ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

CAMPUS DI CESENA
SCUOLA DI INGEGNERIA E ARCHITETTURA

Corso di Laurea Magistrale in Ingegneria Informatica

IL RUOLO DELLE ARCHITETTURE DI CONTROLLO NELLA PROGETTAZIONE E SVILUPPO DI APPLICAZIONI WEB: UN CONFRONTO FRA EVENT-LOOP E CONTROL-LOOP IL LINGUAGGIO DART E LINGUAGGI AGENT-ORIENTED

Elaborata nel corso di:
Programmazione Concorrente e Distribuita LM

Tesi di Laurea di:
ENRICO GALASSI

Relatore:
Prof. ALESSANDRO RICCI

ANNO ACCADEMICO 2013–2014
SESSIONE II - SECONDO APPELLO

PAROLE CHIAVE

Web Application
Event-Driven Programming
Google Dart
Agent-Oriented Programming
JaCaMo

a coloro che hanno creduto in me

Indice

Abstract

Quando si parla di architetture di controllo in ambito Web, il *Modello ad Eventi* è indubbiamente quello più diffuso e adottato. L'asincronicità e l'elevata interazione con l'utente sono caratteristiche tipiche delle Web Applications, ed un architettura ad eventi, grazie all'adozione del suo tipico ciclo di controllo chiamato *Event Loop*, fornisce un astrazione semplice ma sufficientemente espressiva per soddisfare tali requisiti. La crescita di Internet e delle tecnologie ad esso associate, assieme alle recenti conquiste in ambito di CPU multi-core, ha fornito terreno fertile per lo sviluppo di Web Applications sempre più complesse. Questo aumento di complessità ha portato però alla luce alcuni limiti del modello ad eventi, ancora oggi non del tutto risolti. Con questo lavoro si intende proporre un differente approccio a questa tipologia di problemi, che superi i limiti riscontrati nel modello ad eventi proponendo un architettura diversa, nata in ambito di IA ma che sta guadagno popolarità anche nel general-purpose: il *Modello ad Agenti*. Le architetture ad agenti adottano un ciclo di controllo simile all'Event Loop del modello ad eventi, ma con alcune profonde differenze: il *Control Loop*.

Lo scopo di questa tesi sarà dunque approfondire le due tipologie di architetture evidenziandone le differenze, mostrando cosa significa affrontare un progetto e lo sviluppo di una Web Applications avendo tecnologie diverse con differenti cicli di controllo, mettendo in luce pregi e difetti dei due approcci.

Capitolo 1

Introduzione

Il World Wide Web, conosciuto anche come WWW, Web o W3, ad oggi registra cifre enormi (per numero di dispositivi connessi, di utenti che ne usufruiscono e così via), in continua e rapida crescita. Il Web è diventata una piattaforma talmente vasta e diffusa che vi si pò trovare praticamente ogni tipo di applicazione, molte di queste oramai considerate scontate, come i servizi di social network o siti di condivisione video. Anche se non sembrerebbe, la nascita del Web è piuttosto recente; è stata la velocità con cui si è evoluto che ha permesso di raggiungere già oggi, scenari applicativi che fino a non molto tempo fa erano impensabili.

La prima proposta di World Wide Web venne formulata nel 1989 da Tim Berners-Lee. L'idea alla base del progetto era quella di fornire strumenti adatti a condividere documenti statici, in forma ipertestuale, disponibili su rete Internet tramite protocollo semplice e leggero. Lo scopo era quello di rimpiazzare i sistemi di condivisione di documenti basati su protocolli più vecchi come FTP. La prima proposta formale avvenne un anno dopo, nel novembre del 1990, sempre da parte di Berners-Lee assieme a Roberts Cailliau. L'architettura proposta era basata sui moduli client-server. Nell'agosto del 1991 venne messo on-line su Internet il primo sito Web, utilizzato inizialmente solo dalla comunità scientifica. Solamente nell'aprile del 1993 la tecnologia alla base del Web venne resa pubblica. Nella sua prima versione, esso poteva essere rappresentato, in maniera molto sintetica, nelle sue tre componenti principali:

- URL (Uniform Resource Locator): permette di indirizzare le risorse disponibili sui server per mezzo del suo meccanismo di accesso primario.

- HTTP (Hyper Text Transfer Protocol): protocollo di livello applicativo utilizzato per trasferire le risorse web.
- HTML (Hyper Text Markup Language): è il linguaggio utilizzato per la rappresentazione di documenti ipertestuali.

In questa sua prima versione (Web 1.0), il modello descritto ha una natura essenzialmente statica. L'utente può sì percorrere dinamicamente l'ipertesto, tuttavia l'insieme dei contenuti è prefissato staticamente. Questo è sicuramente un modello semplice, potente ed efficiente ma presenta dei grossi limiti. Da questo problema si sono iniziate a sviluppare diverse soluzioni, per fornire contenuti dinamici e servizi sempre più avanzati e complessi. Il World Wide Web stava rapidamente crescendo, supportato dalla nascita di tutta una serie di tecnologie, frameworks, scripting languages, plugins e via dicendo, in grado di fornire allo sviluppatore gli strumenti e le facilities necessarie per sviluppo di applicazioni Web sempre più complesse. Il paradigma di Web statico (o Web 1.0) si stava evolvendo verso un idea di Web *Dinamico*. In ambito di linguaggi di scripting, uno in particolare si affermò su tutti gli altri, arrivando ad essere oggi lo standard de facto per la parte client: JavScript.

Inizialmente concepito per migliorare le pagine Web in Netscape 2.0, JavScript è oggi considerato il linguaggio dominante del Web, facendo diventare potenziali rivali (come Flash o Silverlight) sempre meno utilizzati. In un certo senso, è possibile dire che JavScript sia il linguaggio più vicino al concetto di "*Write once, run everywhere*", originalmente coniato per Java, vista la diffusione dei Web Browser su praticamente ogni dispositivo elettronico (Computer, mobile device, smart-tv, etc.). A riprova di ciò, l'inserimento di JavScript nelle features principali (assieme ad HTML, CSS e DOM) del nuovo standard per il Web, HTML5, il cui scopo è proprio quello di fornire rich-content senza l'ausilio di plugins esterni (come Flash) che sono spesso causa di problemi con il browser.

Nato quindi con l'idea di fornire una rete di supporto per la condivisione di semplici documenti statici in forma ipertestuale, oggi Internet è la base per una moltitudine di applicazione e servizi: **il browser, da semplice client HTTP e visualizzatore di documenti ipertestuali, si è trasformato in una vera e propria piattaforma multimediale per sistemi eterogenei**.

Parallelamente alla crescita del Web, un altro fenomeno che sta caratterizzando gli ultimi anni è la *Rivoluzione del Multi-Core*. La diffusione dei dispositivi con chip contenenti un numero sempre più elevato di processori fisici e del network computing, ha portato la concorrenza e tutti gli aspetti correlati, come la programmazione asincrona e distribuita, ad avere un ruolo sempre più importante nello sviluppo di software di tutti i giorni. Applicazioni con un certo livello di complessità richiedono di saper sfruttare appieno le risorse a disposizione; la capacità di programmare software in grado di servirsi appieno di tutti i core delle moderne CPU, di poter distribuire il carico di lavoro sulla macchina locale ma anche su macchine distribuite, sta diventando un requisito oramai fondamentale. Storicamente l'approccio mainstream per la gestione della concorrenza riguardava l'utilizzo dei *Threads*. Tuttavia la programmazione multi-thread è da sempre considerata molto difficile, anche per il programmatore più esperto. Meccanismi di *lock* e di *sincronizzazione*, se non usati con cognizione di causa, possono portare a situazioni disastrose per il software, come inconsistenza dei dati o, peggio ancora, a *deadlock*. Il modello multi-thread, nella programmazione di software ad alto livello, è stato quindi messo in discussione a causa delle sue problematiche e della sua difficoltà. Questo ha favorito la nascita di nuovi modelli o la riproposizione di alternative, come ad esempio gli *Attori*, atte a risolvere i suddetti problemi ed in grado, al tempo stesso, di adattarsi alla rivoluzione multi-core in atto. Una delle alternative al modello di programmazione multi-threads è quello riguardante la *Programmazione ad Eventi*.

La programmazione asincrona event-driven classica elimina alla base alcuni dei problemi del modello multi-thread, ed introduce concetti semplici quali *Eventi*, *Coda di Eventi* ed *Event Handlers*, spostando quindi l'attenzione del programmatore sul "come reagire a determinati eventi". Ad ogni evento sarà quindi registrato un determinato handler e la composizione di quest'ultimi determinerà il comportamento generale del programma. Fra i contesti applicativi dove l'adozione di questo paradigma risulta particolarmente naturale vi è senza dubbio la programmazione di interfacce grafiche per gli utenti, denominate *GUI* (Graphic User Interface). Uno dei motivi del successo di questo modello è da ricercarsi anche nella stretta correlazione che ha con le tecnologie del Web; la tipologia classica di Web Applications ben si adatta al suddetto paradigma di programmazione ed infatti il modello di controllo di JavScript è proprio quello event-driven. La crescente

complessità delle applicazioni Web ha tuttavia portato alla luce alcuni limiti legati al modello di programmazione ad eventi. La natura asincrona è sì un vantaggio per quanto riguarda, ad esempio, l'esecuzione di operazioni non bloccanti, ma è anche una delle cause di noti problemi di mantenibilità del codice, come l'*asyncronous spaghetti code*. Inoltre la tradizionale architettura ad eventi sfrutta un unico flusso di controllo, non risultando adatto a sfruttare il parallelismo derivante dalle architetture multi-core di ultima generazione. Numerose soluzioni, sia a livello di API dei linguaggi che di estensione del modello base, sono state proposte ed adottate negli anni con lo scopo di risolvere i problemi emersi e per meglio adattarsi alla rivoluzione hardware in atto. Le più conservative si sono "limitate"ad introdurre nuovi concetti ed astrazioni ai linguaggi già esistenti, evolvendoli a tal punto da far diventare API di librerie inizialmente accessorie dei veri e propri standard per la programmazione event-driven. Soluzioni più radicali propongono invece un taglio netto col passato, formalizzando nuovi linguaggi che si presentano come i nuovi standard per la programmazione di Web Applications; uno esempio su tutti è Dart di Google.

La prima parte di questo lavoro si focalizzerà quindi sullo studio del modello ad eventi, del ciclo di controllo che lo regola, sui vantaggi derivanti dall'adozione di un simile modello, sulle problematiche da esso derivanti e sulle soluzioni che vengono adottate per cercare di risolverle. Tutto questo studio sarà fatto con un occhio sia su ciò che è considerato lo standard attuale, JavScript, sia su uno dei linguaggi che si vorrebbe proporre come nuova lingua franca del Web, Dart. Alla luce delle considerazioni fatte nella prima parte e dei problemi evidenziati, la seconda parte proporrà un ulteriore modello, un approccio che presenta un architettura di base con alcune sostanziali differenze rispetto a quella ad eventi: *il Modello ad Agenti*.

La programmazione agent-oriented nasce in ambito di Intelligenza Artificiale, con l'idea di combinare in un unica entità autonoma, l'agente appunto, *reattività* e *proattività* in maniera efficace ed efficiente. Queste entità implementano al loro interno un ciclo di controllo che prende ispirazione dal ciclo di ragionamento umano, chiamato appunto *Control Loop* o *Reasoning Cycle*. Questo ciclo presenta alcune analogie con il tipico Event Loop, tuttavia è esteso con dei concetti distinti che potrebbero essere interessanti anche nell'ambito delle Web Applications. La programmazione ad agenti risulta inoltre naturalmente adattabile per applicativi scalabili. Il carico di lavoro

può essere diviso ed assegnato ad agenti distinti, i quali possono essere distribuiti sui vari core fisici del processore o addirittura su nodi distinti della rete. Questa tipologia di sistemi dove più agenti, ognuno con un proprio compito ben preciso, cooperano per raggiungere un obbiettivo comune viene definita *Multy Agent Systems* (o *MAS*). A tal proposito verrà presentato un tool di programmazione per lo sviluppo di sistemi multi-agente che verrà utilizzato come riferimento nell'ambito di questo lavoro: **JaCaMo**.

JaCaMo è un framework che combina tre separate tecnologie sviluppate per diversi anni e quindi sufficientemente robuste e supportate:

- **Jason**, per la programmazione di agenti autonomi;
- **CArtAgO**, per la programmazione di artefatti;
- **Moise**, per la programmazione di organizzazioni multi-agente.

L'ultima parte si occuperà di effettuare un confronto fra i due modelli proposti, quello ad eventi e quello ad agenti, mediante lo sviluppo e studio di un set di test-case, evidenziando le differenze fra un approccio event-oriented ed uno agent-oriented, sottolineandone i vantaggi e gli svantaggi dell'uno sull'altro e via dicendo.

Capitolo 2

Rivoluzione del Multi Core

Questa sezione prenderà spunto da un lavoro di Manferdelli, Govindaraju e Crall [1] e da un articolo di Herb Sutter [2] per descrivere brevemente la rivoluzione in atto da oramai qualche anno nell'ambito dei personal computers, e non solo.

"The free lunch is over" recita Sutter, descrivendo come la transizione delle CPU da un singolo core a più core, nei dispositivi elettronici d'uso quotidiano, necessiti di un cambiamento che passa dai sistemi operativi e arrivi fino ai linguaggi e gli strumenti di programmazione, che dovrebbero cambiare permanentemente il modo in cui si concepisce il software oggi. Questa svolta porta moltissime opportunità, ma anche altrettante sfide che necessitano di essere risolte. Nuovi modelli ed astrazioni si devono formalizzare, con lo scopo di provvedere allo sviluppatore gli strumenti adatti per poter produrre applicazioni di qualità, che forniscano un esperienza più ricca di contenuti per l'utente finale.

2.1 L'inizio dell'era del Multi Core

Negli ultimi anni, l'industria dei computer ha visto un enorme miglioramento nei prodotti hardware e software. Uno dei settori che più ha richiesto questo cambiamento è quello dell'intrattenimento. I videogiochi ad esempio, devono confrontarsi continuamente con la domanda di avere un livello di realismo maggiore, costringendo quindi i progettisti hardware di processori, memorie e schede grafiche, a sviluppare soluzioni che spingano gli standard di qualità a livelli sempre più elevati. Questo miglioramento continuo è sta-

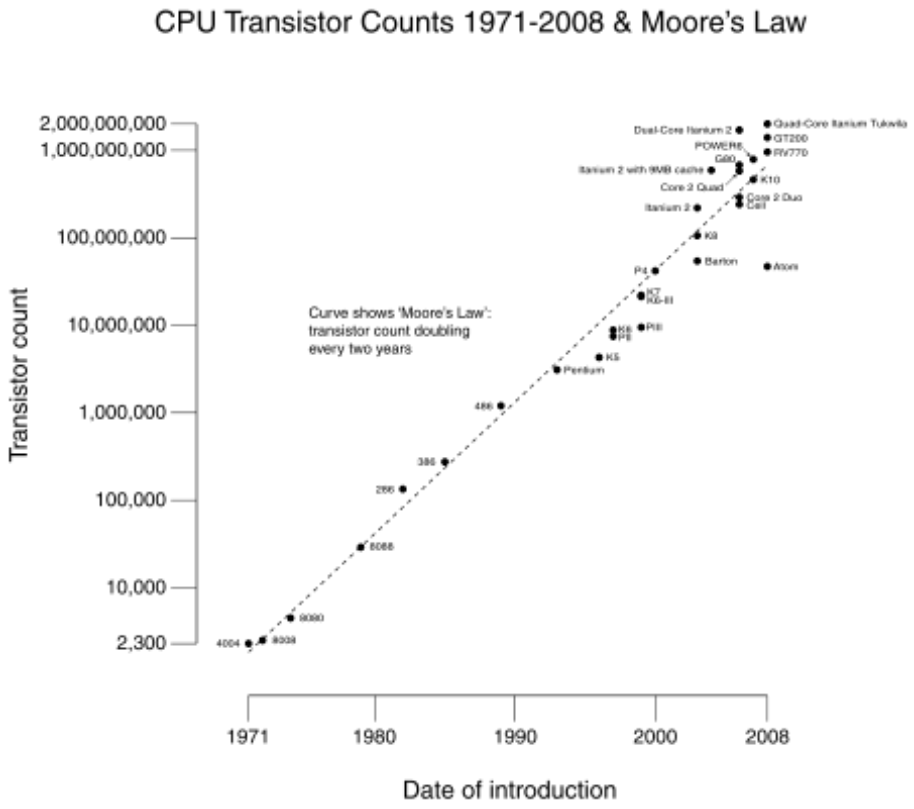

Figura 2.1: Grafico rappresentante la legge di Moore, dal 1971 fino al 2008

to reso possibile grazie anche alla legge che, negli ultimi 40 anni, ha guidato la crescita esponenziale nelle capacità hardware dei computer: la *legge di Moore* (figura 2.1).

La prima legge di Moore recita che *"Le prestazioni dei processori, e il numero di transistor ad esso relativo, raddoppiano ogni 18 mesi"*. Anche se questa legge continuerà a fornire ai progettisti di architetture hardware sempre più risorse, le prestazioni seriali delle CPU general-purpose tuttavia non stavano subendo un altrettanto significativo miglioramento negli anni.

2.2 *"The Free Performance Lunch"*

Fino a qualche tempo fa, il programmatore software non si doveva preoccupare particolarmente di come ottimizzare il codice quando scriveva un programma. Il miglioramento continuo delle performance avveniva regolarmente grazie all'arrivo delle nuove generazioni di CPU e di memorie/dischi rigidi. Grazie a questo, molte classi di applicazioni hanno ottenuto benefici costanti e a gratis per svariati anni, senza nemmeno il bisogno di rilasciare nuove versioni del programma o senza fare nulla di particolare.

Questo era considerato vero fintanto che, per aumentare le prestazioni delle CPU, si lavorava principalmente in tre aree:

- frequenza del clock;
- ottimizzazione dell'esecuzione;
- cache.

Il primo aspetto è facilmente comprensibile: aumentare la frequenza del clock significa incrementare il numero di istruzioni al secondo eseguite dal processore. Semplicemente un dato carico di lavoro viene eseguito più velocemente.

Il secondo aspetto riguarda tutte quelle soluzioni, hardware o di firmware, a livello di CPU che permettono di ottimizzare il carico di lavoro ad ogni singolo ciclo di clock. Alcuni esempi di tecniche di questo tipo sono il pipelining o l'esecuzione speculativa di operazioni che dipendono da risultati non ancora ottenuti delle istruzioni precedenti.

Il terzo aspetto riguarda quello dell'aumento delle dimensioni della cache di primo e secondo livello. Essendo la tipologia di memoria più veloce, accedere alla cache richiede molto meno tempo che, ad esempio, leggere dalla RAM o dal disco rigido. Aumentarne la grandezza significa aumentare il quantitativo di dati che possono immagazzinare e quindi si diminuiscono gli accessi alle memorie più lente ed il rischio di chache-misses.

2.3 *"The Free Performance Lunch is Over"*

Se per quanto riguarda la legge di Moore, ancora oggi fornisce un valido strumento di predizione delle capacità di calcolo dei nuovi processori, qual-

cosa iniziò a cambiare nelle tecniche di miglioramento delle prestazioni delle CPU.

Come si può vedere in figura 2.2, al contrario del numero dei transistors che continuava ad aumentare nel corso degli anni, l'aumento della frequenza del clock delle CPU si stava appiattendo. Il motivo è da ricercarsi nei limiti fisici in cui ci si stava andando a scontrare; aumentare la frequenza significa aumentare anche il calore da dissipare, i consumi e i rischi di perdita di corrente. Il guadagno ottenuto in performance non valeva più il costo per gestirne i problemi derivanti.

Anche per quanto riguarda le tecniche di ottimizzazione non si stavano più ottenendo i benefici sperati. Tenendo sempre come riferimento la figura 2.2, abbiamo l'*ILP*, il *Parallelismo a livello d'istruzione*, che rappresenta una misura delle istruzioni di un programma che il processore riesce ad eseguire parallelamente. Anche in questo caso, il guadagno in termini di performance si stava sempre più azzerando.

Due dei tre aspetti su cui si lavorava in passato per aumentare le prestazioni delle CPU non risultavano più adeguati e nuove vie andavano percorse per le future generazioni di processori. Da qui in avanti, **le applicazioni non avrebbero più potuto godere di incrementi di prestazioni a gratis senza una ri-progettazione dell'applicazione stessa**.

2.4 La Nuova Rivoluzione Software

Mentre ancora oggi ci si può aspettare che le dimensioni delle cache sul chip continuino ad aumentare, in futuro non si potrà più far affidamento sull'aumento della frequenza del clock e sull'ottimizzazione dell'esecuzione delle istruzioni a livello di processore. Per continuare a migliorare le prestazioni delle CPU, si incominciò quindi a lavorare su due nuovi aspetti:

- hyperthreading;
- multi-core.

Per *hyperthreading* si intende la capacità di eseguire due o più thread in parallelo all'interno di una singola CPU, mentre per *multi-core* si indica la possibilità di avere due o più processori fisici su di un singolo chip.

Cosa significa questo? Che affinché le nuove applicazioni beneficino dell'esponenziale aumento del throughput (numero di istruzioni nell'unità

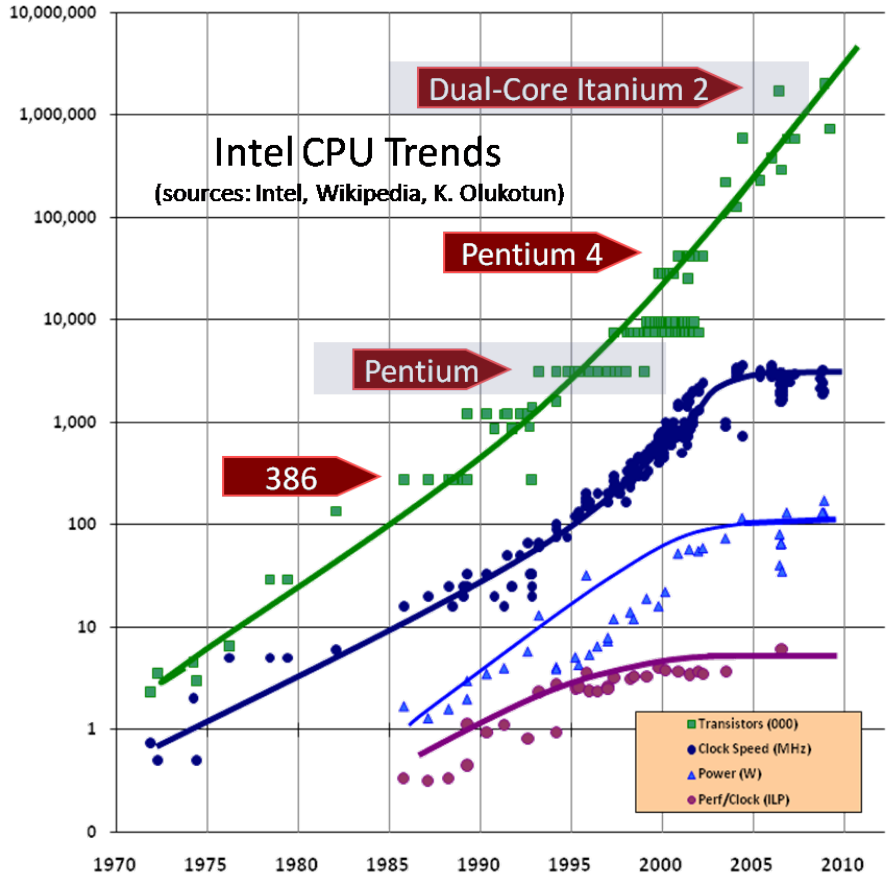

Figura 2.2: Introduzione delle CPU Intel(grafico aggiornato ad Agosto 2009; testo dell'articolo originale del Dicembre 2004)

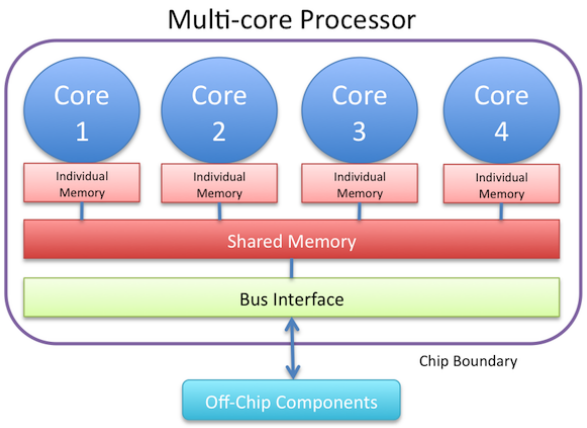

Figura 2.3: Rappresentazione di una CPU multi-core

di tempo) nei nuovi processori, tutto ciò che riguarda la programmazione multi-thread, come la concorrenza, dovrà essere presa in considerazione. Questi aspetti tuttavia già esistono e non sono concetti nuovi, il problema è che sono relegati ad una piccolissima parte degli sviluppatori, perché la programmazione multi-thread è notoriamente difficile!

È necessario quindi ricercare nuove astrazioni che permettano di alleggerire il compito del programmatore e di far fronte a tutte le problematiche legate alla programmazione concorrente.

Capitolo 3

Dai Thread al Modello ad Eventi

La programmazione concorrente e parallela diventerà dunque un aspetto sempre più centrale nella progettazione e nello sviluppo di software di tutti i giorni. Partendo dal modello computazionale sequenziale, familiare alla maggior parte dei programmatori, si è pensato che il modello multi-thread ne fosse la naturale evoluzione. Dal punto di vista dei linguaggi di programmazione, è bastato applicare piccole modifiche sintattiche per supportare i threads, oltre al fatto che i sistemi operativi e le architetture si sono evolute in una direzione tale che permettono il supporto di questi in maniera efficiente.

Nel prossimo paragrafo, si farà una breve panoramica su ciò che riguarda la programmazione multi-thread e si spiegheranno i motivi principali per cui questo approccio potrebbe non essere il più adatto per sfruttare le moderne architetture multi-core.

3.1 Modello Multi-Thread

I threads possono essere visti come flussi di controllo distinti ed indipendenti, che dispongono di uno stato privato e di uno stato condiviso su cui operano assieme agli altri thread (figura 3.1).

Vi sono due importanti aspetti da sottolineare:

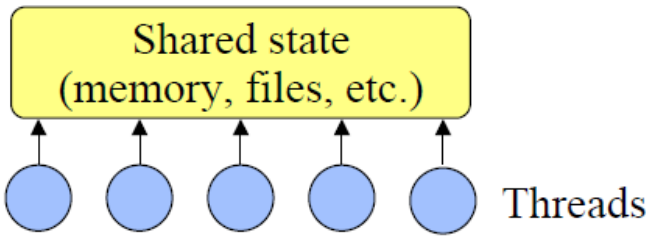

Figura 3.1: Rappresentazione dei Threads che operano sulla memoria condivisa

1. Questi threads comunicano tra di loro mediante lettura e scrittura sincrona della memoria condivisa;

2. L'ordine di esecuzione delle singole operazioni dei vari thread, su di uno stesso core, è determinata dallo scheduler e non dal programmatore.

Il primo aspetto implica che sono necessari meccanismi di sincronizzazione per garantire la mutua esclusione durante l'accesso ai dati, quali *locks*. Se per sbaglio un thread, prima di eseguire una sequenza di operazioni su di un set di dati condivisi, non ne ottiene l'accesso esclusivo, può accadere che altri thread accedono concorrentemente a quei particolari dati, generando il rischio potenziale di avere risultati corrotti. Questo problema viene chiamato *Race Condition* (o *Corse Critiche*) e può sempre accadere se più di un thread accede ad una risorsa condivisa senza essere sicuro che un altro thread abbia finito di operare su di essa, prima di accedervi (figura 3.2).

I locks sono però strumenti da utilizzare con molta attenzione. Un uso eccessivo di questi porta ad un deterioramento delle performance, oltre che aumenta il rischio di introdurre dipendenze circolari fra locks. In particolare quest'ultimo caso può generare situazioni disastrose per il programma. Su tutti, il problema dei *Deadlocks*, dove due flussi di controllo distinti aspettano che l'altro liberi il lock ad una risorsa e quindi nessuno dei due difatti procede (come mostrato in figura 3.3).

Il secondo aspetto implica che l'esecuzione di thread su di un core, può essere interrotta in qualsiasi momento dallo scheduler per mandare in esecuzione un altro thread, sempre sullo stesso core. Il *cambio di contesto*

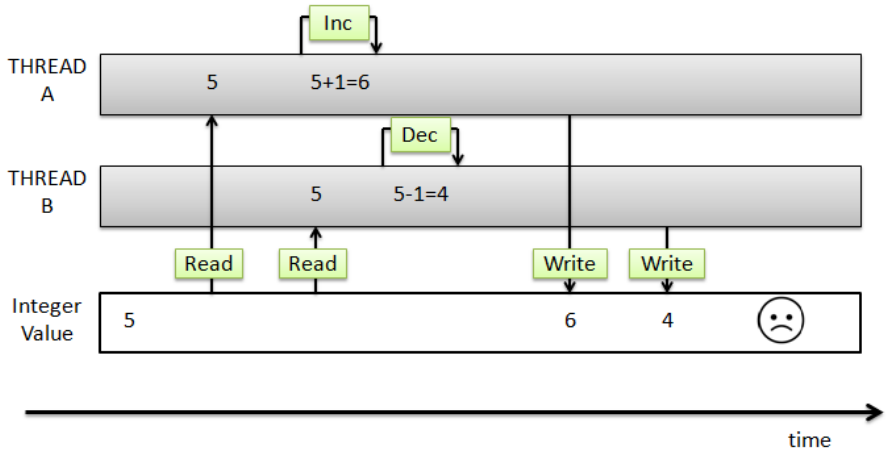

Figura 3.2: Esempio di Corsa Critica, dove l'esecuzione da parte di due thread di un incremento e di un decremento di una variabile, genera un risultato non corretto

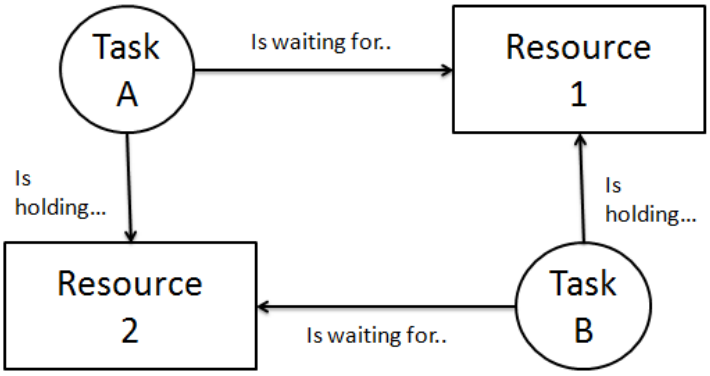

Figura 3.3: Condizione di Deadlock fra due thread

avviene automaticamente e non è mai eseguito esplicitamente da un thread. Tutto ciò introduce un enorme grado di non-determinismo nell'esecuzione del programma.

Il gap fra il modello computazionale sequenziale, caratterizzato da tratti distintivi quali predicibilità e determinismo, e quello multi-thread risulta dunque molto più grande di quanto non sembrasse inizialmente.

Il programmatore, in questo contesto, si ritrova il compito di dover ridurre il non-determinismo del programma, gestendo i vari meccanismi di sincronizzazione fra i vari thread, un lavoro estremamente difficile e prono ad errori non sempre facilmente individuabili.

3.2 Thread ed Eventi: Il Caso delle GUI

Nei tradizionali modelli sequenziali di programmazione, l'ordine delle azioni e delle operazioni da svolgere viene decisa dal programmatore. L'esecuzione è quindi deterministica: dato un certo input al programma, ci si aspetta sempre lo stesso output, ogni volta che viene mandato in esecuzione. Tuttavia una caratteristica delle applicazioni al giorno d'oggi è la loro sempre più crescente interattiva; devono essere in grado di gestire svariate tipologie di eventi, eseguire task ed aggiornare il proprio stato conseguentemente. L'esempio più semplice è la tipica GUI, che deve reagire e coordinare gli eventi "lanciati"dall'utente come la pressione di un tasto o un click del mouse. In questo caso, un modello sequenziale risulterebbe semplicemente inappropriato, per il semplice fatto che non è possibile predire l'ordine con cui gli eventi arriveranno. Il flusso di controllo di un programma non è quindi deciso a priori dal programmatore ma viene guidato dall'ordine con cui arrivano gli eventi. Come si gestiscono dunque gli eventi generati da una GUI in un modello di programmazione multi-thread? Può sembrare strano, tuttavia la prassi è quella di delegare ad **un unico thread** il controllo degli eventi. Per fare ciò, questo thread tipicamente dispone di una *Coda di Eventi* ed un riferimento, per ogni tipo di evento, al corrispettivo *Event Handler* (figura 3.4). In Java ad esempio, il codice per la gestione degli eventi dei componenti Swing gira su di un unico thread speciale chiamato *Event Dispatch Thread*. Il motivo per cui si è deciso di avere un unico flusso di controllo

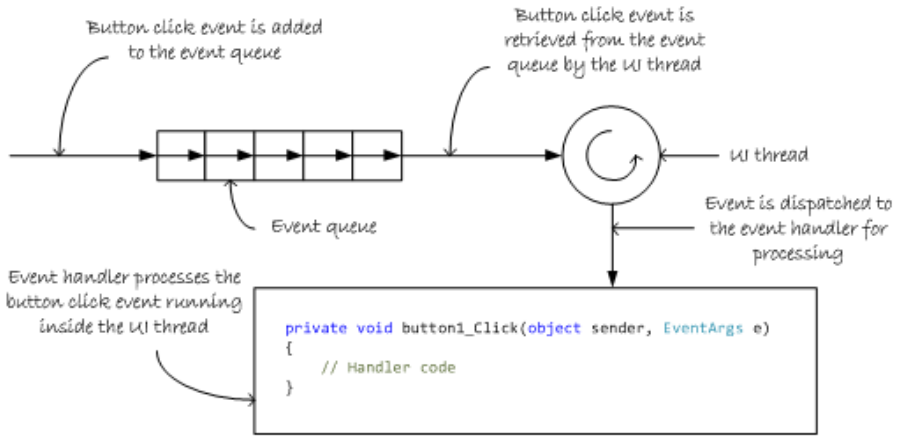

Figura 3.4: Flusso di gestione di un evento *Button Click*

responsabile degli eventi è perché creare una libreria GUI multi-thread safe incontra fondamentali problemi [8].

