Alma Mater Studiorum · Università di Bologna
Campus di Cesena

SCUOLA DI INGEGNERIA E ARCHITETTURA
Corso di Laurea Magistrale in Ingegneria Informatica

Model Driven Design for Embedded Systems

Tesi di Ingegneria dei Sistemi Software

Relatore:
Chiar.mo Prof.
Antonio Natali

Correlatore:
Dott.
Sam Golovchenko

Presentata da:
Marco Vasapollo

Sessione III
Anno Accademico 2012/13

All'indispensabile e sempre presente
sostegno della mia famiglia.

Introduzione

La natura dei sistemi embedded classici è sempre stata saldamente legata a tecnicismi specifici altamente specializzati, come i controllori del mondo dell'industria, sia per il contesto storico in cui tali sistemi hanno fatto la loro prima comparsa, sia per questioni di necessità legate all'efficienza che solo un sistema elettronico puro poteva offrire fino ad ora. Questa natura è rimasta invariata anche quando essi sono stati implicati in scenari di utilizzo meno esigenti rispetto a quello industriale, come la domotica o la robotica di largo consumo. Tuttavia, l'avvento di piattaforme integrate general purpose dai costi contenuti, con le quali è possibile realizzare apparati hardware e software orientati al neologismo dell'Internet of Things ([Cha10]), ha permesso ai sistemi embedded di compiere un significante salto evolutivo, acquisendo così nuove caratteristiche che ne aumentano non solo la rosa di possibili scenari applicativi, ma anche il pubblico interessato alla realizzazione di sistemi originali e sempre più avanzati. Tali piattaforme, come Arduino ([Tim05]) o Raspberry Pi ([Sch12]), nascono da enti non correlati tra loro ed hanno perciò caratteristiche differenti nel modo di approcciarsi con l'utente. Si hanno quindi a disposizione diversi standard de facto tra cui scegliere, con caratteristiche spesso contrastanti tra di loro. L'utente si vedrà quindi costretto a dover imparare ad utilizzare un pool di concetti differenti che renderanno parecchio difficoltosa l'operazione di adattamento di un'applicazione ad un eco-sistema diverso da quello per cui essa è stata pensata e realizzata per la prima volta. Si sente quindi il bisogno di avere a disposizione una visione concettuale ad alto livello di ciò che le varie piattaforme

Embedded di nuova generazione offrono, associando poi tali concetti alle intenzioni degli utenti. Uno degli approcci utilizzabili per ottenere ciò è quello di astrarre la 'filosofia' di utilizzo di ogni piattaforma. Gli spazi concettuali così ottenuti potranno poi essere fattorizzati in costrutti sintattici formali e dalla semantica disambigua, rappresentabili tramite DSL, ossia Domain System Languages, i quali daranno così la possiblità di definire dei Modelli di Sistemi Software, descrivendone le proprietà in termini di Struttura, Interazione e Comportamento. Inoltre, grazie ad opportuni generatori di codice di programmazione, i DSL diventeranno un nuovo strumento per realizzare del software multi-piattaforma, ma che segue esattamente i desideri dell'utente che lo commissiona, su ogni eco-sistema. L'obiettivo di questa tesi è quello di mostrare i benefici dell'utilizzo della modellazione di un sistema che verrà poi implementato su piattaforme differenti. Come esempio pratico si è scelto di progettare un Sistema formato da vari Robot realizzati in piattaforme implementative differenti, ma comunque dipendenti dallo stesso Modello. Tali Robot sfruttano altresì il neologismo dell'Internet Of Things per comunicare tra loro, utilizzando un protocollo di interazione anch'esso frutto di un Modello realizzato a priori. La Modellazione dell'interazione sfrutta degli spazi concettuali personalizzati in grado di definire ad alto livello nuove forme di comunicazione. Tali concetti vengono espressi utilizzando dei Domain Specific Languages (DSL) associati a generatori di codice per facilitare il lavoro di implementazione dei prototipi.

Indice

Elenco delle figure

Capitolo 1

Sistemi Embedded

1.1 I Sistemi Embedded classici, specific purpose e solo per tecnici

I più disparati scenari della vita quotidiana coinvolgono dispositivi composti da hardware e software che non sono percepiti come veri e propri computer. Essi possono avere periferiche di input e di output ben diversi da quelli tradizionali, come tastiera e schermo, nonché eseguire sistemi operativi non standard. Possono altresì essere autosufficienti, ma anche far parte di aggregati meccanici più grandi. Tali dispositivi elettronici integrati vengono chiamati Sistemi Embedded ([Dor06]).
Il successo dell'elettronica integrata è stato reso possibile grazie alle straordinarie capacità delle tecnologie per la produzione di circuiti integrati, che permettono di costruire dispositivi sempre più complessi, e all'impiego di nuove metodologie progettuali che permettono di offrire la massima efficienza. Il tradizionale sviluppo di strumenti puramente meccanici, d'altra parte, aveva raggiunto il suo picco massimo alla metà del 20° secolo, diventando quindi una fonte non significativa di innovazione per l'era moderna, almeno fino a quando tale settore non è stato affiancato dalle tecnologie di produzione di dispositivi elettronici (MicroElectroMechanical Systems MEMS), o Sistemi Embedded. Ci sono molti esempi di sistemi embedded nel mondo reale. Per

esempio, una automobile moderna contiene decine di componenti elettronici (centraline, sensori e attuatori) che svolgono compiti molto diversi. Il primo sistema embedded apparso in una auto era legato al controllo degli aspetti meccanici, come il controllo del motore, sistema di frenatura antibloccaggio, controllo delle sospensioni e trasmissione. Tuttavia, al giorno d'oggi le auto hanno anche una serie di componenti che non sono direttamente collegate agli aspetti meccanici, ma sono per lo più connessi all'uso della macchina come veicolo per muoversi, o per esigenze di comunicazione e di intrattenimento dei passeggeri: sistemi di navigazione, lettori audio e video digitali e telefoni sono solo alcuni esempi. Inoltre, molti di questi sistemi integrati sono collegati tra loro mediante una rete, perché devono condividere le informazioni sullo stato dell'auto.

Altri esempi provengono dal settore della comunicazione. Un telefono cellulare è un sistema integrato il cui ambiente è la rete mobile. Si tratta di computer molto sofisticati il cui compito principale è quello di avviare e ricevere telefonate, ma sono anche attualmente utilizzati come assistenti digitali personali, per i giochi, per inviare e ricevere immagini e messaggi multimediali e per navigare su Internet in modalità wireless. In soli dieci hanni sono diventati essenziali nella nostra vita.

Siamo solo all'inizio di una rivoluzione che avrà un impatto su ogni altro settore industriale. Ovviamente, questa rivoluzione porta con sé delle conseguenze in ogni settore. Ad esempio, i costruttori di automobili hanno bisogno di acquisire una notevole quantità di abilità nella progettazione hardware e software, oltre alle competenze meccaniche già in possesso, o dovrebbero esternalizzare i requisiti ad un fornitore esterno. In entrambi i casi, un'ampia varietà di competenze deve essere messa in gioco, dalla progettazione di architetture software per l'implementazione delle funzionalità alla modellazione delle prestazioni, come ad esempio la gestione real-time dei dati in applicazioni safety-critical. I progettisti di sistemi embedded devono inoltre essere in grado di progettare e analizzare le prestazioni delle reti, nonché convalidare le funzionalità implementate su una particolare architettura e i

protocolli di comunicazione utilizzati. Una rivoluzione simile è accaduta o sta per accadere, in altri settori industriali e socioeconomici come l'intrattenimento, il turismo, l'istruzione, l'agricoltura, il governo, e così via.
E' quindi chiaro che bisognerà sviluppare nuove metodologie più semplici per lo sviluppo di apparati elettronici integrati, in modo da permettere all'industria di utilizzare tutte le tecnologie d'avanguardia. I sistemi embedded sono informalmente definiti come un insieme di parti formate da circuiti integrati programmati per eseguire delle azioni specifiche (Application Specific Integrated Circuit, ASIC) e altri componenti standard (Application Specific Standard Product, ASSP), che interagiscono continuamente con l'ambiente attraverso sensori e attuatori. La collezione può fisicamente essere un insieme di chip-on-board, o un insieme di moduli su un circuito integrato. Il software viene utilizzato per funzionalità e flessibilità, mentre l'hardware dedicato è utilizzato per incrementare le prestazioni ridurre i consumi. I principali componenti programmabili sono microprocessori e DSP (Digital Signal Processors) che implementano la parte software del sistema. Essi presentano aree di utilizzo, costi, prestazioni e caratteristiche elettriche poste a metà tra hardware dedicato e processori. Componenti hardware programmabili dedicati, invece, implementano blocchi di applicazione per specifiche periferiche. Tutti i componenti sono collegati tramite bus di comunicazione standard e dedicati, e le reti e i dati vengono memorizzati su un insieme di memorie.

Spesso i vari sottosistemi più piccoli sono collegati in rete per controllare, ad esempio, una macchina intera, o per costituire una rete cellulare o wireless. Possiamo identificare un insieme di caratteristiche tipiche che si trovano comunemente nei sistemi embedded. Per esempio, di solito non sono molto flessibili e sono progettati per eseguire sempre lo stesso compito. Se si acquista un sistema embedded per il controllo del motore, non è possibile utilizzarlo per controllare i freni dell'auto o per giocare. D'altra parte un PC è molto più flessibile perché può eseguire diverse operazioni molto diverse. Un sistema embedded è spesso parte di un sistema di controllo più grande. Inoltre, il costo, l'affidabilità, le prestazioni e la sicurezza sono spesso

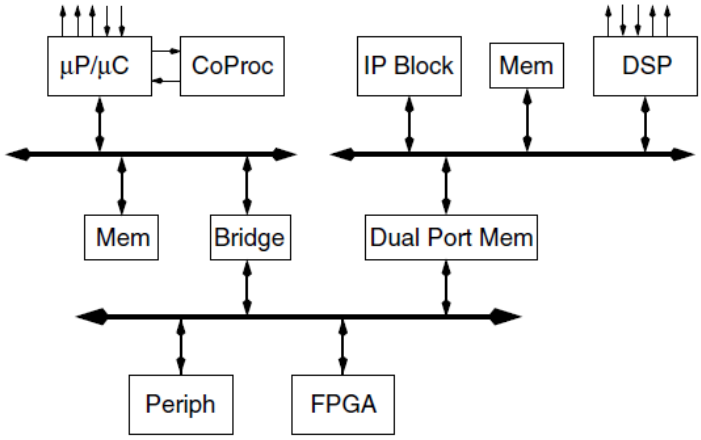

Figura 1.1: Architettura di un Sistema Embedded con reattività realtime. ([LP05])

criteri più importanti delle funzionalità, in quanto il cliente può anche non essere a conoscenza della presenza del sistema integrato, e così guarda altre caratteristiche, quali il costo, la facilità d'uso, o la durata di un prodotto. Un'altra caratteristica comune a molti sistemi embedded è che essi devono essere progettati in un periodo di tempo estremamente breve per soddisfare il time-to-market. E' necessario che ci siano solo pochi mesi di distanza tra la concezione del prodotto e i suoi primi prototipi funzionanti. Se non vengono rispettate queste scadenze, il risultato è un aumento dei costi di progettazione con conseguente diminuzione dei profitti, perché meno articoli saranno venduti. Ritardi del ciclo di progettazione possono così rendere un enorme differenza tra un prodotto di successo e uno senza successo. Attualmente, le metodologie di progettazione dei sistemi embedded utilizzano un approccio ad hoc che si basa per lo più su precedenti esperienze con altri prodotti simili e sul design manuale. Spesso il processo di progettazione richiede varie iterazioni per ottenere il risultato voluto, perché il sistema non è specificato in modo rigoroso e senza ambiguità, e il livello di astrazione, nonché le metodologie di design possono essere differenti. Poiché la complessità dei sistemi

embedded è destinata a salire, questo approccio sta mostrando i suoi limiti, soprattutto per quanto riguarda i tempi di progettazione e collaudo.

1.2 I Sistemi Embedded moderni e l'Internet Of Things

Negli ultimi anni ha fatto la sua comparsa sul mercato una pletora di dispositivi Embedded di nuova generazione aventi come punti di forza l'eterogeneità di utilizzo, il basso costo e la discreta potenza computazionale. Tali dispositivi, come il Raspberry Pi (fig. 1.2), hanno modificato la percezione che il grande pubblico ha dei sistemi embedded, sempre troppo legati al mondo industriale. Infatti essi non sono realizzati per un unico scopo, ma sono bensì delle piattaforme general-purpose provviste di connettori generici in grado di interfacciarsi facilmente con numerosi dispositivi esterni. I sistemi embedded diventano così degli strumenti alla portata di tutti, e possono essere impiegati nei più disparati scenari di utilizzo, a discrezione dell'utente che li usa per progettare le sue idee, senza il bisogno di ricorrere a tecnicismi troppo specifici. Con il neologismo dell' Internet Of Things diventa poi ancora più facile realizzare una rete di questi dispositivi in grado di interfacciarsi con l'utente e di aiutarlo nelle azioni della vita quotidiana.

'L'Internet of Things (abbreviato, IoT) è un neologismo che descrive il coinvolgimento e la collaborazione tra utenti e dispositivi elettronici impiegati quotidianamente, i quali si vedono impegnati in una comunicazione continua e su larga scala, nelle combinazioni 'uomo-macchina', 'macchina-uomo', 'macchina-macchina' o 'uomo-uomo', che sfrutta le più moderne tecnologie di connessione alla rete internet per agevolare e automatizzare le azioni della vita quotidiana tramite condivisione di *conoscenze*' ([Cha10]). L'IoT nasce ufficialmente nel 2005, quando la International Telecommunication Unit (ITU) mostra interesse per il lavoro svolto da EPCglobal ed EAN UCC (European Article Numbering Uniform Code Council) per portare all'interno delle realtà industriali il concetto di EPC (Electronic Product Code), partorito a

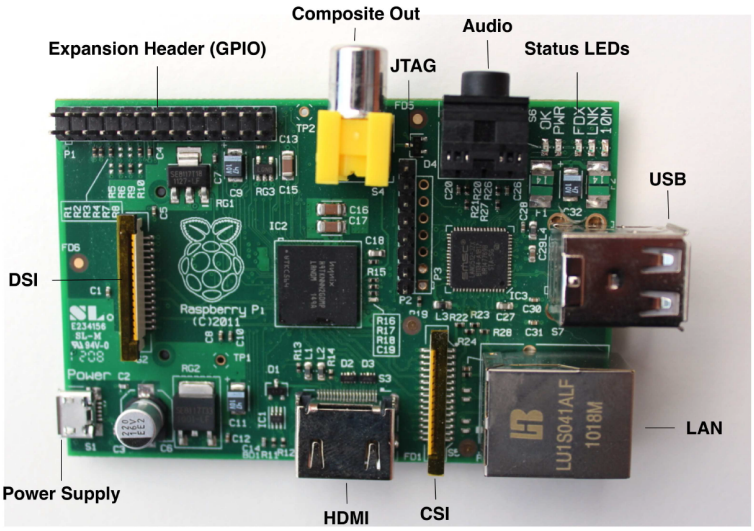

Figura 1.2: Il Raspberry Pi

sua volta dal centro di ricerca Auto-ID del MIT (Massachusetts Institute of Technology). L'obiettivo di EPC era quello di referenziare a livello virtuale (di software) ogni prodotto elettronico presente nella vita reale, e di introdurlo in una catena di comunicazione gestita a livello informatico. Il concetto era quindi quello di realizzare una connessione tra il mondo digitale, rappresentato da un 'network of bits' (la più piccola forma di comunicazione in informatica), e il mondo reale, rappresentato da un 'network of atoms' (il più piccolo elemento della materia).

