Drone mission

Carlo Antenucci, Leonardo Iannacone, Gonzalo Junquera

12 novembre 2013

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.

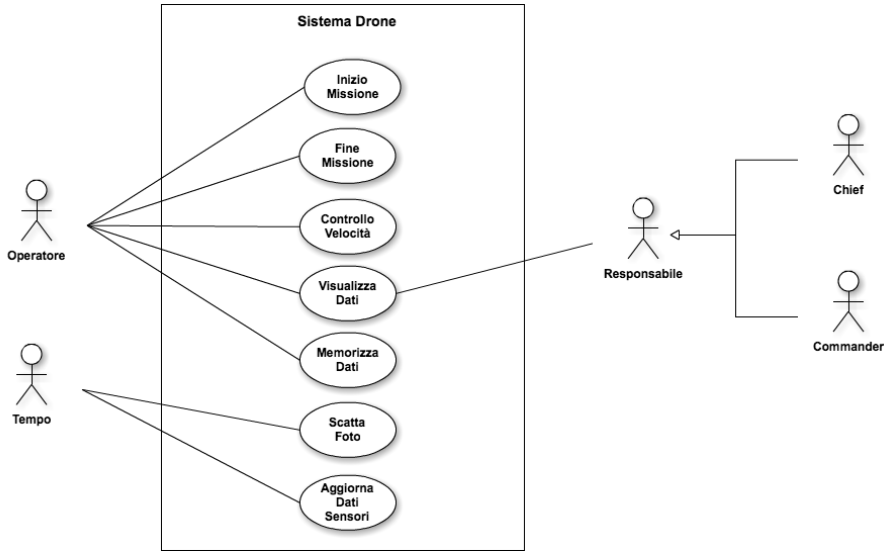


Figura 5.1: Use case

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 -o 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.

```
//Behavior of Drone
```

```
BehaviorOf drone {
    var String msgCommand = ""
    var String cmdName = ""
    var String cmdValue = ""

    var boolean start
    var boolean stop
    var boolean speed

    var String dataSensors
    var String dataPhoto

    action void startMission()
    action void endMission()
    action void setSpeed(String value)

    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 -----")
        doIn receiveCommand()
        set msgCommand = call curInputMsg.msgContent()
        set cmdName = call Drone.getCommandName(msgCommand)
        set start = call cmdName.contains("setspeed")
        if{start} { goToState st_Drone_startMission }
        println ("ERROR: expected 'setspeed' command to start. Received: " + cmdName)
    endstate

    state st_Drone_startMission
        exec startMission() // empty method - maybe can be used in future?
        doOut sendnotify("start")
        goToState st_Drone_setspeed
    endstate

    state st_Drone_setspeed
        set cmdValue =  call Drone.getCommandValue(code.curInputMsgContent)
        exec setSpeed(cmdValue)
        goToState st_Drone_onMission
    endstate

    state st_Drone_onMission
        // send data sensors
        set dataSensors = exec getDataFromSensors()
        doOut sendDataSensors(dataSensors)
        // send photos
        set dataPhoto = exec getDataPhoto()
        doOut sendDataPhoto(dataPhoto) // every x secondi?
        // received a command setSpeed or stop?
        onMessage? command goToState st_Drone_commandHandler
    endstate
```

```
    state st_Drone_commandHandler
        doIn receiveCommand()
        set cmdName = call Drone.getCommandName(code.curInputMsgContent)
        // check if stop
        set stop = call cmdName.contains("stop")
        if {stop} { goToState st_Drone_endMission }
        // check if setspeed
        set speed = call cmdName.contains("setspeed")
        if {speed} { goToState st_Drone_setspeed }
        // get back on mission
        goToState st_Drone_onMission
    endstate

    state st_Drone_endMission
        exec endMission() // empty method - maybe can be used in future?
        doOut sendnotify("end")
        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 (nel codice che segue si è impostato 60 a fini di test) ed attendere l'arrivo di un ACK 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 dataSensorsReceived
    var String photoReceived

    action String getCommandToSend()
    action void updateDashboard(String dataSensorsReceived)
    action void storePhotoData(String photoReceived)

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

