Drone mission

Carlo Antenucci, Leonardo Iannacone, Gonzalo Junquera

15 gennaio 2014

Indice

Elenco degli algoritmi

Capitolo 1

Introduzione

Capitolo 2

Visione

Lo sviluppo di un prodotto software necessita di un processo di produzione maturo, che, al fine di garantire un'elevata qualità e produttività, necessita di una opportuna organizzazione. Per migliorare il processo produttivo il *Software Engineer Institute* (`SEI`) ha introdotto il sistema *Capability Maturity Model* (`CMM`). Tale sistema suddivide le organizzazioni in cinque fasi:

Livello 1: *Initial* (Chaotic): i processi sono ad-hoc, caotici, o pochi processi sono definiti

Livello 2: *Repeteable*: i processi di base sono stabiliti e c'è un livello di disciplina a cui attenersi in questi processi

Livello 3: *Defined*: tutti i processi sono definiti, documentati, standardizzati ed integrati a vicenda

Livello 4: *Managed*: i processi sono misurati raccogliendo dati dettagliati sui processi e sulla loro qualità

Livello 5: *Optimized*: è in atto il processo di miglioramento continuo tramite feedback quantitativi e la fornitura di linee guida per nuove idee e tecnologie

La costruzione di un software, inoltre, è spesso legata alle piattaforme operative su cui il prodotto dovrà operare, che in ogni caso hanno una espressività molto maggiore della *Macchina di Minsky*. Per questo motivo, solitamente, le organizzazioni tendono ad utilizzare approcci diversi per la produzione del software. Le possibili strategie prevedono:

- l'elaborazione di una soluzione partendo da un'analisi del problema, che porta alla stesura di un codice ad hoc per quel determinato contesto (*Top Down*)
- lo sviluppo di una soluzione utilizzando le funzionalità messe a disposizione di una tecnologia (*Bottom Up*)

- la realizzazione di un modello del sistema software da realizzare in modo tale da rendere il prodotto che si sta sviluppando indipendente dalla tecnologia e, allo stesso tempo, riutilizzabile in più contesti (*Model Driven Software Development*)

Le figure professionali che entrano in gioco all'interno di un processo di produzione software sono principalmente tre:

Project manager è colui che coordina lo svolgimento del progetto. Avvalendosi di consulenze tecniche prenderà decisioni in merito alle risorse necessarie per il progetto e distribuirà i compiti agli altri due soggetti in gioco definendo cosa dovrà essere realizzato e come.

System designer è colui che specifica cosa il sistema software deve essere in grado di fare. Il suo compito è quello di specificare la struttura del sistema, i suoi componenti, le sue interfacce ed i suoi moduli. Nell'approccio *Model Driven Software Development* il lavoro che svolge consiste nel modellare le entità del sistema su tre dimensioni: struttura, interazione e comportamento.

Application designer è colui che specifica come le entità descritte dal system designer interagiscono e si comportano al fine di ottenere quanto richiesto dal committente. Il suo compito è, quindi, quello di definire, utilizzando gli strumenti messi a disposizione dalla tecnologia scelta, ed eventualmente dal system designer stesso, la business logic del sistema software.

Il compito del system designer, quindi, è quello di realizzare un modello concettuale del sistema, definendo come detto le entità che entrano in gioco, le loro interazioni e il loro comportamento, e fornire all'application designer un meta-modello dello stesso.

L'idea di utilizzare un approccio Model Driven, anziché uno Top Down o Bottom Up, consente, quindi, lo sviluppo di un prodotto software, non legato in maniera eccessiva alla tecnologia alla base del sistema, né tanto meno alla business logic. Questo garantisce al sistema sviluppato un alto grado di riutilizzabilità in quanto, una volta definito il modello, sarà sufficiente modificare il comportamento o l'interazione dei componenti per far si che questo si adatti ad un nuovo problema. Inoltre l'approccio Model Driven consente di formalizzare ed esplicitare, attraverso la costruzione di una serie di diagrammi che non lasciano spazio ad ambiguità, le conoscenze utili alla risoluzione del problema da risolvere.

Capitolo 3

Obiettivi

L'applicazione intende fornire alla protezione civile un maggiore supporto per l'esplorazione territoriale senza la necessità di mettere a repentaglio vite umane.

Capitolo 4

Requisiti

4.1 Lo scenario applicativo

La protezione civile decide di inviare su un luogo diffcilmente accessibile un aeromobile senza pilota (*drone*), capace di operare in modo teleguidato. Il drone è dotato di un insieme di *sensori di stato* in grado di rilevare la velocità corrente (*speed*) e il carburante disponibile (*fuel*). Il drone dispone anche di un dispositivo *GPS* in grado di determinarne la posizione in termini di latitudine e longitudine.

Il compito del drone è scattare fotografie del territorio ogni DTF (DTF>0) secondi e inviare le immagini a un server installato presso una unità operativa. Il server provvede a memorizzare le immagini ricevute (in un file o in un database) associandole ai dati dei sensori di stato disponibili al momento dello scatto della foto. Il server provvede inoltre a visualizzare su un display dell'unità operativa i valori di stato ricevuti dal drone in una dashboard detta *DroneControlDashboard*.

La *DroneControlDashboard* viene concepita come un dispositivo composto di due parti: una parte detta *GaugeDisplay* e una parte detta *CmdDisplay*. La parte *GaugeDisplay* della *DroneControlDashboard* visualizza i dati provenienti dai sensori del drone riconducendoli ciascuno a uno specifico strumento di misura; uno *Speedometer* (velocità in km/h) un *Odometer* (numero di km percorsi) un *FuelOmeter* (livello corrente di carburante in litri) e un *LocTracker* (posizione del drone). La *GaugeDisplay* può visualizzare i dati in forma digitale e/o grafica; la posizione viene preferibilmente visualizzata fornendo una rappresentazione del drone su una mappa del territorio. La parte *CmdDisplay* della *DroneControlDashboard* include pulsanti di comando per fissare la valocità di crociera (*setSpeed*) avviare (*start*) e fermare (*stop*) il drone[1] e per incrementarne (*incSpeed*) e decrementarne (*decSpeed*) la velocità corrente di una quantità prefissata DS (DS>0 km/h).

[1] Il drone si suppone abbia un sistema di controllo capace di eseguire i comandi di *start* e di *stop* in modo opportuno.

I dati dei sensori del drone sono anche resi disponibili sugli smart device in dotazione al responsabile della protezione civile (*Chief*) e al comandante (*Commander*) della unità operativa. Ogni smartdevice provvederà a visualizzare (su richiesta dell'utente) i dati in una dashboard (*SmartDeviceDahboard*) opportunamente definita per lo specifico dispositivo, preferibilmente in modo analogo alla *GaugeDisplay*.

Il server deve operare in modo che :

- la missione del drone possa iniziare solo dopo che il drone ha dato conferma della ricezione del comando *setSpeed* che fissa la velocità iniziale di crociera;
- la speed del drone sia sempre compresa tra due valori-limite prestabiliti `speedMin` a `speedMax`[2];
- all'avvio di ogni missione, ogni smartdevice `Android` sia messo in grado di generare una *notification* all'utente, la cui selezione provvede ad aprire una applicazione che mostri la *SmartDeviceDahboard*.
- gli smartdevice siano in grado di visualizzare lo stato del drone anche in caso di guasto del server centrale.
- il comando di stop sia inviato in modo automatico non appena il livello del carburante risulta inferiore a un livello prefissato `MinFuel`.

4.2 Il lavoro da svolgere

In questo quadro, si chiede di definire il software da installare sul server della unità operativa e su smartdevice dotati di sistema operativo Android[3]. Opzionalmente: si chiede di definire uno strumento capace di visualizzare le informazioni memorizate dal server dopo una missione del drone. Si chiede anche di costruire un opportuno simulatore delle attività del drone con riferimento ai seguenti parametri:

```
          Parametri per la simulazione del drone
DTF=5 sec, DS=10 km/h, livello fuel iniziale = 30 litri
livello minimo fuel per operatività:  MinFuel = 0,5 litri
speed di crociera compresa tra:  speedMin=60 e speedMax=120 km/h
consumo di carburante = (speed * 30) litri/h
percorso del drone:  in linea retta a una quota fissa di 100m.
```

4.3 Remark

Si ricorda che l'obiettivo del lavoro non è solo la produzione di un sistema software in grado di soddisfare i requisiti funzioniali ma anche (e in primis) il rapporto tra il prodotto e il processo adotttato per generarlo.

[2] Le fasi di decollo e atterraggio sono qui ignorate.

[3] Per il primo protoipo lo smartdevice può essere un computer convenzionale.

Capitolo 5

Analisi dei requisiti

5.1 Use cases

Dalle specifiche del committente si è capito che questi desidera il sistema fornisca tre funzionalità principali: ricezione di informazioni territoriali, controllo della missione e ricezione di informazioni relative ai sensori di stato del drone.

