Consiglio Nazionale delle Ricerche
Istituto di Calcolo e Reti ad Alte Prestazioni

METODOLOGIE DI PROGETTAZIONE E AMBIENTI PER LO SVILUPPO DI APPLICAZIONI AD AGENTI

Massimo Cossentino, Luca Sabatucci, Valeria Seidita, Antonio Chella

RT-ICAR-PA-04-10 **Agosto 2004**

Consiglio Nazionale delle Ricerche, Istituto di Calcolo e Reti ad Alte Prestazioni (ICAR)
– Sede di Cosenza, Via P. Bucci 41C, 87036 Rende, Italy, URL: *www.icar.cnr.it*
– Sezione di Napoli, Via P. Castellino 111, 80131 Napoli, URL: *www.na.icar.cnr.it*
– Sezione di Palermo, Viale delle Scienze, 90128 Palermo, URL: *www.pa.icar.cnr.it*

Consiglio Nazionale delle Ricerche
Istituto di Calcolo e Reti ad Alte Prestazioni

METODOLOGIE DI PROGETTAZIONE E AMBIENTI PER LO SVILUPPO DI APPLICAZIONI AD AGENTI

Massimo Cossentino[1], Luca Sabatucci[2], Valeria Seidita[2], Antonio Chella[1,2]

Rapporto Tecnico N.10: ***Data:***
RT-ICAR-PA-04-10 **Agosto 2004**

[1] Istituto di Calcolo e Reti ad Alte Prestazioni, ICAR-CNR, Sezione di Palermo Viale delle Scienze edificio 11 90128 Palermo

[2] Università degli Studi di Palermo Dipartimento di Ingegneria Informatica Viale delle Scienze 90128 Palermo

Indice

1. Introduzione

Il presente documento illustra gli studi fatti presso l'ICAR-CNR sulle metodologie di progettazione dei sistemi multi-agenti e lo sviluppo di una nuova metodologia (chiamata PASSI Agile) che partendo dalla esperienza compiuta negli anni scorsi su PASSI, raccoglie gli stimoli derivanti dal movimento dei processi agili e dà luogo ad un nuovo approccio specificatamente concepito per la rapida implementazione di sistemi robotici.

Il documento si articola in 6 capitoli (a parte la presente introduzione, capitolo 1): nel capitolo 2 verranno presentati i sistemi multi-agente con una specifica attenzione per la piattaforma FIPA; nel capitolo 3 verranno introdotte le problematiche fondamentali della agent-oriented software engineering (AOSE) che è la disciplina di riferimento per il lavoro che si è svolto; nel capitolo 4 si introduce il concetto di pattern di agente a partire da quello di design pattern (ad oggetti) e sulla base di questa definizione si propone una implementazione di un meta-linguaggio per la specifica dei pattern ad agenti che verrà poi adoperato per l'introduzione del pattern-reuse in PASSI Agile. Il capitolo 5 illustra la cosiddetta method engineering che nell'ambito della AOSE viene adoperata per assemblare una nuova metodologia riusando parti di metodologie esistenti (opportunamente modificate se necessario). Il capitolo 6 illustra lo sviluppo di PASSI Agile e descrive i dettagli della metodologia. Infine nel capitolo 7, si è effettua una valutazione dei risultati ottenuti e si tracciano le conclusioni sul lavoro svolto.

2. Sistemi ad Agenti

2.1 Introduzione agli Agenti.

Il concetto di "agente" rappresenta l'ultima frontiera raggiunta dalla moderna ingegneria del software, che basa le proprie fondamenta sul tentativo di adottare il metodo scientifico nell'implementazione di soluzioni a problemi sempre più complessi attraverso lo strumento informatico.

In principio i primi tentativi di programmazione erano lasciati ad esperti che artigianalmente sviluppavano in assembler programmi dedicati alla risoluzione di problemi specifici, pertanto spesso fini a se stessi.

L'evoluzione dell'informatica ha visto la nascita di un numero elevato di linguaggi e tecniche di programmazione che dall'assembler ci hanno portato alla programmazione strutturata prima, a quella orientata agli oggetti poi e infine alla programmazione orientata agli agenti.

Così come la programmazione strutturata rappresentava il passaggio da un tipo di approccio diretto alla risoluzione di un problema singolo, a quello mirato a riusare il codice scritto per non dovere reinventare la ruota ogni volta; la programmazione ad oggetti ha rappresentato la massima evoluzione del riutilizzo del software, della stabilità, della robustezza, concetti che hanno dato ordine alla programmazione strutturata, facilitando il lavoro del progettista attraverso strumenti che hanno accelerato i tempi per l'analisi, la progettazione e lo sviluppo di sistemi complessi.

Oggi la programmazione orientata agli agenti è un ulteriore passo avanti verso la risoluzione di problemi reali complessi, per loro natura distribuiti e con controllo decentralizzato, categoria di problemi consigliata dalla nascita e dallo sviluppo di una nuova infrastruttura aperta e dinamica quale quella di internet.

Problematiche in continua evoluzione e caratterizzate dalla proprietà di essere spesso imprevedibili in fase di design, hanno fatto sì che un approccio di tipo object oriented spesso potesse non bastare.

L'oggetto pertanto è costretto dalle circostanze a diventare agente, ossia ad espandersi nella struttura, nelle proprie responsabilità, nei propri diritti, nelle azioni che è capace di compiere e nell'individuare obiettivi da raggiungere.

Nasce così il concetto di agente, ossia di un'entità forte dotata di autonomia, di reattività, di proattività, concetti inevitabilmente imposti dal dovere affrontare problematiche potenzialmente mutevoli nel tempo.

L'esempio più semplice che può far capire la differenza fra oggetto e agente è quello secondo cui un oggetto riceve un input, che lo attiva, e risponde sempre con lo stesso output, indipendentemente dal contesto e dal tempo; un agente invece è sempre attivo, anzi proattivo (prende l'iniziativa), riceve un input e può decidere se dare o no l'output al richiedente e se lo restituisce il valore dipende dal contesto in cui è richiesto e dal tempo.

Un oggetto pertanto è un'entità passiva, comunica con messaggi statici, inoltre le relazioni fra gli oggetti sono sempre fisse e note a priori.

Un agente invece è un'entità attiva, ha una base di conoscenza in continua evoluzione, è percettivo nei confronti dell'ambiente in cui vive, è autonomo (decide se operare e quando), interagisce con altri agenti facendo parte di una struttura sociale dinamica, ha degli obiettivi ed è capace di fare pianificazioni per raggiungerli, creando alla bisogna strutture sociali necessarie al raggiungimento del singolo obiettivo per poi slegarsi e costituirne di nuove, riesce anche a mutare con le proprie azioni l'ambiente in cui vive.

Un esempio di agente può essere quello implementato dalla Microsoft per la clip animata del pacchetto Office, oppure basti pensare ai virus polimorfici, o agli avversari software dei videogame 3D come in Quake, o ancora ai web spiders che collezionano dati da ricercare in internet e poi li restituiscono a chi li ha richiesti.

Un agente per vivere ha bisogno di un'infrastruttura, di un sistema multiagente, creando comunità rette da regole sociali, da protocolli da rispettare, comunicando mediante messaggi di tipo ACL (agent communication language), ossia usando opportuni linguaggi, assumendo uno o più ruoli e, in funzione di questi, sfrutta concetti di coordinazione, cooperazione e intelligenza artificiale per il raggiungimento di obiettivi.

Uno dei problemi che bisogna inevitabilmente affrontare nella programmazione ad agenti, così come in quella ad oggetti, è la rappresentazione di questi in fase di design. Ci viene in aiuto l'estensione dell'UML alla Agent Oriented Software Engineering (AOSE).

L'AUML consente, infatti, di creare dei modelli grafici con i quali schematizzare la struttura sociale in cui vivranno i nostri agenti, consente di rappresentare sistemi o singoli agenti e a partire da essi ci dà la possibilità di ottenere la scrittura automatica del codice che implementerà la soluzione al problema.

Tutto ciò ha determinato sempre più l'accorciamento delle distanze fra analista e sviluppatore, fondendo le due attività e facendo in modo che un solo strumento potesse bastare a tutto il ciclo di vita del software.

Esiste al momento un buon numero di architetture per sistemi multiagente, che hanno lo scopo di dare un'interpretazione specifica al concetto di agente e a tutto ciò che lo riguarda.

L'elaborato cercherà di analizzare alcune fra queste architetture, fra cui FIPA, la architettura BDI, la Aglet e infine l'architettura MASIF, evidenziando anche alcune loro implementazioni esistenti, cercando altresì di mostrare come le singole architetture vedono il modello e come da questo siano condizionate le relative implementazioni.

Dall'analisi delle architetture esistenti si evince che i punti chiave siano comuni a tutte ed esattamente, al fine di dare un modello base di sistema multiagente, ci viene in aiuto l'idea di un gruppo di lavoro coordinato che deve raggiungere un obiettivo prefissato.

In tale gruppo c'e' sempre chi ha più esperienza degli altri e pertanto gestisce i colleghi per il raggiungimento di un risultato globale di più alto livello, facendo da coordinatore di risorse; fra gli altri rimasti ci saranno i responsabili, che rappresenteranno i livelli intermedi di management, e infine le risorse, che svolgeranno attività di più basso livello, ma necessarie al contesto globale.

Secondo uno schema piramidale, ognuno sarà dotato di responsabilità, diritti e doveri, azioni possibili e protocolli da rispettare.

Da questo modello si evidenziano i seguenti punti:

- Responsabilità di un agente.

  La responsabilità determina le funzionalità di ogni agente in base al proprio ruolo, ossia ogni agente ha un ruolo preciso nel contesto del sistema e pertanto ha delle responsabilità obbligatorie e necessarie.

  Un manager, per esempio, deve periodicamente tastare il polso al gruppo di lavoro per capire a che punto è arrivato il progetto complessivo.

- I diritti associati al ruolo.

  Sono le risorse allocate al ruolo specifico che consentono di far rispettare le proprie responsabilità, per esempio le informazioni atte a dare la possibilità ad ogni agente di applicare le proprie funzioni.

- Le azioni possibili di un ruolo.

Sono le azioni private che ogni agente può compiere al fine della conquista dei propri obiettivi.

- I protocolli.

  Definiscono le regole da rispettare nell'interazione con gli altri agenti.

A partire da un'architettura specifica esistono varie metodologie di progettazione di sistemi multiagente.

Genericamente una metodologia prevede una serie di passi successivi che portano da un'iniziale visione di più alto livello del problema, al singolo dettaglio implementativo del codice automaticamente generato da un tool CASE.

Alcuni fra i passi da seguire potrebbero essere:

- Individuazione dei requisiti del Sistema.
- Individuazione degli obiettivi del Sistema.
- Individuazione della Società di agenti.
- Individuazione delle regole sociali, delle norme e leggi per creare interazioni dinamiche fra agenti.
- Identificazione dei ruoli degli agenti. (Ogni agente può mutare il proprio ruolo a runtime in funzione del contesto).
  - Identificazione delle responsabilità per ciascun ruolo.
  - Identificazione delle funzionalità per ciascuna responsabilità.
  - Identificazione dei tasks per ciascuna funzionalità.
  - Identificazione delle azioni possibili per ciascun task.
- Individuazione dei singoli agenti e loro implementazione.

- Descrizione della base di conoscenza di ciascun agente.
- Individuazione dei protocolli di comunicazione fra agenti e loro descrizione.
- Modellizzazione del codice.
- Localizzazione logistica degli agenti nel sistema distribuito.

Esistono quindi due livelli principali di astrazione del problema nell'ambito di una metodologia di progetto di un sistema multiagente, uno inteso in termini di sistema e uno in termini di singolo agente.

Quello a livello di sistema permette di descrivere il livello sociale e di interazione fra agenti.

Quello a livello di agente consente di scendere a livello di dettaglio e di generazione del codice.

Ogni metodologia inevitabilmente introduce una visione particolare di agente e di sistema, con pregi e difetti.

Essa e' strettamente legata all'architettura di base adottata e non può che essere un'estensione, seppur vincolata, del concetto di agente introdotto dalla infrastruttura sottostante.

2.2 Definizione di agente.

L'esistenza di diversi modelli di architetture ad agenti ha fatto sì che inevitabilmente ognuno di essi abbia proposto una definizione propria di agente intelligente.

Ad esempio, secondo le specifiche FIPA98, indicate nel documento di descrizione della architettura, un agente è inteso come un processo computazionale che implementa concetti di autonomia e di funzionalità comunicative di un'applicazione.

Allo scopo di comunicare l'agente usa un linguaggio di comunicazione ACL.

In alternativa un agente è visto come un'entità autonoma, fondamentale in un dominio di esistenza, capace di offrire uno o più servizi agli altri, in un modello di esecuzione integrato e

unificato che può includere anche l'accesso a software esterno, a utenti umani e a mezzi di comunicazione fra i più disparati.

Secondo C. Bernon e M. P.Gleizes si può usare la definizione di Ferber come punto di partenza e dire che un agente è una entità virtuale o fisica, capace di fare delle azioni in un ambiente dinamico, di comunicare direttamente con altri agenti, di farsi guidare da stimoli, sotto forma di obiettivi individuali e/o di soddisfazioni personali o istinti di sopravvivenza, che implementa mediante funzioni da ottimizzare; possiede risorse proprie; è capace di percepire il proprio ambiente, anche se in modo limitato; ha una rappresentazione parziale dell'ambiente che lo circonda e possiede la conoscenza del mondo esterno e delle azioni che può eseguire; può offrire dei servizi; può essere in grado di riprodursi; le sue azioni tendono a soddisfare il proprio obiettivo, tenendo conto delle risorse e della conoscenza che ha a disposizione e dipendendo esclusivamente dalle sue percezioni, dalla sua rappresentazione del mondo e dalle comunicazioni che riceve.

Il suo ciclo di vita in genere è fatto dalla terna percezione, decisione e azione.

Secondo IBM un agente intelligente è un'entità software che è in grado di eseguire delle operazioni per conto di un utente o di un altro programma, con un certo grado di indipendenza o autonomia, per il raggiungimento di determinati obiettivi.

Si tratta quindi di un'entità intesa come un programma o un componente hardware che esegue dei task o delle operazioni al posto del suo utente, in un ambiente dinamico.

A differenza di altre applicazioni software, l'agente ha attributi come quelli di autonomia, di mobilità, è orientato agli obiettivi, ha capacità di cooperazione e abilità nell'interagire con altri agenti, indipendentemente dalla presenza o meno del suo utente, è capace di comunicare ed ha abilità sociali.

Per diventare intelligente, l'agente software ha capacità "intellettive" di adattamento, oltre che di ragionamento logico.

Secondo l'architettura Masif un agente è un programma che agisce autonomamente per conto di una persona o di una organizzazione.

Per garantire la portabilità viene sviluppato in Tcl o Java.

Ogni agente ha il suo thread di esecuzione, così i task possono essere eseguiti su sua diretta iniziativa.

In Aglet un agente è "uno che agisce per conto di un altro", quindi un programma che agisce per un utente, con l'aggiunta di un certo grado di autonomia. Ha quindi la capacità di scegliere in funzione di un contesto legato ad un ambiente dinamico.

In particolare poi un agente mobile, in Aglet, è tale da non rimanere legato al sistema dove inizia l'esecuzione del suo thread.

Ha quindi la capacità di spostare se stesso e il proprio stato da un ambiente ad un altro.

Secondo la OMG gli agenti sono oggetti che possono muoversi e iniziare o fermare la loro esecuzione autonomamente.

2.3 Architettura FIPA

La prima architettura ad agenti che andremo ad analizzare è quella descritta nelle specifiche standard proposte dalla FIPA.

FIPA è l'acronimo di Foundation for Intelligent Physical Agents. (http://www.fipa.org/)

Si tratta di una organizzazione internazionale non a scopo di lucro, dedicata alla promozione del proprio concetto di sistema orientato agli agenti.

Il punto di partenza della visione data da FIPA sugli agenti è quello di far sì che sistemi orientati agli agenti, assolutamente eterogenei, possano interagire fra loro.

Da una completa collaborazione fra strutture private e pubbliche nasce lo standard Fipa.

Secondo questa architettura, tutto e' un agente, ossia una entità autonoma che offre servizi al mondo che lo circonda e che ha uno scopo, ed è proprio per il raggiungimento di tale scopo che usufruisce dei servizi offerti da altri agenti.

Un sistema Fipa è una struttura dinamica che prevede dei sottosistemi quali quelli dedicati alla gestione degli agenti, alla gestione dei servizi da loro offerti, alla loro ontologia, al servizio di trasporto dei messaggi, alla gestione delle interfacce verso sistemi eterogenei.

Il sottosistema di gestione degli agenti deve potere prevedere i servizi che seguono:

- Creazione di un agente.
- Registrazione di un agente.
- Localizzazione di un agente.
- Comunicazione fra agenti.

- Migrazione di un agente.
- Distruzione di un agente.

ossia l'**AMS (Agent Management System)** deve potere creare un agente, o una istanza ad un suo modello; deve potere registrare la presenza di un agente nel sistema, senza registrazione un agente non può *vivere* nel sistema; deve essere in grado di localizzarlo, per potere avere il pieno controllo su tutte le sue attività; deve garantire il controllo delle interazioni, delle comunicazioni fra agenti; deve controllare la mobilità degli agenti entro il sistema o fra sistemi diversi; deve infine essere in grado di cancellare agenti ormai inutili o pericolosi alla sicurezza del sistema.

L'AMS al momento della creazione di un agente sfrutta un sottosistema di identificazione che gli associa un identificativo unico e globale in termini di visibilità.

Ogni agente e' una entità autonoma, che offre dei servizi e come tale è necessario che si pubblicizzi dinanzi al mondo usufruendo di un metodo simile alle nostre pagine gialle, chiamato DF.

Una **Directory Facilitator (DF),** è una sorta di blocco di pagine gialle, in cui ogni agente si registra, dichiarando i servizi che offre, oppure cercando i servizi offerti da altri agenti.

Genericamente un DF ingloba un **ADS** e un **SDS**, ossia un agent-directory-service e un service-directory-service, rispettivamente un servizio di elencazione degli agenti registrati e un servizio di elencazione dei servizi a disposizione.

Al fine di garantire il corretto scambio di informazioni fra gli agenti deve essere presente un **Message Transport Service – MTS,** ossia un servizio di trasporto dei messaggi.

A cappello di tutto questo, per far vivere un sistema Fipa, deve essere presente una piattaforma fisica in cui gli agenti possono coesistere, ed esattamente un host con il suo sistema operativo e il software di supporto agli agenti.

Pertanto una prima definizione di sistema Fipa può essere la seguente:

- Sistema Hardware.
- Sistema Operativo.
- Software di supporto.
- AMS , DF e MTS.

2.3.1 Identificazione di un agente.

Un agente è definito attraverso un gruppo di informazioni elementari ed esattamente:

- Un nome, obbligatorio, ad esempio *df@hap_name*.

ossia un identificatore unico e globale all'interno del sistema, in *df@hap_name* *hap_name* è il nome dell' host di appartenenza dell'agente.

- Una lista di indirizzi a cui un messaggio può essere inviato da parte di altri agenti.

ossia una sequenza opzionale e ordinata di indirizzi di trasporto, dove l'agente può essere contattato; questi puntano tutti allo stesso agente e ad ogni indirizzo corrisponde una metodologia di trasporto diversa.

L'ordine stabilisce la priorità da rispettare affinché si possa contattare l'agente secondo le sue esigenze.

- Uno o più localizzatori, ossia una lista di indirizzi che puntano ad agenti dedicati alla ricerca di altri agenti.

Si tratta di una lista di *resolvers* opzionale, ossia una sequenza ordinata di AIDs (Agent Identifiers) che corrispondono a servizi di risoluzione dei nomi, anche in questo caso la sequenza fissa la priorità di utilizzo dei resolvers nei confronti dell'agente in questione.

Esempio:

```
(agent-identifier
 :name agent-b@bar.com
 :resolvers (sequence
  (agent-identifier
   :name ams@foo.com
   :addresses (sequence iiop://foo.com/acc))))
```

In questo caso l'agente si chiama agent-b@bar.com e ams@foo.com è il servizio di risoluzione dei nomi, posto all'indirizzo *iiop://foo.com/acc*.

Se un altro agente interroga il risolutore ams@foo.com, questi restituirà la descrizione dell'agente solo se lo trova registrato, viceversa darà un messaggio di fallimento della ricerca.

Due agenti sono intesi come uguali se hanno lo stesso nome identificativo.

1.2 Directory Facilitator (DF).

Una volta che un agente e' stato creato, poiché offre dei servizi, genericamente si deve registrare presso uno o più DF, che offre un servizio di pagine gialle agli agenti.

Un DF mantiene un accurato, completo, duraturo e continuamente aggiornato elenco di agenti e dei loro corrispettivi servizi.

In una applicazione possono essere presenti uno o più DF, nel qual caso questi possono creare una confederazione in piena collaborazione.

Un agente che vuole pubblicizzare i propri servizi agli altri agenti individua un DF, si registra richiedendo l'iscrizione del suo descrittore.

Quando si registra prevede:

- un nome.
- i servizi che offre e a che costo.
- le ontologie che comprende.
- uno o più locator che permettono di individuare l'agente in termini di metodologie di trasporto supportate dall'agente.

In certi casi un agente può anche essere cancellato, per sua volontà o per volontà del DF in cui si trova il suo descrittore.

Oltre alla semplice registrazione il DF mostra la sua importanza quando un agente richiede informazioni circa un altro agente e/o relativi servizi.

Il DF comunica chi offre un determinato servizio, non garantendo nulla sull'effettivo risultato, in quanto basa il responso della sua ricerca sull'immagine che in quel momento ha del sistema.

Implementa altresì una restrizione di accesso a chi si vuole registrare, a chi vuole modificare parametri di servizio di un agente e a chi richiede informazioni.

Un DF e' un agente con un AID (Agent Identifier) particolare, come il seguente.

(agent-identifier

:name df@*hap_name*

:addresses (sequence *hap_transport_address*))

il nome è *df* presso l'host *hap_name*.

Deve essere in grado di gestire le seguenti funzioni:

- Registrazione di un agente.
- Deregistrazione di un agente.
- Modifica del descrittore di un agente.
- Ricerca di uno o più agenti mediante pattern.

In merito alla ricerca, un DF lavora prima localmente e poi estende l'attività ad altri DF eventualmente confederati.

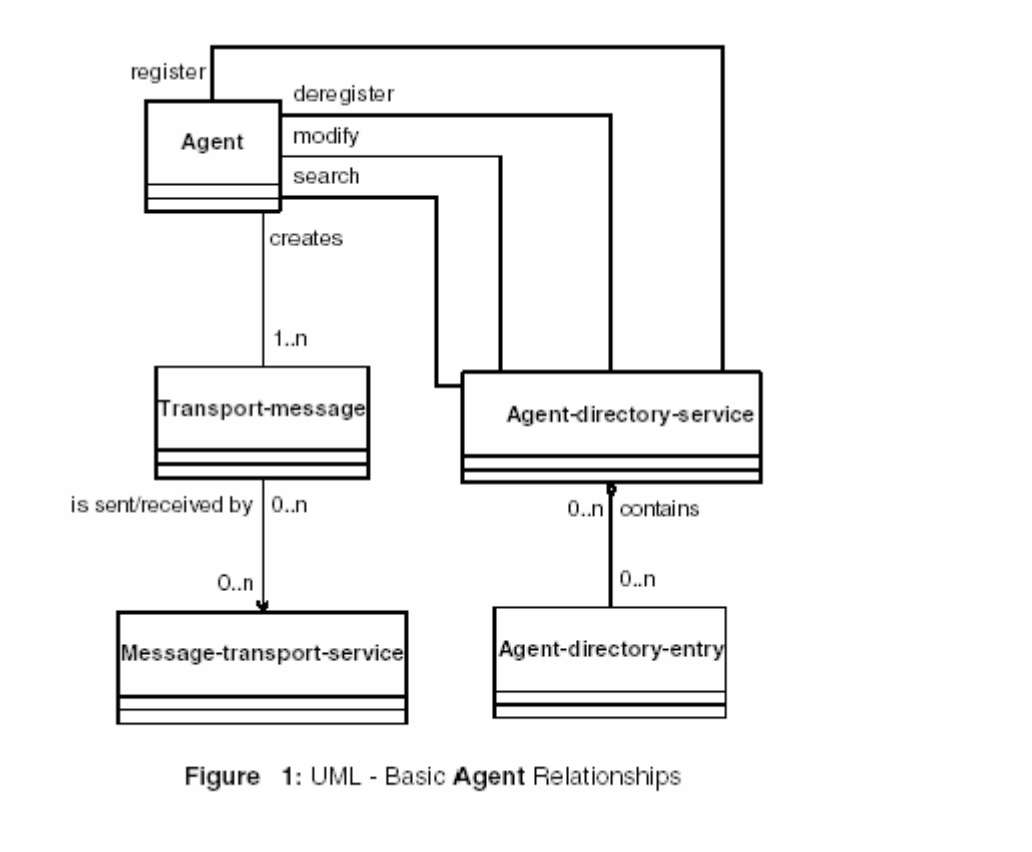


Figure 1: UML - Basic **Agent** Relationships

Questa immagine racchiude molte informazioni, infatti si possono notare l'*Agent-directory-service*, sinonimo di DF, che contiene da 0 a n *Agent-directory-entry* ossia istanze di registrazione da parte di altrettanti agenti concluse con successo.

Un agente relativamente alla propria iscrizione in un DF può registrarsi, cancellare la propria registrazione, modificare il proprio descrittore, ricercare informazioni presso il DF.

Da questo schema UML si evidenzia anche il fatto che un agente possa creare da 1 a n messaggi di trasporto, inviati o ricevuti da 0 a n servizi di trasporto di messaggi MTS.

In un DF oltre che per gli agenti, è possibile la registrazione dei servizi.

Ogni servizio registrato è caratterizzato da :

- un nome, che deve essere unico e globale.
- un tipo di servizio.
- una *entry* in un DF di servizi.
- un *locator* che permette di accedere al servizio.

Il nome deve servire sia per la registrazione, che per la modifica, la deregistrazione e la ricerca.

Ogni registrazione di un servizio presso un DF, in questo caso definibile come “service-directory-service”, prevede:

- il servizio offerto.
- il costo di tale servizio.
- le proprietà offerte.
- le ontologie sostenute.
- i nomi e i nickname con cui il servizio e’ noto alla comunità.

2.3.2 Agent Management System - AMS

Come abbiamo già visto sopra, un AMS è il sistema di gestione degli agenti; non può che essere unico per ogni singola piattaforma.

Essendo esso stesso un agente, ha un identificatore specifico:

(agent-identifier

:name ams*@hap_name*

:addresses (sequence *hap_transport_address*))

Fra le funzionalità che deve supportare abbiamo:

Registrazione di un agente e/o piattaforma.

- Deregistrazione di un agente e/o piattaforma.
- Modifica del descrittore di un agente e/o piattaforma
- Ricerca di descrittori di agenti o del descrittore di una piattaforma.

le attività che seguono sono relative al rapporto fra singolo agente e sistema AMS:

- Sospensione di un agente

  ossia la possibilità di sospendere o mettere in stand-by un agente, bloccando temporaneamente le sue attività.
- Cancellazione di un agente.
- Creazione di un agente.
- Riattivazione della esecuzione di un agente, dopo la sua sospensione.
- Invocazione di un agente.
- Esecuzione di un agente.
- Gestione delle risorse.

Un AMS all'atto della registrazione descriverà un agente attraverso il suo identificativo, il suo proprietario e lo stato.

Descrive altresì una piattaforma di agenti con un nome e con una lista di servizi offerti agli agenti residenti.

Ogni servizio avrà un nome, un tipo e degli indirizzi per referenziarli.

2.3.3 Message Transport Service – MTS

Il sistema MTS gestisce lo scambio dei messaggi fra agenti entro la stessa applicazione o fra applicazioni diverse.

Prevede un message buffering service, ossia un servizio di conservazione in memoria dei messaggi, quando questi non sono recapitabili subito al destinatario.

Poiché spesso si ha a che fare con invio di messaggi tra sistemi remoti e pertanto collegati con metodologie in alcuni casi insicure come le wireless, può capitare di dovere criptografare i messaggi.

Pertanto viene implementato un sistema di sicurezza che prevede l'inserimento nell'inviluppo di:

- dati per la validazione del messaggio.
- dati per la criptatura e decriptatura.

Il sistema di sicurezza deve garantire un sottosistema di identificazione degli agenti e dei servizi mediante una firma, che permette di avere le credenziali di riconoscimento e di abilitazione necessarie.

Mediante le credenziali si potranno implementare delle policies di controllo, l'autenticazione del contenuto di un messaggio, ossia la verifica che non e' stato modificato da nessuno rispetto alla versione originaria durante il trasporto.

Si potranno implementare delle privacies di contenuto, ossia far sì che solo alcune designate identità possono esaminare il contenuto di un messaggio o di altre entità, mediante crittografia e/o trasporto su canali crittografati sicuri.

La validazione delle credenziali dovrà avvenire da parte di una autorità di validazione opportuna.

In conclusione i tre sistemi AMS, DF ed MTS costituiscono l'infrastruttura base di un macrosistema Fipa.

Il modello appena descritto e' il seguente:

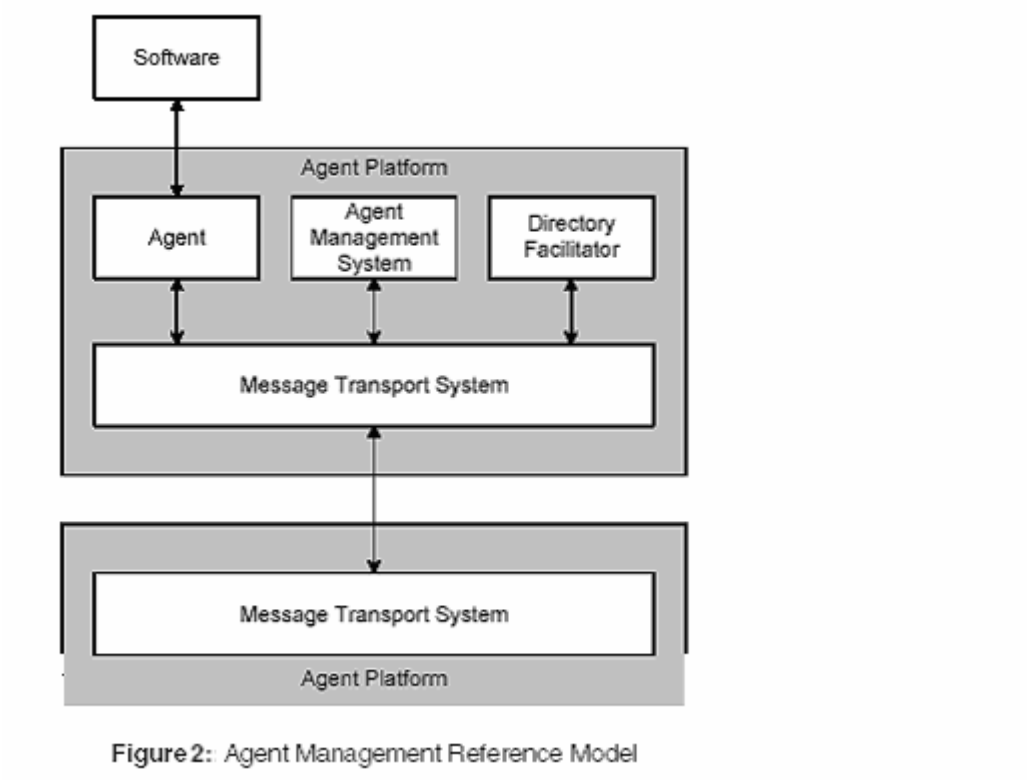


Figure 2: Agent Management Reference Model

2.3.4 Ciclo di vita di un agente.

Un agente prevede un ciclo di vita segnato da stati diversi.

La *vita* di un agente cambia in funzione della sua destinazione d'uso.

Un agente può essere:

- limitato ad una applicazione.

ossia visibile entro l'applicazione in cui e' stato creato.

- Indipendente dall'applicazione.

ossia i suoi servizi possono essere utili al di là di una singola applicazione, come nel caso di un DF.

- Instance-Oriented

ossia vive solo se lo istanzio, viceversa risulta inattivo.

- Unico

esiste in una sola applicazione e non muta mai il proprio stato.

Gli stati possibili di un agente nel suo ciclo di vita sono:

- Attivo

l'agente e' attivo per cui l'MTS può indirizzargli messaggi.

- Iniziato/In attesa/Sospeso

l'MTS memorizza i messaggi indirizzati all'agente e quando questo si riattiva glieli invia, oppure reindirizza i messaggi ad una nuova locazione dell'agente.

- In transito.

l'agente e' dotato di mobilità ed è in transito verso un'altra applicazione, per cui l'MTS memorizza i messaggi finché il comando di movimento dell'agente e' fallito, oppure l'agente si trova sulla nuova applicazione.

- Sconosciuto

in questo caso il sistema AMS non conosce lo stato dell'agente.

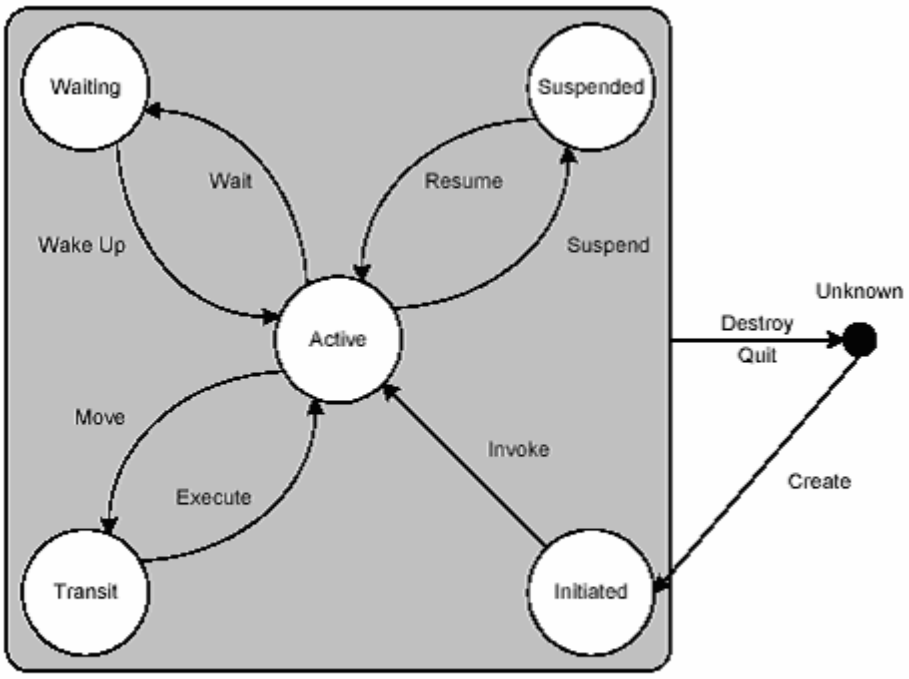


Figure 3: Agent Life Cycle

Un agente può essere:

- Creato
- Invocato
- Distrutto

  solo il sistema AMS può farlo e in ogni caso l'agente non può rifiutarsi.

- Terminato

  Questa azione può essere ignorata dall'agente.

- Sospeso

  Tale azione può essere effettuata dal sistema AMS o da un agente che ha le credenziali per poterlo fare.

- Riattivato

  Il sistema AMS può riattivarlo dopo la sospensione.

- In attesa

Questo stato è richiesto in genere da un agente.

- Risvegliato

Solo il sistema AMS può risvegliare un agente dallo stato di attesa.

Nel caso di agenti dotati di mobilità sono previsti anche i due stati che seguono:

- In movimento

Un agente può richiedere tale stato, mettendosi in fase transitoria, quando deve muoversi da un sistema ad un altro.

- Esecuzione

Riprende un agente dallo stato transitorio di movimento, può essere imposto solo dal sistema AMS.

2.3.5 Registrazione di un agente.

Un agente richiede la registrazione ad un AMS o a un DF quando:

- Viene creato.
- E' migrato da un'altra applicazione.
- Viene registrato assieme alla sua piattaforma.

La registrazione di un agente ha sempre un limite di tempo oltre il quale si può decidere di cancellare l'iscrizione.

All'atto della registrazione su un DF un agente specifica nella sua richiesta il limite di tempo entro il quale mantenere la registrazione.

Il DF può decidere di accettare o no la richiesta e se accetta informa l'agente con un messaggio di ritorno che il tempo stabilito e' identico a quello richiesto.

Se invece non viene accettata la richiesta, oppure l'agente non specifica alcun tempo limite, e' il DF a scegliere tale valore e a comunicare all'agente tale limite.

Può capitare che il DF non supporti limiti di tempo, pertanto nel contenuto del messaggio di ritorno, toglierà il parametro legato a questa informazione e sarà attribuito un tempo illimitato.

Un agente può usare il comando di modifica per aggiornare la sua registrazione e per rinnovare il tempo limite.

In ogni caso sarà sempre il DF a stabilire i valori definitivi.

Un DF descrive un agente appena registrato attraverso le seguenti informazioni:

- Nome dell'agente, opzionale.
- Servizi offerti, opzionale.
- Protocolli supportati dall'agente, opzionale.
- Ontologie note all'agente, opzionale.
- linguaggi noti all'agente fra fipa-SL, fipa-SL0, fipa-SL1, fipa-SL2, parametro opzionale.
- Tempo limite, ossia la durata o la data massima dopo la quale l'agente perde la registrazione.

Un servizio viene descritto in un DF attraverso le seguenti caratteristiche:

- Nome, opzionale.
- Tipo di servizio.
- Protocolli supportati.
- Ontologie note al servizio.
- Linguaggi noti al servizio.
- Proprietario del servizio.
- Proprietà che consentono di caratterizzarlo rispetto ad altri.

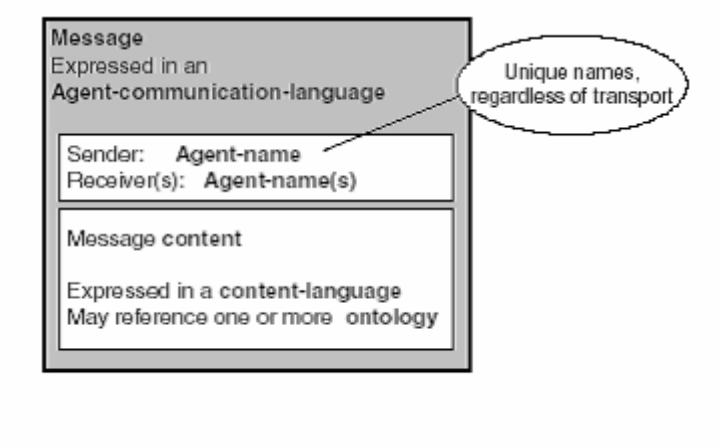
Noti gli agenti e i servizi registrati in un DF, la ricerca da parte di chi necessita informazioni da una Directory Facilitator prevede la possibilità di fissare la massima e la minima profondità di ricerca con un valore intero positivo, per esempio il valore 1 (uno) presuppone che il DF debba estendere la ricerca solo a se stesso, viceversa laddove il parametro sia negativo la ricerca si dovrà estendere a tutti i DF della confederazione.

Un agente può richiedere anche un numero massimo di risultati, se imposta un valore negativo per questo parametro allora li otterrà tutti.

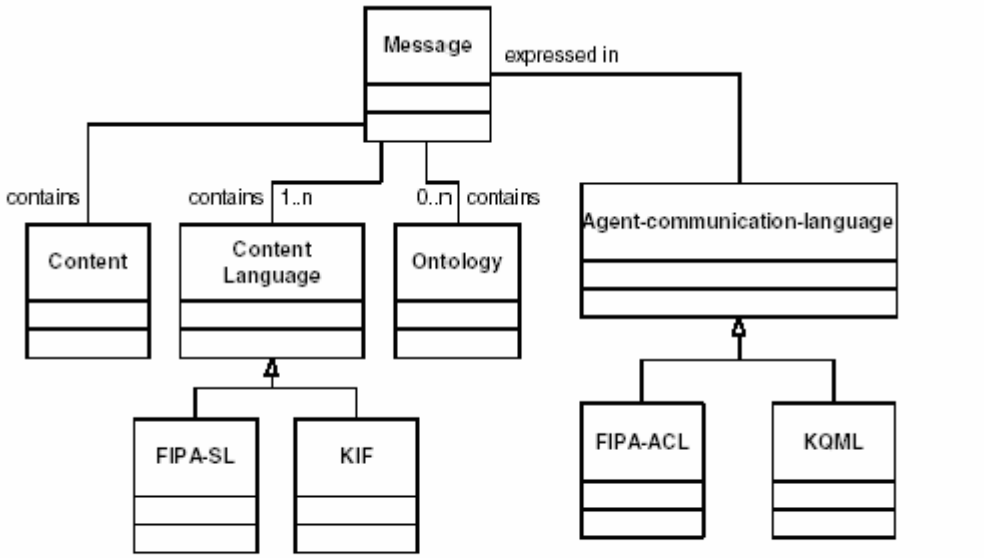
La ricerca in un DF presuppone l'uso opzionale di un identificativo unico e globale.

2.3.6 Messaggi ACL

L'architettura Fipa è fortemente orientata all'uso dei messaggi nella interazione fra agenti.



Un messaggio è espresso in ACL (Agent Communication Language) ossia un linguaggio utile alla comunicazione fra agenti, ed è fatto da una lista di parametri.



UML - **Message** Elements

In questa immagine abbiamo la rappresentazione in UML (Unified Modelling Language) degli elementi di un messaggio.

Questo è espresso in Fipa-Acl o in Kqml, che sono due diversi ACL, ha altresì un contenuto (Content), ha da uno a enne linguaggi di contenuto fra Fipa-SL e KIF, infine riconosce da 0 a enne ontologie, ossia basi di conoscenza diverse.

Fra i possibili componenti di un messaggio abbiamo:

- Receiver/s (uno o più identificativi di agenti che devono ricevere il messaggio)
- Sender (identificativo di agente che invia un messaggio)
- Content (contenuto del messaggio che voglio inviare espresso in CL – Content Language)
    - Esempi di Content Language sono:
        - FIPA-SL
        - FIPA-RDF
        - FIPA-KIF
        - FIPA-CCL

- Language (linguaggio Fipa usato, ad esempio fipa-sl0 o fipa-sl1)
- Encoding (tipo di codifica del messaggio)

In caso di codifica l'agente passa il messaggio e il tipo di codifica da usare al servizio di encoding, oppure può richiedere quali tipi di codifica offre il servizio per scegliere opportunamente.

Infatti l'*encoding service* prevede la possibilità di rendere note le tecniche di decodifica che offre, pertanto può ricevere richieste di query circa la lista che può offrire.

Alla fine il messaggio codificato, dopo l'invio, viene dato all'*encoding service* di destinazione, che estrae il payload e lo decodifica.

- Ontology (il tipo di ontologia usata dal messaggio)

Una ontologia prevede un vocabolario per comunicare e rappresentare la conoscenza di qualcosa; e' un insieme di coppie di simboli e descrizioni che e' condiviso da tutte le entità che ne richiedono l'uso e descrive simboli e/o relazioni fra oggetti.

Descrive altresì vincoli di sistema e proposizioni vere.

- Protocol (protocollo usato)
- Reply-with (tale parametro e' seguito dal nome del task con cui replico alla richiesta)
- In-reply-to
- Reply-by (e' un tempo o una data entro cui il sender vorrebbe una risposta dal/dai receiver/s)
- Reply-to (specifica a chi replicare)
- Conversation-id (identificativo unico di conversazione fra due agenti)
- User-defined parameters (parametri definiti dall'utente, per garantire l'espandibilità del linguaggio).

Un messaggio ACL in genere comunica un'azione con la coppia act (…) e conversation-id, dove al posto dei puntini (…) abbiamo un'azione fra quelle giù elencate e conversation-id è

l'identificativo della conversazione, unico per tutta la durata dell'interazione fra gli agenti interessati.

Le azioni previste sono:

- accept-proposal (l'agente *receiver* risponde accettando una proposta)
- agree (messaggio di conferma da parte del receiver)
- cancel (messaggio di cancellazione dell'interazione fra i due interlocutori)
- cfp (chiamata da parte di un agente per una proposta – call for proposal)
- confirm (conferma)
- disconfirm (rinunzia)
- failure (fallimento di una azione)
- inform (messaggio usato come informazione)
- not-understood (messaggio di non avvenuta comprensione da parte del receiver)
- propose (messaggio di proposta da parte del sender)
- query-if (richiesta di una query in funzione del valore di una proposizione logica)
- query-ref (richiesta di una query che restituisce riferimenti ad agenti)
- refuse (messaggio di rifiuto)
- reject-proposal (messaggio di rigetto di proposta)
- request (richiesta da parte del sender)
- request-when (richiesta che deve essere soddisfatta quando si verifica una condizione)
- request-whenever
- subscribe (messaggio di sottoscrizione)
- inform-if (messaggio di informazione legato ad una condizione booleana)

- inform-ref
- proxy
- propagate

Sono tutte azioni descritte nella sezione dei protocolli di comunicazione.

Nelle specifiche dell'architettura Fipa un messaggio ACL può essere rappresentato in vari modi come ad esempio in XML (Extensible Mark-up Language), oppure in "bit-efficient-mode", che mira all'obiettivo di comunicare attraverso un numero ristretto di bit quanto più limitato possibile.

Un messaggio ACL nel secondo caso è visto come una stringa binaria suddivisa fondamentalmente in quattro parti:

- Header (una intestazione che segna l'inizio del messaggio)
- Tipo di messaggio.
- Parametri (anche nessuno).
- Trailer (ossia una coda di bit a chiusura del messaggio).

Il contenuto binario di ogni singola parte è a sua volta suddiviso in byte e questi rappresenteranno azioni note e parametri, sia nel tipo sia nei valori.

Per esempio un header conterrà un identificativo univoco di messaggio e un identificativo di versione.

L'identificativo del messaggio prevede un byte che permette di specificare se si vorrà usare una tabella dinamica di codici, descritta più avanti, oppure no, mentre il campo versione è un byte che identifica appunto la versione usata nel codificare il messaggio.

Il Trailer analogamente sarà una sequenza di byte opportunamente precostituita per identificare la fine del messaggio.

Fra l'Header e il Trailer si ha il "tipo di messaggio" che può essere preso da una lista predefinita di azioni, vista sopra, ad ognuna delle quali è associato un codice binario predefinito, oppure può essere un tipo definito dall'utente, in questo caso si dovrà rispettare una sequenza di byte specifica, seguita da una stringa binaria che conterrà eventuali parametri.

Sempre all'interno del messaggio e subito dopo il tipo di messaggio sono presenti eventuali parametri, anch'essi potranno essere predefiniti o definiti dall'utente, in quest'ultimo caso seguirà la codifica binaria del nome del parametro e poi il valore ad esso attribuito.

Questo tipo di codifica è prevista nel caso in cui si debbano usare metodologie di comunicazione a banda stretta in cui è importante la dimensione della stringa binaria che si deve inviare.

Dall'analisi delle codifiche possibili di un messaggio si evidenzia l'uso del concetto di tempo nei sistemi multiagente e in particolare dal fatto di potere usare date, ossia anni, mesi, giorni, ore, secondi e millisecondi, in modo relativo o assoluto.

Sempre al fine di ottimizzare il numero di bit che devono essere inviati al receiver, le specifiche Fipa prevedono l'uso di una tabella di codici opzionale, che ha lo scopo di conservare stringhe binarie usate più frequentemente rispetto alle altre e associando ad esse un codice univoco permetterà di usarlo nella codifica del messaggio al posto della stringa intera.

Ovviamente questo è conveniente laddove la memoria resa disponibile dal sistema sia capiente.

Tale tabella, se esiste, deve in ogni caso essere resa visibile a tutti coloro che vorranno usufruirne.

Il servizio "Agent Message Transport" prevede una tecnica di incapsulamento dei parametri di un messaggio.

Questo ci consente di definire strutture di parametri da elaborare come sequenze indicizzate.

