repository e un'applicazione che servono alla gestione e la descrizione di progetti software. Esso definisce un ciclo di vita standard per il building, il test e il deployment di file di distribuzione Java.

Tutti i progetti software, indipendentemente dal loro dominio, dalla loro estensione e dalla tecnologia impiegata, presentano una serie di necessità comuni, quali ad esempio:

- compilazione dei sorgenti in codici eseguibili (build)
- verifica (test)
- assemblaggio
- documentazione
- eventualmente il deployment e la relativa configurazione

La feature che a noi interessa è relativa al processo di build, in particolare la gestione delle dipendenze: Maven incoraggia l'utilizzo di repository sia locali sia remoti, per la memorizzazione dei file di distribuzione. Dispone di una serie di meccanismi che permettono di eseguire il download, da un sito globale, di specifiche librerie richieste dal proprio progetto: questo semplifica il riutilizzo degli stessi file .jar da parte di diversi progetti, fornendo anche informazioni necessarie per gestire problemi relativi alla retrocompatibilità

Maven gestisce progetti basati sul proprio modello a oggetti del progetto (POM, Project Object Model). Tale modello è rappresentato in formato XML nel file pom.xml che contiene i meta-dati del progetto, le sezioni per il build, per la gestione delle

dipendenze, per la gestione del progetto in generale, per i test, per la generazione della documentazione, e così via.

Il project object model segue un modello dichiarativo con il vantaggio che in poche righe è possibile compilare, verificare, generare la documentazione e il file di distribuzione di semplici progetti, a differenza di quanto avviene con altri tool a carattere procedurale come Ant.

Eclipse rende semplice la trasformazione di un progetto esistente in un progetto Maven, scrivendone gran parte del POM in maniera automatica.

Esiste una sezione all'interno del file pom.xml, dedicata alla dichiarazione delle dipendenze. Una volta definita una nuova dipendenza nel pom.xml, non è necessario copiare il relativo file in un'apposita directory del progetto, poiché risiederà in un repository e sarà automaticamente caricato da Maven

Per includere le librerie utilizzate è sufficiente aggiungere alle dipendenze di pom.xml:

```
<dependencies>
  <dependency>
    <groupId>org.jfree</groupId>
    <artifactId>jfreechart</artifactId>
    <version>1.0.19</version>
  </dependency>
  <dependency>
    <groupId>org.jfree</groupId>
    <artifactId>jcommon</artifactId>
    <version>1.0.23</version>
  </dependency>
  <dependency>
    <groupId>org.rxtx</groupId>
    <artifactId>rxtx</artifactId>
    <version>2.1.7</version>
  </dependency>
  <dependency>
    <groupId>org.reflections</groupId>
    <artifactId>reflections</artifactId>
    <version>0.9.10</version>
  </dependency>
</dependencies>
```

## Miglioramento affidabilità dei timestamp delle misure - 26 febbraio 2016

Come già descritto in GraphTelemetryView, l'idea iniziale era quella di associare ad ogni nuovo dato in arrivo un timestamp ottenuto dalla JVM. La creazione dei punti <tempo,valore> da introdurre nei grafici avviene quindi lato VIEW, con una evidente dipendenza dalle tempistiche e dai delay di Java e dalla velocità della comunicazione.

Avviando ripetutamente routine di durata definita, notiamo una discrepanza tra i tempi mostrati nei grafici e i tempi che ci aspettiamo realmente.

Per questo motivo decidiamo di modificare sostanzialmente la modalità di creazione ed invio dei dati attraverso il piped stream dall'ESC alla VIEW.

Lato ESC, ad ogni misurazione, viene calcolato un timestamp in base alla frequenza di telemetria e viene preparata una Map (detta *bundle*) contenente entry <TelemetryParameter, Value>, con i valori già correttamente elaborati in base alla classe associata al TelemetryParameter.

Calcolando il timestamp sulla base della frequenza di telemetria eliminiamo la dipendenza dalle tempistiche e dai delay introdotti dalla JVM, rendendo il più possibile coerente la scala dei tempi con la realtà.

Questi dati vengono quindi inviati sul PipedOutputStream e ricevuti dalla VIEW. Localmente alla VIEW i dati vengono estratti dal *bundle* e vengono aggiunti ai grafici corrispondenti i nuovi punti (timestamp, valore).

Notiamo che talvolta vi sono delle letture spurie da parte dell'ESC, soprattutto con frequenze di telemetrie elevate. In particolare, durante la lettura dei byte dal motore possono mancare dati o essere formattati male. Nel caso si verifichi un problema simile, scegliamo di scartare l'intero *bundle* dei dati per evitare di avere dati corrotti nei grafici.