L'ITU ha prodotto quindi un documento di resoconto su IoT da numerosi punti di vista, tecnico, economico, etico, ecc. Il risultato è l'unione delle due caratteristiche salienti della connettività, ossia 'anywhere' and 'anytime', alla nuova caratteristica, 'anything', che introduce le interazioni del tipo *thing-to-thing* o *machine-to-machine*, oltre a quelle già esistenti *person-to-person* e *person-to-machine* ([Uni05]).

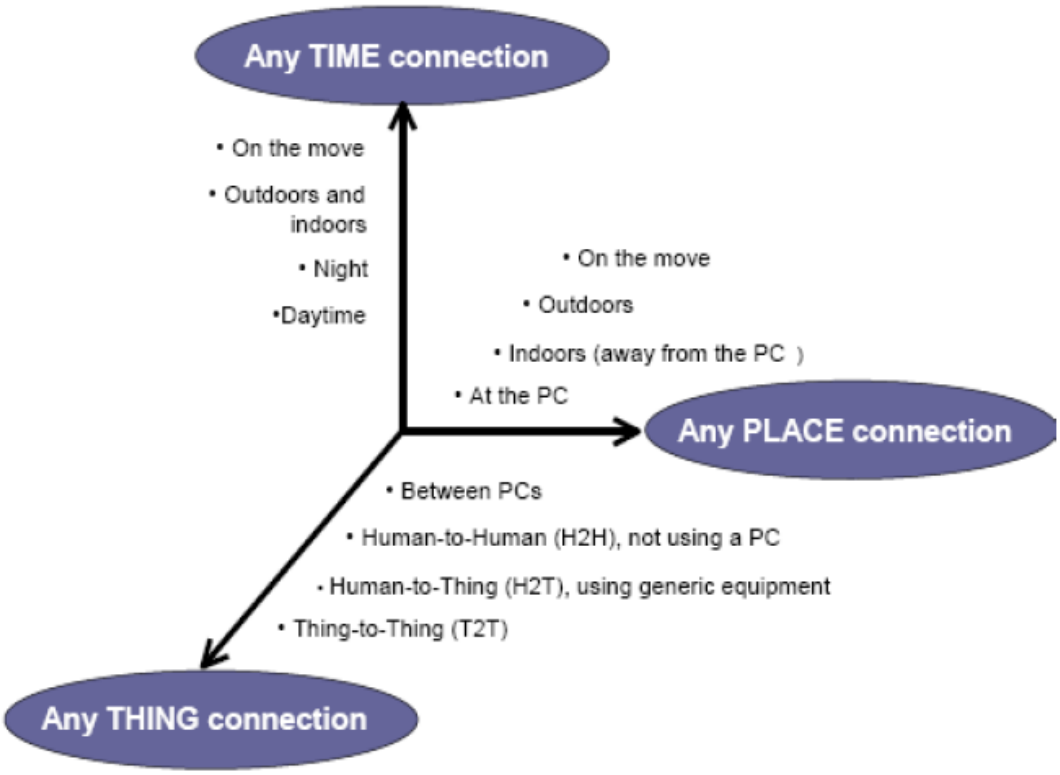

Figura 1.3: Principi dell'Internet of Things

Capitolo 2

Il Model Driven Design

2.1 Origini

Fin dai primi utilizzi pratici del computer, gli sviluppatori hanno sempre sentito il bisogno di astrarre i concetti pratici di una data applicazione dai tecnicismi propri della piattaforma implementativa che la concretizza. La necessità era particolarmente sentita con linguaggi di basso livello come Assembly. Un primo tentativo di realizzazione di strumenti pratici di astrazione è stato effettuato nel 1980 con il Computer Aided Software Engineering (CASE), una corrente di studi che mirava alla realizzazione di strumenti grafici per la definizione astratta di softwares attraverso convenzioni come diagrammi strutturali, macchine a stati finiti, diagrammi del flusso dei dati. Malgrado il notevole interesse suscitato da CASE in ambito ricercativo, esso venne utilizzato poco in ambito produttivo perché, sebbene capace di modellare i software ad alto livello su piattaforme single-node come DOS e OS/2, non gestivano la modellazione di elementi corollarici al software stesso, ma comunque di vitale importanza, come il Quality Of Service, la sicurezza oppure la scalabilità dei sistemi. Inoltre, poiché si parla di piattaforme implementative agli albori, che quindi non avevano le capacità di risolvere tutti i problemi, il codice prodotto grazie alla modellazione attraverso questi strumenti, dovendo colmare le lacune delle piattaforme, diveniva spesso in-

gestibile da manutenere.

Il problema sembrò comunque arginarsi (ma assolutamente NON risolversi) da una parte, con la nascita di piattaforme implementative basate su concetti logici più articolati, utili quindi a coprire più casi di utilizzo, che però aumentarono esponenzialmente di complessità. In più la diffusione dei sistemi distribuiti ha ulteriormente aggravato il peso dato al programmatore, che non si trova più a dover affrontare i problemi su un singolo nodo. A discapito di una buona architettura, gli sviluppatori hanno sempre preferito quindi violare principi teorici di ingegnerizzazione del codice, realizzando soluzioni ad hoc per un dato problema, sacrificando la portabilità del codice, la sua comprensione e la sua apertura alla manutenzione ([Sch06]).

2.1.1 La Model Driven Engineering

Un modo promettente di affrontare la complessità delle piattaforme implementative *general-purpose* di terza generazione nel risolvere problemi *domain-specific* è sicuramente quello di utilizzare la Model Driven Engineering (MDE). Essa combina l'utilizzo di **linguaggi di modellazione di uno specifico dominio applicativo**, i quali riescono ad esprimere in modo formale e dismbiguo struttura, comportamento e requisiti di una determinata classe di problemi, a motori di trasformazione e generazione per la sintetizzazione di tali sintassi e semantiche dei modelli, come codice sorgente o descrittori di deployment. Le nuove tecnologie di MDE prendono quindi spunto dall'immane sforzo necessario per ritagliare a piacimento di uno specifico dominio, scaturito da metamodelli, i concetti di piattaforme generiche che vincolano la realizzazione del software, dirigendo così il focus sulla modellazione formale del problema e non sull'implementazione finale ([Sch06]). Il Model Driven Design (MDD) rappresenta quindi tutto l'insieme di approcci che utilizzano la MDE per la realizzazione di Sistemi Software.

Model-Centric Software Development

Il Model-Centric Software Development è una tecnica di MDD con la quale si definisconno TUTTE le fasi di progettazione del sistema avendo sempre al centro il Modello. Questo sta a significare che ogni concetto del Modello definito ad un determinato step di progettazione di un sistema non è una sua derivazione del Modello realizzato allo step precedente, ma è un nuovo Modello che trova una corrispondenza biunivoca col Modello concepito precedentemente, che sarà sostanzialmente visto da un punto di vista differente rispetto a prima, più specifico. Questo, ovviamente, allontana immediatamente l'idea di avere un linguaggio di modellazione generico per tutte le fasi, ed introduce quindi il concetto di più linguaggi specifici che non perdono alcun tipo di dettaglio del sistema. Ne consegue che il mapping tra gli elementi del Modello e la loro implementazione specifica è ben definito. Ciò consente di ottenere non un semplice scheletro dell'applicazione con parti da implementare, ma un artefatto della maggior parte degli elementi della business-logic del Sistema finale, **totalmente svincolato dalla piattaforma di implementazione**. L'obiettivo su cui bisogna focalizzare l'attenzione è comunque quello di non vincolare il Modello iniettando repliche dei livelli di astrazione di piattaforme specifiche, al fine di mantere la portabilità. [WL06]

2.2 I principi chiave del MDD

Lo scopo che il Model Driven Design si predilige di raggiungere è quello di modellare in modo astratto un sistema: ciò significa elaborare a livello logico quante più informazioni possibili. La fase successiva sarà trasferire e adattare tutte queste informazioni nell'implementazione concreta del sistema. I capisaldi sono quindi due:

- Modello (Model): definisce e raccoglie tutte le informazioni sul sistema, relazionandole a livello logico nella maniera più comprensibile possibile,

al fine di rendere chiaro a chiunque da quali parti è composto il sistema, come queste parti sono relazionate e qual'è il loro scopo;

- Codice: è a sua volta un Modello definito coi costrutti propri della piattaforma implementativa che eseguirà il sistema, la quale però potrebbe non rispecchiare a pieno i requisiti e le necessità del sistema stesso.

Come è facile intuire, la canalizzazione da modello a codice (model-to-code) non sempre è possibile da effettuare in modo diretto, perché potrebbero esserci delle lacune in entrambi i side della trafila, come carenza di espressività del modello oppure impossibilità di implementazione in codice di programmazione di alcune sue parti.
Di seguito verranno analizzati i vantaggi di avere un Modello alla base del software, le tecniche, gli strumenti e le convenzioni utilizzate per una sua buona progettazione e presentazione, e verrà altresì messo in luce il modo in cui è possibile passare facilmente dai Modelli al Codice senza perdere informazioni preziose sulla sua espressività, colmando comunque, ove possibile, l'Abstraction Gap che si crea tra esso e la piattaforma implementativa concreta ([NM08]).

2.2.1 Model

E' evidente che il Modello rappresenta il principio cardine del MDD, ed è la ragione per cui il processo di realizzazione del software tramite questa tecnica è detto *model driven*. Esso rappresenta in modo astratto tutto il sistema che si vuole progettare. Le sue caratteristiche di entità astratta risultano essere molto utili in varie fasi della progettazione, e se ne colgono i benefici fin dai colloqui preliminari col committente del sistema, in quanto egli potrebbe essere una persona totalmente estranea al mondo dello sviluppo software o addirittura dell'informatica in generale. Il modello è quindi una sorta di ponte di comunicazione che unifica i punti del triangolo formato dal committente, dal progettista (o analista) e dall'implementatore. Quest'ultima frase mette in luce un principio fondamentale del modello, ossia la sua

assoluta comprensibilità. E' bene quindi acquisire dimestichezza col modo di realizzare i modelli, i quali devono essere semplici ma comunque ben strutturati, comprensibili e soprattutto meccanizzabili (ossia facilmente convertibili in codice di programmazione). Un'altra caratteristica peculiare dei modelli è che essi debbano essere espressi, ove possibile, senza ricorrere all'utilizzo di concetti propri di piattaforme implementative, cercando quindi di astrarsi da qualsiasi tecnologia, e favorendo così la portabilità dei principi del sistema su ecosistemi differenti. Come accennato, non sempre questa portabilità è realizzabile con poco sforzo, infatti potrebbe esserci un divario tra l'astrazione del sistema, ossia il modello, e la piattaforma implementativa: l'*Abstraction Gap*.

Definizione di un modello

Il modello racchiude in sé tutte le caratteristiche di un sistema. Tali caratteristiche possono spaziare in base alle necessità individuali di un progetto, ma tre di queste, le principali, sono comuni a tutti, e riescono a descrivere la maggior parte del sistema: si tratta di Struttura, Interazione e Comportamento ([NM08]).

Struttura: Definisce l'architettura complessiva del sistema in modo *ricorsivo*, ossia definendo prima ogni macro-elemento del sistema e successivamente la struttura di ogni sottosistema che costituisce il macro-elemento. Per impostare in modo sistematico la definzione a livello strutturale di un elemento può essere conveniente cercare di dare risposta ad alcune domande quali:

1. l'elemento è *atomico* o *composto*? (Ricorsivamente) quante e quali sono le parti che compongono un elemento composto?
2. l'elemento è *dotato di stato modificabile*?
3. quali sono le *proprietà* dell'elemento, cioè quali *attributi* lo caratterizzano?

4. da quali altri elementi *dipende* e secondo quale forma di dipendenza?

Interazione: Definisce il modo che le parti del sistema hanno di comunicare tra di loro. A prima vista potrebbe sembrare una caratteristica di secondo ordine, ma in realtà è la dimensione più importante, in quanto mette in luce numerosi aspetti di collegamento tra le parti che la dimensione di Struttura non riesce a descrivere. Essa può essere *sincrona* oppure *asincrona* e può avvenire tramite concetti quali *procedure call* (detta anche cessione del flusso di controllo) oppure *message passing*.

Comportamento: Il comportamento descrive come ogni elemento del sistema attua il suo ruolo per ottemperare allo scopo prefissato. Uno dei sistemi molto efficiaci, ma non l'unico, che possono essere utilizzati per descrivere la dimensione di comportamento è quello di definire l'elemento come un Automa a Stati Finiti (ASF) ([NM08]).

Passi per la definizione del modello di un sistema

Come già accennato, la modellazione del sistema inizia a partire dai primi colloqui col committente, seguendo passo passo i vari stati di progettazione a cui sottoporre il sistema e che coinvolgono analisti, designer, e programmatori ([NM08]):

Analisi dei Requisiti: In questa fase tutto ciò che è documentazione redatta dal committente (e/o con la sua supervisione) è passato al setaccio per individuare i punti salienti utili alla progettazione del modello. In particolare questa fase permette di realizzare una parte molto preliminare di Struttura del modello, ovvero di come il committente vede il sistema: in particolare vengono individuati nei testi tutti i *sostantivi* rilevanti, che diventeranno gli elementi strutturali del modello, e verranno arricchiti grazie alle informazioni ottenute dai *verbi* associati, i quali forniranno informazioni quali funzionalità e proprietà di ogni elemento. Il Comportamento e le caratteristiche di tali funzionalità e

proprietà devono essere ulteriormente analizzati al fine di coglierne il vero scopo, e quest'ultimo deve essere assolutamente chiaro fin dal principio, descrivendolo attraverso dei Piani di Collaudo che ne mettono in risalto il funzionamento. E' di vitale importanza che il risultato ottenuto da questa prima modellazione venga visionato e soprattuto validato dal committente, in quanto punto di partenza che porterà poi alla meccanizzazione e alla successiva implementazione dei concetti del sistema, il quale deve appunto rispecchiare appieno le volontà del committente.

Analisi del Problema: Il modello scaturito dall'Analisi dei Requisiti è difficilmente inquadrabile come modello finale. Ci saranno infatti numerosi punti oscuri che dovranno essere ulteriormente visionati e analizzati al fine di caprirne il corretto funzionamento di tutte le parti. Questa tecnica incrementale, (quì denominata '*zooming*'), fa si che il sistema (e il suo modello) venga definito per livelli: ogni qual volta un livello è stato 'completato', ossia se ne è ricavato un modello sufficentemente esaustivo e dettagliato, si passa al livello inferiore. Raramente è possibile che i livelli superiori debbano essere cambiati per meglio adattarsi alle caratteristiche dei livelli inferiori. Scopo dell'analisi del problema è quello di ottenere la cosiddetta *Architettura Logica* del sistema, ossia un modello il più possibile concreto e in grado di definire una volta per tutte il modello visto dalle tre *Dimensioni* principali. Ma l'analisi del Problema può mettere in risalto anche delle lacune da parte del modello o dell'infrastruttura operativa concreta che servirà dall'implementazione, questo buco è chiamato per l'appunto '*Abstraction Gap*', ossia un'incongruenza tra quelli che sono i desideri del progetto e quello che la piattaforma implementativa scelta per lo stesso.