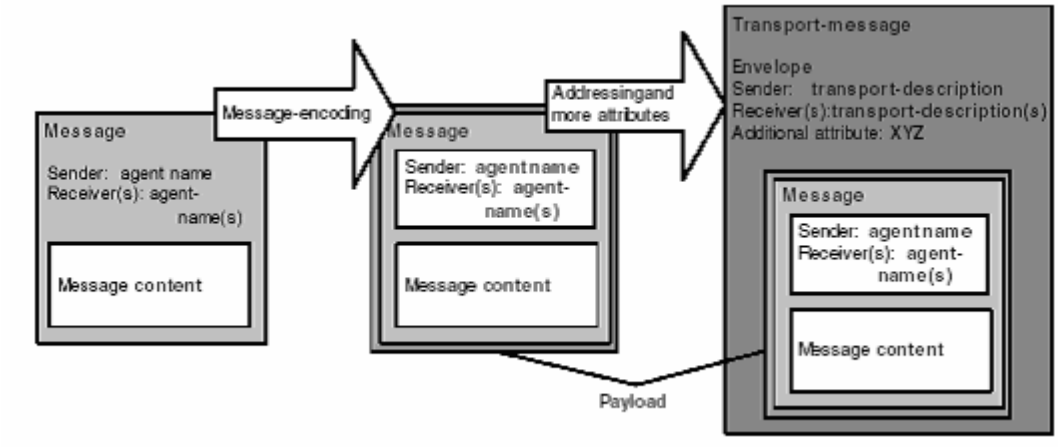
Fra i parametri degli inviluppi, che incapsulano il messaggio, sono previsti i seguenti:

To (seguito da uno o più identificatori di agente a cui mandare il messaggio)

- from (seguito da un identificatore di agente da cui proviene il messaggio)
- comments (commenti inseriti dall'utente)
- acl-representation (ha come valore il nome del componente Fipa usato per la codifica dei messaggi acl, ad esempio fipa.acl.rep.bitefficient.std, fipa.acl.rep.string.std, fipa.acl.rep.xml.std).
- payload-length (lunghezza in bytes del payload, ossia del contenuto del messaggio)

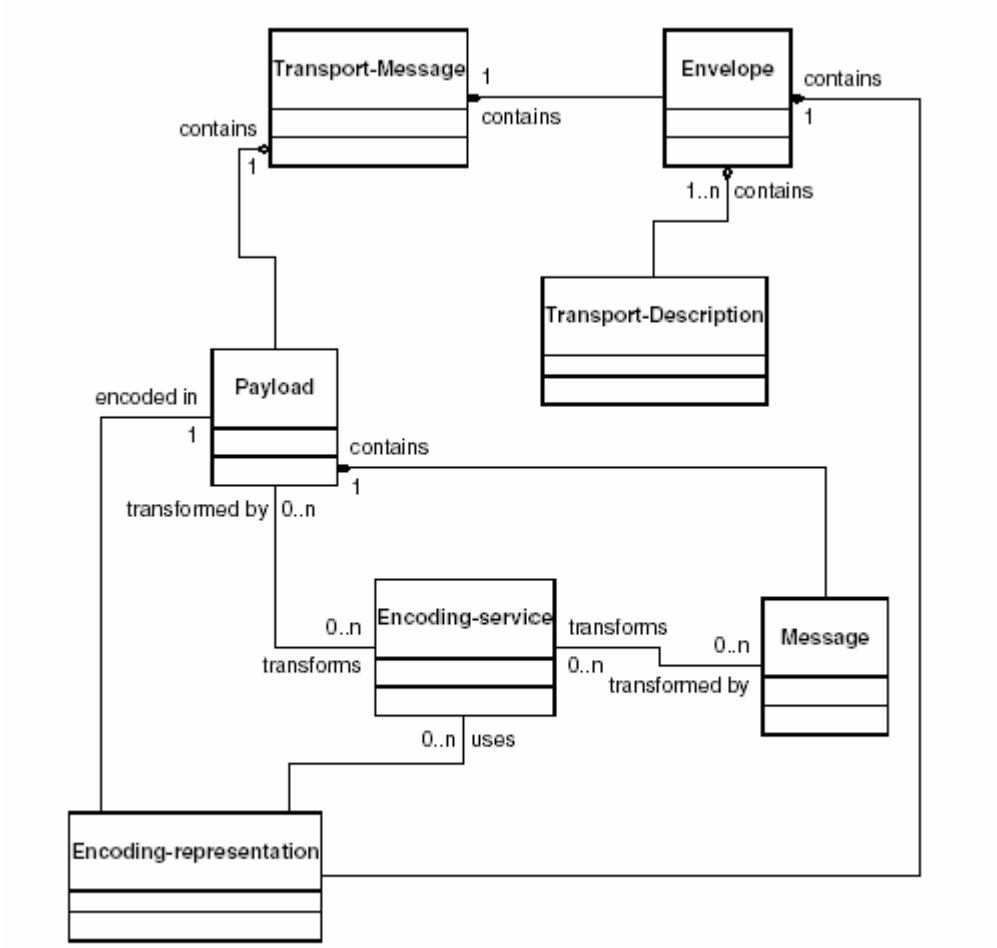
- payload-encoding (ha come valore il linguaggio di codifica del payload, per esempio US-ASCII, ISO-8859-1, ISO-8859-9, UTF-8, Shift_JIS, EUC-JP, ISO-2022-JP, ISO-2022-JP-2)
- date (ha come valore una data opportunamente codificata in standard ISO8601).
- encrypted (se il messaggio è criptato oppure no)
- intended-receiver (ha come valore uno o più identificatori di agente fissati come possibili receivers)
- received (ha come valore un identificatore di agente che ha ricevuto il messaggio)
- received-by (e' un url - Unified Resource Locator - che identifica un agente)
- received-from (e' un url - Unified Resource Locator - che identifica un agente)
- received-date (e' una data di ricezione del messaggio)
- received-id (identificativo del messaggio ricevuto)
- received-via (mezzo usato per la ricezione)
- user-defined (parametri utente)

queste informazioni sono tutte contenute in un inviluppo ossia la parte del messaggio di trasporto dedicata alla codifica di tutte quelle informazioni che servono al corretto indirizzamento e consegna del messaggio, il cui contenuto si trova nel payload.



In questa immagine un messaggio con un sender e uno o più receivers ha un contenuto, viene codificato, e gli vengono aggiunti attributi di indirizzamento, diventando così il payload di un messaggio di trasporto fatto da un inviluppo, che conterrà una descrizione della metodologia di trasporto del sender, una o più descrizioni dei receivers ed altri attributi.

In tale messaggio di trasporto verrà poi inglobato il contenuto del messaggio prima codificato.



UML - **Transport-Message** Relationships

In questa altra immagine sono evidenziate in UML (Unified Modelling Language) le relazioni fra i messaggi di trasporto.

Un messaggio di trasporto contiene un inviluppo, che conterrà da 1 a n descrizioni di trasporto possibili, una rappresentazione di codifica usata nel messaggio per codificare il payload, unico contenuto nel messaggio di trasporto.

Il payload contiene il messaggio trasformato da 0 a n servizi di codifica.

Un messaggio di trasporto può prevedere diverse metodologie di trasporto come ad esempio SMTP o HTTP.

In ogni caso verranno specificati nell'inviluppo corrispondente il tipo di trasporto, gli indirizzi dei risolutori per il trasporto e le proprietà principali, sia per il sender che per il receiver.

Anche gli inviluppi possono essere codificati in bit-efficient-mode al fine di ridurre la dimensione del messaggio da inviare.

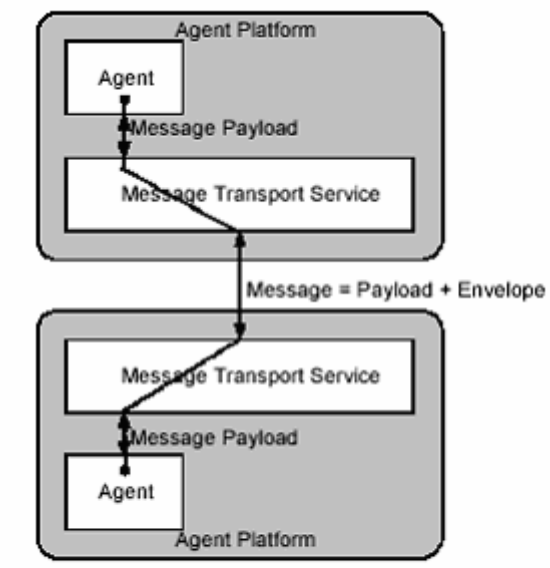
Lo standard Fipa prevede la possibilità di gestire l'invio dei messaggi opportunamente incapsulati, attraverso sistemi eterogenei mediante l'uso di protocolli di interfaccia, ad esempio l'Internet Inter-Orb Protocol (IIOP).

In funzione del protocollo usato, il messaggio e il suo inviluppo, vengono incapsulati in uno di più alto livello, come l'IIOP, e inviati.

La ricezione da parte di un agente prevede prima di tutto la decodifica IIOP e poi quella del messaggio ACL contenuto nell'inviluppo.

Quando un agente di una piattaforma vuole inviare un messaggio, usa il servizio di trasporto dei messaggi (MTS) della sua piattaforma.

Questo a sua volta racchiude il payload del messaggio nell'inviluppo contenente le informazioni utili al corretto indirizzamento del messaggio e lo invia a destinazione, ossia all'altro (MTS) servizio di trasporto dei messaggi della piattaforma destinataria, che lo smista all'agente finale, togliendo le informazioni di trasporto e isolando il payload.



Message Transport Reference Model

2.3.6.1 Linguaggio FIPA-SL.

Uno dei linguaggi col quale codificare entità quali azioni, proposizioni e formule scambiate fra agenti del sistema è il FIPA Semantic Language (SL) usato come Content Language per i contenuti dei messaggi.

Secondo la definizione classica di linguaggio e' prevista la definizione delle specifiche lessicali, sintattiche e semantiche dell' SL.

SL prevede l'uso di

- operatori unari logici e non logici.
- operatori binari logici e non logici.
- operatori *implica* ed *equivale.*
- operatori *iota*, *any*, *all* usati nel contesto di una KB (Base di Conoscenza).
- quantificatori universali come *per ogni* ed *esiste.*
- quantificatori esistenziali che esprimono ciò che un agente crede in merito ad una proposizione, ossia se sia vera o falsa o probabilisticamente vera.
- proposizione logiche.

- variabili.
- funzioni.
- azioni.
- costanti.
- sequenze.
- insiemi.

Esistono anche delle forme contratte di linguaggio SL, sono più semplici e destinate a sistemi di agenti che eseguono dei task non complessi, per cui il costo della gestione dell'intero linguaggio non sarebbe giustificabile.

Per esempio

FIPA SL0: sottoinsieme minimale di SL.

Usa azioni, proposizioni, costanti numeriche, stringhe e date, insiemi, sequenze, funzioni, parametri e predicati.

FIPA SL1: e' il modulo proposizionale che aggiunge al fipa SL0 gli operatori booleani unari e binari.

FIPA SL2: con Decidability Restrictions.

Permette di usare predicati di primo ordine e logica modale.

Usa gli operatori logici unari e binari, tutti gli operatori di SL come i modali, i quantificatori, gli operatori referenziali come gli iota, any e all.

2.3.7 Protocolli di interazione fra agenti.

L'interazione fra agenti in Fipa e' regolata da opportuni protocolli di comunicazione, ossia insiemi di regole di comportamento da adottare affinché un agente che voglia comunicare con un altro abbia la possibilità di farsi comprendere, di scambiare informazioni e usufruire di uno o più servizi.

Fra i protocolli sotto elencati troviamo casi di interazione uno a uno o punto-punto e casi di interazione uno a enne.

Un agente potrà quindi colloquiare con un solo interlocutore, oppure con un gruppo.

E’ previsto in ogni caso un sistema di identificazione delle interazioni o connessioni stabilite fra agenti.

Questo per far sì che ogni interazione sia univocamente identificata e comunque visibile in tutto il sistema multiagente.

Il servizio di attribuzione dell’identificativo deve prevedere anche la possibilità di cancellare una interazione, in caso di perdita di connessione e/o di esplicita chiusura della comunicazione da parte di uno dei partecipanti.

Un altro aspetto che si potrà evidenziare sarà quello secondo cui ogni agente, essendo autonomo, può “*decidere*” se accettare o meno una richiesta e comportarsi di conseguenza.

Inoltre e’ sempre possibile che un’azione intrapresa da un agente non giunga a compimento, oppure dia un risultato positivo da reinviare al mittente, tutto ciò è inevitabilmente legato alla dinamicità del sistema.

Un altro aspetto comune a tutti i protocolli e’ che la richiesta e la notifica da parte di agenti e’ sempre legata a servizi richiesti e offerti.

2.3.7.1 Request Interaction Protocol (RIP)

Il *Request Interaction Protocol* (RIP) e’un protocollo basilare fra coppie di agenti, costruito su richieste da parte dell'agente definito "*Initiator*", e notifiche da parte dell'agente definito come "*Participant*".

L'*initiator* ancor prima di effettuare una richiesta assegna al messaggio un identificativo unico e con visibilità globale, al fine di gestire i messaggi ACL (Agent Communication Language) scambiati con altri agenti.

Questo aiuta a gestire la storia delle comunicazioni effettuate e le strategie da adottare nel raggiungimento di un obiettivo prefissato.

Fondamentalmente l'i*nitiator* effettua una richiesta di interazione al p*articipant*, questi può a sua volta, decidendo opportunamente, rifiutarsi di accettare la richiesta o rispondere (ponendo a *true* il valore del parametro di ritorno *refused* o *agreed*), notificando la sua reazione e comunicando, in seguito all’esecuzione del servizio richiesto, una fra tre informazioni possibili:

- il fallimento dell'azione eseguita. (*failure*)

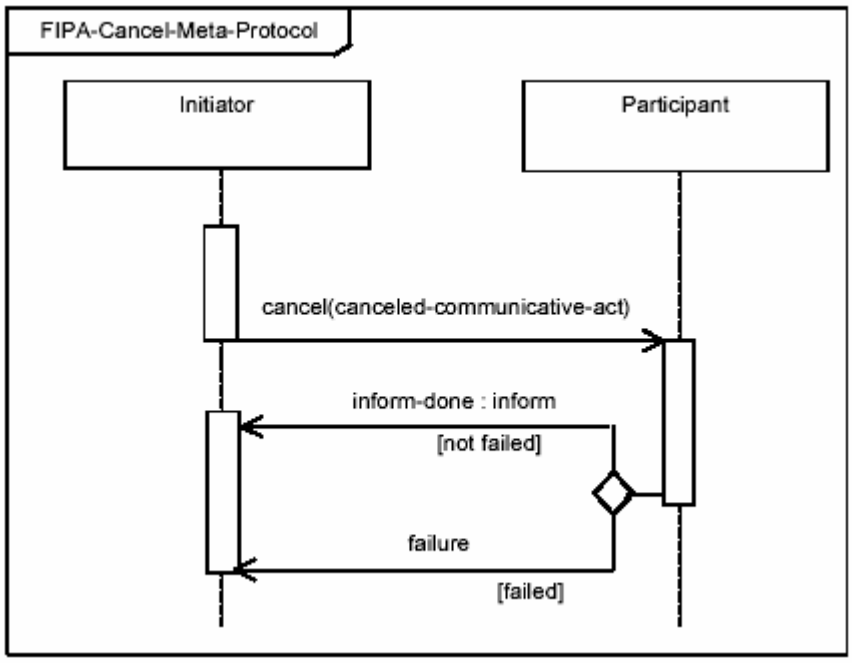
- il successo dell'azione. (*inform-done*)
- il successo dell'azione dotato di risultato. (*inform-result*)

Durante la comunicazione dei messaggi ACL si può verificare l'evento secondo il quale il *participant* non comprenda la richiesta, pertanto reagirà con un messaggio opportuno di non comprensione, cosa che potrà fare ripetere la richiesta all'*initiator*.

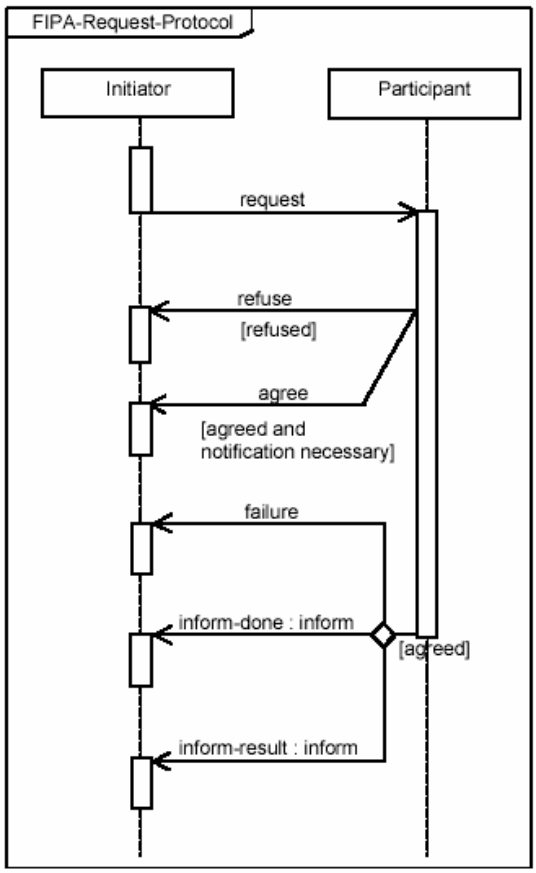
E' possibile che durante l'interazione RIP si perda la connessione fra i due agenti per un evento aleatorio o per esplicita volontà di uno dei partecipanti.

Tale evento si può verificare in qualsiasi istante e causa la perdita della interazione e di tutti i committments effettuati nel frattempo fra i due interlocutori.

E' compito dell'*initiator*, a questo punto, cancellare l'identificativo di conversazione, attraverso la notifica al *participant* di un *Cancel Meta interaction Protocol*, che usando lo stesso identificativo di comunicazione precedente, gestirà la cancellazione della interazione pregressa.



Tale azione e' a tutti gli effetti regolata da una interazione col *participant*, che può quindi rifiutarsi di accettare la richiesta di cancellazione con un *failure-message* o accettarla incondizionatamente.



3.2 Query Interaction Protocol (QIP)

Un protocollo più complesso del precedente e' il *Query Interaction Protocol (QIP).*

In questo caso l'*initiator* richiede una di due possibili azioni al *participant*:

- query-if
- query-ref

Nel primo caso l'*initiator* invia al *participant* una query per verificare se una determinata condizione sia vera o falsa.

Nel secondo caso l'*initiator* richiede determinati riferimenti al *participant*.

In QIP così come in RIP il *participant* può accettare la richiesta o rifiutarsi.

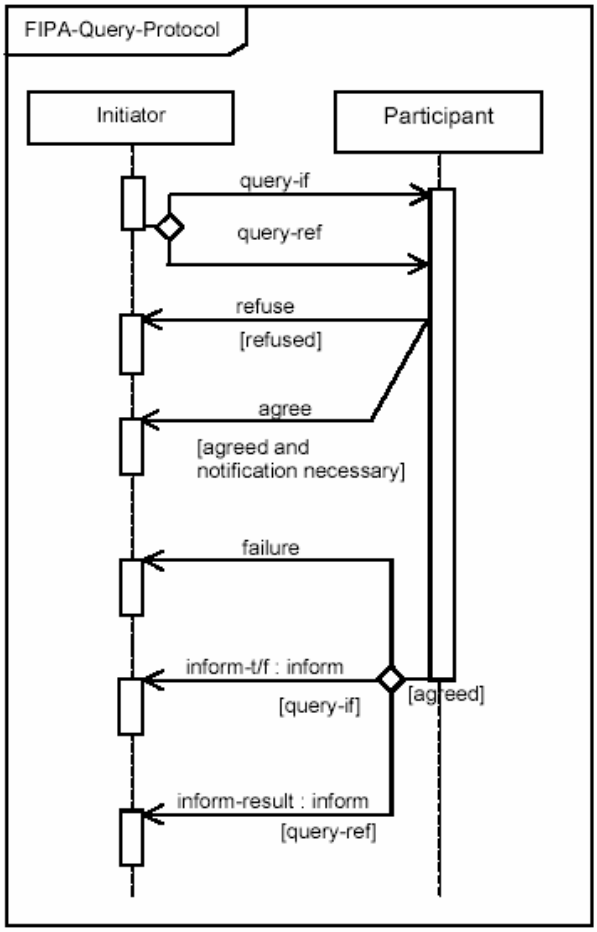
Se accetta si possono verificare i seguenti casi:

- il fallimento dell'azione eseguita. (*failure*)
- il successo dell'azione eseguita .

In relazione al secondo caso il *participant* usa un messaggio di tipo *inform-t/f* in risposta ad una *query-if* ed esattamente con un valore vero o falso come risultato della valutazione di una specifica proposizione logica.

Diversamente il *participant* ritorna un messaggio di tipo *inform-result* in risposta ad una *query-ref* con una espressione degli oggetti richiesti dall'*initiator*.

Anche per questo protocollo così come per i successivi, vale ciò che e' stato detto in merito all'identificazione del messaggio e alla mancata comprensione fra agenti con la relativa perdita di connessione.



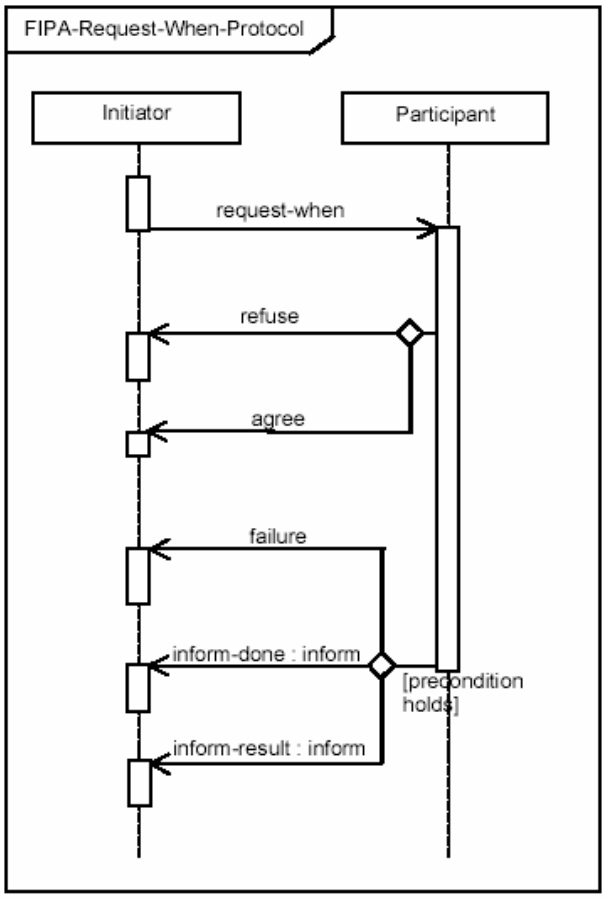
2.3.7.2 Request When Interaction Protocol (RWIP)

Il *Request When Interaction Protocol (RWIP)* permette ad un agente di richiedere un azione/servizio in seguito al fatto che una precondizione sia vera.

Il *participant*, secondo lo stesso schema visto sopra, o si rifiuta subito o accetta la richiesta e aspetterà finché la precondizione non risulterà vera.

In tale circostanza darà il responso di fallimento del servizio o di conclusione corretta con o senza risultato in funzione degli eventi.

Se durante l'attesa accadrà che il *participant* non è più in grado di effettuare l'azione, allora invierà allo *initiator* un messaggio di avvenuto fallimento.



2.3.7.3 Contract Net Interaction Protocol (CNIP)

Il *Contract Net Interaction Protocol (CNIP)* permette di interpretare un agente come un manager che desidera che altri agenti facciano qualcosa, ottimizzando una certa funzione caratteristica.

Per ciascun task un *participant* può proporre un servizio possibile, altri possono accettare la richiesta proponendo a loro volta soluzioni, oppure rifiutandosi.

All'inizio l'*initiator* comunica la richiesta di un servizio ad *m* agenti, specificando:

- una azione di tipo *cfp* (*call for proposals).*
- un task da eseguire.
- condizioni da rispettare.

Su *m participant*, visti come fornitori a cui viene inviato un ordine, solo *n* daranno un responso, e di questi *n*, *solo j* saranno le proposte, gli altri sono solo rifiuti di collaborazione.

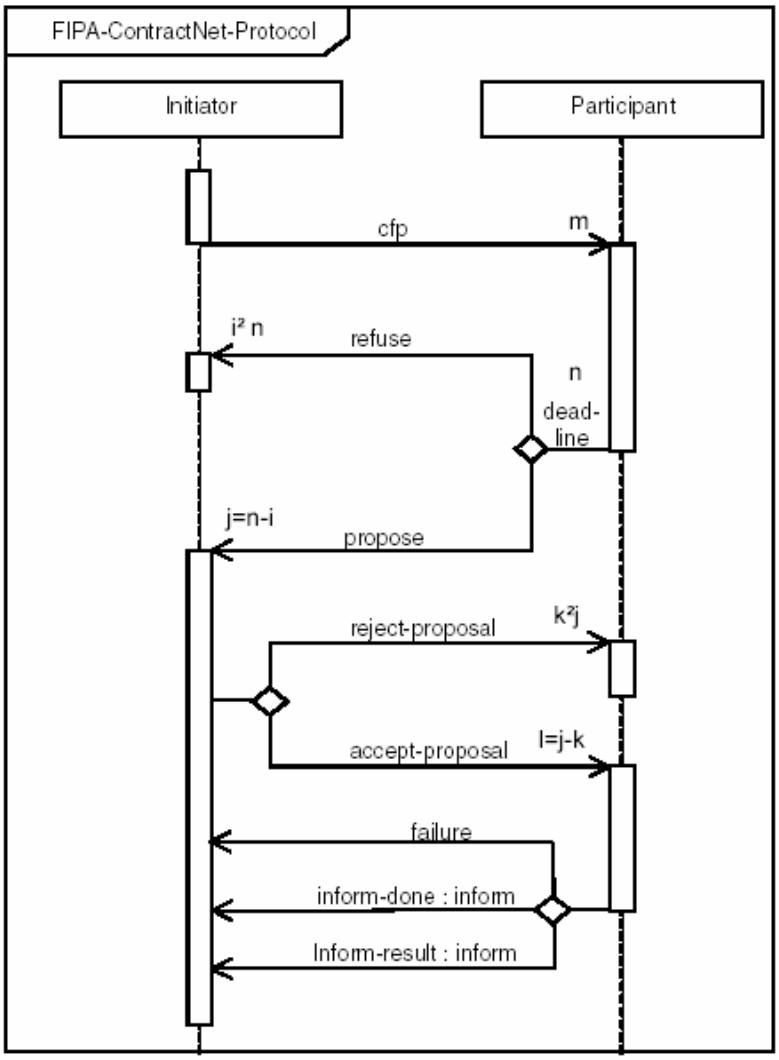
Gli *j participant* che offriranno i loro servizi daranno anche l'informazione circa il costo e/o il tempo di esecuzione del task richiesto dall'*initiator*.

Questi prevedrà un tempo di *deadline*, ossia un limite massimo per attendere sia le proposte che i rifiuti da parte dei contattati.

A questo punto *initiator* deciderà se nessuno, uno o più agenti eseguiranno il task.

A coloro che saranno scelti verrà inviato dall'*initiator* un messaggio di tipo *accept-proposal* per segnalare l'avvenuta accettazione della proposta, viceversa coloro che non verranno scelti, riceveranno un messaggio di tipo *reject-proposal* come rifiuto.

A questo punto gli agenti che avranno ricevuto l'*accept-proposal* eseguiranno il task e risponderanno con i tre casi già noti, ossia fallimento nell'esecuzione del servizio, successo con o senza risultato.


FIPA-ContractNet-Protocol
Initiator
Participant
cfp
m
i² n
refuse
n
dead-
line
j=n-i
propose
reject-proposal
k²j
accept-proposal
l=j-k
failure
inform-done : inform
Inform-result : inform

2.3.7.4 Iterated Contract Net Interaction Protocol (ICNIP)

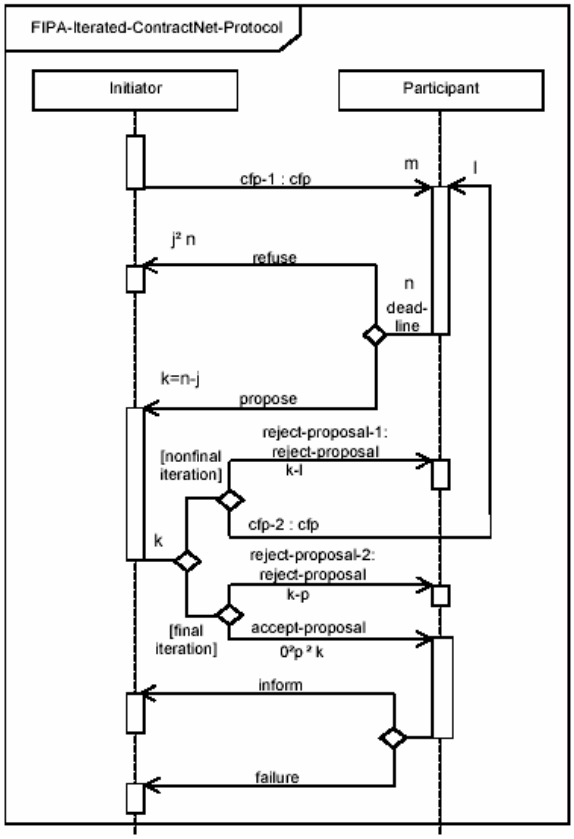
Nel caso dell'Iterated Contract Net Interaction Protocol l'*initiator* effettua delle richieste di servizi con *calls for proposal - cfp* indirizzati a *m participant*, fra questi *n* agenti risponderanno, inviando delle offerte per l'esecuzione del task sotto determinate condizioni, mentre altri rifiuteranno.

Fra le proposte ricevute l'*initiator* deciderà quali accettare.

Deciderà se accettare un sottoinsieme delle proposte e rifiutare le altre, oppure potrà decidere se reiterare il processo, chiedendo nuovamente a un numero via via più ristretto di *participant*.

Il motivo e' legato al fatto che l'*initiator* reimposterà la richiesta con altre condizioni più limitate e questo fino a che non riterrà di potere escludere tutte le proposte o accettare almeno una di queste.

In ogni caso vale sempre il fatto che alla fine, dopo l'esecuzione del task, si possa avere fallimento nella esecuzione del servizio offerto o successo con o senza risultato.



2.3.7.5 Brokering Interaction Protocol (BIP)

Il Brokering Interaction Protocol (BIP) e' progettato per sistemi intermedi multiagente.

Un broker e' un agente che fa da intermediario, usa la sua conoscenza circa gli altri agenti, come un *finder*, ossia riceve richieste di ricerca di altri agenti da parte di uno di essi e li trova, segnalandoli a chi li cercava.

Il Brokering Interaction Protocol (BIP) e' quindi un protocollo macro di più alto livello.

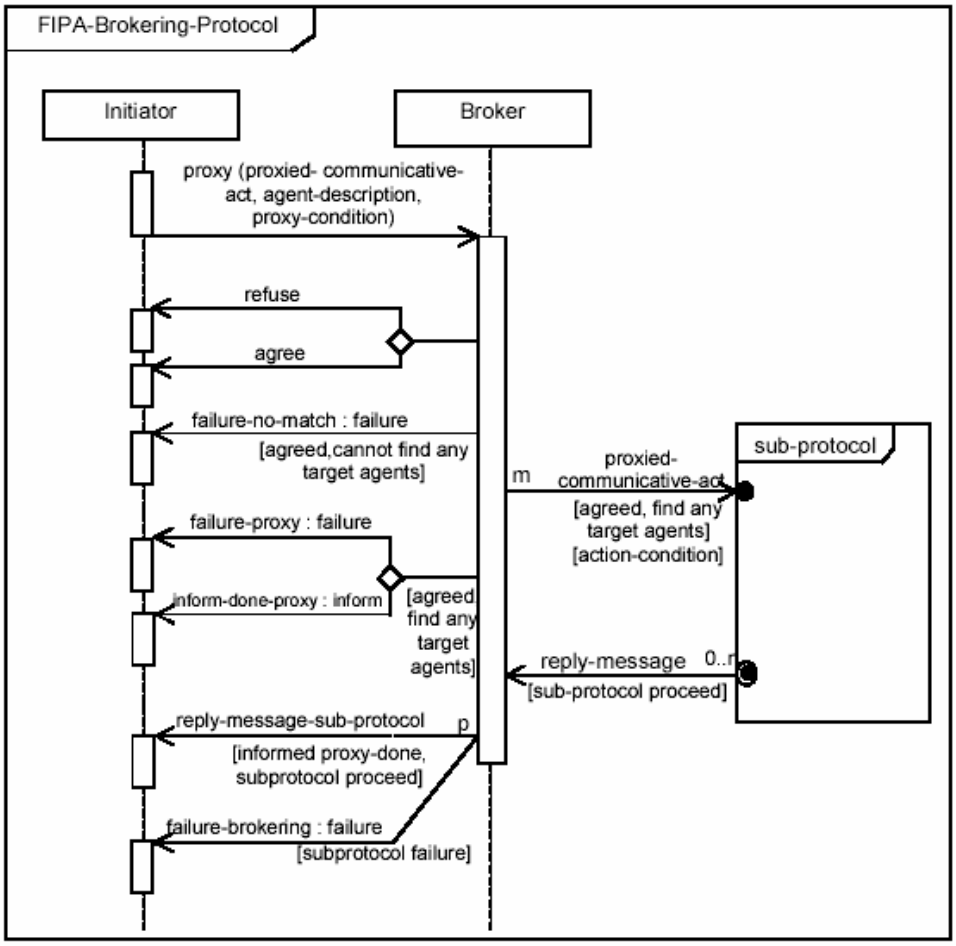
L'*initiator* della interazione di tipo brokering inizia inviando un messaggio proxy, che contiene una espressione referenziale che fissa l'obiettivo, ossia il target di agenti da individuare e verso cui il broker deve inviare il messaggio, invia poi il messaggio di richiesta e le condizioni da soddisfare, per esempio il numero massimo di agenti cui sottoporre il messaggio.

Il suo *participant* e' quindi un broker che può rifiutare la richiesta o accettare con un messaggio di ritorno.

Se accetta la richiesta cerca di individuare gli agenti cui mandare il messaggio, se non ne trova ritorna all'*initiator* un messaggio di tipo *failure-no-match* e l'interazione finisce.

Se il b*roker* trova gli agenti che soddisfano le richieste dell'*initiator* li filtra secondo le condizioni imposte e gli invia il task da eseguire con un subprotocollo.

Alla fine restituisce all'*initiator* il responso circa l'esecuzione del task da parte degli agenti obiettivo.



2.3.7.6 Recruiting IP

Il Recruiting IP e' un protocollo che garantisce una intermediazione fra agenti come nel caso del brokering.

A differenza di questo però, le risposte che tornano indietro non passano di nuovo al broker bensì vanno direttamente a chi le ha richieste, quindi l'*initiator*.

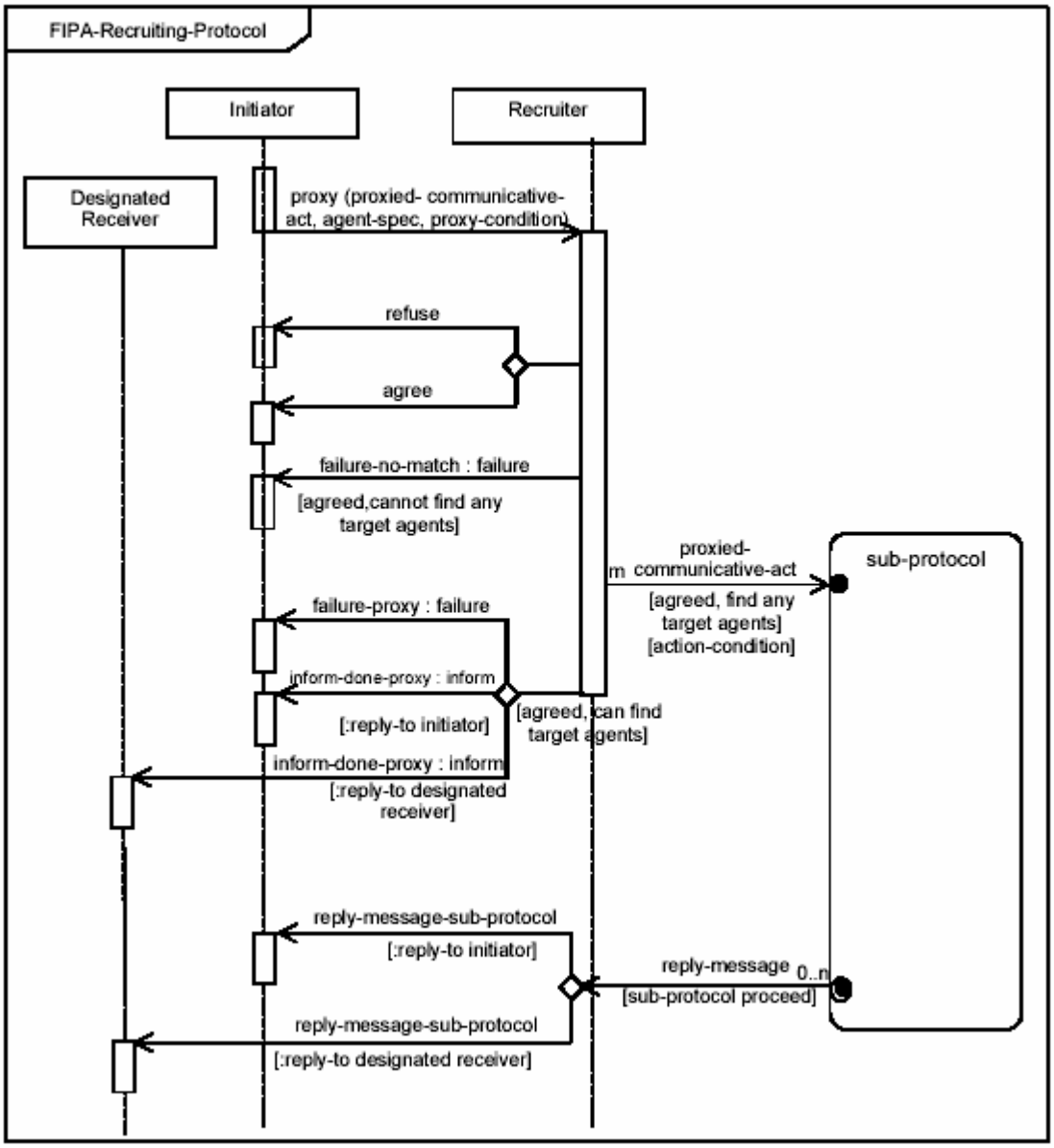
Un *initiator* invia un messaggio proxy ad un *recruiter* con una lista di filtri da usare per individuare gli agenti obiettivo, invia inoltre il messaggio contenente l'azione da eseguire.

Il *recruiter* decide se accettare o meno la richiesta e se accetta, decide chi ricercare in base ai criteri di selezione imposti.

Nel caso in cui non dovesse trovare nessun agente a disposizione, restituisce un messaggio di tipo *failure-no-match* e l'interazione con l'*initiator* finisce.

Nel caso in cui dovesse trovare uno o più agenti obiettivo, comunica a questi l'identificativo della interazione stabilita con l'*initiator* e specifica nel parametro *reply-to* del messaggio, l'indirizzo dell'*initiator,* affinché gli agenti obiettivo possano comunicare direttamente con questo.

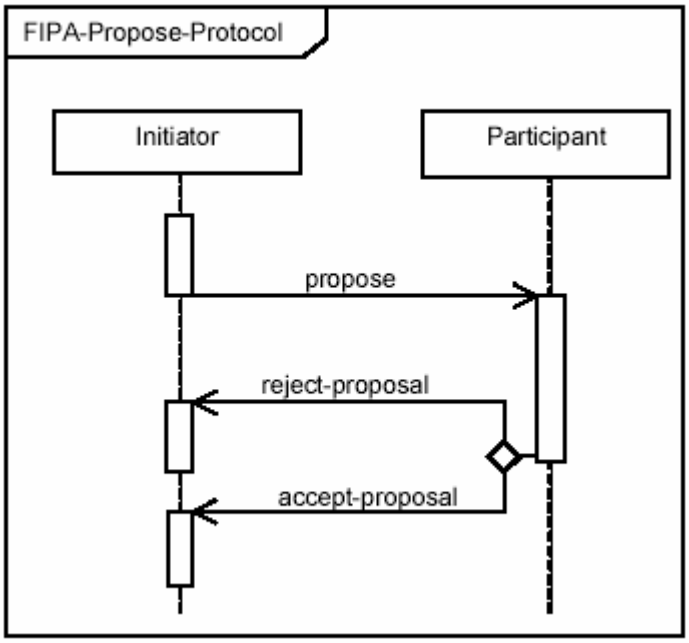
Ogni singolo agente alla fine della esecuzione del servizio potrà rispondere con un messaggio di tipo *inform-done*, oppure di tipo *inform-done-result* o con un *failure*, direttamente all'*initiator* o al ricevente designato, specificando quindi se si e' verificato il fallimento dell'azione eseguita oppure il successo con o senza risultato.


FIPA-Recruiting-Protocol
Initiator
Recruiter
Designated Receiver
proxy (proxied- communicative-act, agent-spec, proxy-condition)
refuse
agree
failure-no-match : failure
[agreed,cannot find any target agents]
proxied-communicative-act
[agreed, find any target agents]
[action-condition]
sub-protocol
failure-proxy : failure
inform-done-proxy : inform
[:reply-to initiator]
[agreed, can find target agents]
inform-done-proxy : inform
[:reply-to designated receiver]
reply-message-sub-protocol
[:reply-to initiator]
reply-message 0..n
[sub-protocol proceed]
reply-message-sub-protocol
[:reply-to designated receiver]

2.3.7.7 Propose Interaction Protocol (PIP)

Il Propose Interaction Protocol (PIP) permette ad un agente di comunicare quello che ha intenzione di fare, specificandolo nel messaggio di proposta e il *participant* può accettare o rifiutare.

Se il *participant* accetta, l'*initiator* esegue le attività proposte e da' il responso alla fine, se il *participant* rifiuta, l'initiator non eseguirà l'attività proposta.



2.3.7.8 Subscribe Interaction Protocol (SIP)

Il Subscribe Interaction Protocol (SIP) permette ad un *initiator* di richiedere ad un *participant* di eseguire un'azione con una sottoscrizione e quando gli oggetti referenziati in essa cambiano stato.

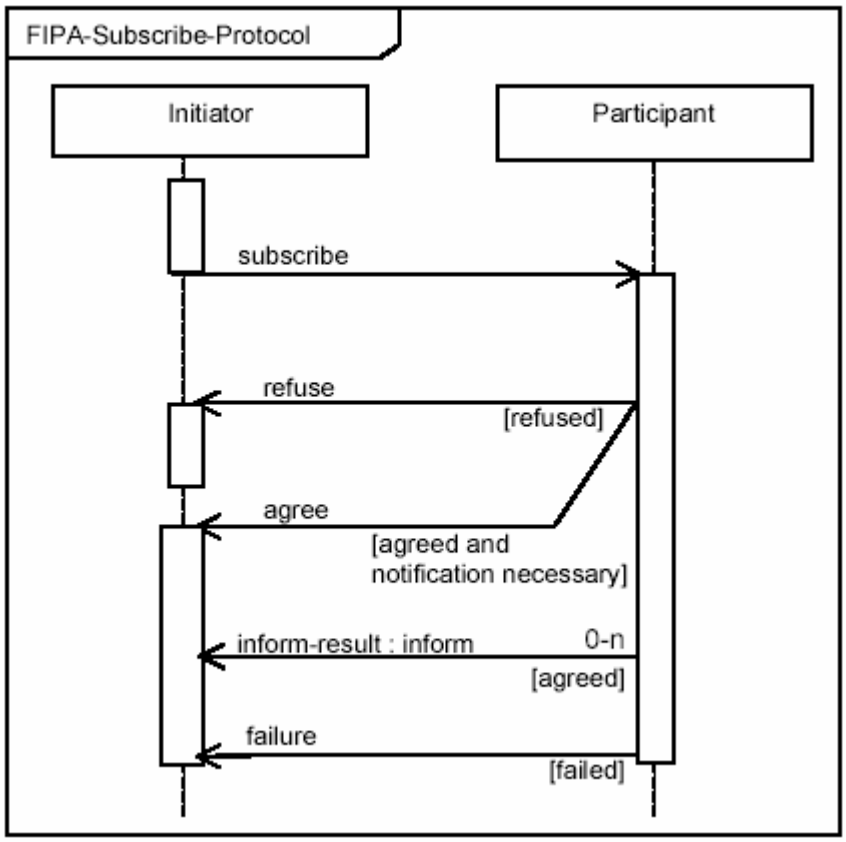
L'*initiator* comincia l'interazione con un messaggio di tipo *subscribe* contenente il riferimento agli oggetti a cui è interessato.

Il *participant* processa il messaggio e decide se accettare o no.

Se non accetta pone a *true* il valore del parametro di ritorno *refused*, altrimenti pone a *true* il valore *agreed*.

In caso di esecuzione con successo il *participant* invia un messaggio di tipo *inform-result* includendo in esso il riferimento agli oggetti sottoscritti, tutto ciò avviene anche in corrispondenza al caso in cui tali oggetti cambiano stato.

In tale protocollo è fondamentale che il servizio di trasporto dei messaggi tenga conto della sequenza temporale dei messaggi di ritorno.



3. Agent Oriented Software Engineering

Negli anni, nello sviluppo del software, si è avuta una crescita costante delle dimensioni dei progetti nonché della complessità dei sistemi da realizzare, che abbracciano ormai un po' tutte le discipline. Nonostante la gestione sistematica delle informazioni abbia assunto un valore strategico per le moderne industrie, la definizione di modelli e metodologie universalmente adottate per lo sviluppo e il mantenimento dei sistemi software (SS) ha tardato molto ad apparire sulla scena.

Contrariamente a quanto avvenuto in altri rami dell'ingegneria, (civile, elettronica, meccanica, ecc.) ed anche in altre discipline non prettamente ingegneristiche, nella realizzazione del software per molto tempo (e tuttora in maniera minore o al più in settori ancora nuovi, come ad esempio nella programmazione ad agenti) gli sviluppatori e gli esperti delle metodologie hanno promosso e spinto i propri metodi, concetti, convinzioni, linguaggi di modellazione e procedure operative. Il risultato di tutto ciò è stata la proliferazione di centinaia di metodologie, apparentemente differenti, e fin troppo spesso identiche negli obbiettivi che si prefiggevano di raggiungere. Analogamente, sono apparsi nella stessa misura tutto un insieme di applicazioni in grado di assistere in modo automatico questi processi. Tra queste emergono i **CASE** [1] tool (Computer Aided Software Engineering), l'insieme degli ambienti integrati di programmazione e progettazione e i linguaggi 4GL (4th generation languages: linguaggi di quarta generazione).

La causa di tutto ciò va ricercata nel tentativo, da parte delle aziende e degli enti, di migliorare i propri processi produttivi e riuscire ad applicare una guida sistematica da seguire nello sviluppo dei propri applicativi, nonché di realizzare un insieme di regole ben condivise entro la stessa organizzazione, atte ad accrescere la propria produttività [2]. La conoscenza maturata durante le attività viene spesso incorporata negli stessi sistemi sviluppati, che in questo modo appaiono come delle linee guida predefinite e sistematiche, per capire e portare a termine altri progetti e nuovi sistemi.

L'adozione di una qualche metodologia, per la realizzazione dei sistemi, si è però rilevata anch'essa presto insufficiente al crescere dei sistemi da realizzare, i quali diventano sempre più difficili da progettare, costruire e, soprattutto, mantenere nel tempo. Questo problema è stato poi amplificato dalle continue innovazioni tecnologiche disponibili [8], come l'avvento della

programmazione ad oggetti, le architetture client/server o peer to peer, internet, l'e-commerce, ecc.; tutte innovazioni che hanno dovuto trovare presto un'adozione nei sistemi concepiti. Tutto ciò ha contribuito a sviluppare sistemi che non rispondessero ai requisiti richiesti, con una grave insoddisfazione da parte dei clienti, indipendentemente dallo specifico ramo applicativo: economico, tecnico o strategico; insoddisfazione che si è tradotta in una grave crisi del settore informatico e meglio nota come: *la crisi del software* [9].