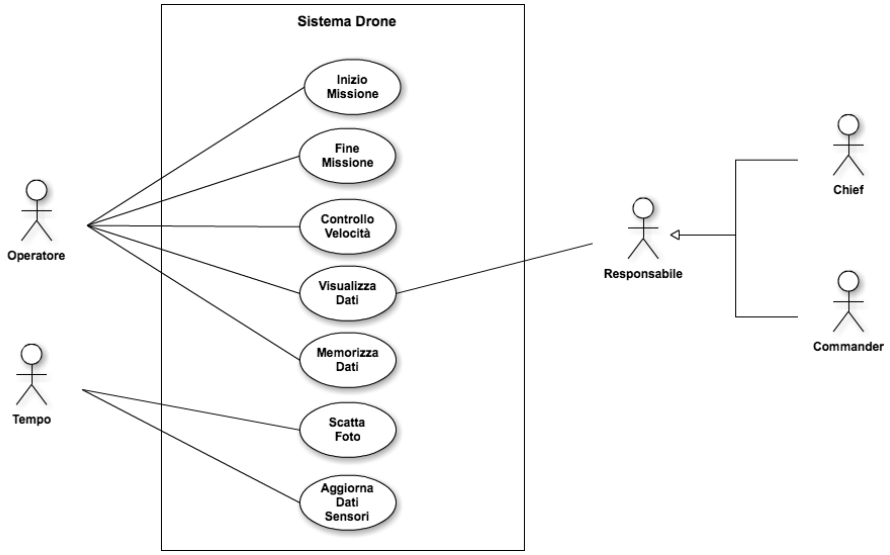

Figura 5.1: Use case

5.2 Glossario

Termine	Significato
Centrale Operativa	**Elemento esterno al sistema da sviluppare. Ha il compito di controllare la missione, ricevere dal drone informazioni relative ai suoi sensori e memorizzare le fotografie scattate.**
Smartphone	**Elemento esterno al sistema. Consente al Chief e al Commander di avere informazioni sullo stato del drone.**
Drone	**Elemento esterno al sistema. È un velivolo privo di pilota che ha il compito di esplorare un territorio difficilmente accessibile, di comunicare i dati relativi ai suoi sensori e di inviare ad intervalli ditempo regolari fotograrfie dell'ambiente esplorato.**
Fotografie territoriali	**Immagine jpg, acquisite dal drone tramite una fotocamera, che riposta informazioni relative alle condizioni ambientali del luogo esplorato e chela centale operativa provvederà a memorizzare.**
Sensori	**Elementi attivi del sistema. Inviano alla centrale operativa informazioni sullo stato del drone, quali chilometri percorsi, velocità attuale, quantità di carburante residuo e le coordinate geografiche del punto in cui si trova.**
DroneControlDashboard	**Elemento del sistema che consente alla centrale operativa di visualizzare le informazioni ricevute dal drone (GaugeDisplay) e, allo stesso tempo, di inviare al velivolo comandi (CmdDislpay).**
GaugeDisplay	**Componente della DroneControlDashboard che consente la visualizzazione delle inforamzioni del drone. È composta da una mappa su cui viene visualizzata la sua posizione e da tre strumenti di misura che riportano i dati rilevati dei sensori, sia in forma analogica che digitale.**
Odometer	**Strumento di misura che consente la visualizzazione del numero di chilometri percorsi dal drone.**
Speedometer	**Strumento di misura che consente la visualizzazione della velocità attuale del drone**
FuelOmeter	**Strumento di misura che consente la visualizzazione della quantità di carburante disponibile nei serbatoi del drone.**

CmdDisplay	**Elemento del sistema che consente alla centrale operativa di inviare comandi al drone attraverso una serie di pulsanti. I comandi consentono di:** **– Iniziare la missione** **– Terminare la missione** **– Impostare la velocità** **– Aumentare la velocità** **– Diminuire la velocità**
Display	**Elemento del sistema in grado di visualizzare i dati provenienti dai sensori**

5.3 Scenari

5.3.1 Inizio missione

Campo	Descrizione
ID(Nome)	**Inizio missione**
Descrizione	**Avvio della missione del drone**
Attori	**Operatore**
Precondizioni	**Il drone è fermo e la velocità iniziale è impostata**
Scenario Principale	**L'operatode, dopo aver impostato la velocità di partenza, avvia la missione e il drone decolla**
Scenari alternativi	–
Postcondizioni	**Il drone è in volo alla velocità impostata**

5.3.2 Fine missione

Campo	Descrizione
ID(Nome)	**Fine missione**
Descrizione	**La missione termina**
Attori	**Operatore**
Precondizioni	**La missione è iniziata**
Scenario Principale	**L'operatore fa terminare la missione e il drone atterra**
Scenari alternativi	**–**
Postcondizioni	**Il drone è atterrato e la missione è terminata**

5.3.3 Controllo velocità

Campo	Descrizione
ID(Nome)	**Controllo velocità**
Descrizione	**Regolare la velocità del Drone**
Attori	**Operatore**
Precondizioni	**La missione è iniziata**
Scenario Principale	**L'operatore aumenta o diminuisce la velocità del Drone a scelta. L'operatore attende dal Drone un messagio che signali il sucesso o il fallimento della modifcica. Nel caso di una velocità maggiore di maxSpeed o minore di minSpeed (fallimento) l'operatore imposta un'altra velocità**
Scenari alternativi	**Se l'operatore non riceve alcun messagio di risposta entro un dato intervallo di tempo ripete l'operazione di Controllo velocità**
Postcondizioni	**Il drone ha la velocità impostata dall'operatore**

5.3.4 Visualizzazione dati

Campo	Descrizione
ID(Nome)	**Visualizza Dati**
Descrizione	**Visualizzazione delle informazioni dei sensori del drone**
Attori	–
Precondizioni	**La missione è iniziata**
Scenario Principale	**L'operatore o il responsabile visualizzanno le informazioni dei sensori del drone. Nel caso in cui i dati non vengano ricevuti si riptete l'operazione**
Scenari alternativi	–
Postcondizioni	**L'operatore o il responsabile hanno sul loro dispositivo le informazioni dei sensori**

5.3.5 Memorizzazione dati

Campo	Descrizione
ID(Nome)	**Memorizza Dati**
Descrizione	**Le fotografie scattate dal drone sono inviate ad un server che provvede a memorizzarle insieme alle coordinate geografiche**
Attori	–
Precondizioni	**La missione è iniziata, il drone ha scattato la foto e l'ha inviata al server**
Scenario Principale	**Le immagini che il drone ha fato sono inviati a un server,dove sono memorizzati**
Scenari alternativi	**Non ci sono nuove immagini per inviare al server**
Postcondizioni	**Il server ha le immagini che il drone ha fato**

5.3.6 Scatta fotografia

Campo	Descrizione
ID(Nome)	**Scatta Foto**
Descrizione	**Il drone scatta una foto del territorio dove ad intervalli di tempo regolari**
Attori	**Tempo**
Precondizioni	**La missione è iniziata, DFT è maggiore di 0**
Scenario Principale	**Il drone è nella latitudine X e longitudine Y e scatta una foto del territorio**
Scenari alternativi	–
Postcondizioni	**La foto viene inviata al server per la memorizzazione**

5.3.7 Aggiornamento dati

Campo	Descrizione
ID(Nome)	**Aggiorna Dati Sensori**
Descrizione	**I diversi sensori del drone aggiornano i loro dati**
Attori	**Tempo**
Precondizioni	**La missione è iniziata**
Scenario Principale	**I sensori attualizzano i loro dati**
Scenari alternativi	–
Postcondizioni	**I sensori sono aggiornati**

5.4 (Domain) Model

Analizzando le specifiche richieste seguendo un approccio top-down sono stati individuati nel sistema DroneMission tre soggetti, tra loro interagenti: Drone, HeadQuarter e Smartphone.

Per modellare tali soggetti, che in seguito verranno analizzati in maniera più approfondita e trattati come sottosistemi in quanto composti anch'essi da soggetti interagenti, si è deciso di utilizzare Contact, un metalinguaggio alla pari di UML (entrambi sono espressi in termini di MOF) in possesso di maggiore

potere espressivo. Attraverso questo metalinguaggio è possibile non solo definire il comportamento del sistema, ma anche, grazie a un motore Prolog, generare il codice (per ora Java) di un prototipo perfettamente funzionante. In questo modo è già possibile testare il progetto totale, avendo solo completato l'analisi.

L'importanza di questo strumento è estremamente evidente: mentre viene effettuata l'analisi del problema viene automaticamente creata un'implementazione già completamente funzionante.

La differenza tra un approccio di risoluzione Extreme Programming e uno attraverso Contact risulta quindi palese: mentre nel primo caso ci si concentra totalmente e direttamente nella scrittura di codice e nel miglioramento dello stesso con conseguenti problematiche dovute a una scarsa analisi e frettolosa ricerca di un prototipo funzionante, nel secondo caso lo sforzo e il tempo impiegati in fasi di analisi (che risulta molto più accurata e precisa) vengono ripagati con un abbattimento del tempo impiegato nella programmazione e anche con una forte e stabile struttura d'analisi dedotta univocamente e semplicemente dai requisiti.

Attraverso l'utilizzo del metalinguaggio Contact viene fortemente mantenuta una completa tracciabilità di tutte le entità: possiamo infatti ritrovare all'interno del codice gli stessi componenti con i metodi definiti all'interno della specifica del metalinguaggio. Contact è un metalinguaggio improntato all'interazione, infatti si può constatare che non si tratta più con POJO, ma con veri e propri Subject, cioè agenti attivi e quindi attori veri e propri del sistema, che comunicano tra loro in modi differenti.

Infine, con l'utilizzo di questo metalinguaggio viene il systemdesigner, che ha il compito di implementare tutti i requisiti richiesti dall'applicationdesigner, come per esempio una nuova tipologia di comunicazione, avrà un lavoro molto semplificato e, soprattutto, riusabile: una volta completata la nuova feature, questa sarà presente in tutti i succesivi utilizzi del metalinguaggio.