Riportiamo il ciclo di lettura, preparazione ed invio dei dati

```
while (shouldRead.get() == true && (singleData = (byte)
input.read()) != -1) {
    inputBuffer.write(singleData);

    if (singleData == '\n') {
        ByteArrayInputStream bin =
            new ByteArrayInputStream(inputBuffer.toByteArray());
        BufferedReader reader =
new BufferedReader(new InputStreamReader(bin, "UTF-8"));
        String line = reader.readLine();
        String[] tokens = line.split("\\s{2,}");
        TelemetryParameter p = null;
        // se c'è almeno un token e se ha parsato la prima parte
        // della stringa come
        // parametro e questo parametro ci interessa
        if (tokens.length != 0
            && (p=TelemetryParameter.parse(tokens[0]))!=null
            && telemetryParameters.contains(p)) {
```

```

        Object value = null;
        try {
            if (p.valueClass == String.class)
                value = tokens[1];
            else if (p.valueClass == Integer.class)
                value = Integer.parseInt(tokens[1]);
            else if (p.valueClass == Double.class)
                value = Double.parseDouble(tokens[1]);
        } catch (NumberFormatException
                | ArrayIndexOutOfBoundsException ignore) {
            // c'è stato un errore di lettura, succede spesso
            con valori alti di telemetria
            System.out.println(line + " " +
                Arrays.toString(tokens));
            ignore.printStackTrace();
        } finally {
            bundle.put(p, value);
        }

        // il bundle è riempito, mando insieme al timestamp
        if (bundle.size() == telemetryParameters.size()) {
            writer.writeDouble(time);
            writer.writeObject(bundle);
            bundle = new HashMap<>();
            // period = 1.0 / telemetryFrequency;
            time += period;
        }
    }
    inputBuffer.reset();
}
}

```

## Esportazione dei dati in formato csv, analisi preliminare in Office - 29 febbraio 2016

Il passo successivo consiste in iniziare ad effettuare analisi sui dati raccolti. A questo scopo modifichiamo `GraphTelemetryView` in modo tale che alla ricezione delle misurazioni, oltre ad aggiornare i grafici, provveda a inserire questi dati in un file .csv, similmente a quanto già fatto nel Prototipo 1.

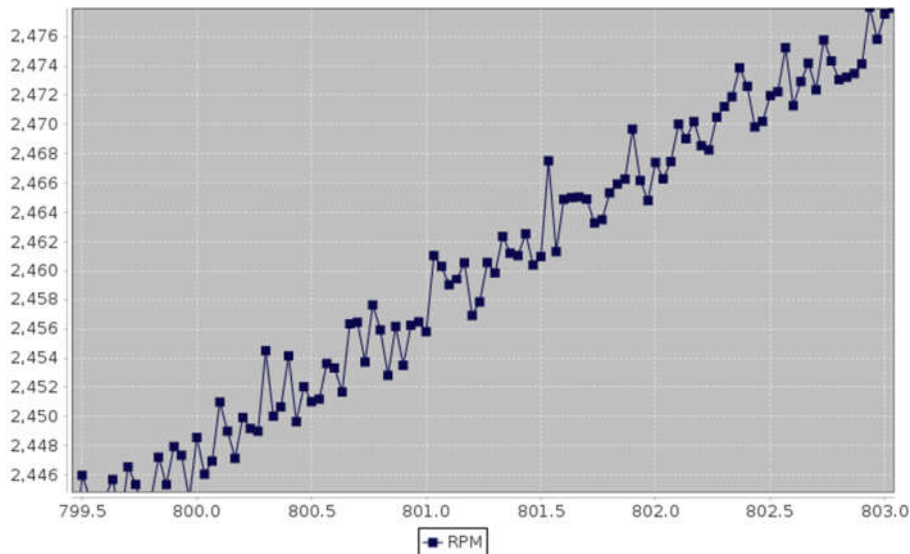
A differenza della prima implementazione però, abbiamo un modello di dati più completo con associata una scala di tempi che ci consente analisi statistiche sui dati.

Elaborando il file .csv come foglio di calcolo, proviamo a fare le prime osservazioni sui dati:

- imponendo al motore un'accelerazione moderatamente bassa, contrariamente a quanto atteso, le velocità non sono strettamente crescenti, ma si verificano



diverse letture per le quali la velocità risulta più bassa della precedente. Nonostante questo, la velocità risulta essere complessivamente in crescita.