È iniziato così uno studio sistematico delle cause che hanno determinato una simile situazione, identificando come la principale del fallimento nello sviluppo dei sistemi l'applicazione irrazionale degli approcci utilizzati. Pertanto, la comunità scientifica ben presto capì la necessità di dover unire i propri sforzi per gettare basi più solide per lo sviluppo scientifico e razionale dei sistemi, ma prima ancora delle stesse metodologie [10].

In generale, gli approcci metodologici sono in grado di indirizzare con maggiore successo, lo sviluppo dei software verso soluzioni accettabili, nonché a semplificare lo stesso processo di sviluppo. L'obiettivo dello studio dei metodi è quello di raccogliere insieme le esperienze maturate nello sviluppo del software e riuscire a trarne delle regole generali che vadano al di là del singolo contesto applicativo. Questo tipo di conoscenza può essere ottenuta attraverso la partecipazione diretta allo sviluppo, attraverso la valutazione critica dei metodi adottati e, infine, conducendo studi sull'uso dei metodi stessi. È in questo contesto che le compagnie e le organizzazioni hanno iniziato una nuova sfida per ideare strategie innovative per lo sviluppo software, unitamente alla ricerca di nuove applicazioni di supporto e tecniche di organizzazione delle attività.

3.1.1 Metodologie ad agenti

Un campo di crescente interesse nel panorama dello sviluppo del software e rappresentato dalla programmazione ad agenti. Il particolare campo dell'ingegneria che si occupa e studia le problematiche legate allo sviluppo di questo tipo di sistemi è chiamata AOSE [11](Agent Oriented Software Engineering) o, per distinguerla meglio dalla concorrente e ormai consolidata OOSE (Object Oriented Software Engineering), ABSE [12](Agent Based Software Engineering). La novità che questo tipo di tecnica introduce è il concetto di agente. Gli agenti possono essere visti come l'evoluzione degli oggetti [13]. Per essi valgono un po' tutte le considerazioni della programmazione ad oggetti con la fondamentale differenza che per gli agenti possono essere definiti ulteriori tre concetti: autonomia, interattività e proattività.

Nella programmazione ad oggetti, attraverso l'incapsulamento, si realizza *l'information hiding* e si rendono pubblici solamente quei metodi che descrivono l'interfaccia verso l'esterno. Nella programmazione ad agenti le cose sono diverse: un agente non ha metodi pubblici e non può invocare i metodi di un altro agente. Esso può solamente richiedere un servizio ad un altro agente, il quale, interpretando la richiesta, decide o meno di esaudirla.

L'interattività implica l'abilità di comunicazione con l'ambiente e con gli altri agenti. Negli oggetti la comunicazione avviene mediante l'invocazione dei metodi e quindi si deve conoscere, oltre all'oggetto destinatario del "messaggio", anche l'esatta *signature* dei suoi metodi e quindi: nome del metodo, numero e tipo dei parametri, tipo di ritorno. Molto più ricca e complessa è l'interazione tra agenti che avviene utilizzando un vero e proprio protocollo di comunicazione. Gli agenti si scambiano messaggi utilizzando un linguaggio unico e comprensibile a tutti e utilizzano concetti esprimibili in diverse forme: testo, ontologie [20], ecc.

Gli oggetti rappresentano all'interno di un sistema delle entità passive aventi uno stato interno e in grado di operare se stimolati dall'esterno attraverso l'invocazione dei propri metodi. Gli agenti di contro, sono entità attive in grado di modificare l'ambiente in cui operano prendendo delle decisioni autonome legate a fattori contingenti.

L'interesse verso questo tipo di tecnologia nasce dalle considerazioni fatte sui sistemi altamente dinamici (come il campo della robotica, intelligenza artificiale, internet, ecc.), nei quali è possibile prevedere una qualche distribuzione dei compiti, autonomia delle sottoparti in gioco e la collaborazione per il raggiungimento degli obiettivi.

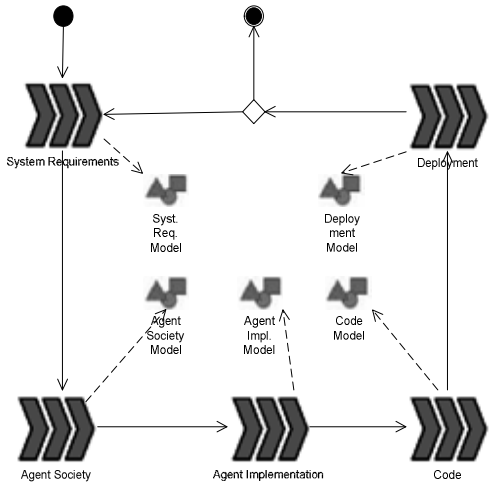
Un'organizzazione del sistema ad agenti fornisce il paradigma entro il quale trattare le interazioni tra gli agenti, attraverso la definizione di ruoli, comportamenti attesi e gerarchie. L'organizzazione di un sistema multi-agente, in generale, viene definita dalle configurazioni dei flussi di informazione tra gli agenti, dalle relazioni stabilite tra questi e dalla distribuzione delle capacità risolutive. Un'organizzazione consiste di un gruppo di agenti, di un insieme di attività da essi perpetrate, di un insieme di connessioni di comunicazione e scambio tra loro, e di un insieme di scopi o di criteri attraverso i quali valutare le attività combinate degli agenti stessi. La struttura organizzativa, dunque, stabilisce i parametri secondo i quali gli agenti comunicano e si coordinano tra loro.

Nell'ingegneria, la nascita di nuovi strumenti è sempre accompagnata dall'introduzione di nuove metodologie che si prefiggono lo scopo di organizzarne e ottimizzarne l'utilizzo nei

diversi contesti di interesse. Due tecniche, e quindi, due approcci diversi si sono maggiormente affermate nella programmazione ad agenti: le tecniche basate su approcci di intelligenza artificiale e tecniche orientate allo sviluppo di sistemi MAS [14] (Multi-Agent System). Quest'ultima tecnica è di fatto quella maggiormente utilizzata perché in grado di fornire soluzione migliori alle problematiche legate alla programmazione distribuita, in quanto considera: l'aspetto sociale delle entità in gioco, la suddivisione dei compiti e l'organizzazione delle attività.

3.1.1.1 La metodologia PASSI

PASSI [15] (Process for Agent Societies Specification and Implementation) è una metodologia iterattiva ed incrementale specificatamente concepita per il progetto di sistemi multi-agente (MAS). Le caratteristiche chiave di PASSI sono l'uso di UML come strumento di modellazione e l'impiego di CASE tool. Questa metodologia prevede una esplicita rappresentazione dei più importanti aspetti di questo tipo di sistemi e il processo di sviluppo si articola in cinque fasi o modelli, a loro volta composte da attività.

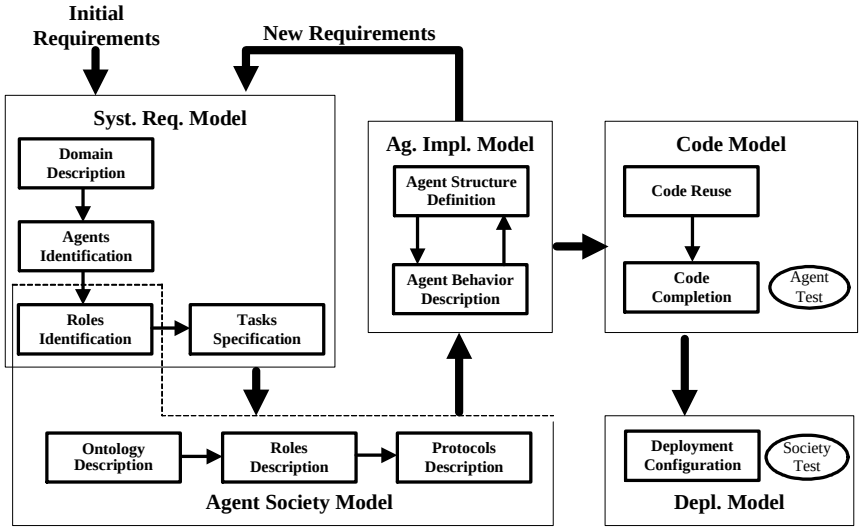


Le fasi di PASSI. **Figura 3-1**

Le fasi coinvolte nel processo di sviluppo sono state specificatamente concepite per affrontare efficacemente tutti gli aspetti significativi dello sviluppo di sistemi complessi, adattandoli ed estendendoli opportunamente per adattarli alle tecniche dello sviluppo ad agenti.

Ogni fase è stata concepita per occuparsi specificatamente di un aspetto del problema dello sviluppo del sistema, raggruppando insieme attività che sono tutte situate ad uno stesso livello logico di operazioni. Le fasi coinvolte sono: la *System Requirements*, la *Agent Society*,

Agent Implementation, la *Code* e la *Deployment*, e ognuna di queste produce un modello che ne rappresenta l'attività.



I modelli di PASSI. **Figura 3-2**

La System Requirement Model è il risultato della prima attività e si concentra sull'analisi dei requisiti e definizione dei compiti affidati a ciascun agente. Si identificano in essa quattro attività: la Domain Description (DD), l'Agent Identification (AI), Role Description (RD) e la Task Specification (TS). Tramite la DD si fa una descrizione del dominio del problema e si rappresenta una descrizione del sistema dal punto di vista delle funzionalità mediante diagramma dei casi d'uso. La AI si concentra sulla individuazione degli agenti coinvolti nel sistema e l'assegnazione ad essi delle funzionalità. Con la RD si specificano i ruoli di ogni singolo agente coinvolto nel sistema e si procede all'identificazione delle responsabilità di ognuno di essi mediante la rappresentazione degli scenari con i diagrammi di sequenza. Infine la TS è utilizzata per ottenere una specifica dei compiti di ogni singolo agente rappresentandoli attraverso i diagrammi delle attività.

L'aspetto sociale e comunicativo è affrontato con l'Agent Society Model attraverso la definizione delle attività: Ontology Description (OD), Roles Description (RD), Protocols Description (PD) e nuovamente la RI nei termini dell'uso dei ruoli già identificati. L'ontologia impiegata dagli agenti è descritta attraverso l'uso di diagrammi di classi e si compone di due sotto-attività: la Domain Ontology Description (DOD) che si occupa di descrivere i concetti, le azioni e i predicati utilizzati tra gli agenti, e della Communication Ontology Description (COD) che specifica la comunicazione che avviene tra di essi. La RD consente di specificare meglio il

ruolo degli agenti. Viene utilizzato un diagramma delle classi in cui i ruoli degli agenti sono rappresentati dalle classi ed entro essi si raccolgono le singole attività di ogni agente. Con la PD, invece, si realizzano i diversi protocolli di comunicazione che intervengono tra gli agenti.

L'Agent Implementation Model affronta l'aspetto implementativo degli agenti realizzando tutta una serie di diagrammi che servono a specificare meglio la struttura degli agenti coinvolti e del dettaglio delle operazioni svolte da ognuno di essi. La prima attività è la Agent Structure Definition (ASD) che descrive meglio la struttura dell'agente attraverso l'uso di un diagramma delle classi e specificando opportunamente i metodi e gli attributi necessari. Quest'attività è realizzata impiegando due diagrammi delle classi: Single Agent Structure Definition (SASD) che si concentra sul singolo agente, unitamente ai suoi task, e la Multi Agent Structure Definition (MASD) che impiega i diagrammi delle classi per mostrare una visione di insieme delle classi degli agenti coinvolti. La seconda fase, denominata Agent Behaviour Description, consente di specificare nel dettaglio tutte le operazioni necessarie all'agente per l'espletamento dei suoi compiti e si avvale dei diagrammi: Single Agent Behaviour Description (SABD) che rappresenta, mediante l'utilizzo di un diagramma delle attività, il comportamento nel dettaglio del singolo agente e il Multi Agent Behaviour Description (MABD) per descrivere e meglio rappresentare l'evoluzione delle operazioni entro l'agente stesso quando é posto in relazione con tutti gli altri.

Il Code Model serve ad ottenere direttamente il codice del sistema a partire dalle descrizioni della fase precedente. Si compone della Code Reuse Library (CRL) che rappresenta una libreria di classi di *patterns* per lo sviluppo di sistemi ad agenti, e della Code Completition Baseline (CCB) che consentono di scrivere il codice sorgente del sistema stesso.

L'ultima delle fasi è rappresentata dal Deployment Model (DM) che è composta dalla sola attività Deployment Configuartion (DC) utilizzata per descrivere la dislocazione degli agenti entro l'intero sistema.

3.1.1.2 La metodologia Gaia

Gaia [16] è una metodologia che supporta due livelli di sviluppo di un sistema multi-agente: a basso livello si occupa della struttura dell'agente, mentre ad un alto livello si occupa della società di agenti e delle relazioni tra essi. La sua realizzazione è dovuta alle considerazioni fatte sulle metodologie esistenti, le quali falliscono nel rappresentare la natura autonoma e proattiva degli agenti. Mediante Gaia gli sviluppatori di software possono sistematicamente progettare una vista implementativa del progetto partendo dall'analisi dei

requisiti del sistema. Il primo passo di Gaia è quello di trovare i ruoli all'interno del sistema e le interazioni tra essi. Un ruolo viene specificato identificando quattro attributi: responsabilità, permessi, attività e protocolli che rappresentano rispettivamente la responsabilità dell'agente in quel ruolo entro il sistema, le attività che espleta entro il sistema e le informazioni a cui ha accesso, le attività interne che esegue per lo specifico ruolo e il tipo di comunicazione che instaura con gli altri agenti.

Gaia possiede degli operatori formali e dei *template* per rappresentare i ruoli e i loro attributi. Ha anche degli schemi che possono essere usati per rappresentare le interazioni. Ed infine, ingloba i ruoli identificati all'interno degli agenti per creare gli agenti fisici che intervengono nel sistema. Per far ciò si determina un Service Model necessario per completare un ruolo di un agente, e si crea un Acquaintance Model per la rappresentazione delle comunicazioni tra gli agenti.

3.1.2 I CASE tool

Il CASE (Computer Aided Software Engineering) tool è una "qualsiasi" applicazione che può essere eseguita su un calcolatore ed essere utilizzata come supporto durante il processo di sviluppo di un software. In questa definizione rientrano tutte quelle attività di: pianificazione, sviluppo ed evoluzione dei software, nonché il supporto agli aspetti manageriali, amministrativi e tecnici di un progetto. L'obbiettivo principale è quello di rinforzare e supportare l'approccio ingegneristico nello sviluppo del software in modo da consentire una più affidabile evoluzione di tutte le fasi coinvolte, fornendo una assistenza di tipo automatizzata e guidata da calcolatore; assistenza che si traduce direttamente in risultati più affidabili perché si riduce il rischio di commettere errori (soprattutto quelli banali) e un incremento della produttività.

I CASE tool possono essere classificati in tanti modi a seconda delle funzionalità che offrono, del tipo di supporto che forniscono al ciclo di sviluppo del software e il grado di integrazione che mettono a disposizione con altri applicativi e altri CASE tool [17].

I primi CASE tool supportavano una singola fase dello sviluppo del software come, ad esempio, lo sviluppo di software ad oggetti secondo una certo metodo (Coad and Yourdan [18], Booch, ecc.). Il limite di questi prodotti, com'è facilmente intuibile, sta proprio nell'assunzione che una azienda adotti esclusivamente un tipo di approccio al problema, il quale spesso si rivela essere inadeguato rispetto alla totalità dei casi che possono presentarsi. Tra i CASE tool oggi

più diffusi, e che offrono un adeguato supporto ad una metodologia, possiamo elencare a titolo di esempio: Rational Rose, Eclipse, ArgoUML, Toghether J ed altri.

Altri tipi di CASE tool consentono invece di utilizzare contemporaneamente più metodologie per rispondere meglio alle esigenze dell'utenza. Il supporto di questo tipo di software è certamente più efficace e si rivela utile in molti campi. Tuttavia ci si scontra con delle difficoltà che sorgono nel momento stesso in cui bisogna riuscire ad integrare e mantenere insieme i progetti che in essi si sviluppano.

La novità più rilevante nel campo dello sviluppo assistito, è stata introdotta con l'avvento dei primi **MetaCASE tool**. Questo tipo di applicazione è in grado di fornire un processo automatico o semi-automatico per la creazione dei CASE tool. I MetaCASE tool (come si vedrà meglio in seguito) sono basati su un metamodello sottostante che viene utilizzato con lo scopo di descrivere il linguaggio, i concetti e le relazioni che si impiegano in una particolare metodologia.

3.1.3 Limiti degli strumenti per la Software Engineering

L'avvento dei CASE tool è stata annunciata come uno degli avvenimenti che avrebbe risolto gran parte dei problemi legati alla produttività e alla realizzazione di soluzioni più facilmente mantenibili nel tempo, con meno spreco di tempo e di risorse e con dei *time to market* più rapidi. Chiaramente queste promesse non sono state interamente mantenute e addirittura non tutti hanno trovato conveniente adottare questo tipo di strumenti (e non soltanto per fattori economici). Delle ricerche fatte in questa direzione hanno rilevato che anche le aziende, che hanno adottato un qualche strumento di questo tipo, hanno finito per usarlo poco o addirittura non l'hanno usato affatto [19]. Le limitazioni maggiori che provengono dall'uso dei CASE tool riguardano tutta una serie di problemi di cui ancora molti non del tutto risolti.

L'enorme differenza di qualità tra CASE tool disponibili e la cospicua varietà di metodologie implementate, associati spesso ai costi non trascurabili dei software, è uno dei fattori materiali più consistenti che hanno impedito l'ingresso dei CASE tool nelle attività produttive. Molte organizzazioni, poi, hanno dovuto fare i conti con una obsolescenza piuttosto accelerata di molte di queste applicazioni; obsolescenza dovuta a fattori di inadattabilità dei metodi e dei modelli ai cambiamenti che si sono susseguiti nel tempo e legati a cambiamenti teorici o pratici delle metodologie stesse.

Un altro aspetto riguarda l'ampio raggio del dominio dei progetti che possono essere realizzati nel campo dello sviluppo del software e le conseguenti difficoltà che si incontrano nel trovare adeguate soluzioni attraverso l'utilizzo di CASE tool preconfezionati. Inoltre, si riscontrano difficoltà intrinseche nel riusare le conoscenze già maturate nei progetti passati.

Le limitazioni dei CASE tool possono essere analizzate con maggiore dettaglio da due prospettive differenti: dal punto di vista *organizzativo* delle aziende che adottano uno strumento CASE e dal punto di vista *tecnico*, nei termini delle limitazioni che gli stessi CASE tool presentano.

3.1.3.1 Limiti organizzativi

Questo tipo di limitazioni riguardano gli effetti che l'adozione di un CASE tool può avere su una organizzazione aziendale e di come l'introduzione di un software, da adottare universalmente entro le attività produttive aziendali, possa influenzare l'andamento stesso delle attività che in essa si svolgono.

Intanto, avremo l'impatto economico (come già accennato) che, per certe organizzazioni e per le realtà con un certo numero di dipendenti, può rilevarsi un fattore da non trascurare, soprattutto se messo in relazione ai costi legati al *training* del personale. Inoltre, il *payback* di questi prodotti può essere molto alto, considerando che gli effetti della loro introduzione nel ciclo produttivo si manifestano soprattutto nel medio e lungo periodo, attraverso una accelerazione dei tempi di presentazione dei prodotti (medio periodo), di una migliore qualità dei software rilasciati (medio periodo) e dei minori costi legati alla manutenzione dei programmi sviluppati (lungo periodo).

Un altro aspetto riguarda i problemi organizzativi che l'adozione di un CASE tool comporta, come il lavoro di squadra e la suddivisione dei compiti. Anche il fattore apprendimento gioca un ruolo fondamentale, considerato che deve essere uno strumento ampiamente condiviso e che, per sua natura, non è sempre facile da imparare. Questo ci porta a considerare anche il forte impatto che può suscitare l'adozione di uno strumento che richiede una rivisitazione delle personali tecniche di sviluppo. Questi strumenti infatti forzano ed enfatizzando soprattutto la parte pre-implementativa e di analisi, esortano la produzione di grafici e di modelli, e rilegano solo alla fine le attività di sviluppo vero e proprio delle applicazioni. Molte persone percepiscono quindi i CASE tool come elementi dequalificanti e fortemente vincolanti, anziché strumenti in grado di accrescere la loro produttività.

Un ultimo aspetto è quello che riguarda il ciclo produttivo vero e proprio delle aziende. Molte compagnie utilizzano propri processi e proprie metodologie. Il loro modo di operare deve essere cambiato e spesso sovvertito in favore di metodologie "ufficiali" e generalmente riconosciute. L'adattamento dei CASE tool alle proprie esigenze non è di solito una cosa semplice e il più delle volte rappresenta una soluzione assolutamente non praticabile, vista la rigidità realizzativa dei CASE tool. La scelta poi dell'adozione della metodologia a cui conformarsi non è facile, viste le numerose metodologie esistenti, la mancanza di standard industriali precisi, e la relativa immaturità dei prodotti CASE. Infine c'è il problema del patrimonio tecnico e progettuale già esistente di una azienda che deve essere salvaguardato e che i CASE tool non riescono a fare proprio.

3.1.3.2 Limiti funzionali

Questo tipo di limitazioni riguardano le caratteristiche e le funzionalità stesse dei CASE tool e i vincoli che essi impongono alla loro usabilità e al supporto offerto alle metodologie che implementano.

L'usabilità di un CASE tool, dal punto di vista dell'interazione con l'utente, è molto poco curata, obbligando di fatto gli utilizzatori a conformarsi al modo di operare di queste applicazioni. Molti dei metodi utilizzati provengono da implementazioni di tecniche all'origine cartacea con l'aggiunta dei controlli di correttezza, consistenza e coerenza sui modelli sviluppati. Il livello di aiuto messo a disposizione da questi programmi si limita all'analisi superficiale dei diagrammi e non sono presenti dei *feedback* relativi ai contenuti di questi, come la correzione automatica o l'analisi qualitativa dei modelli prodotti.

I vincoli imposti dall'uso di una specifica metodologia, come già visto, costringe le aziende a sottostare a delle regole e ad imporre dei precisi metodi per lo sviluppo dei propri progetti. Queste metodologie non possono essere modificate né adattate alle diverse esigenze che si presentano e il supporto offerto si limita il più delle volte a una serie di editor grafici che implementano graficamente o testualmente lo specifico linguaggio di modellazione.

Non vengono considerati gli aspetti legati al processo di sviluppo delle metodologie, né a quelli che portano alla creazione di un singolo modello; anche il supporto offerto all'intero ciclo di sviluppo del software è relativamente limitato.

In conclusione, non esiste un supporto mirato al riutilizzo dei modelli esistenti e, anzi, vi è una grande difficoltà, anche nel riuscire ad utilizzare le informazioni provenienti da progetti di altri CASE tool che adottano le stesse metodologie o che si rifanno allo stesso tipo di

programmazione (ad esempio quella ad oggetti). Uno sforzo in tal senso è stato fatto cercando di definire uno standard comune per l'interscambio dei dati tra le diverse implementazioni dei CASE tool (solo per quelli conformi a predeterminate metodologie). Tuttavia anche in tal caso, la portabilità sarebbe praticabile tutt'al più tra un insieme ristretto di applicazioni.

4. Pattern

Il concetto di pattern nasce in ambiti completamente diversi da quelli della progettazione di software. In architettura un pattern è un modulo predefinito che può essere utilizzato come entità base di un nuovo progetto.

In maniera più astratta un pattern rappresenta un nucleo di idee che incapsula l'essenza di una soluzione ad un problema ricorrente in un determinato contesto.

Quello introdotto dai pattern non è soltanto un principio astratto ma piuttosto una metodologia: un insieme di elementi e di relazioni che porta ad una soluzione reale e concreta.

In altre parole un pattern non indica solo cosa fare per risolvere un problema ma specifica anche quale sia la strada migliore per farlo; in un pattern ha grande rilievo l'aspetto pratico.

Tale concetto, può essere espresso a più livelli di astrazione:

1. *pattern architetturali* (o concettuali); esprimono una organizzazione strutturale o uno schema per definire l'architettura ad alto livello di un sistema software. Essi forniscono un insieme di sottosistemi insieme alle relazioni e alle responsabilità corrispondenti per raggiungere uno scopo. Questi concetti sono descritti mediante concetti e termini propri del dominio dell'applicazione.
2. *design patterns*; i pattern di questa categoria sono costituiti da un insieme di diagrammi e di regole che descrivono la struttura con la quale i componenti del sistema possano interfacciarsi. Per descrivere questi concetti si utilizzano costrutti propri dell'ingegneria del software orientata agli oggetti: classi, ereditarietà, relazioni di aggregazione, di uso o di appartenenza.
3. *idiomi* (o pattern di codifica); si tratta di pattern di basso livello, ovvero applicabili soltanto ad uno specifico linguaggio di programmazione. Essi descrivono come implementare gli aspetti di un problema usando un set specifico e ben definito di comandi e operazioni offerte dal linguaggio.

Nell'analisi del problema sono stati presi in considerazione i design pattern, i quali sono descritti con maggiore dettaglio di seguito.

4.1 Design patterns

Un design pattern [21] descrive una soluzione semplice ma efficace ad un problema specifico che si incontra nella progettazione e nello sviluppo del software. I design pattern catturano soluzioni che sono state sviluppate e che si sono evolute con il tempo e l'esperienza. Essi costituiscono un elemento essenziale per il riuso e la flessibilità del codice.

I design pattern sono nati per la programmazione ad oggetti sebbene non abbiano una implementazione specifica. Possono essere utilizzati in qualunque linguaggio di programmazione che fornisca concetti come "tipo", "polimorfismo" o "interfaccia".

Progettare un sistema complesso è un compito difficile che richiede una grande esperienza. Ancora più difficile è riuscire ad inserire all'interno del progetto delle componenti che possano essere riutilizzate successivamente. Tuttavia il riuso è un punto chiave della progettazione. I progettisti esperti sanno che il modo migliore per affrontare un problema nuovo non è quello di iniziare da zero. E' molto più efficiente impiegare componenti esistenti che con poche modifiche possano essere assemblate.

Ai design pattern si affida talvolta il compito di tramandare l'esperienza fatta dai progettisti. Utilizzare un pattern significa applicare con successo una soluzione che è stata già provata e collaudata, e che per questo ha buone garanzie di essere corretta. Un design pattern viene descritto attraverso un nome che ne identifica lo scopo, una descrizione testuale del problema e una soluzione che può essere testuale generalmente corredata di diagrammi UML. Inoltre è presente una sezione in cui viene descritta l'applicabilità del pattern a determinate situazioni e i rischi che si possono presentare.

Vantaggi nell'uso dei pattern:

- riusabilità
- correttezza
- robustezza
- diminuzione dei tempi di sviluppo
- aumento della produttività

La prima caratteristica, la riusabilità del codice, è stata già esaminata. La correttezza e la robustezza sono assicurate dal fatto che i pattern sono stati sviluppati e testati prima di essere

diffusi: essi derivano dall'esperienza acquisita. La diminuzione dei tempi di sviluppo e l'aumento della produttività sono un fattore importantissimo sul quale si basa la moderna industria del software.

Una critica che viene mossa nei confronti dei pattern è che un sistema ad oggetti complesso presenta una fitta rete di interazioni e dipendenze tra le componenti. Questo rende difficile l'estrapolazione di pattern realmente riutilizzabili; molti dei pattern che si riescono ad estrarre e a riutilizzare effettivamente sono troppo generici e quindi di scarsa utilità pratica.

Un design pattern frequentemente riportato in molti testi come esempio è l'Observer; esso appartiene alla categoria dei pattern comportamentali e nasce per risolvere un problema di relazioni tra oggetti.

Quando un sistema viene suddiviso in una collezione di classi cooperanti è necessario tenere conto delle relazioni che nascono tra gli elementi presenti. Quello delle relazioni tra gli oggetti è uno dei punti chiave più importanti nella progettazione object-oriented. Il progettista deve infatti controllare che le relazioni tra gli oggetti non formino dei cicli, in quanto queste situazioni diventano estremamente difficili da gestire.

Quando un oggetto è interessato alle informazioni contenute da un altro oggetto diventa inevitabile che tra queste due entità ci sia una relazione. Se a questo si aggiunge che la relazione deve essere navigabile in entrambi i versi nasce il problema del ciclo sopra descritto.

Il pattern Observer è descritto da due diagrammi; nel diagramma delle classi viene mostrata la struttura statica del pattern e le relazioni che devono esistere tra gli oggetti del sistema. Le entità usate per descrivere il pattern sono classi concrete e classi astratte, relazioni di eredità e di uso.

Inoltre dei commenti inseriti come note aiutano a comprendere il significato dei vari elementi.

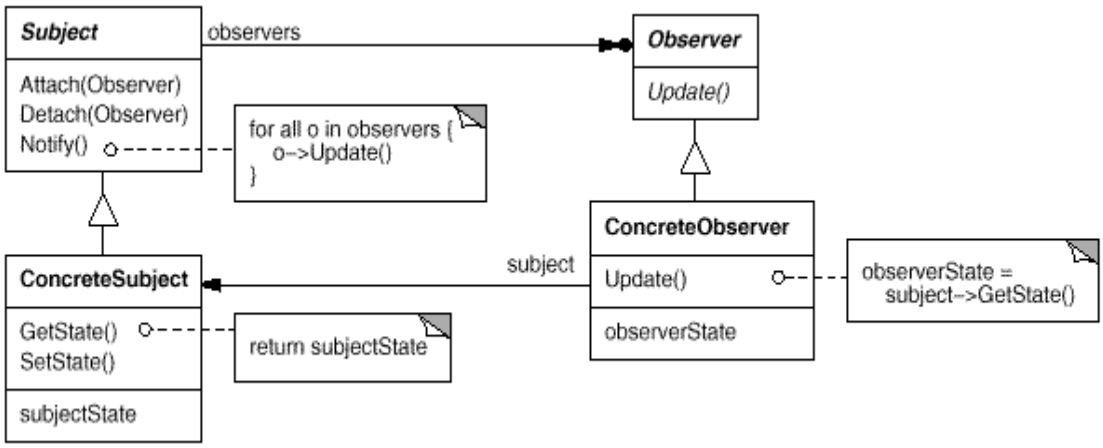


Figura 4-1 Diagramma delle classi che descrive la struttura statica del pattern Observer

Nel diagramma di collaborazione invece viene evidenziata la sequenza di attività che porta alla soluzione del problema. Oltre a questi due diagrammi si può far ricorso ad una descrizione testuale dei partecipanti nella quale si descrive il ruolo svolto da ogni elemento che prende parte nel pattern.

Altri elementi utili per descrivere il pattern sono l'elenco delle situazioni in cui l'uso del pattern può esser utile e le possibili implicazioni dovute all'utilizzo dello stesso.

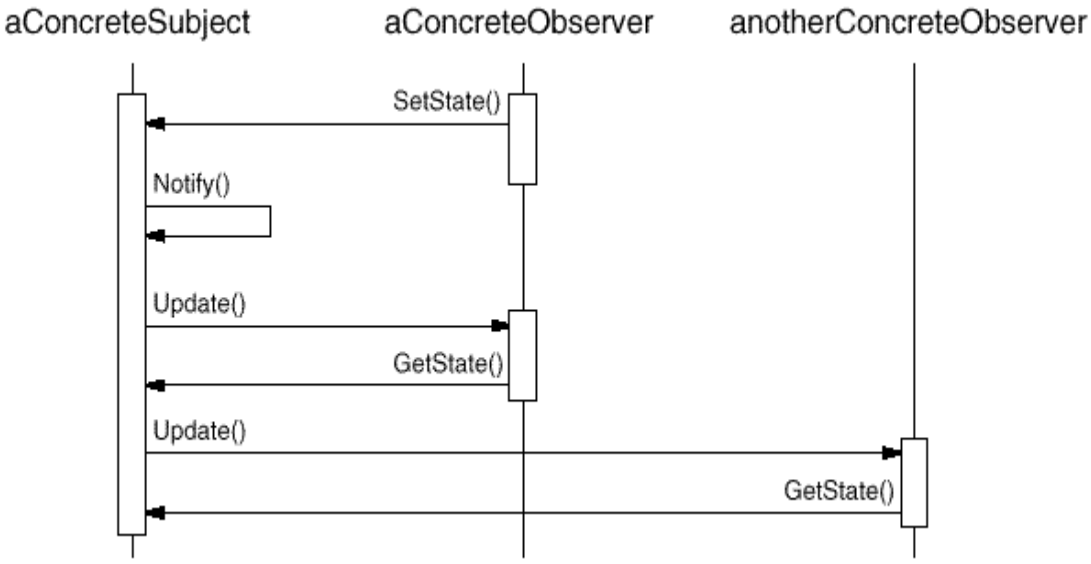


Figura 4-2 Diagramma UML che descrive la struttura dinamica del pattern Observer

4.2 Pattern per agenti

Tutti coloro che si occupano di applicazioni orientate agli agenti sono in qualche modo dei pionieri che inventano e re-inventano qualche cosa. A volte i problemi che necessitano di soluzione sono semplici altre volte sono più complessi. I pattern possono fornire un aiuto catturando le soluzioni a problemi che si incontrano con più frequenza [22][23][24][25].

Un pattern per agenti può fornire garanzie di riuso maggiori rispetto a quelle fornite nell'ambito della programmazione ad oggetti. Il motivo di questa affermazione sta nel fatto che un agente è una entità autonoma, e in quanto tale non possiede dipendenze strette con le altre entità del sistema.

Le uniche dipendenze esistenti tra gli agenti di un sistema sono quelle di comunicazione, e la maggior parte delle piattaforme per agenti le gestisce come dipendenze lasche. Per questo motivo aggiungere o eliminare funzionalità ad un agente ha un effetto locale e non influisce sul corretto funzionamento del resto del sistema.

Queste sono le motivazioni che hanno spinto alla definizione di una struttura per la rappresentazione dei pattern per i sistemi multiagente. Tale struttura verrà proposta nel capitolo seguente "La struttura proposta per i pattern".

4.3 La struttura proposta per i pattern

Il paradigma di programmazione ad agenti si trova attualmente in una fase pionieristica; tuttavia nell'ambito della ricerca sembra esserci grande fiducia sulle caratteristiche innovative che dovrebbero fare emergere questa tecnologia in un prossimo futuro.

Sarà compito dei ricercatori dimostrare come le caratteristiche uniche introdotte dai sistemi multiagenti possano garantire vantaggi notevoli nello sviluppo del software per determinate applicazioni, rispetto alle metodologie tradizionali.

Come tutte le tecniche di programmazione emergenti, molte ricerche attualmente in corso sono mirate ad estendere ed adattare le tecniche di ingegneria del software usate nell'ambito della progettazione orientata agli oggetti.

4.3.1 Uso dei pattern nel progetto

Una delle tecniche di ingegneria del software che sembra trarre nuova vitalità dal paradigma di programmazione ad agenti è quella dei pattern.

Di seguito si illustrerà come sia possibile l'applicazione di pattern nello sviluppo di progetto di un sistema ad agenti usando una metodologia di progettazione quale PASSI.

4.3.1.1 PASSI

PASSI (a Process for Specifying and Implementing Multi-agent Systems Using UML) [15] è una metodologia di sviluppo di sistemi multi-agente che conduce il progettista dall'analisi dei requisiti fino alla codifica mediante cinque modelli e dodici fasi.

Secondo PASSI, un agente è un'unità software che si estende su due livelli, uno astratto e uno concreto. Da un punto di vista astratto un agente possiede ruoli, comportamenti, obiettivi e conoscenze; da un punto di vista concreto un agente è un'istanza di una classe e possiede attributi, metodi e sottoclassi.

Questa dualità risulta essenziale per poter lavorare sugli agenti a diversi livelli di concezione.

4.3.1.1.1 Modello di implementazione degli agenti

In questo modello il progettista definisce l'architettura implementativa del sistema ad agenti. Per far questo si avvale di due fasi: la fase di definizione della struttura dell'agente e la fase di specifica del comportamento degli agenti. Tra i vari diagrammi coinvolti in questo modello quelli che più degli altri si prestano all'applicazione di pattern e alla generazione di codice sono il SASD e il MABD.

4.3.1.1.1.1 Single Agent Structure Description diagram

In questo diagramma si specifica la struttura interna delle classi che compongono un agente. Si producono alcuni diagrammi delle classi; in ogni diagramma si inserisce una classe per l'agente e una classe per ogni suo task.

Nelle classi si specificano attributi e metodi che agente e i suoi task devono possedere.

Questo livello di descrizione dell'agente è di tipo implementativo. Le classi specificate rappresentano le classi necessarie per la realizzazione dell'agente nel linguaggio di codifica desiderato.

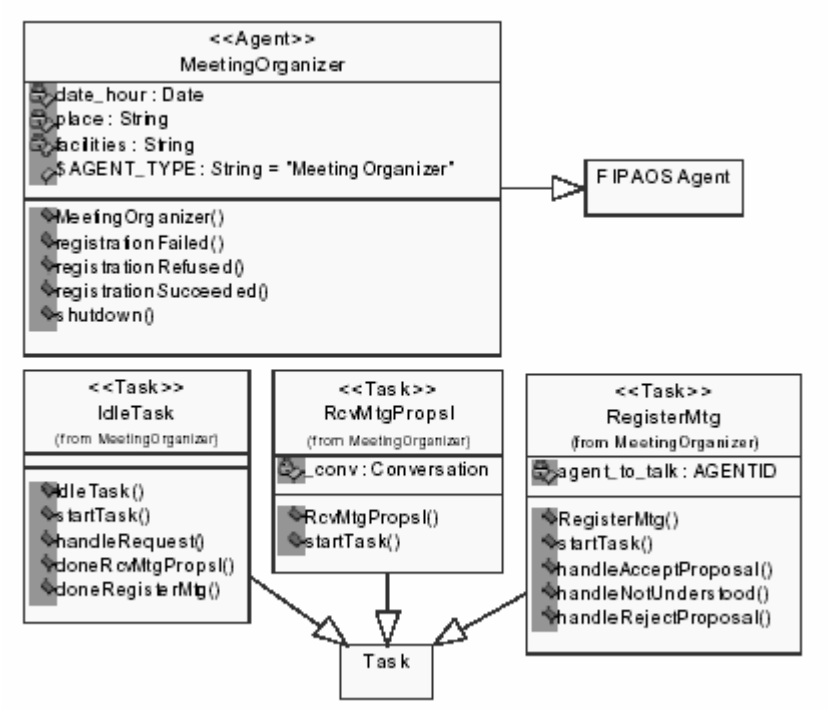


Figura 4-3 Diagramma delle classi UML relativo al Single Agent Structure Description. Un agente o un task sono rappresentati mediante delle classi

4.3.1.1.1.2 Multi Agent Behaviour Description diagram

In questa fase si definiscono le interazioni tra agenti e task di tutto il sistema utilizzando un diagramma delle attività.

Ogni swimlane rappresenta una classe presente nel SASD (agente o task). All'interno delle swimlane si inseriscono le attività, che sono i metodi della classe specificata.

Le relazioni tra le attività rappresentano:

1. invocazioni di metodo, se avvengono tra due attività della stessa swimlane;
2. creazione di nuove istanze, se avvengono tra attività relative a swimlane che appartengono a classi dello stesso agente;
3. comunicazioni, se avvengono tra attività relative a swimlane di agenti differenti.

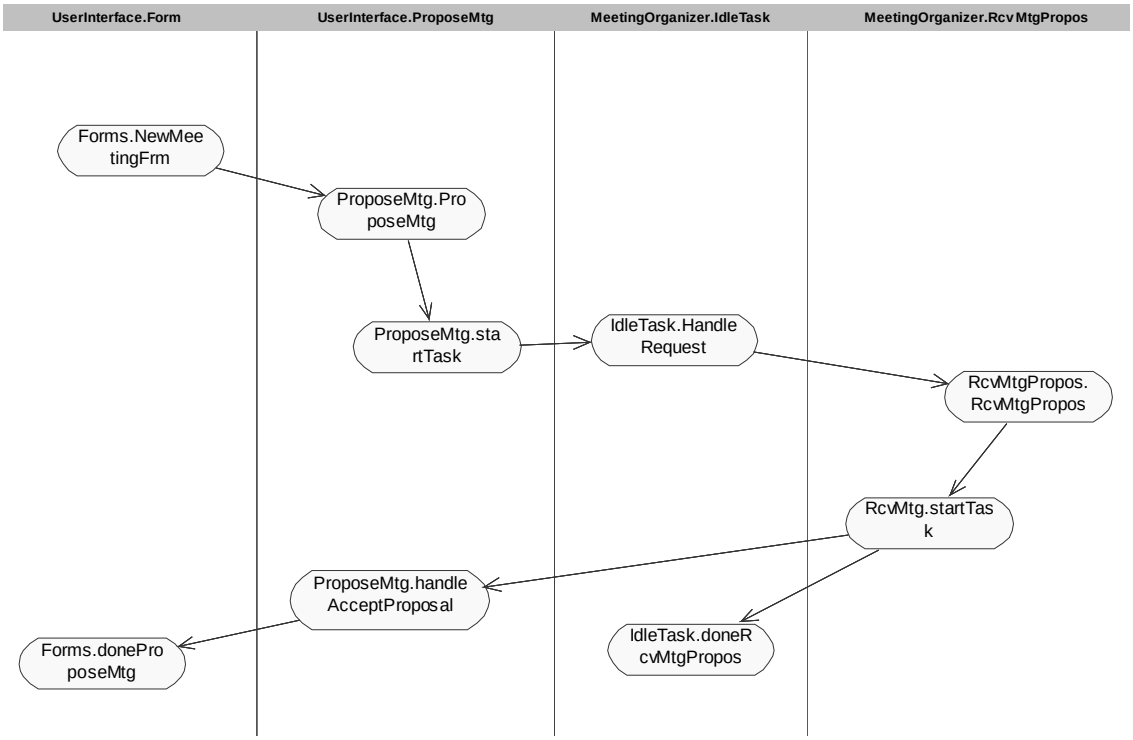


Figura 4-4 Diagramma delle attività UML relativo alla Multi Agent Behaviour Description

4.3.1.1.1.3 Un caso di studio: l'agente dispatcher

In relazione ai due tipi di diagrammi presentati è possibile trovare i scenari e problemi di implementazione che si ripetono con una certa regolarità durante la progettazione.

Questa osservazione ha condotto verso la ricerca di una struttura in grado di catturare le possibili implementazioni dei problemi riscontrati in fase di analisi e progettazione.

Di seguito si riporta un caso di studio relativo ad un tipico scenario che si presenta durante la sincronizzazione delle attività di vari agenti. Questo esempio permette di spiegare in che modo sia possibile definire un pattern per agenti.

La situazione che si vuole analizzare è quella relativa ad un particolare modello di comunicazione usato spesso nell'ambito della robotica: la comunicazione tramite una lavagna condivisa. Questa tecnica di comunicazione è supportata in modo standard su alcune

piattaforme ad agenti, come ad esempio Ethnos, mentre deve essere ricreata in altre piattaforme come FIPA-OS.

La lavagna condivisa è una tecnica usata per migliorare l'efficienza della comunicazione quando sono coinvolti più di due agenti.

Si supponga che vi siano un certo numero di robot che devono coordinarsi per attuare una strategia di squadra, come ad esempio raggiungere un target mobile. In questo caso ogni robot può possedere un agente di visione che effettua una propria stima sulla posizione del target. Nel sistema esisterà però soltanto un agente che elabora la strategia di gruppo.

In ogni istante vi sono tante informazioni relative alla posizione del target che devono convergere verso un unico agente ed essere fuse in un'unica stima, in modo che si possa attuare una strategia.

Una possibile soluzione a questo problema consiste nell'individuare tre categorie di agenti:

- Agente Factory: questo agente produce i dati relativi ad una certa ontologia. La produzione avviene in modo ripetitivo e non costante.
- Agente Consumer: questa categoria di agenti è interessata alla produzione dei dati del Factory per i propri scopi.
- Agente Dispatcher: questo agente è la rappresentazione della lavagna. Esso riceve i dati da parte di uno o più Factory e li conserva in modo che gli agenti Consumer possano accedervi.

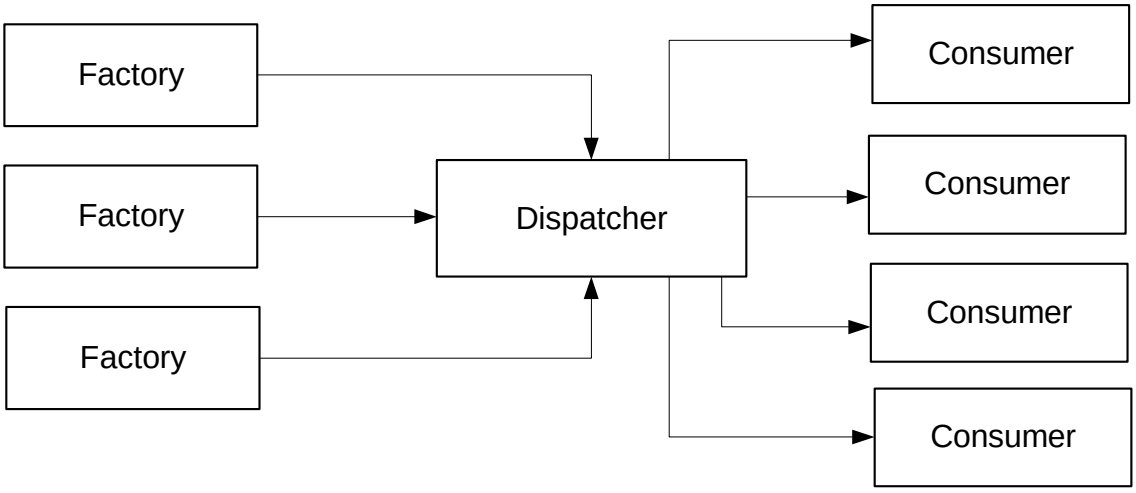


Figura 4-5 Scenario della comunicazione con lavagna condivisa

In uno scenario simile gli agenti che producono un determinato tipo di dato comunicano questa informazione al dispatcher. Il Dispatcher può essere programmato per trattare il dato ricevuto secondo diverse politiche:

- Può mantenere una history di tutti i dati ricevuti e fornire oppure sovrascrivere sempre i vecchi dati con quelli più recenti,
- Può gestire una lista degli agenti Consumer interessati all'informazione e notificare la presenza di un dato aggiornato, oppure attendere che siano gli agenti interessati a richiedere la versione più recente del dato,
- Può inoltre gestire la comunicazione del dato ai Consumer con diversi livelli di priorità.