    state st_HeadQuarter_ready
        println ("----- Ready to send command -----")
        doOutIn sendCommand("setspeed 60")
        acquireAckFor command goToState st_HeadQuarter_onMission
    endstate
```

```
    state st_HeadQuarter_onMission
        // check if command is clicked in Dashboard and send it
        set command = exec getCommandToSend()
        doOutIn sendCommand(command)
        if {command == "stop"} { goToState st_HeadQuarter_endMission }
        // get sensors data
        onMessage? dataSensor goToState st_HeadQuarter_receivedSensorsData
        // get photos
        onMessage? photo goToState st_HeadQuarter_receivedPhoto
    endstate

    state st_HeadQuarter_receivedSensorsData
        // get sensors data
        doPerceive headquarterReceiveDataSensors()
        set dataSensorsReceived = code.curInputMsgContent
        // update Dashboard
        call updateDashboard(dataSensorsReceived)
        goToState st_HeadQuarter_onMission
    endstate
    state st_HeadQuarter_receivedPhoto
        // get photo data
        doIn receiveDataPhoto()
        set photoReceived = code.curInputMsgContent
        // store info
        call storePhotoData(photoReceived)
        goToState st_HeadQuarter_onMission
    endstate
    state st_HeadQuarter_endMission
        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 smartphone {
    var String notifyContent
    var String dataDroneReceived

    action void notifyUserMissionStarted()
    action void showDataSensorsReceived(String data)
    action void missionFinished()

    state st_Smartphone_init initial
        onMessage notify transitTo st_Smartphone_missionStart
    endstate

    state st_Smartphone_missionStart
        call notifyUserMissionStarted()
```

```
        goToState st_Smartphone_waitingForData
    endstate

    state st_Smartphone_waitingForData
        onMessage? dataSensor goToState st_Smartphone_receivedData
        onMessage? notify goToState st_Smartphone_endMission
    endstate

    state st_Smartphone_receivedData
        // get data from drone
        doPerceive smartphoneReceiveDataSensors()
        set dataDroneReceived = code.curInputMsgContent
        call showDataSensorsReceived(dataDroneReceived)
        goToState st_Smartphone_waitingForData
    endstate

    state st_Smartphone_notifyHandler
        doPerceive smartphoneReceiveNotify()
        set notifyContent = code.curInputMsgContent
        if {notifyContent == "start"}{ goToState st_Smartphone_missionStart }
        if {notifyContent == "end" } { goToState st_Smartphone_endMission }
        goToState st_Smartphone_waitingForData
    endstate

    state st_Smartphone_endMission
        call missionFinished()
        transitToEnd
    endstate
}
```

5.5 Test plan

Capitolo 6

Analisi del problema

6.1 Logic architecture

Il sistema tutto viene suddiviso in tre sottisistemi Drone - HeadQuarter - SmarDevice 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

