

Costruire software

Il software, cioè l'insieme di frasi espresse in un qualche linguaggio al fine di istruire un elaboratore o una rete di elaboratori, non ha consistenza fisica; può consumare energia ed altre risorse, produrre effetti utili o dannosi, avere conseguenze rilevanti sul piano economico e sociale, ma è del tutto privo di massa.

Le conseguenze di questa caratteristica sono molteplici, sia sul piano pragmatico che sul piano filosofico. Limitando il discorso al contesto della produzione industriale, si è diffusa la convinzione che la costruzione del software non richieda, per sua natura, processi di produzione tipici dell'ingegneria tradizionale.

Nell'ingegneria tradizionale (meccanica, edile, etc) il costo del materiale costituisce spesso più del 50% del costo totale di un progetto, mentre nella produzione del software è il costo del lavoro ad essere preponderante: si parte dal 70% fino a giungere quasi al 100%. L'ingegneria tradizionale ha anche sperimentato che **un cambiamento di costo 1 in fase di analisi potrebbe costare 1000 in fase di produzione**. Per questo l'ingegneria classica diversifica le fasi di produzione delineando un ben noto flusso di lavoro (**workflow**) costituito da un insieme di passi (o **tasks**): **definizione dei requisiti, analisi del problema, progetto della soluzione, realizzazione del prodotto, collaudo** spesso eseguiti uno dopo l'altro, in un classico processo di sviluppo sequenziale o **a cascata**.

Nella produzione industriale del software è invece piuttosto comune cercare di abbattere i costi di progetto e di sviluppo, anche limitando le dimensioni del gruppo di lavoro, e aggredire il mercato prefissando una data di distribuzione del prodotto, che viene di frequente rilasciato non completamente finito, accollando all'utente parte dell'onere di collaudo. Sotto la spinta di stringenti vincoli di **time to market (TTM)** molte aziende adottano uno schema del tipo "**scrivi, prova e correggi**", mirando alla produzione di codice al minor "costo immediato" possibile. Le fasi di analisi e progetto anche se svolte, non sempre sono adeguatamente documentate, e quasi mai correlate in modo sistematico con il codice prodotto.

Il processo di costruzione del software è quindi influenzato da una potente forza, legata alla natura stessa del software: la spinta a impostare la costruzione di un prodotto in modo **bottom-up**, a partire da una specifica tecnologia costituita da un linguaggio di programmazione, o da un framework applicativo o da una piattaforma operativa.

La principale conseguenza negativa di questa forza è l'assenza di una esplicita descrizione di progetto che permetta di anticipare la valutazione dei rischi e le potenziali difficoltà connesse allo sviluppo. In molti casi adeguate fasi di analisi e di progettazione hanno luogo, anche **in modo sistematico**; ma ciò puttroppo quasi sempre accade solo nella mente dei costruttori; nel codice finale non vi è più traccia alcuna di queste fasi, se non qualche debole segnale legato a sporadici commenti.

Tuttavia, anche se il codice fosse accuratamente documentato sia in relazione all'analisi sia in relazione alle scelte di progetto, la riduzione del prodotto al solo codice sarebbe non accettabile, se non nel caso di sistemi software semplici. All'aumentare della complessità infatti, la mente umana ha bisogno, per comprendere, di decomporre il problema in parti di ampiezza limitata, articolando la descrizione in **livelli di astrazione diversi**; poiché il codice deve inevitabilmente esprimere il sistema finale nei suoi minimi dettagli, la maggior parte delle persone sarebbe incapace di leggerlo con profitto anche se a conoscenza delle regole sintattiche del linguaggio di programmazione.

La crisi del software

Impostare un processo di produzione in assenza di descrizioni del sistema che permettano di anticipare la valutazione dei rischi espone il processo stesso a un potenziale fallimento; non meraviglia dunque che dagli anni settanta-ottanta, si senta parlare di *crisi del software*.

La letteratura [Glass97], [[Software_engineering_disasters](#)] riporta casi di fallimento di un numero sorprendentemente rilevante di progetti software, evidenziando un insieme di cause principali:

- Cattiva specifica e gestione dei requisiti.
- Comunicazioni ambigue ed imprecise tra i diversi attori del processo di produzione (utenti, manager, analisti, progettisti, implementatori).
- Architetture finali del sistema fragili (non robuste).
- Inconsistenze tra requisiti, progetto e realizzazione.
- Collaudi inadeguati o insufficienti.
- Inadeguata capacità di valutare e gestire i rischi e di controllare la propagazione dei cambiamenti.

Queste potenziali fonti di insuccesso hanno amplificato la loro influenza nel momento in cui l'intera disciplina ha vissuto la transizione da una dimensione prevalentemente algoritmico-trasformazionale a un dimensione fortemente sistemistico-architetturale.