La struttura di ogni agente relativo a questo scenario può essere rappresentato mediante dei diagrammi SASD:

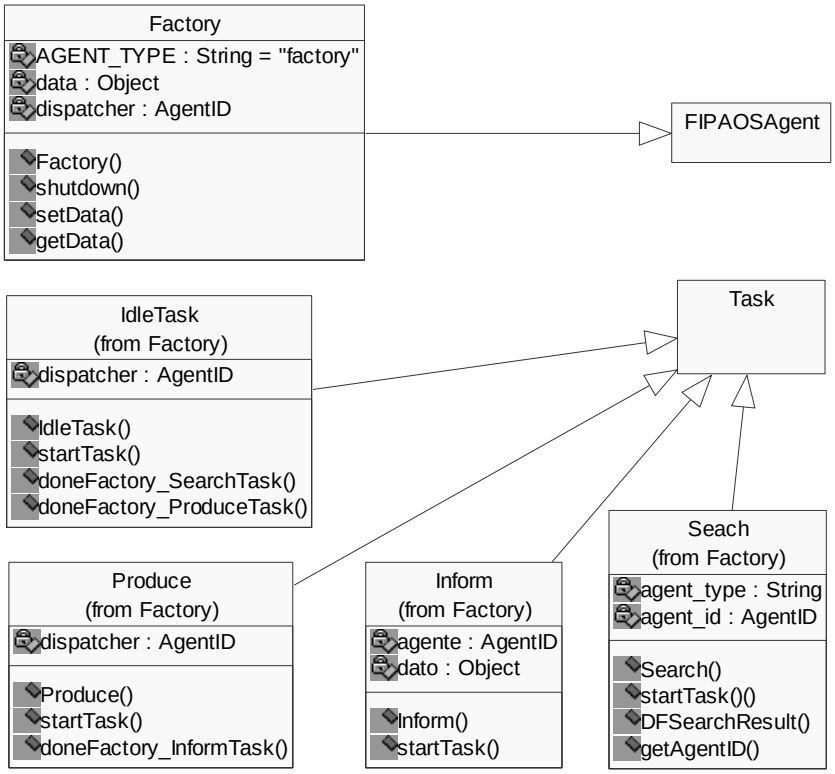


Figura 4-6 Diagramma SASD dell'agente Factory

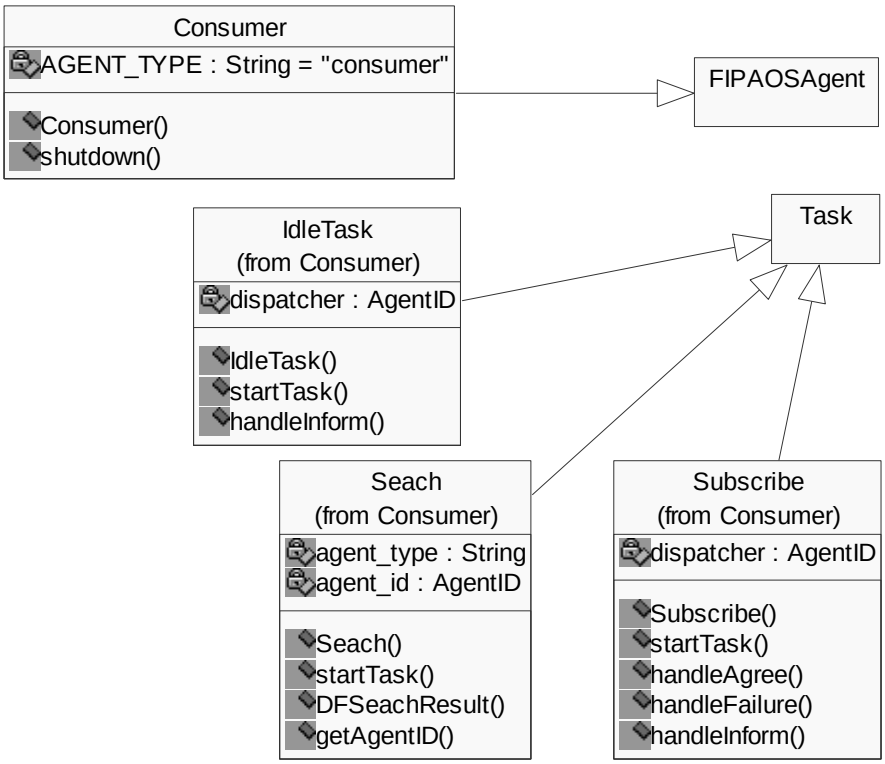


Figura 4-7 Diagramma SASD dell'agente Consumer

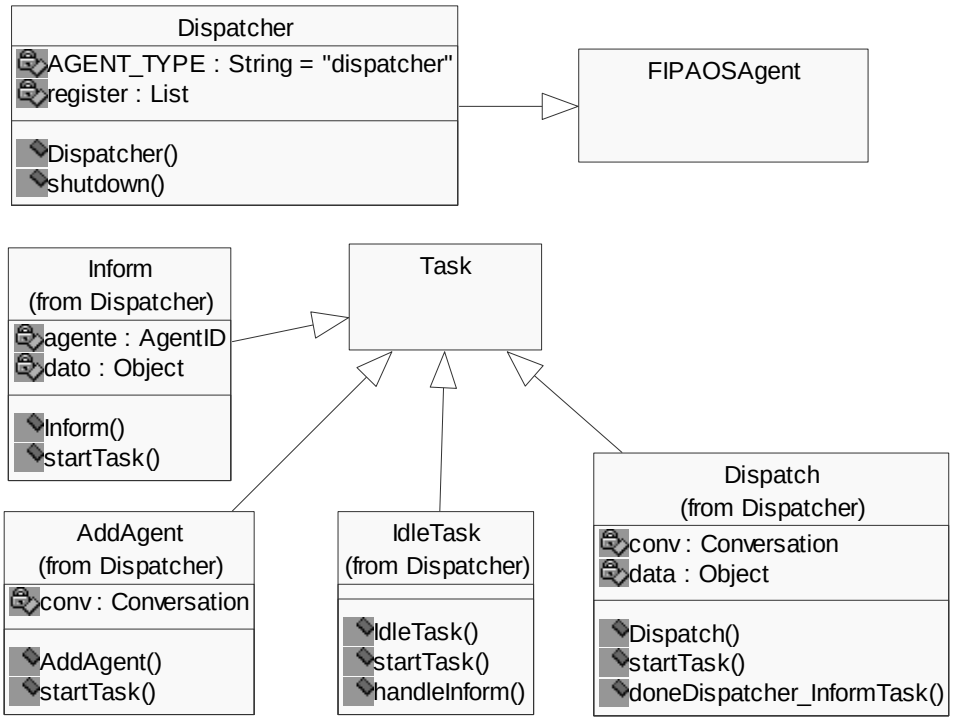


Figura 4-8 Diagramma SASD dell'agente Dispatcher

A sua volta l'interazione tra gli agenti può essere mostrata mediante dei diagrammi MABD. In figura 4-9 viene evidenziato il modo in cui l'agente Factory genera il dato in modo ciclico. Ad ogni ciclo il task Produce genera l'informazione e poi la invia all'agente Dispatcher.

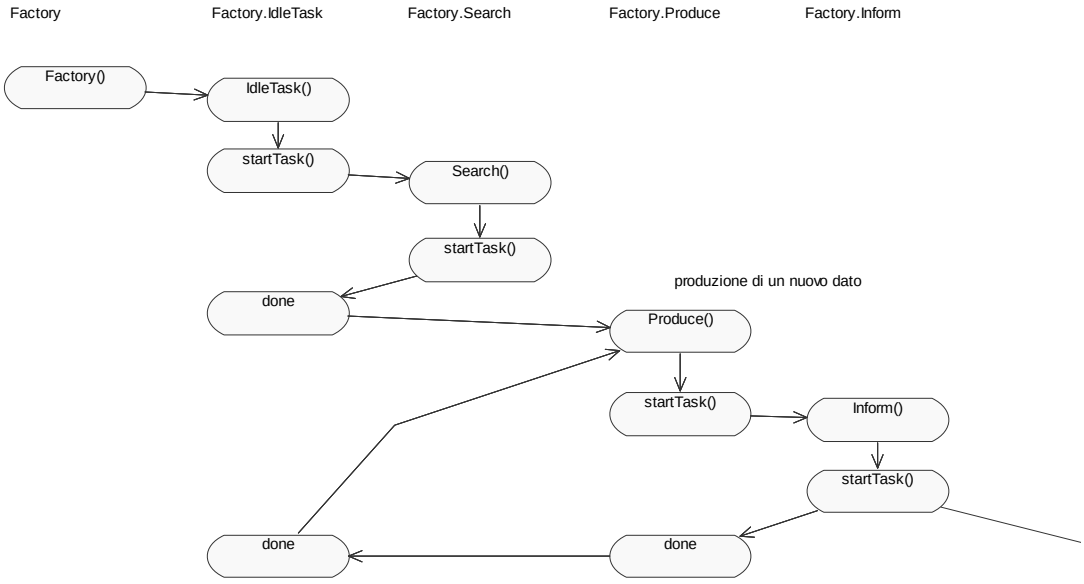


Figura 4-10 Porzione del diagramma MABD che evidenzia la produzione del dato da parte dell'agente Factory

Il diagramma in figura 4-11 invece mette in evidenza il modo in cui viene gestita la comunicazione tra Dispatcher e Consumer. Il Consumer per ottenere la notifica dell'informazione si iscrive ad un registro mantenuto dal Dispatcher.

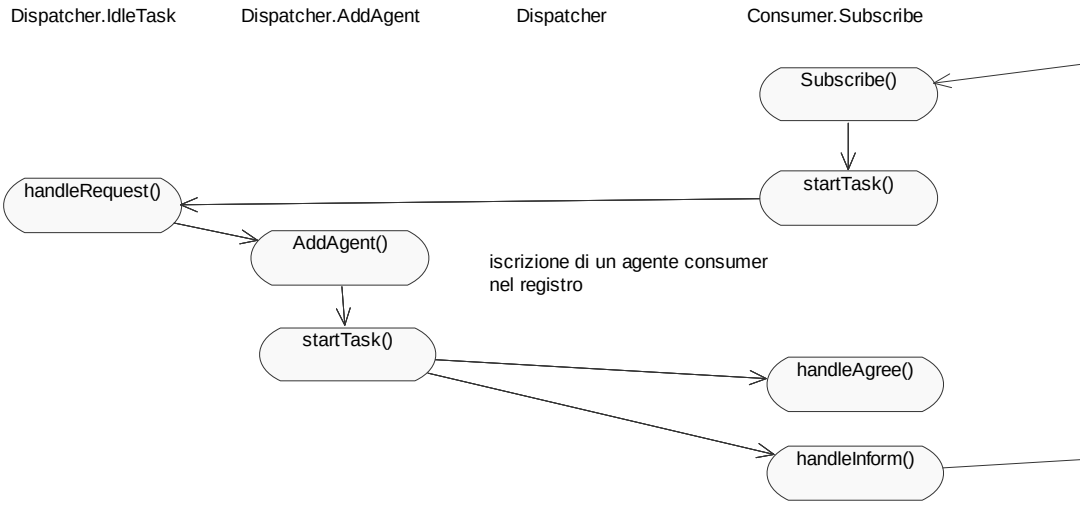


Figura 4-12 Porzione del diagramma MABD che evidenzia la sottoscrizione al Dispatcher da parte del Consumer

Quando il Dispatcher riceve la nuova informazione, la smista a tutti i nominativi del registro.

4.3.1.2 Classificazione dei pattern

L'analisi dello scenario d'esempio mette in evidenza che l'insieme dei diagrammi SASD e MABD è totalmente esplicativa per la rappresentazione di uno tipico scenario di progettazione. L'utilizzo di questi diagrammi permettono di catturare l'essenza di porzioni del progetto relative a problemi incontrati di frequente e di poterle riutilizzare ogni volta che questi si ripresentano.

Prendendo ispirazione dai design pattern, un pattern per agenti viene, quindi, definito oltre che da una coppia di diagrammi SASD e MABD, da una descrizione testuale del contesto nel quale poter inserire il pattern.

Una volta definita la nostra proposta, il passo successivo è quello di fornire una classificazione di pattern. In questa classificazione i pattern hanno target di applicazione differenti.

(Con il termine 'target' si intendono le entità coinvolte nell'applicazione del pattern.)

Nella classificazione, che di seguito è presentata, si può notare come ogni tipologia di pattern sia un'aggregazione di quelli delle categorie precedenti alla quale si aggiunge la logica di controllo.

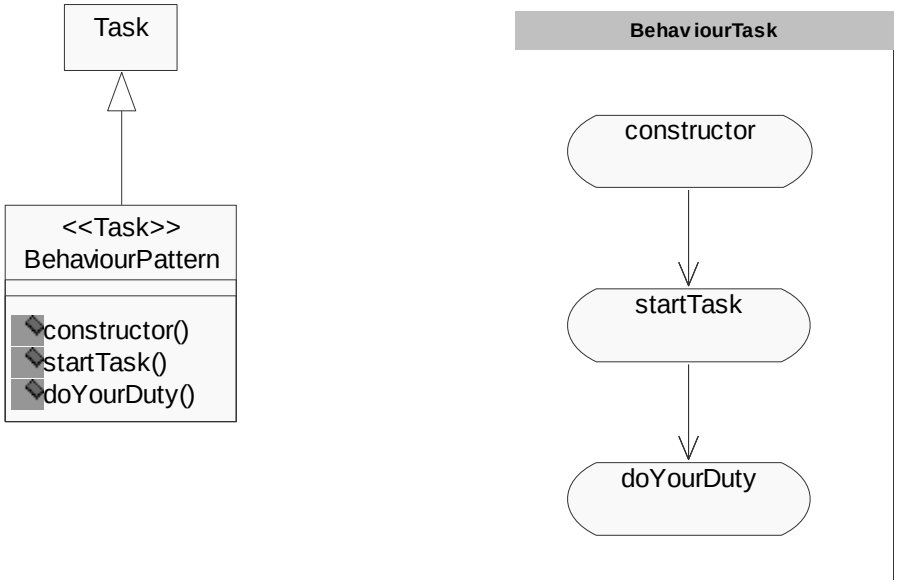
4.3.1.2.1 Pattern di Action

Il pattern di action è il tipo maggiormente legato all'implementazione. Un'action rappresenta una funzionalità del sistema, che può essere applicata ad una classe agente o ad una classe task. Un pattern di action è rappresentato da uno o più metodi e contiene anche le istruzioni che ne definiscono il corpo. Tale tipo di pattern è strettamente legato al linguaggio usato per l'implementazione e al framework ad agenti usato.

4.3.1.2.2 Pattern di Behaviour

Un pattern di behaviour definisce un comportamento specifico di un agente; il pattern è inteso come una collezione di funzionalità. A questo si aggiunge anche la logica di interconessione di queste funzionalità.

Il target di applicazione del pattern è un task.



SASD MABD

Figura 4-13 Coppia di diagrammi SASD e MABD relativi ad un pattern di behaviour

4.3.1.2.3 Pattern di Component

Un pattern di component coinvolge l'intera struttura dell'agente e dei suoi task. La struttura di un pattern di questo tipo è dato da una collezione di attributi e metodi per la classe agente, più le classi dei task (completi di attributi e metodi).

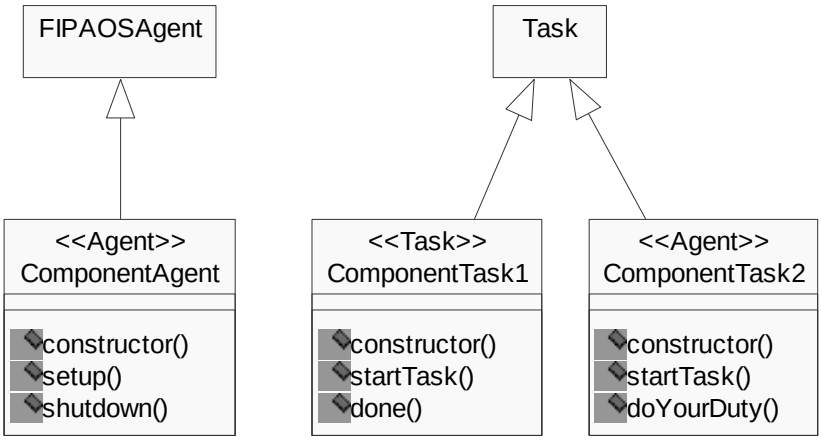


Figura 4-14 Diagramma SASD relativo ad un pattern di component

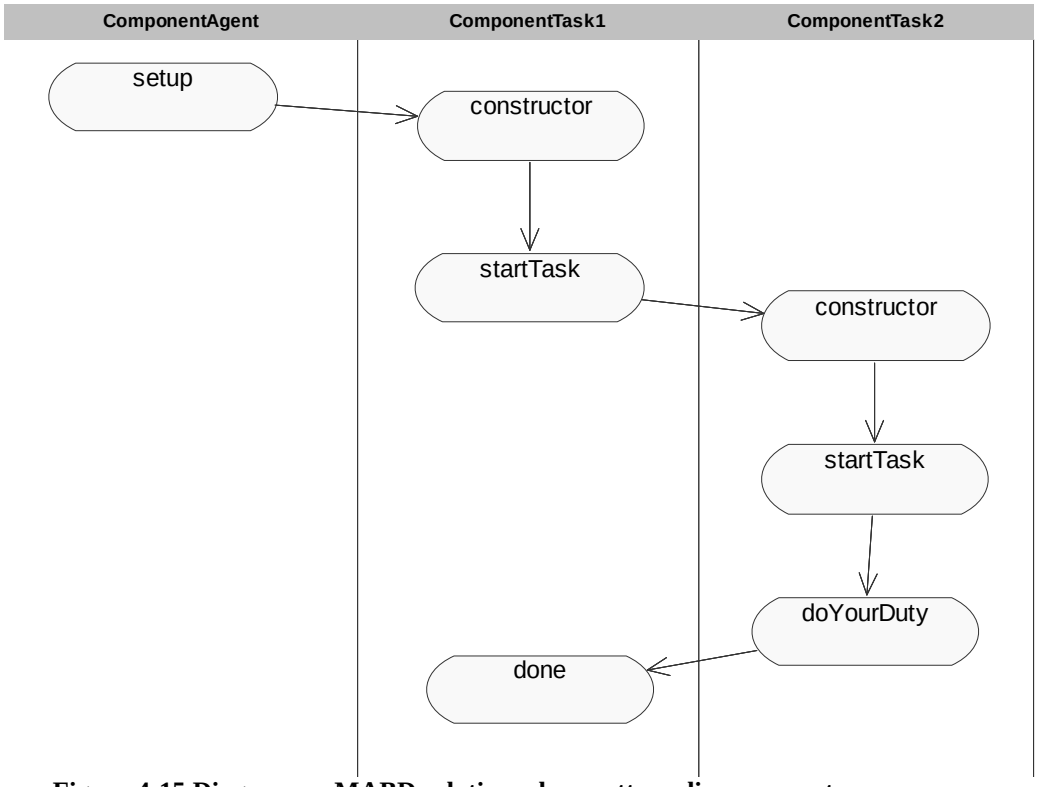


Figura 4-15 Diagramma MABD relativo ad un pattern di component

4.3.1.2.4 Pattern di Service

Un pattern di service coinvolge più di un agente in una collaborazione allo scopo di ottenere un risultato. Un pattern di questo tipo è un'aggregazione di component. Ogni component coinvolto costituisce un ruolo all'interno della collaborazione. I pattern di comunicazione rappresentano i service pattern a due ruoli (initiator e participant) più semplici che è possibile definire.

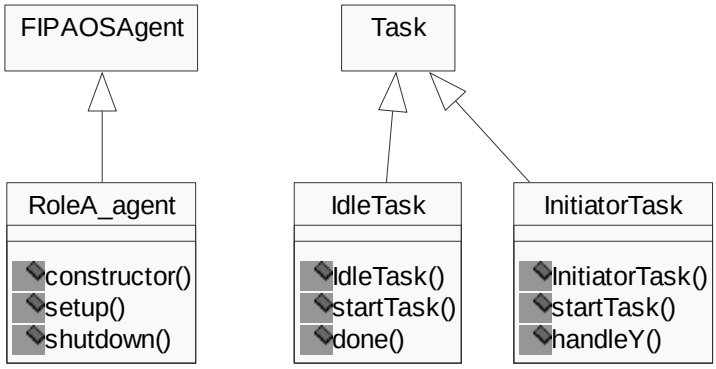


Figura 4-16 Diagramma SASD relativo al ruolo A del service pattern

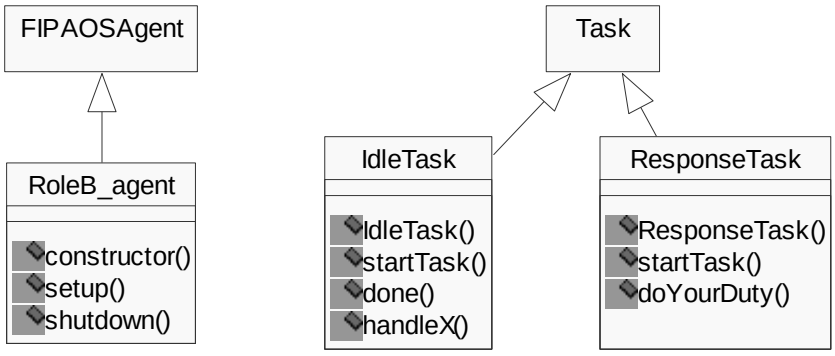


Figura 4-17 Diagramma SASD relativo al ruolo B del service pattern

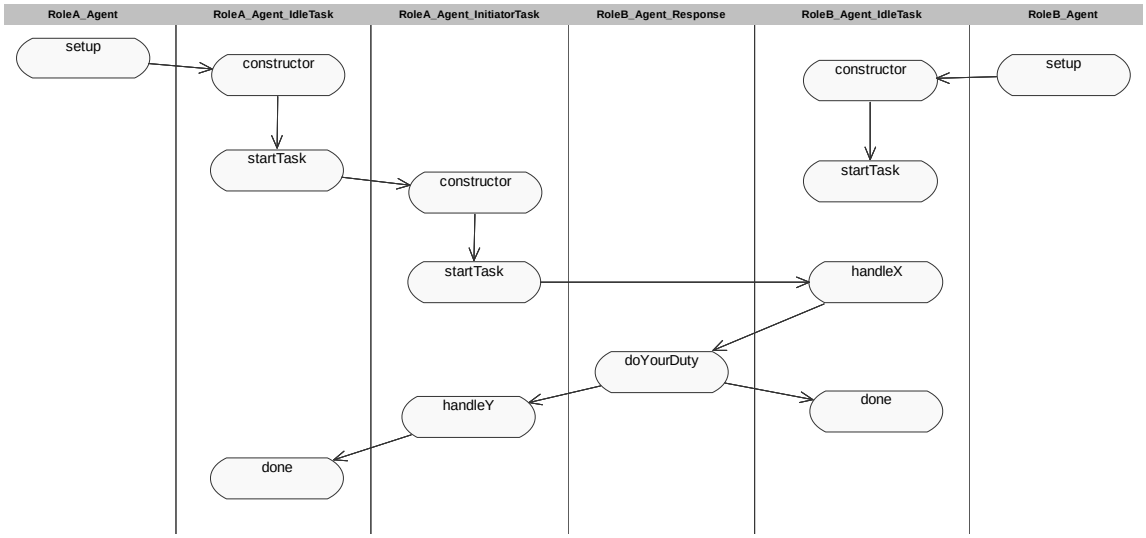


Figura 4-18 Interazione tra i due ruoli del service pattern

Poiché un service coinvolge più agenti e le interazioni tra di essi, i pattern di comunicazione sono sempre inclusi all'interno di un service pattern.

4.3.2 Dal progetto all'implementazione

La struttura proposta per la definizione dei pattern di agente si presta per una gestione automatica di un repository e per la generazione automatica del codice implementativo dell'agente.

Da questa osservazione nasce l'idea di sviluppare uno strumento in grado di gestire gli agenti, permettere l'applicazione di pattern e generare il codice dell'agente sul quale poter effettuare delle modifiche.

Di seguito diamo una breve descrizione dello strumento che è stato sviluppato.

Il tool presenta un'interfaccia grafica dalla quale è possibile accedere alle funzionalità del repository di pattern. (La descrizione delle funzionalità del repository è fornita nell'appendice.)

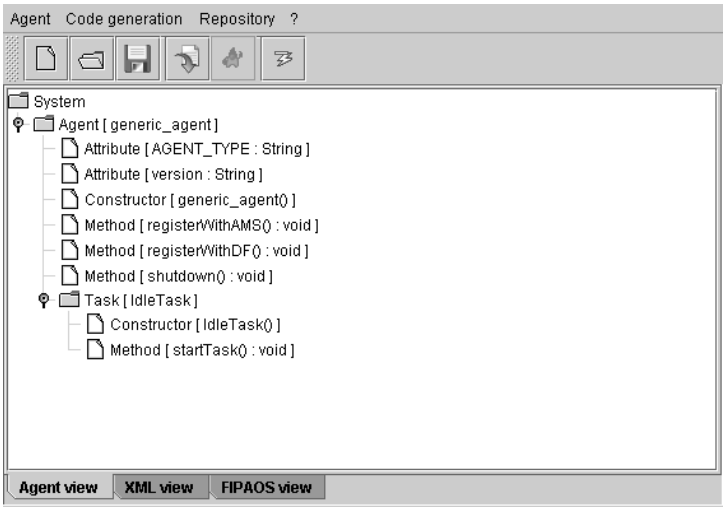


Figura 4-19 Schermata del tool che gestisce il repository di agenti e di patten e ne permette la modifica

Un agente viene rappresentato attraverso struttura statica ad albero che si riferisce in modo univoco al diagramma SASD.

Su questa struttura è possibile effettuare delle modifiche: aggiungere attributi, metodi e task.

In alternativa è possibile prelevare un pattern dal repository e applicarlo all'agente al quale si sta lavorando.

Per gestire la struttura dell'agente e il catalogo di pattern è stato necessario creare un metalinguaggio per la descrizione di sistemi ad agenti, agenti, task e pattern come descritto nel capitolo 4.4

La generazione del codice avviene costruendo lo scheletro dell'agente e dei suoi task dal diagramma delle classi.

Questa struttura viene completata inserendo gli action pattern relativi alla piattaforma ad agenti.

Il codice dell'agente viene quindi mostrato in un apposita finestra che riporta il nome FIPAOS View o Jade view a seconda della piattaforma scelta.

Il tool è stato studiato per lavorare in collaborazione con un plug-in di Rational Rose il cui scopo è quello di guidare l'utente nella realizzazione di un progetto secondo la metodologia PASSI. L'interazione avviene effettuando uno scambio di informazioni che permette all'utente di Rational Rose di accedere alle funzionalità del repository direttamente dal CASE tool.

Un'ulteriore funzionalità messa a disposizione dal tool consiste nel processo di reverse engineering. Dal codice di un agente, attraverso un processo di parsing, si può risalire alla struttura statica dell'agente.

4.4 Rappresentazione degli agenti in metacodice

L'utilizzo dei pattern all'interno di un progetto richiede che la struttura con la quale si descrive tali entità sia definita in modo univoco. Un design pattern è generalmente descritto in termini di costrutti software quali oggetti e classi, e di operatori come ad esempio l'ereditarietà o il polimorfismo.

La descrizione di un pattern mediante i tipici costrutti di programmazione introduce, infatti, una dipendenza dalla piattaforma ad agenti per la quale si vuole procedere allo sviluppo. Se da un lato questo provoca una perdita di generalità del pattern stesso, d'altro canto è necessario per rendere questo strumento realmente utilizzabile.

Nel lavoro svolto la piattaforma presa in esame è FIPAOS, di conseguenza un agente viene descritto in funzione dei costrutti di programmazione di java: classi, classi interne, metodi, attributi, eredità e relazioni gerarchiche tra gli elementi.

Una volta fissata la struttura dell'agente è possibile introdurre un meta-linguaggio per la descrizione delle entità di un sistema ad agenti (agenti e task) e dei relativi pattern.

L'introduzione di questo meta-linguaggio risulta un passo avanti nella definizione di uno strumento in grado di manipolare agenti e pattern per agenti all'interno di un progetto in modo automatico.

L'utilizzo del metalinguaggio infatti permette di effettuare delle manipolazioni di alto livello su un agente come l'introduzione di nuove funzionalità o la modifica di quelle esistenti.

Altro punto essenziale nella definizione del metalinguaggio è quello di poter creare un repository di agenti e di pattern per agenti nel quale catalogare gli elementi usati più di frequente in modo che possano essere riutilizzati facilmente.

Inoltre la gestione degli agenti attraverso un meta-linguaggio che ne descrive la struttura permette di effettuare la generazione automatica del codice relativo ad una piattaforma ad agenti specifica.

4.4.1 Descrizione di un agente

La rappresentazione dell'agente cui siamo interessati, essendo rivolta anche alla generazione del codice oltre che all'applicazione di pattern nel progetto, è di tipo implementativo; essa risulta pertanto legata al framework utilizzato per lo sviluppo degli agenti.

Sia FIPA-OS che Jade presentano una struttura ad agenti di tipo gerarchico: l'elemento radice della gerarchia è l'agente che contiene come sotto-elementi le sue proprietà, quali attributi, costruttori, metodi e task.

A sua volta ognuno di questi elementi presenta alcune caratteristiche proprie che vanno espresse come sotto-elementi della gerarchia.

Da questa analisi il linguaggio che si desidera utilizzare per la descrizione dell'agente deve essere in grado di gestire facilmente le strutture ad albero. XML risulta una buona scelta per questo scopo; esso è uno standard consolidato e facilmente portabile tra diverse piattaforme o sistemi operativi.

Una caratteristica interessante che nasce dalle scelte implementative effettuate è che la descrizione dell'agente può essere ricostruita automaticamente direttamente dal progetto. Essa infatti può essere ricavata direttamente dall'analisi del diagramma SASD (Single Agent Structure Description) e MABD (Multi Agent Behaviour Description) relativi alla metodologia di progettazione PASSI.

Il diagramma SASD è un diagramma delle classi UML nel quale viene rappresentata la struttura di un singolo agente. Un agente viene rappresentato mediante una classe e può possedere attributi, metodi e task intese come sottoclassi; a sua volta un task è una classe, ma può possedere solo attributi e metodi. Nelle fasi precedenti della metodologia il progettista deve aver già deciso molte caratteristiche del suo sistema: agenti, ruoli, comunicazioni, ontologie, task, obiettivi e sotto-obiettivi. In questa fase il lavoro è quello di modellare l'agente e i suoi task come generiche "classi" di un linguaggio di programmazione ad oggetti così come prevede lo standard UML.

4.4.1.1 Generazione del codice

Come precedentemente accennato la scelta di introdurre un meta-linguaggio permette di generare il codice di un agente in modo del tutto automatico. Il codice così sviluppato risulta naturalmente incompleto, in quanto mancano le parti relative al comportamento specifico dell'agente. Tuttavia questa funzionalità risulta ugualmente molto utile in quanto completa il processo di analisi e progettazione di un sistema ad agenti realizzato mediante la metodologia PASSI con la fase di sviluppo del codice. Risulta estremamente comodo per un progettista poter ricavare il codice del progetto a cui ha lavorato, semplicemente assemblando i moduli generati in modo automatico e aggiungendo ad essi soltanto le parti specifiche e uniche del sistema.

L'utilizzo di pattern in particolare rende la progettazione ancora più flessibile ed estendibile; maggiore è il numero di pattern usati nel progetto più la fase di generazione di codice può raggiungere livelli di riutilizzo interessanti.

Sebbene il linguaggio preso come obiettivo per lo sviluppo degli agenti sia java, gli elementi descrittivi del linguaggio sono volutamente astratti e comuni a molti linguaggi di programmazione orientati agli oggetti. Questa scelta è stata operata in modo che l'agente non fosse legato ad una particolare implementazione sia del sistema ad agenti, sia del linguaggio di programmazione usato.

In questo modo la generazione automatica di codice verso altre piattaforme è possibile senza apportare nessuna modifica al metalinguaggio.

4.4.2 Il meta-linguaggio

In questo paragrafo verrà analizzata la grammatica del metalinguaggio basato su XML usato per descrivere un agente.

Il DTD usato viene riportato per intero alla fine del paragrafo, mentre di seguito si vedrà in dettaglio il significato di tutti i tag definiti nella nostra grammatica.

4.4.2.1 XML

XML è uno standard istituito dal consorzio W3C (World Wide Web Consortium) nel tentativo di creare una sintassi generica per descrivere documenti, usando una sintassi simile indipendentemente dalla natura del contenuto. Il fattore che ha portato XML al successo è la

sua estrema flessibilità che ne permette l'impiego in ambiti completamente diversi e la sua completa portabilità tra sistemi operativi.

XML è un linguaggio di markup per documenti di tipo testuale che nasce come estensione del linguaggio HTML. Come per l'HTML i dati sono racchiusi tra alcuni tag e sono espressi sotto forma di stringhe di testo. A differenza dell'HTML però in XML non esistono limitazioni ai nomi dei tag, (XML è un linguaggio di meta-markup); l'utente può definire con quali elementi, sottoelementi e attributi sarà costituito il proprio documento.

Altra differenza rispetto ad un linguaggio come HTML è che XML mantiene una certa rigidità sintattica. Infatti un parser XML effettua una netta distinzione tra documenti *ben formati* e documenti validi. Un documento ben formato deve rispettare alcune semplici regole sintattiche dalle quali nessun documento XML può prescindere. Un documento valido invece è un documento che rispetta determinate caratteristiche espresse da un file DTD o XML Schema

XML viene supportato da una serie di tecnologie che costituiscono uno strato superiore il quale può essere facilmente usato nelle applicazioni: Xlinks, XSLT, Xpointers, Xpath, Namespaces, SAX, DOM, ecc.

La scelta di usare XML per la descrizione del meta-linguaggio è stata fatta in quanto si tratta di una tecnologia consolidata, e facilmente portabile su diverse piattaforme o sistemi operativi; questo ha permesso di realizzare un repository di pattern che possa essere trasferito su qualunque macchina che supporti java.

Un altro motivo che ha spinto verso questa scelta è il grande supporto da parte dei linguaggi di programmazione verso l'implementazione delle tecnologie connesse con XML. Molti parser sono disponibili su Internet per effettuare una analisi della struttura di un file XML, inoltre anche la definizione di una nuova grammatica definita dall'utente è molto semplice grazie ai DTD o a XML Schema.

Infine la maggior parte dei parser per XML offrono anche un engine per le trasformazioni XSLT che permettono di trasformare un file xml in qualunque altro formato testuale.

4.4.2.2 DTD

Il DTD (Document Type Definition) è un linguaggio usato in combinazione a XML per definire quali elementi ed entità possono apparire in un file XML e in che modo.

Lo standard XML prevede che un documento non valido sia comunque leggibile dalle applicazioni: in altre parole è il parser che deve decidere se ignorare eventuali violazioni alla struttura definita nel DTD. La responsabilità di questa decisione è lasciata al progettista dell'applicazione.

Un file DTD è composto da regole di composizione. Ogni regola definisce quali sotto-elementi possono comparire come figli di un elemento specificato.

Utilizzando un DTD in pratica si può specificare una grammatica per un file XML, definendo a tutti gli effetti un'applicazione XML.

4.4.2.3 Esempio di un agente

Di seguito si riporta il file xml usato per la descrizione di agente in grado di attivare una comunicazione di tipo Request. Questo esempio può essere utile per seguire meglio la descrizione degli elementi che compongono il metalinguaggio. Il diagramma relativo alla struttura di questo agente è il seguente:

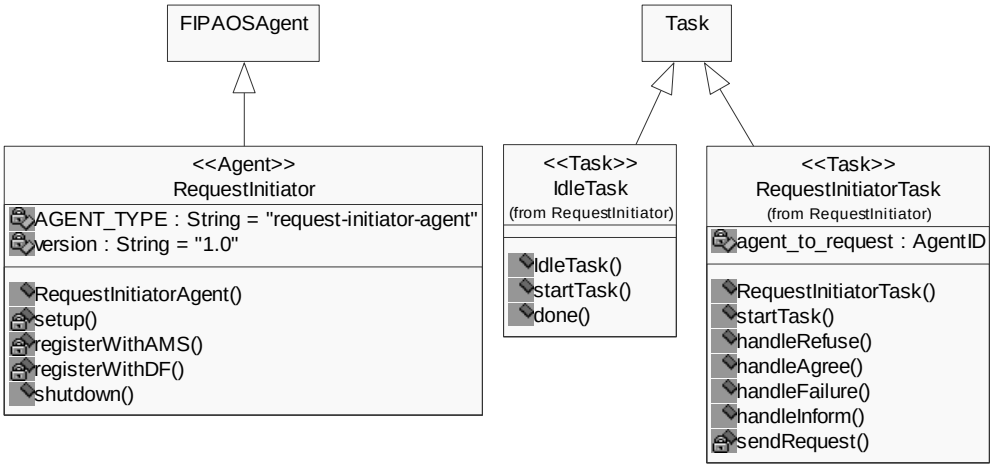


Figura 4-20 Struttura statica di un agente in grado di avviare una comunicazione Request

Questa struttura può essere rappresentata attraverso il metalinguaggio nella maniera seguente:

```
<Agent name="RequestInitiator">
    <Visibility>public</Visibility>
```

```
<Extends>FIPAOSAgent</Extends>
<Attribute type="String" name="AGENT_TYPE">
    <Visibility>private</Visibility>
    <Static/><Final/>
    <Value>"agent_name"</Value>
</Attribute>
<Attribute type="String" name="version">
    <Visibility>private</Visibility>
    <Static/><Final/>
    <Value>"1.0"</Value>
</Attribute>
<Constructor name="RequestInitiator">
    <Visibility>public</Visibility>
    <Argoment type="String" name="platform"/>
    <Argoment type="String" name="name"/>
    <Argoment type="String" name="ownership"/>
</Constructor>
<Method name="setup" type="void">
    <Visibility>private</Visibility>
</Method>
<Method name="register_WithAMS" type="void">
    <Visibility>private</Visibility>
</Method>
<Method name="register_WithDF" type="void">
    <Visibility>private</Visibility>
```

```
            </Method>
            <Method name="shutdown" type="void">
                <Visibility>public</Visibility>
            </Method>
            <Task name="RequestInitiatorTask">
                <Visibility>public</Visibility>
                <Extends>Task</Extends>
                <Attribute type="AgentID"
name="agent_to_request">
                    <Visibility>private</Visibility>
                </Attribute>
                <Constructor name="RequestInitiatorTask">
                    <Visibility>public</Visibility>
                    <Argoment type="AgentID" name="id"/>
                </Constructor>
                <Method name="startTask" type="void">
                    <Visibility>public</Visibility>
                </Method>
                <Method name="handleRefuse" type="void">
                    <Visibility>public</Visibility>
                    <Argoment type="Conversation"
name="conv"/>
                </Method>
                <Method name="handleAgree" type="void">
                    <Visibility>public</Visibility>
```

```
                    <Argoment type="Conversation"
name="conv"/>
                </Method>
                <Method name="handleFailure" type="void">
                    <Visibility>public</Visibility>
                    <Argoment type="Conversation"
name="conv"/>
                </Method>
                <Method name="handleInform" type="void">
                    <Visibility>public</Visibility>
                    <Argoment type="Conversation"
name="conv"/>
                </Method>
                <Method name="sendRequest" type="void">
                    <Visibility>private</Visibility>
                    <Argoment type="Object" name="data"/>
                </Method>
            </Task>
        </Agent>
    </Agent_description>
```

4.4.2.4 Rappresentazione UML del DTD

Per rappresentare un DTD utilizzando dei diagrammi UML è possibile usare la convenzione introdotta dalla Rational Rose [26][27], che sfrutta stereotipi di classe per definire gli elementi di un file XML.

<<DTDElement>>	<<<<DTDElement>>>> GenericElement attribute : CDATA	Rappresenta gli elementi presenti nel DTD. Gli attributi dell'elemento devono essere specificati come attributi della classe.
<<DTDElementEMPTY>>	<<DTDElementEMPTY>> Element	Rappresenta una specializzazione dello stereotipo DTDElement usato per quegli elementi che non possono avere sotto elementi.
<<DTDSequenceGroup>>	element_grp (from Agent_description)	Rappresenta una sequenza di sotto-elementi. Ogni Sotto-elemento è caratterizzato da un vincolo numerico che specifica l'ordine nella sequenza: {1}, {2}, …{n}. Questo elemento è un operatore di "AND ordinato".
<<DTDChoiceGroup>>	element_grp (from Agent_description)	Indica che soltanto uno dei sotto elementi specificati può apparire nell'albero xml. Rappresenta un operatore di "OR esclusivo".

La molteplicità nelle relazioni tra gli elementi illustrati specifica il numero di occorrenze di un elemento in un'istanza di albero XML.

4.4.2.5 Codifica di un agente

Di seguito saranno esposti e motivati tutti gli elementi che costituiscono il meta-linguaggio per la descrizione di un agente.

4.4.2.5.1 Element: Agent_description

L'elemento Agent_description costituisce il root di ogni file XML del nostro metalinguaggio.

```
<!ELEMENT Agent_description
 (Package?, Import*, Global_variable*, Agent*, Task*, Class*, Interface*)>
<!ATTLIST Agent_description
    agent_system CDATA #REQUIRED
>
```

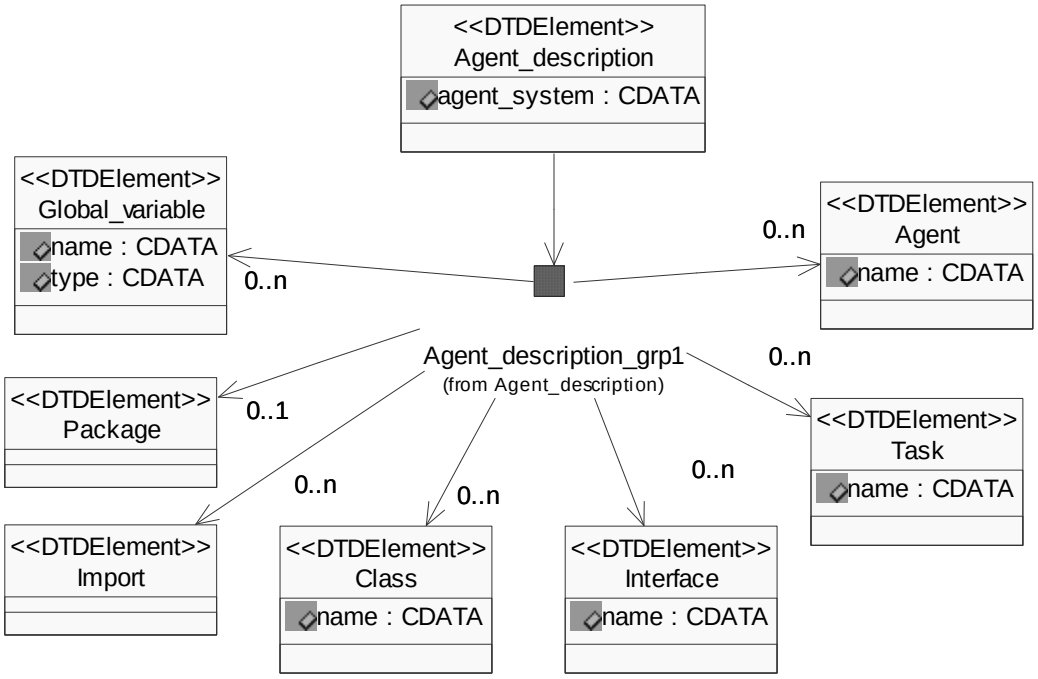


Figura 4-21 Diagramma DTD dell'elemento Agent_system

Attributi		
agent_system	required	Questo attributo descrive quale sistema ad agenti deve essere usato per generare il codice relativo all'agente descritto di

		seguito. Se viene specificato il valore “unknown” allora l’agente è completamente astratto.
Sotto-elementi		
Package	0..1	Un agente può appartenere ad un *package*, inteso come gruppo di classi e funzionalità con un fine comune.
Import	0..n	Un agente può contenere dei legami verso altri package. Tali legami possono essere specificati tramite elementi *import*.
Global_variable	0..n	Tramite questa clausola è possibile definire oggetti globali, esterni all’agente, ma indispensabili al suo funzionamento.
Agent	0..n	Tramite questo elemento è possibile definire un *agente* e i suoi attributi, metodi e task.
Task	0..n	E’ possibile definire *task* anche esterni alla struttura dell’agente. Si può trattare di task generici o di pattern.
Class	0..n	E’ possibile definire *classi* esterni alla struttura dell’agente. Si può trattare di strutture dati o di classi relative all’ontologia di una conversazione.
Interface	0..n	E’ possibile definire *interfacce astratte* anche esterne alla struttura dell’agente. Si può trattare di raccolte di funzionalità e azioni.

4.4.2.5.2 Element: Agent

L'elemento Agent fornisce la descrizione di un agente, degli attributi, dei metodi e dei task che lo costituiscono.

```
<!ELEMENT Agent
     (Description?, Abstract?, Final?, Visibility, Extends?,
 Implements*, Attribute*, Constructor*, Method*, Task*, Class*)>
<!ATTLIST Agent
     name CDATA #REQUIRED
>
```

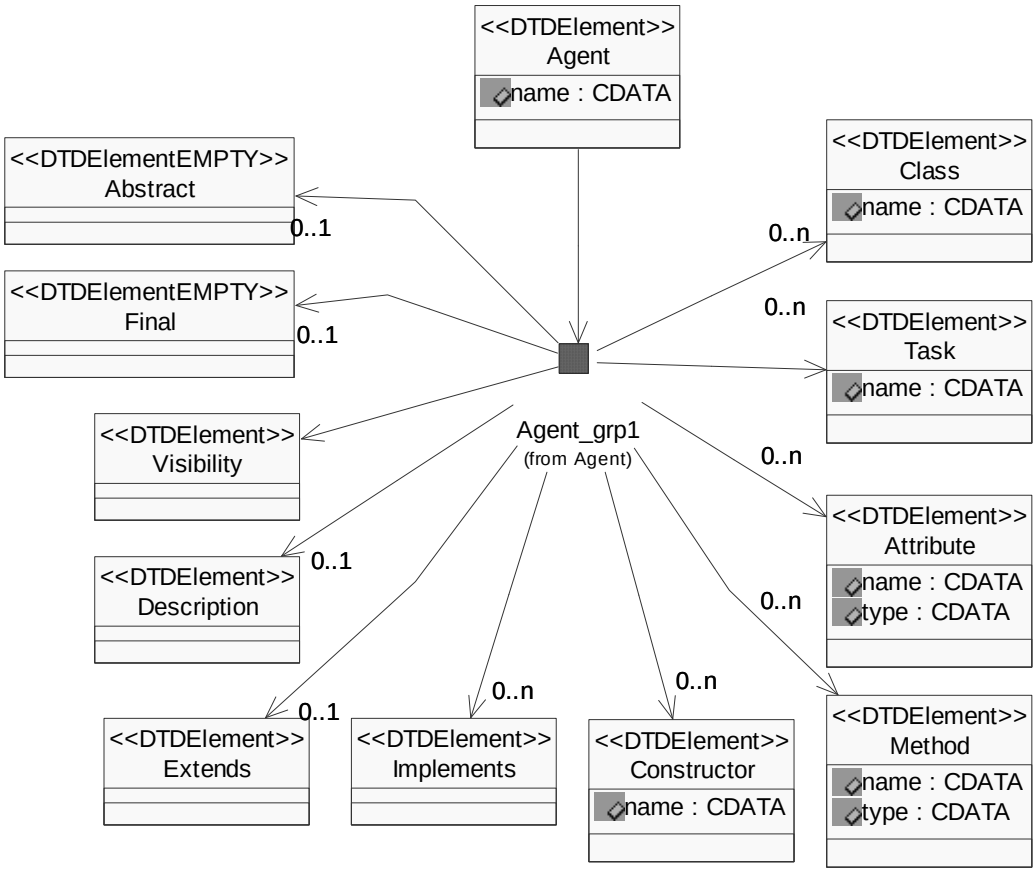


Figura 4-22 Diagramma DTD dell'elemento Agent

Attributi

name	required	Questo attributo è il nome assegnato all'agente.
Sotto-elementi		
Description	0..1	Un agente può avere una descrizione sul suo scopo e sul funzionamento. Questa opzione è utile soprattutto se l'agente deve essere trattato con strumenti di manipolazione automatica.
Abstract	0..1	Se è presente indica che l'agente è astratto e deve essere esteso per poter essere istanziato.
Final	0..1	Se è presente indica che l'agente non può essere ereditato da un altro agente.
Visibility	1	Indica il tipo di visibilità dell'agente verso l'esterno. I valori tipici sono *public*, *private* e *protected*, sebbene possano essere usati anche altri valori (come *package* o *implementation*), definiti da piattaforme specifiche.
Extends	0..1	Per un agente questo valore è univocamente definito dalla piattaforma usata. Ad esempio in FIPAOS il valore è "FIPAOSAgent", mentre in Jade è "Agent".
Implements	0..n	Indica che l'agente è conforme ad una o più interfacce.
Attribute	0..n	Descrive un attributo contenuto nell'agente.
Constructor	0..n	Descrive un costruttore dell'agente.
Method	0..n	Descrive un metodo dell'agente.

Task	0..n	Descrive un task contenuto nell'agente.
Class	0..n	Descrive una sottoclasse dell'agente.