Per l'organizzazione e la progettazione di interfacce grafiche, il pattern di riferimento è senza ombra di dubbio il *Pattern Model-View-Controller* (figura 3.5). Permette di suddividere un'applicazione, o anche la sola interfaccia dell'applicazione, in tre componenti distinti:

- **Modello**: elaborazione / stato;
- **View** (o viewport): output;
- **Controller** input

Il Modello tipicamente gestisce un insieme di dati logicamente correlati, risponde alle interrogazioni sui dati e alle istruzioni di modifica dello stato. Si preoccupa di generare un evento quando lo stato cambia e registra (in forma anonima) gli oggetti interessati alla notifica.

La View altro non è che il gestore di un'area di visualizzazione, nella quale presenta all'utente una vista dei dati gestiti dal modello. Effettua quindi una mappa dei dati (o parte di essi) del modello in oggetti e li visualizza su di un particolare dispositivo di output. Importante è che si deve

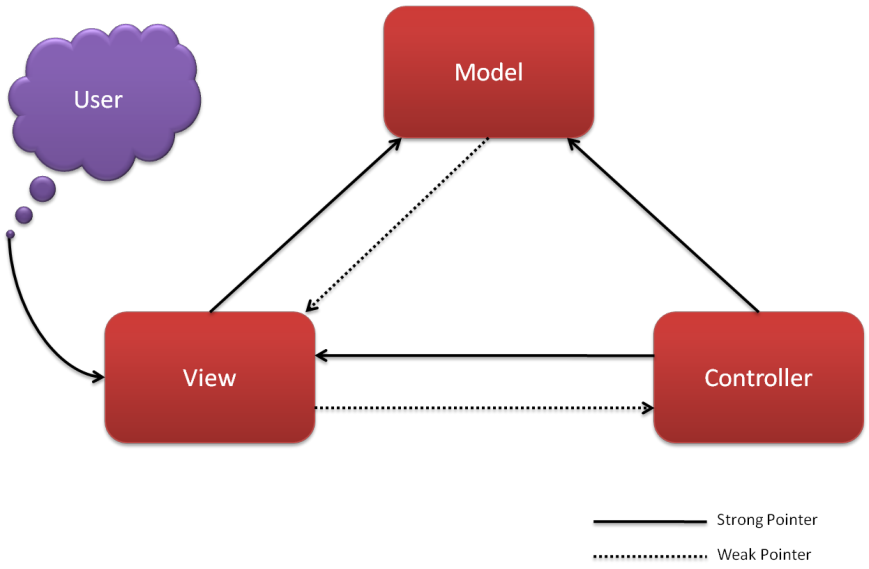

Figura 3.5: Tipica Architettura Model-View-Controller

registrare presso il modello per ricevere l'evento di cambiamento di stato.

Il Controller è la parte che gestisce gli input dell'utente (mouse, tastiera, etc.), mappa le azione dell'utente in comandi che invia al modello e/o alla View che effettuano le operazioni appropriate.

La politica di comportamento dell'aggiornamento delle varie parti esaminate qui sopra viene definita mediante un ulteriore design pattern, il *Pattern Observer* (figura 3.6). Infatti, prendendo ancora come esempio Java, un Modello deve estendere la classe *java.util.Observable* (nella figura 3.6, si tratta del *Subject*) mentre la View deve implementare l'interfaccia *java.util.Observer*. Anche il Controller è una specie di *Observer* degli eventi lanciati dalla GUI, *listener*. L'idea alla base di questo design pattern è quella di mantenere separati il più possibile contesti distinti; nell'esempio della GUI abbiamo il Modello, la View ed il Controller. Per ognuno di questi componenti esistono compiti distinti da assolvere, che è possibile identificare in una serie di task. In un approccio multi-thread, per ogni task si cerca di assegnare un thread, i quali sono poi organizzati in modo

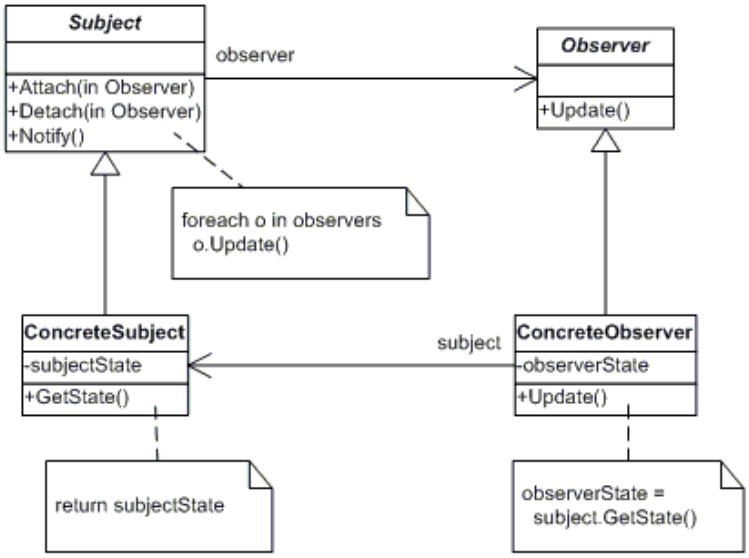

Figura 3.6: Il Pattern Observer

opportuno; riprendendo nuovamente l'esempio della GUI, la gestione degli eventi dell'interfaccia viene affidata all'Event Dispatch Thread.

Ogniqualvolta si percepisce un nuovo evento, seguendo le specifiche del Pattern Observer, il thread che si occupa della loro gestione notifica a tutti i soggetti interessati dell'avvenuta di tale evento, **trasferendo** il proprio flusso di controllo in ognuno di essi ed eseguendone gli handler definiti. Riprendendo l'esempio di Java, l'Event Dispatch Thread andrà quindi ad eseguire i metodi definiti nel Controller per gestire gli eventi.

Si occupa dunque di qualcosa che non gli dovrebbe competere, andando in un contesto che non gli appartiene, eseguendo qualcosa per cui non era stato pensato.

Questo fatto viene definito *Inversione di Controllo*, perché il flusso di controllo del programma non è più affidato esclusivamente ai threads che dovrebbero occuparsene, ma viene **invertito** e passa al thread che semplicemente gestisce gli eventi della GUI. Dunque il controllo del programma, specialmente nelle applicazioni altamente interattive, dipenderà in senso stretto dagli eventi generati, perché nei fatti è il thread che gestisce gli eventi che eseguirà il controllo associato. Parallelamente a questo, i threads

realmente incaricati al controllo del programma, potrebbero essere impegnati nell'esecuzione di altre funzioni, rischiando di andare a creare delle interferenze con il thread che gestisce gli eventi, con tutte le conseguenze già descritte nella precedente sezione.

3.3 Architettura ad Eventi: Un Modello Generale

Dati i problemi evidenziati nel modello multi-thread, si è vista la necessità di effettuare un passo indietro: ritornare ad un modello più semplice che, allo stesso tempo, fosse adatto alle crescenti esigenze di interattività delle moderne applicazioni, che possa gestire in maniera adeguata lunghe operazioni di I/O e che eviti il problema dell'*Inversione di Controllo*. Queste esigenze hanno portato a formalizzare un nuovo modello, quello ad eventi, che ci si appresta ad illustrare nel dettaglio.

3.3.1 Descrizione del Modello

Tipicamente, in tutte le architetture o modelli ad eventi, si possono identificare i seguenti concetti:

- Eventi: la nozione centrale di questo modello, gli eventi rappresentano un qualcosa che accade in un certo momento all'interno del ciclo di vita dell'applicazione;
- Coda degli Eventi (o Event Queue): coda FIFO (First In, First Out) dove gli eventi "catturati"vengono depositati temporaneamente per essere poi processati;
- Event Loop: il flusso di controllo che regola l'esecuzione all'interno del modello, si tratta di un che ad ogni iterazione estrae un evento dalla Event Queue ed esegue l'Event Handler associato;
- Event Handlers: procedura di gestione associata ad un particolare evento che viene eseguita ogni volta che il suddetto viene estratto dall'Event Queue;

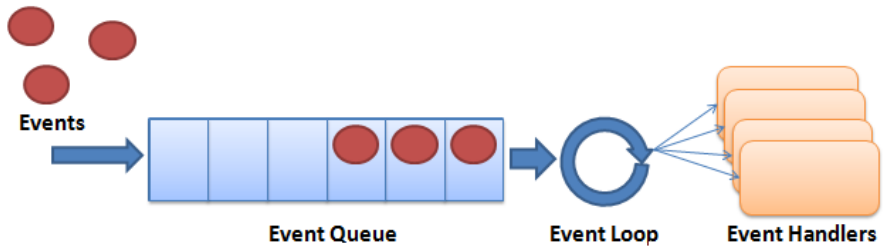

Figura 3.7: Il modello ad eventi con le sue principali componenti: Event Queue, Event Loop ed Event Handlers

La figura 3.7 dà una rappresentazione di come sono organizzati i concetti sopra esposti mentre il seguente pseudo-codice fornisce una implementazione del funzionamento del modello generale ad eventi:

```
while(true) {
 Event e = eventQueue.next();
 switch(e.type) {
   ...
 }
}
...
void onEventFired(FiredEvent e) {
//Process the event
}
...
```

Si usa quindi un meccanismo a *callback asincrona* dove una callback è fondamentalmente una funzione che viene passata come parametro ad un evento che può accadere nel corso dell'esecuzione del programma. Se una callback è in esecuzione vuol dire che l'evento associato è stato percepito dall'Event Loop. Si noti che, quando è in esecuzione una callback, il flusso di controllo rimane all'interno della callback stessa finché l'esecuzione di questa non termina. Questo, unito al fatto di avere un solo flusso di controllo, impedisce che avvengano corse critiche fra callback distinte. In figura 3.8 viene mostrato un esempio che mette in evidenza il funzionamento dell'Event Loop nella gestione di eventi distinti.

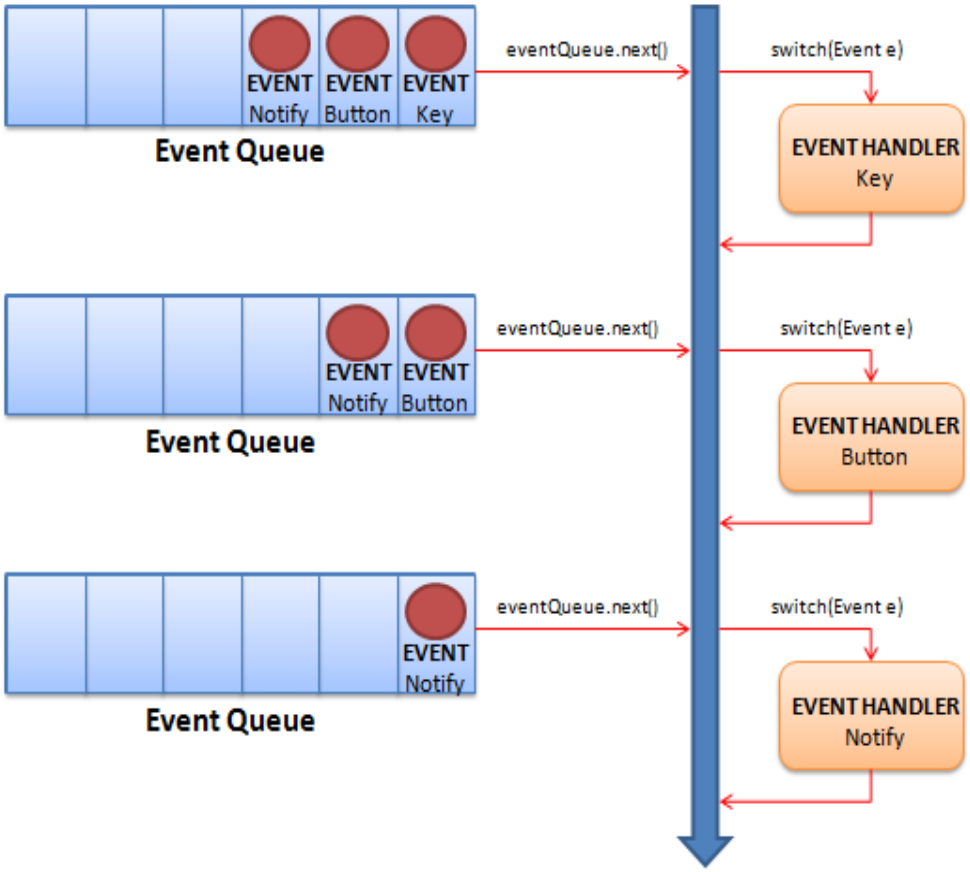

Figura 3.8: Esempio di esecuzione dell'Event Loop, avendo ricevuto in ordine un evento di bottone premuto, un evento di click del mouse e un evento di notifica esterna

3.3.2 Problematiche

Le callback sono largamente utilizzate nella programmazione event-driven per ottenere risultati di computazioni asincrone. Invece che bloccare il flusso di controllo in attesa di un particolare evento (il cui avvenimento non è deterministico) semplicemente si registra un metodo da eseguire nel momento che si percepisce quel dato evento. Una volta terminato il metodo, si prosegue gestendo o attendendo (nel caso di Event Queue vuota) l'evento successivo. Questo modello di esecuzione può però generare dei problemi.

Durante l'esecuzione di un Event Handler, il ciclo dell'Event Loop si ferma finché il metodo non viene terminato. Questo impedisce sì il problema corse critiche tra Event Handlers distinti, ma di fatto blocca momentaneamente la gestione degli eventi. Con riferimento la figura 3.9, se associamo ad un evento una funzione che richiede di eseguire tutta una serie di operazioni computazionalmente costose in termini di tempo, tutti gli altri eventi dovranno aspettare la terminazione del suddetto prima di poter essere gestiti. Per ridurre questo problema, di prassi si cerca di mantenere gli Event Handlers di breve durata, al fine di avere un Event Loop che sia il più reattivo possibile. In caso di lunghe operazioni da eseguire, si cerca di suddividerle in frammenti più piccoli e meno pesanti; la terminazione di una genererà poi un evento la cui callback è la prosecuzione delle operazioni precedenti. Il codice organizzato in questo modo risulta tuttavia estremamente frammentato e di più difficile lettura rispetto ad un programma scritto con un modello sequenziale, dove le istruzione vengono semplicemente lette dall'alto verso il basso. Qui l'ordine di arrivo degli eventi non è predicibile e quindi l'esecuzione generale del programma non è chiara perché si "salta"continuamente da un frammento di codice all'altro. Questo problema viene definito con il termine di *Spaghetti Code*, con il quale si evidenzia l'analogia fra codice di questo tipo e un piatto di spaghetti, ovvero un mucchio di fili intrecciati ed annodati.

Non solo, quando un certo numero di eventi sono attesi, con uno stato che deve essere mantenuto da un handler all'altro, si crea una gerarchia di callbacks annidate che rendono il codice illeggibile e poco intuitivo da comprendere. Questa strutturazione di callbacks innestate viene anche chiamata *Pyramid of Doom*. Ad esempio:

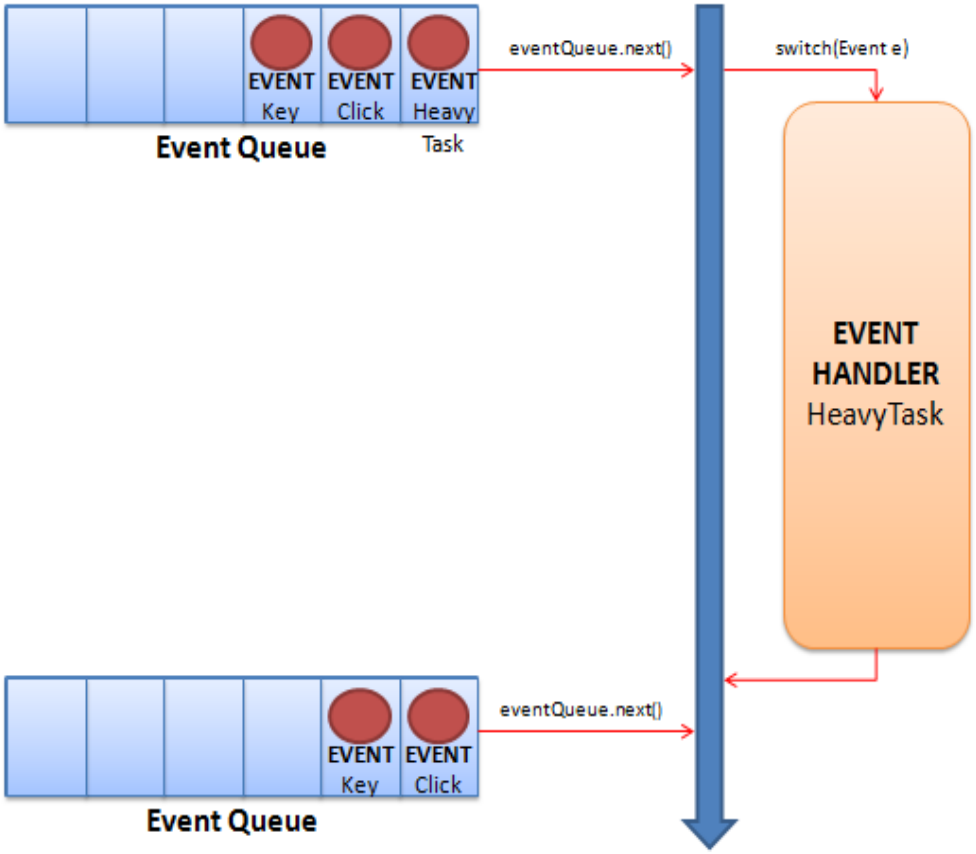

Figura 3.9: Esempio di evento a cui è associato un Event Handler che richiede una lunga computazione che blocca l'Event Loop dal processare nuovi eventi

```
getMessag{ function(result) {
 sendResponse{ function(code) {
  registerLog{ function(log) {
   //and so on...
    }
   }
  }
 }
}
```

Il programmatore è quindi costretto a scrivere e pensare il codice in maniera poco familiare rispetto alla semplicità di un programma sequenziale, dovendosi preoccupare di gestire la logica di tutti questi Event Handlers, separati o innestati che siano. Tutto questo definisce ciò che viene più comunemente conosciuto come *Callback Hell* o *Callback Hell*.

3.3.3 Soluzioni Proposte

Promises

Una soluzione per gestire i problemi derivanti dalle callbacks sono le *Promises*. In generale, una Promise può essere definita come un oggetto che rappresenta il risultato di una operazione ancora incerto che può avere esito positivo o negativo.

In JavScript, le principali librerie che fanno utilizzo di questo meccanismo sono JQuery e Q (quest'ultima segue la specifica Promise/A), entrambe offrono le stesse potenzialità con piccole differenze a livello sintattico. Facendo un piccolo esempio, mostriamo come sarebbe un frammento di codice con le vecchie callbacks:

```
object.save({ key: value }, {
  success: function(object) {
    // the object was saved.
  },
  error: function(object, error) {
    // saving the object failed.
  }
});
```

e il corrispettivo con le Promises:

```
object.save({ key: value }).then(
  function(object) {
    // the object was saved.
  },
  function(error) {
    // saving the object failed.
  });
```

Anche se ad una prima occhiata sembrerebbe tutto molto simile, il vero potenziale delle Promises viene fuori quando si tratta di incatenarne una dietro l'altra.

Si mostra un altro esempio che mette a confronto ancora una volta le callbacks e le Promises, dove nel primo caso è possibile notare come la dipendenza fra una callback e l'altra generi una struttura del codice "a piramide"di difficile comprensione, da cui deriva il nome *Pyramid of Doom*:

```
Parse.User.logIn("user", "pass", {
  success: function(user) {
    query.find({
      success: function(results) {
        results[0].save({ key: value }, {
          success: function(result) {
            // the object was saved.
          }
        });
      }
    });
  }
});
```

Utilizzando invece Promises incatenate, il tutto risulta molto più leggibile e meglio strutturato:

```
Parse.User.logIn("user", "pass").then(function(user) {
  return query.find();
}).then(function(results) {
  return results[0].save({ key: value });
}).then(function(result) {
  // the object was saved.
});
```

Reactive Paradigm

Un'altra proposta che sta prendendo piede come soluzione per sviluppare applicazioni event-driven è quella di adottare il *Paradigma di Programmazione Reattivo*. Il concetto alla base di quest'ultimo è quello di scrivere il programma come se fosse una serie di relazioni fra dati e/o eventi esterni che saranno poi risolte dal linguaggio stesso, liberando il programmatore dal compito di gestire manualmente le computazioni e dalla preoccupazione dall'ordine di arrivo degli eventi. Non è quindi strano intuire il motivo per cui nel mondo Web, dove alle applicazioni è richiesto di comunicare in maniera asincrona coi server e contemporaneamente fornire un elevato grado di interattività con l'utente, questo paradigma stia guadagnando interesse.

Prendendo ancora una volta come esempio JavScript, il principale supporto a questo paradigma è dato da Flapjax, che può essere usato sia come linguaggio che viene compilato in JavScript, sia come libreria. Le due idee che stanno alla base di questo paradigma di programmazione sono le *variabili a tempo continuo* e la *propagazione di eventi*. Il programmatore si deve preoccupare di scrivere il programma in termini di cosa fare, sarà poi l'architettura sottostante che deciderà il quando. Brevemente, per spiegare la semantica reattiva, illustriamo il seguente semplice esempio:

```
var x = 2;
var y = x * x;
```

In un tipico programma sequenziale, la variabile y avrà sempre il valore 4, indipendentemente dal fatto che successivamente la variabile x verrà aggiornata ad un nuovo valore. Al contrario, in un programma reattivo, la variabile y corrisponderà sempre al valore x elevata al quadrato; quindi se ad un certo punto della computazione il valore di x cambierà in 3, sarà

compito dell'architettura sottostante, e non del programmatore, aggiornare automaticamente il valore della variabile y a 9.

Con piccole variazioni sintattiche da linguaggio all'altro, due sono le principali astrazioni che caratterizzano questo paradigma:

- *Behaviours*: rappresentano le variabili a tempo continuo. Ad ogni istante della computazione, esse assumono un determinato valore, a seconda delle variabili da cui dipendono. Un esempio di Behaviour è il tempo stesso.

- *Events*: sono uno stream, potenzialmente infinito, di eventi che accadono a tempo discreto. Un esempio di Event può essere la pressione di un tasto.

Mostriamo quindi, mediante un esempio utilizzando Flapjax come linguaggio, una piccola applicazione che visualizza il tempo trascorso dall'avvio del programma o da quando viene premuto un pulsante:

```
var nowB = timerB(1000);
var startTm = nowB.valueNow();
var clickTmsB = $E("reset", "click").snapshotE(nowB)
.startsWith(startTm);
var elapsedB = nowB - clickTmsB;
insertValueB(elapsedB, "curTime", "innerHTML");

<body onload="loader()">
<input id="reset" type="button" value="Reset"/>
<div id="curTime"> </div>
</body>
```

timerB(1000) crea un Behavior che si aggiorna ogni 1000 millisecondi (quindi ogni secondo trascorso). Il metodo *valueNow* estrae un'immagine del valore corrente del Behavior *nowB*, ogni volta che viene invocato. *startTm* quindi non si aggiorna automaticamente ma solo quando viene richiamato il metodo *nowB.valueNow*. *$E* definisce un Events e genera un evento per ogni click del pulsante *reset*; ciò che restituisce *$E* è dunque uno stream di oggetti *DOM event*. Il metodo *snapshotE* trasforma lo stream di eventi del DOM in uno stream di valori del timer. *startsWith* converte lo

stream di eventi in un Behaviour, inizializzato con il valore *startTm*. Infine *insertValueB* rappresenta il metodo che inserisce e tiene aggiornato nel DOM, all'elemento *curTime*, il valore del behaviour *elapsedB*.

3.4 Modello a Event Loops Comunicanti

Fin dall'inizio, il modello ad eventi è stato introdotto come alternativa a quello multi-thread. Tutti i problemi causati dall'interazione concorrente di thread distinti su variabili condivise, vengono risolti alla radice nell'architettura ad eventi introducendo un unico flusso di controllo. Questa soluzione tuttavia non risulta soddisfacente per sfruttare le moderne architetture multi-core, perché di fatto non vi è alcun tipo di parallelismo nell'esecuzione. Com'è possibile dunque adattare il modello descritto ad eventi, prettamente a singolo thread, per poter sfruttare appieno i processori di ultima generazione e le architetture distribuite?

La soluzione è quella di estendere il concetto di un solo Event Loop a più Event Loops, ognuno con la propria Event Queue, con i propri Event Handlers e con il proprio flusso di controllo. L'unico modo che hanno di interagire questi Event Loops è tramite scambio di messaggi asincrono (figura 3.10). Per non ricadere nei problemi evidenziati della programmazione multi-thread, è tuttavia necessario formalizzare una serie di proprietà a cui il modello dovrà aderire:

1. *Esecuzione Seriale*: come già descritto nel modello generale, ogni Event Loop processa sequenzialmente gli eventi della propria Event Queue uno alla volta e gli Event Handlers associati vengono eseguiti fino alla loro terminazione, senza possibilità di interromperli in alcun modo. Questa proprietà dell'esecuzione delle operazioni in maniera atomica viene definita *Semantica Macro-Step*.

2. *Comunicazione Non-Bloccante*: tra Event Loops distinti, la comunicazione avviene tramite scambio di messaggi asincrono, i quali vengono trattati come eventi di notifica. Questa proprietà è necessaria per non bloccare un Event Loop in attesa di una risposta che non sa quando può arrivare.

3. *Accesso Esclusivo allo Stato*: ogni Event Loop dispone di un proprio set di dati, uno stato, a cui può accedere in maniera esclusiva. Non

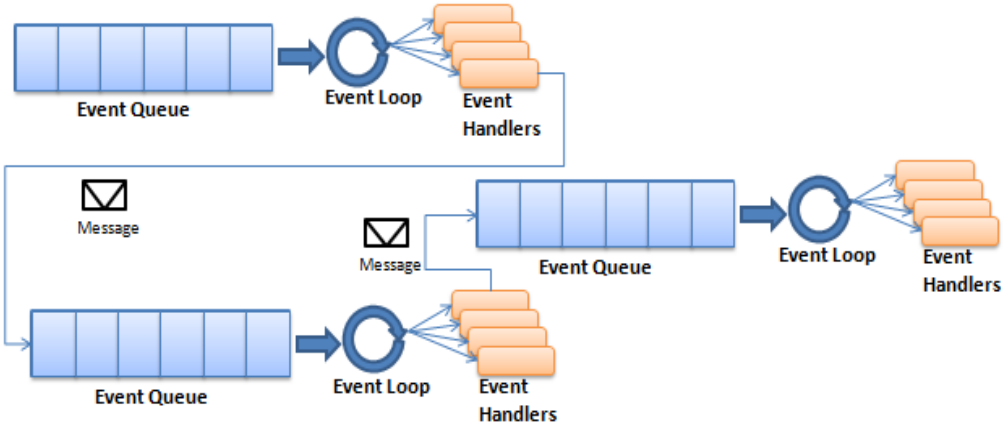

Figura 3.10: Comunicazione tramite scambio di messaggi fra Event Loops distinti

> vi è alcun modo per gli Event Loops di accedere allo stato di un altro Event Loop. L'unico modo di reperire informazioni sullo stato di un altro flusso di controllo è mediante lo scambio di messaggi.

Un architettura di questo tipo, impone il programmatore a pensare al programma come un insieme distinto di entità reattive, ognuna con un proprio stato esclusivo su cui può lavorare senza preoccuparsi di dover gestire l'accesso concorrente, che interagiscono tramite scambio di messaggi.

Le proprietà semantiche di questo modello ricordano molto da vicino quelle degli *Attori*, che verranno brevemente descritti nel paragrafo successivo.

3.4.1 Modello ad Attori

In questa parte si farà una breve introduzione al modello degli attori che, in un certo senso, rappresenta un modello intermediario tra quello ad eventi illustrato nelle sezioni precedenti (e che verrà approfondito nelle sue implementazioni più note, appartenenti al Web, dal capitolo 4) e quello ad agenti, che verrà introdotto nel capitolo 5.

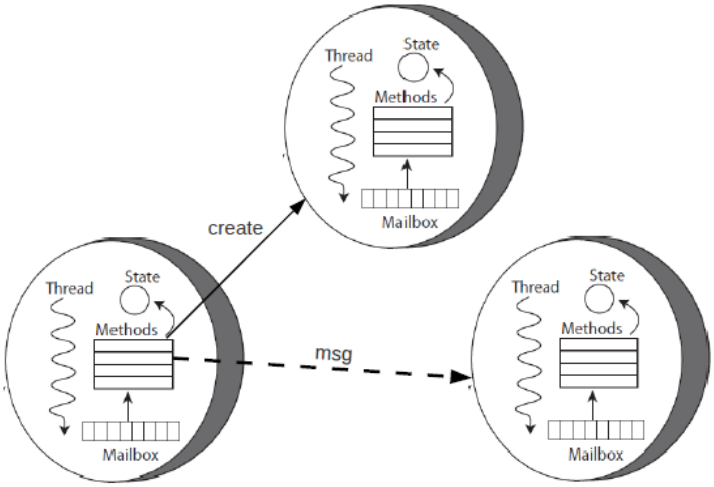

Figura 3.11: Rappresentazione di Attori, entità auto-contenute e concorrenti in grado di comunicare tramite scambio di messaggi

Il modello ad attori è molto simile, in tutto e per tutto, a quello ad Event Loops comunicanti. Si può dire che gli Attori implementino il modello ad eventi, utilizzando nomi distinti per identificare strutture analoghe.

Gli Attori sono entità autonome ed auto-contenute, che operano concorrentemente e con modalità asincrone scambia messaggi con gli altri Attori, ne crea di nuovi e aggiorna il proprio stato (figura 3.11). Un *Sistema ad Attori* consiste in una collezione di Attori, i quali possono interagire, inviando o ricevendo messaggi, con altri Attori situati all'interno o all'esterno del sistema in cui sono inseriti [12]. Ogni Attore dispone di una *Mailbox*, dove vengono accodati i messaggi, e di un proprio stato locale ad accesso esclusivo. Gli Attori hanno un comportamento puramente *reattivo*, rimangono in attesa di poter prelevare e processare un messaggio della propria Mailbox. Ad ogni tipologia di messaggio corrisponde l'esecuzione di un handler; questa esecuzione viene definita *atomica* poiché prima di poter processare il messaggio successivo, si deve terminare l'esecuzione l'handler corrente. Tre sono le principali tipologie di azioni:

- Inviare un nuovo messaggio;

- Creare un nuovo attore;
- Aggiornare il proprio stato locale.

Una rappresentazione in pseudo-codice del ciclo di vita di un Attore può essere la seguente:

```
while(true) {
  Message msg = mailbox.nextMessage();

  MessageHandler h = selectMessageHandler(msg);
  MessageArguments args = getMessageArguments(msg);

  executeMessageHandler(h, args);
}
```

Riprendendo il modello ad eventi descritto in 3.3 ed in 3.4, è possibile ritrovare tutti gli elementi caratteristici in quello ad Attori. L'Event Queue altro non è che la Mailbox, contenente gli Eventi nel primo caso o i Messaggi nel secondo caso. In entrambi i casi ci si blocca finché non c'è almeno un elemento della coda. Ad ogni elemento (messaggio o evento che sia) corrisponde un handler che viene eseguito fino al suo termine prima di processare l'elemento successivo (semantica macro-step).

Le tecnologie ad Attori sono diventate molto popolari, specialmente nello sviluppo di sistemi concorrenti/distribuiti/mobile, fornendo una valida alternativa nel mainstream al modello multi-thread.

Capitolo 4

Web Development oggi

Nell'introduzione e nel corso dei precedenti capitoli, si è fatto continuamente riferimento a JavScript. Questo linguaggio, assieme ad HTML, CSS e DOM, è una delle features principali di quello che è il nuovo standard HTML5.

JavScript ed il suo modello ad eventi hanno avuto un importanza sempre maggiore negli anni, grazie anche all'implementazione di librerie che ne hanno semplificato lo sviluppo e che di fatto sono diventate degli standard (su tutte jQuery). La sua popolarità non si è limitata solo al browser ma si è diffusa anche in altri ambiti; lato server ad esempio troviamo Node.js, piattaforma software scritta in JavScript progettata per massimizzare l'efficienza e limitare l'overhead dovuto alle operazioni di I/O. Tuttavia non si deve pensare a JavScript come unica possibilità per lo sviluppo di applicazioni web. Nonostante l'enorme diffusione del suddetto linguaggio, molte alternative si stanno presentando per lo sviluppo di Web Applications, che cercano di superare alcune delle lacune di JavScript fornendo al tempo stesso tool atti alla semplificazione dello sviluppo di programmi web. Tra le varie alternative, si citano TypeScript e Dart.

La prima è la proposta free ed open-source di Microsoft. Si tratta di un linguaggio di programmazione che estende la sintassi di base di JavScript, introducendo nuove features quali le annotazioni di tipo (grazie alle quali è possibile fare debug sui tipi a compile-time), le interfacce e le classi analoghe al paradigma object-oriented, il supporto ai moduli. La particolarità di TypeScript è che si tratta di un super-set di JavScript, quindi un programma scritto in JavScript è al tempo stesso valido per TypeScript. Questo rende possibile, con TypeScript, il riutilizzo di codice esistente JavScript,

riutilizzandone le librerie di interesse.

La seconda proposta è quella sviluppata da Google, con lo scopo di essere la nuova lingua franca del Web. Dart, al contrario di TypeScript, è un linguaggio totalmente a se stante, nonostante riprenda molte delle astrazioni diventate popolari in JavScript.

Si precisa che ad oggi, la maggior parte delle alternative proposte, se non proprio la totalità, presenta un compilatore da codice sorgente a codice JavScript, TypeScript e Dart inclusi.

4.1 JavScript

JavScript nacque con lo scopo di dare interattività e dinamicità alle pagine HTML. Nonostante il nome, non ha nulla a che fare con Java, l'unica similitudine è legata al fatto che entrambi adottano la sintassi del C; oltretutto il nome ufficiale è ECMAScript. Sviluppato inizialmente da Netscape, con il nome di LiveScript, ed introdotto in Netscape 2 nel 1995, JavScript è diventato standard ECMA (Eurioean Computer Manufactures Association) (ECMA-262) nel 1997 ed è anche uno standard ISO (ISO/IEC 16262).