5.4.1 DroneMissionSystem

Come detto il DroneMissionSystem sarà composto da tre sub systems eterogenei e distribuiti, per questo verranno introdotti tre contesti, uno per il drone, uno per la centrale operativa ed il terzo per lo smartdevice:

```
ContactSystem DroneMissionSystem -awt spaceUpdater [host="localhost" port=4010];

Context subSystemDrone;
Context subSystemHeadQuarter;
Context subSystemSmartdevice;

Subject smartphone context subSystemSmartdevice -w;
Subject drone context subSystemDrone -w;
Subject headQuarter context subSystemHeadQuarter -w;
```

Dopo aver definito quali saranno i sottosistemi che entreranno in gioco si procederà a definire i messaggi che questi si scambieranno e successivamente le loro comunicazioni di alto livello.

I messaggi saranno di diverso tipo a seconda della loro funzionalità: i comandi inviati dalla centrale di controllo saranno visti come Request/Response (una volta inviati, il mittente si metterà in attesa di un risposta da parte del destinatario, con l'esito dell'operazione svolta: "COMPLETED" o "ERROR") poiché ci si aspetta che all'invio di ogni comando il drone metta in atto la richiesta e risponda con l'esito dell'operazione, le notifiche ed i dati dei sensori inviati dal drone saranno invece dei Signal (verranno inseriti in uno shared-space e potranno essere prelevati dagli altri soggetti in gioco, senza dover specificare chi), mentre per quanto riguarda le foto, queste saranno inviate sotto forma di Dispatch alla centrale di controllo (senza attendere l'unico sotto sistema in grado di riceverla) in quanto, una volta ricevuta ci si aspetta che quest'ultima la memorizzi insieme ai dati dell'istante in cui è stata inviata.

```
//      Highlevel communications
//Drone sends photo
sendDataPhoto: drone forward photo to headQuarter;
//Headquarter receives photo
receiveDataPhoto: headQuarter serve photo support=TCP [host="localhost" port=4060];

//Drone sends data of sensors
sendDataSensors: drone emit dataSensor;
//Headquarter and Smartphone receive data of sensors
headquarterReceiveDataSensors: headQuarter sense dataSensor;
smartphoneReceiveDataSensors: smartphone sense dataSensor;

//Drone notifies start/end mission
sendnotify: drone emit notify;
//Smartphone receives notifications about mission
smartphoneReceiveNotify: smartphone sense notify;

//HeadQuarter sends command
sendCommand: headQuarter ask command to drone;
//Drone receives command
receiveCommand: drone accept command support=TCP [host="localhost" port=4050];
```

A questo punto è possibile specificare il comportamento di ogni singolo soggetto. Per quanto riguarda il drone, questo, una volta inizializzato, transiterà nello stato ready, in cui attenderà la ricezione del comando setspeed dalla centrale di controllo per transitare nello stato startMission, in cui avvierà la missione e comunicherà agli Smartphone l'avvenuto decollo prima di spostarsi nel nuovo stato onMission. In questo stato il velivolo invierà i dati dei sensori e, ogni DS secondi, un pacchetto con la foto e i dati attualizzati; potrebbe inoltre ricevere dei comandi dalla centrale di controllo, quali setspeed o stop. Alla ricezione di tali messaggi lo stato diventerà commandHandler, in cui, prima si analizza il contenuto del messaggio, poi, a seconda del comando ricevuto il drone transiterà in uno dei due possibli stati: setspeed o endMission. Nel primo si provvederà ad aggiornare la velocità di crociera, mentre nel secondo si notificherà agli smarphone il termine della missione e si provvederà a dar atterrare il drone.

```
BehaviorOf drone
{
```

```
var String msgCommand = ""
var String cmdReply = ""
var boolean droneCheck
var String sensorsDatas
var String dataPhoto
action void startMission()
action boolean isMissionEnding()
action void endMission()
// commands
action String handleCommand(String cmd)
action boolean isCommandStart(String cmd)
action String getFailReplyToCommand()
action String getOkReplyToCommand()
// notify
action String getNotifyStart()
action String getNotifyEnd()
// sensors
action String getDataFromSensors()
action String getDataPhoto()
state st_Drone_init initial

    println ("----- Drone Initialized -----")
    goToState st_Drone_ready
endstate
state st_Drone_ready

    println ("----- Waiting setSpeed -----")
    doInOut receiveCommand()
    set msgCommand = code.curInputMsgContent
    set droneCheck = exec isCommandStart(msgCommand)
    if{ droneCheck == true } { goToState st_Drone_startMission }
    replyToRequest command(call getFailReplyToCommand());
    println ("ERROR: expected 'start' command. Received: " + msgCommand)
endstate
state st_Drone_startMission

    replyToRequest command(call getOkReplyToCommand());
    exec startMission()
    doOut sendnotify(call getNotifyStart())
    goToState st_Drone_onMission
endstate
state st_Drone_onMission
    // send data sensors
    set sensorsDatas = exec getDataFromSensors()
```

```
            doOut sendsensorsDatas(sensorsDatas)
            // send photos
            set dataPhoto = exec getDataPhoto()
            doOut sendDataPhoto(dataPhoto)
            set droneCheck = exec isMissionEnding()
            if { droneCheck == true } { goToState st_Drone_endMission }
            // received a command
            onMessage? command goToState st_Drone_commandHandler
        endstate
        state st_Drone_commandHandler
            doInOut receiveCommand()
            set msgCommand = code.curInputMsgContent
            set cmdReply = exec handleCommand(msgCommand)
            replyToRequest command(cmdReply);
            // get back on mission
            goToState st_Drone_onMission
        endstate
        state st_Drone_endMission
            // send last data sensors
            set sensorsDatas = exec getDataFromSensors()
            doOut sendsensorsDatas(sensorsDatas)
            exec endMission()
            doOut sendnotify(exec getNotifyEnd())
            transitToEnd
        endstate
    }
```

Per quanto concerne invece HeadQuarter, anch'esso dopo l'inizializzazione transiterà nello stato ready, in cui non farà altro che inviare al drone il comando setspeed con la velocità di crociera ed attendere l'arrivo di una risposta prima di spostarsi nello stato onMission. Qui si controllerà se bisogna inviare qualche comando al drone, in caso di risposta affermativa verrà eseguita l'operazione. Anche qui, se il messaggio inviato è stop, vi sarà un transito verso lo stato endMission, altrimenti si rimarrà nello stato analizzando la presenza di eventuali messaggi da parte del drone: se nello shared-space sono presenti i dati dei sensori si transiterà nello stato receivedSensorsData che, dopo aver acquisito il messaggio (lasciandolo a disposizione di altri dispositivi), provvederà all'aggiornamento dei dati e tornerà su onMission; se invece verrà ricevuto un messaggio di tipo photo si andrà nello stato receivedPhoto che provvederà, prima di tornare in onMission, a recuperare il messaggio e memorizzarlo.

```
    BehaviorOf headQuarter
    {
        var String command
```

```
var String sensorsDatasReceived
var String photoReceived
var String commandAnswer
var boolean tmpCheck
var boolean missionEnd
action String getCommandStart()
action String getCommandToSend()
action boolean replyIsOk(String reply)
action void updateDashboard(String sensorsDatasReceived)
action void storeSensorsData(String sensorsDatasReceived)
action void storePhotoData(String photoReceived)
action void showPicturePackage(String photoReceived)
action boolean missionIsGoingToEnd()
action void shutdown()
state st_HeadQuarter_init initial

    println ("----- HeadQuarter Initialized -----")
    goToState st_HeadQuarter_ready
endstate
state st_HeadQuarter_ready

    println ("----- Ready to send command -----")
    doOutIn sendCommand(exec getCommandStart())
    acquireAnswerFor command
    set commandAnswer = code.curReplyContent
    println("DRONE REPLY: " + commandAnswer)
    set tmpCheck = exec replyIsOk(commandAnswer)
    if { tmpCheck == true } { goToState st_HeadQuarter_onMission }
endstate
state st_HeadQuarter_onMission

    set missionEnd = exec missionIsGoingToEnd()
    if { missionEnd == true } {

       goToState st_HeadQuarter_endMission
    }
    // get sensors data
    onMessage? sensorsData goToState st_HeadQuarter_receivedSensorsData
    // get photos
    onMessage? photo goToState st_HeadQuarter_receivedPhoto
    // check if command is clicked in Dashboard and send it
    set command = exec getCommandToSend()
    doOutIn sendCommand(command)
    acquireAnswerFor command
```

```
            set commandAnswer = code.curReplyContent
            set tmpCheck = exec replyIsOk(commandAnswer)
            if { tmpCheck == false } { println("DRONE CMD FAILED: " + commandAnswer) }
        endstate
        state st_HeadQuarter_receivedSensorsData
            // get sensors data
            doPerceive headquarterReceivesensorsDatas()
            set sensorsDatasReceived = code.curInputMsgContent
            // update Dashboard
            exec updateDashboard(sensorsDatasReceived)
            exec storeSensorsData(sensorsDatasReceived)
            goToState st_HeadQuarter_onMission
        endstate
        state st_HeadQuarter_receivedPhoto
            // get photo data
            doIn receiveDataPhoto()
            set photoReceived = code.curInputMsgContent
            // store info
            call storePhotoData(photoReceived)
            call showPicturePackage(photoReceived)
            goToState st_HeadQuarter_onMission
        endstate
        state st_HeadQuarter_endMission
            call shutdown()
            transitToEnd
        endstate

    }
```