Progettazione: L'Architettura Logica diventa la base di lancio per l'istanziazione delle varie implementazioni del (modello del) sistema. Una volta ottenuta nel modo più astratto possibile, la si può adattare alla piattaforma implementativa desiderata, avendo cura di realizzare un

metodo di conversione e di interpretazione dei concetti del modello con i paradigmi propri del'eco-sistema di destinazione. Tale porting dei concetti a volte può non essere immediato, ed è necessario per cui che alcuni dei concetti del modello siano implementati attraverso opportune infrastrutture da porre tra i due lati, i *middleware*, i quali hanno il compito di limitare l'Abstraction Gap.

E' importante notare che durante lo svolgimento di uno dei passi su indicati ci si renda conto che è necessario effettuare delle modifiche al passo precedente. Le modifiche effettuate implicano la modifica a cascata dei passi successivi.

Come rappresentare un modello - UML

Uno degli standard utilizzati per esprimere modelli è sicuramente UML. Esso mette a disposizione diversi tipi di diagrammi utili per modellare le tre dimensioni che descrivono i modelli. Ad esempio per esprimere la dimensione di Struttura possono essere utilizzati i Class Diagram, ai quali è tra l'altro possibile associare i Design Pattern ([GHJV94]) di tipo strutturale, ideati per lo scopo. Dopodiché vi sono Package Diagram, Component Diagram, Deployment Diagram. Per la dimensione Interazione sono molto importanti i diagrammi di Sequenza, di Comunicazione, di Interaction Overview, Temporali. Come già detto, il Comportamento può essere modellato tramite automi a stati finiti, definibili in UML tramite gli State Diagram.

I limiti di UML

Benché gli UML rappresenti uno standard riconosciuto per la descrizione formale e disambigua, nonché meccanizzabile dei modelli, potrebbero esserci dei costrutti di alto livello che UML non è in grado di esprimere al pieno delle loro potenzialità, come ad esempio non è possibile, per la dimensione di Interazione, definire una comunicazione basata sullo scambio di messaggi (pag 55 di [NM08]).

Ampliamento dello spazio concettuale e rappresentazione '*dedicata*' - DSL

Realizzare il modello di un sistema significa censirne le caratteristiche e le potenzialità, avvalendosi di uno spazio concettuale altresì detto *meta-modello* per uno specifico dominio applicativo ([NM08]). Ad esempio, UML è un meta-modello che specifica modelli, ed è a sua volta istanza del meta-meta modello scaturito dallo standard MOF (Meta-Object Facility) realizzato da OMG (Object Management Group, [Gro95]). Al di là della rappresentazione grafica/iconica, esiste tuttavia un altro modo per esprimere un modello in modo disambiguo e formale, ossia utilizzare dei Domain Specific Languages (DSL, pag 205 di [NM08]). Tali linguaggi sono appunto ritagliati per descrivere una data classe di modelli, esprimendo i concetti comuni tramite parole chiave, che diventano delle *primitive*. Ad esempio, per rimediare alle lacune dell'espressività della semantica in termini di Interazione di tipo message-passing, è possibile ampliare lo spazio concettuale tramite la specifica Contact ([Nat09]), con il quale è possibile realizzare modelli rappresentandoli tramite un DSL realizzato ad hoc.

2.2.2 Introduzione al metamodello Contact e al suo DSL

Contact è nato con l'intento di colmare il gap di espressività di UML riguardo la modellazione delle interazioni. Strutturalmente, esso introduce i *Subject* ossia entità proattive che si scambiano *messaggi* di varia natura, che rappresentano la parte di Interazione:

- *request*: Messaggio punto-a-punto che prevede una risposta dalla controparte;
- *dispatch*: Messaggio punto-a-punto che non prevede una risposta dalla controparte;

- *invitation*: Messaggio punto-a-punto che prevede un ack dalla controparte;

- *signal*: Messaggio uno-a-molti senza nessuna informazione sui destinatari e che non prevede alcuna risposta.

Per definire modelli tramite una specifica custom, vengono realizzati dei Domain Specific Languages (DSL), che permettono di esprimere i concetti della specifica tramite una sintassi grammaticale definita a priori.

2.2.3 I Benefici del Modello definito da un DSL come Contact: la generazione del codice implementativo

Un DSL, oltre a dare la possibilità di esprimere dei concetti ad hoc, ha comunque in se tutte le caratteristiche di un qualsiasi altro linguaggio: data la sua natura disambigua può essere dotato di un interprete che, analizzandone la sintassi, costruisce un Abstract Syntax Tree (AST), ossia la base con cui poi i linguaggi di programmazione trasformano il codice sorgente in binario. Esistono dei tool in grado di realizzare, una volta definite le regole grammaticali, un interprete che generi l'AST e da quì, tramite degli opportuni motori, generare il codice sorgente per ogni linguaggio di programmazione desiderato, abbattendo notevolmente i costi di produzione dei prototipi, ma anche del software vero e proprio. Uno di questi tool è XText ([Xte13]) per Eclipse. La sintassi del DSL Contact è stata realizzata proprio con XText, e grazie ai suoi tool sono stati realizzati i generatori di codice che trasformano un listato (**Modello**) Contact in un sistema Java funzionante. Per una prima implementazione dei generatori è stato scelto Java perché uno dei linguaggi di programmazione più usati. Ovviamente questa è solo una scelta, ma a nessuno impedisce di realizzare generatori multipli per ogni qualsivoglia linguaggio, anche funzionanti in parallelo. Per facilitare ulteriormente la portabilità cross-platform del neologismo Contact, i generatori sono mediati da

un middleware che esprime il modello in concetti di ancora più basso livello ([Nat09]).

2.2.4 Modellazione di Software IoT-based per Sistemi Embedded Moderni - Limiti all'interazione Thread-based

Grazie ai nuovi Sistemi Embedded, i cui processori danno la possibilità di eseguire applicazioni scritte in linguaggi di alto livello, è possibile realizzare facilmente applicazioni che si avvalgono dell'Internet Of Things. Le caratteristiche general purpose di una piattaforma come Raspberry Pi, associate ad elementi elettronici, come sensori ed attuatori, possono dare vita a dispositivi personalizzati 'intelligenti' che comunicano e cooperano tra loro in scenari di domotica o robotica. Poiché, come si è visto, è estremamente facile scrivere software di taglio classico per le nuove piattaforme embedded, è quindi possibile adoperare tutte le metodologie di sviluppo model driven viste fino ad ora. Inoltre, visto e considerato che Internet of Things è un neologismo strettamente legato alla comunicazione tra entità, uno spazio concettuale come Contact, basato sullo scambio di messaggi, torna parecchio utile.
A fronte di una buona progettazione, vi sono però dei gap di fondo nell'utilizzo di tecniche di implementazione old-style, ad esempio Thread Based: le applicazioni che si avvalgono dell'Internet of Things, per definizione, fanno un utilizzo massiccio delle comunicazioni di rete, sia per quanto riguarda il traffico di dati, sia per quanto riguarda la mole di entità comunicant ed è per cui impensabile poter specificare una connessione peer to peer tra tutti i possibili attori coinvolti nella comunicazione. Per quanto riguarda la concezione dell'invio di un messaggio senza destinatario, Contact mette a disposizione i *signal*, ovvero dei messaggi che la sorgente immette nel sistema **senza sapere se esso verrà effettivamente letto e in quanti eventualmente lo leggeranno**. Per quanto riguarda invece la questione legata alla massiccia quantità di dati da scambiare, vi è un problema ancora più pratico, ossia la

potenza computazionale disponibile in una piattaforma embedded moderna, che è sì più elevata rispetto al passato, ma comunque poco performante se sfruttata da tanti thread concorrenti.

2.2.5 L'approccio Event-Driven

In un sistema con capacità computazionali ridotte, dove si vuole eseguire una mole di operazioni di varia natura (comunicazioni, reazioni immediate ad input esterni, gestione di informazioni sull'ambiente), è impensabile utilizzare un'approccio thread-based. Infatti, la concorrenza tra thread implica che lo scheduler abbia una politica di tipo preemptive ([LW13]), la quale implica un continuo context-switch tra thread, che diventa del tutto ingestibile a fronte di un elevato numero di thread. Una soluzione possibile a tale problema potrebbe essere l'introduzione di interrupt, ma questo è un concetto troppo di basso livello e strettamente legato all'harware. D'altra parte, il polling continuo impiegherebbe le risorse del sistema inutilmente. La soluzione ottimale potrebbe essere quella di adottare una architettura basata sugli Eventi. Tale architettura, adottata recentemente per risolvere problemi di scalabilità dei server (come il C10K Problem, [Keg]), prevede una schedulabilità di tipo non-preemptive ([LW13]), ossia in cui è l'entità in esecuzione che libera il controllo delle risorse quando lo ritiene necessario. Un esempio di utilizzo di tale approccio lo si trova nel framework Node.js ([CRH]). L'Architettura Event Driven promuove la produzione, l'individuziazine e la consumazione di *eventi*, definiti come un 'significate cambio di *stato*'. Da una prospettiva formale, ciò che viene prodotto, pubblicato, propagato e rilevato è un messaggio, tipicamente asincrono, il quale non contiene l'evento in sé, ma una notifica del cambio di stato che ha generato l'evento stesso. La particolarità da tenere in considerazione è che una notifica di evento non viene esplicitamente *inviata*, in quanto gli eventi non viaggiano, *accadono*. Un sistema a eventi è di solito composto da *emettitori* di eventi (agenti) e *consumatori* di eventi. Quando viene emesso un nuovo evento, i consumatori reagiscono in maniera quasi istantanea, ma non sempre immediata. Questo perché la

seconda caratteristica degli eventi è che essi vengono consumati in un unico Thread, il che rende l'applicazione vantaggiosa sotto molti aspetti: per prima cosa la potenza computazionale necessaria è veramente irrisoria, visto che all'applicazione è necessario un solo thread, in seconda battuta abbiamo la totale eliminazione della concorrenza tra le varie parti dell'applicazione, cosa che aumenta di gran lunga la reattività del sistema, e infine il codice su cui gli sviluppatori devono fare manutenzione è facilmente testabile e già logicamente organizzati in moduli, ossia *event handlers*, parti di applicazione che reagiscono allo scatenamento di un determinato evento preregistrato ([Sli03]).
Un approccio così potente e versatile può risolvere molti problemi di reattività e scalabilità nel mondo del software, ed è altrettanto utile, se non di più, per il mondo dei sistemi embedded che sfruttano massicciamente IoT. E' comunque importante fare una piccola precisazione sulla programmazione a eveni: poiché tutta la logica applicativa verrà eseguita, tramite *event handlers*, in un unico Thread, è di fondamentale importanza che tali *handlers* siano realizzati in modo da utilizzare il flusso di controllo per un tempo necessario all'esecuzione della logica di business, in modo che il sistema mantenga ottimali catteristiche di reattività.

2.2.6 Un nuovo spazio concettuale e il suo relativo DSL: ECSL

Siamo quindi davanti ad un nuovo spazio concettuale, che introduce nuovi concetti strutturali e di interazione. Tale metamodello specifica, ovviamente, una nuova forma di interazione: l'*event*. Esso è formato da un header, ossia il nome specifico dell'evento, e da un corpo contenente il messaggio vero e proprio. Una volta *lanciato* da una sorgente, esso verrà recepito da n ¿ 0 *EventHandlers*, di cui la sorgente ignora totalmente cardinalità e identità. La differenza con il *signal* di Contact sta nel fatto che quest'ultimo può essere inviato a n ¿= 0 destinatari, ciò vuol dire che, anche se nessuno è interessato a quel messaggio, quest'ultimo sarà sempre disponibile a posteriori dal suo

lancio fino alla cessazione delle attività del sistema, mentre l'*event* è consumabile solo se vi è almeno un *EventHandler* interessato prima che l'evento venga lanciato. I concetti di tipo strutturale introdotti sono l'*EventHandler*, ossia un elemento che può essere registrato ad uno o più *event* del sistema e viene eseguito nel Thread principale dell'applicazione (chiamato Main Loop) ogni qual volta viene rilevato uno di questi *event*, ed il *Task*, un elemento molto simile al *Subject* contact dal punto di vista comportamentale, eseguito sempre dal Main Loop, ma in modo regolare, ossia anche in assenza di eventi. E' importante che i *Task* vengano realizzati in modo da occupare il flusso di controllo del Main Loop per il più breve tempo possibile. E' stato altresì realizzato un nuovo DSL personalizzato che serve a definire modelli per sistemi software Event Driven: Event Contact Specifcation Language (ECSL). Tale DSL, realizzato sempre con XText ([Xte13]), è stato dotato di generatori di codice per l'implementazione in Java dei modelli, realizzando l'infrastruttura di Runtime che esegue il MainLoop. A differenza di Contact, questo nuovo DSL non ha una struttura mediana di implementazione che ne accentua le caratteristiche cross-platform per una più facile esportazione su più linguaggi.

Capitolo 3

Caso di studio - RobotSystem

Di seguito verrà mostrato un caso di studio concreto per lo sviluppo di un sistema software composto da elementi distribuiti e technology-variant. Dopo aver evidenziato i requisiti ed effettuata l'analisi del problema, verranno proposte due architetture logiche: una prima versione preliminare su cui verranno poi effettuati due ragionamenti che porteranno ad un suo refactoring. Le architetture verranno mostrate sia in forma UML, sia in forma di Contact, per evidenziarne le differenze.

3.1 Requisiti

Sia dato un **Robot**, il quale può essere azionato a distanza tramite uno o più **Telecomandi** in grado di impartire al Robot dei **Comandi** di **Movimento** a varie **Velocità**. Ogni Comando può modificare lo **Stato** del Robot, il quale vincolerà i Telecomandi a poter impartire solo alcuni tipi di Comandi in futuro. Lo Stato del Robot è perennemente monitorato da alcuni **Visualizzatori** anche essi remoti rispetto al Robot.

3.2 Analisi dei Requisiti

Per prima cosa, si evidenziano nel testo redatto dal committente tutti i sostantivi utili a definire le entità del sistema. Le entità del sistema possono essere dei semplici dati aventi solo proprietà oppure delle entità dotate di behavior (comportamento). E' bene parlare prima dei dati, in quanto essi saranno poi utilizzati dai restanti elementi nelle loro comunicazioni. I dati rilevati nel documento dei requisiti sono: **Comandi**, **Velocità** con cui impartire i comandi, **Stati** del Robot. Le entità con behavior del sistema sono: **Telecomandi**, **Robot**, **Visualizzatori** dello Stato del Robot.

3.2.1 Informazioni mancanti, ulteriori informazioni da parte del committente

Dopo questa prima parte dell'Analisi dei Requisiti potrebbero esserci alcune parti ancora non molto chiare sugli elementi del sistema. Il committente è l'unico che può chiarire questi punti oscuri, utili a realizzare un Glossario dei termini.

Le domande sono:

Quali sono i Comandi impartibili al Robot dal Telecomando? I Comandi del Robot sono:

- Move Forward: muove il Robot in avanti ad una data Velocità;
- Move Backward : muove il Robot indietro ad una data Velocità;
- Turn Left : gira il Robot a sinistra ad una data Velocità;
- Turn Right : gira il Robot a destra ad una data Velocità;
- Halt : arresta i movimenti del Robot.

Che valori di Velocità è possibile attribuire ai Comandi che la richiedono? Vanno bene velocità del tipo 'Alta', 'Media', 'Bassa'

Quali sono gli Stati del Robot? Per ora Stato di Movimento e Stato di Arresto.