JavScript è un linguaggio di scripting interpretato e non compilato, è object-based ma non class-based (esiste cioè il concetto di oggetto ma non di classe) ed è debolmente tipizzato. Le variabili infatti vengono semplicemente dichiarato usando la keyword *var*, non hanno un tipo. Al contrario, sono i dati ad averlo.

In JavScript esisono solo due categorie di tipi:

- **Tipi Primitivi**: numeri, booleani e stringhe;
- **Oggetti**:
  - **Oggetti generici**;
  - **Funzioni**;
  - **Array**;
  - **Espressioni regolari**;
  - **Oggetti predefiniti**: Date, Math, etc.;
  - **Oggetti wrapper**: String, Number, Boolean;

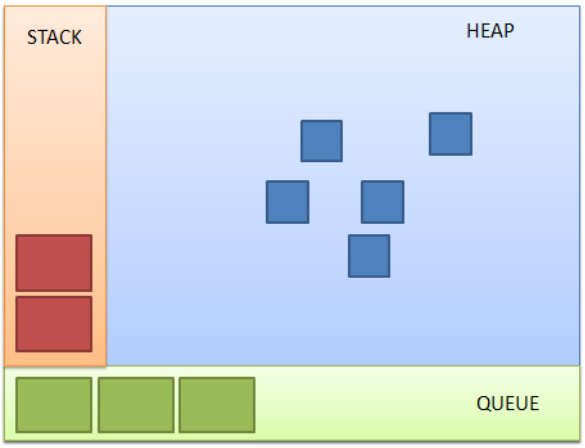

Figura 4.1: Javascript Event Loop Model

– **Oggetti definiti dall'utente**: mediante la definizione di un costruttore.

Notare come anche le funzioni vengono trattate come oggetti. Sono entità di prima classe che possono essere assegnate ad una variabile, passate come un argomento, ritornate da altre funzioni, e manipolate come tutti gli altri oggetti

4.1.1 Javascript Runtime Model

JavScript nasce come linguaggio single-thread dove task asincroni vengono gestiti tramite eventi. Il modello che ne deriva è quindi del tutto analogo a quello mostrato nel capitolo 3.3.1. Ovviamente, anche se a grandi linee funzionano tutti più o meno allo stesso modo, ognuno degli interpreti JavScript implementa una versione altamente ottimizzata dell'architettura descritta.

La struttura di base può essere rappresentata come in figura 4.1, con le tre strutture di base: la *Queue*, lo *Stack* e l'*Heap*.

Pensando ad un programma JavScript come una composizione di task, avendo un task X che deve eseguire un task Y, l'esecuzione di Y può essere schedulata in due modi:

1. *Immediatamente*: si interrompe X, si esegue Y e poi si prosegue con l'esecuzione di X.

2. *In Futuro*: si inserisce Y in una lista di "cose da fare in futuro", si finisce X e poi si procede con Y.

La prima opzione corrisponde sostanzialmente ad una chiamata a funzione. Prendendo come riferimento l'immagine 4.1, queste chiamate a funzione generano uno Stack di *Frames*. Se ad esempio si ha:

```
function f(b){
  var a = 12;
  return a+b+35;
}

function g(x){
  var m = 4;
  return f(m*x);
}

g(21);
```

quando la funzione g viene chiamata, genera un primo Frame contenente i suoi argomenti e le variabili locali. Quando g chiama f, un secondo Frame, contenente gli argomenti e le variabili locali di f, viene creato e posizionato in cima allo Stack (operazione di *push*). Quando f ritorna il valore, il Frame in cima allo Stack viene tolto (operazione di *pop*), lasciando solamente il Frame g e così via.

La seconda opzione invece corrisponde ad inserire l'operazione nella lista delle "cose da fare". Avendo sempre l'immagine come riferimento 4.1, significa inserire nella coda un nuovo oggetto. Questa coda non è altro che la Event Queue illustrata nel capitolo 3.3.1 e l'oggetto in questione è un evento. Prendendo come esempio il codice seguente:

```
for (var i = 1; i <= 3; i++) {
setTimeout(function(){ console.log(i); }, 0);
};
```

si ha che nella Event Queue vengono inseriti tre eventi uguali. L'Event Handler associato deve semplicemente eseguire la stampa sulla console del

valore della variabile i. Quando lo Stack è vuoto, si estrae il primo evento nella coda e si esegue la funzione associata (che genera un Frame nello Stack). Una volta terminato l'Event Handler e che quindi lo Stack torna ad essere vuoto, si estrae l'evento successivo e così via.

Si noti che, con questo modello di esecuzione, i valori della stampa non saranno $1, 2, 3$, bensì $4, 4, 4$. Questo perché, ad ogni chiamata della funzione *setTimeout*, un evento viene inserito nella coda (a cui si associa l'Event Handler *function() console.log(i);*), ma finché lo Stack non sarà vuoto, questi eventi non saranno processati. Terminato il ciclo *for*, la variabile i avrà raggiunto il valore 4 e solo allora verranno gestiti i tre eventi precedentemente inseriti, che stamperanno su console il valore di i nel momento che sono richiamati, quindi $4, 4, 4$.

L'Heap è la zona di memoria dove vengono allocati gli oggetti. Si ricorda che in JavScript praticamente ogni cosa è un oggetto, anche le funzioni.

In generale si dice che il modello dell'Event Loop di JavScript non è bloccante perché praticamente ogni cosa viene gestita con eventi e callbacks associate, anche le operazioni di I/O che notoriamente sono molto costose dal punto di vista del tempo richiesto.

4.1.2 JavScript Frameworks

JavScript non sarebbe quello che è senza il contributo dei numerosissimi frameworks e librerie che sono stati implementati col tempo. Addirittura alcuni pensano che JavScript sia diventato un linguaggio di tutto rispetto, che gli ha permesso di elevarsi dai principali rivali, con l'arrivo della libreria jQuery (2006) che ha definito uno standard de facto nella programmazione event-driven.

Ad oggi, ve ne sono un numero davvero elevato di queste estensioni (figura 4.2), progettate con differenti scopi e che portano a diversi benefici. L'approccio odierno per lo sviluppo di Web Applications è quello di combinare insieme tutte queste tecnologie e costruirci l'applicazione sfruttando le API e le facilities disponibili.

Nonostante il grande numero, è possibile suddividere l'ecosistema JavScript in due grandi famiglie:

1. Frameworks per la manipolazione del DOM;
2. Frameworks MVC.

Figura 4.2: Alcune delle librerie e frameworks di JavScript

Questi due gruppi rappresentano la quasi totalità dei frameworks lato client. Si menziona anche l'esistenza di una terza famiglia per lo sviluppo lato server, rappresentata da Node.js, un framework ideato nel 2009 da Ryan Dahl con lo scopo di creare velocemente Web servers scalabili, utilizzando il modello event-driven con operazioni di I/O non bloccanti.

La prima famiglia è sicuramente rappresentata da jQuery. Con il motto "scrivi di meno, fai di più"e con una libreria contenente API di alto livello, studiate per semplificare la navigazione dei documenti, la selezione degli elementi DOM, creare animazioni, gestire eventi e implementare funzionalità AJAX, ha raggiunto un livello di utilizzo davvero impressionante (si parla di oltre il 55% fra i 10000 siti più visti del Web).

La seconda famiglia può essere rappresentata da Backbone.js, una libreria con un interfaccia RESTful JSON e basata su di una variante del pattern di progettazione MVC, il *Model-View-Presenter* (o *MVP*), progettata per applicazioni Web a singola pagina e per mantenere le varie parti delle applicazioni web sincronizzate. Questi framework nascono dal fatto che spesso, nel codice JavScript, si mescola la business logic con la parte riguardante l'interfaccia grafica, andando a violare pattern quali *separation of concerns* e, appunto, il *MVC*. Appartenente a questa famiglia, un nuovo framework sta acquisendo di popolarità in questi ultimi anni, ovvero Angular.js.

4.1.3 JavScript e la Concorrenza: Web Workers

Come già accennato precedentemente (capitolo 3.4), l'idea alla base per poter sfruttare i nuovi processori multi-core senza abbandonare un modello

ad eventi è quello di sfruttare distinti event-loops che comunicano mediante scambio di messaggi.

In JavScript, il modello a event-loops comunicanti è realizzato con quelli che vengono definiti i *Web Workers*, i quali fanno parte anche dello standard HTML5 (ma com'è già stato detto, HTML5 e JavScript sono strettamente correlati). Una definizione di questi può essere: *JavScript eseguiti in background senza compromettere le performance della pagina*[10].

Fondamentalmente si trattano di script eseguiti in background, gli uni indipendenti dagli altri, su thread separati, così che l'utente può continuare ad interagire con la pagina senza che questa ne risenti. I Workers sono relativamente pesanti come processi, e non sono pensati per essere usati in grande numero. Generalmente, ci si aspetta che siano long-lived e che abbiano un costo molto alto sia di inizializzazione in termini di performance che di memoria per singola istanza [11]. L'idea è ovviamente quella di poter sfruttare le oramai diffusissime CPU multi-core, suddividendo ed assegnando task computazionalmente costosi a più Workers, così da ottenere migliori performance.

Vi sono due tipologie di Web Workers: *Dedicated Workers* e *Shared Workers*. La prima indica un Worker che può essere indirizzato solamente dallo script che l'ha creato, la seconda invece rappresenta un Worker che può essere indirizzato da qualsiasi script appartenente allo stesso dominio in cui è stato creato. Al di la della tipologia, l'unico modo che i Workers hanno di interagire è tramite scambio di messaggi asincrono il cui arrivo genera un evento che viene accodato nella Event Queue del Worker che l'ha ricevuto.

L'accesso al DOM rimane comunque esclusivo del main script. Se si rendesse possibile la modifica del DOM da parte di tutti i Workers, si reintrodurrebbero tutte le problematiche legate al modello multi-thread, dovendo controllare tutti gli elementi del DOM con Semafori e Monitor, per evitare il problema delle corse critiche (capitolo 3.1).

4.2 Dart

4.2.1 Le Ragioni di un Nuovo Linguaggio

Dart (in origine, Dash), nelle intenzioni originali scritte in una mail interna trapelata da Google [13], nasce con l'idea di "sostituire JavScript come

lingua franca per lo sviluppo web sulla piattaforma Web aperto". Nell'analisi fatta all'interno della suddetta e-mail, si evidenzia come JavScript abbia dei difetti alle fondamenta, che difficilmente possono essere risolti semplicemente evolvendo il linguaggio.

Due sono le strategie di base che possono essere adottate per il futuro di JavScript:

- **Strategia Basso Rischio/Basso Guadagno**: lavorare per evolvere le parti standard di JavScript;
- **Strategia Alto Rischio/Alto Guadagno**: sviluppare un nuovo linguaggio che punta a mantenere la natura dinamica di JavScript ma con un miglior profilo di performance e con il supporto adeguato per lo sviluppo di grandi progetti.

La prima delle due strategie è molto conservativa e rischia di non risolvere problemi che stanno alle basi fondanti del linguaggio, come l'utilizzo di un unico tipo primitivo Number per identificare i valori numerici.

La seconda invece è la tipica *strategia del salto della rana* (*leap frog strategy*): riprogettare le fondamenta di JavScript, nate in un periodo storico del Web differente e con diversi requisiti/problematiche da tener conto.

È comunque importante identificare tutto ciò che c'è di buono nello sviluppo odierno in ambito di Web Applications:

- JavScript è flessibile e supporta uno sviluppo incrementale;
- Sviluppare e deployare piccole applicazioni è piuttosto facile e rapido, e non richiede alcun tipo di installazione, grazie anche al *browser* che di fatto è una piattaforma di esecuzione indipendente.

Mantenere le suddette caratteristiche è necessario, risolvendo al tempo stesso alcune delle problematiche più note:

- Lo sviluppo di applicazioni in larga scala è molto difficile, per motivi già evidenziati, riguardanti la struttura del programma su cui è complesso ragionarci;
- È difficile documentare gli intenti e le interfacce poiché in JavScript non esistono tipi statici a livello di variabili;

- Nessun supporto per moduli, packages o librerie;
- Tool di supporto non sufficientemente adeguati.

Dart nasce quindi con alcuni, ben specifici obbiettivi in mente:

- Raggiungere determinati livelli di **Performance**, creando una nuova *VM* (*Virtual Machine*) che non sia afflitta da tutti i problemi di performance che le EscmaScript *VM* devono avere;
- Mantenere una certa **Usabilità** del linguaggio, preservando le caratteristiche quali la dinamicità, la praticità e la natura di linguaggio non compilato di JavScript;
- La possibilità di essere **Più Facilmente Mantenibile**, introducendo ad esempio i *Tipi Opzionali* per le variabili, col fine di semplificare lo sviluppo di progetti in larga scala che richiedono caratteristiche di comprensione del codice.

4.2.2 Il Linguaggio

Presentato per la prima volta alla conferenza GOTO dell'ottobre 2011, Dart è un linguaggio strutturato di programmazione per il web, con una sintassi piuttosto familiare ad altri linguaggi come Java, con alcune caratteristiche fondamentali:

- È un linguaggio di programmazione object-oriented;
- È basato su classi a singola derivazione con interfacce;
- Tipi opzionali statici;
- *Lexical Scope* (o *Static Scope*, poiché lo scope di una funzione è deducibile staticamente a compile-time);

Dart è un linguaggio piuttosto giovane, nonostante ciò è in continua crescita e presenta già delle solide basi nonché il supporto ad alcuni dei meccanismi più noti come:

- Cascading;

- Closoures;
- Mixins;
- Promises (quì chiamate Futures);
- etc.

Vi sono tuttavia ancora questioni aperte o non ben definite, una su tutte la gestione degli errori, su cui Google sta ancora lavorando.

Nell'ottica dello scopo di questo lavoro, ci si soffermerà sul ciclo di controllo adottato in Dart e sui costrutti supportati per ciò che riguarda l'aspetto asincrono e concorrente del linguaggio.

4.2.3 Dart Event Loop

Dart presenta codice asincrono in moltissime parti del suo linguaggio. Approfondire dunque l'architettura ed il ciclo di controllo adottato è fondamentale per capirne il funzionamento dei programmi. Avendo familiarizzato già con concetti quali Event Loop ed Event Queue, il modello adottato da Dart sarà piuttosto intuitivo, nonostante qualche piccola differenza. La prima riguarda il fatto che non vi è più una singola coda di eventi, bensì due:

1. *Event Queue*: la tipica coda contenente ogni tipo di evento come I/O, mouse, update della grafica, timer, messaggi etc.
2. *MicroTask Queue*: questa coda contiene tutte quelle richieste di operazioni che si dovranno eseguire in futuro ma prima di gestire qualsiasi evento nell'Event Queue.

Un'applicazione Dart inizia con l'eseguire prima tutte le funzioni contenute nel *main()*; una volta terminate, si procede con l'esecuzione dell'Event Loop, che può essere descritto in pseudo-codice come segue:

```
while(true) {
 if(microtaskQueue.HasElement())
 {
  MicroTask mt = microtaskQueue.next();
  execute(mt);
```

```
    continue;
  }
  else if(eventQueue.hasElement())
  {
    Event ev = eventQueue.next();
    handle(ev);

    continue;
  }
  else
  {
    //The application can exit
    return;
  }
}
```

L'architettura adottata delinea dunque un concetto di priorità fra eventi; l'Event Loop gestisce prima tutte le richieste della Microtask Queue e solo una volta che quest'ultima coda è vuota si passa a processare quelle dell'Event Queue. Questa scelta tuttavia presenta alcuni rischi e per questo motivo vengono fornite delle best practice da seguire.

La prima è quella di tenere la coda dei Microtask più breve possibile per evitare di stallare per troppo tempo la Event Queue, rendendo così l'applicazione non reattiva. Ad esempio, il semplice click del mouse non verrà gestito fintanto che vi sono degli elementi/richieste nella MicroTask Queue (figura 4.4).

Questa situazione è analoga a quella descritta in figura 3.9 dove se non si dimensionano correttamente gli Event Handlers si rischia di avere un applicazione poco reattiva.

4.2.4 Dart e la Programmazione Asincrona: Futures e Streams

Anche Dart, come JavScript, è un linguaggio di programmazione single-thread e fa largo utilizzo di codice asincrono per evitare di bloccare il ciclo di esecuzione in operazioni time-consuming. Si è già visto come con la

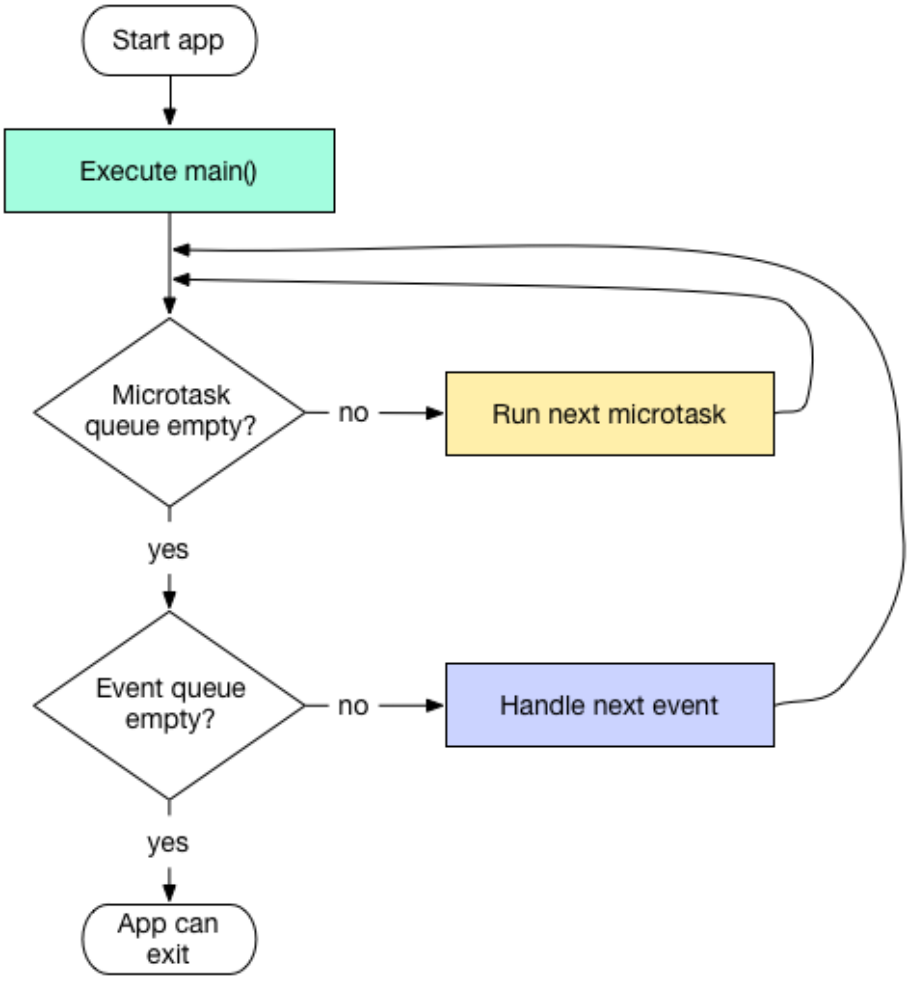

Figura 4.3: Ciclo di vita di una tipica applicazione Dart

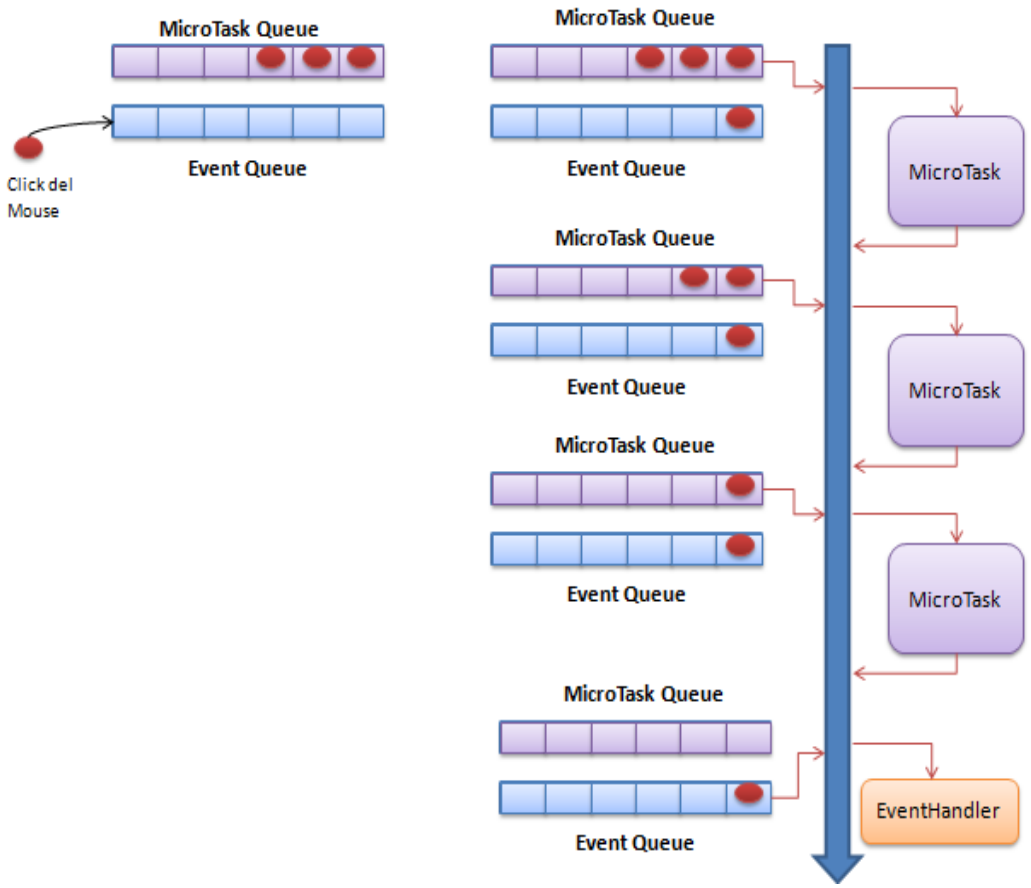

Figura 4.4: Situazione dove la MicroTask Queue ritarda la gestione del mouse

programmazione a callbacks, è molto facile produrre codice non leggibile, specialmente al crescere della complessità dell'applicazione.

Dart adotta due tipologie base di costrutti, per la gestione di operazioni asincrone:

- **Futures**: risultato di una singola operazione asincrona;
- **Streams**: una sequenza asincrona di elementi.

Futures

Le *Futures* in Dart, altro non sono che le analoghe Promises di JavScript. Sono entità di prima classe che rappresentano il risultato di una computazione che verrà completata in futuro, restituendo immediatamente un oggetto Future. Nello specifico, due sono le operazioni che avvengo appena viene chiamata una funzione che restituisce questo tipo di oggetti:

1. Viene accodato del lavoro da fare (tipicamente, un operazione time-consuming) e si ritorna immediatamente un oggetto Future incompleto.
2. Quando il lavoro verrà completato, l'oggetto Future avrà un valore definito, se l'operazione ha avuto successo, o restituirà un errore.

Anche con le Futures è possibile concatenare operazioni asincrone la cui esecuzione della successiva dipende dal risultato della precedente, utilizzando il metodo *.then()*:

```
expensiveA().then((aValue) => expensiveB())
            .then((bValue) => expensiveC())
            .then((cValue) => doSomethingWith(cValue))
               ;
```

Se l'oggetto Future completa con un errore, è possibile "catturarlo" utilizzando un costrutto del tutto simile a quello del try-catch:

```
File file = new File("myFileToRead.txt");
Future future = file.readAsString();
future.then((content) => doSomethingWith(content))
      .catchError((e) => handleError(e));
```

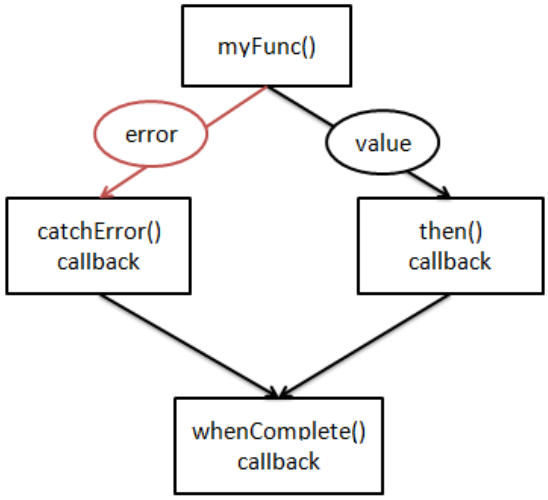

Figura 4.5: Flusso di esecuzione di una funzione che restituisce una *Future*

Se il *then().catchError()* è quindi l'analogo del try-catch tradizionale, il *whenComplete()* è l'equivalente del finally; serve per registrare un metodo che viene eseguito in qualsiasi caso, che sia stato restituito un valore o un errore (figura 4.5).

```
var server = connectToServer();
server.post(myUrl, fields: {"name": "Enrico", "
    profession": "student"})
        .then(handleResponse)
        .catchError(handleError)
        .whenComplete(server.close);
```

Per completezza, mostriamo infine l'ultimo costrutto che può essere definito un ponte fra callback-based API con Future-based API, *Completer*. Un esempio più essere un operazione che legge da un database che non fa uso delle Futures, ma nell'applicazione si vorrebbe comunque ritornare un valore Future:

```
Future doStuff() {
    Completer completer = new Completer();
    runDatabaseQuery(sql, (results) {
```

```
      completer.complete(results);
    });
    return completer.future;
  }
```

Streams

In Dart, come in altri linguaggi, è possibile ritrovare elementi tipici della programmazione reattiva-funzionale.

Citando Meijer[23], *Il Web e le applicazioni mobile si stanno incrementalmente componendo di servizi streaming, asincroni o realtime, e di notifiche push, una forma particalare di big data dove i dati hanno una certa velocità*. Il mouse, ad esempio, è una sorgente continua di eventi legati al movimento o alla pressione di uno dei tasti.

Dart riprende quindi il concetto di Events del paradigma reattivo (3.3.3), definendo il costrutto *Streams*.

Uno Stream può essere tutto ciò che invia una sequenza di dati tramite *Push*. La distinzione tra unoStream ed un *Iterable* di Dart è che in quest'ultimo il recupero dei dati avviene mediante una *Pull*.

```
// Iterables are pulled.
var
iterator = [1, 2, 3].iterator;
while(iterator.moveNext()) {
        print(iterator.current);
}

// Streams push.
var stream = element.onClick;
stream.listen(print);
```

Come per le Futures, anche con gli Streams è possibile "catturare"e gestire gli errori, oltre che una sono dotati di una funzione analoga a *whenComplete* quando lo Stream termina o viene rilevato un errore.

```
// setup the handlers through the subscription's
   handler methods
var subscription = stream.listen(null);
```

```
subscription.onData((value) => print("listen:$value"))
    ;
subscription.onError((err) => print("error:$err"));
subscription.onDone(() => print("done"));
```

Il metodo *listen()* ritorna un oggetto *StreamSubscription* con il quale è possibile smettere di ricevere gli eventi dello Stream.

```
var subscription = stream.listen(null);
subscription.onData((value) {
  print("listen:$value");
  if (value == 2) subscription.cancel(); // cancel the
      subscription
});
subscription.onError((err) => print("error:$err"));
subscription.onDone(() => print("done"));
```

Ad esempio, come si era visto per Flapjax nel capitolo sulla programmazione reattiva (3.3.3), anche in Dart si può modellare la pressione di un tasto come Stream di eventi:

```
var button = new ButtonElement();
button.onClick.listen((mouseEvent) {
    print("clicked"); // remain subscribed for all
        clicks
  });
```

onStream è definito come uno *Stream* di *MouseEvents*.

In Dart esistono due tipologie di Streams che presentano similarità ma anche importanti differenze:

1. *Single Subscription*, ovvero gli Stream a singola sottoscrizione.
2. *Multiple Subscription* (o *Broadcast*), ovvero gli Stream a multipla sottoscrizione.

In un certo senso, si può effettuare un paragone fra le due tipologie con il TCP e l'UDP, dove nel primo caso lo Stream è stabile e presenta delle proprietà di garanzia (come il TCP) mentre nel secondo caso gli eventi

possono essere persi e non c'è uno stretto accoppiamento fra l'ascoltatore e la sorgente (come nell'UDP).

	Single Subscription	**Broadcast**
Numero di ascoltatori	1	∞
Possibilità di Perdere Eventi	No	Sì
Ciclo di Vita Definito	Sì	No
Facilità d'Uso	Difficile	Facile

Il *Numero di ascoltatori* indica quanti sottoscrittori possono mettersi in ascolto: uno solo nel caso dei Single Subscription Stream mentre non vi sono limitazioni a tal proposito per quanto riguarda i Broadcast Stream.

Per quanto riguarda la *Possibilità di Perdere Eventi*, i Single Subscription Stream non perdono mai eventi ma al più vengono bufferizzati finché non vi sarà un ascoltatore. Al contrario nei Broadcast Stream è possibile che gli eventi vengano persi se non vi è nessuno in ascolto.

Per *Ciclo di Vita Definito* s'intende che i Single Subscription Stream "nascono"quando qualcuno si mette in ascolto e terminano quando si cancella o se lo Stream genera l'evento di chiusura. I Broadcast Stream invece non vengono influenzati dalle azioni lato ascoltatore.

Facilità d'Uso nel senso che, avendo i Single Subscription Stream un unico ascoltatore, l'invocazione di metodi tipo *first* possono consumare lo stream, cosa che non succede nei Broadcast Stream dov'è possibile invocare i suddetti metodi più di una volta.

In entrambe le tipologie di Stream (a singola o multipla sottoscrizione), per potersi mettere in ascolto degli eventi si usa il metodo *listen* la cui implementazione è, nella realtà, l'unica che differisce dal tipo di Single Subscription Stream.

Il metodo serve per connettere la sorgente ai suoi ascoltatori, creando un nuovo oggetto *Stream Subscription*, il quale viene connesso alla sorgente degli eventi ed inizializza le callbacks (se date) nella sottoscrizione.

Una volta creato lo Stream Subscription, lo Stream non ha più bisogno di tenere traccia degli ascoltatori.

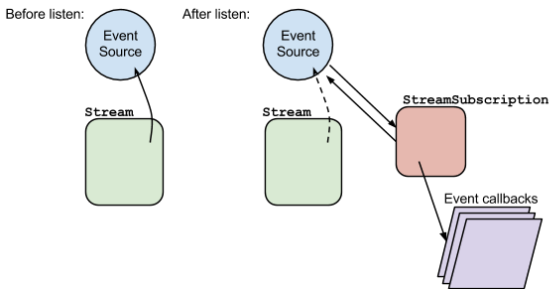

Figura 4.6: Prima e dopo la chiamata del metodo *listen*, nel caso di *Single-Subscription Stream*

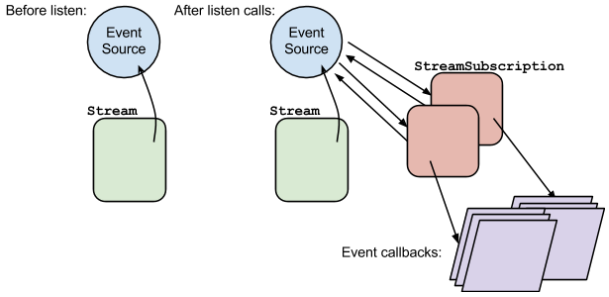

Figura 4.7: Prima e dopo la chiamata del metodo *listen*, nel caso di *Broadcast Streams*

L'esempio del funzionamento della chiamata *listen* è mostrato in figura 4.6 per quanto riguarda i Single Subscription Stream; la freccia tratteggiata indica proprio che, dopo la chiamata del metodo, lo Stream non ha più bisogno di tenere il riferimento alla sorgente degli eventi poiché se ne occuperà lo Stream Subscription. Inoltre, un'altra conseguenza è che lo Stream non avrà più alcuna informazione sullo stato della sorgente degli eventi.

Ciò non avviene per quanto riguarda i Broadcast Stream; il fatto di poter avere più ascoltatori significa che possono esserci più sottoscrizioni, lo Stream dovrà quindi mantenere il riferimento alla sorgente di eventi, come mostrato in figura 4.7.

L'uso di quest'ultima tipologia di Stream è quella per sorgenti di eventi che producono un output indipendentemente se ci sono ascoltatori o meno,

e perdere qualche evento non è un problema. Ad esempio, in Dart tutti gli elementi del DOM che producono eventi sono Broadcast Stream.

4.2.5 Dart e la Concorrenza: Isolates e Workers

Come ripetuto più volte, l'aspetto della concorrenza giocherà sempre più un ruolo fondamentale nello sviluppo di Web Applications. Lo standard HTML5 ha inserito fra le sue specifiche i Web Workers, seguendo il modello descritto precedentemente (capitolo 4.1.3).

Analogamente anche Dart fornisce il supporto alla concorrenza tramite gli *Isolates* o i *Workers*. C'è da precisare che ad oggi, le specifiche non sono molto chiare, vista la natura ancora incompleta del linguaggio [25]. Come specificato nei docs di Google, nella versione 1.0, le Web Applications non supportano gli Isolates e i Workers del linguaggio Dart. Tuttavia vi è una classe Worker che permette di aggiunge ad un applicazione Dart, un Web Worker JavScript.

Nelle API reference, un Isolate è definito come un Worker (di Dart) indipendente, simile ai threads ma che non condivide memoria ed interagisce solo tramite scambio di messaggi. Per poter comunicare, gli Isolates sfruttando due costrutti: *ReceivePort* e *SendPort*. Il primo è nei fatti, uno Stream, che genera un evento ogniqualvolta riceve un messaggio. Il secondo invece ha la funzionalità duale di poter inviare messaggi, utilizzando come destinatario l'id della rispettiva ReceivePort. La definizione di un Isolate è dunque analoga a quella dei Web Workers JavScript; il fatto che ancora non siano definiti i *Workers* del linguaggio Dart, genera solo confusione.

Web Workers, Isolates e Workers sono tutte astrazioni che implementano il modello a event-loops comunicanti e perciò risultano analoghe agli Attori descritti precedentemente (capitolo 3.4.1).

4.3 Web Development nel Prossimo Domani

Che le applicazioni Web stiano crescendo per dimensioni e complessità, è oramai un dato di fatto. Aspetti legati alle performance, gestione e leggibilità del codice, manutenzione ed estendibilità del software sono diventati oramai centrali, e la comunità si sta muovendo a tal proposito.