Lo Smartphone, invece, uscirà dallo stato di init dopo aver ricevuto una notifica e andrà in missionStart che notificherà all'utente l'avvio della missione e transiterà in waitingForData. In questo stato si attenderanno messaggi provenienti dal drone contenenti o i dati dei sensori, o una nuova notifica, questa volta di fine missione. Alla ricezione dei dati relativi ai sensori lo smartphone transiterà in receivedData, dove provvederà al recupero e alla visualizzazione dei valori forniti dal drone, mentre nel caso in cui dovesse ricevere la notifica transiterà in notifyHandler che provvederà a far terminare la sessione nel caso in cui questa sia uno stop.

```
    BehaviorOf smartdevice
    {
        var String notifyContent
        var String dataDroneReceived
        var boolean tmpNotify
        action void notifyUserMissionStarted()
```

```
        action void updateGauges(String data)
        action boolean isNotifyStart(String notify)
        action void missionFinished()
        state st_Smartdevice_init initial

            onMessage notify transitTo st_Smartdevice_missionStart
        endstate
        state st_Smartdevice_missionStart
            call notifyUserMissionStarted()
            goToState st_Smartdevice_waitingForData
        endstate
        state st_Smartdevice_waitingForData

            onMessage? sensorsData goToState st_Smartdevice_receivedData
            onMessage? notify goToState st_Smartdevice_endMission
        endstate
        state st_Smartdevice_receivedData
            // get data from drone
            doPerceive smartdeviceReceivesensorsDatas()
            set dataDroneReceived = code.curInputMsgContent
            call updateGauges(dataDroneReceived)
            goToState st_Smartdevice_waitingForData
        endstate
        state st_Smartdevice_notifyHandler
            doPerceive smartdeviceReceiveNotify()
            set notifyContent = code.curInputMsgContent
            set tmpNotify = exec isNotifyStart(notifyContent)
            if { tmpNotify == true } { goToState st_Smartdevice_missionStart }
            if { tmpNotify == false } { goToState st_Smartdevice_endMission }
            goToState st_Smartdevice_waitingForData
        endstate
        state st_Smartdevice_endMission
            call missionFinished()
            transitToEnd
        endstate

    }
```

5.5 Test plan

È possibile sfruttare il codice autogenerato da contact per avere un primo test preliminare. È sufficiente definire i metodi e le classi mancanti per dare al sistema un minimo di logica. Verrà utilizzata anche una classe statica Messages

che dichiara i tipo di messaggi scambiati fra le varie entità del sistema. Tale classe verrà poi ampliata successivamente in un package a parte per descrivere in maniera completa ed esaustiva la struttura dei dati.

I Gauge sono invece già forniti nel pacchetto it.unibo.droneMission.gauge, ampliamente analizzati e sviluppati già in altri progetti (leggi lezione).

```
public class Messages {

    public static String NOTIFY_START = "start";
    public static String NOTIFY_END = "end";
    public static String REPLY_OK = "ok";
    public static String REPLY_NO = "no";
    public static String COMMAND_SETSPEED = "setspeed";
    public static String COMMAND_START = "start";
    public static String COMMAND_STOP = "stop";
    public static String SENSORS_LAST = "0";

}
```

Mentre ecco un primo prototipo del simulatore del drone Drone:

```
public class Drone extends DroneSupport {

    private int num_sensors_sent;
    private int MAX_SENSORS_SENT = 20;
    public Drone(String s) throws Exception{
        super(s);
        num_sensors_sent = 0;
    }
    // contact clean message
    private static String cleanMessage(String msgString) {

        return msgString.replace("\"","").replace("'","");
    }
    @Override

        protected void startMission() throws Exception {
        env.println("START MISSION");
    }
    @Override
    protected void endMission() throws Exception {
        env.println("STOP MISSION");
    }
    @Override
    protected String getDataFromSensors() throws Exception {
        String result = "";
        if (num_sensors_sent < MAX_SENSORS_SENT) {
```

```
            int odomenter = (int)(Math.random() * 100);
            int speedometer = (int)(Math.random() * 100);
            int fuel = (int)(Math.random() * 100);
            result = String.format("odomoter:%s;speedometer:%s;fuel:%s", odomenter, spee
        }
        else
            result = Messages.SENSORS_LAST;
        num_sensors_sent++;
        env.println("Sending sensor: " + result);
        return result;
    }
    @Override
    protected String getDataPhoto() throws Exception {
        String photo = "photoX;dataY;timeZ";
        env.println("Sending photo: " + photo);
        return photo;
    }
    @Override
    protected String handleCommand(String cmd) throws Exception {
        cmd = cleanCommand(cmd);
        if(cmd.startsWith(Messages.COMMAND_SETSPEED) || cmd.startsWith(Messages.COMMAND

            env.println("COMMAND handler OK: " + cmd);
            return Messages.REPLY_OK;
        }
        else {

            env.println("COMMAND handler FAIL: " + cmd);
            return Messages.REPLY_NO;
        }
    }
    @Override
    protected boolean isCommandStart(String cmd) throws Exception {

        env.println(cmd + " " + Messages.COMMAND_START);
        return cleanMessage(cmd).equalsIgnoreCase(Messages.COMMAND_START);
    }
    @Override
    protected String getFailReplyToCommand() throws Exception {
        return Messages.REPLY_NO;
    }
    @Override
    protected String getOkReplyToCommand() throws Exception {
```

```
            return Messages.REPLY_OK;
        }
        @Override
        protected String getNotifyStart() throws Exception {
            return Messages.NOTIFY_START;
        }
        @Override
        protected String getNotifyEnd() throws Exception {
            return Messages.NOTIFY_END;
        }
        @Override
        protected boolean isMissionEnding() throws Exception {

            return num_sensors_sent >= MAX_SENSORS_SENT;
        }
    }
```

Ecco il prototipo dell'HeadQuarter, la centrale operativa:

```
    public class HeadQuarter extends HeadQuarterSupport {
        // Just for testing purpose
        private int commandCounter;
        private String sensors_received;
        private int MAX_CMD = 3;
        public HeadQuarter(String s) throws Exception{
            super(s);
            commandCounter = 0;
            sensors_received = "";
        }
        @Override
        protected String getCommandToSend() throws Exception {
            String cmd;
            if (commandCounter >= MAX_CMD) {
                cmd = Messages.COMMAND_STOP;
            }
            else {
                commandCounter += 1;
                cmd = Messages.COMMAND_SETSPEED + " " + ((int) (Math.round(Math.random() * 1
            }
            return cmd;
        }
        @Override
        protected void updateDashboard(String dataSensorsReceived)
        throws Exception {
```

```
            env.println("DATA RECEIVED: " +dataSensorsReceived);
        }
        @Override
        protected void storePhotoData(String photoReceived) throws Exception {

            env.println("PHOTO RECEIVED: " +photoReceived);
        }
        @Override
        protected void shutdown() throws Exception {
            env.println("MISSION END - SHUTDOWN.");
        }
        @Override
        protected String getCommandStart() throws Exception {
            return Messages.COMMAND_START;
        }
        @Override
        protected boolean replyIsOk(String reply) throws Exception {

            return reply.equalsIgnoreCase(Messages.REPLY_OK);
        }
        @Override
        protected void storeSensorsData(String sensorsDatasReceived)
        throws Exception {
            sensors_received = sensorsDatasReceived;
        }
        @Override
        protected void showPicturePackage(String photoReceived)
        throws Exception {

            env.println("PHOTO RECEIVED: " +photoReceived);
        }
        @Override
        protected boolean missionIsGoingToEnd() throws Exception {

            return sensors_received.equalsIgnoreCase(Messages.SENSORS_LAST);
        }

    }
```

Ecco invece il prototipo de lo Smartdevice:

```
    public class Smartdevice extends SmartdeviceSupport {
```

```
        public Smartdevice(String s) throws Exception{
            super(s);
        }
        @Override
        protected void notifyUserMissionStarted() throws Exception {
            env.println("MISSION_STARTED");
        }
        @Override
        protected void missionFinished() throws Exception {
            env.println("MISSION END");
        }
        @Override
        protected void updateGauges(String data) throws Exception {
            env.println("DATA DRONE: " + data);
        }
        @Override
        protected boolean isNotifyStart(String notify) throws Exception {
            return notify == Messages.NOTIFY_START;
        }

    }
```

Capitolo 6

Analisi del problema

6.1 Logic architecture

Il sistema tutto viene suddiviso in tre sottisistemi Drone - HeadQuarter - SmartDevice che verranno poi analizzati uno ad uno:

```
Context subSystemDrone;
Context subSystemHeadQuarter;
Context subSystemSmartDevice;
```

Viene subito naturale definire anche una univoca e comune rappresentazione dei dati che le varie entità (o Subject) dei sottosistemi andranno a scambiare.

6.1.1 Rappresentazione dei dati

È necessario definire un modello unico di rappresentazione dei dati affinché le varie unità riescano a interpretare correttamente le informazioni scambiate.