3.2.2 Modellazione dei dati

Avendo ora tutte le informazioni a disposizione, è possibile iniziare a modellare i dati che, come già detto, possiedono solo la dimensione di Struttura. Si parte dalle interfacce, le quali rappresentano un contratto vincolante che specifica i desideri di chi ne ha bisogno, senza però impelagarsi troppo nell'implementazione. La convenzione che verrà utilizzata è quella di utilizzare l'inglese e di anteporre la vocale 'I' all'inizio del nome di ogni interfaccia. Poiché il sistema è distribuito e tali dati dovranno viaggiare tra le varie entità, bisogna fornire una loro rappresentazione in stringa di testo comprensibile a tutti. In questo stistema verrà utilizzata la notazione Prolog. Ad ogni dato verrà associato anche un Piano di Collaudo, al fine di far comprendere il loro funzionamento in modo formale e disambiguo.

- Velocità (ISpeed)

Le velocità utilizzabili sono 'ALTA' (HIGH), 'MEDIA' (MEDIUM) e 'BASSA' (LOW):

- Struttura: Vedi figura 3.1
- Rappresentazione:
  - *speed(**X**)*

  dove '***X***', secondo la notazione Prolog, è una variabile che, in questo caso specifico rappresenta uno dei tre valori 'HIGH', 'MEDIUM', 'LOW'.
- Collaudo: Nel seguente collaudo, realizzato con JUnit, è possibile vedere un esempio di utilizzo della Velocità:

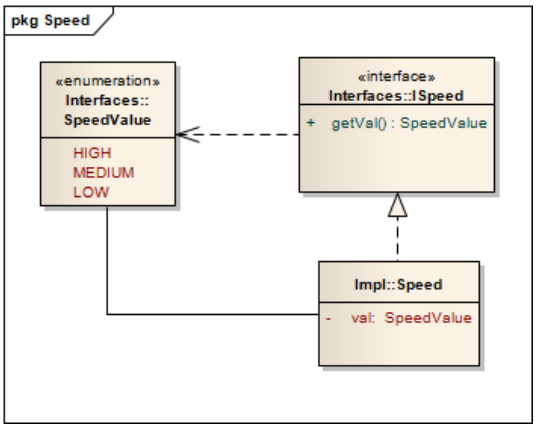

Figura 3.1: Modello della Velocità

```
1  public class ISpeedTest {
2
3      @Test
4      public void speedValueTest() {
5
6          ISpeed speed;
7          SpeedValue speedValue = SpeedValue.MEDIUM;
8          speed = new Speed(speedValue);
9          assertEquals(speed.getVal(), speedValue);
10     }
11
12     @Test
13     public void speedStringRepTest()
14     {
15         ISpeed speed;
16         SpeedValue speedValue = SpeedValue.MEDIUM;
17         speed = new Speed(speedValue);
18         String speedStringRep = "speed(medium)";
19         assertEquals(speed.getDefaultStringRep(), speedStringRep);
20     }
21 }
```

- Comandi (ICommand)

- Struttura: I Comandi del Robot possono essere raggruppati in comandi con velocità e senza velocità (figura 3.2)

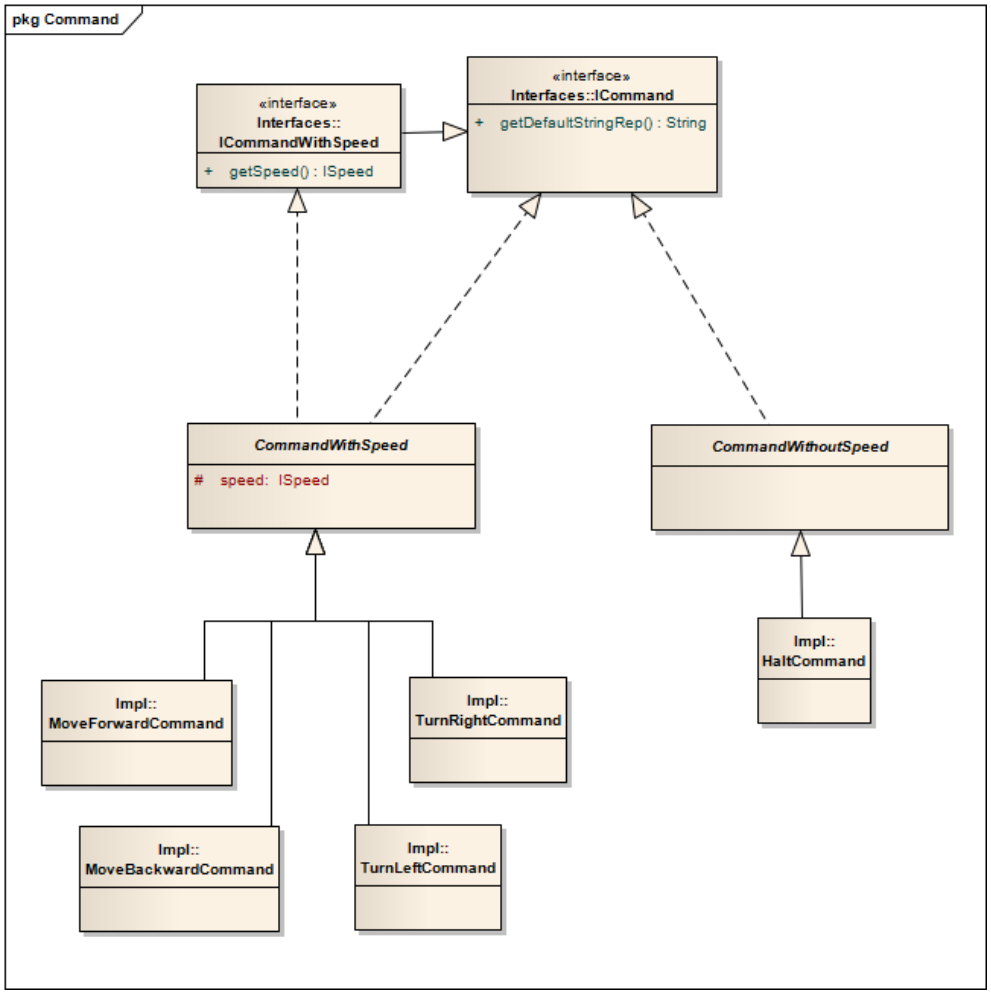

Figura 3.2: Modello dei Comandi del Robot

- Rappresentazione:
  - *command(halt)*
  - *command(moveForward(**S**))*
  - *command(moveBackward(**S**))*

- *command(turnLeft(**S**))*
- *command(turnRight(**S**))*

dove '***S***' è una variabile da sostituire con la rappresentazione di **ISpeed**;

- Collaudo:

```
1  public class ICommandTest {
2
3      @Test
4      public void haltCommandTest()
5      {
6          ICommand command;
7          command = new HaltCommand();
8          String haltCommandString = "command(halt)";
9          assertEquals(command.getDefaultStringRep(),
               haltCommandString);
10     }
11
12     @Test
13     public void commandWithSpeedTest()
14     {
15         ICommandWithSpeed command;
16         ISpeed speed;
17
18         speed = new Speed(SpeedValue.LOW);
19
20         command = new MoveForwardCommand(speed);
21
22         String moveForwardString = "command(moveForward(" +
               speed.getDefaultStringRep() + "))";
23
24         assertEquals(command.getDefaultStringRep(),
               moveForwardString);
25     }
26
27 }
```

- Stati del Robot (IRobotStatus)

- Struttura: Gli Stati del Robot sono, almeno per il momento, associati solo al fatto che il Robot si stia muovendo o meno (fig. 3.3).

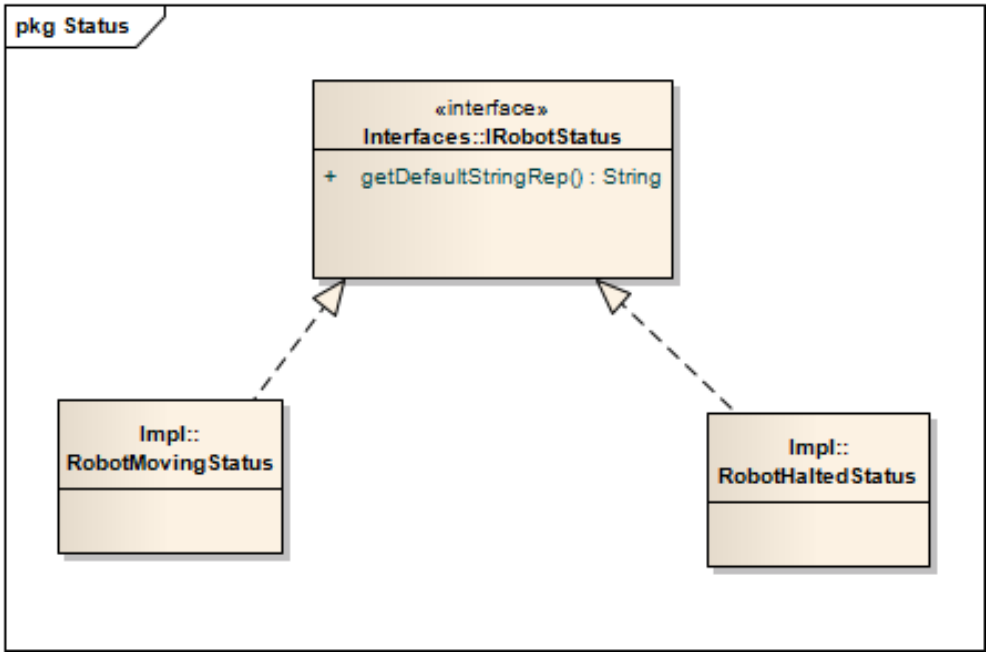

Figura 3.3: Modello degli Stati del Robot

- Rappresentazione:
  - *status(halted)*
  - *status(moving)*
- Collaudo:

```
1  public class IRobotStatusTest {
2
3      @Test
4      public void robotHaltedStatusTest()
5      {
6          IRobotStatus robotStatus;
7
8          robotStatus = new RobotHaltedStatus();
```

```
9
10          String haltedString = "status(halted)";
11
12          assertEquals(robotStatus.getDefaultStringRep(),
                haltedString);
13      }
14
15      @Test
16      public void robotMovingStatusTest()
17      {
18          IRobotStatus robotStatus;
19
20          robotStatus = new RobotMovingStatus();
21
22          String movingString = "status(moving)";
23
24          assertEquals(robotStatus.getDefaultStringRep(),
                movingString);
25      }
26  }
```

3.2.3 Individuazione delle entità

Lo scopo dell'Analisi dei Requisti è, al momento, soltanto quello di definire il Modello del Dominio, per cui, al momento, le tre entità cardine del sistema saranno descritte soltanto in modo superficiale, per vedere se la prospettiva dell'Analista è allineata con quella del cliente. Anche queste entità saranno definite tramite interfacce, seguendo la stessa convenzione dei dati.

- Telecomando (IRemoteControl)

Il RemoteControl sarà in grado di inviare Comandi al Robot e di tenersi aggiornato sul suo Stato (del Robot) (fig. 3.4).

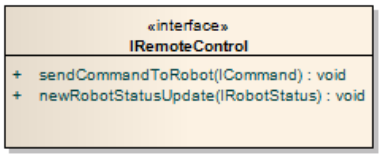

Figura 3.4: Struttura del Telecomando

- Robot (IRobot)

In base alle disposizioni del committente, il Robot dovrà eseguire Comandi e rendere disponibile il proprio Stato (3.5)

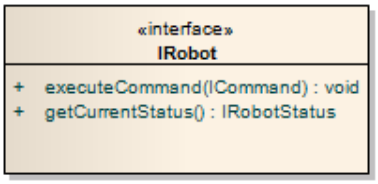

Figura 3.5: Struttura del Robot

- Visualizzatore dello Stato del Robot (IRobotStatusVisualizer)

Il Visualizzatore verrà aggiornato ogni qual volta il Robot cambierà di Stato (fig. 3.6).

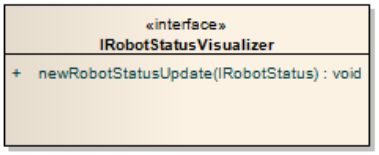

Figura 3.6: Struttura del Visualizzatore

3.3 Analisi del Problema

Con l'Analisi dei Requisiti vengono fissati in modo Permanente, Formale e Disambiguo tutti i concetti che fanno capo al Modello del Dominio, il quale

rispecchierà tutti i desideri del Committente. In pratica esprime in chiave ingegneristica *cosa vuole il Committente*. La seconda fase di Analisi, quella del Problema, è un'accurata ispezione del Modello del Dominio per mettere in luce *come realizzare ciò che il Committente vuole*.

Ricordando sempre che la qualità di un Modello è data dal suo livello di astrazione da qualsiasi tecnologia, è importante che il *come* non venga tradotto in concetti di una piattaforma specifica, altrimenti si vincolerebbe troppo il sistema. In questa fase è tuttavia possibile applicare concetti caratteristici del modo di costruzione dei Modelli della Software House che si occupa di realizzare il Sistema.

In questo Sistema, ciò che bisogna ora analizzare è il modo in cui le tre entità, **RemoteControl**, **Robot** e **RobotStatusVisualizer** possano interagire pur essendo, come riferiscono i Requisiti, remote. E' evidente per cui che gli elementi non possono, per forza di cose, essere modellati come dei comuni Oggetti dello spazio concettuale di OOP (Object Oriented Programming), referenziabili tra di loro. Una possibile soluzione a questo problema di comunicazione può essere, ad esempio, lo scambio di informazioni tramite messaggi. Sfortunatamente UML non ha le capacità sintattiche e semantiche di esprimere al meglio questo tipo di Interazione, per cui verrà utilizzato uno spazio concettuale personalizzato che riesce a rendere meglio l'idea di una comunicazione a scambio di messaggi, Contact ([Nat09]). Contact non si limita solo a definire i tipi di messaggi scambiati, ma anche chi scambia i messaggi e con che intenzione. In pratica riesce a definire, una volta ottenuto il Modello del Dominio, Struttura, Interazione e Comportamento.

3.3.1 Architettura Logica tramite UML

L'Architettura Logica di seguito riportata utilizzerà lo spazio concettuale Contact. Il sistema verrà analizzato secondo le tre dimensioni cardine utili a definire un Modello

Struttura

I tre elementi del sistema (**RemoteControl**, **RobotStatusVisualizer** e **Robot**) sono a loro volta dei sotto-sistemi indipendenti, il cui unico legame tra loro è rappresentato dal modello dei Dati che essi scambiano. Essi possono essere classificati come dei Subject ([Nat09]).

Interazione

- I RemoteControl inviano i Comandi al Robot, e i Requisiti non fanno riferimento al fatto che ci debba essere una risposta di ritorno, per cui essi possono essere modellati come dei *dispatch* ([Nat09]);
- Il Robot invia informazioni sul suo RobotStatus ai RemoteControl e ai RobotStatusVisualizer in modo del tutto imprevedibile a priori, e poiché esso non sa il numero esatto e l'ubicazione di chi riceverà il messaggio, la scelta più accurata per inviare quest'ultimo è sicuramente quella del *signal* ([Nat09]);

Una prima rappresentazione UML del sistema può essere quella rappresentata in figura 3.7.

Poiché UML non ha la capacità semantica di indicare Subject, Dispatch e Signal, essi verranno indicati con degli *stereotype* e, nel caso della Signal, anche col costrutto dei *lost message* e *found message*.