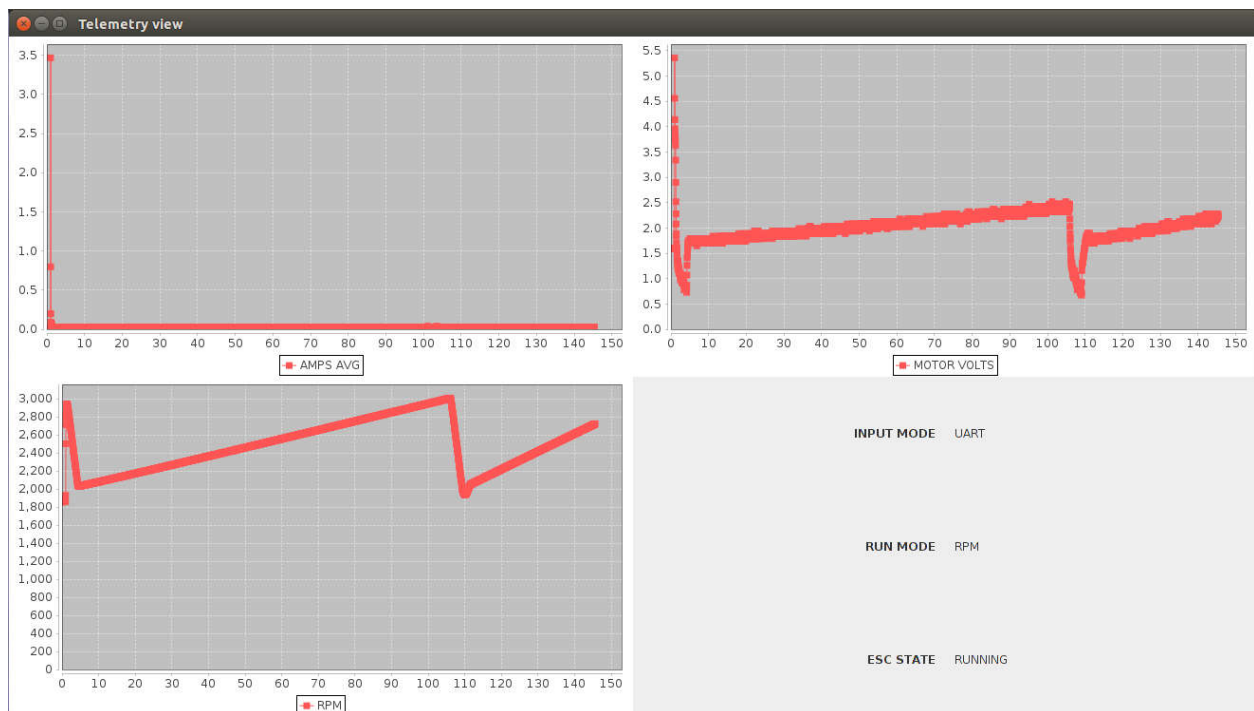
- la corrente non rimane costante su tutti i range di rpm e di accelerazioni. Essendo la corrente costante un'ipotesi necessaria per il calcolo dei parametri del motore che chiuderà questo progetto, sarà necessario trovare dei range di valori nei quali ciò si verifica.

## Aggiornamento di codice deprecato e piccoli miglioramenti - 1 marzo 2016

SimpleTelemetryView è rimasta al modello di trasmissione dei dati precedente al Bundle, nonostante sia stata ora completamente sostituita dalla GraphTelemetryView vogliamo aggiornarne il codice per mantenere la compatibilità con l'applicazione. Sono sufficienti poche modifiche per rendere nuovamente SimpleTelemetryView intercambiabile con la ben migliore GraphTelemetryView.

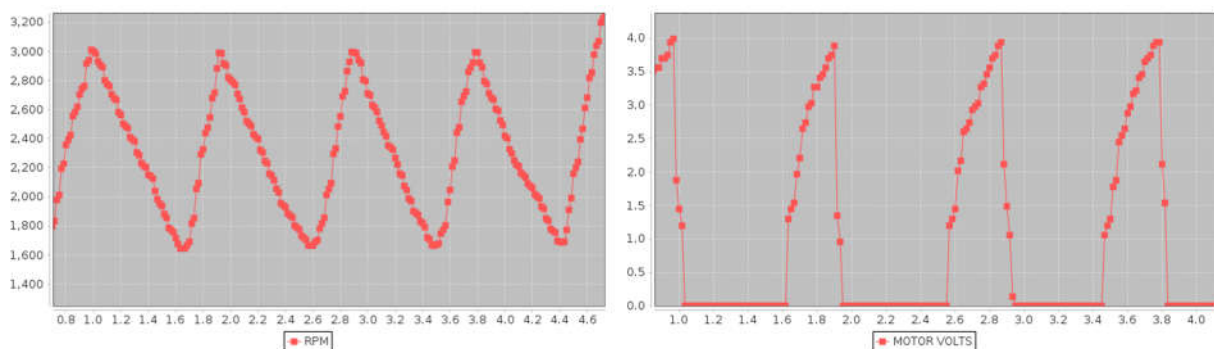
Finora, in GraphTelemetryView, abbiamo scelto di occuparci solo dei valori della telemetria numerici e quindi rappresentabili graficamente. Andiamo quindi ad aggiungere un nuovo pannello per la presentazione dei dati in forma testuale, che riprende sostanzialmente quanto già visto per SimpleTelemetryView. In particolare vengono presi in esame tutti quei TelemetryParameter di interesse della routine ai quali è associato un valore di tipo String, e vengono presentati in una tabella generata ed aggiornata dinamicamente.