Constatato che lo scambio di messaggi in Contact avviene tramite l'utilizzo di stringhe, si è scelto di rappresentare i dati attraverso il formato JSON. Ogni messaggio (IMessage) sarà poi trasformato in classe (o instanza) attraverso l'utilizzo di Factory.

```
package it.unibo.droneMission.interfaces.messages;
public interface IMessage {

    public String toJSON();

}
```

Analizzando i requisiti, emergono diversi tipi di dati che le entità dei sottosistemi si scambiano:

```
ICommand         -- comandi inviati al drone
IReply           -- le risposte dei comandi che il drone manda indietro
```

```
INotify          -- le notifiche inviate dal drone
ISensorsData     -- i dati dei sensori del drone
IPicturePackage  -- le foto con allegato i dati dei sensori
```

È utile sottolineare che, per motivi di overhead nella comunicazione, si è scelto di utlizzare un unico "pacchetto" di informazione (ISensorsData) che collezioni tutti i dati dei sensori del drone e li invii alla centrale di controllo. Un altro approccio sarebbe stato quello di inviare singolarmente lo stato interno di ogni sensore, ma questo avrebbe aumentato enormemente il numero di messaggi scambiati (e di costo, in termini di comunicazione) senza alcun vantaggio apparente.

È necessario anche far notare che IPicturePackage non è solo la rappresentazione della fotografia che il drone ha scattato, ma contiene anche i dati dei sensori al momento dello scatto, così come richiesto dal richiedente.

ICommand, IReply e INotify sono invece dei IMessage con tipo (in questo singolo caso, solo intero IMessageTypeAsInt) e un valore, un intero per IComand (IMessageTypeAsInt) e una stringa per IReply e INotify (IMessageValueAsString).

In aggiunta, si è scelto di aggiungere un timestamp ai vari messaggi in modo da poter ricostruire in qualisiasi momento la cronologia dell'interazione:

```
package it.unibo.droneMission.interfaces.messages;
public interface IMessageWithTime extends IMessage {

    public long getTime();
    public void setTime(long time);

}
```

Ed ecco quindi il modello UML dei messaggi:
#FIX_ME - qui UML interfaces.messages

6.1.2 Sottosistema Drone

#FIX_ME: QUI CI ANDREBBE LA DESCRIZIONE DEL DRONE E DEI GAUGES

6.1.3 Sottosistema HeadQuarter

La centrale operativa avrà dunque il compito di controllare il drone e di registrare (mostrandole agli operatori) le informazioni che esso invia.

6.1.3.1 Struttura

Analizzando la centrale operativa, si decide di suddividere il Subject headquarter in quattro nuove entità:

```
// Utente che visualizza informazioni
Subject uiLog context subSystemHeadQuarter;
```

```
// Utente che invia comandi al drone
Subject uiCommand context subSystemHeadQuarter;
// Server che interagisce con utente
Subject server context subSystemHeadQuarter;
// Unità che interagisce con drone
Subject controlUnit context subSystemHeadQuarter;
```

6.1.3.2 Interazione

Segue la parte della comunicazione fra i vari Subject.

L'utente Log che vuole visualizzare i dati relativi ad una missione o le immagini o i dati dei sensori (passati o correnti) invia una Request (resta in attesa quindi di una risposta) al Server

```
// User log messages
Request showMeMission;
Request showMeSensorsData;
Request showMePicturePackage;
Request showMeNotifies;
// show me mission
sendShowMeMission: uiLog demand showMeMission to server;
receiveShowMeMission: server grant showMeMission;
// show me sensors data
sendShowMeSensorsData: uiLog demand showMeSensorsData to server;
receiveShowMeSensorsData: server grant showMeSensorsData;
// show me picture package
sendShowMePackagepicturePackage: uiLog demand showMePicturePackage to server; receiveSh
// show me notifies
sendShowMeNotifies: uiLog demand showMeNotifies to server;
receiveShowMeNotifies: server grant showMeNotifies;
```

Per quanto riguarda invece l'invio dei comandi, l'utente Command invia al Server una Request newCommand che verrà inoltrata poi alla unitControl attraverso una nuova Request forwardCommand. Questa scelta è stata effettuata al fine di dividere le responsabilità dei vari Subject, in particolare solo alla controlUnit viene affidato il compito di interagire con il drone

```
// User command
Request newCommand;
// Server forward command to drone through controlUnit
Request forwardCommand;
// UI send command
sendUINewCommand: uiCommand demand newCommand to server;
receiveUINewCommand: server grant newCommand;
// server sends command to drone handler
sendForwardCommand: server demand forwardCommand to controlUnit;
receiveForwardCommand: controlUnit grant forwardCommand;
```

La controlUnit interagisce con il drone attraverso le scelte prese in fase di analisi dei requisiti:

```
// messages between drone and controlUnit
Dispatch picturePackage;
Request command;
Signal sensorsData;
// drone sends sensors data
sendSensorsDatas: drone emit sensorsData;
controlUnitReceiveSensorsDatas: controlUnit sense sensorsData;
// controlUnit sends command
sendCommand: controlUnit demand command to drone;
receiveCommand: drone grant command;
// drone sends picturePackage
sendDatapicturePackage: drone forward picturePackage to controlUnit;
receiveDatapicturePackage: controlUnit serve picturePackage;
```

È utile sottolineare che nel codice riportarto è sottointesa l'idea che i Subject drone e controlUnit possono essere eseguiti su JVM diverse. A tal fine è sufficiente specificare il relativo al supporto. In fase di progettazione verrà utilizzato TCP, ad esempio support=TCP [host="localhost" port=4050].

6.1.3.3 Comportamento

Seque la descrizione del comportamento della controlUnit, l'entità responsabile dell'interazione con il drone.

```
BehaviorOf controlUnit {

    var String cmd
    var String rpl
    var String sensorsDatasReceived
    var String picturePackageReceived
    var boolean tmpCheck
    // init
    action void init()
    // store mission starts
    action void storeMissionStarted()
    // store info from drone
    action void storeDataSensors(String sensorsDatasReceived)
    action void storePicturePackage(String picturePackageReceived)
    // check commands to send
    action boolean checkCommandStart(String command)
    action boolean checkReplyCommandStart(String reply)
    action String getWrongStartCommandReply()
    action void storeCommandAndReply(String c, String r)
    // shutdown
```

```
action boolean checkEndMission()
action void shutdown()
state st_controlUnit_init initial
    showMsg("Control Unit - initial")
    exec init()
    goToState st_controlUnit_ready
endstate
state st_controlUnit_ready
    doInOut receiveForwardCommand()
    set tmpCheck = exec checkCommandStart(code.curInputMsgContent)
    if { tmpCheck == true } {

       goToState st_controlUnit_startMission
    }
    if { tmpCheck == false } {

       goToState st_controlUnit_wrongStartCommand
    }
endstate
state st_controlUnit_startMission
    set cmd = code.curInputMsgContent
    doOutIn sendCommand(cmd)
    acquireAnswerFor command
    set rpl = code.curReplyContent
    replyToRequest forwardCommand(rpl)
    set tmpCheck = exec checkReplyCommandStart(rpl)
    if { tmpCheck == false } {
       goToState st_controlUnit_ready
    }
    call storeMissionStarted()
    call storeCommandAndReply(cmd, rpl)
    goToState st_controlUnit_onMission
endstate
// in case of wrong start command received
state st_controlUnit_wrongStartCommand
    set cmd = code.curInputMsgContent
    set rpl = exec getWrongStartCommandReply()
    replyToRequest forwardCommand(rpl)
    goToState st_controlUnit_ready
endstate
state st_controlUnit_sendCommand
    doInOut receiveForwardCommand()
```

```
            set cmd = code.curInputMsgContent
            doOutIn sendCommand(cmd)
            acquireAnswerFor command
            set rpl = code.curReplyContent
            call storeCommandAndReply(cmd, rpl)
            replyToRequest forwardCommand(rpl)
            goToState st_controlUnit_onMission
        endstate
        state st_controlUnit_onMission
            // check if there are commands to send
            onMessage? forwardCommand goToState st_controlUnit_sendCommand
            // get sensors data
            onMessage? sensorsData goToState st_controlUnit_receivedSensorsData
            // get picturePackages
            onMessage? picturePackage goToState st_controlUnit_receivedpicturePackage
            // check if end mission
            set tmpCheck = exec checkEndMission()
            if { tmpCheck == true } {

               goToState st_controlUnit_endMission
            }
        endstate
        state st_controlUnit_receivedSensorsData

            doPerceive controlUnitReceiveSensorsDatas()
            set sensorsDatasReceived = code.curInputMsgContent
            call storeDataSensors(sensorsDatasReceived)
            goToState st_controlUnit_onMission
        endstate
        state st_controlUnit_receivedpicturePackage
            doIn receiveDatapicturePackage()
            set picturePackageReceived = code.curInputMsgContent
            call storePicturePackage(picturePackageReceived)
            goToState st_controlUnit_onMission
        endstate
        state st_controlUnit_endMission
            call shutdown()
            //goToState st_controlUnit_init // just for test
            transitToEnd
        endstate
    }
```