L'Architettura logica alla figura 3.7 mette in risalto un particolare non indifferente: per quanto riguarda il *signal*, è vero che la comunicazione avviene anche se il Robot effettivamente non sa chi riceverà lo Stato, ma è altrettanto vero che, essendo un Sistema distribuito, il Robot dovrà conoscere l'identificativo univoco di tutti i RemoteControl e i RobotStatusVisualizer (es. indirizzi IP della rete), al fine di instaurare una connessione preventiva all'invio del *signal*. Per cui vi è un refactoring da effettuare sull'architettura per ovviare a tale problema di identificazione: poichè sia i RemoteControl che i RobotStatusVisualizer dovranno comunque Conoscere il Robot, mentre non è vero il contrario, al momento della loro inizializzazione essi dovranno

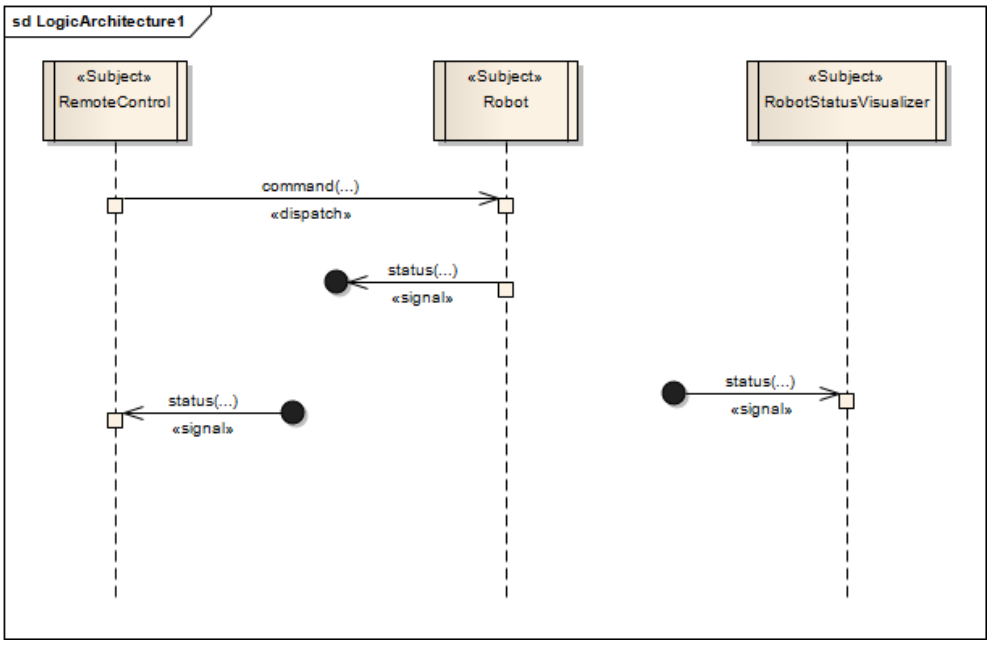

Figura 3.7: Struttura Logica del Sistema

connettersi al Robot per potergli inviare i Comandi o ricevere da lui lo Stato. Così facendo, l'unico parametro di connessione richiesto è l'identificativo univoco del Robot, il quale sfrutterà il canale di comunicazione realizzato dal richiedente per future comunicazioni. Poichè la connessione al Robot può avvenire in diversi istanti, chi si connette deve sapere lo Stato del Robot al momento della sua connessione, per questo motivo il messaggio di connessione sarà modellato come una *request* ([Nat09]). La figura 3.8 mostra la nuova Architettura Logica.

Comportamento

Il comportamento dei componenti del sistema può essere esposto tramite degli State Machine Diagram UML.

- RemoteControl: I RemoteControl per prima cosa si connettono al Robot e ricevono il RobotStatus corrente. Inviano i Comandi tramite *dispatch* e ricevono lo Stato del Robot ogni qualvolta esso cambia.

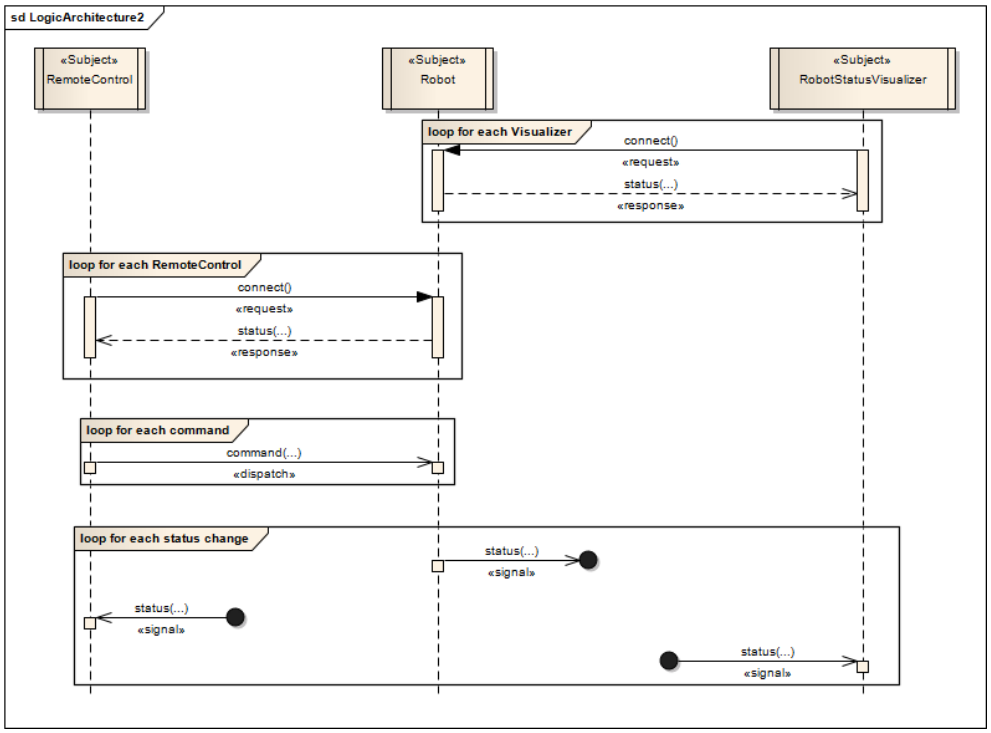

Figura 3.8: Raffinamento della Struttura Logica

- Robot: Il Robot risponde alle *request* di connessine dei RemoteControl e dei RobotStatusVisualizer, esegue i Comandi ricevuti tramite *dispatch* ed invia dei *signal* ogni qual volta il suo stato cambia.

- RobotStatusVisualizer: I RobotStatusVisualizer si connettono al Robot, e si attivano ogni qual volta il Robot invia il *signal* col cambiamento di Stato.

3.3.2 Architettura Logica tramite DSL Contact

Dai diagrammi UML precedentemente mostrati, pur avendo 'aggirato' in parte il problema della rappresentazione dei messaggi, esso non è stato risolto completamente perché il risultato finale non fa conseguire gli obiettivi prefissati da ogni modello, ossia la disambiguità e la formalità delle espressioni usate, al fine di ottenere una struttura facilmente meccanizzabile.

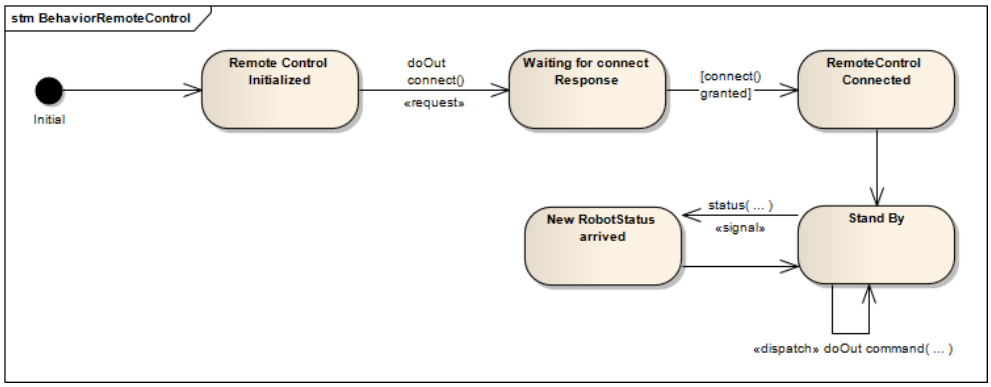

Figura 3.9: Comportamento del RemoteControl

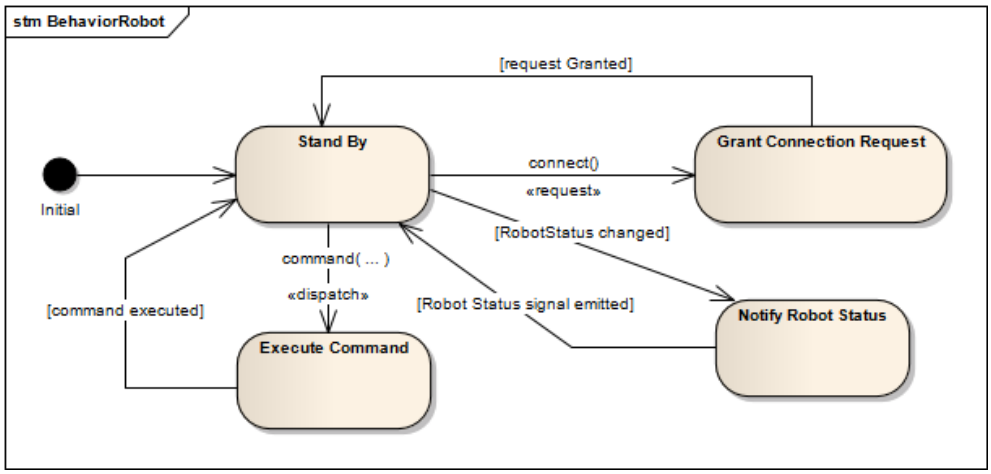

Figura 3.10: Comportamento del Robot

Ecco per cui una rappresentazione dello stesso sistema, realizzato però utilizzando il DSL Contact:

```
1 ContactSystem robotSystem;
2
3 Context remoteControlContext;
4 Context robotContex;
5 Context robotStatusVisualizerContext;
6
7 Subject remoteControl context remoteControlContext;
8 Subject robot context robotContex;
9 Subject robotStatusVisualizer context robotStatusVisualizerContext;
```

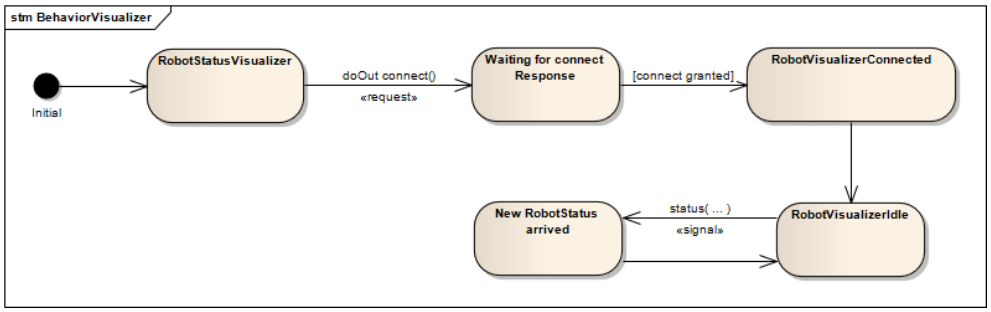

Figura 3.11: Comportamento del RobotStatusVisualizer

```
10
11  Request connect;
12  Dispatch command;
13  Signal status;
14
15  remoteControlDemandConnect : remoteControl demand connect to robot;
16  robotStatusVisualizerDemandConnect : robotStatusVisualizer demand connect
        to robot;
17  robotGrantConnect : robot grant connect;
18
19  forwardCommand : remoteControl forward command to robot;
20  serveCommand : robot serve command;
21
22  emitStatus : robot emit status;
23  remoteSenseStatus : remoteControl sense status;
24  visualizerSenseStatus : robotStatusVisualizer sense status;
25
26  BehaviorOf remoteControl
27  {
28      var org.mycompany.robotSystem.domain.interfaces.ICommand command
29      var String commandString
30      var org.mycompany.robotSystem.domain.interfaces.IRobotStatus status
31
32      action org.mycompany.robotSystem.domain.interfaces.IRobotStatus
            getRobotStatusFromMessage()
33      action void
            updateRobotStatus(org.mycompany.robotSystem.domain.interfaces.IRobotStatus
```

```
        status)
34
35      state remoteControlInit initial
36          goToState remoteControlInitalized
37      endstate
38
39      state remoteControlInitalized
40          doOutIn remoteControlDemandConnect("")
41          goToState waitingForConnectResponse
42      endstate
43
44      state waitingForConnectResponse
45          acquireAnswerFor connect
46          goToState remoteControlConnected
47      endstate
48
49      state remoteControlConnected
50          goToState remoteControlStandBy
51      endstate
52
53      state remoteControlStandBy
54          onMessage? status goToState remoteControlNewRobotStatusArrived
55  //      set command = new
        org.mycompany.robotSystem.domain.models.command.impl.HaltCommand()
56          set commandString = call command.getDefaultStringRep()
57          doOut forwardCommand(commandString)
58          goToState remoteControlStandBy
59      endstate
60
61      state remoteControlNewRobotStatusArrived
62          set status = exec getRobotStatusFromMessage()
63          exec updateRobotStatus(status)
64          goToState remoteControlStandBy
65      endstate
66  }
67
68  BehaviorOf robot
69  {
```

```
70    var org.mycompany.robotSystem.domain.interfaces.ICommand
          receivedCommand
71    var org.mycompany.robotSystem.domain.interfaces.IRobotStatus
          currentStatus
72    var String currentStatusString
73
74    action org.mycompany.robotSystem.domain.interfaces.ICommand
          getCommandFromMessage()
75
76    action org.mycompany.robotSystem.domain.interfaces.IRobotStatus
          getCurrentStatus()
77    action void
          executeCommand(org.mycompany.robotSystem.domain.interfaces.ICommand
          command)
78
79    state robotInit initial
80        goToState robotStandBy
81    endstate
82
83    state robotStandBy
84        onMessage? connect goToState grantConnectionRequest
85        onMessage? command goToState executeCommand
86        goToState notifyRobotStatus
87    endstate
88
89    state grantConnectionRequest
90        replyToRequest connect("");
91        goToState robotStandBy
92    endstate
93
94    state executeCommand
95        set receivedCommand = exec getCommandFromMessage()
96        exec executeCommand(receivedCommand)
97        goToState robotStandBy
98    endstate
99
100   state notifyRobotStatus
101       set currentStatus = exec getCurrentStatus()
```

```
102         set currentStatusString = call currentStatus.getDefaultStringRep()
103         doOut emitStatus(currentStatusString)
104         goToState robotStandBy
105     endstate
106 }
107
108 BehaviorOf robotStatusVisualizer
109 {
110     var org.mycompany.robotSystem.domain.interfaces.IRobotStatus
            robotStatus
111
112     action org.mycompany.robotSystem.domain.interfaces.IRobotStatus
            getStatusFromMessage()
113
114     action void
            visualizerUpdateRobotStatus(org.mycompany.robotSystem.domain.interfaces.IRobotStatus
            status)
115
116     state robotStatusVisualizerInit initial
117         goToState robotStatusVisualizerInitialized
118     endstate
119
120     state robotStatusVisualizerInitialized
121         doOutIn robotStatusVisualizerDemandConnect("")
122         goToState visualizerWaitingForConnectResponse
123     endstate
124
125     state visualizerWaitingForConnectResponse
126         acquireAnswerFor connect
127         goToState robotVisualizerConnected
128     endstate
129
130     state robotVisualizerConnected
131         goToState robotVisualizerIdle
132     endstate
133
134     state robotVisualizerIdle
135         onMessage status transitTo newRobotStatusArrived
```

```
136     endstate
137
138     state newRobotStatusArrived
139         set robotStatus = exec getStatusFromMessage()
140         exec visualizerUpdateRobotStatus(robotStatus)
141         goToState robotVisualizerIdle
142     endstate
143 }
```

Come è possibile notare, utilizzando il DSL Contact si è potuto esprimere in un unico file e con poche righe tutte e tre le dimensioni di Struttura, Interazione e Comportamento.