La dimensione algoritmica

Per molto tempo l'informatica ha coinciso con l'idea di una disciplina volta al progetto e alla realizzazione di processi di soluzione meccanizzabili detti *algoritmi*. Un algoritmo determina la trasformazione da un ingresso prespecificato che ubbidisce a *precondizioni* ad una uscita che ubbidisce a *postcondizioni*. Le proprietà caratterizzanti gli algoritmi sono le seguenti:

- correttezza;
- efficienza;
- il funzionamento osservabile non dipende dal tempo;
- l'elaborazione avviene eseguendo una sequenza finita di azioni interne al sistema;
- il flusso completo di una computazione può essere specificato *off-line*;
- un algoritmo è *chiuso*, cioè non soggetto a interazioni con l'esterno durante il suo funzionamento;
- il progetto può avvenire mediante un *raffinamento top-down* delle specifiche.

Nel progetto di un algoritmo, lo scopo è realizzare un processo che *termina*, producendo "nuova" informazione a partire da quella disponibile all'inizio della computazione. Problemi di natura algoritmica possono essere definiti da specifiche formali, sono implementati da programmi la cui correttezza può essere dimostrata in linea di principio e hanno una precisa nozione di complessità computazionale.

Il funzionamento di un algoritmo può essere specificato da un "contratto" simile a un contratto di vendita: i clienti forniscono un valore che soddisfa le *precondizioni* e si garantisce un risultato che soddisfa le *postcondizioni*.

La dimensione sistemistica

La esponenziale diminuzione dei costi degli elaboratori elettronici e la conseguente esponenziale diffusione dell'elaborazione dell'informazione nei più disparati settori applicativi, ha promosso una rapida evoluzione dalla originaria dimensione

computazionale ed algoritmica dell'informatica (nota anche come *programmazione in-the-small*) ad una dimensione più sistemistica. Oggi non si tratta solo di adeguare le metodologie e gli strumenti di produzione ad una forma di *programmazione in-the-large*, ma di affrontare il problema della progettazione e costruzione di veri e propri *sistemi software*, spesso molto articolati ed eterogenei, cioè da realizzarsi con più tecnologie e linguaggi.

Senza alcuna pretesa di introdurre qui una precisa definizione di *sistema* (ammesso che un'univoca ed accettata definizione possa esistere) osserviamo che, diversamente dal caso di un algoritmo, la costruzione di un *sistema software* implica la realizzazione di un ente caratterizzato dalle seguenti proprietà:

- ha un funzionamento osservabile descritto da "storie di interazione" che specificano i "messaggi" ricevuti e trasmessi;
- l'attività di elaborazione è spesso inclusa in un contesto di *interazione* tra il sistema e il mondo esterno; il sistema "calcola" attraverso un pattern di interazioni (*interaction history*) iniziate dall'esterno, spesso al di fuori del controllo del sistema stesso;
- opera on-line;
- è aperto;
- possiede sia proprietà trasformazionali sia proprietà temporali;
- ha come requisito primario il costo del ciclo di vita;
- viene spesso progettato partendo da una descrizione *bottom-up* dell'ambiente di interazione.

Queste proprietà possono essere ricondotte al fatto che le applicazioni software sono quasi sempre "situate", cioè collocate in precisi ambienti operativi con cui devono interagire.

La transizione dalla dimensione algoritmica a quella sistemistica è stata accompagnata da una evoluzione dei linguaggi di programmazione e delle metodologie proposte per l'analisi e la progettazione dei sistemi software (si veda [Breve storia delle metodologie](#)). Allo stato attuale questa transizione trova la sua sintesi nel *paradigma ad oggetti*, cui convergono i linguaggi di programmazione e le metodologie più utilizzati in ambiente industriale. La dimensione algoritmica non è certo scomparsa e il suo ruolo rimane importante; tuttavia la progressiva rilevanza della dimensione sistemistica ha indotto a porre in primo piano il concetto di *architettura di un sistema software* (si veda [Il ruolo dell'architettura](#)).

L'ingegneria del software

Sommerville definisce [Sommerville07] l'Ingegneria del Software come "*an engineering discipline that is concerned with all aspects of software production from the early stage of system specification to maintaining the system after it has gone into use.*" In questa definizione ci sono due frasi chiave:

1. *Engineering discipline* - gli ingegneri fanno funzionare le cose. Essi applicano teorie, metodologie e tool dove questi sono appropriati, ma li usano selettivamente e spesso cercano di scoprire soluzioni a problemi anche quando non ci sono teorie e metodologie applicabili a questi. Gli ingegneri inoltre riconoscono che devono lavorare sotto stretti vincoli organizzazionali e finanziari, così cercano soluzioni che rispettino anche tali vincoli.
2. *All aspects of software production* - l'ingegneria del software non si occupa solamente del processo tecnico dello sviluppo software, ma si occupa anche dello

sviluppo dei tool, delle metodologie e delle teorie per supportare la produzione software.