4.4.2.5.3 Element: Task

L'elemento Task fornisce la descrizione di un task, degli attributi, dei metodi e delle sottoclassi che lo costituiscono.

```
<!ELEMENT Task
     (Description?, Abstract?, Final?, Visibility, Extends?,
 Implements*, Attribute*, Constructor*, Method*, Class*)>
<!ATTLIST Task
    name CDATA #REQUIRED
>
```

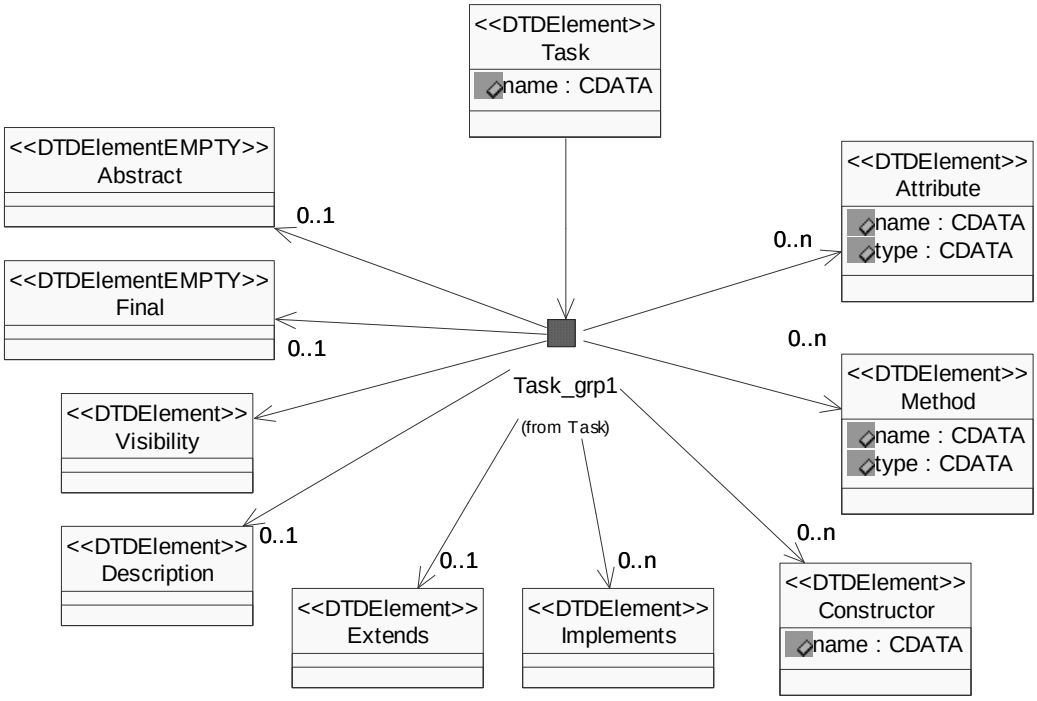


Figura 4-23 Diagramma DTD dell'elemento Task

Attributi		
name	required	Questo attributo è il nome assegnato al task.
Sotto-elementi		
Description	0..1	Un task può avere una descrizione sul suo scopo e sul funzionamento. Questa opzione è utile soprattutto se il task deve essere trattato con strumenti di manipolazione automatica.
Abstract	0..1	Se presente indica che il task è astratto e deve essere esteso da un altro task per poter essere istanziato.
Final	0..1	Se presente indica che il task non può essere ereditato da un altro task.
Visibility	1	Indica il tipo di visibilità del task verso l'esterno. I valori tipici sono *public*, *private* e *protected*, sebbene possano essere usati anche altri valori (come *package* o *implementation*), definiti da piattaforme specifiche.
Extends	0..1	Per un task questo valore è univocamente definito dalla piattaforma ad agenti usata. In FIPAOS il valore è "Task", mentre in Jade è "Behaviour" o una sua estensione.
Implements	0..n	Indica che il task è conforme ad una o più interfacce.
Attribute	0..n	Descrive un attributo contenuto nel task.
Constructor	0..n	Descrive un costruttore del task.

Method	0..n	Descrive un metodo del task.
Task	0..n	Descrive un task contenuto nel task.
Class	0..n	Descrive una sottoclasse del task.

4.4.2.5.4 Element: Class

L'elemento Class fornisce la descrizione generica di un oggetto che può essere usato all'interno di Agenti o di Task o come contenuto di una conversazione. Una classe può possedere degli attributi, dei metodi e delle sottoclassi.

```
<!ELEMENT Class
      (Description?, Abstract?, Final?, Visibility, Extends?,
 Implements*, Attribute*, Constructor*, Method*, Class*)>
<!ATTLIST Class
     name CDATA #REQUIRED
>
```

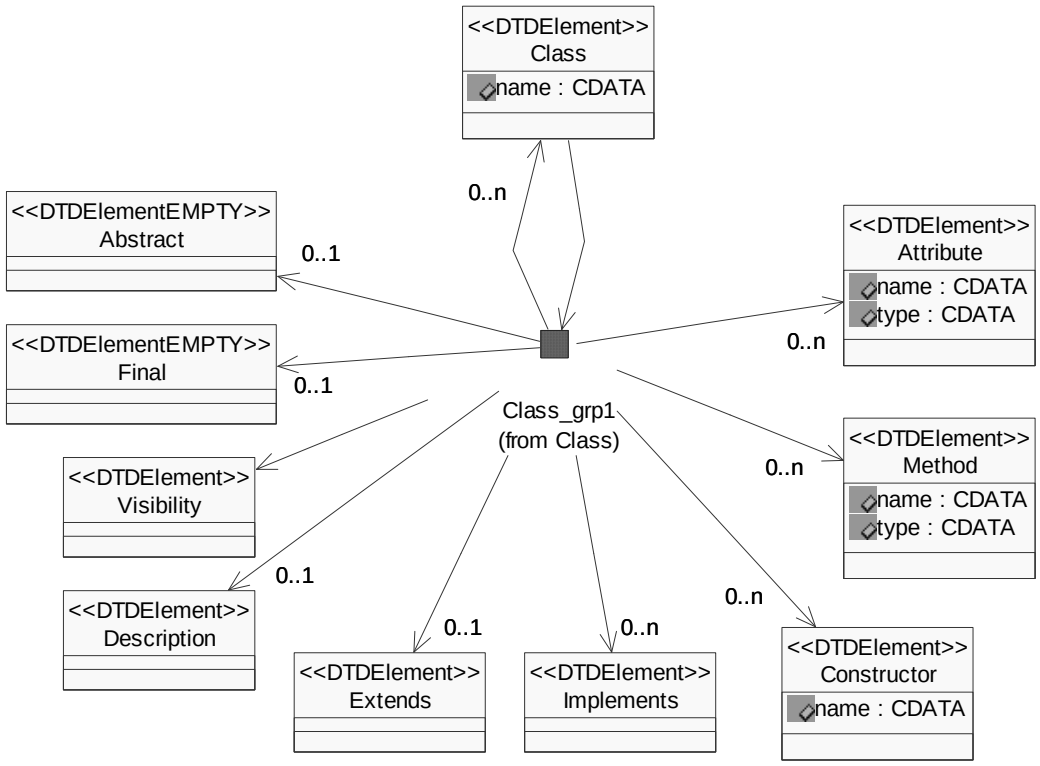


Figura 4-24 Diagramma DTD dell'elemento Class

Attributi		
name	required	Questo attributo è il nome assegnato alla classe.
Sotto-elementi		
Description	0..1	Una classe può contenere una breve descrizione sul suo scopo e sul funzionamento.
Abstract	0..1	Se presente indica che la classe è astratta e deve essere estesa da un altra classe per poter essere istanziata.
Final	0..1	Se presente indica che la classe non può

		essere ereditata da un altra classe.
Visibility	1	Indica il tipo di visibilità della classe verso l'esterno. I valori tipici sono *public*, *private* e *protected*, sebbene possano essere usati anche altri valori (come *package* o *implementation*), definiti da piattaforme specifiche.
Extends	0..1	Indica che la classe estende le funzionalità di un'altra classe.
Implements	0..n	Indica che la classe è conforme ad una o più interfacce.
Attribute	0..n	Descrive un attributo contenuto nella classe.
Constructor	0..n	Descrive un costruttore della classe.
Method	0..n	Descrive un metodo della classe.
Task	0..n	Descrive un task contenuto nella classe.
Class	0..n	Descrive una sottoclasse della classe.

4.4.2.5.5 Element: Interface

L'elemento Interface definisce una raccolta di funzionalità. Qualora un agente, un task o una classe implementino tali interfaccia, devono fornire tali funzionalità all'esterno.

```
<!ELEMENT Interface
 (Description?, Visibility, Extends*, Attribute*, Constructor*,
Method*)>
<!ATTLIST Interface
     name CDATA #REQUIRED
>
```

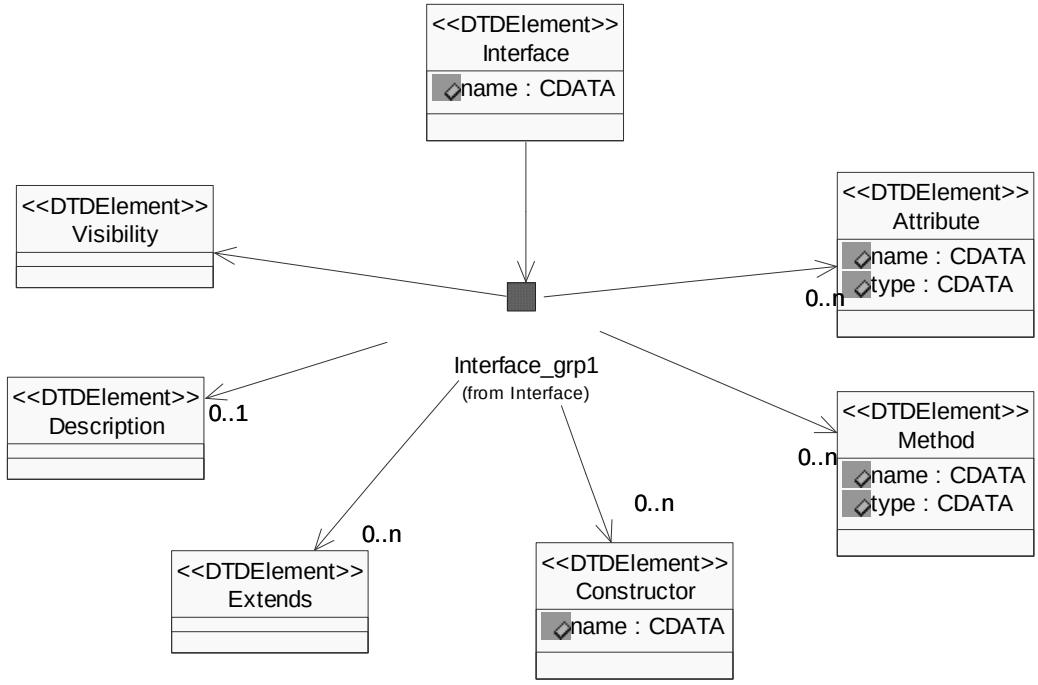


Figura 4-25 Diagramma DTD dell'elemento Interface

Attributi		
name	required	Questo attributo è il nome assegnato all'interfaccia.
Sotto-elementi		
Description	0..1	Una interfaccia può contenere una breve descrizione sul suo scopo e sulle funzionalità che supporta.
Visibility	1	Indica il tipo di visibilità dell'interfaccia verso l'esterno. I valori tipici sono *public*, *private* e *protected*, sebbene possano essere usati anche altri valori (come *package* o *implementation*), definiti da piattaforme specifiche.

Extends	0..n	Indica che l'interfaccia eredita le funzionalità messe a disposizione da un'altra interfaccia.
Attribute	0..n	Descrive un attributo che deve essere contenuto nell'agente, task o classe che implementa tale interfaccia. Il valore di visibilità dell'attributo deve essere impostato su public.
Constructor	0..n	Descrive un costruttore che deve essere contenuto nell'agente, task o classe che implementa tale interfaccia. Il valore di visibilità del costruttore deve essere impostato su public.
Method	0..n	Descrive un metodo che deve essere contenuto nell'agente, task o classe che implementa tale interfaccia. Il valore di visibilità del metodo deve essere impostato su public.

4.4.2.5.6 Element: Attribute

L'elemento Attribute costituisce un oggetto interno all'agente, task o classe. Il paradigma di programmazione ad oggetti prevede (ma non impone) che gli attributi siano sempre settati con visibilità *private* e che l'accesso possa avvenire mediante particolari metodi *public*.

```
<!ELEMENT Attribute
 (Visibility, Static?, Final?, Native?, Volatile?, Value?)>
<!ATTLIST Attribute
     name CDATA #REQUIRED
     type CDATA #REQUIRED
>
```

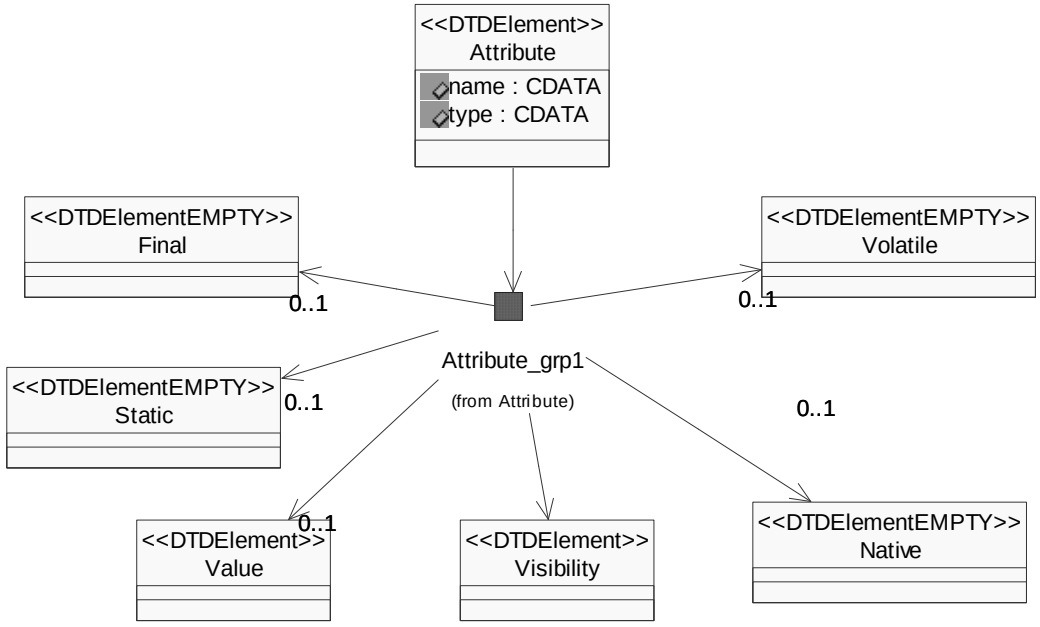


Figura 4-26 Diagramma DTD dell'elemento Attribute

Attributi		
name	required	Questo attributo è il nome assegnato alla variabile di classe.
type	required	Questo attributo è il tipo assegnato alla variabile di classe. Per convenzione per definire che si tratta di un array si usa il suffisso parentesi quadra aperta parentesi quadra chiusa: [].
Sotto-elementi		
Final	0..1	Indica che il valore dell'attributo non può essere modificato. Usando questo elemento si specifica una costante.
Static	0..1	Indica che l'attributo non appartiene ad

		una particolare istanza della classe, ma è comune a tutti gli oggetti della classe. Usando questo elemento si definisce una variabile di classe.
Native	0..1	Questo elemento è stato aggiunto per rendere il metalinguaggio completamente compatibile con Java. Un attributo native rappresenta un oggetto esterno al codice Java, scritto con un qualunque altro linguaggio di programmazione.
Volatile	0..1	Indica che il valore dell'attributo può variare per cause non specificate. Un esempio tipico è il valore timer interno.
Visibility	1	Indica il tipo di visibilità dell'attributo verso l'esterno. Il valore standard è *private*.
Value	0..1	Indica il valore iniziale dell'attributo.

4.4.2.5.7 Element: Method

L'elemento Method costituisce un metodo interno all'agente, task o classe.

```
<!ELEMENT Method
      (Visibility,    Abstract?,    Static?,    Final?,    Native?,
 Synchronized?, Argoment*, Throws*, Code?)>
<!ATTLIST Method
     name CDATA #REQUIRED
     type CDATA #REQUIRED
>
```

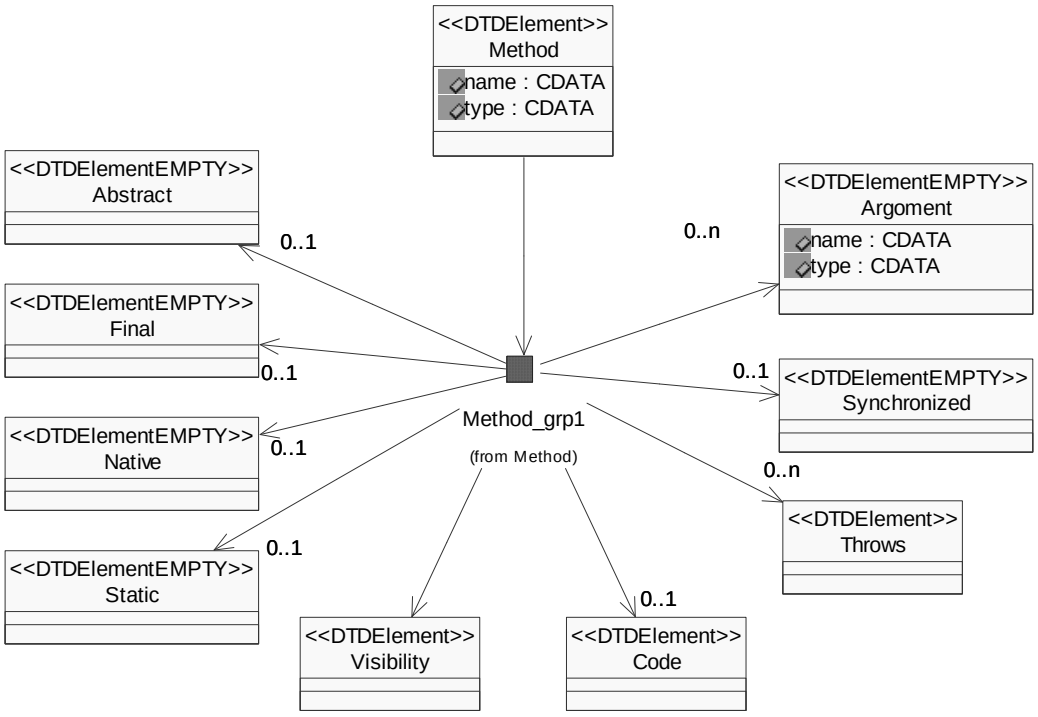


Figura 4-27 Diagramma DTD dell'elemento Method

Attributi		
name	required	Questo attributo è il nome assegnato al metodo.
type	required	Questo attributo è il tipo ritornato dal metodo. Per convenzione per definire che il metodo non restituisce niente, si usa il tipo *void*.
Sotto-elementi		
Abstract	0..1	Un metodo astratto non contiene codice. Esso deve essere ridefinito all'interno della sottoclasse concreta.
Final	0..1	Indica che il metodo non può essere ridefinito all'interno della sottoclasse che

		estende la classe di appartenenza.
Static	0..1	Indica che il metodo non appartiene ad una particolare istanza della classe, ma è comune a tutti gli oggetti della classe. Usando questo elemento si definisce un metodo di classe.
Native	0..1	Questo elemento è stato aggiunto per rendere il metalinguaggio completamente compatibile con Java. Un metodo native rappresenta una chiamata di funzione esterna al codice Java, scritta con un qualunque altro linguaggio di programmazione.
Visibility	0..1	Indica il tipo di visibilità del metodo verso l'esterno. I valori tipici sono *public*, *private* e *protected*, sebbene possano essere usati anche altri valori (come *package* o *implementation*), definiti da piattaforme specifiche.
Synchronized	0..1	Indica che il metodo accede a dati condivisi da più thread. Questa clausola regola l'accesso al metodo in modo che non vi siano perdite di dati.
Throws	0..n	Indica che il metodo può generare delle eccezioni che devono essere catturate dall'esterno.
Argoment	0..n	Indica che il metodo accetta dei parametri in ingresso.
Code	0..1	Indica che il metodo possiede del codice. Il valore di questo elemento è un campo CDATA che contiene il codice oppure un

		riferimento esterno ad un file che lo contiene.

4.4.2.5.8 Element: Constructor

L'elemento Constructor costituisce un metodo speciale all'interno dell'agente, task o classe. Tale metodo viene richiamato al momento dell'instanziazione e contiene il codice di inizializzazione.

```
<!ELEMENT Constructor
      (Visibility,  Abstract?,  Static?,  Final?,  Native?,
 Synchronized?, Argoment*, Throws*, Code?)>
<!ATTLIST Method
     name CDATA #REQUIRED
>
```

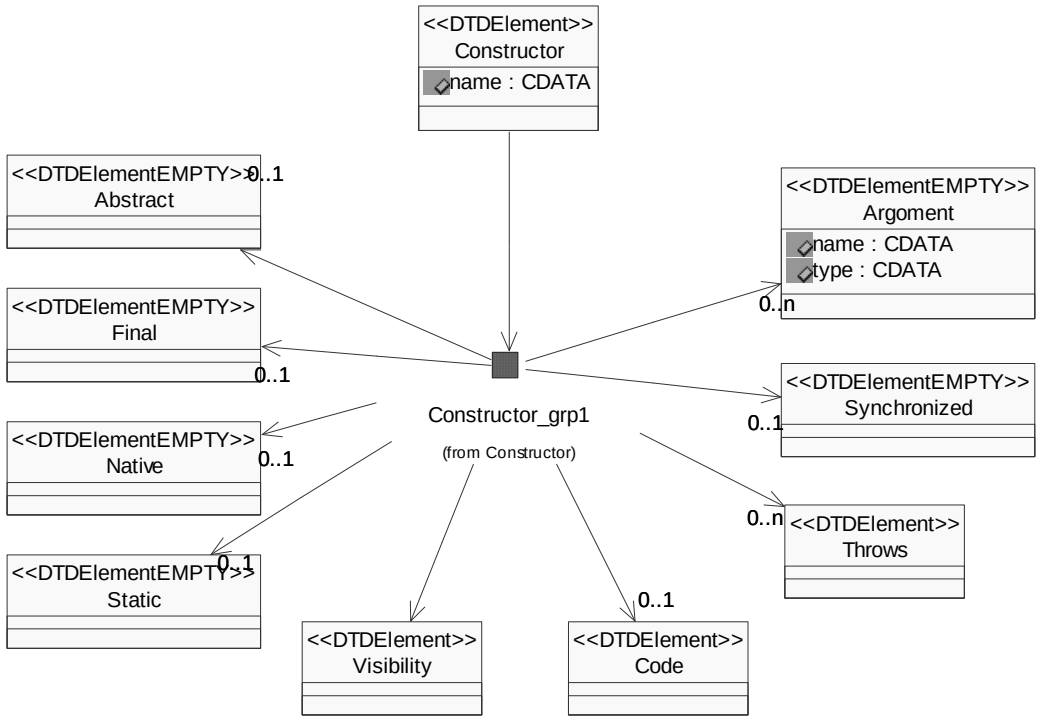


Figura 4-28 Diagramma DTD dell'elemento Constructor

Attributi		
name	required	Questo attributo è il nome assegnato al costruttore. In genere la sintassi dei linguaggi di programmazione prevede che il nome sia identico a quello della classe di appartenenza.
Sotto-elementi		
Abstract	0..1	Un costruttore astratto non contiene codice. Esso deve essere ridefinito all'interno della sottoclasse concreta.
Final	0..1	Indica che il costruttore non può essere ridefinito all'interno della sottoclasse che estende la classe di appartenenza.

Visibility	0..1	Indica il tipo di visibilità del costruttore verso l'esterno. I valori tipici sono *public*, *private* e *protected*, sebbene possano essere usati anche altri valori (come *package* o *implementation*), definiti da piattaforme specifiche.
Synchronized	0..1	Indica che il costruttore accede a dati condivisi da più thread. Questa clausola regola l'accesso al metodo in modo che non vi siano perdite di dati.
Throws	0..n	Indica che il costruttore può generare delle eccezioni che devono essere catturate dall'esterno.
Argoment	0..n	Indica che il costruttore accetta dei parametri in ingresso.
Code	0..1	Indica che il costruttore possiede del codice. Il valore di questo elemento è un campo CDATA che contiene il codice oppure un riferimento esterno ad un file che lo contiene.

4.4.2.5.9 Element: Argoment

L'elemento Argoment costituisce un parametro di un metodo o un costruttore.

```
<!ELEMENT Argoment EMPTY>
<!ATTLIST Argoment
     name CDATA #REQUIRED
     type CDATA #REQUIRED
>
```

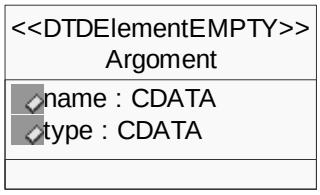


Figura 4-29 Diagramma DTD dell'elemento Argoment

Attributi		
name	required	Questo attributo è il nome assegnato all'argomento
type	required	Questo attributo è il tipo assegnato all'argomento

4.4.2.5.10 Altri elementi

Altri elementi usati nel nostro metalinguaggio sono i segenti:

```
<!ELEMENT Description (#PCDATA)>
<!ELEMENT Abstract EMPTY>
<!ELEMENT Final EMPTY>
<!ELEMENT Static EMPTY>
<!ELEMENT Native EMPTY>
<!ELEMENT Volatile EMPTY>
<!ELEMENT Synchronized EMPTY>
<!ELEMENT Visibility (#PCDATA)>
<!ELEMENT Extends (#PCDATA)>
<!ELEMENT Implements (#PCDATA)>
<!ELEMENT Throws (#PCDATA)>
<!ELEMENT Value (#PCDATA)>
<!ELEMENT Code (#PCDATA)>
```

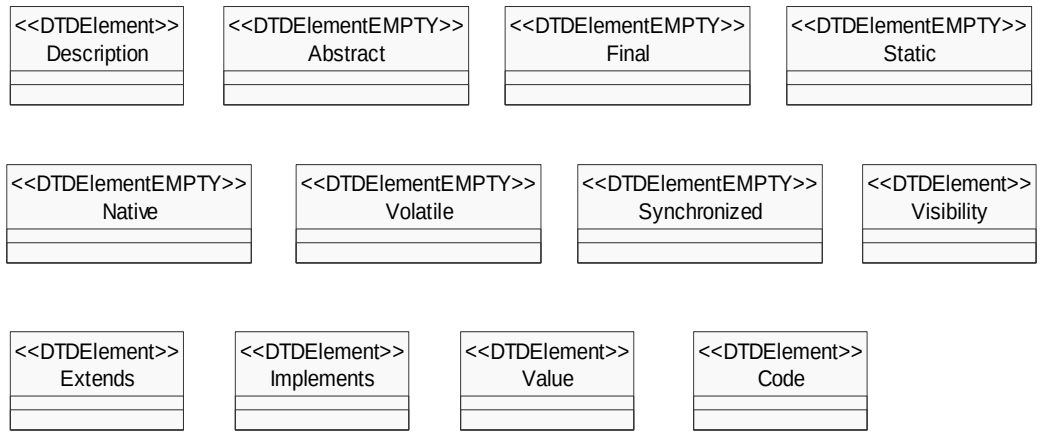


Figura 4-30 Diagramma degli elementi semplici del linguaggio

Nessuno di questi elementi possiede attributi. Gli elementi contrassegnati dallo stereotipo DTDElementEMPTY non posseggono elementi figli. Quelli contrassegnati dallo stereotipo DTDElement devono contenere un figlio di tipo Text.

4.4.3 Estensione del metalinguaggio

Il linguaggio che è stato sviluppato permette di esprimere un pattern per agente. Un pattern per agente può essere visto come un particolare agente (o una sua parte) che risolve un problema specifico di progettazione.

Per la descrizione di questa entità si usa la struttura decritta fin qui, con l'aggiunta dei vincoli.

La necessità di introdurre i vincoli è dovuta al fatto che quando si applica un pattern ad un progetto, si tende ad introdurre una entità nuova che modifica il sistema nel quale viene inserito.

L'aggiunta di nuove funzionalità si esplica nell'introduzione di nuovi metodi, attributi e task all'agente. Questa operazione può essere sottoposta a vincoli strutturali. Per esempio quando in FIPA-OS si vuole aggiungere ad un agente la capacità di gestire una comunicazione, in genere si inserisce un task apposito dedito a gestire la comunicazione.

Questo è corretto ma non è del tutto completo, infatti, oltre al task di comunicazione, deve esistere un task Listener in grado di ascoltare le comunicazioni in ingresso per l'agente. Tale task (che è una entità diversa dal task di comunicazione) deve inoltre possedere un metodo *handle* specifico per il tipo di comunicazione che si vuole gestire.

Questa relazione tra il pattern e il sistema può essere gestita agevolmente mediante l'uso di vincoli. Si può infatti specificare che quando un pattern di comunicazione viene applicato ad un agente, al task listener deve essere aggiunto il metodo desiderato.

4.4.3.1 Vincoli e relazioni

Un vincolo può essere considerato come una regola che deve essere soddisfatta affinché il pattern possa essere introdotto nel sistema ed è composto da due parti: obiettivo e contenuto.

L'obiettivo specifica quale componente del progetto deve subire la modifica, mentre il contenuto rappresenta la modifica da applicare. Quest'ultimo può essere espresso come un aggregazione di costruttori, metodi e attributi.

Il valore che può assumere l'obiettivo invece varia tra ParentTask, IdleTask e OwnerAgent.

Un vincolo sul ParentTask avrà effetto sul task che è legato da una relazione di padre-figlio con il pattern. Questo genere di vincolo può essere usato comunemente, ad esempio in FIPA-OS per aggiungere il metodo *done* nel task padre per catturare l'evento di terminazione del task figlio.

Un vincolo sull'IdleTask avrà effetto sul task impostato per ricevere le comunicazioni in ingresso per l'agente. Questo genere di vincolo può essere usato principalmente per inserire i metodi handle per le comunicazioni gestite dai vari task.

Un vincolo sull'OwnerAgent avrà effetto sull'agente di cui il task è figlio. Può essere utilizzato per aggiungere attributi e metodi all'agente utili per l'attività del task.

Un vincolo può possedere un ulteriore attributo opzionale che specifica per quale piattaforma è stato introdotto. Quando un vincolo possiede questa specifica allorà avrà effetto solo nel caso in cui la piattaforma usata per la generazione del codice è quella specificata. Verrà ignorato in tutti gli altri casi.

4.4.3.1.1 Clausola Constraints

Per esprimere una collezione di vincoli da applicare insieme ad un pattern si utilizza la clausola Constraints.

```
<!ELEMENT Constraint (Import*, Idle_task?, Parent_task?, Owner_agent?)>
```

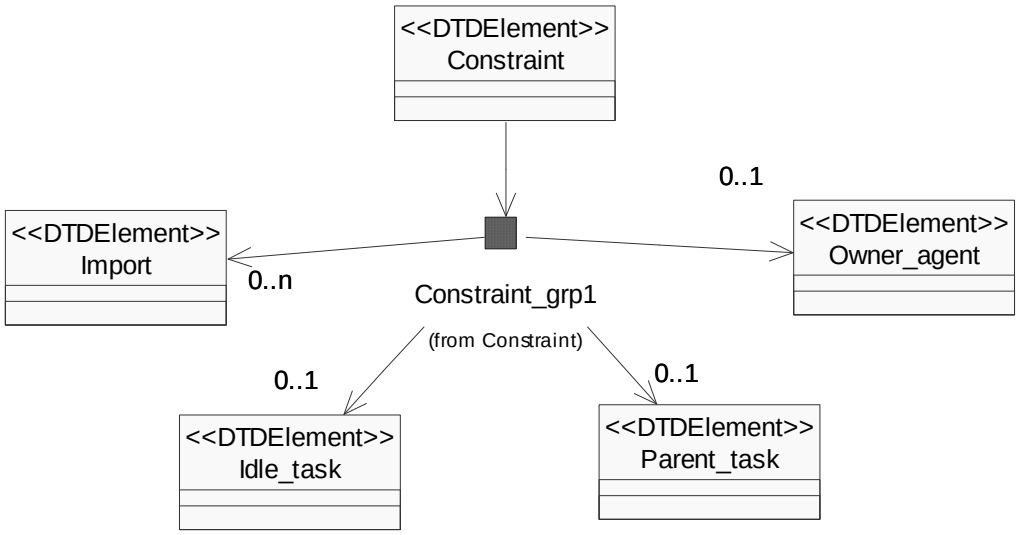


Figura 4-31 Diagramma DTD della clausola Constraint

Sotto-elementi		
Import	0..n	Questo sotto elemento rappresenta un vincolo di relazione con una libreria esterna che deve essere importata nel codice.
Idle_task	0..1	L'Idle Task rappresenta un task particolare che si occupa di gestire la ricezione degli eventi da parte della piattaforma. Un vincolo sull'Idle Task indica che l'Idle Task deve contenere attributi o metodi secondo una sintassi specificata.

Parent_task	0..1	Dato un task, il suo parent task è il task che avvia l'esecuzione dello stesso. Un vincolo sul Parent Task indica gli attributi ed i metodi che tale task deve contenere.
Owner_agent	0..1	Un vincolo sull'Owner Agent indica attributi e metodi che l'agente, al quale viene applicato il pattern, deve contenere.

A sua volta un vincolo di IdleTask, Parent Task o Owner Agent è espresso da un elemento la cui sintassi è:

```
<!ELEMENT Parent_task (Attribute*, Method*)>
<!ELEMENT Idle_task (Attribute*, Method*, Task*, Class*, Interface*)>
<!ELEMENT Owner_agent (Attribute*, Method*, Task*, Class*, Interface*)>
```

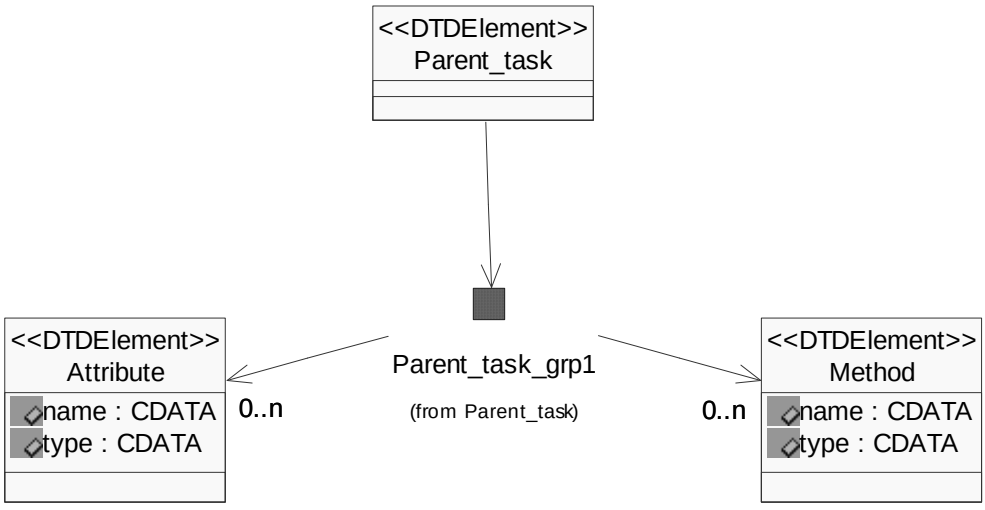


Figura 4-32 Diagramma DTD dell'elemento Parent-task

Sotto-elementi		
Attribute	0..n	Indica un elenco di attributi che il task parent deve necessariamente possedere per il corretto funzionamento del pattern.
Method	0..n	Indica un elenco di metodi che il task parent deve necessariamente possedere per il corretto funzionamento del pattern.

4.4.3.1.2 Relazioni

Nell'applicazione dei vincoli, l'IdleTask è riconoscibile in quanto è un task particolare; l'Owner Agent è l'agente che contiene il pattern in questione. Non esiste nessun elemento che fornisca il legame parent-child tra due task. Questa informazione, deve essere estratta dal diagramma MABD del progetto relativo all'agente in esame; per memorizzare questa informazione nella struttura dell'agente deve essere introdotto un ulteriore elemento: la clausola <Parent > sotto-elemento aggiuntivo di <Task>.

```
<!ELEMENT Parent (#PCDATA)>
```

Poiché i vincoli di un pattern non vengono memorizzati all'interno dell'agente in cui il pattern è stato applicato allora è necessario ricordare per ogni agente o task quali pattern sono stati applicati. In fase di elaborazione o di generazione del codice in questo modo è possibile rintracciare tali vincoli ed effettuare le modifiche opportune. Per memorizzare l'elenco dei pattern applicati si usa la clausola <Pattern> così definita:

```
<!ELEMENT Pattern (#PCDATA)>
```

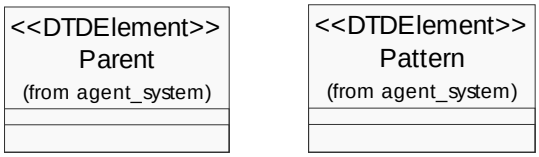


Figura 4-33 Diagramma DTD degli elementi Parent e Pattern

4.4.3.2 Entità di riferimento

Quando un agente viene memorizzato su file, i campi <Code> dei metodi di agente e task contengono il testo relativo al corpo delle funzioni.

Invece quando un pattern viene prelevato dal repository i campi <Code> non contengono direttamente il codice bensì un riferimento esterno al luogo nel quale trovare tale codice.

Il luogo dove solitamente viene memorizzato il codice interno a metodi e costruttori è la tabella ActionPattern del database.

Per poter esprimere un legame univoco tra un corpo di metodo presente in un agente o in un task e un action pattern presente nel database si utilizza una sintassi ben definita.

nome_metodo . [nome_task] @ nome_pattern

L'intera stringa rappresenta una chiave univoca per entrare nella tabella degli action pattern. Questa struttura verrà descritta meglio nel capitolo 5, *Repository di pattern.*

4.4.3.2.1 DTD completo

Di seguito si riporta per intero il DTD relativo al linguaggio sviluppato.

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT Agent_description (Package?, Import*, Global_variable*,
Agent*, Task*, Class*, Interface*)>
<!ATTLIST Agent_description
```

```
        agent_system CDATA #REQUIRED
    >
    <!ELEMENT Package (#PCDATA)>
    <!ELEMENT Import (#PCDATA)>
    <!ELEMENT Global_variable (Final?, Static?, Native?, Volatile?,
Visibility, Value?)>
    <!ATTLIST Global_variable
        name CDATA #REQUIRED
        type CDATA #REQUIRED
    >
    <!ELEMENT Agent (Description?, Abstract?, Final?, Visibility,
Extends?, Implements*, Attribute*, Constructor*, Method*, Task*,
Class*)>
    <!ATTLIST Agent
        name CDATA #REQUIRED
    >
    <!ELEMENT Task (Description?, Abstract?, Final?, Visibility,
Extends?, Implements*, Attribute*, Constructor*, Method*,
Constraint*)>
    <!ATTLIST Task
        name CDATA #REQUIRED
    >
    <!ELEMENT Class (Description?, Abstract?, Final?, Visibility,
Extends?, Implements*, Attribute*, Constructor*, Method*, Class*)>
    <!ATTLIST Class
        name CDATA #REQUIRED
    >
    <!ELEMENT Interface (Description?, Visibility, Extends*,
Attribute*, Constructor*, Method*)>
    <!ATTLIST Interface
        name CDATA #REQUIRED
```

```
    >
    <!ELEMENT Attribute (Visibility, Static?, Final?, Native?,
Volatile?, Value?)>
    <!ATTLIST Attribute
        name CDATA #REQUIRED
        type CDATA #REQUIRED
    >
    <!ELEMENT Method (Visibility, Abstract?, Static?, Final?, Native?,
Synchronized?, Argoment*, Throws*, Code?)>
    <!ATTLIST Method
        name CDATA #REQUIRED
        type CDATA #REQUIRED
    >
    <!ELEMENT Constructor (Visibility, Abstract?, Static?, Final?,
Native?, Synchronized?, Argoment*, Throws*, Code?)>
    <!ATTLIST Constructor
        name CDATA #REQUIRED
    >
    <!ELEMENT Argoment EMPTY>
    <!ATTLIST Argoment
        name CDATA #REQUIRED
        type CDATA #REQUIRED
    >
    <!ELEMENT Description (#PCDATA)>
    <!ELEMENT Abstract EMPTY>
    <!ELEMENT Final EMPTY>
    <!ELEMENT Static EMPTY>
    <!ELEMENT Native EMPTY>
    <!ELEMENT Volatile EMPTY>
    <!ELEMENT Synchronized EMPTY>
```

```
    <!ELEMENT Visibility (#PCDATA)>
    <!ELEMENT Extends (#PCDATA)>
    <!ELEMENT Implements (#PCDATA)>
    <!ELEMENT Throws (#PCDATA)>
    <!ELEMENT Value (#PCDATA)>
    <!ELEMENT Code (#PCDATA)>