Il risultato ottenuto è simile al seguente:



## Testing del motore su diverse routines - 2 marzo 2016

Avendo a disposizione uno strumento di definizione ed esecuzione di routines personalizzate possiamo ora facilmente eseguire diversi test sul motore.



Osservando i grafici relativi a rapide accelerazioni e decelerazioni in successione, abbiamo notato che c'è un limite alla decelerazione di circa -400 rpm/s. Per cambiamenti più rapidi si nota che l'esc pone a 0V i motor volts e non riesce ad ottenere la decelerazione richiesta.

Per quanto riguarda l'accelerazione questa sarà limitata superiormente dal valore per il quale l'esc mette i motor volts a 15V (tensione di alimentazione), ma questa accelerazione si trova al di sopra di quelle comunemente utilizzate nei nostri test e non abbiamo mai riscontrato il problema.

In riferimento al problema individuato in precedenza relativo alla non costanza della corrente durante le accelerazioni ci siamo impegnati ad individuare sperimentalmente dei range validi per l'analisi. Si osserva che per mantenere la corrente che scorre nel motore costante, è necessario mantenere velocità angolari e accelerazioni angolari basse, in particolare non superare i 3000 rpm in velocità e i 60 rpm/s di accelerazione.

La routine migliore individuate è composta da una serie di 5 accelerazioni tra 2000 e 3000 rpm, con vari profili di accelerazione costante (da 10 a 50 rpm/s), ripetute per due volte. Tra ogni accelerazione e la successiva vi è una decelerazione tra 3000 e 2000, alla massima rapidità consentita dal motore, influente per il computo dei dati.

### Progettazione del package Analyzer - 3, 4 marzo 2016

L'obiettivo iniziale del progetto di tirocinio era permettere l'analisi dei dati della telemetria a fronte di una specifica routine e con il fine di individuare specifici parametri del motore. Avendo però realizzato, per desiderio di completezza, un software in grado di eseguire routines arbitrarie riteniamo limitante la possibilità di analizzare i dati secondo un'unica metodologia predefinita. Ci impegniamo quindi per realizzare un'astrazione del processo di analisi dei dati, che utilizzeremo poi per subclassare l'analizzatore vero e proprio richiestoci dal tirocinio.

Individuiamo come necessità comuni a qualsiasi tipo di analisi si voglia progettare le seguenti:

- avere accesso ai dati della telemetria, in forma completa o almeno ai parametri necessari per la specifica analisi
- ulteriori parametri e costanti per i calcoli, caricabili file per comodità di riesecuzione di un'analisi oppure specificabili sul momento
- produrre un risultato sintetico visualizzabile direttamente a video
- produrre risultati intermedi e più dettagliati sotto forma di ulteriori file

Queste caratteristiche rappresentano la visione "ai morsetti" di una generica analisi e per questo motivo confluiranno nella classe astratta Analyzer.

Vogliamo che l'utente finale possa essere in grado di definire nuove metodologie di analisi in maniera semplice estendendo la classe Analyzer, con la possibilità di caricare le librerie user-defined a runtime senza necessità di ricompilare il progetto di base. Questo pattern coincide con quanto già fatto per i modelli di ESC e dunque realizzeremo questa funzionalità sfruttando ancora una volta la reflection.

La classe astratta Analyzer si basa principalmente su tre mappe per indicare di quali dati di telemetria e parametri aggiuntivi ha bisogno e quali risultati produrrà in output. Abbiamo deciso per semplicità di limitare la tipologia dei parametri e dell'output a valori numerici, ritenendo che non costituisca una limitazione troppo pesante visto il contesto scientifico in cui verrà utilizzato un Analyzer.

Dal momento in cui le sottoclassi specificano nelle tre mappe i dati che gestiranno, la classe astratta mette a disposizione metodi per effettuare il caricamento dei dati e dei parametri rispettivamente da file .csv e .properties. In questo modo si alleggerisce molto il carico di lavoro per uno sviluppatore che deve implementare un Analyzer in quanto tutta la parte di parsing di file è già presa in carico dalla classe astratta.

Lo sviluppatore dovrà unicamente occuparsi di realizzare il metodo calcola() con la sicurezza che al momento dell'invocazione le mappe degli input sono già state correttamente riempite con i dati necessari e ricordandosi alla fine dell'esecuzione di settare i risultati nelle mappe di output.