Ecco invece l'analisi completa del comportamento del server

```
BehaviorOf server {

    var String command
    var String reply
    var String sensorsData
    var String picturePackage
    var String mission
    action String showReplyToCommand(String reply)
    action String getSensorsData(String mission_id)
    action String getPicturePackage(String  mission_id)
    action String getNotifies(String mission_id)
    action String getMission(String mission_id)
    state st_Server_init initial
    goToState st_Server_Handler
    endstate
    state st_Server_Handler

        onMessage? newCommand goToState st_Server_forwardCommand
        onMessage? showMeMission goToState st_Server_showMission
        onMessage? showMePicturePackage goToState st_Server_showPicturePackage
        onMessage? showMeSensorsData goToState st_Server_showSensorsData
    endstate
    state st_Server_showMission
        doInOut receiveShowMeMission()
        set mission = exec getMission(code.curInputMsgContent)
        replyToRequest showMeMission(mission)
        goToState st_Server_Handler
    endstate
    state st_Server_showPicturePackage

        doInOut receiveShowMePackagepicturePackage()
        set picturePackage = exec getPicturePackage(code.curInputMsgContent)
        replyToRequest showMePicturePackage(picturePackage)
        goToState st_Server_Handler
    endstate
    state st_Server_showSensorsData
        doInOut receiveShowMeSensorsData()
        set sensorsData = exec getSensorsData(code.curInputMsgContent)
        replyToRequest showMeMission(sensorsData)
        goToState st_Server_Handler
    endstate
    state st_Server_forwardCommand
        doInOut receiveUINewCommand()
        set command = code.curInputMsgContent
```

```
            doOutIn sendForwardCommand(command)
            acquireAnswerFor forwardCommand
            set reply = code.curInputMsgContent
            exec showReplyToCommand(reply)
            goToState st_Server_Handler
        endstate
    }
```

È utile ricordare che tutte le operzione di Get e Store saranno implementate in fase di progettazione attraverso una nuova entità detta Storage che sarà responsabile della persistenza dei dati.

L'entità Storage dovrà implementare la seguente interfaccia:

```
public interface IStorage {
    // init storage
    public void init();
    // mission
    public void startMission();
    public void endMission();
    public boolean isOnMission();
    public int getCurrentMissionID();
    public IMission getMission(int id);
    // commands
    public void storeCommandAndReply(ICommand command, IReply reply);
    public LinkedHashMap<ICommand, IReply> getLatestCommands(int n);
    public LinkedHashMap<ICommand, IReply> getCommandsByMission(int missionID);
    // notify
    public void storeNotify(INotify notify);
    public INotify getLatestNotify();
    public List<INotify> getLatestNotifies(int n);
    public List<INotify> getNotifiesByMission(int missionID);
    // sensors data
    public void storeSensorsData(ISensorsData data);
    public ISensorsData getLatestSensorsData();
    public List<ISensorsData> getLatestSensorsDatas(int n);
    public List<ISensorsData> getSensorsDatasByMission(int missionID);
    // picture package
    public void storePicturePackage(IPicturePackage pack);
    public IPicturePackage getLatestPicturePackage();
    public List<IPicturePackage> getLatestPicturePackages(int n);
    public List<IPicturePackage> getPicturePackagesByMission(int missionID);
    // general file
    public void storeFile(IFile file);
    public IFile getFile(String filename);
    public IFile getFile(long time);
```

```
        public List<IFile> getLatestFiles(int n);
        // for debugging purpose
        public void setDebug(int level);
        public void debug(String s, int level);

    }
```

Si sceglie di accorpare ICommand e IReply (strettamente connessi) attraverso un LinkedHashMap in modo da mantenere un'ordine preciso dei comandi inviati, in altre parole una lista ordinata cronologicamente.

L'interfaccia IMission è ora definita come segue e rappresenta una completa missione del drone con riferimento ai dati interni, alle fotografie scattate, alle notifiche ricevute e ai comandi inviati:

```
    public interface IMission {

        // mission ID
        public void setId(long id);
        public long getId();
        // start mission time
        public long getStartTime();
        public void setStartTime(long startTime);
        // end mission time
        public long getEndTime();
        public void setEndTime(long endTime);
        // Commands and replies
        public void setCommands(LinkedHashMap<ICommand, IReply> commands);
        public LinkedHashMap<ICommand, IReply> getCommands();
        // notifies
        public List<INotify> getNotifies();
        public void setNotifies(List<INotify> notifies);
        // picture packages
        public List<IPicturePackage> getPicturePackages();
        public void setPicturePackages(List<IPicturePackage> picturePackages);
        // sensors data
        List<ISensorsData> getSensorsDatas();
        void setSensorsDatas(List<ISensorsData> sensorsDatas);

    }
```

6.1.4 Sottosistema SmartDevice

Il sottosistema SmartDevice, invece, non necessita di particolari accorgimenti rispetto a quanto prodotto nell'analisi dei requisiti: la struttura rimane invariata (fatta eccezione per la definizione di external dei subject drone e headQuarter e la rimozione dell'implementazione grafica), quindi non sarà necessario modificare l'interazione (il subject Smartdevice riceve solo delle notify da parte del Drone). Mentre, per quanto riguarda il comportamento del sottosistema, si è

previsto che, una volta ricevuta dal drone la notifica della fine della missione il sottosistema tornerà nello stato di ready, mettendosi in attesa dell'inizio di una nuova missione:

```
BehaviorOf smartdevice {
var String notifyContent
var String dataDroneReceived
action void notifyUserMissionStarted()
action void updateGauges(String data)
action void missionFinished()
state st_Smartdevice_init initial
showMsg("waiting for Start Mission")
onMessage notify transitTo st_Smartdevice_missionStart
endstate
state st_Smartdevice_missionStart
exec notifyUserMissionStarted()
goToState st_Smartdevice_waitingForData
endstate
state st_Smartdevice_waitingForData
onMessage? sensorsData goToState st_Smartdevice_receivedData
onMessage? notify goToState st_Smartdevice_endMission
endstate
state st_Smartdevice_receivedData
// get data from drone
doPerceive smartdeviceReceivesensorsDatas()
set dataDroneReceived = code.curInputMsgContent
exec updateGauges(dataDroneReceived)
goToState st_Smartdevice_waitingForData
endstate
state st_Smartdevice_notifyHandler
doPerceive smartdeviceReceiveNotify()
set notifyContent = code.curInputMsgContent
if {notifyContent == "start"} { goToState st_Smartdevice_missionStart }
if {notifyContent == "end" } { goToState st_Smartdevice_endMission }
goToState st_Smartdevice_waitingForData
endstate
state st_Smartdevice_endMission
call missionFinished()
goToState st_Smartdevice_init
endstate
}
```

6.2 Abstraction gap

6.3 Risk analysis

Dopo aver analizzato i requisiti e aver definito quella che sarà l'architettura logica del sistema il system designer avrà la possibilità di scegliere quelle che secondo lui sono le soluzioni migliori ad alcuni problemi che il committente non ha specificato chiaramente nei requisiti. In questo contesto rientra tutto ciò che concerne le modalità di comunicazione:

- i comandi inviati dalla centrale operativa vengono gestiti come una request-response, operando una comunicazione diretta tra i due sotto sistemi in cui, dopo aver inviato un comando, il quartier generale, prima di eseguire altre operazioni, si metterà in attesa di una risposta da parte del drone, che potrà confermare la ricezione e l'esecuzione del comando, segnalare un errore, o comunicare che il comando non può essere eseguito;
- le notifiche di avvio e fine missione, inviate dal drone agli smart device, sono gestite come dei segnali (comunicazione uno a molti, in cui non si conoscono gli effettivi destinatari né il loro numero); le notifiche vengono inserite in uno shared space (condiviso da tutti i sottosistemi) da dove i destinatari che saranno sconosciuti al mittente le estrarranno;
- come le notifiche di avvio e fine missione, anche i dati dei sensori verranno inviati sotto forma di segnale per lo stesso motivo, inoltre, per ottimizzare il dispendio di risorse di comunicazione, tali dati verranno inviati in un unico messaggio sfruttando la codifica JSon (potrebbero essere previste ulteriori forme di codifica, quali ad esempio XML o Prolog);

Capitolo 7

Piano di lavoro

Dopo aver analizzato i requisiti e il problema, ed ottenuti dal system designer le specifiche, definite in modo generale e non ambiguo grazie all'utilizzo di Contact, dei tre sottosistemi è possibile suddividere il lavoro da assegnare ai team di application design, ognuno dei quali dovrà attenersi alle specifiche e sarà guidato nel suo lavoro di implementazione sia dalle classi generate dal motore Contact, sia dai Test Plan definiti nell'analisi dei requisiti: Contact definirà le modalità di interazione tra i sottosistemi, garantendo quindi che i messaggi inviati e ricevuti saranno sicuramente comprensibili a prescidere da come verrà implementata una entità rispetto all'altra, mentre i test plan garantiranno la correttezza del comportamente di ogni singola entità al fine di garantire la coerenza generale del sistema. In questo modo, quindi, si semplifica il lavoro dell'application designer che, una volta ricevute le specifiche Contact, non dovrà fare altro che implementare i metodi del sottosistema a lui assegnato senza preoccuparsi di come verranno implementati gli altri o di come il suo sottosistema di competenza dovrà relazionarsi con il sistema globale.

Nel caso in esame, a ciascuno dei tre team di application designer, verrà assegnato un sottosistema.