    <!ELEMENT Constraint (Import*, Idle_task?, Parent_task?,
Owner_agent?)>
    <!ELEMENT Parent_task (Attribute*, Method*)>
    <!ELEMENT Idle_task (Attribute*, Method*, Task*, Class*,
Interface*)>
    <!ELEMENT Owner_agent (Attribute*, Method*, Task*, Class*,
Interface*)>
```

5. Ideare Metodologie di Sviluppo del Software

5.1 Introduzione

In questo capitolo si fornirà una introduzione ai concetti che stanno alla base dei processi di sviluppo del software allo scopo di porre i presupposti per la identificazione della metodologia che verrà sviluppata nel seguito.

Il capitolo si articola in due parti principali: nella prima (paragrafo “Metodologie di Sviluppo del Software”) si introdurranno i concetti relativi ai processi di sviluppo del software ed un approccio noto come method engineering per la composizione di nuovi processi sulla base del riuso di parti di processi noti.

Nella seconda parte del capitolo (paragrafo “Agile software development”) si presenteranno le caratteristiche delle metodologie cosiddette agili alle quali ci si è ispirati per la creazione del nuovo processo di progettazione detto “Passi Agile”.

5.2 Metodologie di Sviluppo del Software

5.2.1 Introduzione

Una metodologia di sviluppo software può essere considerata come un modo per comporre certi tipi di documento, per esempio specifiche di requisiti o di progetto, codice e manuali usando un linguaggio naturale od una notazione particolare che rispetti determinate regole semantiche e sintattiche.

Un metodologia di sviluppo software fa sempre riferimento ad un particolare dominio applicativo dal quale bisogna astrarre e modellare le informazioni necessarie per la costruzione dei suddetti documenti.

Un metodo, o una metodologia, aiuta nella modellazione di sistemi ed ambienti indicando le regole da seguire e le attività da compiere, specificandone modo e sequenza, per produrre un software.

Modellazione ed astrazione diventano assolutamente necessari adesso che i sistemi software stanno diventando sempre più complessi e costosi, così nel corso degli anni sono stati sviluppati molti metodi con i relativi tool di supporto.

Man mano che i sistemi software diventano sempre più complessi diventa sempre più oneroso, dal punto di vista tempi e costi, applicare metodi già esistenti; questo perché ogni metodo contiene parti che non si possono adattare a tutte le specifiche situazioni di progetto.

Alla domanda, allora, "esiste un metodo universalmente adatto ad ogni progetto di sviluppo software ?" La risposta è rigorosamente NO.

Un modo per affrontare e risolvere questo problema è, come viene fatto ad oggi nelle aziende e nelle organizzazioni, quello di costruire un proprio metodo od adattarne uno per la soluzione dello specifico problema che si sta affrontando e dello specifico ambiente di sviluppo che si ha a disposizione.

A questo proposito la Method Engineering è quella disciplina che mette a disposizione le tecniche per la costruzione di uno specifico metodo.

Da Brinkkemper: " *La Method Engineering è la disciplina per progettare, costruire ed adattare metodi, tecniche e tool per la costruzione di un information system*".

Uno dei metodi più semplici per costruire un metodo , secondo il paradigma della method engineering, è quello fondato sul riuso di parti di metodi già esistenti

Nei successivi paragrafi si illustreranno metodi e tecniche della ingegneria del software e della method engineering ponendo l'accento sui problemi e le motivazione che hanno portato allo sviluppo della seconda.

In quanto segue si farà spesso riferimento allo sviluppo software o allo sviluppo di *information system*, intendendo con entrambi lo stesso concetto.

5.2.2 Software Engineering

Il termine Software Engineering è stato coniato nel 1968 in risposta alla carenza di tecniche per lo sviluppo di software di alta qualità. In quegli anni si dava molta più importanza all' hardware che al software, tutti gli sforzi erano tesi allo sviluppo del primo con conseguente quasi totale assenza di prospettive ingegneristiche per lo sviluppo e la produzione del secondo; le tecniche erano assolutamente empiriche ed artigianali.

A metà egli anni ottanta le potenzialità dell'hardware superavano di gran lunga la capacità di costruire software adeguato, con tempi e costi ragionevoli. La tendenza si è allora invertita con la consapevolezza che a parità di hardware è il software a fare la differenza.

Un software è un insieme di programmi, istruzioni e moduli, che fornisce le prestazioni e le funzioni desiderate, strutture dati che consentono ai programmi di manipolare le informazioni, documenti e procedure operative mediante i quali i computer possono essere resi operativi.

Caratteristica fondamentale di un software è che esso non è un prodotto fisico, non si fabbrica ma si sviluppa e si ingegnerizza, non si logora e non si guasta ma si deteriora. Un guasto di un software è causato sempre da errori commessi in fase di progettazione e di sviluppo, per questo motivo tutta la fase di produzione di un software va fatta secondo canoni avulsi dalla produzione di un prodotto commerciale comune.

Un software è un sistema molto complesso, formato da diversi componenti ed orientato al raggiungimento di obiettivi spesso in conflitto e soprattutto orientato al soddisfacimento dei bisogni dell'utente/cliente.

Alla luce di tutto questo, gli sviluppatori software, spesso, non erano capaci di allocare le risorse necessarie per ottenere un insieme di obiettivi prefissati mantenendo tempi e costi in linea con i bisogni del cliente.

L'ingegneria del software è una disciplina tecnologica e manageriale i cui scopi sono la produzione ed il mantenimento sistematico di prodotti software che devono essere sviluppati e modificati entro certi limiti stimati di tempo e costi. Essa consta di quattro attività fondamentali: modellazione, risoluzione di problemi, acquisizione di conoscenza e gestione delle scelte progettuali.

5.2.2.1 Lo sviluppo di un software

Un metodo di sviluppo software include tre fasi fondamentali : definizione, sviluppo e manutenzione.

La definizione comprende l'individuare "il che cosa", cioè raccolta dei requisiti ed analisi del sistema corrispondenti alla formulazione ed all'analisi del problema; durante questa fase gli sviluppatori insieme a clienti ed utenti elaborano un modello problema.

Durante la fase di sviluppo tutti i requisiti vengono tradotti in rappresentazioni formali, diagrammi tabelle e testi; viene suddiviso il problema in sottoproblemi per i quali si analizzano le strategie di progetto che si concretizzano poi nella fase successiva nella selezione delle soluzioni adatte ad ogni singolo pezzo, nella codifica ed infine nel *testing*.

La fase di manutenzione comprende correzione degli errori, adattamenti per modifiche hardware e miglioramenti nel tempo.

Lo sviluppo di un software è regolato da quelli che si chiamano paradigmi dell'ingegneria del software, tra i quali due sono stati presi in considerazione per lo sviluppo di questo lavoro: la prototipizzazione e le tecniche di quarta generazione.

Il prototipo è un modello del software, che dia un idea del prodotto finale, sotto forma testuale o di semi-lavorato funzionante.

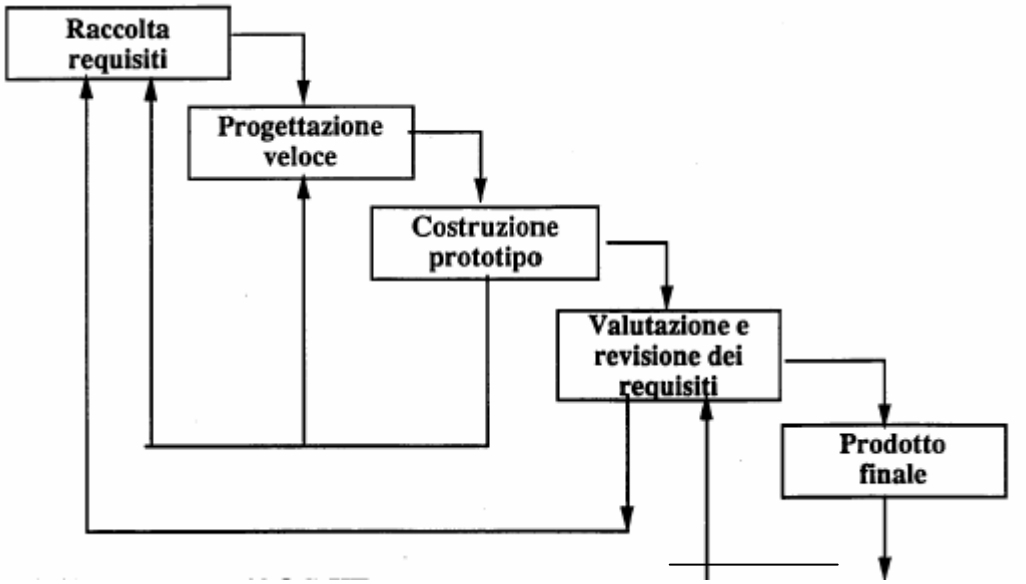


Figure 5-1 Processo costruzione software

Come si vede dalla figura il processo si evolve dalla fase di raccolta dei requisiti al prodotto finale attraverso dei passi intermedi sui quali operando delle iterazioni, se necessario, si produce un prodotto finale rispondente alle specifiche iniziali.

Le tecniche di quarta generazione consistono nell'utilizzo di quelli che si chiamano linguaggi di quarta generazione che sono strumenti software capaci di descrivere le specifiche

di progetto con linguaggi simili a quello naturale e di tradurle automaticamente in codice eseguibile.

Come si vede dalla figura il processo è simile al precedente

Figure 5-2 Processo costruzione software(II)

5.2.2.2 Modellazione di processi

Lo sviluppo di un software porta, in genere, l'ingegnere del software ad interfacciarsi con ambienti molto complessi dei quali egli deve descrivere e comprendere ogni singolo aspetto. Da qui la necessità di riuscire ad astrarsi e di ridurre la complessità del sistema attraverso modelli.

Costruire un modello porta a prendere in considerazione due entità, il mondo reale, costituito da fenomeni, ed il dominio del problema formato da un insieme di concetti interdipendenti che descrivono il problema sotto esame. Un modello è, allora, una rappresentazione del sistema preso in considerazione che permette di ridurre i fenomeni in concetti così da poter trattare sistemi comunque complessi, troppo grandi o troppo costosi.

Gli ingegneri del software non devono conoscere ogni singolo dettaglio del domino del loro problema ma solo i concetti fondamentali che permettano loro di costruirne un modello adeguato per valutare tutte le possibili soluzioni.

È ormai riconosciuto da molto tempo che seguire un processo, o una metodologia, per lo sviluppo software migliora notevolmente il software stesso; la modellazione dei processi offre un supporto all'identificazione, al miglioramento ed alla rappresentazione del software. Una metodologia di sviluppo software fa riferimento sia ai prodotti software che ai processi software, gli ambienti di supporto per queste metodologie richiedono modelli di metodologie i quali trattano allo stesso modo entrambi gli aspetti [28].

Ci sono allora due approcci alla modellazione di processi, uno basato sulle attività ed un altro basato sui processi; la strategia adottata influisce notevolmente sui modelli del processo in termini di granularità, maneggevolezza, promulgabilità e prescrivibilità.

La creazione di un modello di una metodologia permette di catturarne i requisiti per costruirne l'ambiente di supporto assegnando ad ogni concetto significativo della metodologia un elemento del relativo modello

La ricerca di una soluzione adeguata al problema è una attività di pura ingegnera, si deve: formulare ed analizzare il problema, cercare e decidere per una soluzione appropriata selezionandola tra diverse alternative e specificarla.

Negli ultimi anni sono stati sviluppati ed usati moltissime metodologie, tecniche, linguaggi e tool usati per sviluppare sistemi; molte di queste tecniche e di questi linguaggi vengono formalizzati in base ad un approccio orientato agli oggetti che dà la possibilità di rappresentare il sistema si adal punto di vista strutturale che comportamentale.

5.2.2.3 I tool di supporto

L'applicazione di metodi per lo sviluppo di sistemi informativi non ha senso senza adeguati tool di supporto; a causa dell'aumento sempre crescente della complessità dei progetti, i metodi di sviluppo ed i tool di supporto finiscono per essere uno dei fattori chiave più significativi , del progetto di sviluppo, per raggiungere un grosso successo .

Un tool è una applicazione per computer che supporta la tecnica di modellazione adottata. Avvalersi di un tool di supporto include la possibilità o la capacità di astrarre il sistema sotto esame in modelli, controllare che questi siano consistenti, convertire i risultati da una forma di modello ad un'altra e fornire specifiche (spiegazioni) per la revisione.

I tool contengono parte dello schema concettuale, supportano la modellazione con delle notazioni. L'uso di un tool non fornisce soltanto una descrizione ma assicura anche la possibilità di imparare l'uso del metodo tramite un interfaccia utente che mostra come il tool va usato.

5.2.2.4 Linguaggi di modellazione

L'applicazione di una metodologia al progetto di un sistema informativo impone l' identificazione di tutti i suoi elementi e la loro descrizione in modo preciso, non ambiguo e che possa essere compreso dagli utenti che partecipano alla progettazione ed utilizzano il sistema.

Tra le tante metodologie che sono state proposte negli ultimi, solo alcune hanno avuto grande diffusione. Si citano alcune tra le più importanti:

- Object Modeling Technique (OMT) *(James Rumbaugh)*
- Object Oriented Software Engineering (OOSE) (*Ivan Jacobso)*
- Object-Oriented Analysis and Design with Applications (OOADA) (*Grady Booch*)

Ognuna di queste possiede un proprio linguaggio di modellazione che, però, analizza solo alcuni aspetti del processo di sviluppo, diventando, così, adatto solo ad alcune applicazioni; per tale motivo si è imposto lo standard Unified Modeling Language (UML) che nasce come successore di questi linguaggi e pone fine alle differenze tra le varie notazioni mediante un formalismo più versatile. UML è attualmente lo standard per la modellazione orientata agli oggetti.

5.2.2.5 UML (Unified Modeling Language)

UML è un linguaggio per specificare, visualizzare, e documentare modelli dei sistemi software e non, sia dal punto di vista strutturale che progettuale; esso si adatta principalmente ad ambienti di progettazione e di programmazione orientati agli oggetti ma può andare bene anche per altri contesti applicativi.

Come altri linguaggi di specifica UML include :

- Elementi di modellazione – semantica e concetti fondamentali per la modellazione.
- Notazione – rappresentazione grafica degli elementi del modello
- Linee guida – raccomandazioni per l'utilizzo

Una delle caratteristiche fondamentali di UML è l'indipendenza dalla metodologia usata per l'analisi ed il progetto del proprio sistema; qualunque essa sia UML fornisce il supporto per produrne i risultati.

Inoltre con l'ausilio di UML, soprattutto in progetti di grandi dimensioni, si ha la possibilità di focalizzare l'attenzione su uno o più aspetti specifici del problema rappresentandone i diversi aspetti anche tramite diagrammi diversi.

UML definisce dodici tipi di diagrammi, suddivisi in tre categorie:

- Diagrammi strutturali – rappresentano la struttura statica dell'applicazione, essi sono *Class Diagram* (diagramma delle classi), *Object Diagram* (Diagramma degli oggetti), *Component Diagram* (diagramma dei componenti) e *Deployment Diagram* (diagramma di dislocazione dei componenti).
- Diagrammi comportamentali – rappresentano i differenti aspetti del comportamento dinamico del sistema, essi sono *Use Case Diagram* (diagramma dei casi d'uso usato da alcune metodologia durante la fase di raccolta dei requisiti), *Sequence Diagram* (diagramma di sequena), *Activity Diagram* (diagramma delle attività), *Collaboration Diagram* (diagramma di collaborazione) e *Statechart Diagram* (diagrammi di stato)
- Diagrammi di gestione dei modelli – per organizzare e gestire i moduli dell'applicazione, includono Package, Subsystems e Model.

Scopo fondamentale di tutti questi modelli è di permettere l'astrazione, cioè la rappresentazione di un problema complesso in termini più semplici.

5.2.3 Method Engineering

La Method Engineering nasce dalla necessità di avere una struttura di ricerca per la costruzione di metodi e dei relativi tools di supporto per lo sviluppo di sistemi informativi (*information system development*) [29].

La complessità dei progetti per sviluppare *information systems* sta aumentando sempre più con il passare del tempo, tanto che i metodi ed i tool di supporto finiscono per essere uno dei fattori chiave più significativi del progetto di sviluppo.

Negli anni passati sono stati divulgati molti manuali e testi sui metodi, i loro tool di supporto hanno avuto un grosso sviluppo commerciale, e sono stati largamente utilizzati per le attività di sviluppo.

Un metodo fornisce agli sviluppatori le direttive per astrarre e modellare gli aspetti essenziali di sistemi ed ambienti e per scrivere tutti quei documenti che compongono il progetto, e che guidano nelle varie fasi di sviluppo dello stesso, come per esempio specifiche di requisiti e di progetto, codice sorgente, manuali e così via.

Si fa notare che nell'ambito dello sviluppo di sistemi informativi, si fa un uso promiscuo ed errato delle parole metodo e metodologia; l'origine greca della parola metodologia ci dice che questa significa studio di metodi, inoltre il Dizionario di Oxford definisce la metodologia lo studio di metodi sistematici di ricerca scientifica, invece viene spesso usata con il significato di collezione di metodi atti a risolvere una determinata classe di problemi se non addirittura come sinonimo stesso di metodo.

Per evitare confusione si è preso come riferimento, in questo lavoro, quanto detto da Brinkkemper : " *il cattivo uso del termine metodologia al posto di metodo dà un segno dell'immaturità del nostro campo e dovrebbe essere abbandonato* "; egli inoltre fornisce la definizione di metodo, che si prende come riferimento, considerandolo anche sinonimo di metodologia : " *un metodo è un modo per costruire progetti di sviluppo di sistemi, basati su uno specifico modo di pensare, consistente di direttive e regole, strutturate in modo semantico in attività di sviluppo con i corrispondenti prodotti* ".

Ma man mano che il sistema da sviluppare diventa grande e complesso vengono spesi tempo, denaro e risorse umane nell'apprendimento e nell'applicazione di un metodo già

esistente. La difficoltà effettiva degli sviluppatori e dei progettisti di imparare ad usare completamente un metodo e, soprattutto, di adattarlo al proprio ambiente di sviluppo, al proprio progetto, sta nel fatto che non tutti i metodi sono adatti ad uno specifico problema, alcuni metodi lavorano bene in uno specifico dominio altri no; questa situazione è particolarmente stressante quando i sistemi che devono essere modellati sono intrinsecamente dinamici come per esempio nel caso di sistemi real time o reattivi o quando si devono cambiare gli attuali sistemi di lavoro per avvantaggiarsi dei cambiamenti nella tecnologia.

Si pensa sia impossibile avere un metodo standardizzato ed universale ed allora per migliorare il risultato dell'applicazione di un metodo ad ogni progetto reale sa ha bisogno di adattare i metodi esistenti o di costruirne altri , nuovi, che siano adatti al progetto in questione.

La tendenza attuale da parte di progettisti e di sviluppatori è quella di andare incontro ai differenti tipi di requisiti di un progetto di sviluppo software provando a modificare metodi già esistenti o a svilupparne di nuovi, i cosiddetti *method in house* (metodi casalinghi) adattabili ai propri domini del problema ed ai propri ambienti di sviluppo, così da avere progetti di sviluppo di sistemi informativi prodotti non con un set di metodi standardizzati ma con metodi i cui requisiti variano.

La *Method Engineering*, ed in particolare la *Situational Method Engineering*, è la disciplina per costruire specifici progetti di metodi, chiamati *situational methods* da parti di metodi già esistenti, chiamati *method fragments*, tramite delle tecniche di assemblaggio chiamate *method assembly* [30][31].

La *method engineering* è una disciplina dove il metodo stesso è il target del progetto, essa ricerca ed esplora tecniche di ingegneria per costruire metodi.

Secondo Brikkemper *la method engineering è una disciplina per progettare, costruire ed adattare metodi , tecniche e tool per lo sviluppo di information systems.*

Uno dei modi migliori per costruire un metodo è un approccio basato sul riuso; elementi costitutivi significativi di una metodologia o parti di metodi, chiamati *method fragments*, vengono immagazzinati in un database chiamato *method base*.

Particolari ingegneri chiamati *method engineers* recuperano i *method fragments* appropriati per il loro scopo , dal *method base*, li adattano e li integrano tra loro in un nuovo metodo.

Gli ingegneri del metodo devono avere adeguata conoscenza sui metodi esistenti e capacità tali da consentire loro la costruzione del nuovo metodo utilizzando, come detto prima, le tecniche del riuso e della integrazione dei frammenti.

Lo scopo fondamentale della Method Engineering è di migliorare [32] i metodi di sviluppo di sistemi informativi, a tal scopo essa può essere vista come un processo di apprendimento nel quale un individuo o una organizzazione crea nuova conoscenza circa i metodi e come applicarli. L'apprendimento risulta importane alla luce di quanto detto prima circa la difficoltà ad accettare ed usare metodi già preconfezionati.

Questo è un punto cruciale in quanto non è facile costruire e mantenere conoscenza ed esperienza su un metodo se non lo si usa costantemente , e questo spiega, in parte, perché le organizzazioni non usano e a volte addirittura abbandonano i metodi già esistenti; l'introduzione di un metodo è un investimento a lungo termine che dà i suoi frutti dopo un periodo relativamente lungo.

In questa ottica invece di considerare i metodi rigidi e standardizzati, si suppone che i loro requisiti varino, infatti un metodo dovrebbe esser costruito in funzione delle particolari situazioni e contingenze dell'ambiente in cui ci si trova ad operare.

Per migliorare i metodi di sviluppo di *information system* ed aumentare la loro flessibilità devono essere sviluppate delle direttive (le cosiddette *guidelines*) che specifichino la conoscenza relativa a quali metodi devono essere descritti, analizzati e mantenuti per il progetto e come questi devono essere adattati in tool di supporto.

Lo sviluppo di metodo locale è molto importante; nelle organizzazioni diventa, allora, essenziale la conoscenza sullo sviluppo e l'uso del metodo stesso; tale conoscenza è fondamentale soprattutto per quello che riguarda il raffinamento o il miglioramento del metodo creato che non rimane mai rigido ma cambia, method engineering incrementale.

5.2.3.1 Processo di sviluppo di un method in house

Si diceva che la complessità e la grandezza sempre crescente dei prodotti software hanno portato ad un scarsissimo uso dei metodi di *information system* già esistenti .

In genere all'interno di una organizzazione, gli sviluppatori, basandosi su propri concetti, opinioni, linguaggi di modellazione e procedure si creano una certa esperienza e conoscenza sui metodi di sviluppo di *information system*.

In tal modo essi possono creare il proprio *method in house* o modificare o estendere quelli esistenti, principalmente valutando i metodi a disposizione e conducendo studi sull'uso degli stessi.

Il risultato è una quantità eccessiva di metodi e di tools di supporto che costituiscono soltanto dei "dialetti" delle singole aziende; nessun metodo, e soprattutto l'esperienza degli sviluppatori e il *method use*, è adatto ad un'altra azienda, un altro progetto o una situazione specifica.

Un *local method* è strettamente legato a tecniche e concetti di modellazione e vincoli, propri dell'azienda nella quale sono stati sviluppati e degli ambienti nei quali questa è abituata ad operare.

Ormai questa tendenza è sempre più diffusa ma vi è anche una scarsità di principi su come selezionare costruire e adattare i metodi esistenti per i propri scopi e le proprie necessità.

Di seguito i passi che uno sviluppatore (una azienda) deve svolgere per creare il proprio method in house

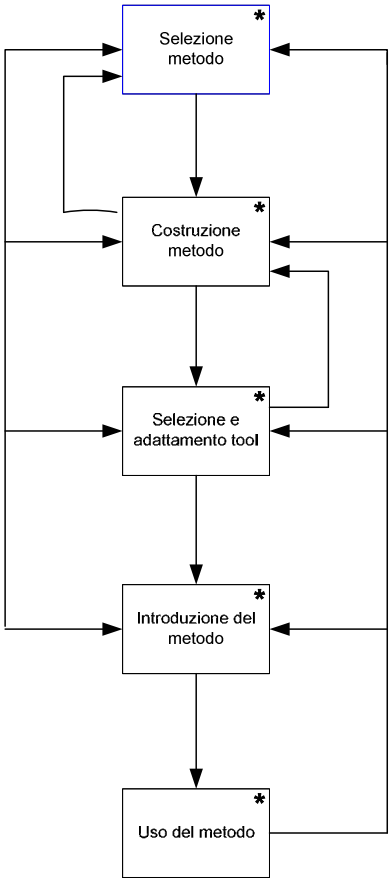


Figure 5-3 Processo di sviluppo di un method in house

Ovviamente alcuni passi possono essere omessi o saltati, mentre i loop indicati (method refinements) sono necessari per rifinire il metodo in qualunque momento e durante qualunque fase, qualora si presentassero nuovi requisiti, nuova conoscenza o l'esperienza condotta utilizzando il metodo sviluppato portasse a delle modifiche nello stesso.

Sarebbe bene che i rifinimenti siano costantemente documentati per rendere disponibile la conoscenza sul metodo e le sue variazioni e quindi agevolare i futuri utenti o gli sviluppatori nel miglioramento del metodo.

E' fondamentale allora la capacità di una organizzazione di imparare dall'uso del proprio metodo, di creare conoscenza circa l'applicabilità del metodo e di usare questa conoscenza per il rifinimento del metodo stesso.

Lo sforzo principale diventa: come definire, rifinire, rafforzare ed applicare la conoscenza esplicita maturata sullo sviluppo di un sistema di informazione.

Ci sono diversi modi in cui i metodi possono essere selezionati e integrati per lo sviluppo del *local method* .

Tolvanen [32] ha condotto un'analisi approfondita sui metodi, la loro selezione ed il loro sviluppo e ha stabilito tre strategie per lo sviluppo di local method:

- text_book,
- contingency
- method enginerring

nel primo caso viene scelto ed eventualmente modificato un intero metodo, nel secondo ci si basa sulla teoria della contingenza secondo la quale non esiste un metodo universale adatto ad ogni situazione ed infine nella *method engineering* la selezione è fatta scegliendo parti di metodi, i cosiddetti frammenti, ed associandoli in uno nuovo.

Le tre strategie non sono mutuamente esclusive, ma, in questo lavoro, si è scelto di usare principalmente un approccio basato sulla *method engineering* .

5.2.3.2 Le tecniche della method engineering

Una tecnica è un insieme di passi attraverso il quale gli sviluppatori percepiscono, definiscono o comunicano gli aspetti essenziali del dominio del problema in questione. Le strutture concettuali e le notazioni per rappresentare un sistema informativo vengono chiamate "un modo di modellare".

Una struttura concettuale è costituita da un insieme di concetti, relazioni e vincoli tra loro, alcuni concetti vengono applicati direttamente nelle notazioni , per esempio la classe nella notazione UML. Un concetto è strettamente correlato all'ambiente cui si rivolge *l'information system.*

Ogni concetto della tecnica di modellazione ha solo una rappresentazione notazionale o simbolo. La disciplina della method engineering consiste di quattro tecniche essenziali:

- Tecniche di Meta-Modellazione
- Tecniche per l'adattamento dei metodi e la loro integrazione

- Relazioni tra gli attuali progetti di sviluppo ed i metodi
- Generazione di CASE tools

La tecnica della meta-modellazione consiste nella creazione di un meta-modello, cioè un modello ed una descrizione di un metodo necessari per rappresentare e modellare frammenti di metodo o metodi stessi in modo tale da potere essere memorizzati nel database; sono necessari allora tecniche e linguaggi.

Scopo di un metodo è indicare quali attività un progettista deve svolgere, ed in che ordine, per potere sviluppare e produrre tutti quei documenti facenti parte del processo di sviluppo.

Per sviluppare un meta-modello bisogna tenere in considerazione che un frammento di metodo, o comunque un metodo, è formato da due parti fondamentali : prodotto e processo, quindi la struttura del prodotto e le procedure per generarlo.

La meta-modellazione è sempre più usata come un software e un tool di *method engineering*. E' provato in letteratura che l'uso della meta-modellazione effettivamente migliora l'usabilità, la comprensibilità e la leggibilità durante gli studi (analisi, progetto, comparazione ed adattamento) di linguaggi e tecniche. Un buon meta-modello di un metodo è essenziale per ridurne la complessità e facilitarne la comprensione e l'uso.

Una volta memorizzati nel method base, i method fragments vengono riutilizzati mediante ri-assemblaggio ed adattamento , da un method engineer, per dare vita ad un nuovo metodo.

A questo scopo sono necessarie le cosiddette tecniche di method integration, molto spesso suportate da CAME tools.

Tali tecniche sono di due tipi: quelle orientate all'integrazione dei prodotti e quelle orientate all'integrazione dei processi, nella prima si integrano diversi tipi di prodotti nella seconda invece diverse parti dello stesso metodo o di metodi diversi.

Un CAME (Computer Aided Method Engineering) è il tool di supporto per la costruzione e definizione dei method fragments e per la loro integrazione.

Un CAME può anche avere un linguaggio di modellazione che serva sia per descrivere i metodi che per guidare nella integrazione degli stessi.

Un tool è una applicazione che supporta la tecnica di modellazione.

Avvalersi di un tool di supporto include la possibilità o la capacità di astrarre l'ambiente del progetto in modelli, controllare che questi siano consistenti, convertire i risultati da una forma di modello ad un'altra e fornire specifiche o spiegazioni per la revisione.

Come precedentemente detto, i tool contengono parte dello schema concettuale, supportano la modellazione con delle notazioni, inoltre l'uso di un tool assicura che venga applicata tutta la conoscenza acquisita durante l'uso di un metodo, non è soltanto una descrizione notazionale.

5.2.3.3 Relazione tra method engineering e sviluppo software

Un ingegnere del metodo può considerarsi colui che produce sistemi informativi per lo sviluppo di sistemi informativi; questo gioco di parole serve per spiegare che la *method engineering* si pone ad un livello di astrazione superiore rispetto alla *software engineering*.

La *software engineering* si occupa di sviluppare e implementare delle specifiche sul sistema in esame, mentre *la method enginnering*, come detto prima si occupa di creare o modificare metodi o parti di metodi per migliorare o adattare quelli già esistenti ad una specifica situazione.

Si guardino le due figure seguenti, la prima illustra un processo di sviluppo di un sistema informativo

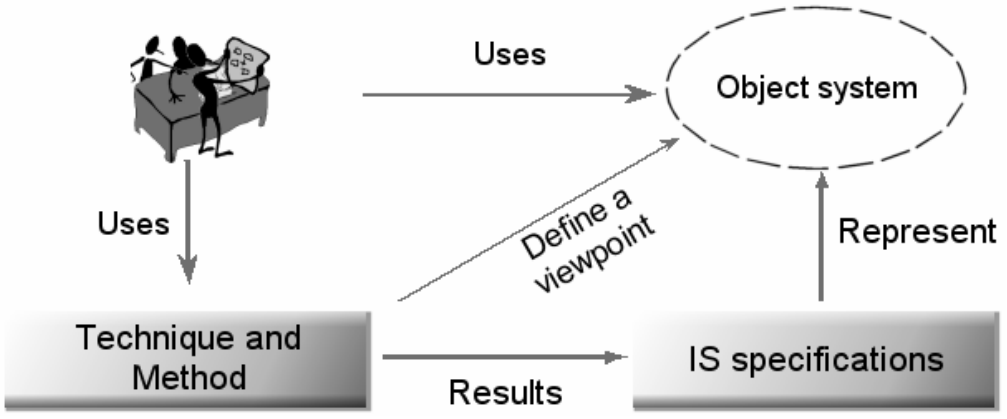


Figure 5-4 Processo di sviluppo software

la seconda illustra un processo complessivo e comprensivo del lavoro degli ingegneri del metodo.

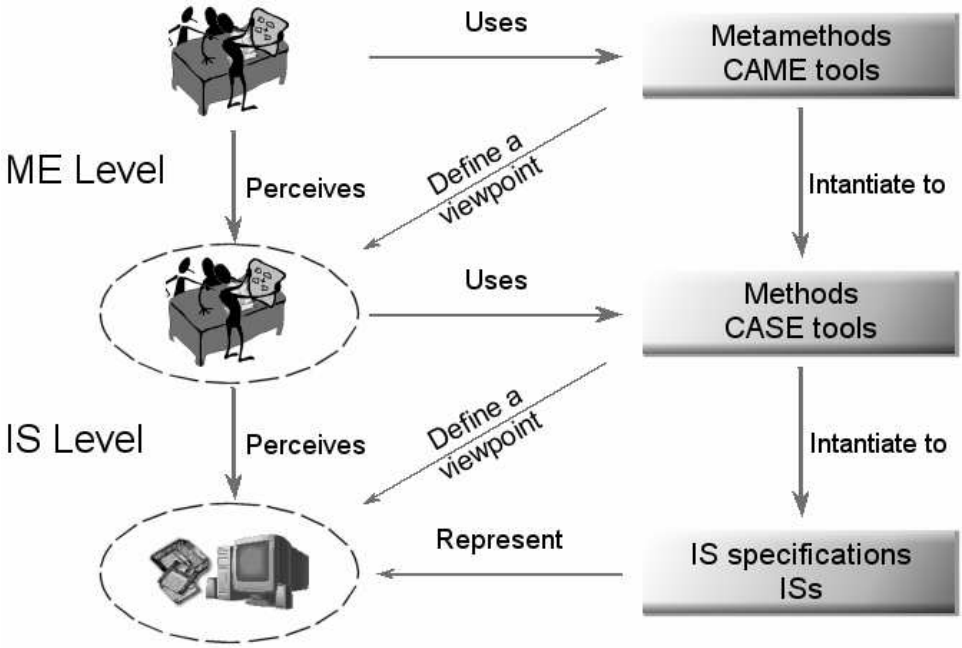


Figure 5-5 Method engineering e sviluppo software

Al livello della *method engineering* gli ingegneri del metodo, guardando allo stato corrente del processo sviluppo software, sono in grado di definire, scegliere, modellare e produrre specifiche di metodo e tool fatti su misura nello stesso modo in cui al livello di sviluppo software vengono prodotte le specifiche del sistema in esame.

Lo scopo dei processi di sviluppo software e della *method engineering* è lo stesso: il rilascio del software; inoltre, così come la software engineering è supportata da metodologie e tool, anche la method engineering è supportata da metodi e tool ai quali, per non far confusione, viene aggiunto il prefisso meta; a livello di method engineering abbiamo, allora, i meta-metodi, la meta-modellazione ed i meta-CASE, questi ultimi più comunemente chiamati CAME tool (Computer Aided Method Engineering tool).

5.2.3.4 Method Fragment

Un metodo può essere considerato come una collezione predefinita ed organizzata di tecniche ed un insieme di regole che stabiliscono da cosa, in che ordine ed in quale modo dette tecniche vengono usate per raggiungere e mantenere alcuni obiettivi, quindi metodo è la procedura per ottenere un oggetto .

Con questa definizione si enfatizza più il processo in se che la rappresentazione cioè i prodotti o artefatti di detto processo. Si deve tenere in considerazione che un metodo include anche conoscenze sugli utenti dello stesso, valori ed obiettivi di sviluppo.

Un metodo descrive non solo i modelli che vengono sviluppati ma anche come sono strutturati ed organizzati.

Quindi un metodo si caratterizza tramite la: struttura, contenuti ed uso(*Booch Rumbaugh*)

Il concetto di method fragment è stato definito da Harmsen nel 1997, esso è *una descrizione di un metodo di ingegneria del software o qualunque parte coerente di questo.*

Un metodo, ancora, secondo Brinkkemper è *un approccio alla costruzione di sistemi di sviluppo di progetti, basato su uno specifico modo di pensare, composto da direttive e regole, strutturato in modo sistematico in attività di sviluppo con corrispondenti prodotti dello sviluppo.*

Più in dettaglio ancora possiamo dire che un method fragment è un pezzo di processo, esso rappresenta un contesto fatto da una coppia, situazione /intenzione dove la *situazione* rappresenta la parte del prodotto sottoposto al processo o l'input del pezzo di processo e *l'intenzione* riflette il goal che deve essere raggiunto in una situazione, il corpo di una direttiva (la guideline) dettaglia su come applicare il frammento per realizzare una intenzione.

Allora un metodo, e di conseguenza un method fragment, è composto da una parte relativa alla struttura del prodotto (gli artefatti derivanti dall' attività di sviluppo) e le procedure necessarie per costruire questi artefatti.

5.2.3.5 Come definire i method fragments

Quando si crea un nuovo metodo [33][34], secondo il paradigma della method engineering, il method engineer estrae dal method base un set di frammenti di metodologie già

esistenti ,già definiti e memorizzati, e li integra tra loro secondo una ben specifica tecnica di assemblaggio..

Il nuovo metodo viene costruito a partire da pezzi di metodologie esistenti, da qui i nuovi concetti di *method fragments,* che vengono memorizzati in un repository ed assemblati tra loro, riutilizzati, da un method engineer.

A monte sta il lavoro, a questo punto fondamentale visto il numero delle metodologie esistenti, della modularizzazione delle metodologie e della definizione dei method fragments .

Si ha a che fare con una re-ingegnerizzazione [35] dei metodi per creare i fragments, per cui diventa essenziale il modo in cui questi vengono rappresentati, e descritti, queste parti di metodo e soprattutto il modo in cui si recuperano dal repository e si assemblano per formare il nuovo metodo.

Il processo di re-ingegnerizzazione parte dalla rappresentazione della metodologia esistente in una maniera modulare dalla quale vengono, poi, prelevati i frammenti da depositare nel method base.

Ogni frammento sarà adatto ad uno o più progetti di sviluppo di information system, così la costruzione del nuovo metodo parte dalla scelta dei frammenti adatti allo specifico progetto.

La rappresentazione modulare di un metodo viene effettuata dal method engineer tramite quello che si chiama Method Reengeneering Process Model che lo guida nella ricostruzione di ogni metodo in un insieme di method fragments , ne fornisce le direttive, nel frattempo istanzia un Method Meta-Model per descrivere i method fragment identificati ,come si può vedere dalla figura successiva

Un metodo è formato, come abbiamo detto, da due parti: il processo ed il prodotto; un method fragment è una parte di metodo che include queste due parti, è quindi un modulo coerente che accoppia parti del processo ad i relativi prodotti.

La definizione di un method fragment è un processo guidato nel senso che il method engineer ha bisogno di direttive (guidelines) per creare un modulo completo che possa esser facilmente riutilizzato.

Queste direttive diventano allora parte fondamentale del processo di modularizzazione di una metodologia.

5.3 Agile software development

5.3.1 Introduzione

Se si guarda alla vita di una persona comune si può vedere come questa stia diventando, col tempo, sempre più frenetica e stressante; questa frenesia si riscontra, come riflessa, anche nella evoluzione del mondo e del mercato dei computer e dei software.

I computer stanno diventando sempre più veloci ed offrono sempre più potenzialità e i professionisti dell'ambiente cercano di sviluppare software che stiano al passo con tali nuove potenzialità, creando processi che, non solo rispondano bene ai cambiamenti che si verificano durante la fase di produzione, ma soprattutto che li seguano e li comprendano. L' obiettivo è che il software tenga conto del requisito fondamentale che è il cambiamento del mondo cui è rivolto.

La attività di sviluppo software può risultare molto caotica e complessa se non organizzata in maniera adeguata attraverso un progetto o un piano di sviluppo.

Attualmente, soprattutto per progetti relativamente piccoli, viene scritta la maggior parte del codice a braccio, e man mano che i progetti crescono diventa sempre più difficile pianificare il lavoro, svilupparlo e soprattutto individuarne gli errori, che soltanto una lunga fase di test, a progetto completato, può scoprire ed eventualmente eliminare.

Per affrontare tutto ciò ed imporre uno stile alla progettazione ed allo sviluppo software si è fatto ricorso, negli ultimi anni, alle metodologie ingegneristiche.

Seguire una metodologia significa imporre al proprio progetto uno sviluppo disciplinato, che segua determinate regole, rendendo così lo sviluppo del software molto efficiente e soprattutto prevedibile.

5.3.2 Agile software development

Le metodologie di progettazione classiche sono note per essere di grande successo ma sono anche state criticate per essere troppo burocratizzate, nel senso che c'è bisogno di una

buona dose di conoscenza ed una notevole quantità di materiale per seguirle e metterle in pratica.

Una via di mezzo tra la totale assenza di processo e metodologie troppe complesse è quella che viene chiamata *metodologia agile*, dove il termine agile dà esattamente l'idea dell'obiettivo che si vuole raggiungere con tali metodologie: una metodologia che aiuti in maniera agevole e semplice nello sviluppo di un progetto software e soprattutto che sia adattiva e disponibile ai cambiamenti, che è la cosa più importante in questo nuovo approccio metodologico.

Le metodologie agili sono una reazione allo sviluppo software tradizionale ed una alternativa ai quei metodi, definiti pesanti, fortemente orientati alla produzione di documentazione dettagliata e di buon livello.

I processi tradizionali comprendono una prima fase di raccolta dei requisiti ed una abbondante documentazione di questi, poi un progetto architetturale, lo sviluppo ed il testing.

Negli ultimi anni è diventato sempre più difficile seguire questo approccio alla modellazione dei processi a causa, come detto prima, del continuo variare dei requisiti, e della difficoltà del cliente ad interagire con gli sviluppatori dai quali si aspetta software sempre più complessi che soddisfino i suoi bisogni.

Uno dei risultati di tutto ciò è stato che molti sviluppatori, basandosi sulla loro esperienza hanno cominciato a confezionare dei metodi adatti ai loro scopi, come detto nella precedente sezione al riguardo della method engineering.

I metodi agili sono, attualmente, un insieme di tecniche e norme che condividono gli stessi valori e le stesse norme, per esempio molti si basano sull'*incremento iterativo.*

La maggior parte delle norme agili non rappresenta nulla di nuovo, è piuttosto, nuova la filosofia di lavoro, potremmo dire, sulla quale si basano i metodi agili, essi tendono al miglioramento del processo software; sono una evoluzione nella quale un nuovo processo è costruito sempre sui fallimenti e i successi di quelli precedenti.

Una prima, importante, differenza che si nota utilizzando un metodo agile, a dispetto di un metodo classico, è la minore quantità di documentazione da produrre orientando i propri sforzi principalmente allo sviluppo di codice.

Un processo agile si può dire sia orientato al codice, poiché si adotta il punto di vista secondo il quale l'elemento chiave della documentazione è il codice sorgente. Ridurre la quantità di documentazione significa anche essere più aperti e flessibili ai cambiamenti.

Non è, però, tanto questa la differenza fondamentale, quanto il fatto che una minore quantità di documentazione porta i metodi agili ad essere più adattivi che predittivi e più orientati alla persona che al processo.

Più adattivi che predittivi perché i metodi classici sono pensati per pianificare ogni cosa nei minimi dettagli, diventando così poco inclini al cambiamento, mentre l'imprevedibilità dei requisisti è quasi sempre l'unica costante di un progetto di sviluppo software. Un metodo adattivo è in grado di adattarsi, appunto, alla variabilità dei requisiti attraverso successive fasi di iterazioni ed incremento.

Inoltre, i metodi classici tendono a separare le fasi strettamente inerenti al progetto da quelle inerenti la realizzazione vera e propria, portando all'inconveniente di dovere utilizzare una notazione ben specifica, per esempio UML, che descriva tutte le decisioni significative prese in fase di progetto, in modo tale da essere poi facilmente passate ai programmatori.

I modelli e le notazioni utilizzate possono risultare ottimi su carta ma di difficile trasformazione in codice.

Più orientati alla persona che al processo nel senso che non si focalizzano nel creare conoscenza ed abilità particolari negli sviluppatori ma piuttosto li aiutano e supportano nel processo di sviluppo.

In un ambiente predittivo l'apprendimento è spesso scoraggiato. Si prendono decisioni in anticipo e quindi si segue quel progetto.

In un ambiente adattivo l'apprendimento coinvolge tutte le parti in causa - gli sviluppatori e i loro clienti - al fine di valutare le loro assunzioni e di utilizzare i risultati di ogni ciclo di sviluppo per adattare il successivo.

Il beneficio più importante, potente, inseparabile, predominante del Ciclo di Vita dello Sviluppo Adattivo sta nel fatto che ci costringe ad affrontare i modelli che sono alla base delle nostre illusioni. Ci costringe a valutare in modo più realistico le nostre capacità .[Highsmit]

5.3.2.1 Prevedibilità/imprevedibilità dei requisiti

Si diceva che una delle certezze durante lo sviluppo di un software è che i requisiti cambiano continuamente.

Spesso si tende a considerare questo aspetto come l'effetto di una ingegneria dei requisiti condotta in maniera errata, infatti la base dell'ingegneria dei requisiti è il tentativo di ottenere una rappresentazione chiara dei requisiti prima di iniziare la realizzazione del software e di fare accettare tali requisiti al cliente, ed infine preoccuparsi di minimizzare le modifiche ai requisiti solo dopo che questi sono stati accettati.

In genere, però una società di sviluppo non fornisce informazioni sui costi relativi ai requisiti, allora diventa difficile produrre delle stime, anche perché lo sviluppo del software è un'attività di progettazione, e quindi è difficile da pianificare e valutare nei costi ed inoltre i materiali di base continuano a cambiare rapidamente.

La natura intangibile del software, inoltre, gioca un ruolo fondamentale in questo caso, in quanto fino a quando esso non viene rilasciato e non viene messo in funzione non ci si può accorgere del valore delle sue funzionalità, cioè quali sono quelle veramente utili e quali non lo sono.