Di seguito è riportato lo scheletro della classe Analyzer e nei prossimi paragrafi si mostrerà l'esempio di implementazione relativo all'analisi richiesta per il tirocinio.

```
public abstract class Analyzer {
    protected Map<String, List<Double>> table;
    public Map<String, Double> parametersRequired;
    public Map<String, Double> results;

    public abstract void calcola();

    protected Analyzer(File dataFile, File propertyFile) {...}
    protected void loadParameters() {...}
    protected void readDataFromFile() throws IOException,
        FileFormatException {...}
}
```

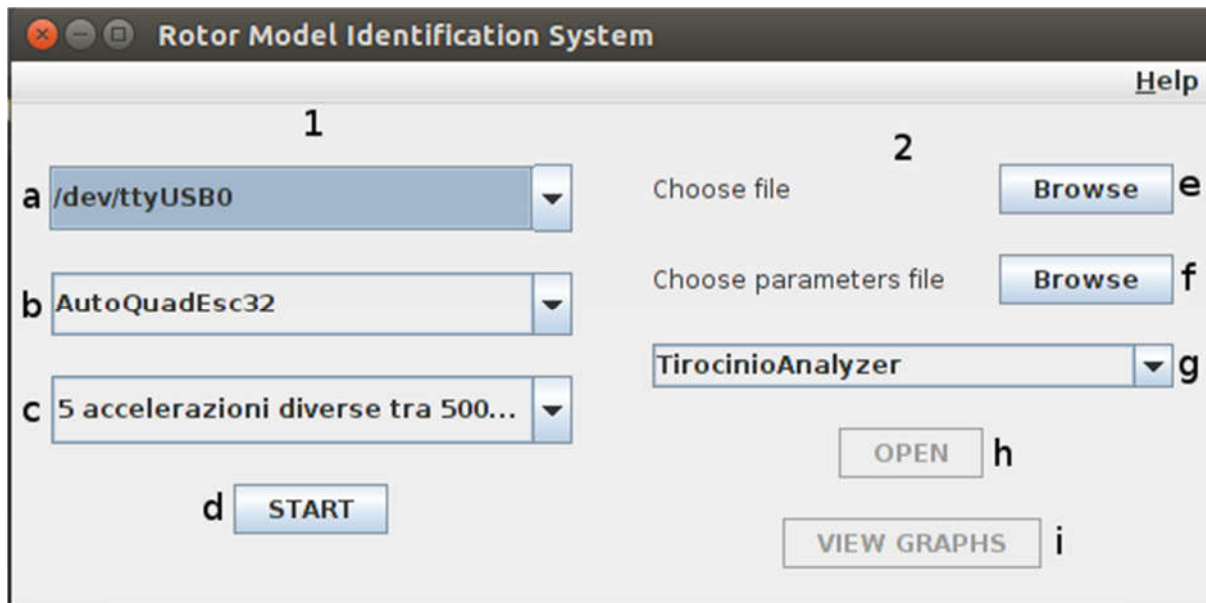
## Interfaccia grafica per l'analisi - 7, 9 marzo 2016

Con l'aggiunta del package Analyzer è necessario rimodellare notevolmente la GUI per presentare all'utente le nuove funzionalità offerte. L'idea è quella di ottenere un'unica vista principale in grado di presentare la selezione e l'avvio delle funzioni offerte dai due package Esc e Analyzer. Per fare ciò realizziamo due pannelli, denominati LeftPanel e RightPanel:

- 1. LeftPanel:** all'interno di questa vista sono mostrate tutte le funzionalità che riguardano le Routines e gli ESCs, riprendendo sostanzialmente quanto già fatto con EscSelectorGUI e la precedente MainView. In particolare consiste di:

- a. Una prima JComboBox che permette di selezionare la porta seriale da utilizzare, ottenendo la lista delle porte disponibili tramite il metodo statico *PortSelector.scanPorts()*.
  - b. Una seconda JComboBox che permette di selezionare quale implementazione di ESC si desidera utilizzare. Per mostrare la lista degli Esc implementati disponibili utilizza il metodo statico *EscLoader.getEscsList()* che come già visto utilizza introspezione e reflection.
  - c. Una terza ed ultima JComboBox permette di selezionare la routine da eseguire. Le routine disponibili vengono caricate e parsate dinamicamente dalla cartella */routines*.
  - d. Un JButton START che permette di lanciare la routine selezionata sull'Esc desiderato. In particolare alla pressione del tasto viene invocato il metodo *getConnectedEsc()* il quale crea un'istanza dell'Esc voluto connesso alla porta voluta, tramite il metodo *EscLoader.newInstanceOf(Class<? extends AbstractEsc>, SerialPort)*. Se la creazione e connessione dell'esc è andata a buon fine, viene eseguita la routine, con la contestuale visualizzazione dell'andamento sulla vista *GraphTelemetryView*.
2. **RightPanel:** all'interno di questa vista invece, sono mostrate tutte le opzioni riguardanti l'analisi dei file di log della telemetria. In particolare consiste di:
- e. Un JFileChooser che permette di selezionare un file .csv contenente i dati da analizzare relativi a una precedente esecuzione di una routine.
  - f. Un secondo JFileChooser che permette di scegliere un file .properties (facoltativo) contenente eventuali parametri necessari per il funzionamento dell'Analyzer desiderato.
  - g. Una JComboBox che, come fatto per gli Esc, permette di selezionare l'implementazione di Analyzer da utilizzare tramite il metodo statico *AnalyzersFactory.getAnalyzersList()*
  - h. Un bottone OPEN alla pressione del quale viene creata un'istanza dell'Analyzer selezionato e viene generata una nuova vista *AnalyzerFrame* per la presentazione dei risultati dell'analisi.
  - i. Un bottone VIEWGRAPH alla pressione del quale viene generata una nuova vista *GraphFrame* per la ricostruzione dei grafici di una run precedente a partire dal file .csv selezionato.