Google con il suo linguaggio cerca di proporre una valida alternativa, nonostante ancora non è in grado di slegarsi completamente dagli standard

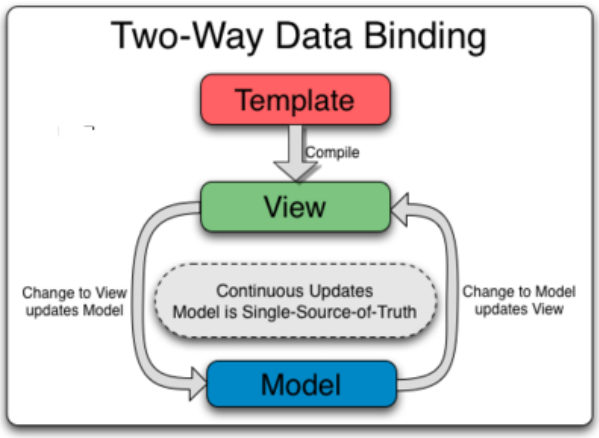

Figura 4.8: Two-Way Data Binding fra il Model e la View

attuali. Dart infatti presenta la possibilità di compilare il codice sorgente in codice JavScript, eseguibile quindi su tutti i comuni browser contenenti una JavScript VMs.

Le due proposte ora vanno di pari passo, proponendo nuovi framework e librerie sia per una versione, sia per l'altra. Su tutti, abbiamo *Angular*, un framework MVC (*Model-View-Controller*) per la costruzione di Web Applications, e *Polymer*, una libreria per la creazione di Web Components.

Lo scopo di questi tool è quello di fornire astrazioni sempre più di alto livello, in grado di ridurre l'abstraction gap fra il problema e le tecnologie disponibili. Alcune delle features distintive sono, per quanto riguarda Angular, le API di alto livello per il routing, comunicazione con server, unit testing, dependency injection e altro ancora; per Polymer, la caratteristica principale riguarda le opzioni per la costruzione di elementi personalizzabili, con grande livello di espressività e riutilizzabilità. Singolare come entrambi i tools abbiano introdotto il *two-way data binding*, la possibilità di collegare un dato rappresentato nel Model con il dato nella View; in questo modo la modifica in una delle due parti del dato viene propagata verso l'altra e vice versa (figura 4.8).

L'esecuzione di debug su codice asincrono è un'altra delle sfide nel prossimo futuro. Al contrario dei linguaggi procedurali, dove tipicamente abbiamo una *call stack* che definisce esattamente la sequenza delle chiamate

che hanno portato all'errore, nei linguaggi asincroni ad eventi non abbiamo niente di tutto ciò. Usualmente un errore viene visto, ma è difficile capirne l'origine.

Dart propone un costrutto chiamato *Zones*, che prende ispirazione dall'omonimo per JavScript, zone.js. In poche parole, una Zone è un area di memorizzazione locale del thread per la JavScript VMs [24]. Grazie a questi *contesti di esecuzione* che incorporano tutte le operazioni asincrone, è possibile ripercorrere all'indietro lo Stack per capire l'origine dell'errore.

Come ultimo, Dart sicuramente dovrà dare una definizione formale per quanto riguarda i suoi Isolates e Workers, vista la necessità di avere astrazioni per sfruttare le CPU multi-core.

Capitolo 5

Dal Modello ad Attori e ad Eventi al Modello ad Agenti

Fino ad ora si è parlato del modello generale ad eventi 3.3 e di due linguaggi di programmazione basati sull'architettura descritta: JavScript, per l'importanza che ha nel contesto attuale del Web, e Dart, per essersi proposto come il nuovo riferimento per lo sviluppo di Web Applications 4.

Si è mostrato come il paradigma ad eventi sia particolarmente adatto per applicazioni con un elevato grado di interattività e come sia possibile estenderlo per poter sfruttare le architetture multi-core. Tuttavia si è visto anche come questo modello possa presentare diversi problemi quando si tratta di dover effettuare lunghe computazioni che rischiano di bloccare l'Event Loop rendendo l'applicazione poco reattiva. Si cerca quindi di limitare questo problema spezzando il task in più sotto-task. Questa frammentazione del codice, oltre a generare *Spaghetti Code*, rischia di causare corse critiche a livello di eventi; infatti, durante l'esecuzione di un task possono frapporsi Event Handlers che vanno a modificare variabili rendendo il risultato finale del task inconsistente.

Prendendo come riferimento d'ora in avanti Dart, il problema di mantenere l'applicazione reattiva mentre si esegue un certo tipo di computazione potrebbe essere risolto delegando il task ad un Isolate distinto mentre il main Isolate si preoccupa di mantenere reattiva l'applicazione; una volta terminato il compito assegnatogli, l'Isolate di supporto invierà il risultato al main Isolate che deciderà cosa farsene. L'utente però potrebbe essere interessato, ad esempio, al livello di completamento del task raggiunto dal-

l'applicazione. Possiamo quindi fare in modo che l'Isolate di supporto invii ad intervalli regolari lo stato di avanzamento della computazione. Sarebbe comodo tuttavia che questa informazione possa essere reperita a comando, mediante la pressione di tasto. Per mantenere l'Isolate di supporto reattivo si dovrà comunque spezzare la computazione in sotto-task concatenati gli uni con gli altri, ritornando inevitabilmente al problema del codice frammentato. Ciò che risulta chiaro da questo esempio è che **combinare reattività e proattività, in maniera efficace ed efficiente in un programma con un modello ad eventi, è ancora oggi un problema!**.

In letteratura, quando si parla di entità autonome orientate allo svolgimento di un task (*proattività*) mantenendo comunque un certo grado di interattività con l'ambiente in cui sono inserite, adattando lo svolgimento in base agli stimoli ricevuti dall'esterno (*reattività*), spesso ci si riferisce agli *Agenti*. Introdotti in ambito di intelligenza artificiale, molti nuovi linguaggi agent-oriented sono stati proposti di recente in ambito general-purpose, fornendo alcuni spunti interessanti per nuove soluzioni ai problemi riguardanti la programmazione asincrona e concorrente.

5.0.1 Introduzione agli Agenti

Per meglio intendere cosa si intende con il termine di 'agente', si effettuerà un parallelo con gli altri paradigmi di programmazione.

Abbiano un primo tipo di programma, quello *funzionale*, considerabile il più semplice, dove il software prende in ingresso tutta una serie di dati in input, li elabora e produce un set di outputs. Il compilatore è un esempio di questo tipo di programmi. Il nome è così detto perché, matematicamente parlando, è possibile vedere il programma come una funzione $f : I \longrightarrow O$ da un certo dominio I di possibili inputs ad un certo dominio O di possibili outputs. Questi tipi di programmi sono molto lineari e concettualmente semplici dal punto di vista della risoluzione e programmazione. La realtà, purtroppo, è che questo tipo di software non sempre è abbastanza. Per quanto riguarda la costruzione di molti sistemi software, la struttura operativa di input-calcolo-output non è sufficiente e nella pratica si necessitano caratteristiche definibili reattive, nel senso che devono mantenere una interazione continuata e duratura con l'ambiente in cui sono inseriti. Sistemi Operativi, Web Servers e Sistemi di Controllo di Processi sono solo alcuni esempi appartenenti a questa categoria. Questi sistemi sono molto più arti-

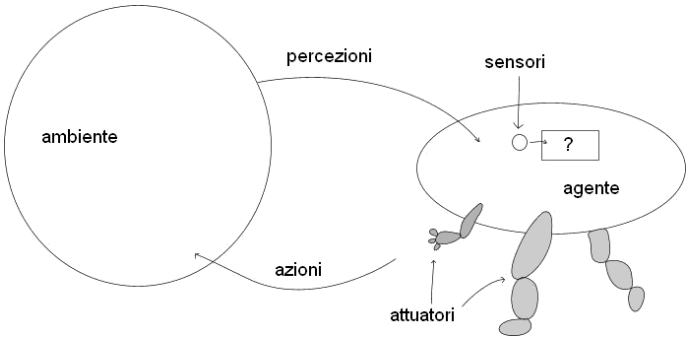

Figura 5.1: Una rappresentazione di un agente

colati rispetto a quelli funzionali, tuttavia esiste una terza categoria, ancora più complessa rispetto a quelle precedenti. Questa nuova tipologia di sistemi, che è un subset di quelli reattivi, verrà chiamata "ad agenti"poiché è possibile pensare a questi agenti come entità autonome ed attive, che agiscono di loro iniziativa poiché programmati al fine di raggiungere dei Goals (obbiettivi), ragionando per conto proprio su come farlo nel migliore dei modi.

5.0.2 Caratteristiche di un Agente

Diamo dunque una descrizione formale di quali devono essere le caratteristiche di un agente. Consideriamo gli agenti come sistemi (o parte di essi) situati in un qualche tipo di *Environment* (o ambiente). Una prima proprietà è quella di poter *percepire* l'ambiente circostante con l'ausilio di un certo tipo di *sensori*. Un ulteriore proprietà è che devono essere in grado di eseguire un repertorio di possibili *azioni*, con l'ausilio di *attuatori*, in grado di modificare l'ambiente circostante. La difficoltà e il punto cruciale per questo tipo di entità è *cosa decidere di fare in base a ciò che si percepisce dall'ambiente*. A parte il fatto di essere situati in un Environment, gli agenti dovrebbero mostrare le seguenti caratteristiche [27]:

- Autonomia;
- Proattività;
- Reattività;

- Abilità Sociali.

Per quanto riguarda l'*Autonomia*, possiamo definirla come la capacità di un agente di operare indipendentemente al fine di raggiungere uno scopo a lui affidato. Un agente autonomo è quindi in grado di prendere decisioni, in maniera indipendente, su come realizzare un obbiettivo: le sue decisioni e le conseguenti azioni sono sotto il suo controllo e non guidate da altri.

La *Proattività* significa l'abilità di mostrare un comportamento con la finalità di raggiungere uno o più Goals. Da un agente a cui è stato delegato un compito, ci si aspetta che provi in tutti i modi ad eseguire quel determinato compito. Un agente proattivo è l'opposto di un agente passivo, il quale agisce sempre e solo in risposta agli stimoli esterni.

La *Reattività* è la capacità di reagire, in un qualche modo, agli stimoli derivanti dall'esterno. Quest'ultima proprietà è necessaria in un agente poiché qualcosa potrebbe accadere durante l'esecuzione di azioni atte a raggiungere un obbiettivo. Un cambiamento nell'ambiente circostante può rendere determinate azioni migliori di altre, o addirittura può far diventare un Goal irraggiungibile; essere in grado di sapersi adattare, cambiando il piano in corso d'opera, è quindi una caratteristica fondamentale per gli agenti. È difficile tuttavia riuscire a bilanciare in maniera efficiente, entità con un comportamento *proattivo* e *reattivo*.

L'*Abilità Sociale* è ciò che concerne la capacità di questi agenti di poter *cooperare* e *coordinare* le loro attività al fine di raggiungere i propri obbiettivi. Questi tipi di interazione avvengono tramite una qualche sorta di comunicazione, *indiretta* se mediate da azioni atte a cambiare l'ambiente percepito da entrambi, *diretta* se gli agenti sono in grado di comunicare tra loro direttamente. Quest'ultima proprietà è necessaria per quanto riguarda i sistemi che andremo a considerare. Infatti, è difficile pensare a sistemi complessi che comprendono un solo agente. Al contrario, è più plausibile avere molti agenti, ad ognuno dei quali assegnato un task, che cooperano ed interagiscono tra di loro. Questa tipologia viene comunemente chiamata *Multi-Agent Systems* (*Sistemi Multi-Agente* o *MAS*).

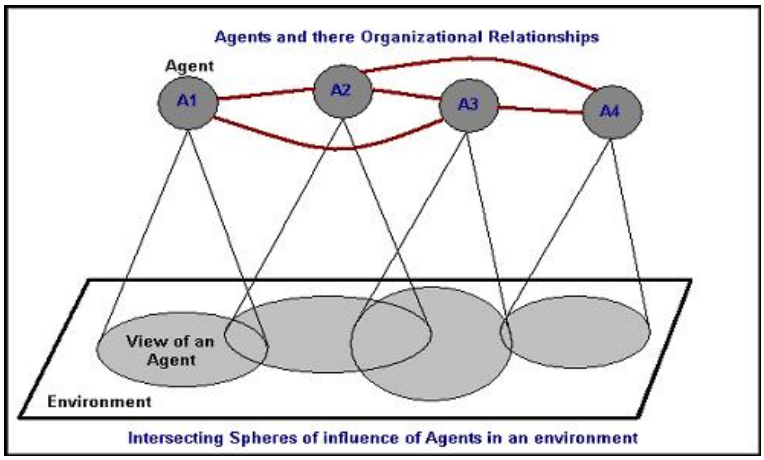

Figura 5.2: Rappresentazione di un Sistema Multi-Agente

5.0.3 Sistemi Multi-Agente

Un sistema multi-agente può essere rappresentato come in figura 5.2. Nella parte più in basso si ha l'Environment, l'ambiente condiviso dagli agenti che lo popolano. Ogni agente ha una 'sfera di influenza', ovvero una porzione dell'ambiente che è in grado di controllare totalmente o in parte. Nonostante vi sia la possibilità che un agente abbia il totale controllo su di una parte dell'Environment, è più probabile queste sfere di influenze si sovrappongono. Questo significa che certe sotto-porzioni saranno controllate congiuntamente da più agenti. Per raggiungere i loro Goals, devono quindi monitorare continuativamente la porzione dell'ambiente di loro competenza poiché le modifiche all'Environment possono risultare in un cambio di piani e azioni da eseguire.

Gli agenti possono inoltre essere situati in diverse categorie di *organizzazioni* che li mette in un certo tipo di relazione gli uni con gli altri. L'interazione fra questi può avvenire, come già detto, in maniera indiretta o diretta. Per poter comunicare direttamente, è naturale pensare che l'agente comunicante debba avere un qualche tipo di informazione o conoscenza verso colui che vuole comunicare. In ogni caso non è necessario che gli agenti abbiano una totale conoscenza delle altre entità presenti nel sistema.

5.0.4 Modello di Agenti BDI

Fino ad ora si è parlato delle caratteristiche generali che dovrebbe avere un agente e di come può essere inserito in un contesto dove più agenti agiscono sullo stesso Environment.

A questo punto, è importante definire un modello formale che possa descrivere il comportamento effettivo di questi agenti. Uno possibile è quello chiamato *Belife-Desire-Intention* (che può essere tradotto in *Convinzione-Desiderio-Intenzione*), o più brevemente *BDI*.

Questo modello ha origine nella teoria dell'*human pactical reasoning* (l'uso della ragione che porta a compiere determinate azione nell'essere umano) sviluppata dal filosofo Michael Bratman [28]. L'idea centrale di questo modello è di pensare ai programmi per computer come se avessero uno "stato mentale". Parlando quindi di sistemi BDI, ci si riferisce a programmi con analogie computazionali per quanto riguardo i Beliefs (ciò che crede sia vero), i Desires (ciò che si desidera ottenere) e gli Intents (ciò che ha intenzione di fare).

Riprendendo il concetto di agente:

- i *Beliefs* sono le informazioni che l'agente possiede riguardo all'ambiente in cui si trova e possono essere veritiere o meno. Ad esempio, un agente potrebbe credere che il livello di un contatore sia ad un certo valore quando in realtà è stato aggiornato ad uno differente. In un normale programma, un Belief può essere visto come una variabile contenente un dato valore.

- i *Desires* sono tutti i possibili compiti che l'agente vorrebbe svolgere. Il fatto che un agente abbia un Desire non implica necessariamente che l'agente si stia adoperando per raggiungerlo. Si tratta semplicemente di un potenziale condizionatore per le azioni dell'agente.

- gli *Intents* sono tutti i compiti su cui l'agente decide di lavorare. Le Intents potrebbero essere alcuni dei Goals che sono stati affidati all'agente. In pratica, gli Intents sono Desires che l'agente sceglie di svolgere.

A questo punto, definiti i Beliefs, Desires e Intents, rimane da formalizzare come l'agente sceglie le azioni da eseguire.

5.0.5 Ragionamento Pratico

Il *Practical Reasoning* (o *Ragionamento Pratico*) è un tipo di processo orientato alle azioni, al 'ciò che è da fare '.

"Il *Ragionamento Pratico* riguarda il pesare considerazioni in conflitto, a favore e contro opzioni contrapposte, dove le considerazioni rilevanti sono fornite da ciò che l'agente desidera/valuta/si preoccupa di e da cosa crede "[29]

Un altro tipo di ragionamento è l'*Epistemic Reasoning* (o *Ragionamento Epistemico*), orientato alla conoscenza. Si tratta del processo di aggiornamento delle informazioni, rimpiazzando quelle vecchie (non più consistenti con lo stato del mondo) con quelle nuove.

Il *Ragionamento Pratico* umano sembrerebbe consistere in due attività principali:

- *Deliberation*: la decisione dell'agente di quali compiti desidera svolgere in quel momento;
- *Means-Ends Reasoning*: la decisione dell'agente di come svolgere i compiti scelti.

L'output della fase di *Deliberation* (o *Deliberazione*) sono le Intents mentre l'output della fase di *Means-Ends Reasoning* (o *Ragionamento Mezzi-Fini*) sono un percorso di azioni atte al raggiungimento dei Goals.

Esaminando le Intents, esse possono essere descritte con quattro proprietà fondamentali:

1. Sono *pro-attitudinali*, ovvero portano a compiere delle azioni. Se un individuo ha l'intenzione di andare a correre, ci si aspetta che farà un qualche tipo di sforzo per raggiungere la sua intenzione; se non fa nulla di tutto ciò, si è inclini a pensare che quell'individuo non ha mai avuto quel tipo di intenzione.
2. Tendono a *persistere*, nel senso che adoperarsi per una intenzione fa sì che si persisterà su quella finché non sarà raggiunta. Ovviamente, se non ci sono più le condizioni per cui potrebbe valer la pena perseguire una data intenzione, è ragionevole che essa possa essere abbandonata.

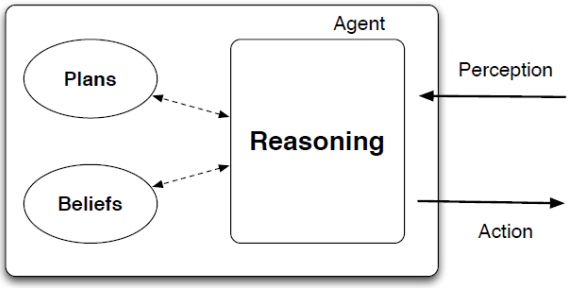

Figura 5.3: Architettura Base di un Agente

3. Il semplice fatto di avere date intenzioni, *condizionano* (o *vincolano*) il *Ragionamento Pratico* futuro dell'individuo.

4. Le intenzioni sono strettamente correlate su *ciò che si crede riguardo il futuro*. Intendere qualcosa, implica che si crede che quella cosa sia in principio possibile e, sotto normali circostanze, raggiungibile. È comunque ragionevole credere che si possa fallire nel tentativo di adempire ad una certa intenzione.

Per quanto riguarda il *Means-Ends Reasoning* è il processo di come raggiungere un fine (cioè un intenzione scelta) utilizzando i mezzi disponibili (cioè le azioni eseguibili nell'ambiente). In alternativa, il *Means-Ends Reasoning* può essere definito come un *planner*.

Prima di tutto, si parte con l'idea base di costituire gli agenti con tre tipologie di *Rappresentazioni*:

- Una rappresentazione dei *Goal/Intention* da raggiungere.
- Una rappresentazione delle *Actions/Plans* in repertorio.
- Una rappresentazione dell'Environment.

Presi in ingresso questi elementi, l'algoritmo del *planner* genererà in output un *Plan* (o un *Piano*), ovvero una sequenza di azioni che dovrebbe portare l'agente a raggiungere l'intenzione prefissata se, dallo stato dell'ambiente in cui si trovava inizialmente, riesce ad eseguire tutte le azioni pianificate.

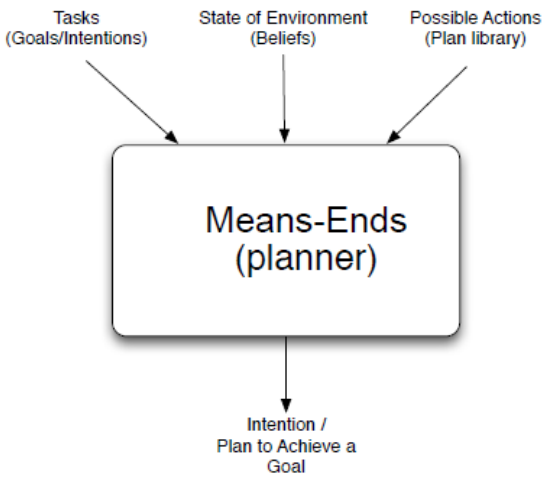

Figura 5.4: Schema di un Agent Planner

Esistono diversi approcci su come intendere i *Plans*; uno di questi è quello di definirli come una sequenza predefinita di azioni (essenzialmente un programma) dove le componenti atomiche di queste rappresentano le azioni disponibili all'agente. Un ulteriore approccio è quello di rigettare completamente i *Plans* nel processo di decisione cercando invece modelli alternativi.

Nel modello che prenderemo in considerazione invece, il programmatore svilupperà una collezione di *Partial Plans* (o *Piani Parziali*) per un agente in fase di progettazione, sarà poi compito dell'agente quello di assemblarli a run time, in modo tale da poter raggiungere il Goal scelto.

5.0.6 Modello Computazionale per il Practical Reeasoning BDI

Per implementare un agente basato sul modello *Practical Reasoning* definito precedentemente, è possibile dare una prima definizione di *Control Loop* (o *Ciclo di Controllo*) dove l'agente continuamente:

- osserva l'Environment e aggiorna i propri Beliefs;

- decide quale Intent vuole raggiungere;
- usa il *Means-Ends Reasoning* per trovare un Plan su come raggiungere l'Intent deciso;
- esegue il Plan.

Tuttavia in questo modo l'agente è troppo focalizzato sull'Intent; anche se nell'Environment succedesse qualcosa per cui renderebbe il raggiungimento di tale obbiettivo inutile, l'agente continuerebbe nella prosecuzione delle azioni.

Per questo motivo, c'è bisogno di un agente più "reattivo", che rimanga sì impegnato per l'Intent scelto finché non viene raggiunta, ma che controlli anche continuamente se questa sia ancora possibile da raggiungere o se c'è qualcosa di più utile su cui l'agente potrebbe lavorare.

Ricapitolando, le architetture *BDI* sono basate sui seguenti costrutti:

- Un set di *Beliefs*;
- Un set di *Desires*;
- Un set di *Intents* (descritti come un subset dei Goals a cui sono associati degli Stack di Plans per poterli raggiungere);
- Un set di *Internal Events* (*Eventi Interni*), derivati da un cambio nei Beliefs dell'agente (cioè o ne viene aggiunto uno nuovo, cancellato o aggiornato uno già presente) o da eventi generati da Goals (cioè il completamento o l'adozione di un nuovo obbiettivo);
- Un set di *External Events* (*Eventi esterni*), derivati dall'interazione con le entità esterne (ad esempio, la ricezione di un messaggio da parte di un altro agente);
- Una libreria di Plan (o repertorio di azioni).

Un'architettura di base può essere quella in 5.5, dove vengono mostrati i tre concetti centrali di questo modello, i Beliefs, i Desires e gli Intents, e le funzioni che li regolano e li mettono in relazione tra loro.

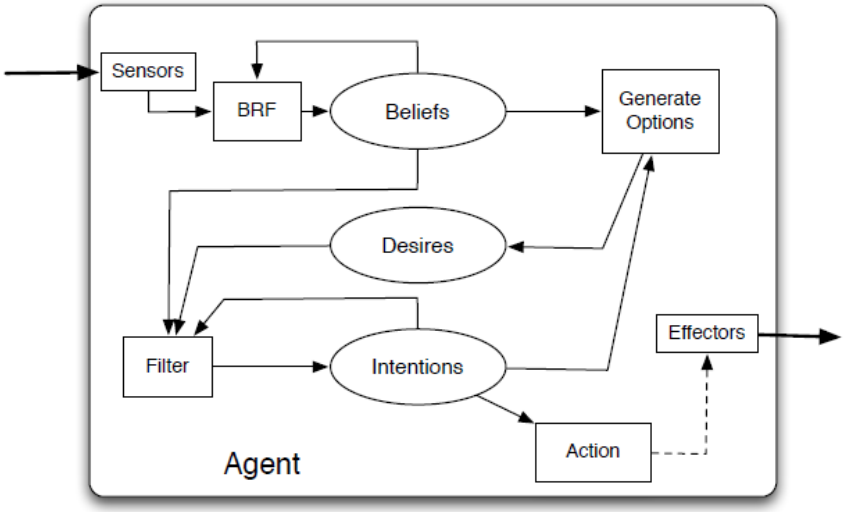

Figura 5.5: Architettura Base di un Agente BDI

- *Sensors*:= $sensors(Environment) \rightarrow Events$: funzione che genera un set di eventi percepiti nell'Environment in cui l'agente è situato;
- *BRF* (*Belief Revision Function*):= $brf(Events, Beliefs) \rightarrow Beliefs'$: funzione che aggiorna la base dei Beliefs dell'agente, dipendentemente dagli eventi esterni percepiti;
- *Generate Options*:= $generateOptions(Beliefs', Intentions) \rightarrow Desires'$: funzione che aggiorna i Desires dell'agente, a seconda di ciò che crede riguardante l'ambiente in cui si trova e ciò che sono le sue attuali intenzioni;
- *Filter*:= $filter(Beliefs', Desires', Intentions) \rightarrow Intentions'$: funzione che valuta i Beliefs e i Desires aggiornati dell'agente, li confronta con i vecchi Intents e considera se possono essercene di nuove più adatte per lo stato attuale;
- *Action*:= $action(Intentions') \rightarrow signalsToActuators$: funzione che dall'Intent corrente ottiene la sequenza di azioni da eseguire per poter

raggiungere il proprio obbiettivo. A seconda dello stato della sequenza in cui si trova, dovrà eseguire una diversa azione che dev'essere codificata in segnali per gli attuatori.

- $Actuators := actuators(signalsToActuators)$: l'ultima funzione della catena, a seconda di una data codifica dei segnali, l'agente eseguirà una diversa azione, che potrà o meno cambiare lo stato dell'Environment.

Capitolo 6

JaCaMo

Le tecnologie multi-agente forniscono concetti e strumenti adatti per risolvere le sfide riguardanti la programmazione di sistemi software aperti, che operano in ambienti dinamici e complessi, interagendo ed agendo come un umani.

Almeno quattro separate correnti di ricerca, all'interno della comunità multi-agente, si stanno interessando a specifiche dimensioni riguardanti la programmazione pratica di software. Nello specifico, abbiamo quella riguardante i linguaggi di programmazione agent-oriented, quella che si interessa sui linguaggi e protocolli di interazione, quella sulla progettazione di frameworks, infrastrutture e architetture di environments, ed infine quella che si occupa dei sistemi di gestione dell'organizzazione di agenti.

Ognuna di queste quattro dimensioni è importante nella programmazione di applicazioni complesse e distribuite, tuttavia un approccio che implichi l'utilizzo di una sola di queste risulta limitante. Si è invece visto come, nell'ambito di *AOSE* (*Agent Oriented Software Engineering*) e progettazione *MAS*, un approccio comprensivo di più di una delle suddette dimensioni, possa portare dei grossi benefici.

È proprio questa l'idea alla base di *JaCaMo*, una piattaforma che integra tra di loro tre differenti aspetti, la programmazione di agenti (con *Jason*), la programmazione di environments (con *CArtAgO*) e la programmazione di organizzazioni (con *Moise*).

6.1 La Piattaforma

JaCaMo unisce in un unico modello, tre differenti dimensioni di programmazione -agenti, environment e organizzazione - attraverso collegamenti semantici di concetti analoghi, a livello di meta-modello.

Nello specifico, le tre parti:

- *Jason* è una piattaforma per lo sviluppo di sistemi multi-agente che incorpora un linguaggio di programmazione *agent oriented*, nato a seguito di successive espansioni al linguaggio AgentSpeak che ne hanno reso una variante di quest'ultimo;
- *CArtAgO* è un framework ed una infrastruttura per l' *environment programming* e l'esecuzione di sistemi mulit-agente, la cui idea si fonda sul fatto che l'environment può essere considerato come un entità di prima classe per la progettazione di *MAS* e che incapsula funzionalità e servizi che gli agenti possono sfruttare a runtime;
- *Moise* è un framework che implementa un modello di programmazione per la dimensione organizzativa che include un linguaggio ad hoc, un infrastruttura di gestione dell'organizzazione ed il supporto per meccanismi di ragionamento organisation-based a livello di agenti.

Un sistema multi-agente JaCaMo è quindi un sistema software programmato in JaCaMo, definito da un organizzazione, programmata in Moise, di agenti autonomi, programmati Jason, operanti in un ambiente distribuito e condiviso basato su Artefatti, programmato in CArtAgO (figura 6.1).

6.2 Approccio JaCa

Per lo scopo di questo lavoro, si tralascerà la parte riguardante l'organizzazione degli agenti, la programmazione con Moise, focalizzandosi invece sulle rimanenti parti, *JaCa*, che vanno a definire una concreto modello computazionale e di programmazione, basato sul meta-modello *A&A* (*Agents and Artifacts,* ovvero*Agenti e Artefatti*) dove:

- *Agenti*: le entità autonome che si occupano proattivamente di compiere ad un compito assegnatogli, interagendo con l'ambiente in cui sono logicamente situati.

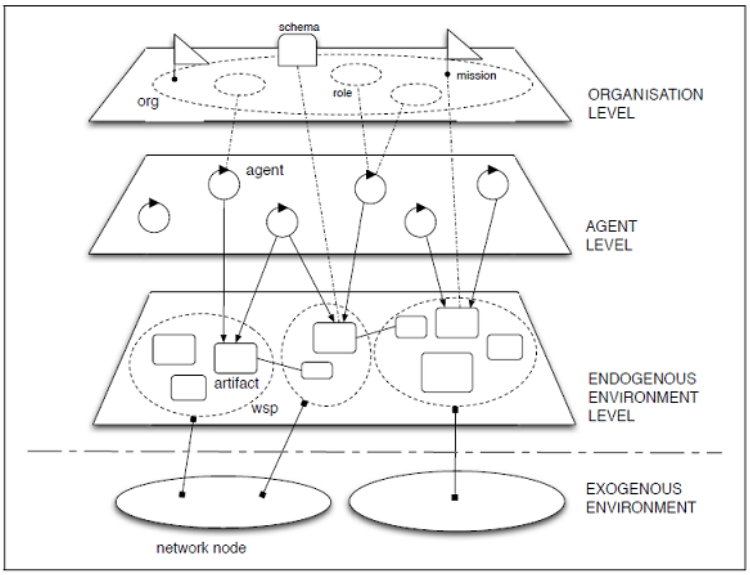

Figura 6.1: Una rappresentazione di un sistema multi-agente JaCaMo

- *Artefatti*: le entità passive che modellano parte dell'ambiente in cui si trovano gli agenti.

Un sistema *JaCa* sarà dunque leggermente diverso da quello mostrato in figura 6.1, risultando in una variante più simile al modello rappresentato in figura 6.2.

Prima di passare allo studio dei due componenti principali, Jason e CArtAgO, verrà introdotto brevemente il modello *A&A*, in cui si chiariranno dei concetti utili per la comprensione delle parti successive.

6.2.1 Il Modello A&A

Come sottolineato da Liebermann [30]: "[...] La storia della Object-Oriented Programmin può essere interpretata come una continua ricerca volta alla cattura della nozione di astrazione - per creare Artefatti computazionali che rappresentassero la natura essenziale delle situazioni ed ignorassero i dettagli irrilevanti. [...]". Metafore ed astrazioni continuano a giocare un ruolo fondamentale per la computer science e per l'ingegneria del software in

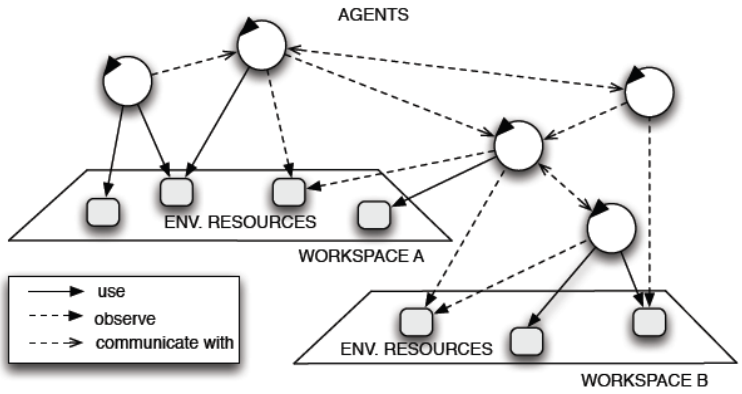

Figura 6.2: Una rappresentazione di un sistema multi-agente *JaCa*

generale, fornendo concetti adatti per modellare, disegnare e programmare sistemi software.

Concentrandosi sullo studio di quei sistemi che si basano su ambienti di lavoro umano cooperativo come esempi di partenza per giungere a delineare un modello per sistemi software analoghi, è possibile identificare due tipi di entità: i lavoratori umani, coloro responsabili di svolgere dei compiti in maniera proattiva, e gli Artefatti, strumenti che i lavoratori usano come supporto alle loro attività. Quest'ultimi possono essere un qualsiasi tipo di risorsa o strumento per la mediazione e la coordinazione collettiva delle attività stesse.

È proprio ispirandosi a questi ambienti di lavoro collettivo che è nato il modello di programmazione chiamato *A&A* (*Agents and Artifacts,* ovvero*Agenti e Artefatti*). Esso si basa su due astrazioni fondamentali, *Agenti* ed *Artefatti*, con i quali è possibile realizzare applicazioni complesse.

Da una parte c'è l'Agente - analogo al lavoratore umano - che è utile per modellare la parte del sistema attiva e task-oriented, incapsulando la logica ed il controllo di ogni attività. Dall'altra si ha l'Artefatto- analogo agli strumenti negli ambienti umani - che è utile per modellare la parte del sistema passiva e function-oriented, usata dagli agenti durante le loro attività.