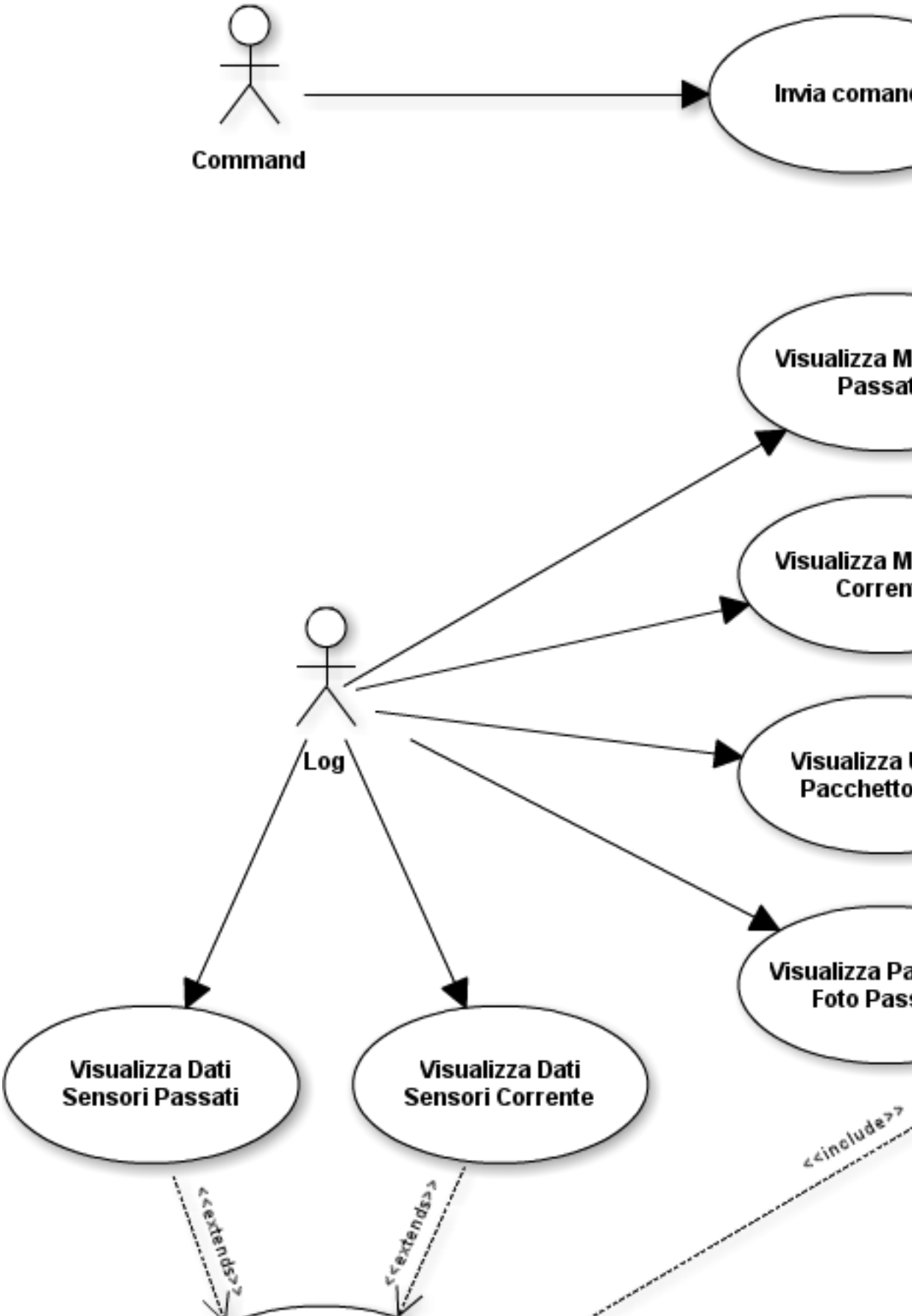
#FIX_ME: QUI CI ANDREBBE LA DESCRIZIONE DEL DRONE E DEI GAUGES

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 Casi d'uso

Al fine di analizzare la centrale operativa vengono proposti i seguenti casi di uso secondo le specifiche:


Command
Invia comand
Log
Visualizza M
Passat
Visualizza M
Corren
Visualizza
Pacchetto
Visualizza Pa
Foto Pass
Visualizza Dati
Sensori Passati
Visualizza Dati
Sensori Corrente
<<extends>>
<<extends>>
<<include>>

#FIX_ME: aggiungere descrizione casi d'uso

6.1.3.2 Struttura

Analizzando la centrale di controllo, si decide 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;
```

6.1.3.3 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].

6.1.3.4 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
    // 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 String getWrongStartCommandReply()
action void storeCommandAndReply(String c, String r)
// shutdown
action boolean checkEndMission()
action void shutdown()
state st_controlUnit_init initial
    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
    call storeMissionStarted()
    goToState st_controlUnit_sendCommand
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
    set cmd = code.curInputMsgContent
    doOutIn sendCommand(cmd)
    acquireAnswerFor command
    set rpl = code.curInputMsgContent
    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 picturePackages
            onMessage? picturePackage goToState st_controlUnit_receivedpicturePackage
            // get sensors data
            onMessage? sensorsData goToState st_controlUnit_receivedSensorsData
            // 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()
            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

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

Capitolo 10

Testing

Capitolo 11

Deployment

Capitolo 12

Maintenance