Integriamo poi entrambi i pannelli in un'unica vista, denominata MainFrame, ottenendo questo risultato finale:



Per la presentazione dell'output della telemetria durante l'esecuzione di una routine si continua ad utilizzare GraphTelemetryView. Per quanto riguarda la parte di analisi, quindi ricostruzione dei grafici di una run precedente e presentazione dei risultati realizziamo le due nuove viste AnalyzerFrame e GraphFrame.

### AnalyzerFrame:

Questa classe rappresenta un frame nel quale vengono richiesti i dati richiesti dal particolare tipo di Analyzer per l'elaborazione dei dati e presentati i risultati dell'elaborazione. La vista consiste di due pannelli ed un bottone.

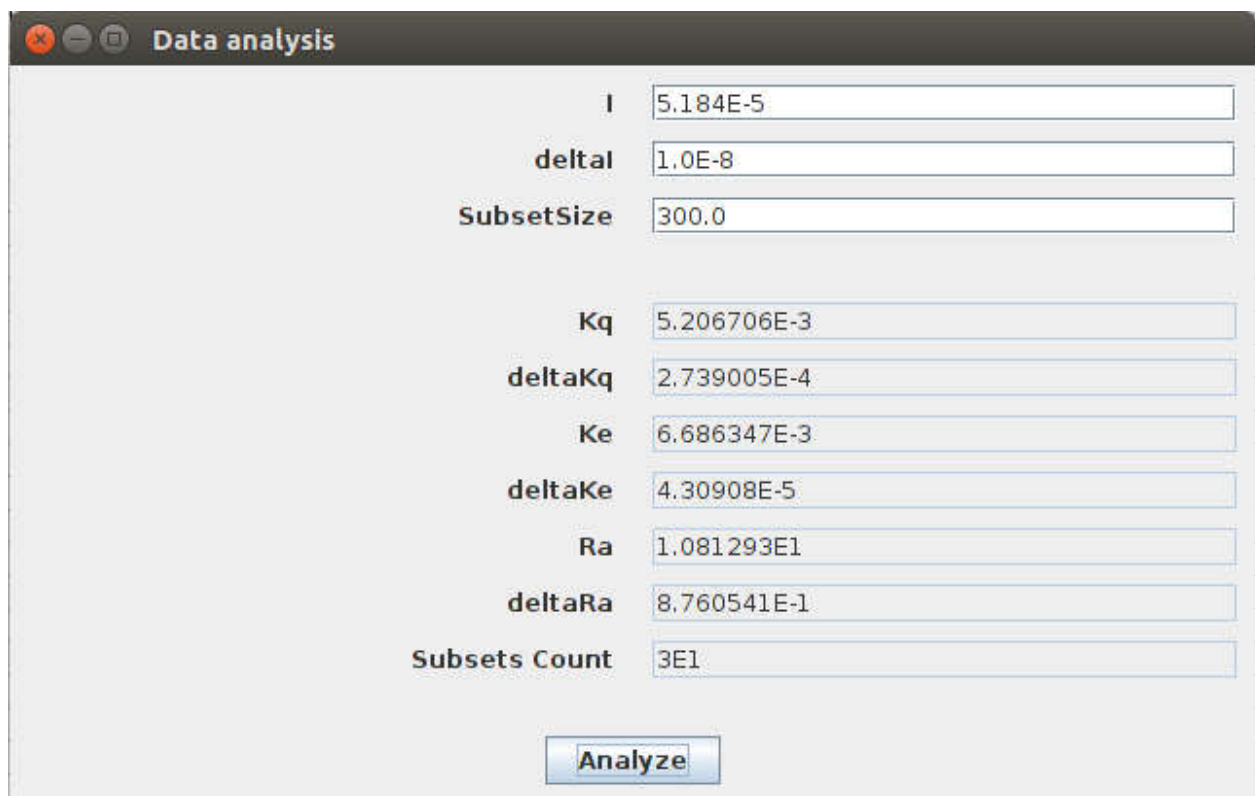
Il primo pannello, che chiamiamo parametersPanel, utilizza un GridLayout la cui dimensione dipende dal numero di parametri specifico richiesto dall'implementazione di Analyzer (quindi dalla dimensione della mappa dei parametri dell'istanza di Analyzer). In particolare la colonna sinistra della griglia viene popolata con i nomi assegnati ai parametri e la colonna destra è riempita con delle JTextField modificabili, eventualmente già riempite con i dati ottenuti dal file .properties specificato nel MainFrame.