E' tuttavia doveroso far notare che con questa rappresentazione si ha comunque il bisogno di avere a parte il modello dei dati scambiati (**ICommand**, **ISpeed** e **IRobotStatus**), ragion per cui diventa ancora più evidente di come il Modello Del Dominio debba essere realizzato il prima possibile.

3.3.3 Dal Modello Astratto all'applicazione Concreta

Una volta definito il Modello si può passare alla realizzazione del software. La prima cosa da fare è sicuramente realizzare il Modello del Dominio che, ricordiamolo, contiene solo le interfacce delle entità e i dati che è possibile interscambiare, insomma tutto ciò che è contratto dell'applicazione stipulato da e col Committente. Esso quindi è inserito in un progetto, ad esempio Java, che conterrà due package: le interfacce di tutti gli elementi e i modelli, astratti e concreti, dei dati scambiati. Per questo progetto si è utilizzato l'IDE Eclipse.

Un altro importante progetto da dover implementare è sicuramente quello dei Piani di Collaudo, che danno al progettista implementatore un'idea preliminare di come le interfacce debbano funzionare.

I restanti progetti da realizzare sono tanti quanti sono i sottosistemi che fanno parte dell'applicazione. Essi fanno riferimento al Modello del Dominio e contengono, in due package differenti, sia la parte di Modelli astratti che le vere implementazioni.

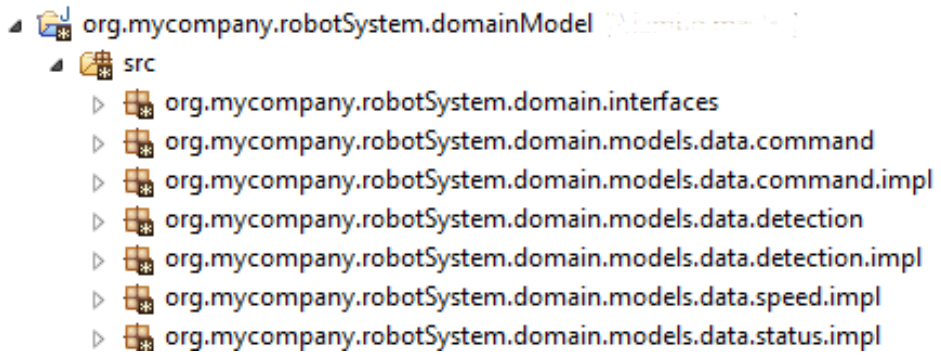

Figura 3.12: Progetto Eclipse del Modello Del Dominio

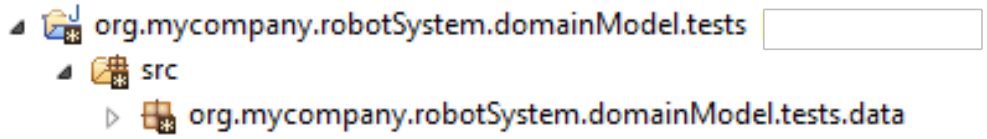

Figura 3.13: Progetto Eclipse dei Piani di Collaudo

3.3.4 Benefici del Modello associato ad un DSL come Contact: la generazione del codice implementativo

Grazie alle potenzialità del DSL Contact, realizzato con XText ([Xte13]), è possibile generare automaticamente un prototipo Java del Sistema (fig 3.15).

Da quì esce fuori la potenza espressiva dei Modelli, che non diventano più solo uno 'spunto' per la progettazione fisica del sistema, ma sono bensì lo strumento principale con cui produrre il software. E' importante per cui riuscire ad esprimerli al meglio e nel modo più dettagliato possibile, e di avere soprattuto strumenti che ne facilitano il lavoro di stesura, che permetterà poi di ottenere sistemi funzionanti in modo univoco su ogni piattaforma implementativa regolare, senza cambiare alcun ché della logica applicativa voluta dal Committente, basterà avvalersi dei generatori di codice.

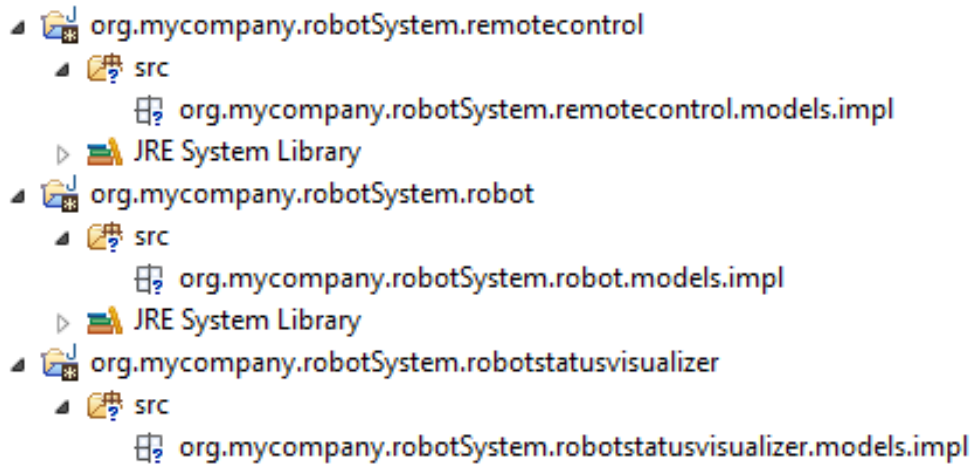

Figura 3.14: Progetti Eclipse di RemoteControl, Robot e RobotStatusVisualizer

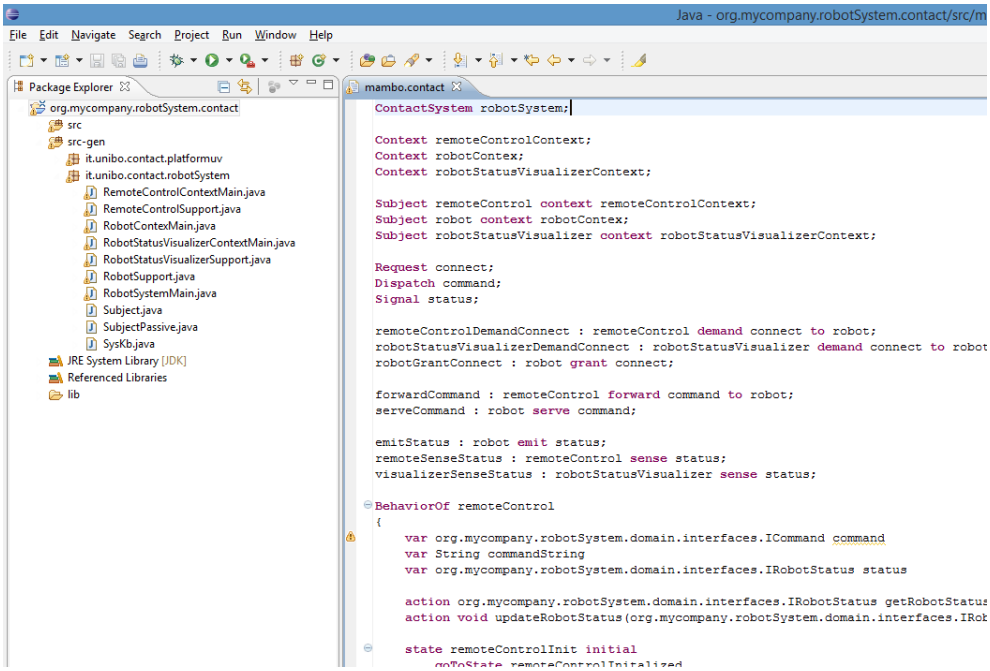

Figura 3.15: Codice Java del sistema generato tramite il modello Contact

Capitolo 4

Introspezione di RobotSystem - Modellazione del Sottosistema Robot

Una volta ottenuto il prototipo del sistema tramite Contact, la successiva mossa è quella di realizzare i tre sottosistemi **RemoteControl**, **Robot** e **RobotStatusVisualizer**. Focalizzeremo la nostra attenzione sul Robot e ne faremo l'introspezione, chiedendo di nuovo al Committente.

4.1 Requisiti del Robot

Dalla documentazione precedente sappiamo che il Robot è in grado di eseguire dei Comandi di movimento. Questi Comandi di movimento vengono attuati tramite **Ruote** mosse da **Motori**. Esso si muove all'interno di un'area circondata da una **Linea** di limitazione, in cui sono presenti anche degli **Ostacoli**. Per salvaguardarsi mentre si muove, il Robot non deve superare la linea e non deve scontrarsi contro gli ostacoli.

4.2 Analisi dei Requisiti

Analizzando il testo si evince che il Robot utilizza i Comandi passati dai RemoteControl per una **Sequenza di Comandi da attuare sulle Ruote** da parte di un **Set di Motori**.

La **Sequenza di Comandi da attuare sulle Ruote** è una pletora di **Comandi individuali su ogni Ruota**, che permettono al **Set di Motori** di spingere una **Specifica** ruota in avanti, indietro o di fermarla. Esistono vari tipi di **Robot dotati di Ruote**, il più semplice è l'**Uniciclo**, dotato di due ruote. Per salvaguardarsi, il Robot avrà un meccanismo di rilevamento che gli consentirà di capire quando ha **Raggiunto** o **Lasciato** la **Linea** di confine dell'area, oppure se si è **Avvicinato** o **Allontanato** da un **Ostacolo**.

4.2.1 Modello dei Dati

- Identificativo della Ruota (IWheelID)

La prima informazione che il **Robot dotato di ruote** e il **Set di motori** si scambiano è l'**Identificativo delle Ruote** che, nel caso dell'**Uniciclo**, ad esempio, sono Ruota DESTRA (RIGHT) e Ruota SINISTRA (LEFT).

- Struttura: Vedi figura 4.1
- Rappresentazione: Di seguito la rappresentazione Prolog delle **UnicycleWheelID**:
  - *unicycleWheelID(**X**)*

  dove '***X***', secondo la notazione Prolog, è una variabile che, in questo caso specifico rappresenta uno dei due valori 'left' o 'right'.
- Collaudo:

```
1 public class UnicycleWheelIDTest {
2
3     @Test
4     public void leftUnicycleWheelIDTest()
```

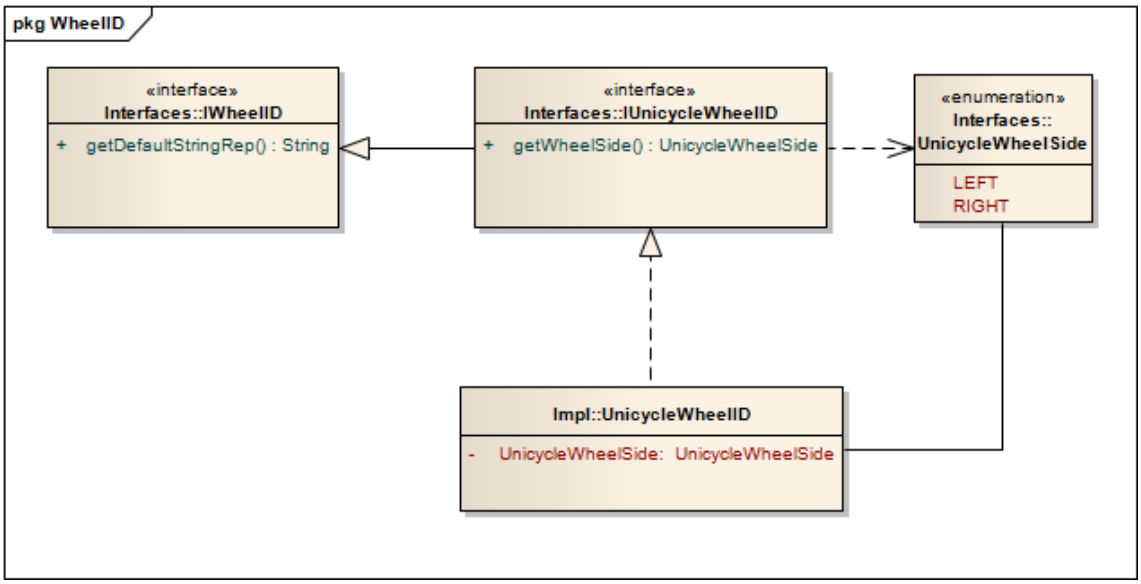

Figura 4.1: Modello dell'Identificativo delle Ruote

```
5       {
6           IWheelID wheelIDLeft;
7           wheelIDLeft = new UnicycleWheelID(UnicycleWheelSide.LEFT);
8
9           String leftString = "unicycleWheelID(left)";
10
11          assertEquals(leftString, wheelIDLeft.getDefaultStringRep());
12      }
13  }
```

- Comando per ogni singola Ruota (IWheelCommand)

I comandi per ruota agiscono per ogni singola ruota, identificandone anche il movimento. Tali comandi possono essere con o senza velocità.

- Struttura: Vedi figura 4.2
- Rappresentazione:
  - *wheelCommand(**ID**, **M**)*

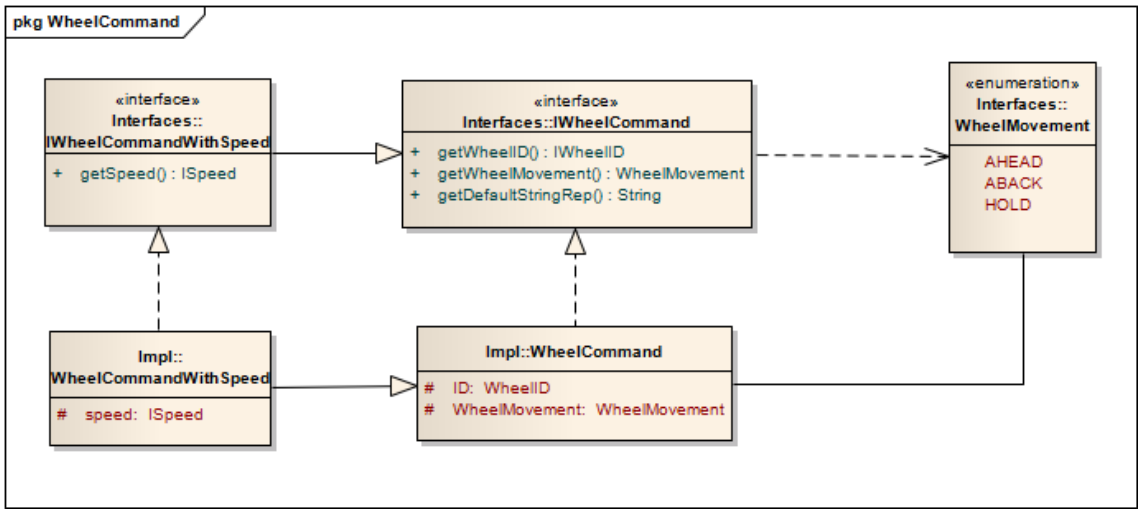

Figura 4.2: Modello del comando delle Ruote

– *wheelCommand(**ID**, **M**, **S**)*

dove '***ID***', è una variabile che rappresenta la rappresentazione in stringa Prolog di **IWheelID**, '***M***' è un valore del movimento come 'AHEAD', 'ABACK' o 'HOLD', e '***S***' è una rappresentazione in stringa Prolog di **ISpeed**.