Dietro ad Agenti ed Artefatti, la nozione di *Workspace* completa il set di astrazioni definite nel modello *A&A*: un Workspace è un contenitore logico di Agenti ed Artefatti, è il concetto base per la definizione esplicita di una

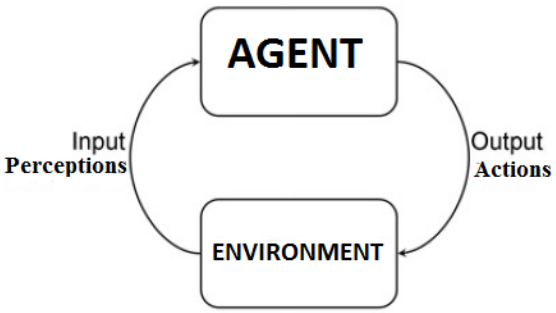

Figura 6.3: Interazione Agente-Ambiente *JaCa*

topologia dell'ambiente di lavoro.

In questo modello, il significato di Agente è proprio quello di un entità che *agisce* sull'ambiente e ne *percepisce* i cambiamenti (6.3).

Nel modello *A&A*, le azioni e le percezioni di un Agente concernono l'interazione con altri agenti ed Artefatti. La nozione di attività è usata per raggruppare azioni collegate tra loro, in modo da strutturare il comportamento proattivo generale dell'Agente.

L'Artefatto invece è strettamente correlato alla nozione di uso. Sono entità passive che vengono usate dagli Agenti, che assemblate assieme formano l'ambiente di lavoro. Le funzionalità sono *operazioni*, attivabili dagli agenti tramite delle *interfacce*. Qui la nozione di interfaccia è analoga a quella della programmazione object-oriented; l'interfaccia definisce *un set di operazioni* che un Agente può utilizzare per i propri scopi, a patto di fornire una determinata lista di parametri in ingresso, ed *un set di proprietà osservabili* dagli agenti, utili per conoscere lo stato dell'Artefatto senza necessariamente utilizzarlo.

La principale differenza fra gli Artefatti e gli oggetti nella programmazione object-oriented è che il flusso di controllo dell'agente rimane all'interno dell'agente e non passa all'Artefatto che ne possiede uno proprio. Questo permette di svincolare l'Agente, che così può tornare ad eseguire altre operazioni, dall'esecuzione vera e propria dell'operazione, che avviene su di un flusso di controllo distinto e che quindi viene portata a termine in maniera indipendente ed asincrona.

L'interazione avviene quindi con gli Agenti che mediante *azioni*, eseguo-

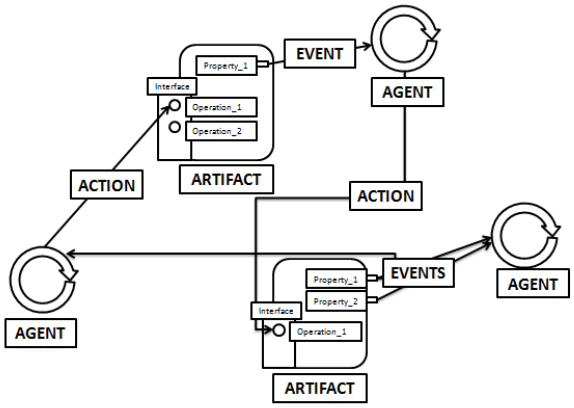

Figura 6.4: Rappresentazione di un *MAS* con agenti che eseguono operazioni che scaturiscono eventi percepiti da altri agenti *JaCa*

no le operazioni, richiamando il corrispettivo nome con i parametri necessari definiti dall'interfaccia, rese disponibili dagli Artefatti. Queste operazioni possono cambiare lo stato dell'Artefatto a livello delle proprietà osservabili e che quindi generano *eventi*, percepibili dagli Agenti i quali possono agire e reagire di conseguenza.

6.2.2 Agenti in Jason

Jason è il linguaggio utilizzato per la programmazione delle entità autonome, gli agenti.

Si tratta di un'estensione del linguaggio AgentSpeak, il modello degli agenti Jason è quindi praticamente identico a quello BDI. Questo significa che gli agenti non sono programmati per calcolare il valore di una funzione e terminare, piuttosto sono progettati per poter essere sempre in esecuzione, reagendo ad un qualche tipo di eventi. La modalità con gli agenti reagiscono è mediante l'esecuzione di piani, una sequenza di azioni per gestire i dati eventi. Ogni azione è finalizzata a cambiare l'ambiente circostante, in modo tale che l'agente si aspetti che lo scopo prefissato sia stato raggiunto.

In poche parole, **l'agente costantemente percepisce l'ambiente in cui si trova, ragiona su come agire in modo tale da raggiungere**

i propri obbiettivi e agisce di conseguenza, cambiando l'ambiente circostante.

Nonostante Jason presenti delle ovvie differenze rispetto ad AgentSpeak, si possono ritrovare tutti i principali concetti del modello BDI.

I **Beliefs** rappresentano ciò che l'agente crede (ma che non necessariamente debba essere vero) riguardo al mondo esterno (stato dell'ambiente e degli altri agenti). Vengono espressi come una collezione di *predicati*, intesi come nella programmazione logica, a cui è possibile allegare delle *annotazioni*. Queste annotazioni possono essere utilizzate in vario modo, il motivo per cui sono state introdotte è per aggiungere alle informazioni la *sorgente* che ha portato l'agente a credere riguardo qualcosa. In generale, in un sistema multi-agente, vi possono essere tre sorgenti di informazioni: i sensori dell'agente (informazioni percepite), comunicazione con altri agenti e note mentali aggiunte dall'agente stesso come parte dell'esecuzione di un piano. Vi sono inoltre entrambe le assunzioni di negazione, quella del mondo-chiuso (tutto ciò che non è a conoscenza e che non è derivabile, è considerato falso) e quella del mondo-aperto (si crede esplicitamente che qualcosa è falso). È inoltre possibile esprimere i *Beliefs* mediante regole prolog-like.

I **Goals** esprimono delle proprietà del mondo che l'agente vorrebbe che si realizzino. Fintanto che l'agente non crederà che un dato goal sia vero, continuerà ad operare per raggiungerlo. Vi sono due tipi di goal: gli *Achievemt Goals*, il raggiungimento di un certo stato delle cose, o i *Test Goals*, un semplice obbiettivo per recuperare delle informazioni.

Infine **Plans** rappresentano le strategie di esecuzione per il raggiungimento degli obbiettivi. Sono costituiti da tre parti fondamentali: i *Triggering Event*, il *Context* ed il *Body*. Per quanto riguarda il Triggering Event, si tratta dell'evento che può scatenare l'esecuzione di un piano. Il Context rappresenta una serie di condizioni che devono essere verificate affinché si possa eseguire un certo piano; è nei fatti un discriminatore o una precondizione che indica se un dato piano è applicabile in un dato momento, quello di percezione del Triggering Event. Il Body è definito mediante una serie di formule che determinano il corso delle azioni; non necessariamente ogni formula è una semplice azione che dev'essere eseguita, ma può essere a sua

volta un sotto-goal che l'agente deve poter compiere al fine di eseguire il piano correttamente.

Dopo aver descritto le caratteristiche di un agente Jason, vediamo qual'è il **Ciclo di Controllo** effettivo e quali sono gli step principali.

1. Percezione dell'ambiente;
2. Aggiornamento della *Belief Base*;
3. Ricezione di comunicazioni da parte di altri agenti;
4. Selezione dei messaggi "socialmente accettabili";
5. Selezione di un evento;
6. Ricerca dei piani rilevanti;
7. Determinazione dell'insieme dei piani rilevanti;
8. Selezione dei piani applicabili;
9. Selezione di una *Intention*;
10. Esecuzione di uno step dell'*Intention* selezionata.

Si mette in evidenza che l'agente potrebbe avere un set di intenzioni sospese, in attesa di un qualche tipo di feedback per l'azione compiuta o che aspettano la ricezione di un messaggio di risposta da un altro agente. Prima del nuovo ciclo del *Control Loop*, l'agente verifica se vi è uno di quei feedback o messaggi per cui alcune intenzioni sono in attesa; in caso affermativo, vengono re-introdotte le corrispettive intenzioni nel set di quelle attive, così che possono essere nuovamente selezionate al successivo ciclo di controllo.

6.2.3 Artefatti in CArtAgO

CArtAgO (Common ARtifact infrastructure for AGent Open environments) è un framework ed una infrastruttura per la programmazione e l'esecuzione di ambineti artifact-based. Fornisce un set di API Java-based per programmare gli Artefatti e l'ambiente runtime per l'esecuzione di questi, assieme

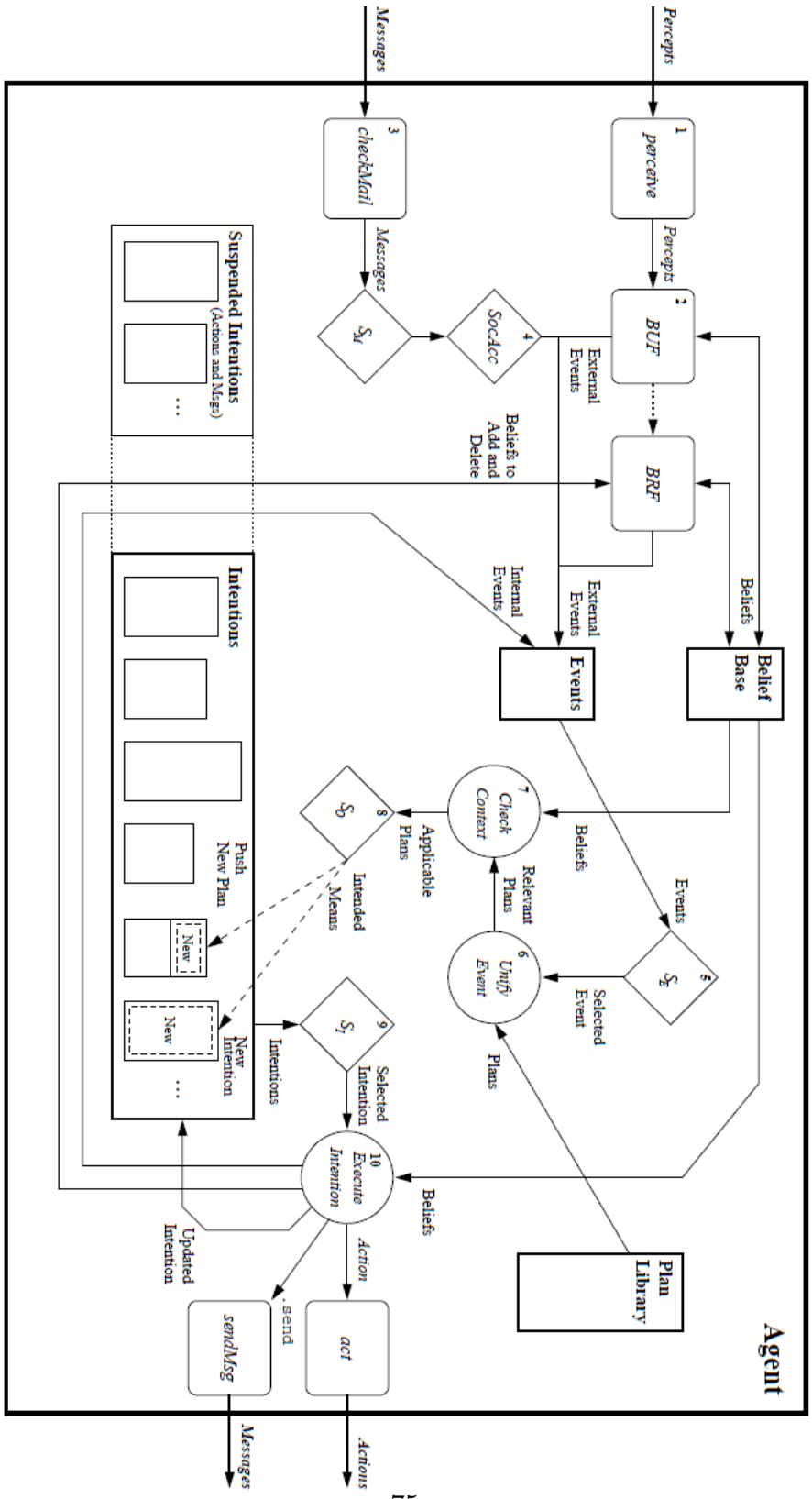

Figura 6.5: Ciclo di controllo di un agente Jason

ad una libreria con un set di tipi d'Artefatti predefiniti general-purpose. Gli Artefatti sono programmati in termini di classi e tipi di dati base Java, senza l'utilizzo di introdurre un altro linguaggio di programmazione specifico. Un tipo di Artefatto è programmato direttamente definendo una classe Java che estende *cartago.Artifact*, di cui usa un set base di annotazioni e ne eredita i metodi per definirne la struttura ed il comportamento. Le Proprietà Osservabili sono definite mediante la primitiva *defineObsProperty*, che ne specifica nome ed il valore iniziale. Le operazioni eseguibili dagli agenti sono definite mediante metodi annotati con il tag *@OPERATION* e il valore di ritorno *void*. Gli avvenimenti principali scatenanti eventi percepibili dagli agenti, qualora essi ne siano interessati, sono:

1. L'aggiornamento di una proprietà osservabile;
2. L'invio esplicito di una *signal* da parte dell'Artefatto.

Indirettamente, anche il completamento con successo o il fallimento di una operazione, genera un evento percepibile dall'agente che ha richiamato l'operazione.

Si precisa che un operazione può essere composta da più sotto-operazioni divise in diversi step (ad esempio, le operazioni interne sono metodi a cui è associato il tag *@INTERNAL_OPERATION*). Ogni step è eseguito **atomicamente**. Le operazioni a singolo step sono perciò eseguite in maniera mutualmente esclusiva, al contrario per quanto riguarda le operazioni composte da più step, queste possono essere eseguite concorrentemente, combinando arbitrariamente i vari step atomici.

Esistono altri tipologie di operazioni, definite da altrettanti tag, come quelle richiamabili da altri Artefatti (devono essere annotate con il tag *@LINK*); questa non vuol essere una disamina esaustiva di tutte le possibili operazioni ma solo una presentazione di quelle principali.

La prossima parte riguarda il come far interagire gli agenti Jason con gli Artefatti CArtAgO.

6.2.4 Integrazione Agenti Jason e Artefatti CArtAgO

Come è stato spiegato in precedenza, JaCa si compone di Jason per la definizione di agenti BDI e del framework CArtAgO per la realizzazione dell'ambiente in cui gli agenti sono situati.

L'integrazione è resa possibile grazie ad una tecnologia che fa da bridge tra questi due mondi: C4Jason.

Questo modulo permette di estendere il repertorio di azioni interne che l'agente può eseguire con un set in grado di poter operare con gli ambienti basati su Artefatti:

1. *joinWorkspace(+Workspace[,Node])*
2. *quitWorkspace*
3. *makeArtifact(+Artifact,+ArtifactType[,ArtifactConfig])*
4. *lookupArtifact(+ArtifactDesc,?Artifact)*
5. *disposeArtifact(+Artifact)*
6. *use(+Artifact,+UIControl([Params])[,Sensor] [,Timeout] [,Filter])*
7. *sense(+Sensor,?PerceivedEvent[,Filter] [,Timeout])*
8. *focus(+Artifact[,Sensor] [,Filter])*
9. *stopFocussing(+Artifact)*
10. *observeProperty(+Artifact,+Property,?PropertyValue)*

Le suddette azioni possono essere raggruppate in 5 gruppi, quelle riguardanti la gestione dei Workspaces (1-2), quelle sulla creazione, distruzione e ricerca di Artefatti (3-5), quelle per poter utilizzare le operazioni messe a disposizione dell'Artefatto (6-7) e quelle per poter osservare gli eventi emessi dall'Artefatto (8-10). La sintassi è espressa in notazione logica, dove gli elementi all'interno delle parentesi quadre sono opzionali.

Capitolo 7

Cicli di Controllo: Event Loop e Control Loop

In questo capitolo ci si concentrerà su quelli che sono i cicli di esecuzione del modello ad eventi e quello ad agenti proposto: Event Loop e Control Loop. Questi presentano sì delle similitudini, ma anche profonde differenze che verranno analizzate e confrontate. Come riferimento, si utilizzerà il modello generico di Event Loop, descritto nel capitolo 3.3. Possono esistere varianti allo schema tradizionale, come quello adottato in Dart, tuttavia il funzionamento generale è analogo al modello generale.

Per effettuare il confronto fra i due cicli, si mostreranno le varie fasi del *Control Loop* di AgentSpeak (da cui trae Jason) e si paragonerà a ciò che viene fatto in corrispondenza nell'Event Loop.

7.1 Semantica Operazionale del Control Loop

Per aiutare a comprendere meglio i vari passaggi, si mostrerà un grafico dove si evidenziano tutte le possibili transizioni fra i vari step del ciclo di controllo (7.1).

- **ProcMsg**: processare i messaggi ricevuti;
- **SelEv**: selezionare un evento dal set di eventi;
- **RelPl**: recuperare i Plans rilevanti;

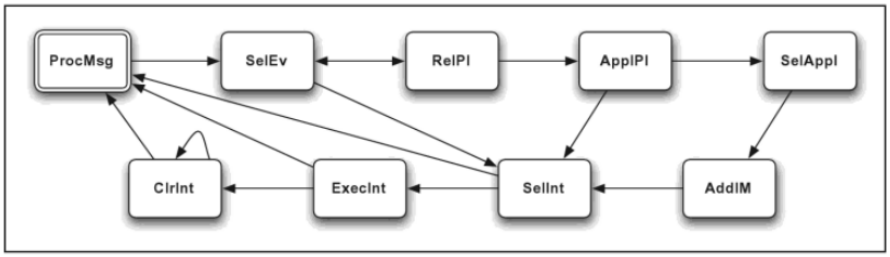

Figura 7.1: Possibili transizioni di stato all'interno di un ciclo di controllo

- **ApplPl**: selezionare i Plans applicabili;
- **SelAppl**: selezionare un particolare Plan applicabile - l'azione che si intende compiere;
- **AddIM**: aggiungere l'azione che l'agente vuole compiere al set delle Intents;
- **SelInt**: selezionare un Intent;
- **ExecInt**: eseguire l'Intent selezionato;
- **ClrInt**: rimuovere un Intent o l'azione intesa che può essere terminata nello step precedente.

Si noti che il ciclo incomincia con il processare i messaggi ricevuti ma non si entrerà nel dettaglio di come vengono effettivamente scelti, definendoli semplicemente degli eventi.

Definizione Formale di un Agente

Prima di analizzare i vari step del Control Loop, è necessario definire formalmente un Agente, in tutte le sue strutture principali. La descrizione che si da è una versione semplificata ma sufficientemente esaustiva per poter comprendere le sezioni successive.

Un Agente ag è formato da un set di Beliefs bs ed un set di Plans ps. La singola Belief ed il singolo Plan verranno rispettivamente rappresentati con b e p.

Per un singolo Plan p, te rappresenta il Triggering Event, ct il Context di un Plan mentre h il Body di un Plan (con $\top$ si identifica un Body vuoto).

Lo stato di un agente è rappresentato da una tupla $< I, E, A >$ dove:

- I rappresenta un set di Intenzioni $i_0, i_1, \ldots$, ognuna delle quali è **uno Stack di Plans parzialmente istanziati**.
- E rappresenta un set di eventi $(te_0, i_0), (te_1, i_1), \ldots$ dove i te sono gli Eventi Scatenanti mentre i è un Intent (uno Stack di Plans se si trattano di eventi interni, un'Intent vuoto $\top$ nel caso di eventi esterni). L'aggiornamento della base di Credenza viene trattato come un evento esterno mentre l'aggiunta o la rimozione di un Goal è considerato un evento interno.
- A rappresenta un set di Azioni che devono essere eseguite nell'ambiente.

All'interno di un singolo Ciclo di Controllo, si usa un set informazioni temporanee, rappresentate dalla tupla $< R, Ap, \iota, \varepsilon, \rho >$ dove:

- R rappresenta il set di Plans rilevanti, per l'evento da gestire.
- Ap rappresenta il set di Plans applicabili, ovvero i Plans per cui il Contesto è considerato vero dall'agente.
- ι indica un particolare Intent.
- ε indica un particolare evento.
- ρ indica un particolare Plan applicabile.

Vi sono infine tre funzioni di selezione:

- S_ε: seleziona un evento ε dal set di eventi E.
- S_O: seleziona un Plan applicabile ρ dal set di Plans applicabili Ap.
- S_I: seleziona un Intent ι dal set di Intenzioni I.

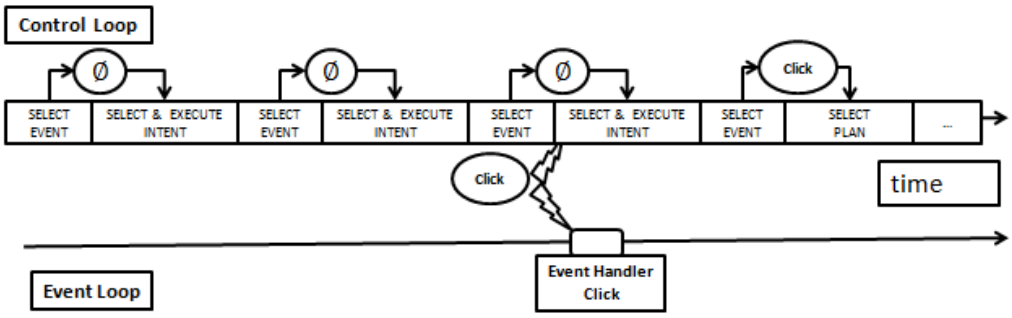

Figura 7.2: Esecuzione temporale dei due cicli con la coda vuota (∅ rappresenta l'evento nullo) e l'arrivo di un evento ad un istante arbitrario

7.1.1 Selezione dell'Evento

In questa fase, tramite la funzione S_ε viene scelto un evento dal set di possibili eventi E.

Qui incontriamo la prima differenza con l'Event Loop, dove tipicamente si ha una coda di eventi, ovvero il primo evento che arriva è il primo ad essere elaborato.

Nel Control Loop è possibile avere una priorità fra gli eventi, definendo la funzione sopracitata (Dart definisce una regola di precedenza fra eventi utilizzando due code distinte).

È tuttavia possibile ottenere un comportamento analogo all'Event Loop definendo E come una coda (FIFO, il primo elemento inserito è anche il primo ad essere estratto) e non più come un set, con la rispettiva funzione S_ε che si limita ad estrarre il primo elemento della coda.

La seconda rilevante differenza è il comportamento dei due cicli di controllo quando la coda degli eventi è vuota. Per quanto riguarda l'Event Loop, il ciclo viene interrotto in attesa di un nuovo evento da processare. Al contrario, il Control Loop non si blocca ad aspettare e passa direttamente nella fase di *SelInt*.

Mostriamo cosa significa questo con un piccolo esempio in figura 7.2.

Come si può vedere bene, fintanto che la coda degli eventi è vuota, l'Event Loop si blocca e rimane in attesa di processare un nuovo evento. Il Control Loop invece non si interrompe mai e prosegue nel verificare se

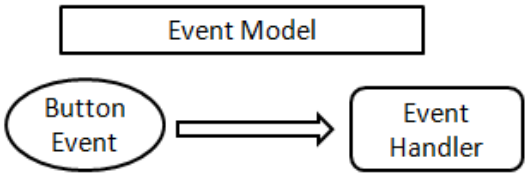

Figura 7.3: Associazione tra Evento ed Event Handler

ci sono intenzioni selezionabili da eseguire. Questa è una caratteristica fondamentale che permette agli agenti di essere realmente proattivi. Essi non sono necessariamente vincolati alla ricezione di un evento per svolgere dei compiti o eseguire delle operazioni, e possono avere comunque delle funzioni da svolgere che non dipendono strettamente da ciò che succede nell'ambiente in cui sono inseriti. A favore dell'Event Loop si potrebbe dire che l'evento viene gestito il prima possibile, rendendo questo ciclo più reattivo rispetto alla sua controparte.

7.1.2 Recupero dei Plans Rilevanti

In questa fase viene assegnato al componente R, un set di Plans rilevanti per l'evento ε selezionato. Se per caso non vi è alcun Plan rilevante applicabile per un dato evento, si ritorna nella fase di *SelEv*.

Com'era stato precedentemente detto, i Plans hanno una parte chiamata *Triggering Event* che rappresenta un evento, generato internamento o esternamente, che rende il Plan in questione, rilevante per quel dato tipo di evento (l'applicabilità effettiva o meno dipenderà anche dal Contesto del Plan, che verrà analizzato nella fase successiva).

Il set di Plans rilevanti è proprio definito in base al match fra l'evento scelto nella fase precedente e il *Triggering Event* dei Plans stessi.

Nel modello con Event-Loop, ad ogni evento è tipicamente associato un solo Event Handler (7.3).

Al contrario, nel modello ad agenti, un evento può avere associato più Plans, alcuni più adatti di altri a seconda dello stato in cui l'ambiente si trova.

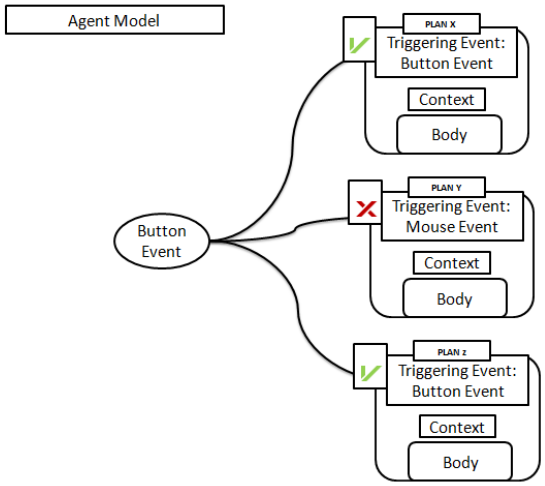

Figura 7.4: Un agente, per ogni tipo di evento, può avere una serie di Plans da cui scegliere

7.1.3 Selezione dei Plans Applicabili

Questa fase assegna un set di Plans applicabili al componente *Ap*. Se il set è vuoto, l'evento viene semplicemente scartato e si passa alla fase di *SelInt*.

Questo è di fatto uno step aggiuntivo nel Control Loop rispetto all'Event Loop, dovuto al fatto che ad ogni evento abbiamo più Plans applicabili e che l'applicabilità di questi dipende anche dal Contesto di ogni Plan. Questa fase aggiuntiva può essere vista come una esplicitazione di ciò che, in un semplice modello ad eventi, verrebbe fatto nel seguente modo):

```
eventHandler(Event e)
{
 if(currentState == X)
 {
  ...
 }
 else if(currentState == Z)
 {
  ...
```

```
  }
  else
  {
    ...
  }
}
```

7.1.4 Selezione di un Plan Applicabile

In questa fase, tramite la funzione S_O, viene scelto un Plan dal set dei Plans applicabili Ap, che viene assegnato alla componente ρ della configurazione.

Anche questa parte risulta essere esclusiva del Control Loop, conseguenza del fatto che ancora l'agente non sa cosa fare a partire dall'evento selezionato. In questo caso, la funzione S_O deve essere in grado di selezionare un Plan dalla Libreria dei Plans, potenzialmente applicabile; un Plan potrebbe essere più prioritario dell'altro anche in dipendenza di Note Mentali che l'Agente si è appuntato.

A questo punto si inizia ad intravedere come, la necessità di dover definire entità con determinate caratteristiche e comportamenti, si rifletti in un ciclo di controllo molto più articolato e complesso.

7.1.5 Aggiunta di un Comportamento Strumentale al Set di Intents

Un evento può essere classificato come esterno o interno, a seconda che se son stati generati da una percezione dell'Ambiente da parte dell'Agente o se provengono dall'esecuzione di Plans precedenti.

Se l'evento ε è esterno, un nuovo Intent viene creato e l'unico Comportamento Strumentale che l'agente intende intraprendere è il Plan p assegnato al componente ρ. Se l'evento è interno, il Plan p verrà posto in cima alle Intenzioni associate a quel determinato evento (figura 7.5).

Questa organizzazione delle Intents, permette all'agente di portare avanti più task in parallelo. Nel modello con il Control Loop, i compiti dell'agente sono ben definiti dal set di Plan di cui dispone.

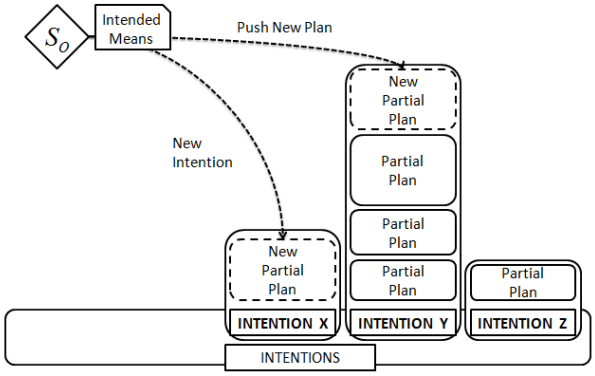

Figura 7.5: Il risultato della funzione S_O è la creazione di un nuovo Intent, nel caso di evento esterno, o l'aggiunta di un Plan in cima allo Stack dei Plans

Nel modello ad eventi, questa distinzione marcata non c'è perché non esiste il concetto di Plan o di Intent, esiste semplicemente il concetto di Event Handler, che a conti fatti è una funzione.

7.1.6 Selezione dell'Intent

In questa fase, tramite una funzione S_I viene selezionato un Intent (ovvero uno Stack di Plan) da processare nello step successivo.

Nel caso in cui il set di Intents sia vuoto, il ciclo ritorna alla fase iniziale.

Anche in questo caso ci troviamo in una fase specifica per il Control Loop, dove non esiste un vero e proprio corrispettivo nell'Event Loop. Un agente, potendo portare avanti più Intents in parallelo, necessita di dover definire questo tipo di funzione.

7.1.7 Esecuzione di un Comportamento Strumentale

Fase rappresentante l'esecuzione vera e propria del Body di un Plan. Il Plan che viene eseguito è sempre quello in cima all'Intent che è stato selezionato nella fase precedente; la formula che viene eseguita è la prima del Body del Plan. Com'è già stato detto, il Body di un Plan può essere composto da varie tipologie di formule:

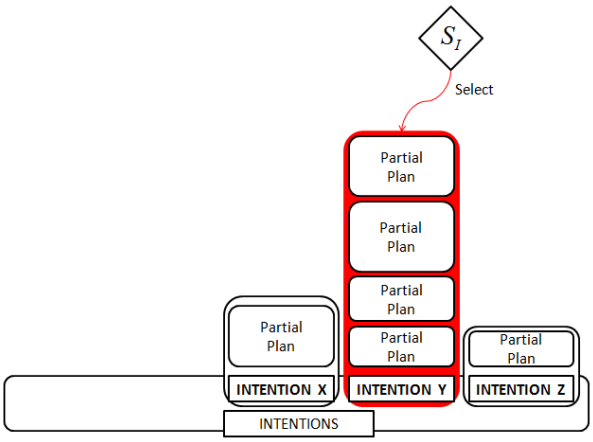

Figura 7.6: La funzione S_I seleziona un Intent, fra il set di Intents disponibili, da eseguire

- Actions (Azioni): l'Azione a nel Body del Plan viene aggiunta ad un set di Azioni A; successivamente viene rimossa dal Body del Plan e l'Intent viene modificato per riflettere questo aggiornamento.
- Achievement Goals (Goal di Raggiungimento): si registra un nuovo evento interno nel set di eventi E che in futuro verrà selezionato. Quando la formula eseguita è un Goal, essa non viene rimossa dal Body del Plan fintanto che non viene completato con successo. Il motivo è perché il Plan può essere istanziato ulteriormente, oltre che a servire per gestire i fallimenti dei Plans associati al raggiungimento del sotto-Goal.
- Updating Beliefs (Aggiornamento della Base di Credenza): viene aggiunto un nuovo evento al set di eventi E, si rimuove la formula dal Body del Plan e si aggiorna il set di Intenzioni in maniera appropriata.

7.1.8 Rimozione delle Intents

Ultima fase, riguardante la rimozione dei Comportamenti Strumentali o delle Intenzioni vuote dal corrispettivo set delle Intenzioni. Si rimuove un intero Intent se non vi è più altro da eseguire per quell'Intent. Inoltre viene

eliminato il resto di un Plan avente un Body vuoto che si trova in cima ad un (non vuoto) Intent; in questo caso, è necessario l'istanziamento del Plan al di sotto di quello appena terminato (quello che si trovava in cima all'Intent) e rimuovere il Goal che era lasciato all'inizio del Body del Plan sotto.

Si evidenzia come ulteriori fasi di *ClrInt* possano essere necessarie prima di poter ricominciare il ciclo di controllo.

7.2 Considerazioni

Ad una prima occhiata, è piuttosto evidente come il Control Loop sia estremamente più complesso rispetto all'Event Loop. Per poter descrivere comportamenti proattiviti e reattiviti, per poter formalizzare bene l'idea di assegnazione di uno o più compiti, i quali possono essere sospesi, ripresi o abbandonati, si sono dovuti introdurre concetti di alto livello ed aggiungere ulteriori step nel ciclo di controllo.

Sicuramente esistono dei contesti applicativi dove il Control Loop descritto risulterebbe troppo complesso, favorendo dunque un approccio più semplice ed immediato fornito dall'Event Loop. Tuttavia è possibile che, per certe tipologie di Web Applications, avere a disposizione determinate astrazioni possa semplificare il lavoro del programmatore.

In un certo senso, i due approcci "colmano"l'*Abstraction Gap*, gap di astrazione che intercorre dai requisiti del problema alla macchina computazionale base, in maniera differente. Il modello ad agenti fornisce un supporto tale che, rispetto a quello ad eventi, riduce la logica applicativa necessaria da implementare per soddisfare i requisiti (figura 7.7). Nonostante ad una prima occhiata, il modello ad agenti parrebbe essere avvantaggiato su quello ad eventi, non è detto che questa complessità aggiunta, nella pratica, possa portare degli effettivi benefici a livello implementativo. Ed è ciò che verrà analizzato nel capitolo successivo.