Il secondo pannello, che chiameremo resultsPanel, utilizza anch'esso un GridLayout la cui dimensione dipende dal numero di risultati che verranno elaborati (quindi dalla dimensione della mappa dei risultati dell'Analyzer). Come sopra, la parte sinistra è riempita con le etichette dei risultati, e la parte destra da JTextField non editabili in cui verranno presentati i valori calcolati.

Si noti che, facendo le cose in questo modo i due pannelli avranno sempre le dimensioni e la presentazione corretta, poiché generati dinamicamente a partire dall'implementazione di Analyzer scelta.

Tra il primo e il secondo pannello si trova un bottone ANALYZE alla pressione del quale i valori dei parametri sono prelevati dalle text box e assegnati all'istanza di Analyzer e ne avviene l'elaborazione, con il seguente riempimento dei campi dei risultati. Nel caso in cui si verifichi qualche problema di parsing o di altra natura, viene visualizzato un messaggio di errore.

Il risultato finale è di questo tipo:



The screenshot shows a window titled "Data analysis" with a standard Mac OS X title bar (red, yellow, and green buttons). The window contains several input fields for parameters and their corresponding results. The parameters are listed on the left, and the results are in text boxes on the right. At the bottom, there is an "Analyze" button.

Parameter	Value
I	5.184E-5
deltaI	1.0E-8
SubsetSize	300.0
Kq	5.206706E-3
deltaKq	2.739005E-4
Ke	6.686347E-3
deltaKe	4.30908E-5
Ra	1.081293E1
deltaRa	8.760541E-1
Subsets Count	3E1

At the bottom center is a button labeled "Analyze".

**GraphFrame:** in quest'ultima vista viene ricostruita graficamente la precedente esecuzione di una routine rappresentata dal file .csv selezionato nel MainFrame. La resa è in tutto e per tutto identica a quanto si ottiene con GraphTelemetryView. Essendo una funzionalità piccola e a sè stante abbiamo preferito realizzare l'intera logica applicativa all'interno della classe stessa, riutilizzando ovviamente il codice già scritto in GraphTelemetryView.

Dal punto di vista grafico fino a questo momento ci siamo accontentati dell'implementazione standard delle JComboBox. Le quali, passata una lista di oggetti da visualizzare, sfruttano per ognuno il metodo toString per ottenere una stringa da mettere in elenco. Per quanto questo problema sia facilmente aggirabile cambiando l'implementazione di default del toString delle classi da noi definite, non è possibile

agire in questo modo per classi pre-esistenti come `CommPortIdentifier` e `Class`. È inoltre sintomo di cattiva programmazione mischiare quella che è la descrizione del modello (la classe) con la sua rappresentazione grafica (il metodo `toString`).

Per questo motivo scegliamo di realizzare due componenti aggiuntivi in grado di presentare graficamente le istanze degli oggetti, mantenendo così ben separati model e view. Le `JComboBox` fanno affidamento all'interfaccia `ListCellRenderer` per disegnare un oggetto al loro interno. Come già detto sopra, l'implementazione in uso di default non fa altro che invocare `toString` dell'oggetto. Sta a noi quindi reimplementare `ListCellRenderer` per adattarla ai nostri scopi e in seguito impostare le `JComboBox` affinché usino la nostra implementazione.

Riportiamo l'implementazione relativa al rendering degli oggetti di tipo Classe, usata dunque nelle liste che presentano gli ESC e gli Analyzer disponibili:

```
public class CustomClassRenderer extends JLabel implements
    ListCellRenderer<Class<?>> {
    public CustomClassRenderer() {
        setOpaque(true);
        setHorizontalAlignment(LEFT);
        setVerticalAlignment(CENTER);
    }
    public Component getListCellRendererComponent(
        JList<? extends Class<?>> list, Class<?> value, int index,
        boolean isSelected, boolean cellHasFocus) {
        if (isSelected) {
            setBackground(list.getSelectionBackground());
            setForeground(list.getSelectionForeground());
        } else {
            setBackground(list.getBackground());
            setForeground(list.getForeground());
        }
        setText(value != null ?
            value.getSimpleName() : "No classes available");
        return this;
    }
}
```

Evitando di addentrarsi nei dettagli dei metodi si noti solamente come questo renderer sfrutti una `JLabel` per scrivere al suo interno il nome semplice della classe da rappresentare, modificandone inoltre l'aspetto grafico in base allo stato della selezione. Nel caso in cui non ci siano implementazioni disponibili presenta un messaggio personalizzato all'utente.