Capitolo 8

Progetto

8.1 Struttura

8.2 Interazione

8.3 Behavior

Capitolo 9

Implementazione

9.1 Head Quarter

Si decide di dividere l'implementazione del Context HeadQuarter in tre diversi progetti:

- systems.headquarter.storage
- systems.headquarter.server
- systems.headquarter.controlunit

9.1.1 Storage

Lo Storage è l'unità che si occuperà di registrare e di rendere disponibili su richiesta i dati delle missioni. Partendo dall'interfaccia IStorage si decide di estenderla in una nuova interfaccia IDataBase:

```
package it.unibo.droneMission.interfaces.headquarter;
import java.sql.ResultSet;
import java.util.Hashtable;
public interface IDataBase extends IStorage {

    public void connect();
    public void disconnect();
    public boolean isConnected();
    public void setUsername(String username);
    public String getUsername();
    public void setPassword(String password);
    public String getPassword();
    public void setHostname(String hostname);
    public String getHostname();
    public void setPort(int port);
    public int getPort();
```

```
    public void setDatabaseName(String dbname);
    public String getDatabaseName();
    // common db interaction
    public void select(String column);
    public void select(String[] columns);
    public void from(String table);
    public void from(String[] tables);
    public void where(String key, String value);
    public void where(Hashtable<String, String> set);
    public void orderBy(String column, String direction);
    public void limit(int n);
    public void offset(int n);
    public int update(Hashtable<String, String> set);
    public int insert(Hashtable<String, String> set);
    public ResultSet get();

}
```

L'interfaccia descrive le generiche operazioni di interazione con un database SQL. Nell'implementazione viene scelto MySQL, la gestione della connessione è affidata al connettore già sviluppato e disponibile nel pacchetto mysql-connector-java-5.1.16.jar.

L'implementazione dello Storage avviene tramite l'estensione di diverse classi:

1. public abstract class Storage implements IStorage { }
2. public abstract class DataBase extends Storage implements IDataBase { }
3. public class MySQL extends DataBase { } // Singleton

L'instanza finale di un oggetto MySQL verrà fornità attraverso una Factory che, in questo progetto, vede la seguente implementazione:

```
package it.unibo.droneMission.systems.headquarter.storage;
import it.unibo.droneMission.interfaces.headquarter.IStorage;
public class FactoryStorage {

    public static int MYSQL = 1;
    public static IStorage getInstance(int databaseType) throws Exception {
        if (databaseType == MYSQL)
            try {

              MySQL db = MySQL.getInstance();
              if (!db.isConnected()) {
```

```
                db.setDatabaseName("dronemission");
                db.setUsername("dronemission");
                db.setPassword("estate");
                db.setHostname("127.0.0.1");
                db.connect();
            }
            return db;
            } catch (Exception e) {
                e.printStackTrace();
            }
        else

            throw new Exception("Type storage: " + databaseType + " is not valid.");
        return null;
    }
}
```

9.1.2 Server

Il subject Server è l'unità che si occupa di interagire con l'utente, inoltrando i comandi alla ControlUnit e fornendo i log delle varie missioni (compreso i dati in tempo reale della missione corrente).

Si decide di implementare il Server sfruttando tecnologie web già esistenti. In questo modo i subject UILog e UICommand prenderano la forma di semplici browser web.

Ll server verrà quindi suddiviso in due sottoprogetti:

- systems.headquarter.server: codice autogenerato da contact (no Behaviour, solo comunicazione)
- system.headquarter.server.web: project DJango per definire il contenitore web.

9.1.2.1 system.headquarter.server

La prima parte del server vede l'implementazione della comunicazione con l'unità di controllo in modo da avere un canale di interazione già pronto. Viene eliminato il Behaviour dalla specifica contact e viene implementato il Server come segue:

```
package it.unibo.contact.headquarter_server;
public class Server extends ServerSupport {

    public Server(String name) throws Exception {
```

```
            super(name);
        }
        @Override
        public void doJob() throws Exception {
        }

    }
```

In pratica, una classe vuota. Il Container SubSystemHeadQuarterMain viene invece esteso in modo da avere un metodo publico capace di inoltrare i comandi attraverso il Subject Server all'unità di controllo:

```
    public class ServerStandAlone extends SubSystemHeadQuarterMain {

        public ServerStandAlone() {
            super();
            initProperty();
            init();
            configure();
        }
        public void doJob() {}
        public IReply forwardCommand(ICommand command) {

            String cmd = Utils.adaptJSONToContact(command.toJSON());
            try {

                server.curAcquireOneReply = server.hl_server_demand_forwardCommand_controlUn
                server.curReply = server.curAcquireOneReply.acquireReply();
            } catch (Exception e) {
                e.printStackTrace();
            }
            server.curReplyContent = server.curReply.msgContent();
            String reply = server.curReplyContent;
            IReply r = Factory.createReply(Utils.cleanJSONFromContact(reply));
            return r;
        }

    }
```

9.1.2.2 systems.headquarter.server.web

Per gestire l'interazione con l'utente si decide di sviluppare il Server come un contenitore web.

Dopo un'attenta e approfondita ricerca si decide di utilizzare tecnologie già esistenti. In particolare si sceglie di implementare il server.web in DJango (un

framework web scritto in Python) compatibile con Jython (un'implementazione del linguaggio python scritto in Java, compatibile quindi con l'altra parte del progetto Server).

Vengono definiti gli url come seguono:

```
urlpatterns = patterns('',

    (r'^libs/(?P<path>.*)$', 'django.views.static.serve', {'document_root': settings.ME
    # ajax
    (r'ajax/commands/send/type/(?P<type>\d+)/value/(?P<value>\d+)$', 'headquarter.views
    (r'^ajax/sensors/latest$', 'headquarter.views.latest_sensors'),
    (r'^ajax/pictures/latest$', 'headquarter.views.latest_picture'),
    # mission
    (r'^missions/(?P<id>\d+)/$', 'headquarter.views.get_mission'),
    (r'^missions/new$', 'headquarter.views.new_mission'),
    # pictures
    (r'^pictures/(?P<path>.*)$', 'django.views.static.serve',
    {'document_root': storage.FILE_PATH, 'show_indexes': True}),
    # index
    (r'^$', 'headquarter.views.index'),

)
```

Il Server e lo Storage vengono inizializzati nel file models (che in un classico progetto DJango si occuperebbe della descrizione del Database):

```
from it.unibo.droneMission.systems.headquarter.storage import FactoryStorage
from it.unibo.contact.headquarter_server import ServerStandAlone
server = ServerStandAlone()
storage = FactoryStorage.getInstance(FactoryStorage.MYSQL)
```

Mentre le view sono così definite:

```
from django.shortcuts import render_to_response
from headquarter.models import server, storage
import time
# time / 1000.0:
# java takes in account milliseconds, python uses
# float for them
def get_time(java_time):
    date = java_time / 1000.0
    return time.strftime("%d %b %Y %H:%M:%S", time.gmtime(date))

# here format fucntions
# ...
#
def latest_sensors(request):
```

```
        sensors = storage.getLatestSensorsData()
        f_s = {}
        if sensors is not None:
        f_s = format_sensors(sensors)
        return render_to_response('ajax/sensors_latest.html', f_s)

    def index(request):

        missions = storage.getPastMissions()
        info = {}
        info["missions"] = []
        for m in missions:
            formatted = {}
            formatted["id"] = m.getId()
            # .... format mission in formatted
            info["missions"].insert(0, formatted)
        return render_to_response('index.html', info)

    def get_mission(request, id):

        mission = storage.getMission(int(id))
        info = {}
        # .... format mission info info
        return render_to_response('mission.html',info)

    def new_mission(request):

        info = {}
        info["commands"] = {}
        info["commands"]["start"] = {}
        info["commands"]["start"]["type"] = TypesCommand.START_MISSION
        info["commands"]["start"]["value"] = 0
        info["commands"]["speed_set"] = {}
        info["commands"]["speed_set"]["type"] = TypesCommand.SPEED_SET
        info["commands"]["speed_set"]["value"] = 0
        info["commands"]["speed_increase"] = {}
        info["commands"]["speed_increase"]["type"] = TypesCommand.SPEED_INCREASE
        info["commands"]["speed_increase"]["value"] = 0
        info["commands"]["speed_decrease"] = {}
        info["commands"]["speed_decrease"]["type"] = TypesCommand.SPEED_DECREASE
        info["commands"]["speed_decrease"]["value"] = 0
        # reset internall mission ID
        storage.resetCurrentMissionID()
        return render_to_response('new-mission.html', info)

    def send_command(request, type, value):

        c = Command(int(type))
        c.setValue(int(value))
```

```
        reply = server.forwardCommand(c)
        info = format_reply(reply)
        return render_to_response('ajax/send-command.html', info)
```

In questo modo è possibile avere una pagina mission/new auto-aggiornabile attraverso l'uso di AJAX. A tal fine sono stati sviluppati alcuni metodi javascript in modo da offrire all'utente un interazione real-time con il drone.