- Sequenza di comandi delle ruote (IWheelCommandsSequence)

Questi sono i comandi veri e propri da passare al Set di Motori. Ogni sequenza ha una cardinalità che identifica da quanti Comandi di Ruota è composta e da la possibilità di recuperare i Comandi di Ruota individuali.

- Struttura: Vedi figura 4.3
- Rappresentazione:

  – *wheelCommandsSequence(**X**, **Y**...)*

  dove '***X**, **Y**...*', sono delle variabili che esprimono la rappresentazione in stringa Prolog di **IWheelCommand**.

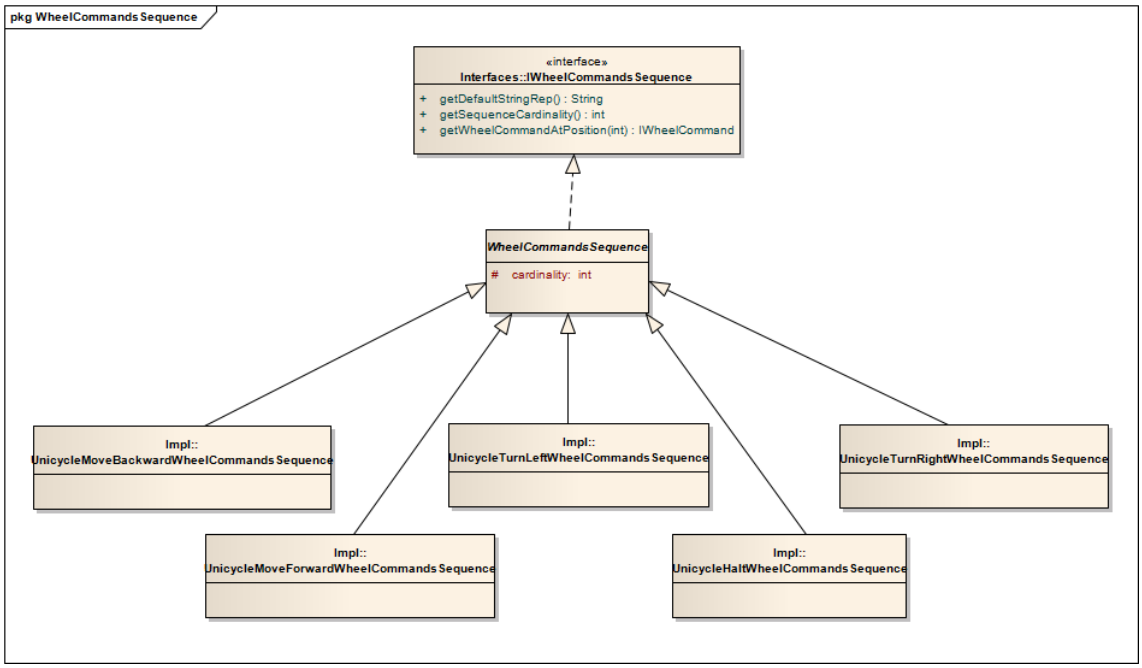

Figura 4.3: Modello della Sequenza di comandi delle Ruote

- Collaudo: Di seguito il collaudo delle Sequenze di Comandi sulle Ruote per gli Unicycle.

```
1  public class IWheelCommandsSequenceTest {
2
3      @Test
4      public void testUnicycleWheelHaltCommand()
5      {
6          IWheelCommandsSequence sequence;
7
8          sequence = new UnicycleHaltWheelCommandsSequence();
9
10         assertTrue(sequence.getSequenceCardinality() == 2);
11
12         IWheelCommand leftWheelCommand =
               sequence.getWheelCommandAtPosition(0);
13         IWheelCommand rightWheelCommand =
               sequence.getWheelCommandAtPosition(1);
14
```

```
15          assertEquals(leftWheelCommand.getWheelMovement(),
                WheelMovement.HOLD);
16          assertEquals(rightWheelCommand.getWheelMovement(),
                WheelMovement.HOLD);
17
18          IUnicycleWheelID leftID = (IUnicycleWheelID)
                leftWheelCommand.getWheelID();
19          IUnicycleWheelID rightID = (IUnicycleWheelID)
                rightWheelCommand.getWheelID();
20
21          assertEquals(leftID.getWheelSide(), UnicycleWheelSide.LEFT);
22          assertEquals(rightID.getWheelSide(),
                UnicycleWheelSide.RIGHT);
23      }
24
25  }
```

- Rilevamento di perturbazioni nell'ambiente del Robot (IDetection)

- Struttura: Vedi figura 4.4
- Rappresentazione:
  - *detection(lineLeft)*
  - *detection(lineReached)*
  - *detection(obstacleReached(**P**))*
  - *detection(obstacleLeft(**P**))*

  dove '***P***' è che esprime la rappresentazione in stringa Prolog di **ObstaclePosition**.

4.2.2 Modello delle nuove Entità

L'Analisi dei Requisiti dello Zooming ha portato alla luce due nuove entità nel sistema: un tipo particolare di Robot munito di Ruote, e un set di Motori.

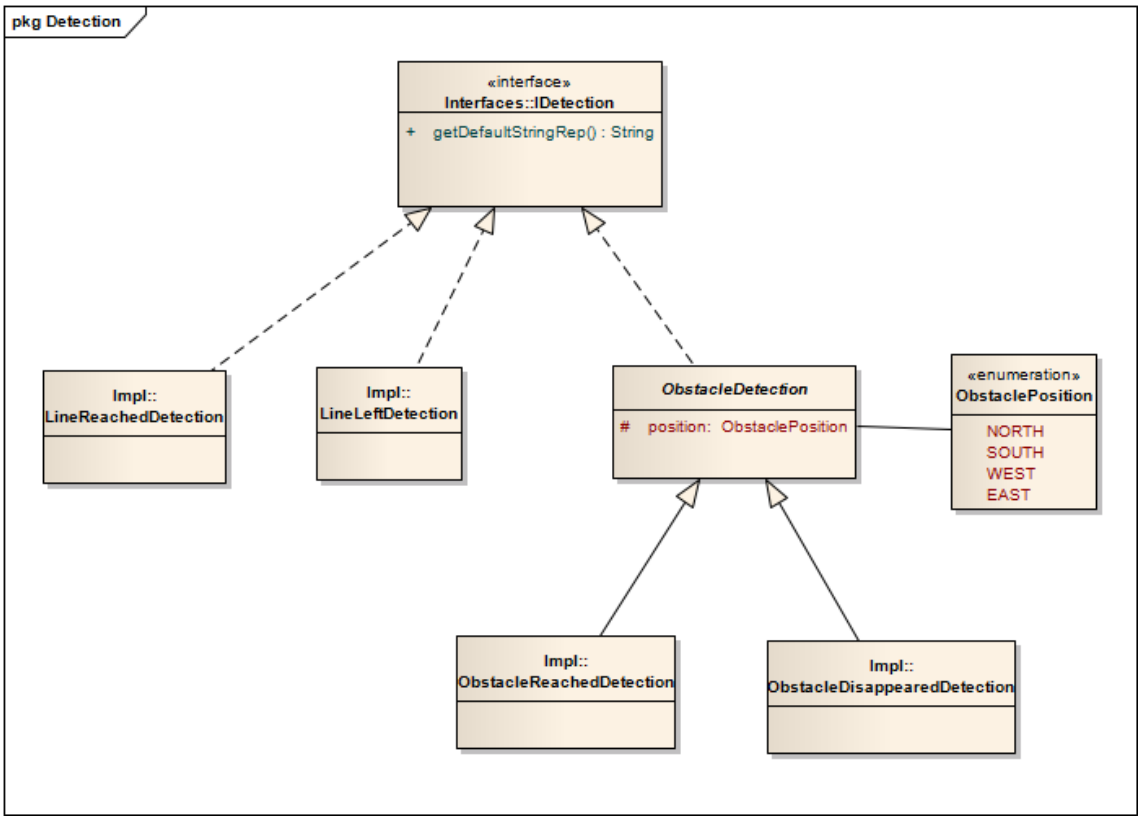

Figura 4.4: Modello delle IDetection

- Robot munito di Ruote (IWheelsEquippedRobot)

- Struttura: Questo Robot dotato di ruote trasforma i Comandi semplici in **IWheelCommandsSequence** (fig. 4.5).

  E' stato inoltre realizzato anche un Modello di un tipo di Robot dotato di Ruote, l'**Unicycle** (fig. 4.6):

- Collaudo dell'**Unicycle**:

```
1 public class UnicycleRobotTest {
2
3     @Test
4     public void moveForwardConversionTest()
5     {
6         IWheelsEquippedRobot robot;
```

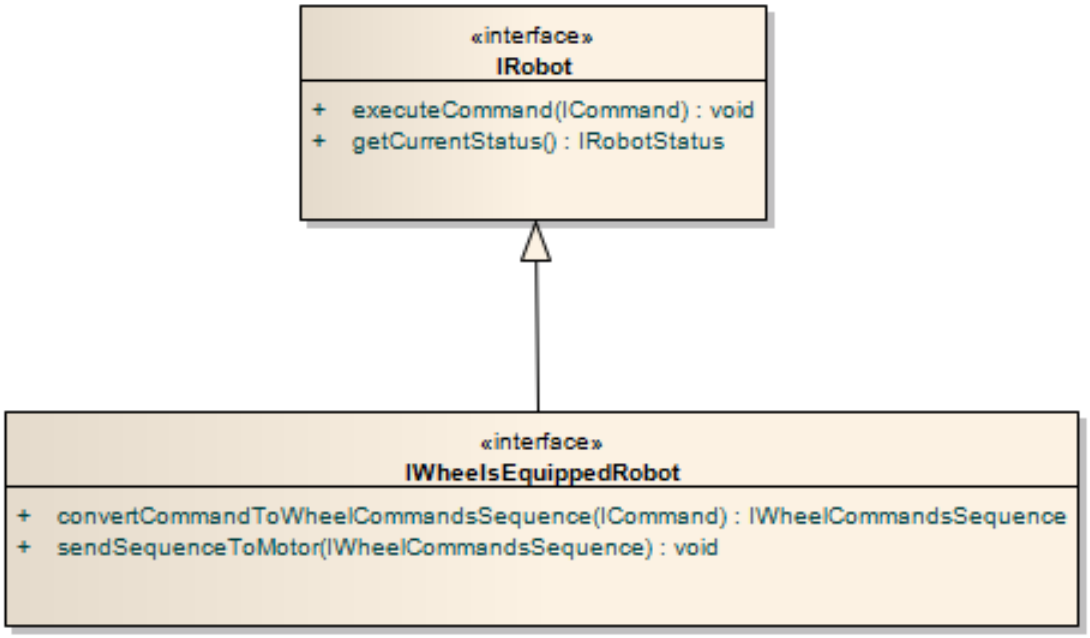

Figura 4.5: Modello del Robot dotato di Ruote

```
7         robot = new MockUnicycleRobot();
8
9         ISpeed speed = new Speed(SpeedValue.HIGH);
10
11        ICommand command = new MoveForwardCommand(speed);
12
13        IWheelCommandsSequence wheelCommandsSequence =
              robot.convertCommandToWheelCommandsSequence(command);
14
15        assertEquals(UnicycleMoveForwardWheelCommandsSequence.class,
              wheelCommandsSequence.getClass());
16    }
17
18 }
```

- Set di Motori (IMotorSet)

L'IMotorSet servirà per attuare i Comandi Ruota (fig. 4.7)

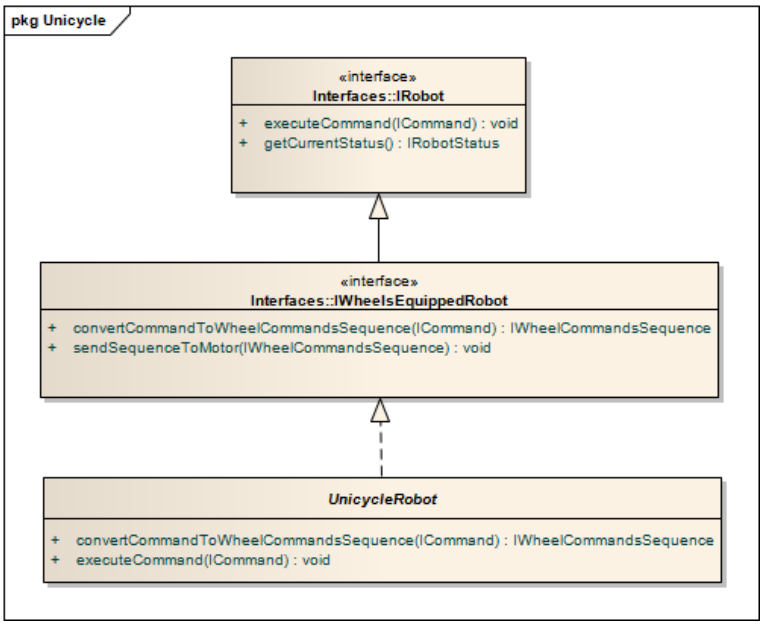

Figura 4.6: Modello dell'**Unicycle**

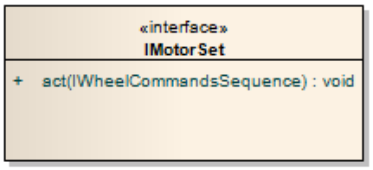

Figura 4.7: Modello dell'**IMotorSet**

4.3 Analisi del Problema

4.3.1 Come rilevare gli ostacoli e la linea, l'IDetector

I Requisiti parlano di IDetection che vengono effettuate mentre il Robot si muove. Viene perciò introdotto un nuovo elemento che provvederà a perturbare il sistema quando verrà rilevata una IDetection, l'IDetector (fig 4.8).

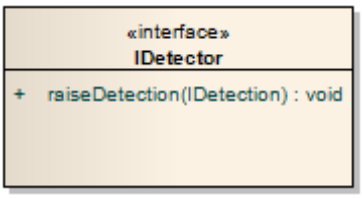

Figura 4.8: Modello dell'**IDetector**

4.3.2 Troppi doveri per il Robot, introduzione dell'IRobotController

Con l'avvento dell'**l'IDetector**, il Robot si troverebbe a gestire le detection, che non competono a lui, in quanto mero esecutore di Comandi. Ciò Comporta l'introduzione di un'entità di mediazione che coordina i Comandi e le **IDetection**: l'**IRobotController** (fig.4.9).

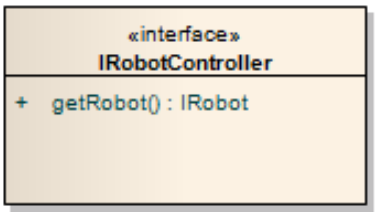

Figura 4.9: Modello dell'**IRobotController**

4.3.3 Architettura Logica

Anche in questa Architettura Logica verrà utilizzato lo spazio concettuale proprio di Contact ([Nat09]).

Struttura

- L'IWheelsEquippedRobot, un tipo di IRobot, diventa ora un Oggetto passivo, di proprietà dell'IRobotController
- L'IMotor è un Subject
- L'IRobotController è un Subject con riferimento all'IWheelsEquippedRobot.

- L'IDetector è un Subject atomico, ce ne sono tanti quanti sono i tipi di IDetection da lanciare (in questo caso 4).