Questo conduce al punto di ritenere che i requisiti dovrebbero essere modificabili, ma purtroppo nello sviluppo del software qualsiasi cosa dipende dai requisiti e se non si è in grado di avere dei requisiti stabili non è possibile ottenere una pianificazione prevedibile.

In molti campi applicativi, in realtà la predittività è possibile, per esempio tutti coloro che lavorano nello sviluppo del software per lo Space Shuttle della NASA sono un eccellente esempio di contesto in cui tale sviluppo può essere considerato prevedibile.

La predittività è una proprietà molto desiderabile e molto ambita, però cercare di essere predittivi quando invece non è possibile, porta a situazioni di mancanza di risorse per la soluzione dei problemi che sorgono quando una pianificazione, fissata troppo rigidamente in anticipo, fallisce; generalmente il fallimento è doloroso e costoso.

Allora quando ci si trova un una situazione che presenta requisiti non prevedibili diventa impossibile usare una metodologia predittiva?

Così si perderebbero molti dei modelli già sviluppati ed usati da tempo, ma rinunciare alla predittività non significa ricadere in un caos incontrollabile.

Serve è allora un processo che consenta un controllo sull'imprevedibilità, che è esattamente il significato dell'adattività.

5.3.2.2 Iteratività

Un modo molto efficace per tenere sotto controllo i continui cambiamenti è procedere, nella modellazione, attraverso piccoli incrementi provando tutto con il codice, quindi uno sviluppo iterativo.

L'idea dello sviluppo iterativo in realtà non è completamente nuova ed il suo risvolto pratico fondamentale è quello di realizzare continuamente dei sottosistemi funzionanti che non hanno tutte le funzionalità del sistema finale ma una volta testati attentamente ed integrati dovrebbero rispettarne tutte le caratteristiche.

Lo sviluppo iterativo ha senso nei processi predittivi, ma è essenziale nel contesto di quelli adattivi, perché un processo adattivo deve essere in grado di affrontare le modifiche nelle funzionalità richieste.

Questo porta ad un tipo di pianificazione con piani a lungo termine poco specificati, mentre i piani stabili sono soltanto quelli a breve termine che sono relativi ad una singola iterazione.

Lo sviluppo iterativo fornisce ad ogni iterazione un solido fondamento sul quale è possibile basare le successive pianificazioni.

Un sistema testato ed integrato è il modo migliore per rendere realistico un progetto poiché i documenti ed il codice non testato possono nascondere qualunque tipo di errore , rimandando la scoperta dei malfunzionamenti solo in fase di uso del sistema progettato.

È doveroso, però, a questo punto dire che le metodologie agili non sono delle metodologie complete ma un supplemento a quelle esistenti.

5.3.3 Il Processo agile

Da tutto quanto detto sopra si evince che i processi agili di sviluppo software danno la possibilità di produrre software o di modificarne di esistenti nella maniera più veloce possibile utilizzando delle tecniche che si adattano, continuamente, ai cambiamenti dei requisiti e dell'ambiente di sviluppo ed ai cambiamenti dovuti all'esperienza acquisita durante il processo di sviluppo stesso.

I processi agili sono orientati principalmente allo sviluppo di codice strutturando il processo in iterazioni alla fine di ognuna delle quali viene tirato fuori codice funzionante e qualunque altro tipo di documento che sia utile al cliente, prima di tutto, ed al progetto poi.

A questo proposito i Metodi Agili contano un certo numero di sostenitori e di oppositori. Alcuni detrattori delle modellazioni agili pensano che tutta questa focalizzazione sul codice possa portare ad una carenza dal punto di vista dell'analisi e della progettazione del modello e della relativa documentazione ed anche ad una minore propensione da parte di coloro che utilizzano il metodo a fare esperienza ed imparare l'uso del metodo; l'esperienza è chiaramente alimentata dalla buona documentazione su quanto viene fatto.

Inoltre ci sono sicuramente alcune limitazioni nell'utilizzo di processi agili in alcuni domini di applicazione.

È da notare la mancanza in letteratura di riferimenti agli insuccessi delle metodologie agili, a causa della constatazione che un eventuale fallimento di un progetto è dovuto principalmente alla errata applicazione delle norme fondamentali delle metodologie agili piuttosto che alla effettiva funzionalità delle stesse.

Questa potrebbe essere una indicazione del fatto che, se ben implementato,

un processo agile funziona.

Da quando si è cominciata a sentire la necessità di snellire i metodi di sviluppo software classici, per fronteggiare i nuovi crescenti bisogni di velocità e facilità di progettazione, molti processi sono stati definiti agili senza in realtà esserlo.

Per evitare confusione su quello che deve esattamente essere un processo agile, un gruppo di sostenitori si è incontrato, dandosi il nome di Agile Alliance [36], e messo d'accordo su quello che viene chiamato Agile Software Development Manifesto

(http://www.agilealliance.org/principles.html) nel quale sono elencati una serie di principi fondamentali e fondanti un processo agile.

Essi puntualizzarono che il loro movimento non era anti-metodologia, anzi volevano ridare credibilità alle metodologie, utilizzando tecniche di modellazione, ma non per produrre diagrammi fini a se stessi, e producendo documentazione, ma non producendo migliaia di pagine che rimangono poi non lette ed abbandonate.

5.3.3.1 The Agile Manifesto

Il manifesto [37] suggerisce delle preferenze non delle alternative. I criteri elencati sono:

- Soddisfare il cliente consegnandogli continuamente, fin dai primi giorni di sviluppo software, il materiale prodotto.
- Affrontare i cambiamenti nei requisiti anche a lavoro inoltrato per avvantaggiare il cliente (rendendolo competitivo).
- Consegnare frequentemente software lavorante, ad intervalli di due settimane fino a due mesi, preferibilmente questi intervalli devono essere i più piccoli possibili.
- Utenti clienti e sviluppatori devono lavorare insieme lungo tutto il periodo di sviluppo del progetto
- Costruire i progetti intorno a persone motivate dando loro ambiente di sviluppo e supporto di cui hanno bisogno.
- Il modo più efficace ed efficiente per scambiare informazioni all'interno del team di sviluppo è quello di parlare, molto, faccia a faccia.
- Una prima misura del progresso è il software lavorante.
- I processi agili promuovono lo sviluppo "sostenibile". Lo sponsor gli sviluppatori e gli utenti dovrebbero essere capaci di mantenere un andamento costante indefinitamente.
- Continua attenzione all'eccellenza tecnica e ad un buon progetto migliora l'agilità.
- La semplicità, l'arte di minimizzare il lavoro non fatto, è essenziale.
- Le migliori architetture, requisiti e progetti emergono dai team auto_organizzati.

I concetti esposti nel manifesto si possono riassumere tramite la seguente figura:

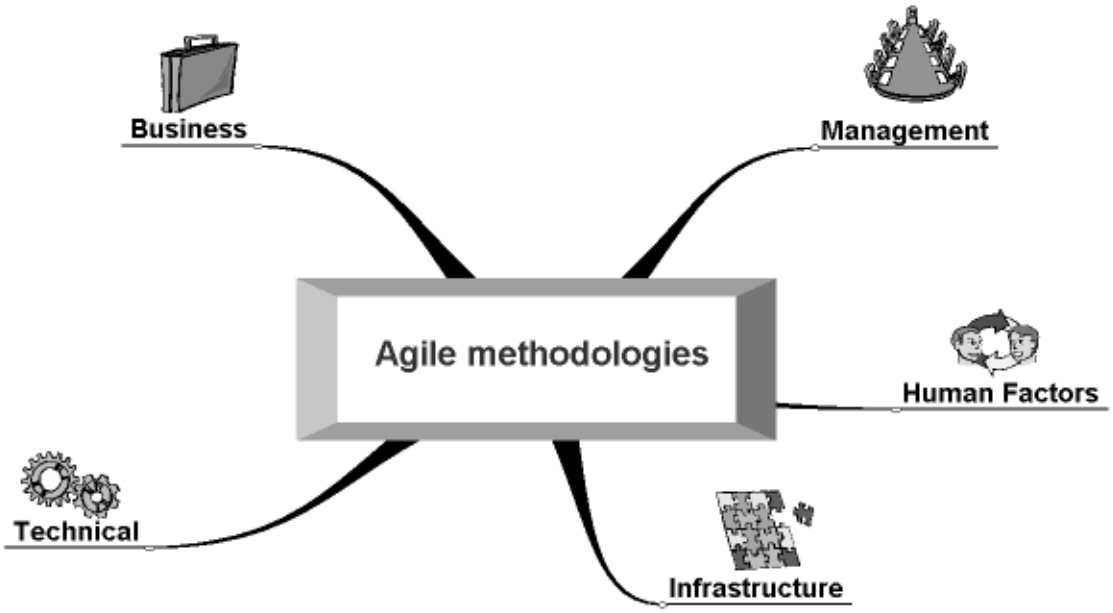


Figure 5-6 . Le metodologie agili

dove, si evidenziano:

- l'importanza che si attribuisce più alla persona, lo sviluppatore, che ai processi strettamente normativi ed ai tool che li supportano,
- l'importanza della comunicazione tra gli elementi del team e della programmazione in coppia;
- le ricerche che vengono fatte per scoprire quali tool siano di aiuto alla modellazione agile;
- l'orientamento verso lo sviluppo di quelle applicazioni commerciali fortemente sensibili a vincoli di tempo e di costi

5.3.3.2 Cenni ad alcuni metodi precedenti quelli agili

Metodo Waterfall (Royce, 1970), venne realizzato per valutare e modellare i bisogni dell'utente, esso inizia con un'attenta analisi dei requisiti e dei bisogni dell'utente/cliente,

stilando una dettagliata lista dei requisiti funzionali e non, lungo mesi, se non addirittura anni, di lavoro tra utenti e sviluppatori, al termine dei quali i requisiti fissati all'inizio potrebbero essere cambiati.

Tutta la documentazione tirata fuori in questa prima fase serve per la successiva, quella del progetto il cui scopo è tracciare un'architettura dettagliata, e ovviamente documentata, del sistema; questa ultima documentazione viene poi usata dai programmatori per l'implementazione del progetto che, perfettamente progettato, viene testato.

Teoricamente questo approccio sembra ottimo ma in pratica non lo è perché l'utente potrebbe non esser più molto sicuro su quello che vuole, i requisiti possono cambiare e risulterebbe difficoltoso fermare il processo di produzione di tutta la quantità di documenti, di cui si diceva sopra, per introdurre una variazione che potrebbe portare a buttare tutto quanto fatto, o ancora, se le variazioni intervengono con intervalli di tempo troppo brevi gli sviluppatori potrebbero non avere il tempo di produrre la giusta mole di lavoro per accomodare anche piccoli cambiamenti. In questo metodo allora è fondamentale la rigidezza dei requisiti.

Una piccola miglioria è data dalle tecniche incrementali ed iterative, le prime tendono a ridurre i tempi di sviluppo spezzando il progetto in incrementi che si possono sovrapporre, le altre dividono il progetto in iterazioni di lunghezza variabile ognuna delle quali fornisce un prodotto completo costruito su tutto il codice e la documentazione prodotti nei passi precedenti; ogni pezzo è un piccolo processo completo, sviluppato dal processo Waterfall, che consta di tutte le sue parti, dalla raccolta dei requisiti al testing.

Il Capablity Maturità Model (CMM) è un modello su cinque livelli che descrive le norme di buon *management* ed imposta le priorità di cui tenere conto perché una organizzazione migliori dal primo livello (quello definito Chaotic) al quinto (Optimized).

Esso è orientato alle organizzazioni ed ai progetti molto vasti ed il suo obiettivo principale è di realizzare un processo che sia consistente, prevedibile e credibile.

Nel 2001 M. Poppendieck,[35] facendo riferimento ad altre discipline di ingegneria, e soprattutto guardando al successo della Lean Manufacturing una azienda di fabbricazione di macchine agricole che raggiunse livelli di produzione molto elevati non seguendo alcun

processo predefinito, elencò dieci norme base distintive della bontà e del successo dei processi che avevano studiato:

1. eliminare lo spreco
2. minimizzare le scorte: gli artefatti intermedi come documenti di requisti e di progetto
3. massimizzare il flusso: usare uno sviluppo iterativo per ridurre il tempo di sviluppo
4. estrarre informazioni dalle richieste: supportare requisiti flessibili
5. dare pieni poteri ai lavoratori
6. andare incontro alle richieste del cliente : lavorare a stretto contatto con lui
7. fare bene fin dalla prima volta : eseguire test presto e quando necessario *refactoring (tecnica per migliorare l'architettura e la struttura interna di un sistema preservando i comportamenti osservabili)*
8. abolire l'ottimizzazione locale : poter dare libertà d'azione
9. far coppia con i fornitori : evitare i contrasti e lavorare insieme per lo sviluppo del software migliore
10. creare la cultura del continuo miglioramento : imparare dagli errori e dai successi.

Lei si rese conto che i dieci principi appena esposti potevano essere alla base di un processo che accettasse i cambiamenti piuttosto che irrigidirsi sulla prevedibilità, e che enfatizzasse la creatività del singolo, gratificando il suo lavoro, piuttosto che stereotiparsi nella produzione di documenti complessi.

Negli anni 90, indipendentemente dalla Poppendiek, Kent Beck [38][39] riscoprì ed applicò gli stessi principi ad un progetto, Chrysler Comprehensive Compensation (C3), di gestione del pagamento di stipendi. Questo progetto era stato un completo fallimento e quando Beck e Ron Jeffris vi misero mano decisero di buttare tutto il codice esistente e di ricominciare daccapo.

Nel giro di un anno il progetto divenne effettivo e fu un successo; era la prima volta che veniva usata la eXtreme Programming, questa nuova metodologia faceva riferimento ai valori espressi dalla Lean Programming di M. Poppendiek.

Queste nuovo approcci metodologici fecero il giro del mondo e fecero perdere in po' di fascino alle metodologie tradizionali che, ci si rese conto , non potevano essere adatte a tutte le situazioni, riconoscendo quindi il bisogno di nuove norme, necessarie per affrontare al meglio i cambiamenti, flessibili ed orientate alle persone.

Nel 2001 Highsmith e Cockburn riassunsero questa nuova tendenza nello sviluppo software con queste frasi:

- Soddisfare il cliente deve avere la precedenza sul rispettare i progetti originali
- I cambiamenti possono accadere, ci si deve concentrare non su come prevenirli ma su come affrontarli per ridurne i costi lungo il processo di sviluppo
- L'eliminazione dei cambiamenti troppo presto significa insensibilità alle condizioni del mercato e quindi fallimento dal punto di vista commerciale
- Il mercato richiede e si aspetta software innovativi e di alta qualità che soddisfi le sue necessità, e presto

5.3.3.3 Limiti delle metodologie agili

Basandosi sull'osservazione dei sopraesposti principi si possono muovere alcune critiche alle metodologie agili [40].

Sicuramente le metodologie classiche non si adattano ai principi esposti nel manifesto ma neanche alcuni processi agili i quali devono, a volte, essere estesi tramite le norme degli sviluppi classici per riuscire ad implementare tutti i principi del manifesto.

Si è detto dell'importanza della comunicazione tra gli elementi dello stesso team e tra il team e il cliente; questo è possibile quando non ci si trova in ambienti di sviluppo distribuito. In tali casi la comunicazione tra gli sviluppatori, principalmente, è importante e necessaria quanto quella degli ambienti non distribuiti e la si potrebbe realizzare, per esempio, in video conferenza o con l'ausilio di documenti, testuali o porzioni di codice, con i quali si illustra il lavoro fatto e si dà la possibilità a tutti di seguire l'evoluzione del progetto.

Questo va contro uno dei principi del manifesto in quanto documenti e modelli dovrebbero essere creati soltanto, per dare valore al progetto, non per gestire la comunicazione tra gli sviluppatori; la comunicazione deve essere elusivamente verbale solo così non si rallenta un processo di sviluppo.

Inoltre la modellazione agile supporta i meccanismi di comunicazione, controllo e coordinazione che si possono applicare soltanto in ambienti con un piccolo numero sviluppatori.

In un team molto ampio il numero ed il tipo di comunicazioni è tale da non potere essere semplicemente *face-to-face*, in questo caso sono molto più adatte le tecniche di sviluppo classiche.

Il modo di produrre artefatti (*deliverable*) contrasta con tutto quello che riguarda la stipula di un contratto da parte di un cliente, questo vorrebbe sapere a priori quali sono i tempi di sviluppo i requisiti e quant'altro possa essere fissato per stabile un compenso.

Ma tutto quello che è "fissato" è avulso dalla filosofia della metodologia agile ed allora risulterebbe impensabile riuscire a stipulare un contratto a meno che non si suddivida in due

parti, una fissa ed una variabile che definisce quali requisiti e quali *deliverables* possono variare all'interno delle condizioni fissate nella prima parte del contratto.

Le tecniche di modellazione agile, da sole, potrebbero non essere sufficienti per sviluppare un progetto orientato ad applicazioni software *safety-critical*.

Per questo tipo di applicazioni, sicuramente, le tecniche standard sono più adatte dal punto di vista della rigorosità delle specifiche e delle procedure di testing e per la formalità dell'attività di analisi, ma di conseguenza più costose.

Alcune norme dei processi agili potrebbero portare benefici a questo tipo di applicazioni, per esempio la produzione precoce di codice supportata da processi incrementali ed iterativi può essere utile nel caso di mancanza di requisiti ben definiti, il tipo di approccio al test o la programmazione in coppia rappresentano delle valide integrazioni ai metodi classici.

Uno dei principi delle metodologie agili si basa sul presupposto che non c'è bisogno di un progetto per apportare dei cambiamenti al codice perché ogni cambiamento è gestito da quello che si chiama *refactoring* del codice.

Quanto appena detto non è, però, applicabile al caso di sistemi complessi nei quali il cambiamento anche di una minima parte di codice può portare a costi notevolissimi richiedendo quindi che questo venga abbondantemente previsto in anticipo, ed in questo caso un modello gioca un ruolo essenziale.

Allora si può dire che ci sono delle applicazioni in cui uno sviluppo agile è indicato ed anzi facilita, migliora ed abbassa i costi del processo di sviluppo ed altre che per la loro natura necessitano delle metodologie tradizionali con modelli e requisiti rigidamente previsti e pianificati dall'inizio.

Sembra chiaro allora che molte applicazioni sono disponibili ad esser trattate da metodi agili, alcune di queste sono, per esempio quelle in Internet che seguono velocemente l'andamento del mercato e nelle quali il miglioramento di una *release* non comporta eccessivi contributi in termini economici ed in termini di tempo.

È, inoltre, chiaro che alcuni aspetti di un processo di sviluppo software possono beneficiare di un approccio agile mentre altri hanno, per loro natura, necessariamente bisogno di un approccio più rigido e sistematico che segua ben precise norme.

Possiamo classificare i processi di sviluppo software in base al loro grado di agilità in una scala che va da quello puramente normativo, approccio classico, nel quale vengono definiti dettagliamene l'ordine ed il modo in cui sviluppare tutte le fasi che conducono al raggiungimento di un obiettivo che rimane pressoché identico dall'inizio alla fine del processo, a quello puramente agile nel quale i passi ed i goal vengono determinati dinamicamente in base a precedenti esecuzioni delle stesse fasi del processo ed al cambiamento nei requisiti e nell' ambiente di sviluppo.

Uno tra i più popolari e più normativi metodi agili è l'eXtreme Programming (XP), questo è stato sviluppato per venire incontro ai bisogni di piccoli gruppi di sviluppatori che dovevano confrontarsi spesso con problemi nei quali i requisiti erano spesso vaghi e mutevoli.

5.3.4 Extreme Programming

XP (Extreme Programming) è attualmente un approccio disciplinato e ponderato allo sviluppo software [41].

Caratteristica peculiare dell'XP è quella di sottolineare le necessità e le aspettative del cliente mettendo lo sviluppatore nelle condizioni di affrontare costantemente, anche a lavoro inoltrato, le variazioni dei requisiti e delle richieste del cliente stesso.

I quattro valori fondamentali della XP sono: la comunicazione tra gli sviluppatori ed il cliente per produrre software di alta qualità nel minor tempo possibile esaudendo costantemente i suoi bisogni; la semplicità e la pulizia nella stesura dei documenti correlati al prodotto software e soprattutto nel codice risultante; il feedback costante che viene fuori testando il software a partire dal primo giorno e consegnando il prodotto al cliente il più presto possibile così da potere implementare le eventuali modifiche da lui suggerite.

Tutto quanto appena detto rende lo sviluppatore, e di conseguenza il processo di sviluppo del software, estremamente versatile e disponibile ai cambiamenti senza eccessive perdite di tempo e soprattutto senza incrementi notevoli dei costi.

La XP, infatti, nasce fondamentalmente dall'esigenza di trattare conesti applicativi nei quali i requisiti cambiano, anzi si potrebbe dire, nei quali questa è l'unica costante, ponendo attenzione sull'osservazione di cosa rende una tecnica di programmazione veloce e di cosa, invece , la rende lenta .

Questa nuova metodologia è importante per due motivi: essa è un riesame di tutte le procedure di sviluppo software che sono diventate standard negli ultimi tempi ed è una delle nuove cosiddette "*lightweight software methodologies*" create per ridurre il costo dei software.

Questo nuovo tipo di approccio metodologico tende ad enfatizzare il lavoro del team di sviluppo creando soprattutto quello che si chiama stile di sviluppo di gruppo.

Inoltre fino ad ora si è sempre pensato che un software complesso e difficile da mantenere e modificare fosse migliore di uno semplice ed elegante mentre XP si basa sulla consapevolezza che questo non è vero; pensiamo a quanto si investe in termini di tempo, di risorse umane e di costi per lo hardware quando si deve sviluppare un progetto, quanto si

risparmierebbe se scrivessimo il nostro codice in maniera semplice comprensibile e facile da estendere?

Tutto questo determina una svolta decisiva da quanto era inteso finora essere una metodologia di sviluppo software ed impone una rivalutazione nel modo in cui un software viene creato.

È rivalutato e modificato anche il concetto di team di sviluppo; innanzi tutto esso deve essere formato da un numero di persone compreso tra due e dodici, a seconda della complessità del progetto, e deve comprendere, non solo gli sviluppatori, ma anche i manager ed i clienti che, lavorando a stretto contatto, si pongono domande, intavolano trattative e creano test funzionali per la verifica del prodotto e del rispetto dei bisogni del cliente.

L' extreme programming pone, allora, più attenzione al buon test piuttosto che al test stesso; i test sul software sono automatizzati e fatti prima, durante e dopo la scrittura del codice e nuovi test vengono creati quando vengono trovati *bug*.

I programmatori scrivono test parallelamente al codice, questi sono allora integrati al processo di produzione e garantiscono una piattaforma altamente stabile per lo sviluppo futuro. Tutto questo porta ad una più alta produttività.

Avevamo accennato ai quattro valori fondamentali dell' XP programming che servono da guida per le norme, di cui parleremo più avanti, che vengono impiegate.

Essi sono:

Comunicazione - si accentua l'importanza della comunicazione tra gli elementi all'interno del team di lavoro e tra il team stesso ed i clienti per porre rimedio ai fallimenti di alcuni progetti dovuti proprio a problemi causati da scarsità o mancanza di comunicazione.

Semplicità – realizzare il proprio progetto nella maniera più semplice possibile guardando soltanto a quello che si ha a disposizione al momento per risolvere i problemi, senza cercare di anticipare il futuro, il che spesso è un costo troppo alto da affrontare.

Feedback – si basa sulla convinzione che il progetto va avanti su basi solide se i rilasci sono brevi; le integrazioni ed il testing contribuiscono ad un feedback continuo già dalle prime fasi.

Coraggio – è più un approccio psicologico alla nuova metodologia, che non è più difensiva, ma chiede agli sviluppatori di “giocare per vincere”. Il coraggio è potere dire “ho progettato abbastanza per ora, lascio che il futuro abbia il suo corso”.

5.3.4.1 Una lightweight software methodology

Le metodologie software in genere consistono in un insieme di regole e norme che aiutano nella creazione di programmi .

Buona parte di queste, soprattutto quelle che sono state sviluppate e si sono imposte negli anni 80, contengono molte regole, norme e documenti che richiedono tempo e direttive ben disciplinate per essere seguite correttamente.

Per questo motivo queste sono state chiamate “*heavyweigth metodology”;* al contrario le “*lightweight methodology*” contengono poche regole e norme e soprattutto queste sono facili da seguire.

Inoltre per aiutare i progettisti a seguire tutta questa mole di norme sono stati creati dei tool di supporto, CASE tool, che però col tempo si sono dimostrati delle vere e proprie spade di Damocle , nel senso che, creati per aiutare, sono, a volte, difficili da usare essi stessi con il risultato che molto spesso si tende a smussarne gli angoli o a saltarne parti per non perdere troppo tempo.

Un programmatore che si riduce a far questo, non fa altro che ignorare alcuni aspetti, o regole, della metodologia che aveva deciso di utilizzare per il proprio progetto, mettendosi, istintivamente, in una situazione in cui sono presenti poche regole, alla fine quelle più facili da seguire e da gestire.

Durante un progetto si può, allora, rischiare di transitare da una “*heavyweigth*” ad una “*lighweigth methodology*”, intesa però nel senso più brutto del termine, cioè mancanza di regole ben precise e documentate piuttosto che facilità di esecuzione di alcune ben precise.

Perché allora perdere tutta l'esperienza e la conoscenza acquisita durate gli ultimi anni? Perché non continuare a tenere alcune regole, per esempio quelle che ci aiutano a creare software di qualità ed abbandonare quelle complesse a macchinose?

Questo è quello che viene fatto dalla Extreme Programming, che è solo una delle varie *lightweigth methodology* esistente, ed è stata sviluppata basandosi sull'osservazione di cosa rende la produzione software lenta e complessa e di cosa la rende veloce e agevole.

5.3.4.2 Regole e norme

L' extreme programming è implementata con 12 norme che sono descritte in base alla loro aderenza ai quattro valori principali.

Pianificazione : determina la possibilità di una prossima iterazione lavorando con il cliente che da priorità agli affari e con i programmatori che danno le stime tecniche.

Piccoli rilasci : produrre subito anche piccole parti del sistema in modo che si abbia un continuo feedback con l'utente.

Metafora : è importante che il cliente capisca insieme agli sviluppatori come il sistema lavorerà.

Progetto semplice : sviluppare un progetto semplice guardando a come deve lavorare il sistema oggi, guardando alle attuali caratteristiche senza pensare a come si evolverà, questo probabilmente porterebbe a degli errori.

Testing : è un'attività che si divide in unit test, fatto dal programmatore, ed acceptance test, fatto dal cliente. Il test è l'indicatore della completezza del progetto.

Refactoring : consiste nel migliorare il progetto di software già esistenti senza modificarne il comportamento.

Pair programming : scrivere codice insieme ad un partner.

Proprietà collettiva : ogni elemento del team ha facoltà sul cambiamento di qualunque parte del progetto.

Integrazione continua : gli sviluppatori costruiscono ed integrano il software più volte al giorno.

40-ore in una settimana : cioè lavorare il più possibile, ma sospendere quando si è stanchi; è infatti negativo lavorare per troppe ore per più settimane e senza interruzione.

Clienti on_site : il cliente fa parte del team, può fare domande e deve scrivere il l'acceptance test.

Codificazione standard : adottare codice standard per migliorare la comunicazione , essendo quest' ultima, come già detto, un valore chiave.

In breve Extreme Programming è un insieme di regole e norme che ne supportano altre e che usate insieme creano una metodologia.

Le regole fondamentali sono quattro:

- planning
- designing
- coding
- testing

ed ognuna di queste consta di una serie di norme.

5.3.4.2.1 Planning

Al cuore del planning sta la stesura delle "*user stories*" , letteralmente la "versione dell'utente", il cui scopo e la cui forma sono molto simili a quella dei casi d'uso delle tecniche di ingegneria del software classica.

Le user stories vengono scritte dall'utente nel suo linguaggio naturale sotto la forma di un testo con al più tre frasi, che diano una idea di quali siano i requisiti richiesti, cosa essi vogliono che il sistema faccia.

Esse rappresentano un qualcosa di molto simile ad una analisi dei requisiti classica, ma con un livello di dettaglio molto più basso; servono principalmente a dare una stima sul tempo che impiegheranno gli sviluppatori ad implementare una specifica “story”, un requisito del sistema.

Inoltre questa prima fase di raccolta dei requisiti è strettamente orientata all’utente, cioè non si fa nessun riferimento a tecniche specifiche o ad algoritmi che si vogliono usare.

Qualora, in fase di implementazione, dovesse esserci bisogno di maggior dettaglio l’utente verrebbe interrogato e coinvolto costantemente nello sviluppo.

Le *user stories* servono anche alla creazione dei test di accettazione, di cui si dirà più avanti, e dei *release plan*, cioè della pianificazione dell’ordine di implementazione delle funzionalità e del tempo di sviluppo in termini di settimane di lavoro.

Si prediligono rilasci brevi, quindi pianificazioni che si evolvono nell’arco di poche settimane evitando il più possibile lavori extra ma che includano la fase di test.

Inoltre questo modo di pianificare il lavoro include anche una suddivisione in iterazioni

5.3.4.2.2 Designing

Alla base di questa regola sta la convinzione che un progetto semplice si completa in meno tempo di uno complesso, così spesso conviene progettare la cosa più semplice che probabilmente funzionerà e sostituire qualcosa di complesso con qualcosa di semplice.

Risulta alla fine più facile e meno costoso sostituire il codice complesso nelle prime fasi, prima di sprecare troppo tempo su esso.

5.3.4.2.3 Coding

Tutto il codice prodotto nella fase di sviluppo viene realizzato da due sole persone che lavorano insieme allo stesso computer, questi devono utilizzare tecniche e linguaggi di codifica standard e devono costantemente interagire con il cliente anche dopo la fase di stesura delle *user stories*.

Il test va creato prima del codice in quanto ciò aiuta lo sviluppatore a rendersi conto di cosa realmente lui ha bisogno; inoltre quando possibile gli sviluppatori dovrebbero integrare e rilasciare codice funzionante in ogni due ore circa.

5.3.4.2.4 Testing

Per eseguire la fase di testing bisogna creare o avere a disposizione un *test framework* che produca una *test suite* per fare i test automaticamente; un test automatico fa risparmiare il tempo ed i costi sarebbero necessari per creare il test stesso cercando e prevendendo i *bug*.

Quando viene individuato un bug vengono creati test per evitare che tale bug si riproponga e vengono creati quelli che si chiamano *acceptance test* (test di accettazione) che aiutano il cliente ad individuare e definire il problema ed a comunicarlo ai programmatori

I test di accettazione vengono fatti a partire dalle *user stories*, durante ogni iterazione; il cliente specifica una serie di scenari da testare una volta che le *user stories* sono state implementate.

Questi test sono simili ai test di sistema detti *black box*, ognuno di loro rappresenta un risultato atteso. I clienti sono responsabili della verifica della correttezza dei test di accettazione e se necessario decidono quale test fallito ha la priorità più alta.

6. PASSI Agile: una nuova metodologia di sviluppo

6.1 Introduzione

La programmazione orientata agli oggetti ha portato negli ultimi anni una grandissima rivoluzione nel modo di progettare e programmare i sistemi software; gli oggetti rappresentano una astrazione naturale con la quale modellare sistemi complessi [16].

Il grande sviluppo di alcune applicazioni orientate a specifici domini dove è richiesta una interazione tra uomo e macchina sempre più intelligente e la difficoltà nel gestire in modo corretto le interazioni e le dipendenze tra componenti di un sistema molto complesso sta portando la programmazione ad oggetti ad essere messa da parte a favore di un nuovo paradigma di sviluppo di sistemi, quello ad agenti; gli agenti rappresentano un ulteriore avanzamento nel livello di astrazione disponibile.

Perché degli agenti vengano sfruttate tutte le potenzialità si è reso necessario sviluppare delle tecniche di ingegneria del software ad hoc, confezionate in base alla natura specifica degli agenti usati e dedicate all'analisi ed alla progettazione dei sistemi ad agenti.

A tal scopo le tecniche di ingegneria del software utilizzate per la modellazione e progettazione di un sistema ad oggetti (precedentemente descritte) vengono riutilizzate ed adattate per la modellazione di un sistema ad agenti [11].

6.2 Method engineering applicata ai sistemi ad agenti

Il progetto di un sistema multi-agente, o anche di uno ad oggetti, richiede l'istanziazione di un meta-modello del sistema che fornisca una rappresentazione strutturale degli elementi (agenti, ruoli, etc.) coinvolti nella soluzione dello specifico problema e le loro relazioni.

Negli ultimi anni sono state proposte una ventina di metodologie per la progettazione e lo sviluppo di sistemi multi-agente (Cassiopea,Mase, PASSI, Message, Tropos, etc.), ognuna di queste offre differenti vantaggi a seconda delle applicazioni cui è rivolta, questo dimostra che nessuna metodologia può essere standardizzata ed adattata a qualunque situazione; da ognuna di queste possono essere prelevati elementi comuni che possono fungere da modello sul quale basare altre caratteristiche più specializzate [42].

L'orientamento attuale dei progettisti è di realizzare una metodologia per la soluzione di uno specifico problema riusando ed adattando parti di metodologie esistenti, il vantaggio non sarebbe soltanto nella possibilità di gestire meta-modelli del sistema sempre più grandi e complessi ma anche nel potere integrare velocemente sistemi esistenti.

Per gestire questo modo di operare si può ricorre al paradigma della method engineering che, come detto nei capitoli precedenti, permette di costruire specifici progetti di metodi a partire da metodi già esistenti, chiamati *method fragments*. Un frammento [43] contribuisce a migliorare le definizione delle istanze del meta-modello, sia dal punto di vista processuale che notazionale.

Per costruire una propria metodologia è utile identificare e standardizzare gli elementi comuni delle metodologie esistenti e selezionarli per comporre il meta-modello del sistema multi-agente che si vuole costruire, quindi si deve creare una relazione tra elementi del meta-modello ed elementi delle metodologie esistenti per poter individuare, modificare o adattare e ri-assemblare i frammenti.

Queste operazioni possono essere effettuate con l'ausilio di un CAME tool che offre un supporto per la costruzione di una nuova metodologia a partire dai frammenti di altre; inoltre come detto nei capitoli precedenti costruire un nuovo metodo, secondo il paradigma della method engineering, la rappresentazione in forma modulare della metodologia dalla quale si vogliono estrarre i frammenti utili al proprio scopo.

Sarà ora illustrata una metodologia di progettazione per sistemi multi-agente ed i relativi passi di modularizzazione che hanno portato alla selezione dei frammenti per lo sviluppo e la realizzazione della metodologia che è scopo di questo lavoro.

6.2.1 Descrizione della metodologia PASSI

PASSI (Process for Agent Societies Specification and Implementation) è una metodologia per lo sviluppo e l'implementazione di sistemi multi-agente che, utilizzando modelli di progetto tipici della ingegneria del software orientata agli oggetti, approcci di intelligenza artificiale e la notazione UML, supporta il progettista dalla fase di raccolta dei requsiti alla fase della stesura del codice, attraverso cinque modelli e dodici fasi, come illustrato in figura [15].

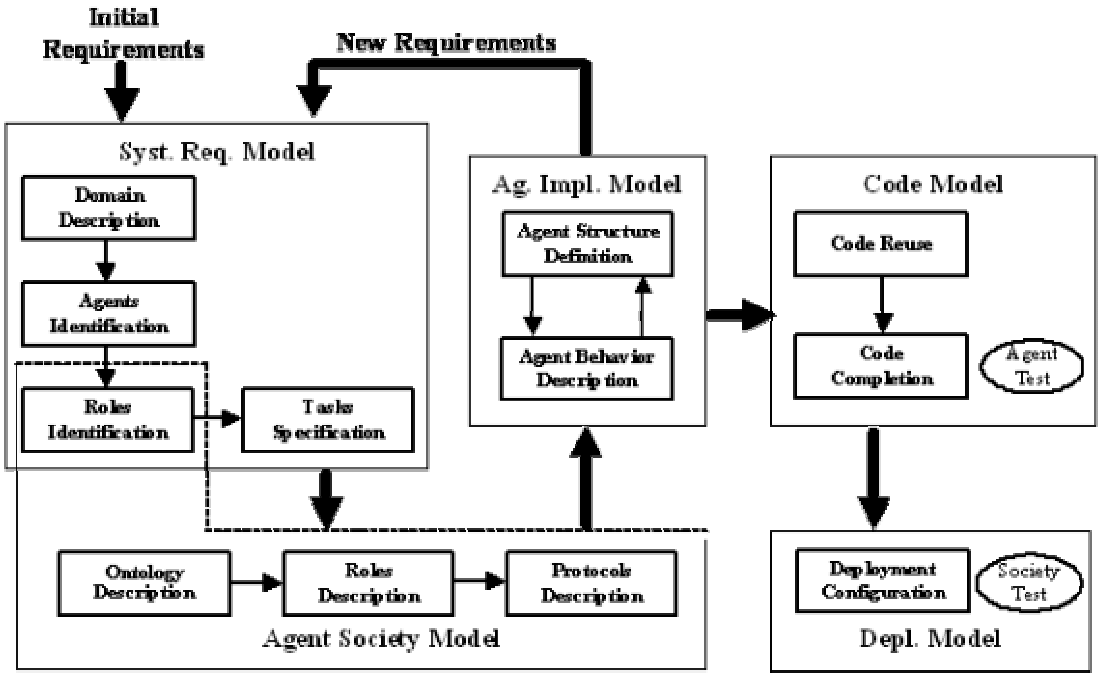


Figure 6-1 Il processo PASSI

I cinque modelli illustrati in figura sono: *System Requirements Model*, un modello dei requisiti del sistema; *Agent Society Model*, il modello della società ad agenti; *Agent Implementation Model* una rappresentazione dell'architettura del sistema sotto forma di classi e metodi; Code Model, rappresentazione a livello del codice del sistema; *Deployment Model*, il modello della dislocazione degli agenti nei singoli componenti hardware.

Durante la *System Requirements Model* vengono svolti quattro passi per fornire una rappresentazione dei requisiti ed una decomposizione del sistema:

- *Domain Description* (Descrizione del dominio) – rappresentazione funzionale del sistema attraverso diagrammi dei casi d'uso.

- *Agent Identification* (Identificazione degli agenti) – in PASSI un agente può essere visto come un caso d'uso o un package di casi d'uso, così partendo dalla precedente descrizione funzionale del sistema si crea un nuovo diagramma raggruppando uno o più casi d'uso in package, ognuno dei quali definisce le funzionalità di ogni singolo agente.
- *Roles Identification* (Identificazione dei ruoli) – in questa fase vengono rappresentati tutti gli scenari di interazione tra gli agenti che interagiscono per ottenere un determinata funzionalità del sistema; ogni agente può appartenere ad uno o più scenari che vengono rappresentati attraverso diagrammi UML di sequenza.
- *Task Specification* (Identificazione dei task) – il diagramma risultante da questa fase rappresenta il comportamento dell'agente mosrìtrando le relazioni tra gli stimoli ricevuti dall'agente ed il suo comportamento; vengono utilizzati diagrammi UMl di attività dove ogni relazione tra le attività indoca un messaggio o una comunicazione tra agenti.

Agent Society Model è il modello della società degli agenti dal punto di vista della loro conoscenza sull'ambiente circostante e delle loro comunicazioni , consiste di tre fasi:

- *Domain Ontology Description* (Descrizione dell'ontologia) – il modello dell'ontologia del sistema, dove la conoscenza dell'agente è descritta in termini di concetti, predicati e azioni attraverso un diagramma delle classi nel quale classi di colore differenti rappresentano elementi differenti dell'ontologia.
- *Communication Ontology description* (Descrizione dell'ontologia delle comunicazioni) – consiste nella composizione di un diagramma delle classi composto principalmente da due elementi, gli agenti e le comunicazioni; ogni agente è descritto in termini della sua ontologia e le relazioni tra gli agenti rappresentano le loro comunicazioni.
- *Roles Description* (Descrizione dei ruoli) – il risultato do questa fase è un diagramma delle classi in cui i ruoli sono classi raggruppate in package; i ruoli sono connessi tra loro mediante relazioni che simboleggiano cambiamenti di ruolo dello stesso agente, dipendenze per un servizio o disponibilità di risorse.
- *Protocol Description* (Descrizione del protocollo) – vengono usati diagrammi di sequenza AUML per rappresentare i protocolli di interazione defininendone l'atto di comunicazione iniziale, le possibili repliche e i passi che concluderanno la comunicazione.

L' *Agent Implementation Model* è composto da due fasi:

- *Agent Structure Definition* – è composto da una serie di diagrammi delle classi che mettono in evidenza la struttura interna degli agenti coinvolti nel processo sia dal punto di vista dell'intero sistema (*Multi Agent Behaviuor Description,*MASD) che del singolo agente (*Single Agent Structure definition* , SASD)
- *Agent Behaviour Description* – anche questa fase è suddivisa in due viste, *Multi Agent Behiour Description* e *Single Agent Behaviour Descriition;* qui diagrammi di attività o di stato vengono usati per descrivere il comportamento di ogni agente mostrando il flusso di eventi tra le classi agente principali e tra le loro classi interne.

Il *Code Model* è composto da due parti strettamente connesse tra loro:

- *Code Reuse* – fase nella quale si tenta di riusare codice sviluppato ed utilizzato in applicazioni precedenti.
- *Coding* – dove si completa la parte del codice, eventualmente mancante, della fase precedente.

Il *Deployment Model* è composto da una singola fase, *Deployment Configuration,* che descrive dove sono localizzati gli agenti e di quali unità di elaborazione si ha bisogno per la comunicazione tra gli agenti.

6.2.1.1 Modularizzazione di PASSI

Come già accennato, scopo di questo lavoro è di sviluppare una nuova metodologia agile a partire da alcuni frammenti scelti da una già esistente; si è deciso di utilizzare come metodologia di partenza PASSI per cui è stato necessario effettuarne una rappresentazione modulare per estrarne successivamente i frammenti utili al lavoro proposto [44][45].

Di seguito si accennerà al linguaggio di meta-modellazione usato per la modularizzazione e se ne farà un esempio; per la trattazione completa della rappresentazione modulare di PASSI si veda l'allegato.

6.2.1.2 SPEM (Software Process Engineering Metamodel)

Software Process Engineering Metamodel (SPEM) [46] è un linguaggio di meta-modellazione utilizzato per descrivere processi di sviluppo software ed i loro componenti. In queste specifiche ci si limita alla definizione di un insieme ristretto di elementi di modellazione di processi necessari per descrivere ogni processo di sviluppo software, vengono esclusi i vincoli e specifiche sulle aree di applicazione. La notazione SPEM risulta efficiente per la standardizzazione di modelli complessi e dettagliati.

SPEM è definito come una estensione di un sottoinsieme di UML e come un *profile* di UML, cioè una specie di variante di UML, con il vantaggio di potere usare l'espressività di UML nel definire i processi software; per esempio la modellazione dei casi d'uso non è specificatamente definita in SPEM ma la eredita da UML. Uno degli scopi principali di SPEM è supportare la definizione di processi di sviluppo software che utilizzano UML.

Fondamentale in SPEM è il concetto che un processo di sviluppo software può essere visto come una collaborazione tra entità attive astratte chiamate *Process Role* che svolgono delle operazioni chiamate *activity* su entità concrete e tangibili chiamate *work product;* questi concetti fondamentali vengono mostrati nella figura seguente.

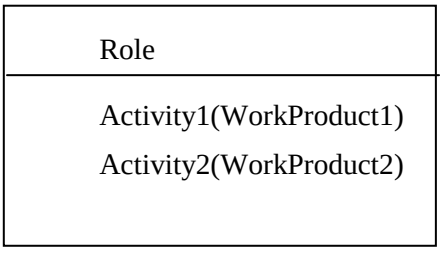


Figure 6-2 Modello concettuale

Un processo di sviluppo software viene, allora, modellato sotto questa ottica utilizzando alcuni diagrammi UML fondamentali come i diagrammi delle classi, i diagrammi di sequenza, i diagrammi di attività, dei casi d'uso ed i *package diagram*.

Questo stesso approccio è stato usato per la rappresentazione di PASSI con notazione SPEM. Per la definizione dei termini della notazione SPEM si faccia riferimento a [46].

6.2.1.3 Rappresentazione di PASSI con SPEM

Il processo PASSI viene descritto in due stadi successivi, nel primo viene considerato l'intero processo e le discipline coinvolte, nel secondo vengono dettagliate le singole fasi e sottofasi, seguendo il modello concettuale descritto sopra.