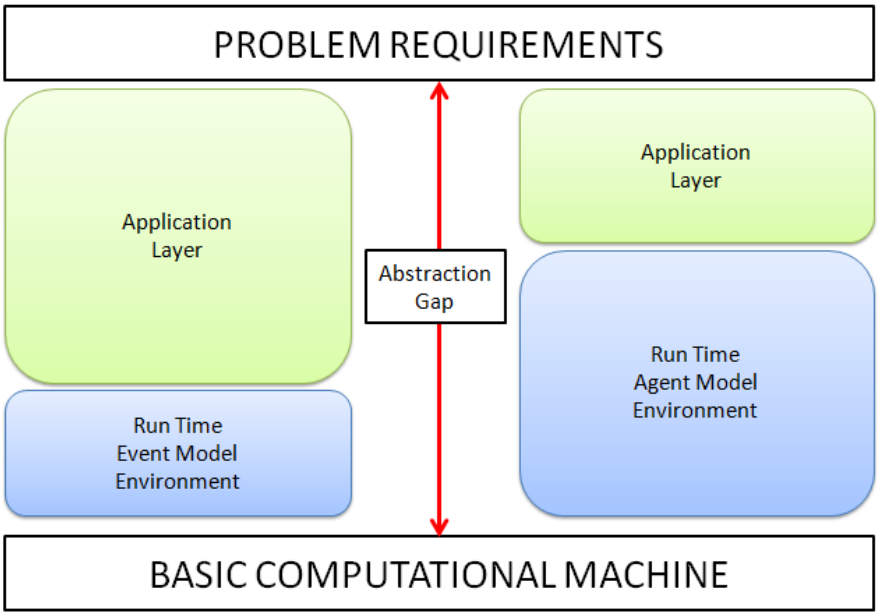

Figura 7.7: Rappresentazione di come viene colmato l'Abstraction Gap seguendo i due approcci, ad Eventi e ad Agenti

Capitolo 8

Casi di Studio

L'analisi fatta fino a questo capitolo ha portato a confrontare due cicli di controllo, l'Event Loop ed il Control Loop, legati rispettivamente al modello ad eventi e ad agenti

Per quanto riguarda il modello ad eventi, sono stati presi in considerazione due linguaggi di programmazione come rappresentanti, JavScript e Dart. Il primo è attualmente lo standard de facto nello sviluppo di Web Applications mentre il secondo è un'alternativa, sviluppata e promossa da Google, che vorrebbe proporsi come il nuovo punto di riferimento per il Web. Per il modello ad agenti, è stato scelto il framework JaCaMo, che fornisce astrazione per la rappresentazione di entità autonome proattive e reattive, gli agenti, e di elementi puramente passivi, gli artefatti.

Il confronto fatto nella sezione precedente è stata puramente teorico, nel senso che si sono analizzati gli step che caratterizzano i due cicli e sono state avanzate delle ipotesi ragionevoli a riguardo (capitolo 7). Arrivati a questo punto, quali sono le implicazioni pratiche nei due approcci? Come si traducono queste differenze nei due cicli di controllo quando si tratta di progettare e programmare Web Applications?

Lo scopo di questo capitolo è proprio quello di presentare, descrivere ed implementare tre diverse tipologie di applicazioni, ognuna delle quali con caratteristiche peculiari.

La prima applicazione si tratta un piccolo gioco interattivo dove l'utente è chiamato ad indovinare una serie di numeri estratti; come primo esempio si è volutamente scelto qualcosa di semplice e con caratteristiche prettamente

reattive, che non si discostasse troppo dalle tradizionali Web Applications.

La seconda concerne l'implementazione di una lavagna distribuita dove più utenti possono partecipare contemporaneamente ad una sessione di disegno; le principali difficoltà risiederanno nell'aspetto distribuito del programma, con gli utenti sparsi tra i vari nodi della rete.

La terza ed ultima applicazione riguarda l'implementazione del noto problema de *I Filosofi a Cena*; le difficoltà risiederanno nel come implementare il comportamento proattivo e reattivo dei filosofi, e nella gestione di risorse condivise da più entità.

Per ognuno di questi casi di studio, verranno fornite due implementazioni, una in Dart e l'altra in JaCaMo, sulle quali si effettueranno delle considerazioni di progettazione ed implementazione adottando un modello rispetto che l'altro.

NOTA: al contrario di JaCaMo che è una tecnologia piuttosto stabile e consolidata, Dart è in continua evoluzione. Nuovi nuove librerie, tecnologie e concetti si aggiungono continuamente all'ecosistema sviluppato da Google, in parallelo quelli preesistenti vengono aggiornati o cambiati, alla luce di nuove considerazioni. Spesso col fine di eliminare codice boilerplate, altre volte apportano modifiche più significative, questi aggiornamenti variano l'approccio del programmatore nel risolvere il problema.

Al momento in cui viene scritto questo lavoro, in Dart sono apparse due keyword per il supporto alle funzioni asincrone: *async* e *await*[22].

La prima serve per definire una funzione che richiama immediatamente una Future. Se prima quindi si poteva scrivere nel seguente modo:

```
foo() => new Future(() => 42);
```

ora può essere scritta come segue:

```
foo() async => 42;
```

L'espressione *await* serve invece per poter scrivere codice asincrono quasi come se fosse sincrono. Se prima, ad esempio, per poter copiare un file in una nuova locazione, proprio perché la funzione di I/O *copy* è asincrona, bisognava scrivere:

```
myFile.copy(newPath).then((f) => f.path == newPath);
```

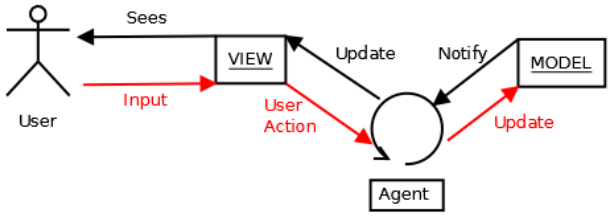

Figura 8.1: Il pattern MVC utilizzando una tecnologia ad agenti

ora può essere semplicemente scritto:

```
(await myFile.copy(newPath)).path == newPath;
```

Questi due esempi sono semplicemente zucchero sintattico, tuttavia in futuro altre espressioni potrebbero essere introdotte, rendendo preferibile la scrittura del codice in un certo modo rispetto che un altro.

Il pattern Model-View-Controller

Come già stato ampiamente detto, le Web Applications sono tipicamente applicazioni con un alto grado di interattività con l'utente che sarà in grado di operare mediante degli opportuni comandi disposti su di una GUI. Uno dei pattern architetturali più importanti riguardante proprio la progettazione di un applicazione dove è necessario separare la vista, la business logic e la data logic (richiamando il principio della **separazione dei contesti**) è il Model-View-Controller (o abbreviato MVC).

In JavScript è molto difficile programmare, in maniera modulare, codice riutilizzabile e scalabile. Per via della natura del linguaggio stesso, è molto facile violare le best practice per la scrittura di buon codice mantenibile. In Dart questo problema è molto più mitigato, anche se spesso ci si deve appoggiare su framework sviluppati ad hoc.

In JaCaMo risulta molto naturale l'adozione del sopracitato pattern strutturale, sfruttando l'agente come il Controller e implementando la View e il Model con due artefatti (8.1). In un modello di questo tipo è possibile sfruttare appieno i Segnali e le Proprietà Osservabili, ad esempio:

- gli eventi della GUI possono essere modellati come Segnali o sfruttando la caratteristica delle Proprietà Osservabili di generare un evento

in caso di cambiamento; il primo caso potrebbe essere adatto alla pressione di un tasto mentre il secondo potrebbe essere utilizzato per mappare la posizione del mouse (le coordinate x,y) all'interno della finestra.

- i dati del Model possono essere facilmente modellati come Proprietà Osservabili, gli aggiornamenti verranno notificati automaticamente a tutti gli agenti interessati.

Il motivo per cui è stato dedicato un paragrafo a questa parte è che, in praticamente tutti i test-case, si dovrà implementare un qualche tipo di interfaccia grafica, definire una certa base di dati (implementate sfruttando gli Artefatti) e sfruttare in qualche modo un entità reattiva-proattiva (implementate sfruttando gli Agenti) che coordini gli eventi generati dalle due parti precedenti.

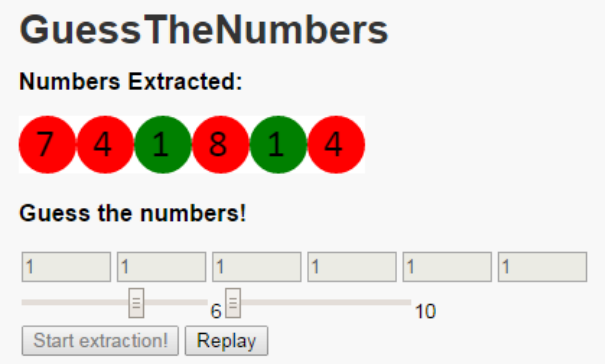

Figura 8.2: Una rappresentazione del gioco "indovina il numero"

8.1 Indovina il Numero

8.1.1 Descrizione

Si vuole realizzare una Web Applications a singola pagina, simile ad un estrazione del lotto, dove l'utente dovrà indovinare i numeri che verranno estratti. All'inizio del gioco, tramite apposita interfaccia grafica, si potrà scegliere il numero di estrazioni da effettuare, il range di valori che potranno essere estratti e i numeri che si pensa verranno estratti. Una volta che l'estrazione verrà avviata, l'utente non avrà più modo di cambiare ne il numero di estrazioni, ne il range di valori, tuttavia finché un numero in una data posizione non sarà estratto, si potrà cambiare la decisione del numero che si pensa verrà estratto. I numeri estratti dovranno essere visualizzati a schermo, evidenziando quali sono stati indovinati e quali no. Una volta terminate le estrazioni, l'utente potrà procedere con un'altra sessione di gioco.

8.1.2 Analisi

In questa semplice applicazione, l'utente ha modo di interagire settando i due parametri di configurazione, il numero delle estrazioni e il range di valori, e scegliendo i numeri che secondo lui verranno estratti (8.3).

Si ricorda che una volta iniziate le estrazioni dei numeri, non dovrà essere più possibile cambiare i parametri di configurazione fino alla successiva

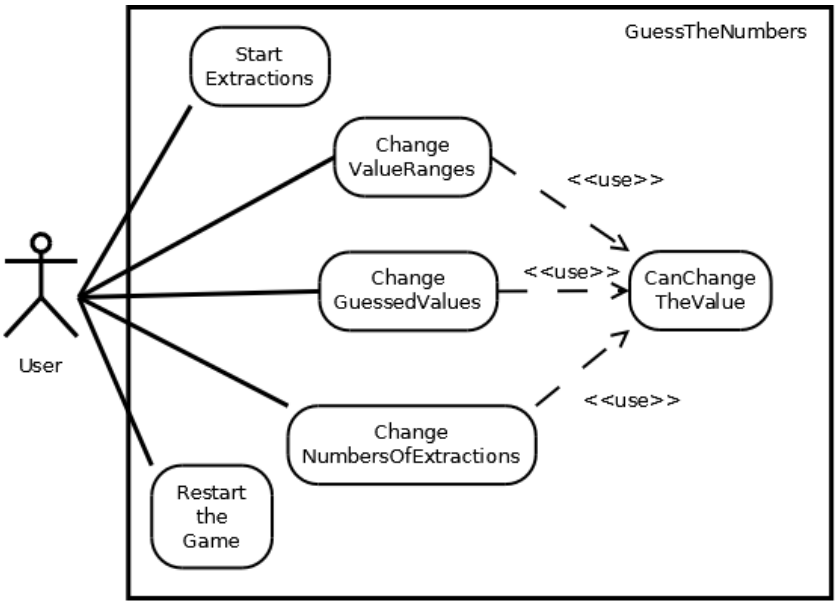

Figura 8.3: Operazioni disponibili per l'utente

estrazione (8.4). I vari numeri delle estrazioni invece dovranno poter essere scelti fintanto che il numero non sarà estratto (8.5). Una volta terminate le estrazioni, l'utente potrà ricominciare con una nuova sessione, ritornando allo stato di partenza.

I valori in gioco sono quindi:

- *numberOfExtractions*: valore **naturale**, **intero** non negativo, che indica il numero di estrazioni da effettuare. È ragionevole pensare che il valore minimo sia 1.

- *valueRanges*: valore **naturale**, **intero** non negativo, (non vi sono specifiche a riguardo, tuttavia è difficile pensare ad estrazioni di valori negativi o addirittura in virgola mobile) che indica il range dei valori che possono essere estratti. È ragionevole pensare che il valore minimo sia 2 (se fosse 1, il gioco risulterebbe banale). Per il massimo valore, nessuna specifica.

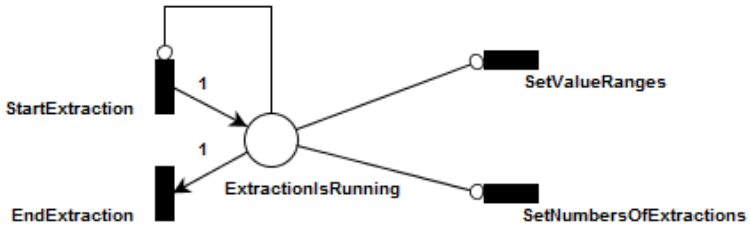

Figura 8.4: Vincoli sui parametri di configurazione evidenziati mediante Rete di Petri

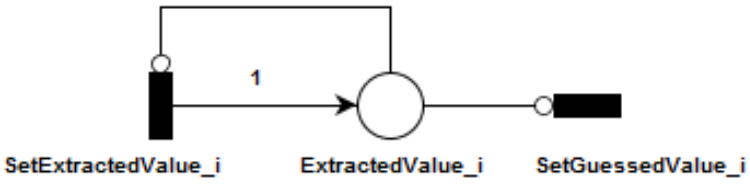

Figura 8.5: Vincolo sull'i-esimo valore estratto, evidenziato mediante Rete di Petri

- *guessedValues_i*: valore i-esimo che indica la predizione dell'utente sull'estrazione di un numero. Il numero dei *guessedValues* dipende dal valore *numberOfExtractions* mentre i valori possibili selezionabili dall'utente dipendono dal valore *valueRanges*.

- *extractedValues_i*: rappresenta il valore i-esimo estratto dal sistema ad estrazioni avviate. Il numero dei *extractedValues* dipende dal valore *numberOfExtractions* mentre i valori possibili estraibili dipendono dal valore *valueRanges*.

In questo caso, l'applicazione non ha particolari componenti attive anzi, si può dire che è quasi totalmente passiva agli input dell'utente, a meno della scelta dei numeri da estrarre che, possibilmente, verrà effettuata con un generatore di numeri pseudo-random.

8.1.3 Approccio con Dart

Dart è un linguaggio che permette di affrontare un problema con molti approcci. Per questo tipo di applicazione, si è deciso di implementare semplicemente uno script con il quale è possibile creare i vari elementi di input necessari, associandogli il corrispettivo handler.

Riprendendo il diagramma dei casi d'uso mostrato in figura 8.3 e quanto detto nell'analisi precedente, avremo sicuramente:

- Un comando di input per decidere il valore di *numberOfExtractions*, chiamati *setGuessedValues*;

- Un comando di input per decidere il valore di *valueRanges*, chiamato *setValueRanges*.

In più avremo:

- Un numero di comandi di input pari al valore *numberOfExtractions*, chiamati *setNumberOfExtractions* che permettono di modificare i valori *guessedValues*;

- Un numero di elementi di output pari al valore *numberOfExtractions*, chiamati *extractedValues*.

Infine serviranno due comandi *startExtractions* e *restartTheGame*, possibilmente dei bottoni, che permettono rispettivamente di avviare le estrazioni e di ricominciare il gioco. Si possono rispettare i vincoli definiti dai requisiti semplicemente abilitando e disabilitando i vari comandi, a seconda dello stato dell'applicazione.

Ad esempio, analizzando uno snippet preso dal main:

```
main() {
  ...

  startExtractions.onClick.listen((_) => startTheGame
     ());
  restartTheGame
       ..disabled = true
       ..onClick.listen((_) => reset());

  initializeOptions();
  initializeInput();

  //Update graphics with scheduleMicrotask
  scheduleMicrotask(updateGraphics);
}
```

startExtractions e *restartTheGame* sono i due comandi di input. Immediatamente si disabilita il secondo e si lascia abilitato il primo; in questo modo si mantiene una coerenza nell'esecuzione dell'applicazione (per cominciare un nuovo gioco dev'essere terminata la sessione precedente). Ad entrambi i bottoni, a seguito dell'evento *onClick* si registrano due funzioni, rispettivamente *startTheGame* e *reset* che analizzeremo in seguito, anche se intuitivamente è facile capire a cosa servono.

Si noti come, per poter mantenere consistente la grafica con i dati, si è deciso di utilizzare il costrutto *scheduleMicrotask* che sfrutta la Microtask Queue di Dart.

La funzione *initializeOptions* che viene richiamata nel main, inizializza le componenti di input *setNumberOfExtractions* e *setValueRanges*:

```
initializeOptions() {
  /*
   * Creates the element for the optionParamenters.
```

```
    * Two sliders for setting numberOfExtractions and
        valueRange values.
    */
  setNumberOfExtractions = new InputElement()
      ..type = "range"
      ..min = "$minNumber"
      ..max = "$maxNumber"
      ..value = "$minNumber"
      ..onInput.listen((_) {
        outputForNumbersOfExtractions.value =
            setNumberOfExtractions.value;
        scheduleMicrotask(
            updateNumberOfExtractionsValue);
        scheduleMicrotask(updateGraphics);
      });

  setValueRanges = new InputElement()
      ..type = "range"
      ..min = "${step*minNumber}"
      ..max = "${step*maxNumber}"
      ..step = "$step"
      ..value = "${step*minNumber}"
      ..onInput.listen((_) {
        outputForValueRanges.value = setValueRanges.
            value;
        scheduleMicrotask(updateValueRangesValue);
        scheduleMicrotask(updateGraphics);
      });
  /*
    * Appending the new created elements in the DOM.
    */
  options.append(setNumberOfExtractions);
  options.append(setValueRanges);
}
```

mentre la funzione *initializeInput* inizializza i *guessedValues*:

```
initializeInput() {
```

```
  guessedValues.nodes.clear();
  guessedValues.nodes.add(new InputElement()
      ..type = "number"
      ..value = "$minNumber"
      ..min = "1"
      ..max = setValueRanges.value);

  extractingValues.clear();
  extractingValues.add("?");
}
```

Si ricorda che dai valori settati mediante comandi di input *setNumberOfExtractions* e *setValueRanges* dipende il numero di elementi *guessedValues*. Infatti, ogni qual volta cambia il valore, vengono eseguite una serie di operazioni che aggiornano i *guessedValues*, mediante le funzioni *updateNumberOfExtractionsValue* e *updateValueRangesValue*, e la grafica. Anche in questo caso si frutta la Microtask Queue per assicurarsi che i valori della grafica corrispondano a quelli settati.

Per completezza, si mostra il contenuto delle due funzioni *updateNumberOfExtractionsValue* e *updateValueRangesValue*, che devono aggiungere o rimuovere elementi *guessedValues* nel primo caso, aggiornare il range dei valori nel secondo caso:

```
updateNumberOfExtractionsValue() {
  int newValue = int.parse(setNumberOfExtractions.
     value);
  int diff = newValue - guessedValues.nodes.length;

  if (diff > 0) {
    // Have to add some input number elements.
    for (int i = 0; i < diff; i++) {
      guessedValues.nodes.add(new InputElement()
          ..type = "number"
          ..value = "$minNumber"
          ..min = "1"
          ..max = setValueRanges.value);
      extractingValues.add("?");
    }
```

```
  } else {
    // Have to remove some input number elements.
    diff = -1 * diff;
    for (int i = 0; i < diff; i++) {
      guessedValues.nodes.removeLast();
      extractingValues.removeLast();
    }
  }
}

updateValueRangesValue() {
  for (Node el in guessedValues.nodes) {
    if (el is InputElement) {
      if (int.parse(el.value) > int.parse(
          setValueRanges.value)) el.value =
          setValueRanges.value;
      el.max = setValueRanges.value;
    }
  }
}
```

La parte più interessante risulta sicuramente la funzione che comincia le estrazioni, *startTheGame*, dov'è possibile vedere con mano come si possono utilizzare le Futures.

Il corpo della funzione è il seguente:

```
startTheGame() {
  // Disable the inputs.
  startExtractions.disabled = true;
  setNumberOfExtractions.disabled = true;
  setValueRanges.disabled = true;

  List<Future<int>> futureElements = new List<Future<
      int>>();

  // Start the extractions.
  for (int i = 0; i < guessedValues.nodes.length; i++)
      {
```

```
    Future<int> fi = getFutureNumber().then((int
        valueExtracted) {
      guessedValues.nodes[i].disabled = true;
      extractingValues[i] = "$valueExtracted";
      print('ValueExtracted:$valueExtracted');
      scheduleMicrotask(updateGraphics);
    });

    futureElements.add(fi);
  }

  // Wait for all the numbers extracted.
  Future.wait(futureElements).then((_) {
    print("ExtractionsConcluded!");
    restartTheGame.disabled = false;
    scheduleMicrotask(updateGraphics);
  });
}
```

Come prima cosa, vengono disabilitati certi comandi di input, per assicurare i vincoli definiti in fase di analisi. Dopodiché vengono avviate tutte le estrazioni. La funzione *getFutureNumber* serve proprio ad estrarre i valori "in futuro", il tipo di oggetto che viene restituito è infatti una Futures¡int¿. Una volta estratto il corrispettivo i-esimo valore, si disabilita la possibilità di modificare il *guessedValues_i*, si setta il valore di *extractedValues_i* e si aggiorna la grafica. Si noti come l'handler di gestione dell'estrazione mantenga il valore della variabile *i*: questo meccanismo si chiama *Closure*. *getFutureNumber* ritorna degli oggetti Futures¡int¿ che possono essere memorizzati in una lista *futureElements*. Sfruttando la funzione *Future.wait* è possibile aspettare che tutte le Futures abbiano un valore effettivo; una volta che si verifica questa condizione, che significa che sono state eseguite tutte le estrazioni, si può terminare il gioco.

8.1.4 Approccio con JaCaMo

Riprendendo lo schema illustrato in figura 8.1, possiamo definire l'architettura come segue:

- *Artefatto View*: rappresenta la vista dell'utente, attraverso il quale può modificare i vari parametri, è possibile modellarla sfruttando un artefatto che genera eventi percepiti da un agente che si preoccuperà di gestirli in maniera appropriata.

- *Agente Controller*: l'agente che fa da ponte fra la View e il Model, cattura gli eventi generati dalle due componenti e si preoccupa di tenerle consistenti.

- *Artefatto Model* (o *Artefatto NumberBoard*): definisce la struttura dati atta a gestire le varie estrazioni dei numeri. Fornisce i metodi necessari all'agente Controller per poter modificare i vari parametri di generazione dei numeri e i valori indovinati dall'utente.

In questo schema è possibile aggiungere un ulteriore agente, che verrà chiamato *Agente Extractor*, il cui compito sarà quello di creare l'*Artefatto Model* con i vari parametri di default, e di estrarre i valori casualmente qualora l'utente decidesse di iniziare le estrazioni.

Una rappresentazione semplificata viene data in figura 8.6, dove si evidenziano le varie componenti sopra descritte e la distinzione fra i metodi richiamabili dall'agente Controller e dall'agente Extractor. I dati del modello possono essere rappresentati mediante *proprietà osservabili* o utilizzando valori interni all'artefatto NumberBoard a cui sono associati dei *segnali*, in caso di modifiche.

In questo particolare caso di studio, l'agente non si dovrà fare altro che preoccuparsi di fare da ponte fra il modello e la view, reagendo agli eventi di una delle due parti e inoltrando le informazioni all'altra. L'Agente Controller agirà quindi in corrispondenza di aggiornamenti nella propria Belief Base o nella ricezione di segnali, ad esempio:

```
// Set the numberOfExtractions and valueRanges plan.
+numbers(Value) : true
<- .print("Updated(numbersOfExtractions)=",Value);
        setNumbers(Value).

+valueRange(Value) : true
<- .print("Updated(valueRanges)=",Value);
        setValueRanges(Value).
```

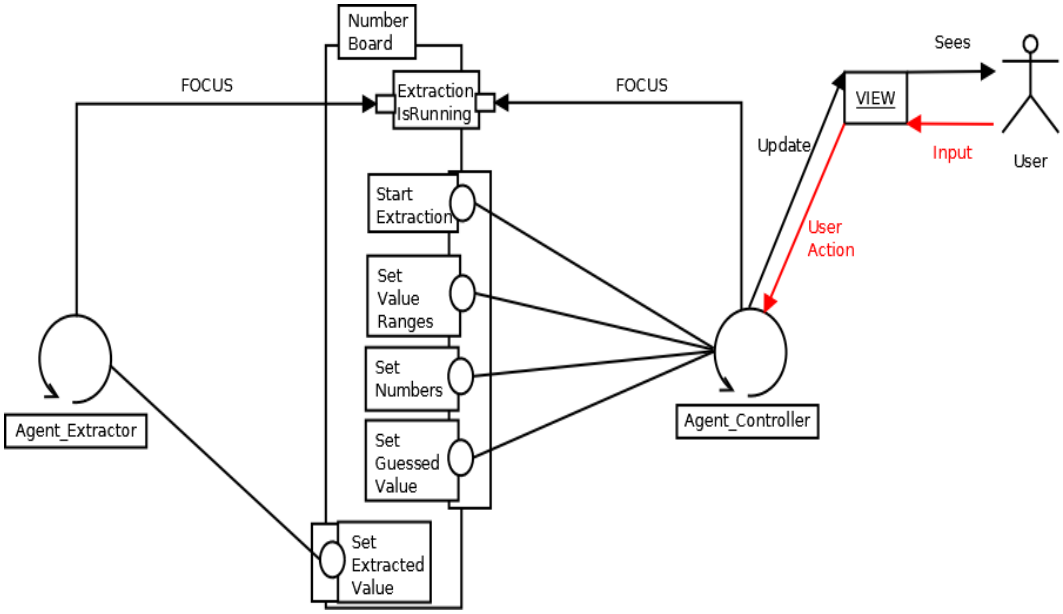

Figura 8.6: Schema semplificato riguardante l'interazione fra i vari componenti artefatti-agenti nel problema *Indovina il Numero*

```
// During extractions plan to handle the extracted
    values.
+extracted_a_number(Index, Value, Result) : Result ==
    true
<-      .print("GUESSED_RIGHT_THE_NUMBER:", (Index+1),
    "!!!");
        enableSliderAtIndex(Index,false);
        updateExtractedValueAtIndex(Index, Value,
            Result).

+extracted_a_number(Index, Value, Result) : Result ==
    false
<-      .print("GUESSED_WRONG_THE_NUMBER:", (Index+1),
    "!!!");
        enableSliderAtIndex(Index,false);
        updateExtractedValueAtIndex(Index, Value,
            Result).
```

sono i piani per gestire il cambio di valore di *numberOfExtractions* e di *valueRanges*, e per gestire la percezione di un numero che è stato estratto.

Le altre parti non richiedono un particolare approfondimento. Ad esempio l'operazione interna all'Artefatto View può essere espressa come segue, con tutte le altre operazioni implementate analogamente:

```
@INTERNAL_OPERATION
void chooseNumbersChanged(ChangeEvent ev) {
  JSlider source = (JSlider)ev.getSource();

  //In this way, only 1 event is fired
  if(!source.getValueIsAdjusting()) {
     try {
        signal("numbers",frame.getNumbers());
     } catch(Exception ex) {
        ex.printStackTrace();
    }
  }
}
```

In questi contesti più semplici, dove l'agente non ha un vero e proprio compito da svolgere, il dover programmare in un certo modo può risultare piuttosto prolisso. Basti pensare che l'Agent Controller, nei fatti, potrebbe benissimo essere totalmente rimpiazzato sfruttando le *@LINKED_OPERATION*, distribuendo la logica di controllo nei due Artefatti Model e View.

Riprendendo l'operazione di aggiornamento del valore *numberOfExtractions*, potrebbe essere benissimo implementata come segue:

```
@INTERNAL_OPERATION
void chooseNumbersChanged(ChangeEvent ev) {
  JSlider source = (JSlider)ev.getSource();

  //In this way, only 1 event is fired
  if(!source.getValueIsAdjusting()) {
     try {
        execLinkedOp("number_board","setNumbers",frame
           .getNumbers());
     } catch(Exception ex) {
        ex.printStackTrace();
    }
  }
}
```

e nell'artefatto Model:

```
void setNumbers(int value) {
  if(value <= _maxNumbers && value >= _minNumbers) {
    ObsProperty prop = getObsProperty("numbers");
        int oldVal = prop.intValue();
        int dif = value - oldVal;

        if(dif != 0) {
          if (dif < 0) {
                // Remove elements
            for (int i = 0; i < Math.abs(dif); i++)
                guessedValues.pop();
          }
          else if (dif > 0) {
```

```
                    // Update all elements with current
                       value range
                    for (int i = 0; i < dif; i++)
                            guessedValues.push(1);
                }
                prop.updateValue(value);
            }
    }
    else {
     failed("OutOfBoundsNumbers", "out_of_bounds_numbers
        ", "min_and_max_value", _minNumbers, _maxNumbers
        );
    }
}
```

8.1.5 Confronto

Questo primo esempio è stato scelto volutamente semplice per cercare di capire come si comportano i due linguaggi, con le rispettive architetture, con applicativi di complessità non elevata.

Con Dart è stato possibile passare velocemente dalla fase di analisi/-progettazione a quella implementativa vera e propria, favorendo un approccio alla programmazione molto agile, e il modello ad eventi ben si adatta a questa tipologia di problemi dove si ha un comportamento per lo più reattivo.

In JaCaMo si è riusciti a soddisfare i requisiti e probabilmente, vista anche l'introduzione delle due astrazioni Agenti ed Artefatti, il progettista è anche "forzato"a pensare il problema in un certo modo, evidenziando e mantenendo ben separati i contesti d'interesse. Anche se l'applicazione non ha aspetti proattivi, l'Agente può comunque agire da entità puramente reattiva; il suo Control Loop gli permette di ricoprire questo ruolo. Certo è che si perde buona parte del suo potenziale e sicuramente un ciclo di controllo come l'Event Loop risulta più efficiente. Anche dal punto di vista prettamente implementativo, in JaCaMo il tutto è risultato più complicato e macchinoso rispetto all'immediatezza fornita con Dart. Questo è dovuto anche dagli strumenti di sviluppo della piattaforma JaCaMo che non aiutano in questo senso il programmatore. Un esempio banale riguarda il fatto

che non è possibile fare il refactoring di un nome di una operazione in un Artefatto; ogni volta bisogna andare a ricontrollare ogni entità che usa quella data operazione e rinominarla con il nuovo nome.

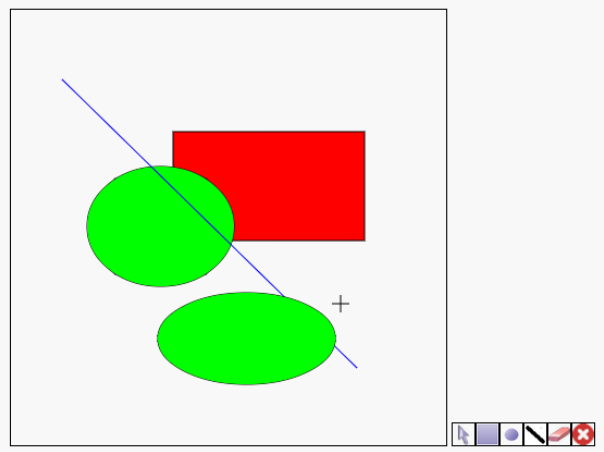

Figura 8.7: Una rappresentazione di una lavagna distribuita

8.2 Lavagna Distribuita

8.2.1 Descrizione

Si vuole realizzare una lavagna distribuita, ovvero un applicazione di disegno collaborativo dove più di un utente può partecipare. Si dovranno fornire alcune funzionalità di base, come il disegno di figure geometriche o la possibilità di cancellarle. Gli utenti che partecipano ad una sessione dovranno poter cambiare lo stato della lavagna in maniera concorrente. L'applicazione dovrà essere distribuita sul Web e gli utenti potranno essere dislocati in nodi distinti.

8.2.2 Analisi

Tipico esempio di problema client-server dove la lavagna distribuita viene gestita sul nodo centrale (il server) al quale si connettono tutti gli utenti che vogliono partecipare alla sessione (i client) (8.8).

In questa architettura:

- il Server si preoccupa di gestire le connessioni con i vari client e le interazioni, nonché di mantenere i dati consistenti e di aggiornare gli utenti con le modifiche avvenute;

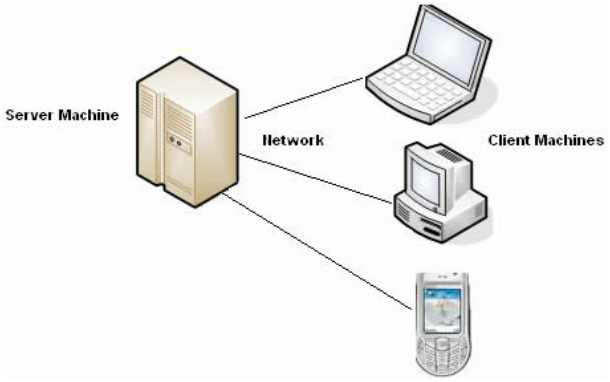

Figura 8.8: Rappresentazione di un architettura client-server, dove i client sono dispositivi eterogenei

- i Client sono gli utenti veri e propri che partecipano e interagiscono mediante il mezzo condiviso, la lavagna distribuita.

In un problema distribuito di questo tipo, sono tre i principali modelli da definire: modello di interazione, modello di gestione delle failure e il modello di sicurezza. In quest'analisi, e nelle parti successive, ci si preoccuperà principalmente del primo dei tre, limitandosi a fornire un supporto base per gestire le failure e gli accessi alla lavagna distribuita.

Una volta connesso alla lavagna distribuita, l'utente avrà svariati modi per interagire con essa; definiamo alcune funzioni di base disponibili:

1. disegnare figure (linee, rettangoli ed ellissi);
2. selezionare e spostare le figure disegnate;
3. eliminare una o più figure presenti;

assieme a queste, le funzioni di base per connettersi e disconnettersi completano il set di metodi utilizzabili dai clienti (8.9).

Come vengono gestite le interazioni dunque? Si può decidere di adottare una delle seguenti vie.