Come da specifica, il commitente vuole visualizzare la posizione del drone direttamente sulla pagina web. Per far questo si è scelto di utilizzare, anche questa volta, tecnologie già ampliamente sviluppate. Nell'ambito web si decide di utilizzare Google Maps, che con semplici metodi javascript è in grado di gestire la geolocalizzazione di oggetto date le sue coordinate. Ecco un'implementazione base:

```
function initMap() {
    google.maps.visualRefresh = true;
    function initialize() {
        var mapOptions = {
        zoom: 8,
        center: new google.maps.LatLng(44.435505,10.976787),
        mapTypeId: google.maps.MapTypeId.ROADMAP,
        streetViewControl: false,
        draggable: false,
    };
    map = new google.maps.Map(document.getElementById('map'),
        mapOptions);
        var image = '/libs/img/drone-icon.png';
        var myLatLng = new google.maps.LatLng(0,0);
        droneIcon = new google.maps.Marker({
            position: myLatLng,
            map: map,
            icon: image
        });
    }
    google.maps.event.addDomListener(window, 'load', initialize);
}
```

L'aggiornamento automatico della mappa e della posizione del drone verrà assegnato al seguente metodo chiamato ciclicamente attraverso un set_timeout:

```
function updateMap(latitude, longitude) {


    myLatLng = new google.maps.LatLng(latitude,longitude);
    map.setCenter(myLatLng);
    droneIcon.setPosition(myLatLng);
}
```

9.1.3 Control Unit

L'unità di controllo, ovvero quell'entità responsabile dell'interazione con l'utente, viene implementata estendo il codice autogenerato da contact. Questa la classe che estente il supporto:

```
public class ControlUnit extends ControlUnitSupport {

    private IStorage storage;
    private double fuelLevel;
    public ControlUnit(String name) throws Exception {
        super(name);
        storage = FactoryStorage.getInstance(FactoryStorage.MYSQL);
    }
    protected void init() {
        fuelLevel = Fuelometer.MAX;
    }
    private void setFuelLevelFromGauges(ISensorsData s) {
        List<IGauge> gauges = s.getGauges();
        for (IGauge g : gauges) {

           if (g.getClass() == Fuelometer.class) {

             fuelLevel = g.getVal().valAsDouble();
             break;
           }
        }
    }
    @Override
    protected void storeDataSensors(String sensorsDatasReceived)
    throws Exception {

        sensorsDatasReceived = Utils.cleanJSONFromContact(sensorsDatasReceived);
        ISensorsData s = Factory.createSensorsData(sensorsDatasReceived);
        String val = "";
        for (IGauge g : s.getGauges())
        val += " " + g.getCurValRepDisplayed();
        env.println("Received Sensors:" + val);
        setFuelLevelFromGauges(s);
        storage.storeSensorsData(s);
    }
    @Override
    protected void storePicturePackage(String picturePackageReceived)
    throws Exception {
```

```
        picturePackageReceived = Utils.cleanJSONFromContact(picturePackageReceived);
        IPicturePackage p = Factory.createPicturePackage(picturePackageReceived);
        env.println("Received picture: " + p.getFile().getName());
        storage.storePicturePackage(p);
    }
    @Override
    protected boolean checkCommandStart(String command) throws Exception {

        command = Utils.cleanJSONFromContact(command);
        ICommand c = Factory.createCommand(command);
        return c.getType() == TypesCommand.START_MISSION || c.getType() == TypesCommand
    }
    @Override
    protected boolean checkReplyCommandStart(String reply) throws Exception {

        reply = Utils.cleanJSONFromContact(reply);
        IReply r = Factory.createReply(reply);
        return r.getType() == TypesReply.REPLY_OK;
    };
    @Override
    protected String getWrongStartCommandReply() throws Exception {

        Reply r = new Reply(TypesReply.REPLY_FAIL);
        r.setValue("Wrong start mission command");
        return Utils.adaptJSONToContact(r.toJSON());
    }
    @Override
    protected void storeCommandAndReply(String c, String r) throws Exception {
        c = Utils.cleanJSONFromContact(c);
        r = Utils.cleanJSONFromContact(r);
        ICommand command = Factory.createCommand(c);
        IReply reply = Factory.createReply(r);
        env.println("Forwarding: " + command.toString() + " " + reply.toString());
        storage.storeCommandAndReply(command, reply);
    }
    @Override
    protected boolean checkEndMission() throws Exception {
        return fuelLevel <= Fuelometer.MIN;
    }
    @Override
    protected void shutdown() throws Exception {
        if (storage.isOnMission()) {
```

```
            env.println("MISSION END");
            storage.endMission();
        }
    }
    @Override
    protected void storeMissionStarted() throws Exception {
        env.println("MISSION START");
        storage.startMission();
    }
}
```

9.2 SmartDevice

L'applicativo per lo SmartDevice è stato realizzato in un unico progetto, scomponendolo però al suo interno in due package da implementare:

- it.unibo.contact.DroneSmartDashboard
- it.unibo.droneMission.smartDevice.android

e importando al suo interno i package dei progetti che implementano i messaggi (it.unibo.droneMission.messages), le interfacce (it.unibo.droneMission.interfaces.messages e it.unibo.droneMission.interfaces.gauges) e l'implementazione dei gauges (it.unibo.droneMission.gauge), necessaria per ricostruire i dati comunicati dal drone attraverso la stringa JSON.

Inoltre si fa presente che all'interno del pacchetto, nella directory libs, vengono copiate tutte le librerie necessarie affiché l'applicazione possa funzionare: saranno presenti, oltre alla libreria di supporto di Android (android-support-v4.jar) anche la libreria JSON (gson-2.2.4.jar), le interfacce e le librerie di supporto per xtext e per prolog (it.unibo.interfaces_1.6.12.jar, it.unibo.tuprolog_1.0.1.jar e org.eclipse.xtext.xbase.lib_2.2.1.v201112130541.jar).

Il primo package conterrà la business logic del dell'applicazione, mentre nel secondo è contenuta l'implementazione dell'activity. Entrambi i package avranno al loro interno anche tutte le classi di supporto prodotte rispettivamente da contact (utilizzando le specifiche del riportate nel capitolo precedente) e da AAASL, che sulla falsa riga di contact stesso, genera in automatico, a partire da un file .android, tutto ciò che è necessario per renedere l'applicativo funzionante in Android come ad esempio il file AndroidManifest.xml, le classi di supporto per la comunicazione etc. Il file DroneSmardDashboard.android creato dall'application designer è il seguente:

```
AndroidSystem DroneSmartDashboard
avd 15
permissions INTERNET
package it.unibo.droneMission.smartDevice.android
action notifyReceived category "android.intent.category.DEFAULT"
```

```
action startMission category "android.intent.category.DEFAULT"
action updateValues category "android.intent.category.DEFAULT"
action endMission category "android.intent.category.DEFAULT"
Activity SmartDashboard launchable
action startMission
action endMission
action updateValues
action notifyReceived
useLayout layout ;
<Layout> name layout as
  <LinearLayout>
orientation VERTICAL
width FILL_PARENT
height FILL_PARENT
<TextView> output
width FILL_PARENT
height FILL_PARENT
text ""
size 6
background WHITE
textColor BLACK
useOutputForActivity SmartDashboard
</TextView>
</LinearLayout>
</Layout>
```

Analizzando il file nello specifico si può notare che il primo blocco di codice specifica il nome che prenderà l'applicazione, le API da utilizzare, i permessi che richiesti ed infine il nome del package all'interno del quale generare le classi di supporto

```
AndroidSystem DroneSmartDashboard
avd 15
permissions INTERNET
package it.unibo.droneMission.smartDevice.android
```

Il secondo blocco invece contiene la definizione delle Actions, che saranno utilizzate dall'activity, categorizzandole attraverso un intent filter.

```
action notifyReceived category "android.intent.category.DEFAULT"

action startMission category "android.intent.category.DEFAULT"
action updateValues category "android.intent.category.DEFAULT"
action endMission category "android.intent.category.DEFAULT"
```

Si procede quindi a definire l'unica activity dell'applicazione, alla quale saranno associate le Actions definite nel blocco precedente e il layout definito subito dopo.

```
Activity SmartDashboard launchable
action startMission
action endMission
action updateValues
action notifyReceived
useLayout layout ;
<Layout> name layout as
  <LinearLayout>
orientation VERTICAL
width FILL_PARENT
height FILL_PARENT
<TextView> output
width FILL_PARENT
height FILL_PARENT
text ""
size 6
background WHITE
textColor BLACK
useOutputForActivity SmartDashboard
</TextView>
</LinearLayout>
</Layout>
```

Il layout, in questo caso sarà molto semplice e minimale in quanto l'activity non dovrà far altro che visualizzare sul display i dati, per tanto avremo una TextView a tutto schermo di colore bianco sulla quale verranno di volta in volta visualizzati gli ultimi dati ricevuti dal drone.

9.2.1 DroneSmartDashboard

9.2.2 smartDevice.android

Capitolo 10

Installazione

10.1 Head Quarter

10.1.1 Storage

È necessario creare un database MySQL sulla macchina che ospiterà il Server e la ControlUnit. Il context HeadQuarter è stato pensato per girare anche su diverse macchine, ma bisognerà gli indirizzi della FactoryStorage affinché i vari subject possanno contattare corretamente il database.

La control unit e il server si connettono al database con le seguenti credenziali:

- *nome_db*: **dronemission**
- *user*: **dronemission**
- *password*: **estate**

La struttura del database è descritta nel file **init/init.mysql.sql**.

Notare che le fotografie ricevute dal drone saranno salvate nella directory **/media/dronemission**.

10.1.2 Server Web

L'installazione del server web è descritta nel file **init/headquarter.server.web.README_INSTALL**.

In base alle proprie esigenze, sarà necessario aggiornare i vari path riportati nel file.

Capitolo 11

Esecuzione

Nella directory **bin** sono riportati i vari jar e file eseguibili per le applicazioni.
È necessario avviare in sequenza i seguenti file:

1. java -jar updateServer.jar
2. java -jar drone.jar
3. java -jar headquarter.controlunit.jar
4. bash headquarter.webserver.sh

Capitolo 12

Testing

Capitolo 13

Deployment

Capitolo 14

Maintenance