A partire da una rappresentazione procedurale di PASSI attraverso cinque fasi (Figure 6-3), ognuna rappresentante un modello descritto nel paragrafo precedente, ed i relativi *work product* risultanti, si identificano per ogni singola fase i *process role* che svolgono le attività principali, i diagrammi UML ed i documenti prodotti.

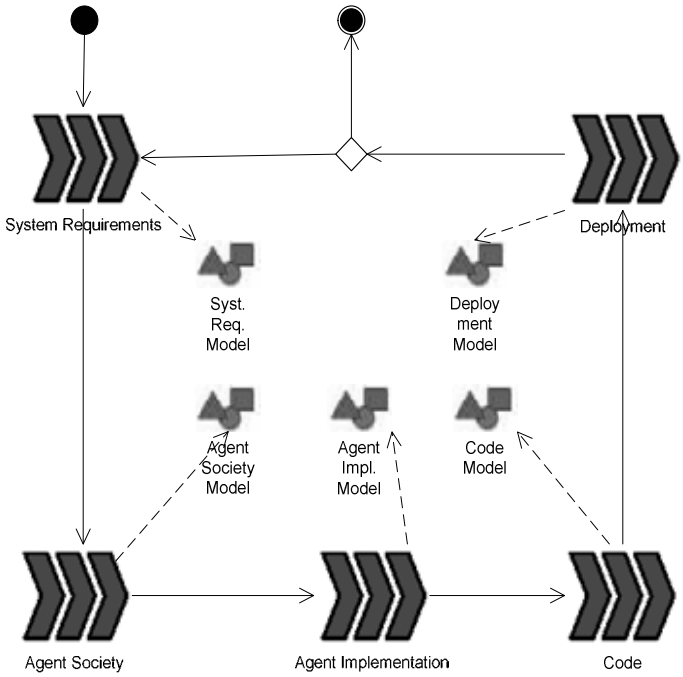


Figure 6-3 Il processo PASSI

Si procede come segue, prendendo come riferimento la prima disciplina del processo, la cui struttura principale (Process Role, work product ed activity) è mostrata in figura.

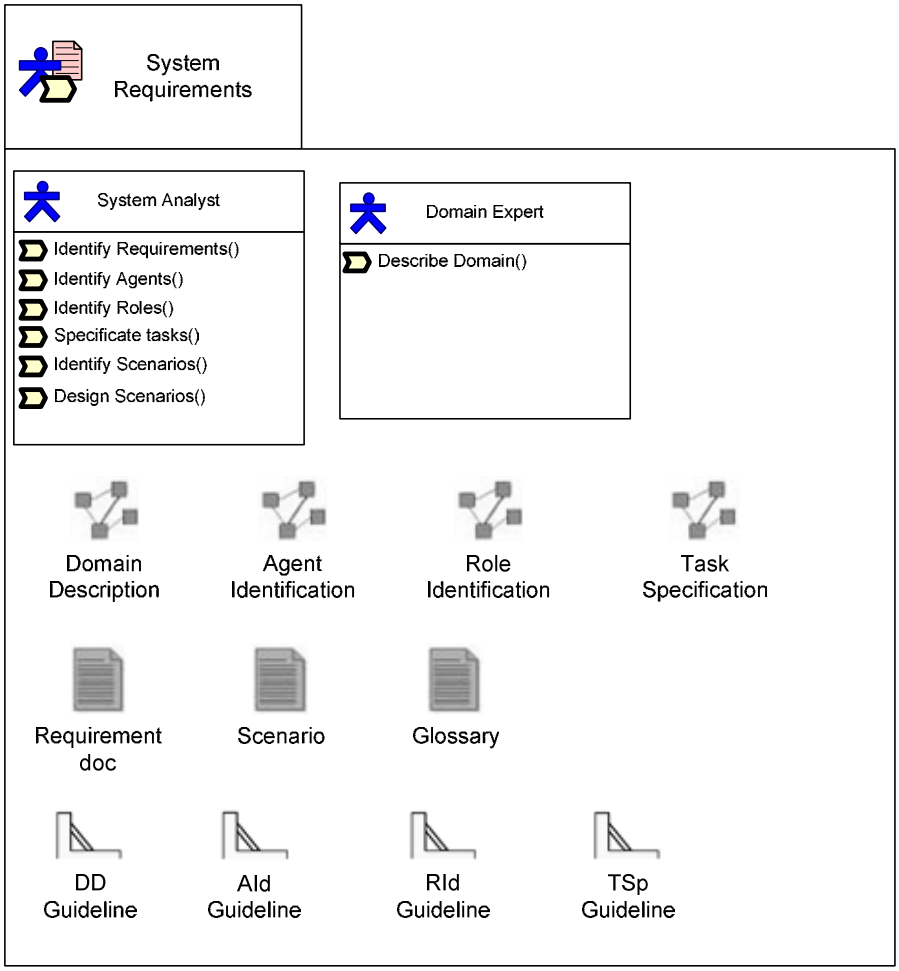


Figure 6-4 Struttura della disciplina "System Requirements"

Da qui si vede che la *System Requirements* coinvolge due *Process Role*, otto *work product* e quattro *guidance*.

Il processo svolto in questa fase è rappresentato in Figure 6-5, esso consiste in quattro *Workdefinition* (*Domain Requirements description, Agents Identification, Roles Identification and Task Specification*) con i relativi *work product*

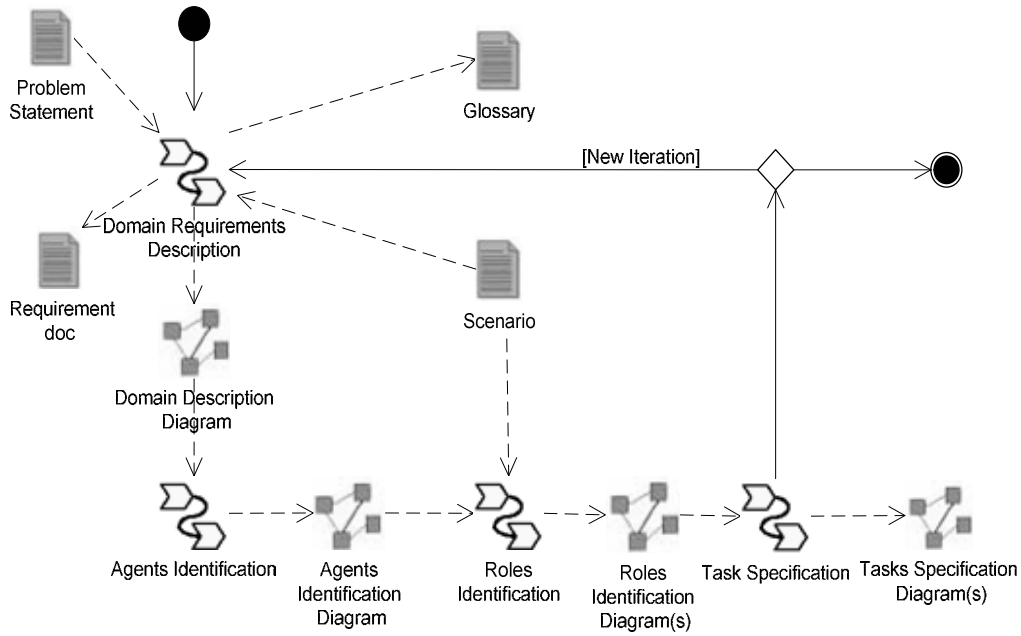


Figure 6-5 La fase System Requirements

Le quattro *work definition* sono composte da quattro *activity* svolte dai rispettivi *Process Role*; in figura sono rappresentate le suddivisioni in *activity* e le relazioni tra tutti i *process role* e le relative *activity* svolte, in particolare ogni *process role* può essere il principale attore di una *activity* (*perform)* oppure un assistente (*assist*).Tutte le fasi e le activity vengono poi dettagliate in una tabella simile a questa:

Phase	Activity	Activity Description	Roles involved
Domain Description	Identificare i casi d'uso	I casi d'uso sono usati per rappresentare i requisiti del sistema .	System Analyst (perform)

Domain Description	Rifinire i casi d'uso	I casi d'uso vengono rifiniti con l'aiuto dello esperto del dominio .	System Analyst (perform) Domain Expert (assist)
Role Identification	Identifica i ruoli	L'analista del sistema studia gli scenari testuali ed i requisiti del sistema ed identifica i ruoli giocati dagli agenti.	*System Analyst (perform*)

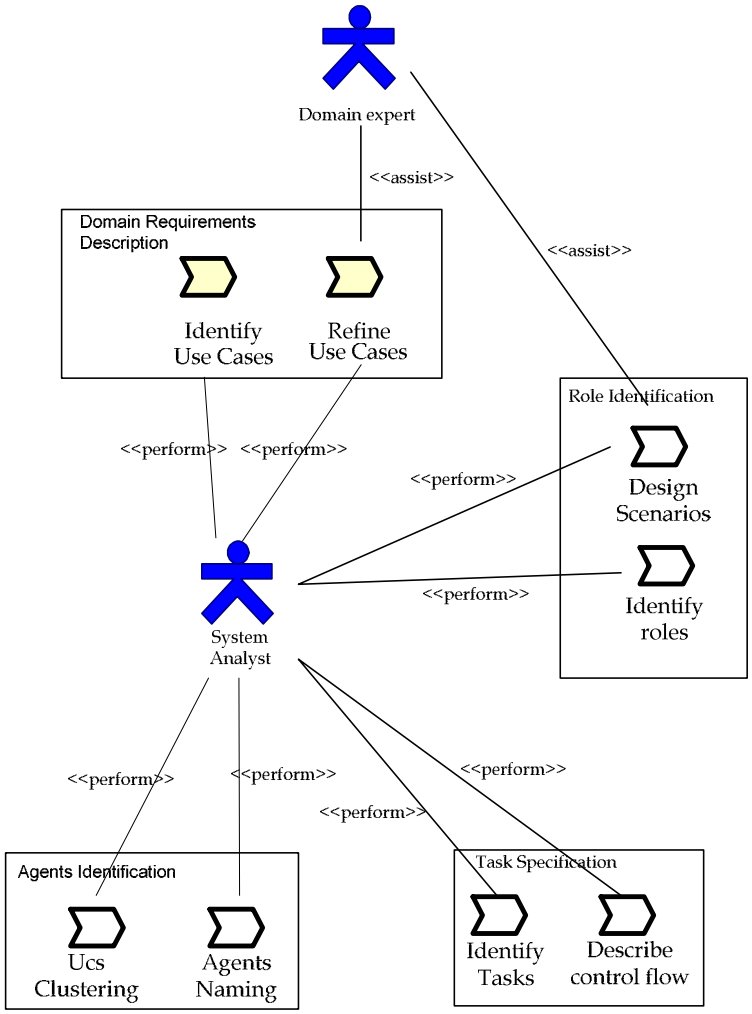


Figure 6-6 Activity della Domain Requirements Description

Ogni singola *work definition* può essere ulteriormente dettagliata tramite un diagramma delle attività dove si mettono in evidenza le parti di processo svolte dai singoli *process role* ed i relativi *workproduct*.

Infine il processo di reingegnerizzazione di una metodologia si conclude con l'identificazione delle relazioni tra ogni singolo workproduct e ogni elemento del meta-modello del MAS (Multi Agent System) e con un diagramma delle dipendenze tra i vari workproduct.

Queste ultime rappresentazioni sono molto utili per l'identificazione successiva dei frammenti in quanto descrivono i legami tra gli input e gli output dei frammenti identificati e,

cosa più importante, danno la possibilità di riferire ogni elemento del meta-modello ad un fragment, semplificandone così la identificazione e la scelta.

6.3 Lo sviluppo della nuova metodologia

Verranno ora illustrate tutte le fasi e le implicazioni teoriche che hanno portato allo sviluppo della nuova metodologia chiamata PASSI Agile.

6.3.1 Il contesto e le ipotesi di partenza

Nel corso degli ultimi anni i sistemi multi-agente hanno avuto un grandissimo incremento e sono stati utilizzati per lo sviluppo di progetti in contesti applicativi anche molto complessi.

Applicazioni di sistemi multi-agente si trovano nel campo della robotica, dell'intelligenza artificiale, delle telecomunicazioni e nello sviluppo di sistemi distribuiti; in particolar modo si sottolinea che i sistemi multi-agente ben si adattano a modellare applicazioni nel campo della robotica grazie all'alto livello di astrazione a cui si collocano, quello dello scambio di conoscenza tra sistemi autonomi.

Il processo di progettazione di un sistema multi-agente include oltre alla modellazione dell'agente in sé anche la modellazione dell' aspetto sociale (interazione inter-agente), e la rappresentazione dell'ontologia del suo dominio.

Negli ultimi venti anni sono state fatte molte ricerche in questo campo ma i linguaggi di modellazione e le metodologie sviluppate sono spesso incomplete e non sono riuscite a catturare ed a rappresentare completamente la complessità dei sistemi multi-agente.

Risultato di una ricerca orientata allo sviluppo di una metodologia teorica e pratica per la progettazione, l'implementazione ed il testing di un sistema multi-agente è PASSI (Process for Agent Societies Specification and Implementation) [15].

PASSI è una metodologia per il progetto e lo sviluppo di sistemi multi-agente il cui scopo è guidare i progettisti dalla fase della raccolta dei requisiti al codice attraverso cinque modelli e dodici fasi.

PASSI è una metodologia classica, pesantemente orientata alla produzione di documentazione di alto livello ed inoltre essa è stata pensata per progetti soggetti ad un basso livello di cambiamenti nei requisiti.

Il tentativo, in questo lavoro, è quello di riesaminare questa metodologia, basandosi sui principi e sulle tecniche della Agile Modelling, in modo da creare una nuova metodologia, una

cosiddetta “ligthweight metodology” più semplice e facile da usare, e soprattutto orientata principalmente alla produzione di codice funzionante piuttosto che alla documentazione.

6.3.1.1 Analisi dei requisiti

Un esame critico ed attento condotto sulle metodologie di sviluppo dei sistemi multi-agente, in generale, e sulla metodologia PASSI, in particolare, ne ha evidenziato alcuni limiti e problemi; tra i più importanti la difficoltà degli progettisti ad imparare l’uso della metodologia in poco tempo ed in molti casi l’avversione alla produzione della gran mole di documentazione dettagliata che queste metodologie richiedono, a scapito della possibilità di concentrarsi principalmente sulla fase di programmazione.

Inoltre, facendo riferimento principalmente alla metodologia PASSI, si è notato che questa non è sufficientemente flessibile ai cambiamenti.

Poiché in molte applicazioni, e tra queste in particolare la robotica, si ha la necessità di trattare e risolvere alcuni problemi tipici del dominio in questione, i progettisti sono portati, molto spesso, a dedicare pochissimo tempo allo sviluppo di un progetto ordinato e sistematico, pur nella consapevolezza che la produzione di documentazione di buona qualità è di aiuto sia nello svolgimento di ogni fase del progetto che nel mantenimento dello stesso nel futuro.

Alla luce di tutto questo si è deciso di modificare la metodologia PASSI, per ridurne le limitazioni di cui si è parlato, e creare una metodologia agile mantenendo però le caratteristiche principali di quella di partenza.

I requisiti richiesti per lo sviluppo di PASSI Agile sono:

- Sviluppo del progetto nel minor tempo possibile con un processo snello che dia la possibilità di concentrarsi subito sulla produzione di codice da testare e mostrare al cliente. Così facendo si ha la possibilità di verificare subito la rispondenza di quanto si è fatto ai bisogni ed ai desideri del cliente.
- Riutilizzo di parti di metodologie già esistenti, i cosiddetti *fragment*, soprattutto della metodologia di partenza, PASSI. Infatti si è già detto che le metodologie agili non sono

delle vere e proprie metodologie ma si basano molto su quelle già esistenti ed utilizzate prendendone e cambiandone solo le parti utili allo scopo.

- Riutilizzo di parti di progetti già fatti ed applicati per fare in modo che i progettisti impieghino il minimo sforzo nella soluzione dei problemi loro proposti. L'utilizzo di componenti software, algoritmi, progetti o qualunque altra forma di esperienza sviluppata nell'ambito della risoluzione di uno specifico problema porta alla riduzione dei tempi di sviluppo e di testing dei componenti. Questo requisito è fondamentale in quei campi, per esempio la robotica, come detto prima, nei quali c'è una ricorrenza costante di alcuni specifici problemi che risolti la prima volta possono essere sempre riutilizzati.
- Produzione di documentazione possibilmente a costo zero. Le metodologie agili non sono completamente contrarie alla produzione di documentazione; e poiché nei nostri domini applicativi produrre documentazione dà dei vantaggi in fase di mantenimento del prodotto e, soprattutto, permette di divulgare le conoscenze acquisite, uno dei requisiti che ci si è proposti di soddisfare è quello di produrre documenti nel minor tempo possibile e con il minore sforzo possibile.

6.3.1.2 Processi Agili

Le metodologie classiche impongono allo sviluppo del software un processo disciplinato, con lo scopo di renderlo più prevedibile e più efficiente attraverso un processo dettagliato che dà grande enfasi alla pianificazione.

In reazione a queste metodologie, negli ultimi anni è comparso un nuovo gruppo di metodologie, conosciute per un breve periodo come metodologie leggere, ma adesso più note con il termine metodologie agili **[36]**.

La differenza più immediata tra le due è la minore quantità di documentazione prodotta per ogni progetto; esse sono infatti più orientate al codice, poiché adottano il punto di vista secondo il quale l'elemento chiave della documentazione è il codice sorgente.

La produzione di una grande quantità di documentazione durante l'applicazione di una metodologia ha come conseguenza una notevole limitazione nell'affrontare i cambiamenti dei requisiti del progetto da sviluppare.

Un modo molto efficace per tenere sotto controllo i continui cambiamenti è procedere, nella modellazione, attraverso piccoli incrementi provando tutto con il codice, quindi uno sviluppo iterativo.

L'idea dello sviluppo iterativo in realtà non è completamente nuova ed il suo risvolto pratico fondamentale è quello di realizzare continuamente dei sottosistemi funzionanti che non hanno, ancora, tutte le funzionalità del sistema finale ma una volta testati attentamente ed integrati dovrebbero rispettarne tutte le caratteristiche.

Inoltre lo sviluppo iterativo fornisce ad ogni iterazione un solido fondamento sul quale è possibile basare le successive pianificazioni.

6.3.1.3 Riuso di pattern

Uno dei modi migliori per abbassare i tempi, ed i costi, per lo sviluppo del codice è fornire al programmatore un supporto per l'automazione della produzione di più codice possibile.

A tale scopo viene introdotto il concetto di Pattern nell'accezione di Christopher Alexander: *ogni pattern rappresenta un problema che si presenta più e più volte nel nostro ambiente e ne descrive il nocciolo della soluzione.*

L' utilizzo ed il riuso di un pattern contribuisce a migliorare significativamente la qualità del progetto e ad aumentare la quantità di codice prodotta in poco tempo.

Per gli scopi di questo lavoro si usano i cosiddetti pattern di agenti che sono entità formate da due viste, quella progettuale ed una implementativa.

Per quanto riguarda la prima si deve dire che si considera un agente come composto da due parti, una strutturale ed una comportamentale.

La struttura di un agente è fatta da una classe principale e da un insieme di classi che rappresentano i suoi task, il suo comportamento è costituito, invece, da un diagramma dinamico, un activity o uno state chart.

Da un punto di vista strutturale si hanno quattro classi di pattern:

- Action pattern – una funzionalità del sistema che può essere un metodo o anche un classe agente o una classe task
- Behaviour pattern – uno specifico comportamento dell'agente.
- Component pattern – un pattern di agente che comprende l'intera struttura di una agente compresi i suoi task
- Service pattern – una collaborazione tra due o più agenti, è una aggregazione di componenti.

L'utilizzo di un pattern è qualcosa di simile ad un fragment , nel secondo caso si tratta di parti di metodi o di processi, nel primo di porzioni di codice e relativi diagrammi di progetto, mentre un fragment viene estratto da una metodologia già esistente, un pattern spesso può essere il risultato di un'analisi svolta su applicazioni precedenti, nelle quali possono essere identificate alcune soluzioni, o parti di esse, che vengono quindi generalizzate per il riuso in altri contesti applicativi.

Possiamo, quindi, identificare specifici pattern da specifici campi applicativi, per esempio, nella robotica, pattern di task che rappresentano comportamenti abituali come la pianificazione di un percorso o l'evitare un ostacolo.

Inoltre le funzionalità dei pattern possono essere ulteriormente suddivise in quattro categorie:

- Mobilità – descrive la possibilità di un agente di muoversi da una piattaforma ad un'altra mantenendo la sua conoscenza.
- Comunicazione – rappresenta la comunicazione tra due agenti tramite un protocollo di interazione
- Elaborazione – sono pattern usati per rappresentare funzionalità, di un agente, di elaborazione su una grande quantità di dati
- Accesso alle risorse locali – pattern adibiti al recupero di informazioni ed alla manipolazione di strema di dati provenienti da dispositivi hardware o da sensori telecamere etc.

Per quanto riguarda il supporto all'automazione, abbiamo preso come riferimento un repository di pattern di agenti : Agent Factory.

Questo è un tool per la gestione di agenti e di pattern per agenti, mediante il quale si può sviluppare un sistema multi agente progettando la struttura ed il comportamento degli agenti ex novo oppure facendo riferimento ad un catalogo di pattern, a cui si può accedere per aggiungere funzionalità ad un agente.

6.3.1.4 Motivazioni per la scelta dei frammenti

Per la scelta e l'assemblaggio dei frammenti sono state tenute in considerazione, oltre alle regole suddette, quelle che sono le fasi principali ed imprescindibili del processo di sviluppo di un sistema software in generale e di un sistema multi-agente in particolare.

Per un sistema software si deve procedere alla raccolta ed analisi dei requisiti, corrispondente alla fase di modellazione del domino del problema, al system ed all'object design, dominio della soluzione, ed alla implementazione.

In più in un sistema multi-agente si devono modellare la conoscenza dell'agente e le sue interazioni sociali.

Scopo principale di PASSI Agile è arrivare il più velocemente possibile dai requisiti al codice degli agenti, cui corrispondono le funzionalità del sistema, evitando la produzione di documentazione che rallenta ed appesantisce il processo.

Il tutto sfruttando il più possibile i concetti di iteratività ed incrementalità, attraverso i quali si può suddividere il dominio del problema in sotto domini ed affrontare e risolvere poco alla volta i problemi consegnando in breve tempo all'utente pezzi di sistema funzionanti con i quali egli può interfacciarsi per apportare eventuali modifiche ai requisiti del sistema.

La scelta di andare direttamente al codice non significa che sono state abolite completamente tutte le fasi produzione di buona documentazione, ma abbiamo preferito porre più attenzione al codice ed al testing, come prevedono le norme della XP [41].

Tutto quello che in PASSI era documentazione si è cercato di produrlo a costo zero con l'ausilio di una piccola applicazione che, a partire da un tool per il riuso dei pattern di agenti, estrae tutte le informazioni per generare automaticamente la detta documentazione.

L'obiettivo è che il progettista focalizzi la sua attenzione su codice e test in modo da produrre nel più breve tempo possibile un prodotto funzionante, a partire dalle specifiche dei requisiti del sistema.

Si è, allora, snellita notevolmente tutta la parte riguardante la modellazione del dominio della soluzione, demandando a questa più una funzione di mera documentazione del progetto svolto che di ausilio al progetto in sé.

Invece le fasi riguardanti la modellazione del dominio del problema sono rimaste quasi intatte, in quanto necessarie per il processo di sviluppo e per la produzione di codice.

In questo caso ci si limita a consigliare di ridurre se non addirittura di eliminare la fase di stesura dei casi d'uso, sostituendola con una fase, più confacente alle metodologie agili, di

comunicazione tra cliente, utenti e sviluppatori/progettisti per tracciare quelle che devono essere le funzionalità del sistema sotto esame e prevedere una prima possibile soluzione.

6.3.1.5 I frammenti scelti

Lo scopo di questo lavoro è di riesaminare la metodologia PASSI, usando i principi e le tecniche delle metodologie agili, per creare una nuova metodologia più semplice e facile da usare ed orientata principalmente alla produzione di codice piuttosto che di documentazione.

Per sviluppare la nuova metodologia si fa riferimento ai valori principali della Agile Alliance (http://www.agilealliance.org) :

- Individui e interazioni piuttosto che processi e strumenti.
- Software funzionante piuttosto che documentazione completa.
- Collaborazione degli utenti piuttosto che negoziazione contrattuale.
- Risposta ai cambiamenti piuttosto che esecuzione di un piano.

e si applicano i principi ed alcune norme di una delle più usate tre le metodologie agili, la eXtreme Programming , le cui regole principali sono:

- planning
- designing
- coding
- testing

Di questi viene, sicuramente, trattato con maggiore enfasi il testing; sebbene molti metodi menzionino il testing, la maggior parte di essi lo sviluppa in maniera superficiale, mentre la XP pone il testing come fondamento dello sviluppo; ciascun programmatore scrive i test parallelamente al codice richiesto per la produzione.

I test sono integrati in un processo di integrazione continua e di costruzione, il che porta ad ottenere una piattaforma altamente stabile per lo sviluppo futuro.

Tenendo in considerazione quanto detto nei paragrafi precedenti e seguendo l'approccio offerto dalla method engineering sono stati assemblati alcuni frammenti dalla metodologia PASSI, quelli che sono stati considerati essenziali e distintivi per una metodologia PASSI che fosse agile; essi sono :

- raccolta dei requisiti - requirement elicitation
- identificazione società agente – agent identification, domain ontology description
- codifica – code reuse
- testing – test

Questi frammenti sono stati scelti definiti ed assemblati seguendo il processo illustrato nella figura seguente e con l'aiuto di un CAME (Computer Aided Method Engineering tool) che permette la composizione di nuove metodologie a partire da frammenti di altre e fornisce un modo veloce e potente per implementare CASE tool di supporto per la propria metodologia.

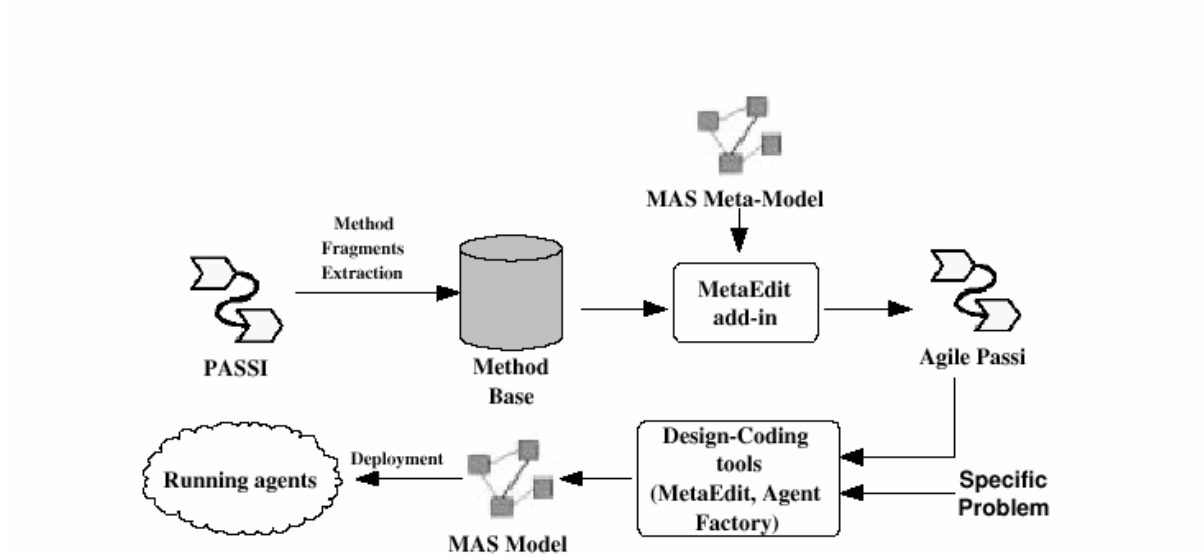


Figure 6-7 Il processo di costruzione di PASSI Agile

Si nota che, a partire dalla metodologia PASSI alcuni frammenti vengono estratti e memorizzati in un data base (Method Base) dal quale poi vengono estratti ed assemblati nella nuova metodologia con l'aiuto di un add-in di MetaEdit; la nuova metodologia è poi applicata ad un problema specifico e tramite dei tool, per l'ausilio al progetto e per la produzione del codice, vengono istanziati gli elementi del MAS metamodel, dando così luogo al sistema

voluto, i cui agenti vengono successivamente allocati su uno o più nodi di elaborazione per l'esecuzione.

6.3.2 Descrizione della metodologia PASSI Agile

A partire dai requisiti individuati nelle fasi precedenti e dai frammenti scelti, la nuova metodologia PASSI Agile è stata assemblata secondo lo schema in figura 5.2 dove le quattro fasi individuate nel precedente paragrafo sono rappresentate utilizzando la notazione SPEM.

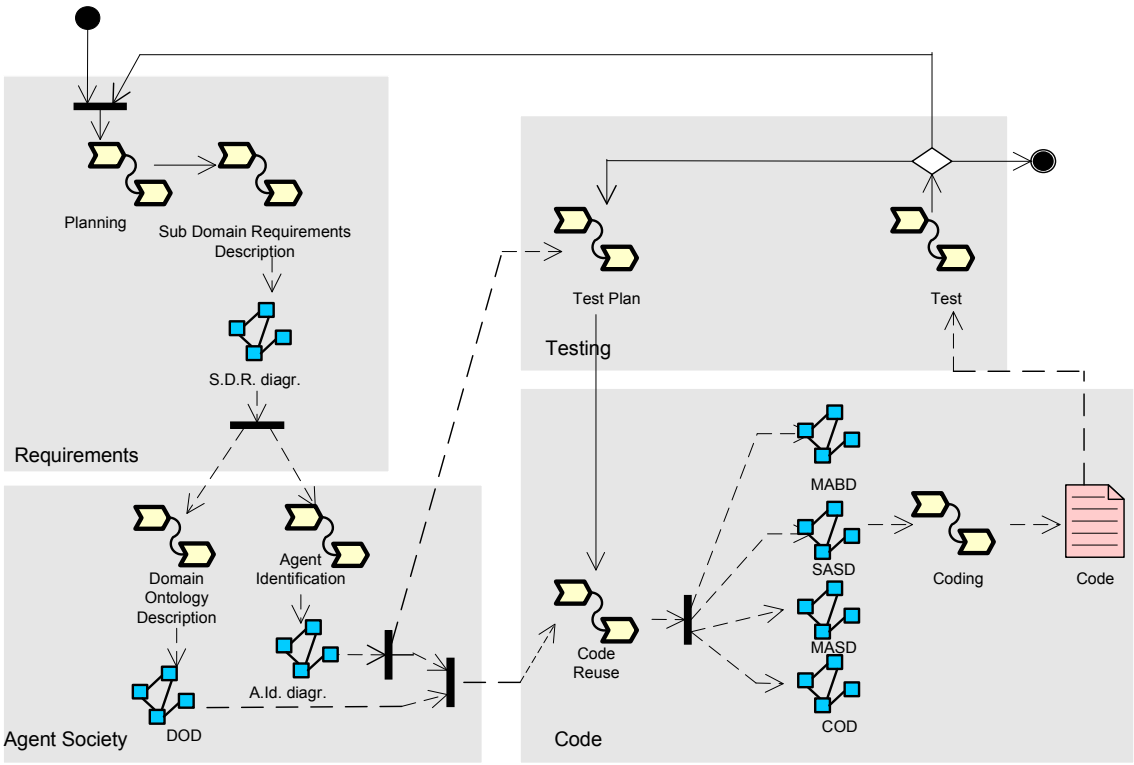


Figure 6-8 Il processo PASSI Agile

Possiamo distinguere, allora, quattro modelli:

- **requirements** , un modello dei requisiti del sistema composto da due parti: Planning e Sub Domain Requirements Description.
- **agent society**, un modello degli agenti che intervengono nella soluzione, le loro interazioni e la loro conoscenza sul mondo circostante. È composto da due parti: Domain Ontology Description e Agent Identification.
- **code**, la fase di produzione del codice degli agenti coinvolti nella soluzione del problema proposto. È suddiviso in due parti Code Reuse e Coding.
- **testing**, la preparazione e l'esecuzione di attività di test sui componenti. È composta da due parti Test Plan e Testing.

6.3.2.1 Il modello "Requirements"

In questo modello attraverso due passi successivi :

- Planning – dove progettisti, sviluppatori, utenti e clienti stabiliscono le funzionalità che il sistema deve avere
- Sub domain requirements description – una descrizione funzionale attraverso diagrammi dei casi d'uso. I termine *sub* si riferisce alla possibilità di suddividere il sistema in sotto_sistemi.

si procede all'analisi ad alto livello del sistema.

Per la realizzazione di questa fase abbiamo fatto pesantemente riferimento ad una delle norme principali della eXtreme Programming della quale abbiamo poi mantenuto il nome, appunto, "Planning".

Tra le regole di questa norma le prime due si adattano perfettamente al nostro caso : *release planning* ed *user stories*; la prima suggerisce che il team si incontri per stimare i tempi di sviluppo in funzione del problema, per prendere decisioni tecniche e commerciali e per schedulare le parti di progetto che ogni elemento del team deve svolgere. Tutto questo viene stimato attraverso quelle che si chiamano 'user stories' che sono qualcosa di simile ai casi d'uso, vengono scritte dal cliente per rappresentare le cose che il sistema deve fare per loro.

L'approccio adottato, in realtà, si mantiene in una via di mezzo tra la release planning dell' XP e la fase di raccolta dei requisiti delle metodologie classiche.

Questo riduce notevolmente la quantità di documentazione prodotta, soprattutto in questa prima fase, dove una raccolta veloce dei requisiti si traduce in una ridotta probabilità che il cliente cambi idea su una o più funzionalità del sistema ed in un più veloce rilascio di parti funzionanti dello stesso.

Inoltre le persone coinvolte in questa fase, se lo ritenessero necessario, potrebbero suddividere il problema in sotto problemi in modo da sfruttare appieno le caratteristiche di incrementalità ed iteratività proprie delle metodologie agili più che di altre.

Queste due caratteristiche sono rappresentate nel diagramma dai due cicli, infatti per rilasciare continuamente, ed al più presto possibile, codice funzionante si potrebbe iterare lo stesso processo (dai requisiti al codice) , o anche solo piccole porzioni di processo, su parti del sistema, o meglio su differenti funzionalità che il sistema deve fornire.

6.3.2.1.1 “Planning”

Si è scelto di privilegiare, nella fase di raccolta dei requisiti, la comunicazione tra gli elementi del team di sviluppo ed evitare la fase di stesura di tutti i casi d’uso, passando direttamente da una fase di pianificazione delle strategie e di valutazione degli elementi, che si hanno a disposizione per la soluzione del problema , ad una di rappresentazione del dominio del problema. La fase di pianificazione delle strategie comprende anche una analisi dei rischi ed una schedulazione delle attività da svolgere.

Ricordiamo che la comunicazione tra gli appartenenti lo stesso team è uno dei valori fondamentali delle metodologie agili.

Questa fase potrebbe avere come risultato la stesura di un documento di riepilogo sulle considerazioni fatte e sulle soluzioni proposte dal team; si lascia in questa fase piene autonomia di azione al progettista perché come detto in precedenza una metodologia agile non abolisce rigorosamente la documentazione.

Il team di sviluppo in questa fase deve essere rigorosamente formato da progettisti, sviluppatori, utenti e cliente.

6.3.2.1.2 "Sub Domain requirements description"

Ha come risultato un diagramma dei casi d'uso classico per una descrizione funzionale dei requisiti del sistema. Questo diagramma viene compilato iterativamente in più fasi, da qui il prefisso *sub*, nel senso che il sistema può essere suddiviso in sotto_sistemi dei quali si traccia il diagramma dei casi d'uso.

Ad ogni iterazione si incrementa la descrizione del sistema aggiungendo funzionalità.

Di seguito un esempio di diagramma parzialmente riempito (fa riferimento al progetto di cui in appendice):

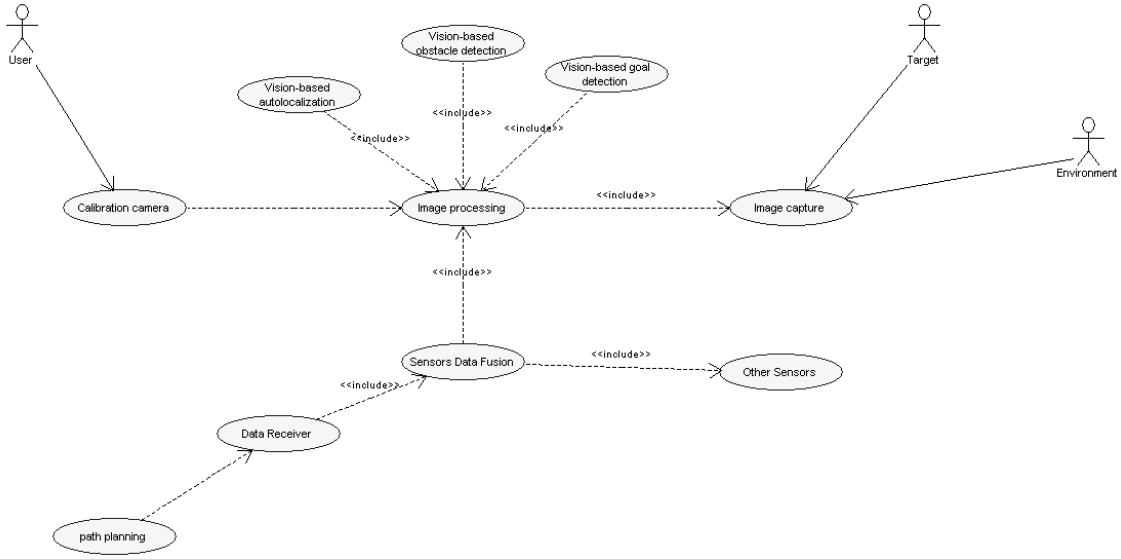


Figure 6-9 Sub Domain Description diagram

6.3.2.2 Il modello "Agent Society"

In questo modello il progettista procede con una identificazione degli agenti coinvolti nella soluzione, suddividendo le funzionalità del sistema, precedentemente tracciato, in unità

logiche, ognuna descritta da un package, ed alla definizione dell'ontologia relativa alla conoscenza dell'agente.

Il modello è composto da due parti:

- Agent Identification
- Domain Ontology Description

Si prevede che queste due fasi vengano svolte quasi parallelamente; subito dopo l'identificazione di ogni agente si procede alla definizione della sua conoscenza.

Anche questo modello viene quindi generato con successive iterazioni

6.3.2.2.1 "Agent Identification"

Il diagramma prodotto in questa fase è un diagramma dei casi d'uso, praticamente identico al precedente, nel quale ogni package è composto da uno o più use cases del diagramma "Sub Domain Decsription" e gli viene associato un agente del sistema (il nome del package corrisponde con quello dell'agente); ad ogni agente vengono, così, assegnate delle responsabilità in termini di requisiti da soddisfare.

In questo diagramma le relazioni tra casi d'uso di package differenti sono indicate con lo stereotipo di 'communication', in quanto danno una prima indicazione sulle interazioni della società di agenti, ed il simbolo convenzionale adottato è una freccia che va dall' agente che inizia la comunicazione a quello che ne partecipa.

Di seguito un esempio di diagramma :

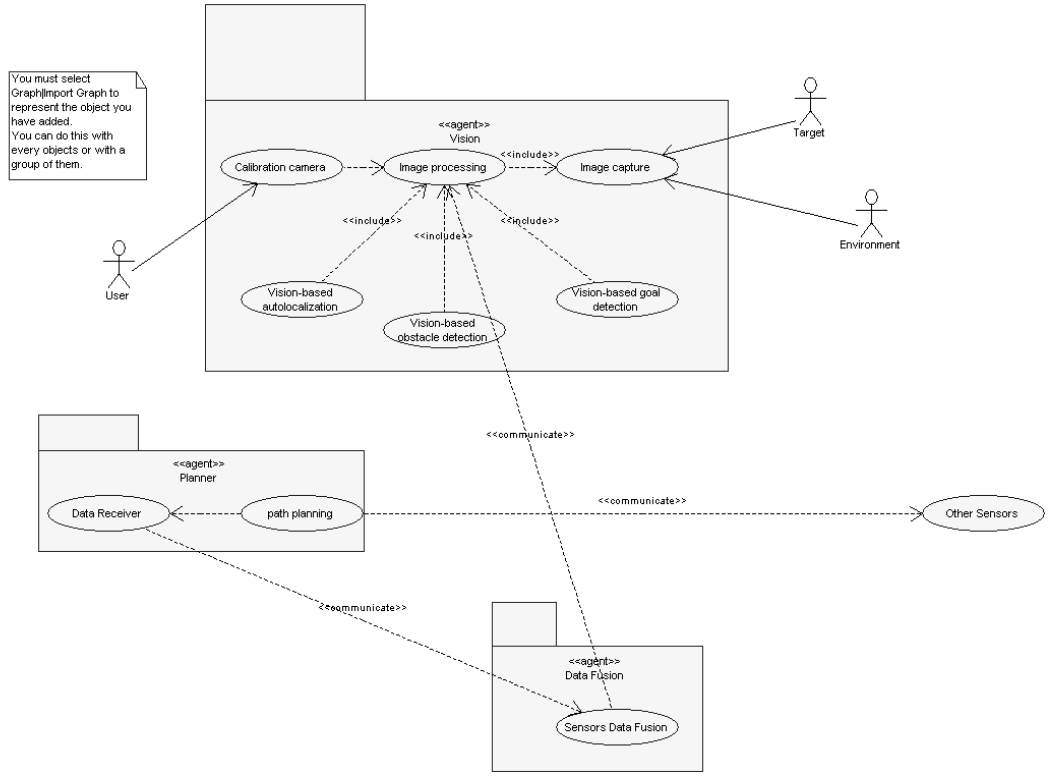


Figure 6-10 Agent Identification Description Diagram

6.3.2.2.2 "Domain Ontology description"

In questa fase si usa un diagramma delle classi per rappresentare l'ontologia del sistema, le entità coinvolte sono rappresentate attraverso classi, di colori differenti, esse sono: concetti (colore giallo), predicati (celeste) e azioni (bianco). Le relazioni tra questi tre elementi sono le tre UML standard, generalizzazione, aggregazione ed associazione

Come si diceva prima questa fase procede in parallelo alla precedente; man mano che si identificano le entità, gli agenti, cui assegnare le varie funzionalità del sistema è bene individuare e rappresentare la conoscenza che queste devono avere del dominio.

Di seguito un esempio di diagramma di "Domain Ontology Description":

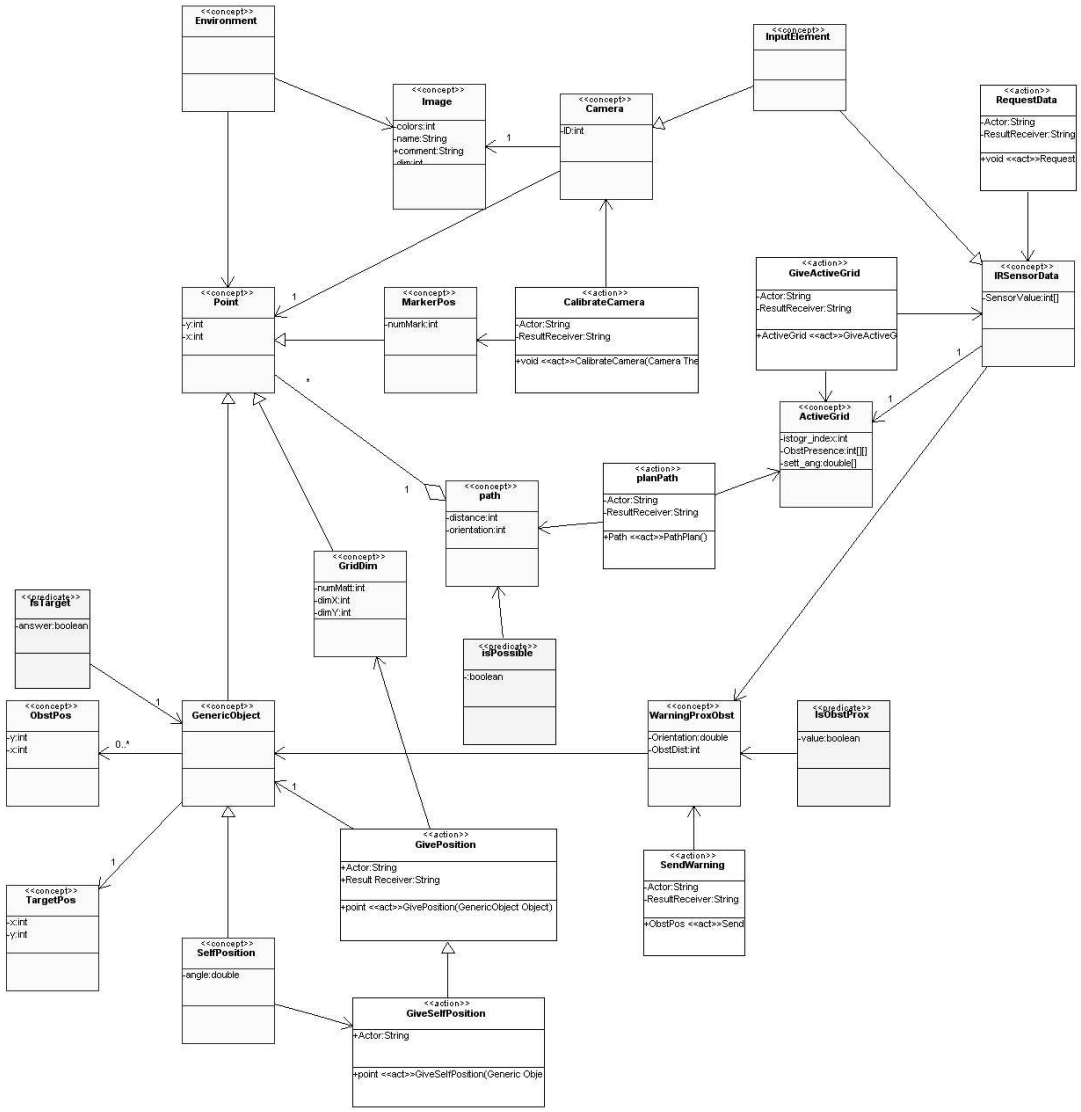


Figure 6-11 Domain Ontology Description diagram

6.3.2.3 Il modello “Code”

Il modello è composto da due parti strettamente connesse tra loro:

- Code Reuse – fase nella quale si tenta di riusare codice sviluppato ed utilizzato in applicazioni precedenti.

- Coding – dove si completa la parte del codice, eventualmente mancante, della fase precedente.

6.3.2.3.1 Code Reuse

In questa fase si cerca di riutilizzare pattern di agenti memorizzati in un repository, come detto nel paragrafo 5.3.2 ed inoltre vengono prodotte porzioni di codice riusabile documentate attraverso i loro diagrammi strutturali e comportamentali (MABD e SASD) con l'aiuto di un tool già esistente, *Agent Factory.*

Questo tool permette di creare una società multi-agente progettando nuovi agenti o facendo riferimento ad un *repository* di pattern per aggiungervi delle funzionalità, fornisce un supporto per la produzione automatica di più codice possibile ed offre la possibilità di effettuare il riverse engineering su tutto il codice aggiunto o modificato manualmente (nella fase di "Coding").

Inoltre in questa fase si ha a disposizione una applicazione di supporto, appositamente sviluppata in questo lavoro, per la creazione automatica di tutta la documentazione necessaria per modellazione del problema.

A partire dalle informazioni memorizzate nel diagramma "Agent Identification" e dai modelli comportamentali e strutturali generati da Agent Factory, questa applicazione genera automaticamente i seguenti documenti:

- Communication Ontology Description diagram – un diagramma delle classi composto da due elementi, gli agenti e le comunicazioni. Le classi che rappresentano gli agenti hanno attributi che si riferiscono alla conoscenza dell'agente; in quelle che rappresentano la comunicazione vi è l'indicazione sull'ontologia condivisa tra i due agenti che effettuano la comunicazione, il linguaggio ed il protocollo usati in questa comunicazione.
- (M)ASD diagram – (Multi) Agent Structure Description diagram, un diagramma delle classi che mette in evidenza l'architettura generale del sistema. Ogni classe simbolizza un agente del sistema, gli attributi sono usati per rappresentare la conoscenza dell'agente, i metodi rappresentano i task e le relazioni rappresentano il flusso di informazioni scambiate tra gli agenti (le comunicazioni).

- SASD diagram – Singol Agent Structure Definition , un diagramma delle classi per ogni agente è usato per rappresentare la struttura interna del singolo agente e di tutti i suoi task.
- (M)ABD diagram – Multi Agent Behaviour Description diagram, un activity diagram che rappresenta il flusso di eventi tra le classi agente principali

6.3.2.3.2 Coding

Come detto prima si completa la parte del codice mancante.

In questa fase si mettono in pratica tutte le regole e le norme proprie della XP per tirare fuori codice funzionante nel più breve tempo possibile.

6.3.2.4 Il modello "Testing"

La fase di testing è composta da due parti:

- Test Plan – si preparano i test dei componenti alla luce di quanto sviluppato nella fase di "Agent Identifiaction".
- Test – si esegue il test sui componenti codificati

6.3.2.4.1 Test Plan

L'attività di test è stata eseguita facendo stretto riferimento ai principi del manifesto agile ed in particolar modo alle norme dell' *eXtreme Programming*, il processo agile che si prende come riferimento; in base a dette regole il *testing* deve cominciare prima di procedere alla programmazione vera e propria dei componenti, il programmatore deve preparare uno o più test che il componente deve soddisfare alla fine della fase di codifica (*coding*).

In questo modo si tiene sotto costante controllo il lavoro del programmatore infatti se il test su un componente fallisce questo viene sottoposto a rifinimento e rielaborazione fino a quando il test non è soddisfatto.

6.3.2.4.2 Test

Dopo il completamento dello scheletro del codice prodotto da Agent Factory si procede al test vero e proprio di uno o più componenti; se questa fase termina con successo viene rilasciata una prima versione di componente (agente in questo caso) funzionante che può fungere pure da prototipo per verificare se i desideri ed i bisogni dell'utente sono stati soddisfatti, se così non fosse si ritorna alla fase di "Planning" per rifinire i requisiti individuati la prima volta o per introdurne di nuovi.

7. Conclusioni

Negli ultimi anni molte metodologie di progetto per sistemi multi-agente sono state utilizzate con buoni risultati; recentemente si è avuta la necessità di processi più versatili e veloci. Da qui l'idea di presentare una nuova metodologia, PASSI Agile, che è stata concepita per avere un processo adatto ad alcuni specifici contesti applicativi.

La costruzione della nuova metodologia è partita da una attenta analisi dei requisiti, dallo studio dei processi agili e della disciplina della *method engoneering* e dal conseguente riuso dell'esperienza fatta sulle metodologie già esistenti, ed in particolar modo di PASSI, concretizzatosi sotto forma di frammenti riassemblabili.

PASSI Agile è stata dotata di un tool di supporto, concepito come *add.in* di un altro tool , MetaEdit+ della MetaCase, scelto per alcune sue funzionalità interessanti,che tramite interfacciamento con un altro tool ancora permette la generazione automatica di buona parte del codice e di tutta la documentazione del progetto.

Una prima valutazione della nuova metodologia e delle funzionalità del tool è stata fatta tramite un esempio di una applicazione seguendo l'approccio proposto in [47]; a fine lavoro si sono potuti fare alcuni interessanti confronti:

1. lo sviluppo di questa applicazione con la metodologia PASSI classica ha richiesto due settimane per il progetto ed una per la fase di programmazione e di testing, il tutto risultante nella produzione di circa cinquemila linee di codice ;
2. l'applicazione sviluppata con PASSI Agile ha richiesto in tutto due settimane di lavoro ed il progetto completo è stato ottenuto dopo quattro iterazioni alla fine di ognuna delle quali è stato presentato un prototipo funzionante della porzione di problema affrontato.
3. Durante la fase di "Code Reuse" l'utilizzo di pattern e la generazione automatica di codice hanno portato ad una quantità di riuso del codice di circa il 50%, riducendo, così, notevolmente il lavoro del programmatore; inoltre sono stati prodotti automaticamente più della metà dei documenti, con ovvia riduzione dei tempi di progettazione.

Nel futuro si cercherà di migliorare la possibilità di utilizzo del tool, del quale si è fatto l'*add-in* ,poiché è richiesta una buona conoscenza dello stesso da parte del progettista e non è ancora stato risolto il problema del posizionamento degli oggetti.

Il lavoro che è stato fatto si presta, inoltre, ad ulteriori sviluppi futuri; infatti la funzionalità offerta dal tool di produrre i documenti di progetto sotto forma di file XML, od anche la modellazione iniziale mediante UML, lascia ampio spazio allo studio ed alla ricerca di formati intermedi di rappresentazione che permettano di passare da una rappresentazione XML, o UML, ad una in linguaggi di descrizione hardware per implementare così su dispositivi hardware quegli agenti che ad oggi sembrano la soluzione migliore ad alcuni specifici problemi.

Il raggiungimento degli obiettivi posti è anche testimoniato dal positivo riscontro avuto nella comunità scientifica degli articoli che presentano sia Passi Agile [4]che il processo (basato sulla method engineering) che ha portato alla composizione della metodologia [1] Altrettanto interessanti dal punto di vista scientifico sono stati anche i risultati degli studi fatti sui pattern che hanno portato alle pubblicazioni riportate in [5][6][7].

Bibliografia

[1] M. Chen, J. F. Nunamaker, E. S. Weber, “Present status and future directions”, Computer-aided software engineering, 1989.

[2] B. Bruegge, A. H. Dutoit, Conquering complex and changing systems, Object Oriented Software Engineering, Peritence Hall, 2002.

[3] M. Cossentino, V. Seidita. Composition of a New Process to Meet Agile Needs Using Method Engineering. Software Engineering for Large Multi-Agent Systems vol. III. LNCS Series, Elsivier Ed. (2004). Invited Paper. (in printing)

[4] Chella, M. Cossentino, L. Sabatucci, and V. Seidita. From passi to agile passi: Tailoring a design process to meet new needs. In 2004 IEEE/WIC/ACM International Joint Conference on Intelligent Agent Technology (IAT-04), Beijing, China, Sept 2004.

[5] A. Chella, M. Cossentino, and L. Sabatucci. Tools and patterns in designing multi-agent systems with passi. WSEAS Transactions on Communications, 3(1):352-358, Jan 2004.

[6] M. Cossentino, L. Sabatucci, and A. Chella. Patterns reuse in the PASSI methodology. In Engineering Societies in the Agents World IV, 4th International Workshop, ESAW 2003, Revised Selected and Invited Papers, volume XIII of Lecture Notes in Artificial Intelligence. Springer-Verlag, 2004.

[7] A. Chella, M. Cossentino, and L. Sabatucci. Designing jade systems with the support of case tools and patterns. Exp Journal, 3(3):86-95, Sept 2003.

[8] G. Succi, “L’evoluzione dei linguaggi di programmazione: analisi e prospettive”, Mondo Digitale, 4, dicembre 2003.

[9] A. Natali, “La costruzione del software: dalle tecnologie ai modelli”, 2003.

[10] J. Iivari, R. Hirschheim, H. K. Klein, “Beyond Methodologies: Keeping up with Information Systems Development”, Approaches through Dynamic Classification, 1999.

[11] M. Wooldridge, P. Ceccarini, “Agent-Oriented Software Engineering: The state of the Art”, Springer-Verlang, Berlin . 2001.

[12] N. R. Jennings, “On agent-based software engineering in Artificial Intelligence”, 2000.

[13] J. Odell, "Objects and Agents Compared," Journal of Object Technology, Vol 1, Number 1, May, 2002

[14] P. Ciancarini, A. Omicidi, F. Zambonelli, “Multiagent System Engineering: the Coordination Viewpoint”, Lecture Notes In Computer Science (LNCS), 6th International Workshop on Intelligent Agents VI, Agent Theories, Architectures, and Languages (ATAL), 1999. Springer-Verlag.

[15] M. Cossentino and L. Sabatucci. “Agent System Implementation” in “Agent-Based Manufacturing and Control Systems : New Agile Manufacturing Solutions for Achieving Peak Performance”. M. Paolucci, R. Sacile Editors. CRC Press. ISBN: 1574443364. April 2004

[16] Wooldridge, M., Jennings N.R. , Kinny D. (1999).A Methodology for Agent-Oriented Analysis and Design. Proceedings of the Third International Conference on Autonomous Agents (Agents'99)

[17] W. Harrison, H. Ossher, P. Tarr, “Software Engineering Tools and Environments:A Roadmap”, International Conference on Software Engineering (ICSE) 2000. Limerick, Ireland.

[18] P. Coad, E. Youdon, Object-Oriented Desing, Yourdan Press, 1996.

[19] J. K. Tolvanen, "Incremental Method Engineering with Modeling Tools", Theoretical Principles and Empirical Evidence, Thesis, University of Jyväskylä, Lievestuore, 1998.

[20] B. Chandrasekaran, J. R. Josephson, V. R. Benjamins, “What Are Ontologies, and Why Do We Need Them?”, IEEE Intelligent Systems, Volume 14 , Issue 1 (January 1999)

[21] GAMMA, E. – HELM, R. – JOHNSON, R. – VLISSIDES, J., “Designer patterns: elements of reusable object-oriented software”, Addison-Wesley, Boston, (1995).

[22] ARIDOR, Y. – LANGE, D.B. – “Agent designer patterns: elements of agent application design“, in “Proceedings of autonomous agents”, ACM Press, (1998).

[23] DEUGO, D. - WEISS, M. - KENDALL, E. – “Reusable Patterns for Agent Coordination”, in OMICINI, A.- (2001) “Coordination of Internet Agents”, Springer, (2001).

[24] LAVANDER, G. – SCHMIDT, D. – “Active object: an object behavioural pattern for concurrent programming” in VLISSIDES, J.M. – COPLIEN, J.O. – KERTH, N.L. – (1996) “Pattern languages of program design”, Addison-wesley, Boston, (1996).

[25] KENDALL, E. A. - MURALI KRISHNA P.V. - CHIRAG.V.PATHAK - SURESH, C.B. “Patterns of intelligents and mobile agents”, in Proc. of the Second International Conference On Autnomous Agent, Minneapolis, May 1998.

[26] CARLSON, D. – (2001) “Modeling XML application with UML: pratical e-business application”, Addison-Wesley, Boston.

[27] BOOCH, G. et.al. – (1999) “UML for XML Schema Mapping Specification”, Rational Software White Paper.

[28] Han, J., Welsh, J., (1993) Methodology Modelling: Combining Software Processes with Software Products. Australian Computer Science Comm. Proc. 17th Annual Computer Science Conf., ACSC.

[29] Brinkkemper, S.,(1995) Method engineering: engineering the information systems development methods and tools.Information and Software Technology, 37, (11).

[30] Kumar, K., Welke, R.J., (1992) Methodology engineering: a proposal for situation-specific methodology construction. Challenges and Strategies for Research in Systems Development (eds. W.W. Cotterman, J.A. Senn), John Wiley & Sons Ltd, pp. 257-269.

[31] Saeki, M., (1994). Software specification & design methods and method engineering. International Journal of Software and Knowledge Engineering. World Scientific Public Company.

[32] Juha-Pekka Tolvanen,(1998) Incremental Method Engineering with Modeling Tools: Theoretical Principles and Empirical Evidence (Ph.D. thesis), Jyväskylä Studies in Computer Science, Economics and Statistics, Jyväskylä: University of Jyväskylä, 1998, p. 301.

[33] Ralyté, J., Rolland, C. (2001). An Approach for Method Reengineering. Lecture Notes in Computer Science, 2224 – H.S. Kunii, S. Jajodia, A. Sølvberg (Eds.): Conceptual Modeling - ER 2001 : 20th International Conference on Conceptual Modeling, Yokohama, Japan, November 27-30, 2001. Proceedings. Chapter: p. 471.

[34] Brinkkemper, S., Lyytinen,K., Welke,R.,(1996) Method Engineering: Principles of method construction and tool support. International Federational for Information Processing 65. August 1996 Hardbound pp.336.

[35] S. Brinkkemper, M. Saeki, and F. Harmsen, (1999) Meta-Modelling Based Assembly Techniques for Situational Method Engineering. Information Systems, Vol. 24, No. 3, pp.209 -228, 1999.

[36] Sito dell' Agile Alliance , http://www.agilealliance.org

[37] Manifesto del Movimento Agile, http://agilemanifesto.org

[38] Beck, K., Extreme Programming Explained: Embracing Change, Addison-Wesley,1999.

[39] Beck, K., Cockburn, A., Jeffries, R., and Highsmith, J. Agile Manifesto. http://www.agilemanifesto.org. 2001. 12-4-2002.

[40] Turk, D., France, R., Rumpe, B., (2002).Limitations of Agile Software Processes. Third International Conference on eXtreme Programming and Agile Processes in Software Engineering (XP 2002).

[41] Extreme Programming. A gentle Introduction http://www.extremeprogramming.org.

[42] Juan,T., Sterling, L., Winikoff, M., (2002). Assembling Agent Oriented Software Engineering Methodologies from Features. F. Giunchiglia et al. (Eds.): AOSE 2002, LNCS 2585, pp. 198–209, 2003.

[43] Method fragment definition. FIPA Document, http://www.fipa.org/activities/methodology.html, Nov2003.

[44] M. Cossentino, L. Sabatucci and V. Seidita. Method fragments from the PASSI process. Rapporto tecnico ICAR-CNR, (21-03), 2003.

[45] M. Cossentino, L. Sabatucci and V. Seidita. SPEM description of the PASSI process. Rapporto tecnico ICAR-CNR, (20-03), 2003.

[46] Software Process Engineering Metamodel, rel 1.0, http://www.omg.org/tecnology/documents/spec_catalog.htm

[47] Khanh Hoa Dam and Michael Winikoff. Comparing agent-oriented methodologies. In Fifth International Bi-Conference Workshop on Agent-Oriented Informa-tion Systems (AOIS-2003), Melbourne, Australia, 14 July 2003 2003.