Interazione

- L'IRobotController avrà un riferimento all'IWheelsEquippedRobot, così potrà chiamarlo direttamente quando ci sarà bisogno di eseguire un comando oppure di reagire ad una IDetection, l'IRobotController inoltre diventa il ricettore dei comandi provenienti dagli IRemoteControl.

- Per eseguire le IWheelCommandsSequence, l'IWheelsEquippedRobot invierà una *request* all'IMotorSet, che risponderà in modo positivo se la sequenza di comandi andrà a buon fine oppure no.

- L'IDetector lancerà un *signal* quando ci sarà una IDetection da notificare, che verrà raccolta dall'IRobotController.

Comportamento

- L'IRobotController riceverà i Comandi dall'IRemoteControl e reagirà ai *signal* degli IDetector e creerà il comando giusto da dare all'IWheelsEquippedRobot leggendo il suo stato attuale agendo sull'IWheelsEquippedRobot;

- L'IRobotController invierà le *request* dei Comandi Ruota all'IMotor, dopo aver chiesto all'IWheelsEquippedRobot di convertire i Comandi di movimento

- L'IMotor rimane in attesa delle *request* dell'IRobotController

- L'IDetector lancia un *signal* quando viene rilevata una IDetection

Per semplicità, la rappresentazione è stata realizzata direttamente in Contact, senza passare per UML:

```
1  ContactSystem robotSystemRobot;
2
3  Subject robotController;
4  Subject motorSet;
5  Subject lineReachedDetector;
6  Subject lineDisappearedDetector;
7  Subject obstacleReachedDetector;
8  Subject obstacleDisappearedDetector;
9
10 Dispatch command;
11 Request wheelCommand;
12 Signal detection;
13
14 serveCommand : robotController serve command;
15
16 demandWheelCommand : robotController demand wheelCommand to motorSet;
17 grantWheelCommand : motorSet grant wheelCommand;
18
19 senseDetection : robotController sense detection;
20
21 lineReachedDetection : lineReachedDetector emit detection;
22 lineDisappearedDetection : lineDisappearedDetector emit detection;
23 obstacleReachedDetection : obstacleReachedDetector emit detection;
24 obstacleDisappearedDetection : obstacleDisappearedDetector emit detection;
25
26 BehaviorOf robotController
27 {
28     var org.mycompany.robotSystem.domain.interfaces.IWheelsEquippedRobot
          robot
29     var
          org.mycompany.robotSystem.domain.interfaces.IWheelCommandsSequence
          sequence
30     var org.mycompany.robotSystem.domain.interfaces.ICommand command
31     var org.mycompany.robotSystem.domain.interfaces.IDetection detection
32     var org.mycompany.robotSystem.domain.interfaces.IRobotStatus status
33     var String sequenceString
34
```

```
35    action
          org.mycompany.robotSystem.domain.interfaces.IWheelsEquippedRobot
          initRobot()
36    action org.mycompany.robotSystem.domain.interfaces.ICommand
          getCommandFromReceivedMessage()
37    action org.mycompany.robotSystem.domain.interfaces.IDetection
          getDetectionFromReceivedMessage()
38    action org.mycompany.robotSystem.domain.interfaces.ICommand
          elaborateCommandFromDetectionAndRobotStatus(org.mycompany.robotSystem.domain.interfaces
          detection,
          org.mycompany.robotSystem.domain.interfaces.IRobotStatus status)
39
40    state robotControllerInit
41        set robot = exec initRobot()
42        goToState robotControllerLoop
43    endstate
44
45    state robotControllerLoop
46        onMessage detection transitTo newDetection
47        onMessage command transitTo newCommand
48    endstate
49
50    state newDetection
51        set detection = exec getDetectionFromReceivedMessage()
52        set status = call robot.getCurrentStatus()
53        set command = exec
              elaborateCommandFromDetectionAndRobotStatus(detection, status)
54        goToState sendWheelCommand
55    endstate
56
57    state newCommand
58        set command = exec getCommandFromReceivedMessage()
59        goToState sendWheelCommand
60    endstate
61
62    state sendWheelCommand
63        set sequence = call
              robot.convertCommandToWheelCommandsSequence(command)
```

```
64          set sequenceString = call sequence.getDefaultStringRep()
65          doOutIn demandWheelCommand(sequenceString)
66          acquireAnswerFor wheelCommand
67          goToState robotControllerLoop
68      endstate
69  }
70
71  BehaviorOf motorSet
72  {
73      var
            org.mycompany.robotSystem.domain.interfaces.IWheelCommandsSequence
            commandsSequence
74
75      action void
            act(org.mycompany.robotSystem.domain.interfaces.IWheelCommandsSequence
            sequence)
76      action
            org.mycompany.robotSystem.domain.interfaces.IWheelCommandsSequence
            getSequenceFromReceivedMessage()
77
78      state motorSetInit initial
79          goToState motorSetIdle
80      endstate
81
82      state motorSetIdle
83          onMessage wheelCommand transitTo newWheelCommandArrived
84      endstate
85
86      state newWheelCommandArrived
87          set commandsSequence = exec getSequenceFromReceivedMessage()
88          exec act(commandsSequence)
89          replyToRequest wheelCommand("")
90          goToState motorSetIdle
91      endstate
92  }
93
94  BehaviorOf lineReachedDetector
95  {
```

```
96      var org.mycompany.robotSystem.domain.interfaces.IDetection
            lineReachedDetection = new
            org.mycompany.robotSystem.domain.models.data.detection.impl.LineReachedDetection()
97      var String lineReachedDetectionString
98
99      state lineReachedDetector initial
100         set lineReachedDetectionString = call
                lineReachedDetection.getDefaultStringRep()
101         doOut lineReachedDetection(lineReachedDetectionString)
102     endstate
103 }
104 \\Same behavior for other detectors...
```

Capitolo 5

Implementazione del Sottosistema Robot su RaspberryPI

Nei capitoli precedenti si è parlato di come è possibile utilizzare i Sistemi Embedded di nuova concezione come Raspberry PI per realizzare dei manufatti Hardware e Software da impiegare nella vita quotidiana, senza il bisogno di essere degli specialisti di controllori elettronici. L'obiettivo di questo capitolo è infatti quello di realizzare la parte logica e fisica del Robot del Sistema Mambo proprio su Raspberry PI, sfruttando GPIO per interfacciarsi con i componenti elettronici, e le capacità connessione alla rete per la comunicazione.

5.1 Linguaggio di Programmazione Scelto

L'ecosistema Raspberry PI supporta numerose distribuzioni di Sistemi Operativi Linux Based, più o meno articolate, ma che comunque danno la possibilità di installare versioni regolari dei motori di esecuzione di alcuni dei linguaggi di programmazione più famosi e usati, come Python, Bash, Java, PHP, C/C++. Per questo progetto si è scelto di utilizzare Java, anche perché

l'utilizzo del DSL Contact, e quindi indirettamente anche dei suoi generatori di codice, ha direttamente prodotto in Java tutta la struttura di comunicazione dei vari Subject, senza alcuno sforzo. Ecco che si evincono ancora una volta i benefici di avere a disposizione la potenza espressiva necessaria per la definizione di Modelli, ma soprattutto l'utilità e l'importanza dei Modelli.

5.2 Parte Hardware

Alcuni dei componenti del sistema hanno bisogno di supporti di comunicazione diretta con dei componenti hardware:

- Il **Motor Set** ha bisogno di impartire ai motori fisici i movimenti da far compiere aller ruote.
- Per la notifica delle **IDetection**, gli **IDetector** hanno bisogno di riscontrare la perturbazione dell'ambiente esterno tramite dei detector fisici, come dei sensori.

Fortunatamente è possibile sfruttare le caratteristiche di piattaforma Embedded di tipo General Purpose di Raspberry PI, la quale non ha accesso dedicato a specifici componenti hardware esterni, ma un connettore in grado di collegare numerosi dispositivi sia di input che di output, il GPIO ([Sch12]).

Il ponte di comunicazione tra GPIO e l'applicazione verrà realizzato con l'ausilio di una libreria Java realizzata appositamente, Pi4J ([Tea13]).

Basterà per cui collegare i sensori e i motori al GPIO e prendere nota dei Pin occupati, in modo da dare la possibilita agli **IDetector** e al **IMotorSet** di instaurare la comunicazione, attraverso PI4J.

5.3 Analisi del Rischio dell'approccio classico (Thread Based)

Grazie all'utilizzo del DSL Contact e dei suoi generatori di codice sorgente (Java), si è subito provvisti di un prototipo addirittura utilizzabile in pro-

duzione, dal punto di vista dell'Interazione. Ma esso è comunque realizzato sfruttando il multi-thread. Tale approccio è sì possibile anche su Raspberry Pi, ma in questo caso specifico è poco performante per via della scarsa memoria a disposizione. Il Robot inoltre, non deve soltanto recepire dei comandi, ma deve anche reagire *nel più breve tempo possibile* alle **IDetection** segnalate dai sensori. E' doveroso quindi utilizzare un altro approccio, quello a eventi, per migliorare le prestazioni del Robot.

5.4 Architettura logica riscritta con approccio Event Driven in ECSL

Per la realizzazione di questo nuovo modello verrà utilizzato il nuovo spazio concettuale degli eventi e il suo DSL ECSL:

5.4.1 Struttura

L'**IRobotController** e i 4 **IDetector** diventano ora dei Task liberi quindi di eseguire, solo se necessario, nello stesso Main Loop.

5.4.2 Interazione

Il messaggio *command* continua ad essere un *dispatch* ricevuto dall'**IRobotController**, mentre le 4 **IDetection**, *lineReached*, *lineDisappeared*, *obstacleDetected* e *obstacleDisappeared* sono ora degli *event* lanciati dagli **IDetector** e raccolti dall'**IRobotController**.

5.4.3 Comportamento

Anche in questo DSL il comportamento dei *Task* viene definito tramite macchine a stati finiti. In questo caso gli stati dell'ASF hanno anche funzione di spezzettare l'esecuzione del *Task* restituendo così il controllo al Main Loop ad ogni transizione di stato, in modo da dare a quest'ultimo la possibilità

di lanciare degli eventi arrivati nel frattempo: gli **IDetector** lanceranno un evento ogni qual volta ce ne sarà il bisogno. **IRobotController** verrà eseguito ogni qual volta arriverà un nuovo ICommand o una nuova IDetection in modo assolutamente reattivo.

Di seguito la specifica ECSL:

```
1  EventSystem robot
2
3  Event lineDetected;
4  Event lineLeft;
5  Event obstacleDetected;
6  Event obstacleLeft;
7
8  Task robotController listenTo lineDetected lineLeft obstacleDetected
       obstacleLeft;
9  Task lineReachedDetector;
10 Task lineDisappearedDetector;
11 Task obstacleReachedDetector;
12 Task obstacleDisappearedDetector;
13
14 BehaviorOf robotController
15 {
16     action void onCommandDispatchMessageGoToStateNewCommandState()
17
18     state robotControllerInit initial
19
20     endstate
21
22     state robotControllerLoop
23         //On command dispatch message goToState newCommandState
24         exec onCommandDispatchMessageGoToStateNewCommandState()
25
26         onEvent lineDetected goToState lineDetectedState
27         onEvent lineLeft goToState lineLeftState
28         onEvent obstacleDetected goToState obstacleDetectedState
29         onEvent obstacleLeft goToState obstacleLeftState
30     endstate
31
```

```
32      state newCommandState
33          showMsg("obstacleLeft")
34          goToState robotControllerLoop
35      endstate
36
37      state lineDetectedState
38          showMsg("lineDetected")
39          goToState robotControllerLoop
40      endstate
41
42      state lineLeftState
43          showMsg("lineLeft")
44          goToState robotControllerLoop
45      endstate
46
47      state obstacleDetectedState
48          showMsg("obstacleDetected")
49          goToState robotControllerLoop
50      endstate
51
52      state obstacleLeftState
53          showMsg("obstacleLeft")
54          goToState robotControllerLoop
55      endstate
56  }
57
58  BehaviorOf lineReachedDetector
59  {
60      state lineReachedDetectorInit initial
61          goToState lineDetection
62      endstate
63
64      state lineDetection
65          raiseEvent lineDetected("")
66      endstate
67  }
68
69  BehaviorOf lineDisappearedDetector
```

```
70  {
71      state lineDisappearedDetectorInit initial
72          goToState lineDisappearance
73      endstate
74
75      state lineDisappearance
76          raiseEvent lineLeft("")
77      endstate
78  }
79
80  BehaviorOf obstacleReachedDetector
81  {
82      state obstacleReachedDetectorInit initial
83          goToState obstacleDetection
84      endstate
85
86      state obstacleDetection
87          raiseEvent obstacleDetected("")
88      endstate
89  }
90
91  BehaviorOf obstacleDisappearedDetector
92  {
93      state obstacleDisappearedDetectorInit initial
94          goToState obstacleDisappearance
95      endstate
96
97      state obstacleDisappearance
98          raiseEvent obstacleLeft ("")
99      endstate
100 }
```

Bibliografia

[Cha10] Hakima Chaouchi. *The Internet Of Things, Connecting Objects to the Web*. ISTE, 2010.

[CRH] Mike Cantelon, Nathan Rajlich, and TJ Holowaychuck. *Node.js in action*. Manning Publications.

[Dor06] Richard C. Dorf. *The Electrical Engineering Handbook, - Systems, Controls, Embedded Systems, Energy, and Machines - Third Edition*. Electrical Engineering Handbook. CRC Press, January 2006.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Computing Series. Addison-Wesley Professional, november 1994.

[Gro95] Object Management Group. Meta-object facility. `http://www.omg.org/mof/`, 1995. [Ultimo accesso verificato: 09-Febbraio-2014].

[Keg] Dan Kegel. The c10k problem. http://www.kegel.com/c10k.html.

[LP05] Luciano Lavagno and Claudio Passerone. *The Embedded Systems Handbook*. CRC Press, 2005.

[LW13] Dawei Li and Jie Wu. *Energy-aware Scheduling on Multiprocessor Platforms*. SpringerBriefs in Computer Science. Springer, 2013.

[Nat09] Antonio Natali. Introduction to the contact system. `http://edu222.deis.unibo.it/Deploy/contact/ContactOverview.pdf`, 2009. [Ultimo accesso verificato: 09-Febbraio-2014].

[NM08] Antonio Natali and Ambra Molesini. *Costruire Sistemi Software: dai modelli al codice*. Esculapio, 2008.

[Sch06] Douglas C. Schmidt. Model driven engineering. *IEEE Computer*, February 2006.

[Sch12] Maik Schmidt. *Raspberry Pi A Quick Start Guide*. The Pragmatic Programmers. The Pragmatic Bookshelf, 2012.

[Sli03] Carol Sliwa. Event-driven architecture poised for wide adoption. `http://www.computerworld.com/s/article/81133/Event_driven_architecture_poised_for_wide_adoption?taxonomyId=063`, May 2003. [Ultimo accesso verificato: 09-Febbraio-2014].

[Tea13] The Pi4J Team. The pi4j project, connecting java to the raspberry pi. http://pi4j.com/project-info.html, 2013.

[Tim05] Harold Timmis. *Pratical Arduino Engineering*. Technology In Action. Apress, November 2005.

[Uni05] International Telecommunication Unit. *The Internet of Things, Strategy and Policy Unit*. November 2005.

[WL06] Daniel Waddington and Patrick Lardieri. Model centric software development. *IEEE Computer*, February 2006.

[Xte13] Xtext. Tutorials and documentation for xtext 2.0. http://www.eclipse.org/Xtext/documentation/, 2013.