## Implementazione Analyzer usando Apache Math commons - 11 marzo 2016

Per ottenere il modello matematico del rotore come da obiettivo finale, abbiamo poi realizzato un'implementazione di Analyzer ad hoc.

Abbiamo quindi realizzato una classe *TirocinioAnalyzer* che ha il compito di eseguire l'analisi dei dati e restituire i tre parametri risultanti già descritti precedentemente.

Per l'analisi matematica abbiamo utilizzato la libreria matematica open source Apache Math commons.

In particolare della libreria abbiamo utilizzato le seguenti classi:

- ***org.apache.commons.math3.stat.regression.SimpleRegression***: che permette di effettuare la regressione lineare su un set di dati e di ottenere coefficiente angolare e ordinata all'origine e i relativi errori tramite i metodi *getSlope()*, *getIntercept()*, *getSlopeStdErr()* e *getInterceptStdErr()*;
- ***org.apache.commons.math3.stat.descriptive.moment.Mean***: per il calcolo della media aritmetica di un set di dati.
- ***org.apache.commons.math3.stat.descriptive.moment.StandardDeviation***: che permette di calcolare la deviazione standard di un set di valori.

Per calcolare i coefficienti  $K_e$ ,  $K_q$  ed  $R_a$  *TirocinioAnalyzer* compie i seguenti passi:

1. Identifica i subset di velocità angolari per i quali si ha derivata positiva (pertanto accelerazione positiva)
2. Per ogni subset effettua il calcolo dei coefficienti:
  - a. Calcolo dell'accelerazione angolare con regressione lineare sugli rpm
  - b. Calcolo della coppia applicata dal motore come prodotto tra accelerazione angolare ed inerzia
  - c. Calcolo della corrente media
  - d. Calcolo di  $K_q$  come rapporto tra coppia applicata e corrente media
  - e. Calcolo di  $K_e$  ed  $R_a$ , rispettivamente come coefficiente angolare (*slope*) e rapporto tra intercetta e corrente media della regressione lineare tra tensione e velocità angolare

## Cleanup codice, risoluzione bug, piccole migliorie - 14, 16 marzo 2016

Avviandoci verso la conclusione del progetto, provvediamo ad apportare gli ultimi ritocchi e a fare una pulizia generale del codice.

- Settaggio della cartella di output del compilatore Maven a *target/classes* per rispettare lo schema delle cartelle definito dall'archetipo di progetto Java.
- Cambiati i titoli delle routine in nomi più significativi in grado di descrivere cosa faccia quella routine anche senza vederne il codice

- Aggiunto un menù di *About* nella vista principale della grafica per indicare gli sviluppatori e il contesto di sviluppo (CASY - Università di Bologna)
- Cambiata la formattazione dei risultati in notazione scientifica per favorirne la leggibilità
- Clean up del codice a mano (nomi dei metodi e dei parametri più significativi, separazione dei blocchi di codice in base alla loro funzione eccetera) e con la utility integrata in Eclipse (regole di indentazione, spaziatura, punteggiatura, riordinamento dei metodi eccetera)
- Aggiunta la generazione di colori casuali nella rappresentazione dei grafici per renderli più piacevoli alla vista
- Rimozione di codice deprecato o non più utilizzato, prima mantenuto come riferimento ma ora non necessario

## Ultimazione documentazione, grafici uml - 17 marzo 2016

Durante la stesura del codice abbiamo cercato il più possibile di mantenerlo ben documentato per ragioni di manutenibilità e facilità di riutilizzo.

In questa giornata verifichiamo nuovamente la documentazione prodotta, miglioriamo le parti meno chiare e integriamo quelle mancanti. Infine con i tool di generazione automatica della javadoc costruiamo nella cartella doc l'ipertesto completo delle nostre classi.

Inoltre con delle utility di software engineering generiamo i grafici UML del codice scritto in modo da rendere più immediata la comprensione del funzionamento del software a futuri ed eventuali sviluppatori che volessero estenderlo.

## Readme per utilizzatori, relazione ai tutor - 18 marzo 2016

In queste ultime ore, realizziamo un readme per i futuri utilizzatori e una breve relazione sulle modalità e i risultati ottenuti da consegnare e presentare ai tutor come conclusione del lavoro svolto. Utilizziamo la sintassi markdown (estensione .md) per allinearci alle linee guida dei progetti open source di GitHub.

Alleghiamo i documenti per completezza.