La prima permette ai vari clienti di eseguire le operazioni in locale e di inviarle, una volta completate, al server. Se queste vengono accettate, tutti

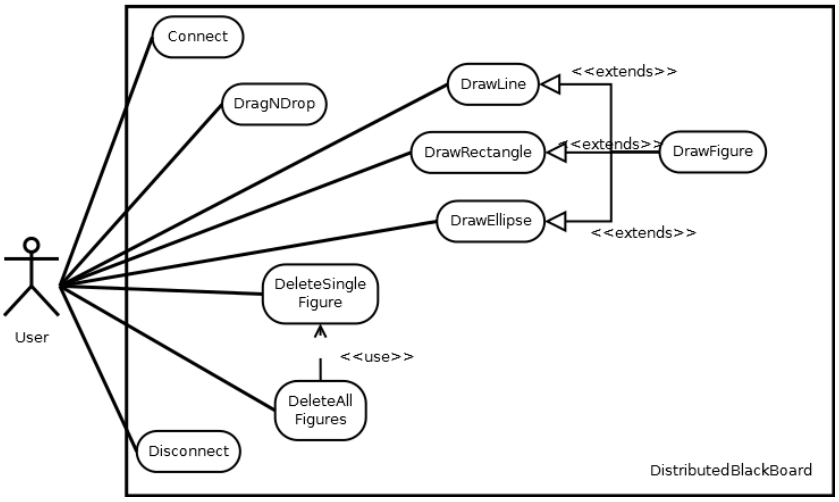

Figura 8.9: Diagramma dei Casi d'Uso disponibili all'Utente

gli utenti connessi (anche quello che ha inviato le modifiche), riceveranno la notifica che la lavagna è stata cambiata e ognuno di essi aggiornerà la propria vista come meglio crede. Questo approccio permette di alleggerire il carico del server, distribuendo parte della logica applicativa sui vari Web Client, tuttavia presenta dei rischi in termini di consistenza dei dati (cosa succede se due user lavorano in locale sullo stesso dato e poi inviano la modifica al server? chi dei due ha la precedenza?) e di future espansioni del modello (operazioni in cui utenti partecipano contemporaneamente, come si gestiscono?). Per chiarezza, si mostra uno schema di una tale modalità in figura 8.10, dove gli Utenti eseguono operazioni di creazione/modifica/cancellazione in maniera concorrente e la lavagna notifica il risultato di quelle operazioni.

La seconda invece alleggerisce totalmente il lato Client (parliamo quindi di Thin-Client), il quale si dovrà semplicemente preoccupare di inviare degli eventi base (ad esempio, click del mouse in un determinato punto del frame) e di gestire la rappresentazione e la visualizzazione dei dati nella GUI. Il suddetto approccio implica che tutte le operazioni, dovranno essere riproducibili mediante una sequenza di azioni base *click-move-release*; il risultato

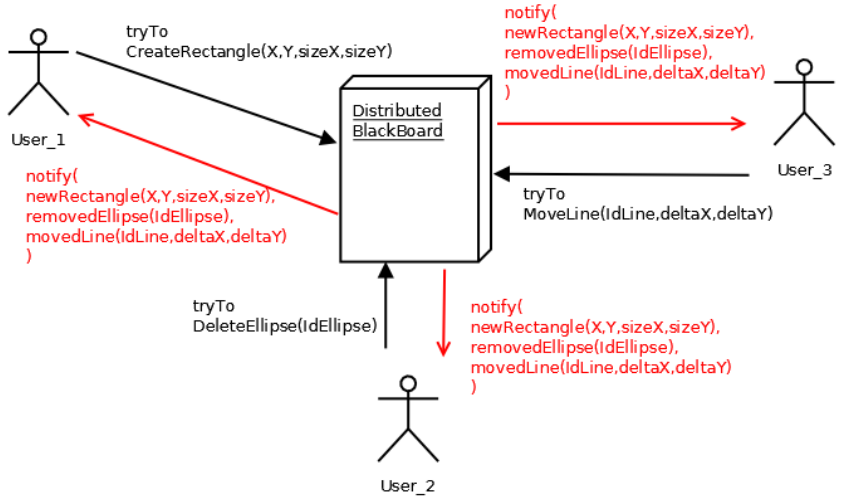

Figura 8.10: Visualizzazione di un modello di lavagna distribuita con i Client che eseguono in locale, parte della logica applicativa e notificano al Server solo l'azione completa. Esso notificherà gli avvenuti cambiamenti a tutti gli Utenti connessi

di questa sequenza dipenderà dunque da uno stato associato ad ogni utente, che per comodità chiameremo *Stato del Puntatore*, che indicherà quale operazione di alto livello si desidera attuare. Il server quindi, non solo avrà il compito di mantenere coerente la struttura dei dati ma dovrà anche gestire le singole azioni dei vari utenti, oltre che il loro stato e notificarli dei cambiamenti avvenuti. Il carico computazionale si sposta tutto sul lato server. Questo implica un modello di gestione della concorrenza fra i vari utenti semplificato, inoltre il client sarà alleggerito della logica applicativa e dovrà semplicemente inviare i singoli eventi e visualizzare i dati. Per chiarezza, si mostra uno schema di una tale modalità in figura 8.11, dove un utente esegue in successione un operazione di *click*, poi *move* ed infine *release*; il risultato di questa sequenza è la creazione di un rettangolo che viene poi notificata a tutti gli Utenti.

In questa prima analisi, e nelle successive implementazioni, si decide di avere come elementi base tre tipologie di figure: linee, rettangoli ed ellissi. È dunque possibile definire un modello di questi dati dove si differenziano gli oggetti logici (utilizzati ad esempio dal server, dove non è richiesta una

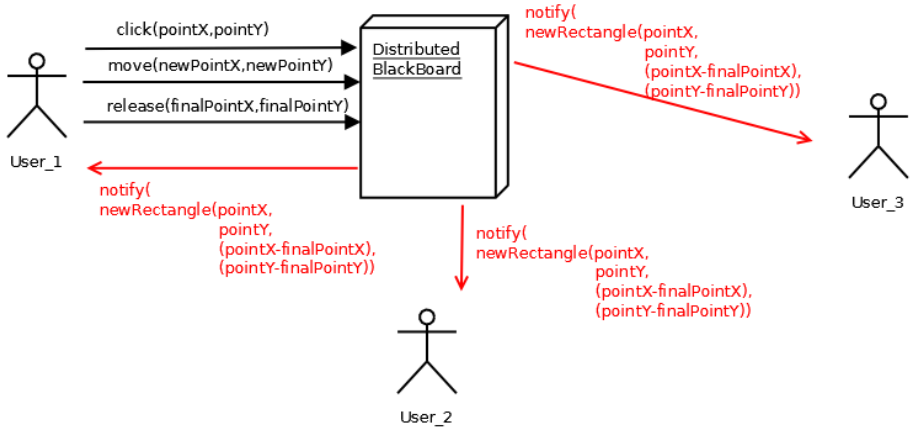

Figura 8.11: Visualizzazione di un modello di lavagna distribuita con i Client che esegue una successioni di operazioni elementari il cui risultato è la modifica dello stato della Lavagna Distribuita

rappresentazione visiva) da quelli rappresentabili sulle GUI (utilizzati dai client dove si devono visualizzare a schermo) (figura 8.12).

8.2.3 Approccio con Dart

In fase di analisi, sono state individuate due entità, il client ed il server. In Dart è possibile programmare entrami questi componenti, utilizzando librerie dedicate.

La parte del server dovrà implementare innanzitutto la logica di gestione delle connessioni dei vari utenti:

```
// For the sake of the example, will be used localhost
    .
final HOST = "127.0.0.1";
final PORT = 8081;

main() {

  runZoned(() {
    HttpServer.bind(HOST, PORT).then((server) {
      print("ServerBinded...");
```

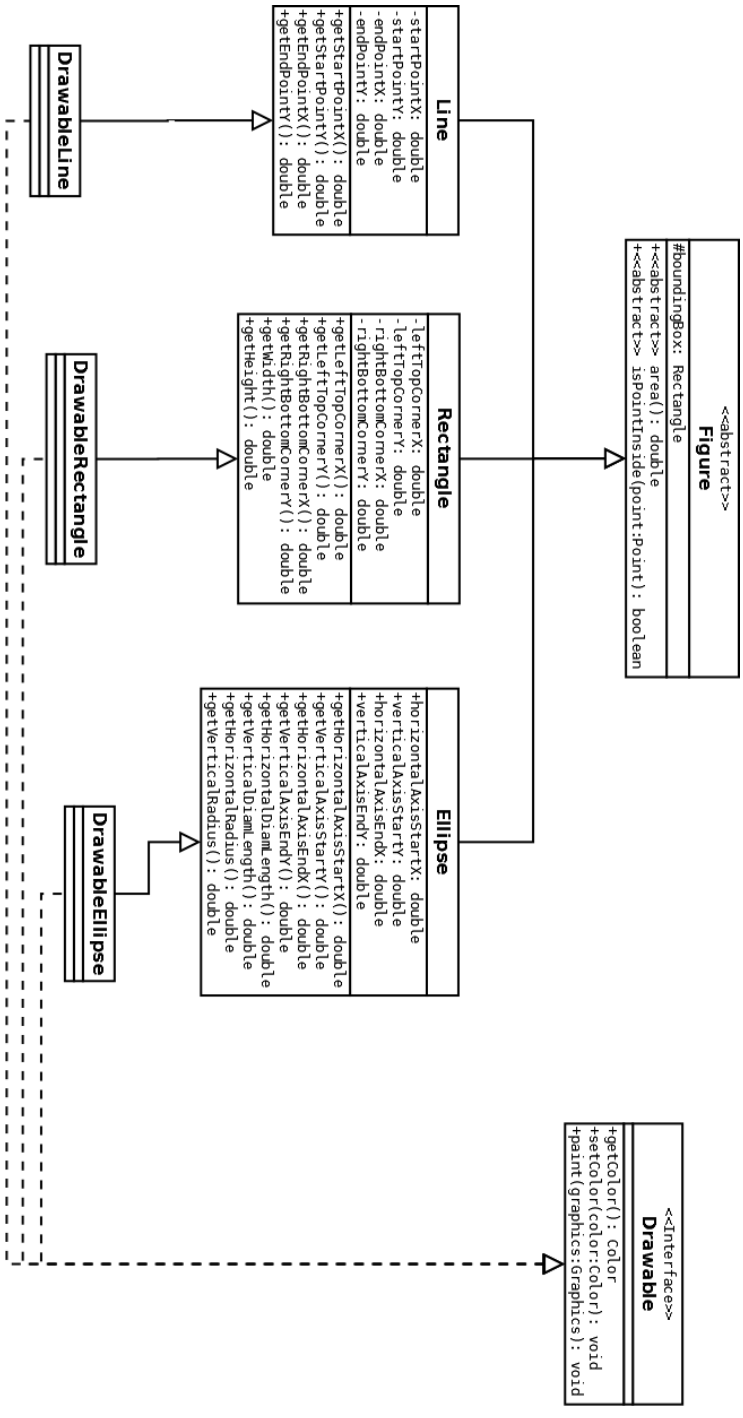

Figura 8.12: Modello UML delle figure rappresentabili dagli utenti

```
      server.listen((HttpRequest req) {
        if (req.uri.path == '/ws') {
          // Upgrade a HttpRequest to a WebSocket
             connection.
          WebSocketTransformer.upgrade(req).then(
             handleSocket);
        }
      });
    });
  }, onError: (e) => print("ERROR:${e.toString()}"));
}
```

Nel codice sopra, il server fa il bind ad una determinata porta e si mette in ascolto. Ogniqualvolta si ricevono delle richieste Http, queste vengono gestite mediante la funzione *handleSocket*, che deve poter gestire la connessione dell'utente alla lavagna (in questo caso, si utilizza una Map che associa una connessione WebSocket ad un id) e registrare un handler che gestisce gli eventi generati da quell'utente:

```
//Association 1-1 socket,id
Map<WebSocket, String> _socketUserMap;

handleSocket(WebSocket s) {
  print("\n--------\nSomeoneIsTryingToConnect...");
  new Future(() => this.connectUser()).then((String id
     ) {
    if (id != null) {
      print("\nAccessAllowed!\n--------\n");
      //Associate the WebSocket to the given Id
      _socketUserMap[s] = id;

      //Activate the periodic timer (can be improved
         to send the updated state only in response of
          an event)
      if (_t == null || !_t.isActive) _t = new Timer.
         periodic(new Duration(milliseconds: 33),
         sendUpdateState);
```

```
      //Set message callback and other functions
         onError and onDone (when socket is closed)
      s.listen((msg) => handleUserEvent(s, msg))
          ..onError((e) {
            String idToRemove = _socketUserMap.remove(
               s);
            if (_socketUserMap.isEmpty) _t.cancel();
          })
          ..onDone(() {
            print("UserQuit!");
            String idToRemove = _socketUserMap.remove(
               s);
            if (_socketUserMap.isEmpty) _t.cancel();
          });
    } else {
      print("\nAccessDenied!\n----------\n");
      s.close(WebSocketStatus.POLICY_VIOLATION, "
         CannotAcceptConnections!");
    }
  });
}
```

Nel codice sopra, si utilizza un Timer per inviare periodicamente lo stato della lavagna. Nel caso non vi siano più utenti connessi, il Timer viene cancellato per poi eventualmente reinizializzarlo quando sarà connesso almeno un utente.

La funzione *sendUpdateState* si preoccuperà di inviare lo stato della lavagna, in formato stringa JSON, a tutti gli utenti connessi.

```
sendUpdateState(Timer t) {
  for (WebSocket s in _socketUserMap.keys) {
    s.add(JSON.encode(this.stateAsJson()));
  }
}
```

Ogni volta che si riceve un messaggio da un utente connesso, viene richiamata la funzione *handleUserEvent* che elabora il messaggio, in formato JSON, e aggiorna lo stato della lavagna:

```
/*
 * Event handler when receive a message from a user
 */
handleUserEvent(WebSocket s, msg) {
  Map data = JSON.decode(msg);

  //First check if the id corresponds to the websocket
  if (_socketUserMap[s] == data['id']) {
    //Different blackboard event handlers for
       different eventType fired by the users
    if (data['eventType'] == "changeState") {
      this.changeUserState(data['id'], data['option'],
          data['suboption']);
    } else if (data['eventType'] == "mouseDown") {
      this.mouseDownEvent(data['id'], new Point(data['
         pointX'], data['pointY']));
    } else if (data['eventType'] == "mouseMove") {
      this.mouseMoveEvent(data['id'], new Point(data['
         pointX'], data['pointY']));
    } else if (data['eventType'] == "mouseUp") {
      this.mouseUpEvent(data['id'], new Point(data['
         pointX'], data['pointY']));
    }
  }
}
```

Come spiegato in fase di analisi, ogni utente avrà associato uno *Stato del Puntatore* e per poterlo cambiare dovrà essere comunque comunicato al server.

La parte del client non sarà particolarmente complessa e dovrà fornire le funzionalità di *connessione*, *invio degli eventi* e *gestione dello stato della lavagna*. Per quanto riguarda la fase di connessione, una possibile implementazione è la seguente:

```
final HOST = "127.0.0.1";
final PORT = 8081;
final SERVER = 'ws://${HOST}:${PORT}/ws';
```

```
main() {

  // Open the connection with the server.
  _ws = new WebSocket(SERVER);

  _ws.onOpen.listen((Event e) {
    setUp(e);
  });

  _onDataSubscriber = _ws.onMessage.listen((
      MessageEvent e) {
    handleServerState(e);
  });

  _ws.onClose.listen((CloseEvent e) {
    close(e);
  });

  _ws.onError.listen((Event e) {
    handleError(e);
  });
}
```

Molto semplicemente, una volta avviato il main Isolate, si associano diversi handler a seconda degli eventi legati alla socket (apertura, chiusura ed errore). In più, con il metodo *onMessage* viene restituito un Stream di MessageEvent, che vengono gestiti con il metodo *handleServerState*.

Durante il *setUp*, vengono registrati gli eventi del mouse e viene richiamato un unico handler *sendEvent* con diversi parametri:

```
setUp(Event e) {
  _onDownSubscriber = _canvas.onMouseDown.listen((
      MouseEvent e) => sendEvent(e, "mouseDown"));
  _onMoveSubscriber = _canvas.onMouseMove.listen((
      MouseEvent e) => sendEvent(e, "mouseMove"));
  _onUpSubscriber = _canvas.onMouseUp.listen((
      MouseEvent e) => sendEvent(e, "mouseUp"));
}
```

La *sendEvent* ottiene gli eventi del mouse ed invia le informazioni al server, sotto forma di stringa JSON:

```
sendEvent(MouseEvent e, String type) {
  //Get mouse point
  Point pg = e.page;
  Rectangle rec = _canvas.getBoundingClientRect();
  Point actualP = new Point(pg.x - rec.left, pg.y -
      rec.top);

  //Set the map to send to the server
  Map mapToSend = {
    'id': _id,
    'eventType': type,
    'pointX': actualP.x,
    'pointY': actualP.y
  };

  //Sending as Json string
  _ws.sendString(JSON.encode(mapToSend));
}
```

L'ultima parte riguarda la gestione dei messaggi di stato della lavagna distribuita. La funzione *handleServerState* non deve far altro che estrapolare tutte le informazioni nel messaggio e rappresentarle sulla GUI. Il fatto che il client ed il server comunichino con messaggi in formato di stringhe, non vincola in alcun modo la scelta della rappresentazione delle figure; in questo esempio si sfrutta il *CanvasRenderingContext2D* dei Web Browser:

```
handleServerState(MessageEvent e) {
  //DEBUG
  //print(e.data);
  Map state = JSON.decode(e.data);

  //Update canvas
  CanvasRenderingContext2D ctx2D = _canvas.context2D;
  ctx2D.clearRect(0, 0, _canvas.width, _canvas.height)
      ;
```

```
  for (Map figure in state['drawn']) {
    drawFigure(ctx2D,figure);
  }

  for (Map figure in state['staging'].values) {
    drawFigure(ctx2D,figure);
  }

  for(Map pointer in state['users'].values) {
    drawCrossPointer(ctx2D, new Point(pointer['
        pointerPositionX'],pointer['pointerPositionY'])
        );
  }
}
```

8.2.4 Approccio con JaCaMo

In JaCaMo, a partire dall'analisi fatta, è possibile identificare le due principali entità come:

- L'*Artefatto Lavagna*;
- L'*Agente Utente*.

Chiaramente, ogni utente potrà avere una GUI-Artefatto modellata come preferisce, tuttavia la logica di controllo e di gestione degli eventi sarà uguale per tutti.

Le Proprietà Osservabili in questo contesto ci tornano molto utili e ci permettono di modellare ogni dato visibile a chiunque (come ad esempio le figure presenti nella lavagna) in maniera naturale e senza dover preoccuparci di inviare l'evento a tutti gli utenti connessi; la modifica della Proprietà Osservabile verrà percepita da tutti gli agenti in automatico.

Adottando quindi un approccio mostrato in figura 8.11, è possibile definire l'artefatto lavagna con i seguenti metodi:

- *Connect* e *Disconnect*: primitive per permettere gli utenti di connettersi alla lavagna distribuita;

- *Click*, *Move* e *Release*: tutti i metodi legati agli eventi base che l'utente può richiamare per poter interagire con la lavagna;

Per quel che riguarda lo *Stato del Puntatore*, analogamente a quanto fatto in Dart, è possibile introdurre un metodo nell'artefatto lavagna *changePointerStatus* che permette agli utenti di cambiare il tipo di operazione che si vuole eseguire.

Il set di Proprietà Osservabili comprenderà quindi:

- *operation*: set di operazioni disponibili all'utente per poter interagire con la lavagna;
- *user*: set di utenti connessi alla lavagna;
- *figure*: set di figure presenti nella lavagna;

Riprendendo ancora una volta lo schema in figura 8.1, è possibile mappare la lavagna distribuita come il Modello, la logica applicativa del client come l'agente Controller, infine la GUI come View. In un contesto reale, avremo un unico Artefatto Lavagna e tanti Agenti Controller, ad ognuno dei quali è associata un Artefatto GUI. Quest'ultima può essere implementata in svariati modi, al contrario la logica dell'agente può essere definita in maniera univoca poiché i compiti che deve assolvere sono uguali per tutti gli utenti connessi (a meno di utenti con particolari diritti d'accesso, ma non è questo il caso).

In figura 8.13 viene mostrata una rappresentazione dell'interazione fra un generico utente con la lavagna distribuita.

A questo punto è possibile definire l'agente con un piano iniziale del tipo:

```
/* Initial goals */

!init.

/* Plans */

+!init : true
<-      ?discover(BBID);
        !tryToConnect;
        !createGUI;
```

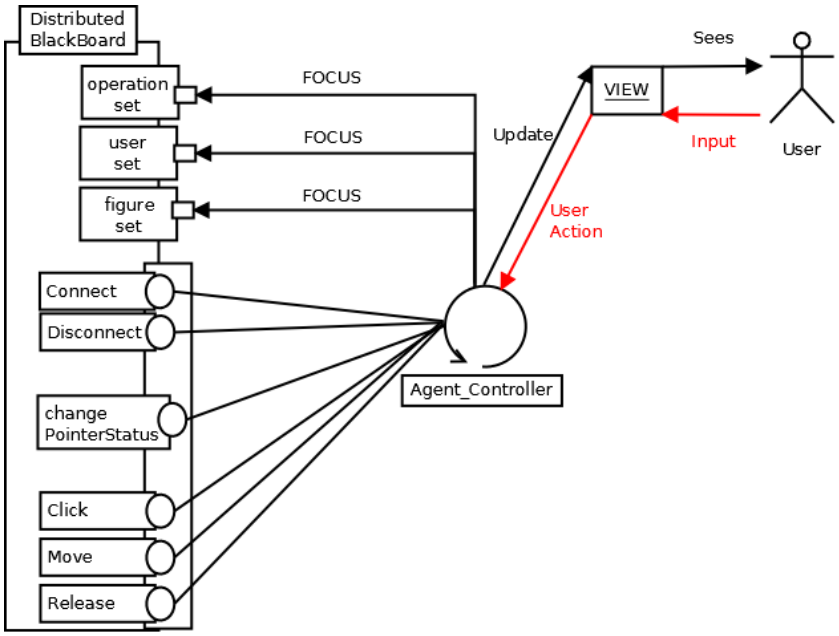

Figura 8.13: Schema rappresentante l'interazione fra l'Artefatto Lavagna e l'Agente Controller

```
        focus(BBID).

//Discover plan
+?discover(BBID) : true
<-      lookupArtifact("BlackBoard", BBID).

-?discover(BBID) : true
<-      .wait(10);
        ?discover(BBID).

//Connecting plan
+!tryToConnect : true
<-      .my_name(Name);
        .concat("",Name,UserName);
        connect(UserName,Id);
        +myIdInBlackBoard(Id,UserName).
```

L'agente cercherà di "scoprire"e di connettersi ad una lavagna distribuita, dopodiché rimarrà in ascolto degli eventi generati mediante il costrutto *focus*.

A questo punto è possibile comunicare i propri eventi e ricevere i cambianti di stato della lavagna come segue:

```
// User Event Sends.
+pointerStatus(Status) : myIdInBlackBoard(Id,Name)
<-      .concat("",Status,NewPointerStatus);
        modifyPointerStatus(Id,Name,NewPointerStatus).

+click(X,Y) : myIdInBlackBoard(Id,Name)
<-      clickEvent(Id,Name,X,Y).
+move(X,Y) : myIdInBlackBoard(Id,Name)
<-      moveEvent(Id,Name,X,Y).
+release(X,Y) : myIdInBlackBoard(Id,Name)
<-      releaseEvent(Id,Name,X,Y).

// User Event Received (Observable Property Update).
+user(UserName,_,WhatDoing,PointX,PointY) : true
<-      updatePointerInView(UserName,PointX,PointY).
```

```
-user(UserName,_,_,_,_) : true
<-        .print("User(", UserName, ")_not_positioned_
    anymore!");
          removePointerInView(UserName).

+figure(Id,Type,Status,StartPointX,StartPointY,
    EndPointX,EndPointY) : true
<-        updateFigureInView(Id,Type,Status,StartPointX,
    StartPointY,EndPointX,EndPointY).
-figure(Id,Type,Status,StartPointX,StartPointY,
    EndPointX,EndPointY) : true
<-        removeFigureInView(Id).
```

Anche in questo caso, come nell'esempio precedente, l'agente si troverà a far da ponte tra la View (la GUI dell'utente) e il Model (la Lavagna Distribuita). La particolarità di questo approccio risiede tuttavia nel come avviene l'interazione fra le parti coinvolte, in un contesto distribuito.

In modo trasparente, un agente A in un workspace X può usare un artefatto B in un workspace Y che risiede su una macchina remota, come se fosse locale. L'unica cosa necessaria che deve fare l'agente è quella di fare il *join* sul workspace dove risiede l'artefatto di interesse, proprio come accade in locale.

La logica e l'implementazione dell'*Artefatto Lavagna* è del tutto analoga a quella in Dart e quindi di poco interesse.

8.2.5 Confronto

Lo scopo di questo esempio è stato quello di mettere a confronto i due modelli principali in un contesto distribuito.

In JaCaMo lo scambio di informazioni avviene in maniere trasparente, semplicemente registrando l'interesse dell'agente a determinate Proprietà Osservabili. Al contrario in Dart si è dovuto definire tutta la parte di connessione fra client e server, con che tipo di messaggi le due parti comunicano, in che modo si deve propagare lo stato della lavagna agli utenti, etc.

In JaCaMo per rappresentare le risorse vengono utilizzati dei nomi logici; questo permette di avere entità anche distribuite nella rete, ma sono tutte *location-transparent*. Che siano in locale o in un nodo qualsiasi della

rete, esse vengono trattate allo stesso modo. Sarà compito dell'architettura sottostante a gestire il come effettuare lo scambio di informazioni. Questo riduce enormemente l'abstraction gap fra i requisiti definiti in fase di progettazione e l'implementazione vera e propria, facilitando lo sviluppo ed il rapido deployment di sistemi distribuiti.

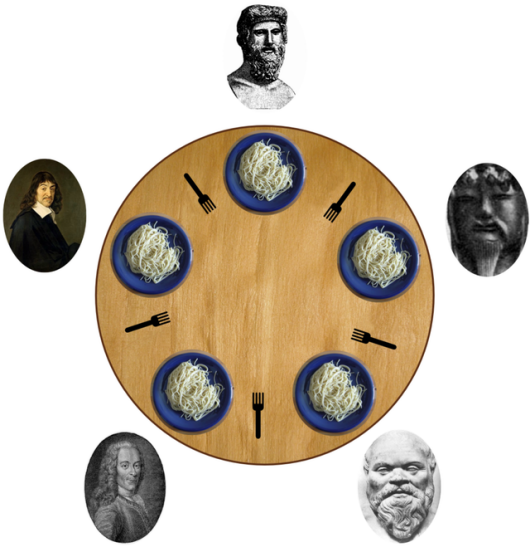

Figura 8.14: La rappresentazione del problema dei filosofi

8.3 I Filosofi a Cena

8.3.1 Descrizione

Si vuole realizzare una Web Applications in grado di simulare il problema de "I Filosofi a Cena". Il suddetto problema è un classico nel campo della programmazione concorrente, formulato da Dijkstra nel 1971 riguardo un problema di sincronizzazione dove cinque computer dovevano competere per accede a cinque risorse condivise. Il problema dice che vi sono cinque filosofi che possono compiere solamente due azioni che vengono ripetute ciclicamente con tempi non determinati: *pensare* e *mangiare*. Questi filosofi sono seduti attorno ad un tavolo davanti a cinque piatti e con cinque forchette; al centro vi è un piatto di spaghetti che viene riempito continuamente. Questi spaghetti sono molto aggrovigliati e un filosofo per mangiare ha bisogno di due forchette. Ogni filosofo può prendere la forchetta alla sua destra e alla sua sinistra. L'applicazione dev'essere in grado di fornire la possibilità di *avviare* la simulazione, metterla in *pausa* (e possibilmente riprenderla) e di *stopparla*, permettendo così l'avvio di una nuova simulazione.

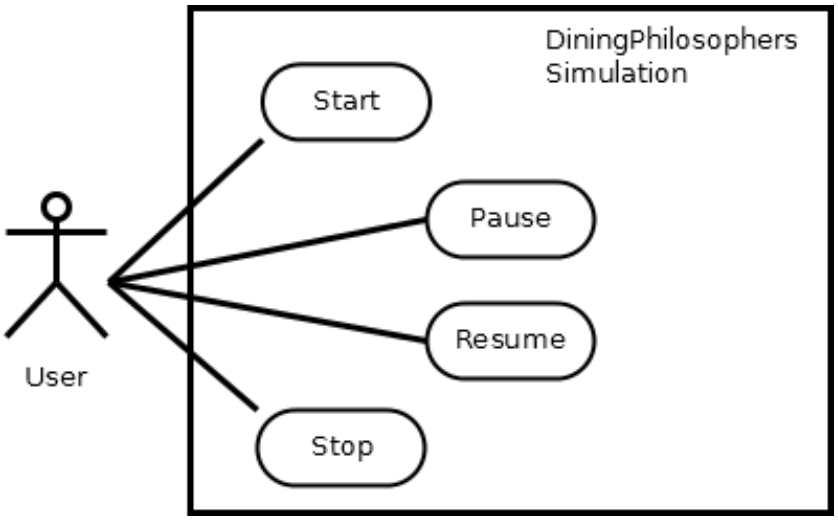

Figura 8.15: Operazioni disponibili per l'utente

8.3.2 Analisi

Dal punto di vista dell'utente, non vi sono molte interazioni che esso può compiere. Le uniche disponibili riguardano l'avvio, l'arresto, la messa in pausa e la ripresa della simulazione. Con solo queste quattro operazioni, l'utente sarà in grado di gestire la progressione della simulazione (8.15).

Entrando nel dettaglio del problema, è possibile dedurre quali sono i due principali concetti all'interno dell'applicazione: i filosofi e le forchette.

I filosofi possono essere visti come le entità autonome attive, il cui unico scopo è quello di perseguire nella loro attività, mangiare o pensare. Al contrario, le forchette sono delle risorse puramente passive, vengono semplicemente utilizzate dai filosofi per mangiare dal piatto di spaghetti.

Il problema è affrontabile seguendo due approcci di alto livello:

1. Approccio Centralizzato (8.16): i cinque filosofi accedono ad un unica risorsa *Waiter* che rappresenta tutte e cinque le forchette, e che gestisce le politiche d'accesso evitando possibili deadlock e assicurando fairness.

2. Approccio Distribuito (8.16): ogni forchetta è una risorsa distinta, sono necessarie politiche di coordinamento fra i filosofi per evitare

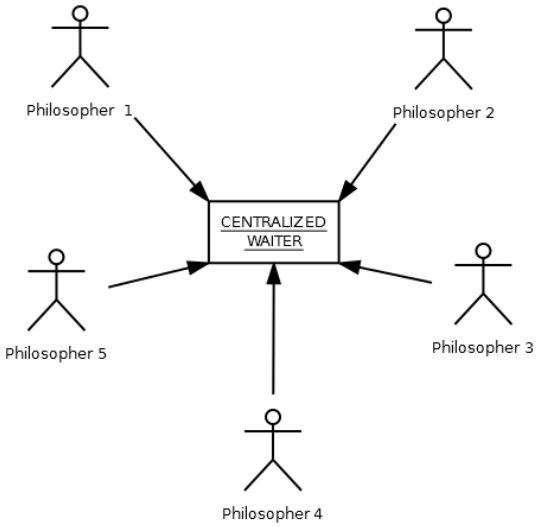

Figura 8.16: Rappresentazione in pseudo-UML del problema dei filosofi, con approccio *Centralizzato*

deadlock.

Per quanto riguarda lo stato delle forchette, è piuttosto facile intuire che esse possono essere o *disponibili* o *in uso* da un altro filosofo. Si mostra una semplice rappresentazione con una Rete di Petri (figura 8.18), dove ragionevolmente le transizioni *take* e *release* saranno attivate dai filosofi (indirettamente o direttamente a seconda se si utilizza un approccio centralizzato o decentralizzato).

Analogamente lo stato dei filosofi può essere facilmente rappresentato con una sequenza di stati dove inizia pensando per un certo periodo di tempo, quando decide di smettere prova a prendere le forchette appena diventano disponibili, quando riesce ad acquisire le forchette mangia per un certo periodo di tempo, una volta terminato di mangiare rilascia le forchette e torna a pensare (figura 8.19).

L'analisi fatta vale fintanto che non vengono considerati possibili eventi esterni. In questo caso però, abbiamo l'utente che può comandare, attraverso le opzioni rese disponibili dall'interfaccia, l'andamento della simulazione.

Una rappresentazione più completa di come lo stato dei filosofi possa variare è mostrata in figura 8.20. Per semplicità e leggibilità del diagram-

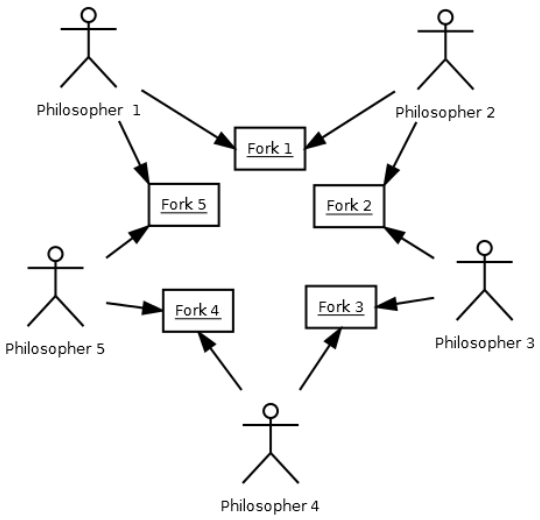

Figura 8.17: Rappresentazione in pseudo-UML del problema dei filosofi, con approccio *Distribuito*

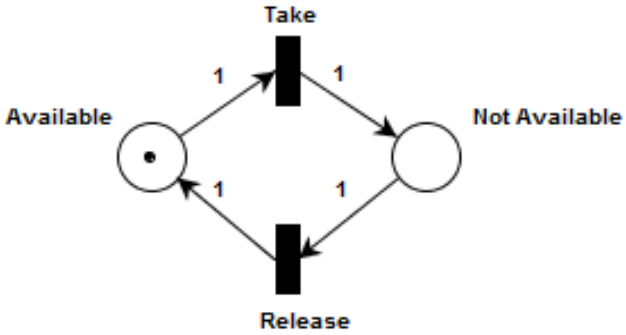

Figura 8.18: Rappresentazione dello stato delle forchette con una Rete di Petri

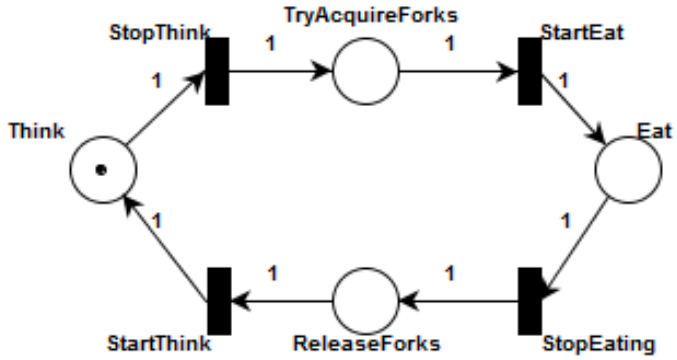

Figura 8.19: Rappresentazione dello stato dei filosofi con una Rete di Petri

ma, si è omessa l'operazione *stop* che dovrà semplicemente ripristinare lo stato iniziale della rete. Le transizioni *start*, *stop*, *pause* e *resume*, sono quelle attivabili dall'utente e modificano il flusso della simulazione dei filosofi. L'unica aggiunta da fare è che all'avvio di una nuova simulazione, le forchette devono essere inizializzate e lo stato di tutte dev'essere *available*, quando si interrompe la simulazione invece si devono de-inizializzare.

8.3.3 Approccio con Dart

In Dart è possibile rappresentare i due concetti principali dell'applicazione, i filosofi e le forchette, rispettivamente come Isolates oggetti. Avendo comunque un main Isolate, che si dovrà preoccupare di aggiornare e gestire gli eventi del DOM, una soluzione centralizzata potrebbe essere la migliore. Riprendendo quindi lo schema in figura 8.16, abbiamo:

- Waiter = main Isolate;
- Filosofi = Isolates generati dal main Isolate.

Gli Isolates generati non hanno bisogno di conoscersi l'uno con l'altro, devono semplicemente interagire con il main Isolate. Seguendo lo schema

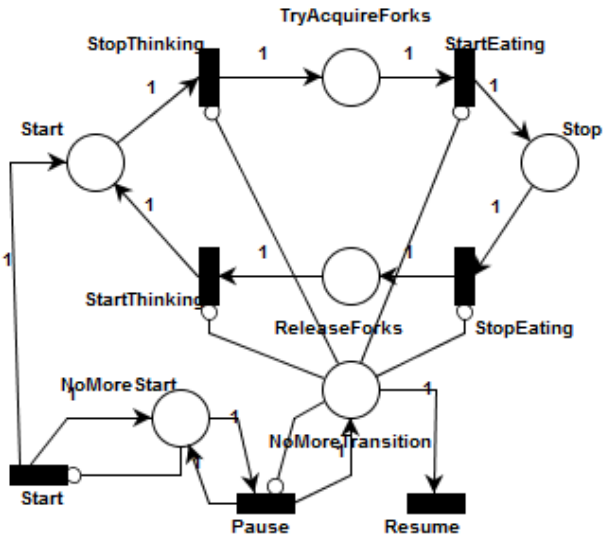

Figura 8.20: Rappresentazione dello stato dei filosofi con una Rete di Petri, dove si mostrano come le operazioni di *start*, *pause* e *resume* influenzino il flusso della simulazione. Per semplicità, si è deciso di omettere lo *stop* che dovrà semplicemente ripristinare lo stato iniziale.

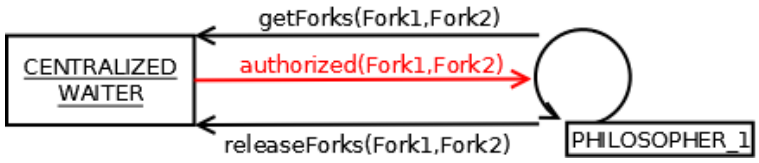

Figura 8.21: Scenario in cui si ha un Waiter centralizzato che comunica con i Filosofi mediante scambio di messaggi

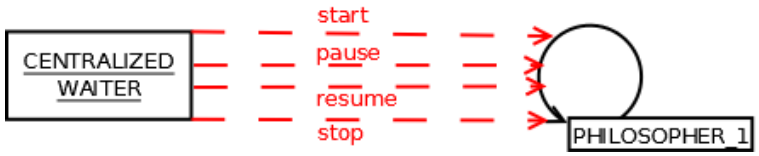

Figura 8.22: Scenario in cui si ha un Waiter centralizzato deve segnalare ai Filosofi l'arrivo dei comandi di Pause o di Stop

in figura 8.19, un Isolate-filosofo cercherà di comunicare con il Waiter negli stati *TryAcquireForks*, quando vuole acquisire le forchette, e *ReleaseForks*, quando ha finito di mangiare e vuole rilasciare le forchette. Nella fase in cui il filosofo cerca di prendere le forchette, dovrà prima aspettare che il Waiter gli invii un messaggio di risposta che lo autorizza ad utilizzarle. Al contrario, quando si tratta di rilasciarle, l'Isolate-filosofo si può limitare semplicemente ad informare che non le sta più usando, senza bisogno di aspettare una risposta dal main Isolate.

Abbiamo quindi tre tipi di messaggi che fondamentalmente vengono scambiati, due a partire dall'Isolate-filosofo, *getForks* e *releaseForks*, ad uno che parte dal Waiter, *authorized* (figura 8.21).

Vi sono anche i quattro comandi disponibili all'utente (figura 8.15) per comandare l'esecuzione della simulazione, *start*, *stop*, *pause* e *resume*, a cui corrisponderanno altrettanti comandi di input, possibilmente dei bottoni, che permettono rispettivamente di avviare, stoppare, mettere in pausa e riprendere la simulazione.

Gli eventi del DOM vengono percepiti solamente dal main Isolate che quindi si dovrà occupare di avvisare gli Isolate-filosofi quando si devono stoppare o riprendere il loro ciclo (figura 8.22).

All'avvio dell'applicazione:

```
// Waiter.
main() {
  ...

  // Enable startPhilo button and disable stopPhilo
     button.
  start.disabled = false;
  stop.disabled = true;
  pause.disabled = true;
  resume.disabled = true;

  start.onClick.listen(startPhiloIsolates);
  stop.onClick.listen(stopPhiloIsolates);
  start.onClick.listen(pausePhiloIsolates);
  stop.onClick.listen(resumePhiloIsolates);
}
```

start, *stop*, *pause* e *resume* sono i quattro comandi di input. Fra questi, solo il primo all'inizio viene lasciato abilitato; in questo modo si mantiene una coerenza nell'esecuzione dell'applicazione (affinché si possa stoppare, mettere in pausa o riprendere una simulazione, deve essere prima mandata in esecuzione).

Alla pressione del bottone *start* viene eseguita la funzione *startPhiloIsolates*:

```
startPhiloIsolates(Event e) {
  // Disable start button and enable stop and pause
     button.
  start.disabled = true;
  stop.disabled = false;
  pause.disabled = true;
  resume.disabled = false;

  // Opens a port for receiving messages.
  receivePort = new ReceivePort();
  msgSubscription = receivePort.listen(handleMessage);
```

```
  for (int i = 0; i < NPhilo; i++) {
    //Init state for philosophers
    philosophers[i].changeState("Init");
    /*
     * Generate NPhilo isolate with args[0] = philoID
        and args[1] = total number of philo
     * with a msg which is the sendPort for the Waiter
        .
     */
    Isolate.spawnUri(uri_isolate, ['$i', '$NPhilo'],
      receivePort.sendPort);
  }
  scheduleMicrotask(redraw);
}
```

All'avvio della simulazione, si disabilita *start* e si abilitano i bottoni *stop* e *pause*. Successivamente si crea una ReceivePort per il main Isolate, la cui SendPort verrà passata come primo messaggio agli Isolate-filosofi, nella funzione *Isolate.spawnUri*. Si noti che la variabile *msgSubscription* è di tipo StreamSubscription; ogni messaggio ricevuto dalla ReceivePort genera un evento che viene gestito con il metodo *handleMessage*.

I messaggi scambiati possono essere qualsiasi tipo di variabile. In questo caso viene comodo organizzarli come array di valori (per lo più **interi** o **stringhe**). Ad esempio, quando un Isolate-filosofo vuole richiedere l'uso delle forchette, richiamerà il seguente metodo:

```
wantsToEat() {
  // The Philosopher wants to Eat.
  state = "WantsToEat";

  waiterSendPort.send([state, '$MyID', fork1, fork2,
     myReceivePort.sendPort]);
}
```

dove tramite la *waiterSendPort.send*, invia un messaggio al Waiter con le seguenti informazioni:

1. *State*: Il primo argomento rappresenta lo stato attuale del filosofo. Il motivo per cui viene inviato è che fornisce sia informazioni su come

rappresentarlo nella GUI, sia identifica il tipo di messaggio; infatti questo in particolare è l'equivalente del *getForks* in figura 8.21.

2. *MyID*: Il secondo argomento si tratta dell'Id del filosofo. Questa informazione è necessaria per distinguere i vari Isolate-filosofi, essendo che il Waiter utilizza una sola ReceivePort per tutti gli Isolates.

3. *Fork1* e *Fork2*: Il terzo e quarto sono gli indici delle forchette che il filosofo vorrebbe acquisire.

4. *myReceivePort.sendPort*: L'ultimo argomento è la SendPort che il Waiter dovrà utilizzare per rispondere a quel filosofo.

La gestione dei messaggi da parte del Waiter avverrà nel seguente modo:

```
handleMessage(var msg) {
 if (msg[0] == "WantsToEat") {
    var id = msg[1],
        indexFirstFork = msg[2],
        indexSecondFork = msg[3];
    SendPort reply = msg[4];

    if (availForks[indexFirstFork] == true &&
       availForks[indexSecondFork] == true) {
      availForks[indexFirstFork] = false;
      availForks[indexSecondFork] = false;

      //Reply forks are available
      reply.send("Authorized");
    } else {
      //Add the request in the queue
      pendingRequest.add(new Request(reply, id,
         indexFirstFork, indexSecondFork));
    }
  } else if (msg[0] == "StopEating") {
    availForks[indexFirstFork] = true;
    availForks[indexSecondFork] = true;

    pendingRequest.forEach((Request request) {
```

```
        if (availForks[request._firstFork] && availForks
            [request._secondFork]) {
            availForks[request._firstFork] = false;
            availForks[request._secondFork] = false;
            var id = request._id;
            SendPort reply = request._sendPort;

            // Remove the current request and then reply
                forks are available.
            new Future(() => pendingRequest.remove(
                request))
            ..then((_) => reply.send("Authorized"));
          }
        });
    });
  }
}
```

Fondamentalmente vi sono due tipi di messaggi che il Waiter può ricevere:

1. Messaggio di Tipo *WantsToEat*: verifica la disponibilità delle forchette richieste. In caso positivo, risponde all'Isolate-filosofo con un messaggio *authorized*. In caso negativo, aggiungo ad una lista di richieste, un nuovo oggetto Request (figura 8.23).

2. Messaggio di Tipo *StopEating*: indica che un filosofo ha terminato di utilizzare le forchette. Questo significa che una Request fatta da un altro filosofo potrebbe essere soddisfatta. Con il ciclo *pendingRequest.forEach((Request request) ...);* si analizzano le richieste in sospeso e se ne si trova una che può essere soddisfatta, si cambia lo stato delle forchette e si invia un messaggio *authorized*.

L'operazione di rimozione della richiesta e di invio del messaggio *authorized* viene effettuata, in maniera molto elegante, sfruttando le Future; la *remove(request)* non viene effettuata durante il ciclo (rischierebbe di causare problemi di lettura) ma "in futuro". Una volta rimossa la richiesta dalla lista, con il metodo *then* si invia subito dopo il messaggio al filosofo.

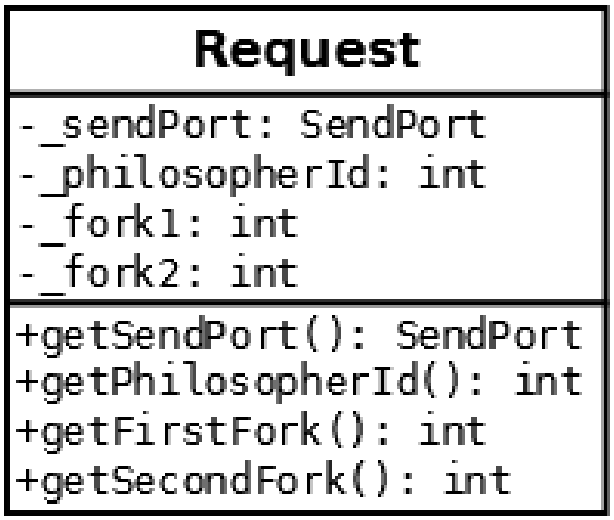

Figura 8.23: Rappresentazione in UML della classe Request

Illustriamo quindi le rimanenti parti importanti che riguardano un Isolate-filosofo:

```
main(List<String> args, SendPort sendPort) {

  // Get the args.
  MyID = args[0];
  NPhilo = args[1];

  // Ordering the forks.
  firstFork = int.parse(MyID);
  secondFork = ((firstFork + 1) >= int.parse(NPhilo))
     ? 0 : firstFork + 1;

  // Set the SendPort and the ReceivePort.
  waiterSendPort = sendPort;
  myReceivePort = new ReceivePort();

  myReceivePort.listen(handleMessage);
}
```

```
handleMessage(var msg) {
  // Select the right msg handler.
  if (msg == "Authorized") {
    eating();
  }
}

thinking() {
  state = "Thinking";
  think().then((_) => wantsToEat());
}

eating() {
  state = "Eating";
  eat().then((_) {
     stopEating();
     thinking();
  });
}

stopEating() {
  // The Philosopher stop eating and release the forks
     .
  state = "StopEating";

  waiterSendPort.send([state, '$MyID', fork1, fork2,
     myReceivePort.sendPort]);
}
```

Fin qui, non sembrerebbero esservi particolari problemi nell'implementazione in Dart. Cosa succede però quando negli Isolate-filosofi si vuole aggiungere la proprietà di poter reagire ai messaggi di *pause*, *resume* e *stop*?

Sicuramente la *handleMessage* dev'essere modificata per poter gestire quel tipo di messaggi:

```
handleMessage(var msg) {
```

```
  // Select the right msg handler.
  if (msg == "Authorized") {
    eating();
  }
  else if (msg == "Pause") {
        handlePause();
  }
  else if (msg == "Resume") {
        handleResume();
  }
  else if (msg == "Stop") {
        handleStop();
  }
}
```

Questo però non basta, una volta che il filosofo si trova impegnato rispettivamente nelle funzioni di *think* e di *eat* diventa "insensibile"ai messaggi che riceve, per via della semantica macro-step nell'esecuzione delle operazioni (esecuzione atomica). In questo caso, gli Isolate-filosofi non riescono ad essere sufficientemente reattivi e solo in determinati punti della loro computazione/ciclo di vita possono gestire la ricezione dei messaggi. Essendo questa una simulazione, nella pratica gli Isolate-filosofi non fanno nulla durante le fasi di *think* e di *eat*. In un contesto reale, dove è possibile avere degli Isolate che devono effettuare delle computazioni, accedere a delle risorse condivise ed essere reattivi a dei comandi esterni, per ottenere un comportamento di questo tipo si dovrebbe frammentare la computazione, concatenandola con una serie di Futures:

```
Future longAsyncComputation() {
  return littleComputationA()
    .then((resultOfA) => littleComputationB(resultOfA)
      )
    .then((resultOfB) => littleComputationC(resultOfB)
      )
    .then((resultOfC) => littleComputationD(resultOfC)
      )
    .then((resultOfD) => littleComputationE(resultOfD)
      );
```

```
    // And so on..
}
```

NOTA: come già accennato nel capitolo 4.2.5, gli Isolates hanno ancora diversi problemi di funzionamento. Oltre che è difficile farvi debug e dei test, l'esecuzione delle Futures non funziona all'interno di un Isolate.

Ad esempio l'esecuzione del seguente frammento di codice, all'interno di Isolate, lo blocca irreversibilmente:

```
new Future.delayed(new Duration(seconds:5),() =>
    waiterSendPort.send(["Thinking", '$MyID']));
```

Il codice mostrato è dunque un approccio non ancora operativo, nel senso che attualmente bloccherebbe il funzionamento dell'applicazione. Qualora questi problemi venissero risolti, sarebbe tuttavia possibile adottare l'approccio sopra mostrato poiché concettualmente corretto.

8.3.4 Approccio con JaCaMo

In JaCaMo abbiamo tutte le astrazioni necessarie per poter modellare sia i filosofi, come agenti, sia le forchette, come Artefatto.

L'unica aggiunta che si dovrà fare sarà riguardante un ulteriore artefatto, che lo possiamo chiamare *Status*, che dirigerà il comportamento ad un metalivello, quello dove l'utente è in grado di comandare la simulazione, dei filosofi.

È possibile quindi riprendere la figura 8.17 ed impostare l'interazione artefatti-agenti come segue (per semplicità viene mostrato il caso di un filosofo, gli altri sono analoghi):

- **Artefatto Status**: Unico artefatto centrale che governa l'andamento della simulazione. In figura 8.17 viene mostrata la parte di interazione con l'agente, si dovranno impostare dei metodi richiamabili dall'utente per poterne modificare lo stato. Gli eventi di *Start* e *Stop* sono modellati come Segnali (indicano quando gli agenti devono cominciare e quando devono terminare), mentre una Proprietà Osservabile booleana *isPause* indica se il filosofo deve interrompersi momentaneamente nell'esecuzione o meno. Si precisa che la suddetta configurazione è

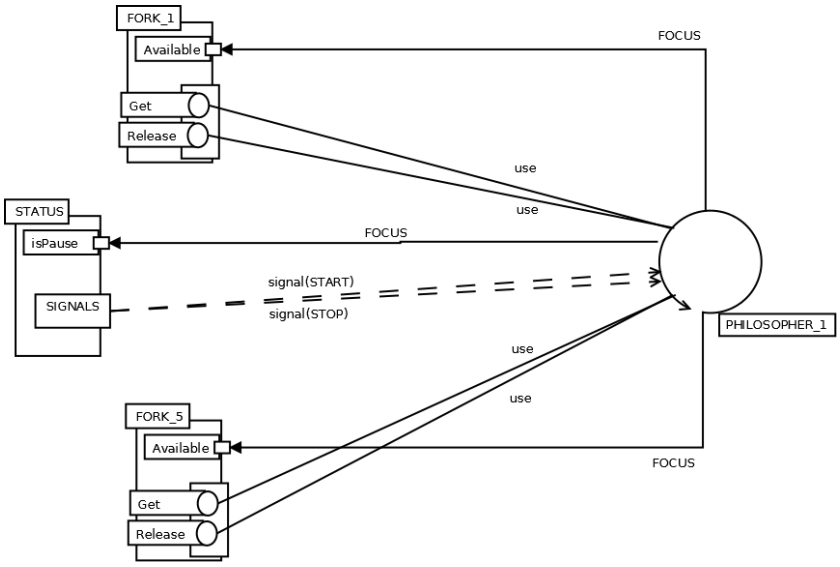

Figura 8.24: Uno schema di interazione fra artefatti *Status* e *Fork*, e l'agente *Philosopher*

solo uno dei possibili modi, l'artefatto poteva benissimo essere modellato con un unica Proprietà Osservabile chiamata *state* di tipo stringa che, a seconda del comando dell'utente, cambia in accordo e gli agenti reagiscono di conseguenza.

- **Artefatto Fork**: Artefatto rappresentante le risorse condivise dagli agenti. I metodi *get* e *release* permettono ai filosofi di acquisirne il possesso, nel primo caso, o di liberarla, nel secondo caso. Una Proprietà Osservabile *available* di tipo booleano viene modificata in accordo allo stato della forchetta.

- **Agente Philosopher**: Ogni agente-filosofo seguirà la successione di stati mostrata in 8.19 ciclicamente. Esso inizierà appena percepisce il *segnale Start*, continuerà ad eseguire le transizioni fintanto la *proprietà osservabile isPause* è falsa e terminerà, preoccupandosi di rilasciare le risorse *Fork*, appena non gli arriverà il *segnale Start*.

Per quanto riguarda l'artefatto *Status*, possiamo quindi definirlo come segue:

```
public class Status extends Artifact {

        void init() {
                defineObsProperty("isPause", false);
        }

        @OPERATION
        void start() {
                signal("start");
        }

        @OPERATION
        void stop() {
                signal("stop");
                getObsProperty("isPause").updateValue(
                    false);
        }

        @OPERATION
        void pause() {
                ObsProperty prop = getObsProperty("
                    isPause");
                if(prop.booleanValue() == false)
                        prop.updateValue(true);
                else
                        prop.updateValue(false);
        }
}
```

I metodi (che possono essere definiti come @OPERATION o @LINK) saranno utilizzati da un altro agente che si preoccuperà di gestire l'interazione con la GUI. L'operazione *stop* resetta anche lo stato della *proprietà osservabile*.

Per quanto riguarda invece la *Fork*, si è deciso di non utilizzare un una *proprietà osservabile*, ma di sfruttare il tag @GUARD per l'esecuzione dell'operazione *get*; nei metodi annotati in questo modo, affinché si possa ese-

guire il metodo, bisogna che la condizione specificata sia verificata altrimenti l'esecuzione viene sospesa.

```
public class Fork extends Artifact {
        int forkId;

        int philoId;
        boolean available;

        void init(int id) {
                forkId = id;
                //-1 means no philosopher has the fork
                philoId = -1;
                available = true;;
        }

        @OPERATION (guard="isAvailable")
        void get(int philoId) {
                available = false;

                this.philoId = philoId;
        }

        @OPERATION
        void release(int philoId) {
                if(this.philoId == philoId &&
                   available == false) {
                        available = true;

                        System.out.println("Fork("+
                           forkId +")->ReleasedBy: " +
                            philoId);
                        this.philoId = -1;
                }
                else {
                        failed("failed_to_release", "
                           release_failed", "
                           YouCannotReleaseThisFork");
```

```
                }
        }

        @GUARD
        boolean isAvailable(int philoId) {
                return available;
        }
}
```

Infine mostriamo come, nell'agente filosofo, la parte dei piani rilevanti:

```
+start : state(State) & State == "WaitingStart"
<-      +state("Running");
        !living.

+isPause(PauseState) : PauseState == false & state(
   State) & State == "Running"
<-      .resume(living).

+isPause(PauseState) : PauseState == true & state(
   State) & State == "Running"
<-      .suspend(living).

+stop : state(State) & State == "Running"
<-      .drop_all_desires;
        .print("ReceivedStop..");
        +state("WaitingStart");
        !releasePlan.

+!living : true
<-      !thinking;

        !acquireFork1;
        !acquireFork2;

        !eat;

        !releaseFork1;
```

```
        !releaseFork2;

        !living.
```

Le funzioni *.suspend* e *.resume* sono delle azioni di default eseguibili dall'agente, che permettono rispettivamente di sospendersi e di riprendere il piano principale, *!living*. Inoltre l'agente filosofo tiene al suo interno un *Belief* di nome *state*, utilizzato per le condizioni degli eventi riguardanti i comandi *start, stop* e *pause*, dall'esterno.

8.3.5 Confronto

Non sorprende che, in questo particolare caso di studio, l'approccio con JaCaMo risulti decisamente più vantaggioso. Entità quali gli agenti si adattano perfettamente a task di questo tipo, dov'è necessario che il programma sia proattivo e reattivo allo stesso tempo. I filosofi devono continuamente eseguire i loro compiti, pensare e mangiare, rimanendo comunque all'erta in caso di segnali esterni.

In Dart, oltre ai problemi di specifiche riguardo gli Isolates e i Workers, riuscire ad implementare un comportamento di questo tipo è piuttosto difficile per via della semantica macro-step del modello ad eventi. Se un Isolate si impegna nell'esecuzione di una funzione, questa viene eseguita fino al suo termine e gli eventi nella coda vengono temporaneamente ignorati. Una soluzione è quella di frammentare il codice, concatenandolo con una serie di Futures. Questo approccio è tuttavia un escamotage per mantenere l'Event Loop reattivo continuando ad eseguire i propri compiti. Una soluzione di questo tipo **riduce il livello di astrazione usato per descrivere la strategia identificata in fase di progettazione**.

In conclusione, la soluzione ad agenti è risultata senza ombra di dubbio molto più elegante e naturale, con un passaggio dal modello derivante dall'analisi all'implementazione immediato e con un codice risultante sicuramente più leggibile e comprensibile rispetto alla controparte ad eventi.

Capitolo 9

Conclusioni e Lavori Futuri

Lo scopo di questo lavoro è stato quello di mettere in luce cosa vuol dire affrontare problematiche reali in ambito Web, sia con gli approcci considerati standard, sia sfruttando modelli alternativi ma non per questo meno espressivi. Al contrario, si è visto come il Control Loop di Jason è nei fatti una versione estesa dell'Event Loop dei tradizionali linguaggi ad eventi, con alcune differenza nel ciclo di controllo atte a supportare comportamenti proattivi e reattivi al tempo stesso.

Queste diversità si sono rilevate particolarmente adatte quando si è trattato di affrontare il problema dei *Filosofi a Cena* e le proprietà intrinseche negli agenti hanno ridotto notevolmente l'abstraction gap dai modelli definiti in fase d'analisi all'implementazione vera e propria.

Anche nel caso della *Lavagna Distribuita*, il framework JaCaMo ha fornito astrazioni atte a gestire facilmente il problema nel distribuito come se fosse in locale, grazie all'utilizzo di nomi logici location-transparent. Lo scambio di informazioni tra agenti ed artefatti avviene in modo automatico, registrando semplicemente un interesse per certe informazioni.

Anche per quanto riguarda applicazioni prettamente reattive, come nel caso di studio *Indovina il Numero*, un approccio ad agent-oriented permette una naturale applicazione del pattern Model-View-Controller, sfruttando gli agenti per definire la business logic e mantenendo separate tutte quelle componenti passive come la presentation e data logic, modellandole con artefatti. In Dart, nonostante vengano forniti numerosi costrutti, vi è sempre il rischio di mescolare contesti che andrebbero mantenuti separati. A riprova di ciò, i numerosi framework sviluppati per JavScript e Dart che

implementano il suddetto pattern Model-View-Controller, su tutti Angular.

Parlando di studi futuri, sarebbe interessante affrontare uno studio di un'applicazione in ambito di *Web Semantico* con una tecnologia ad agenti.

In breve, ciò a cui si sta assistendo negli ultimissimi anni è il passaggio dal cosiddetto *Web 2.0* al *Web 3.0*, la nuova versione del Web. Tra le features che spiccano in quello che sarà la prossima grande rivoluzione, sicuramente una delle più rilevanti è il *Web Semantico*. Citando quel che viene detto a riguardo da Tim Berners-Lee [33], il Web è un enorme database, contenente una miriade di informazioni, che tuttavia sono pensate e strutturate per essere usufruite dagli umani e non da una macchina che naviga sul Web. Le informazioni che leggiamo sono puro testo scritto che noi riusciamo ad interpretare attribuendogli una semantica, per un robot invece è molto più difficile e problematico riuscire ad interpretare in maniera corretta una sequenza di parole. Per questo si stanno studiando tutta una serie di tecnologie, come *RDF* (Resource Description Framework) o *OWL* (Ontology Web Language), atte ad esprimere le informazioni in modo tale che siano processabili da una macchina. Vi sono ancora delle sfide da affrontare in ambito Web Semantico, tuttavia le possibili applicazioni potrebbero aprire a nuovi scenari per le Web Applications dove lo scambio, l'ottenimento ed il processamento di informazioni avviene in base al loro significato semantico. In un contesto di questo tipo, entità come gli agenti troverebbero sicuramente ampio spazio.

Per quanto riguarda i due cicli di controllo, indubbiamente il Control Loop è più completo e permette di svolgere molte più funzioni in maniera più ordinata. D'altro canto, in certi passaggi (com'è stato evidenziato anche nella parte 7), il ciclo di controllo degli agenti risulta essere troppo complicato e troppo poco ottimizzato, specialmente per certe tipologie di applicazioni Web, tipicamente quelle più semplici. A tal proposito, vi è già un lavoro in atto che riguarda un nuovo linguaggio prototipale, chiamato ALOO, dove vi sono agenti ed oggetti come entità di prima classe, con i primi che presentano un Control Loop simile a quello degli agenti in Jason ma molto più semplificato [34].

Per quanto riguarda l'implementazione in JaCaMo, le difficoltà principali sono dovute al fatto che la piattaforma è sì funzionante, ma non dispone di

tutta quella serie di costrutti sintattici che renderebbero molto più facile la programmazione (*Zucchero Sintattico*). Questi costrutti sono superflui dal punto dell'espressività del linguaggio, ma velocizzano lo sviluppo di software e facilitano la leggibilità e mantenibilità del codice. Spesso, anche per fare una cosa molto semplice, si è dovuto scrivere molte più righe del necessario.

Il debugging di applicazioni JaCaMo inoltre presenta molte lacune. Gli strumenti non forniscono sufficienti dettagli per capire bene dove si trova il problema, anche se si tratta di un banale errore di scrittura.

Anche il refactoring di una proprietà osservabile richiede molto più tempo del dovuto, per il semplice fatto che non si ha un vero e proprio oggetto definito nel programma; in un Artefatto, per ricavarsi una proprietà osservabile, bisogna ogni volta fare pattern matching con stringhe e valori (ad esempio, *ObsProperty prop = getObsPropertyByTemplate(user, userName, null, null, null, null)*). Per rinominare una proprietà, si è costretti ad andare a mano nel codice a ricercare ogni volta che viene richiamata, ricordandosi di aggiornare non solo lato Artefatto ma anche lato Agente. Situazione analoga per i Beliefs, dove oltretutto non vi è una netta distinzione fra Belief interni ed esterni.

Sicuramente il modello ad agenti, applicato in contesti general-purpose, può portare dei benefici in termini di progettazione. Si vede però la necessità di lavorare sui tools e le facilities che aiutino il programmatore nel suo compito.

Bibliografia

[1] Jhon L. Manferdelli, Naga K. Govindaraju, Chris Crall, *Challenges and Opportunities in Many-Core Computing*.

[2] Sutter's Mill. Herb Sutter on software, hardware, and concurrency. *"Welcome to the Jungle"*. `http://herbsutter.com/welcome-to-the-jungle/`.

[3] Herb Sutter, *"The free lunch is over"*. `http://www.gotw.ca/publications/concurrency-ddj.htm`.

[4] John Ousterhout, *Why Threads Are Bad Idea (for most purposes)*. Sun Microsystems Laboratories.

[5] Gregor Hohpe, *Programming Without a Call Stack – Event-driven Architectures*. `www.eaipatterns.com`.

[6] Edward A. Lee, *The Problem with Threads*. University of California at Berkeley, `http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.html`.

[7] Java Tutorial, *The Event Dispatch Thread*. `https://docs.oracle.com/javase/tutorial/uiswing/concurrency/dispatch.html`.

[8] Graham Hamilton, *Multithreaded toolkits: A failed dream?*. `https://weblogs.java.net/blog/kgh/archive/2004/10/multithreaded_t.html`.

[9] Flapjax official page. `http://www.flapjax-lang.org/`.

[10] w3schools, *Web Worker Definition*. `http://www.w3schools.com/html/html5_webworkers.asp`.

[11] *Web Worker*. https://html.spec.whatwg.org/multipage/workers.html#introduction-15.

[12] Rajesh K. Karmani, Gul Agha, *Actors*. Department of Computer Science, University of Illinois at Urbana-Champaign.

[13] Internal Google Mail. https://gist.github.com/paulmillr/1208618.

[14] *Dart Event Loop*, https://www.dartlang.org/articles/event-loop.

[15] *Dart Futures*. https://www.dartlang.org/articles/futures-and-error-handling.

[16] *Dart Streams*. https://www.dartlang.org/docs/tutorials/streams.

[17] *Dart Single-Subscription vs. Broadcast*. https://www.dartlang.org/articles/broadcast-streams.

[18] *Dart: Streams are the future*. https://www.dartlang.org/slides/2013/06/dart-streams-are-the-future.pdf.

[19] *Dart: Html of the Future*. http://storage.googleapis.com/io-2013/presentations/219_dart_html_of_the_future_today.pdf.

[20] *Dart: A Modern Web Language*. https://www.dartlang.org/slides/2012/06/io12/Dart-A-Modern-Web-Language.pdf.

[21] *Dart Zones*. https://zones-dot-dart-lang.appspot.com/articles/zones.

[22] *Dart Language Asynchrony Support: Phase 1*. https://www.dartlang.org/articles/await-async/.

[23] Erik Meijer, *Your Mouse is a Database*, http://queue.acm.org/detail.cfm?id=2169076.

[24] Brian Ford, Angular Team, *Zone.js from Angular Team*. http://thechangelog.com/zone-js.

[25] Issue 17667, *Zone, isolates and Worker are poorly documented*. `https://code.google.com/p/dart/issues/detail?id=17667`.

[26] Rafael H. Bordini, Jomi Fred Huber and Michael Wooldridge, *Programming multi-agent systems in AgentSpeak using Jason*, ed. Wiley.

[27] Wooldridge M and Jennings NR, Intelligent agnts: theory and practice *The Knowledge Engineering Review*. 115 - 152, 1995.

[28] Bratman ME, *Intention, Plans, and Practical Reason*. Harvard University Press, Cambridge, Ma, 1987.

[29] Bratman ME, What is intention? *Intentions in Communication*. The MIT Press, Cambridge, Ma, 1987.

[30] H. Lieberman, *The continuing quest for abstraction*. In D.Thomas editor, ECOOP, volume 4067 of Lecture Notes in Computer Science, pages 192-197. Springer, 2006.

[31] *Web 3.0*. `http://it.wikipedia.org/wiki/Web_3.0`.

[32] *Semantic Web*. `http://en.wikipedia.org/wiki/Semantic_Web`.

[33] Tim Berners-Lee, *Semantic Web Road map*. `http://www.w3.org/DesignIssues/Semantic.html`.

[34] Alessandro Ricci, Andrea Santi, *Concurrent Object-Oriented Programming with Agent-Oriented Abstractions – The ALOO Approach*. University of Bologna.

[35] Alessandro Ricci, Andrea Santi, *Beyond the Reactivity Principle*, Marrying Objects and Agents in ALOO. University of Bologna.

[36] Alessandro Ricci, *From Actor Event-Loop to Agent Control-Loop – Impact on Programming*. University of Bologna.