Riflettendo su questa definizione possiamo osservare che l'ingegneria viene considerata una scienza di acquisizione e applicazione di conoscenza rivolta all'analisi, progettazione e costruzione di prodotti dell'ingegno con scopi pratici. L'Associazione Americana degli Ingegneri per lo sviluppo professionale definisce l'ingegneria come *The creative application of scientific principles to design or develop structures, machines, apparatus, or manufacturing processes, or works utilising them singly or in combination; or to construct or operate the same with full cognizance of their design; or to forecast their behaviour under specific operating conditions; all as respects an intended function, economics of operation and safety to life and property.*

Gli ingegneri si basano su teorie dalla fisica e sulla matematica per trovare soluzioni adatte al problema da risolvere: essi applicano il metodo scientifico per derivare la soluzione migliore in presenza di opzioni multiple valutando le diverse scelte di progettazione e scegliendo quella che soddisfa i requisiti del sistema da sviluppare.

Compito cruciale dell'ingegnere è quindi identificare, capire e interpretare tutti i vincoli progettuali al fine di produrre un risultato di successo. Tali vincoli possono essere rappresentati dalla disponibilità delle risorse, da limitazioni fisiche o tecnologiche, flessibilità per future modifiche ed estensioni del prodotto, o altri fattori come per esempio requisiti di costo, sicurezza, marketing, produttività, durevolezza del prodotto. Attraverso la comprensione di questi vincoli gli ingegneri derivano la specifica dei limiti entro cui il nuovo sistema può essere prodotto e potrà poi funzionare.

Gli aspetti salienti legati al lavoro dell'ingegnere possono essere riassunti come segue:

- Seguire un chiaro e disciplinato processo di sviluppo.
- Adottare una metodologia di progettazione.
- Creare un modello (matematico) appropriato del problema che consenta di analizzarlo al meglio.
- Testare le potenziali soluzioni.
- Valutare le differenti scelte progettuali e scegliere quella che meglio incontra i requisiti degli utilizzatori.
- Usare: prototipi, modelli in scala, simulazioni, test distruttivi test non-distruttivi e stress test.

La produzione di software con un ben definito livello di qualità non è però abbastanza: c'è la necessità di modellare e ingegnerizzare anche il processo di sviluppo che deve essere controllabile e ben documentato (si veda [Capability Maturity Model](#)).

Questa necessità richiede la presenza di astrazioni, processi, metodologie e strumenti a supporto di tutto il processo di ingegnerizzazione. Il software ha a che fare con entità che hanno tipicamente una controparte reale come i numeri, le date, i nomi, le persone, i documenti e così via. Queste entità devono essere modellate da opportune *astrazioni*. Le astrazioni rappresentano i "blocchi di costruzione" dell'ingegnerie del software per creare i propri prodotti e dipendono dalle tecnologie disponibili: funzioni, oggetti, componenti, agenti, ecc.

Questa ultima affermazione dovrebbe far riflettere su come ogni paradigma computazionale (funzionale, a processi, ad oggetti, ad agenti, ecc) porti ad una sorta di "specializzazione" dell'ingegneria del software. Di fatto ogni paradigma influenza il modo

in cui il sistema viene pensato e modellato, perchè ogni paradigma porta con sè le proprie astrazioni di base attraverso le quali il sistema viene pensato e costruito: differenti tipi di astrazioni portano sia a differenti modi di pensare il problema e la sua relativa soluzione, sia a differenti livelli di complessità dei modelli: la dimensione sistemistica può essere meglio affrontata usando oggetti, componenti e agenti piuttosto che semplici procedure o funzioni, che potrebbero portare a schemi di soluzione molto più intricati introducendo una ulteriore ed inutile dimensione di complessità.

Possiamo quindi pensare di avere una *famiglia di "Ingegnerie del software"* ognuna ispirata da uno specifico paradigma computazionale e che adotta specifici tipi di astrazioni per la modellazione dei sistemi, ma tutte queste condividono le linee guida generali e i principi cardine del processo di ingenerizzazione dei sistemi.

Metodo e metodologia

Esiste molto disaccordo riguardo alla relazione tra i termini *metodo* e *metodologia*. Nell'uso comune, metodologia è usata frequentemente come sostituto di metodo, raramente avviene il contrario. Alcuni ricercatori sostengono che questo avviene poichè il termine "metodologia" suona in modo molto più accademico e importante che non metodo. A nota a piè di pagina alla definizione di metodologia sull'American Heritage Dictionary del 2006 riporta: *the misuse of methodology obscures an important conceptual distinction between the tools of scientific investigation (properly methods) and the principles that determine how such tools are deployed and interpreted (properly methodologies)*.

Anche nell'ambito dell'ingegneria del software non esiste un comune denominatore: qualche autore sostiene che un metodo dell'ingegneria del software sia una sorta di "ricetta", una serie di passi da eseguire per costruire software, mentre una metodologia sia un insieme codificato di procedure raccomandate. In questa ottica un metodo può essere visto come parte di una metodologia. Altri autori invece sostengono che una metodologia racchiuda in sè un proprio approccio filosofico alla risoluzione dei problemi. Da queste definizioni sembra emergere che l'ingegneria del software sia piena di metodi ma molto scarsa di metodologie.

Esistono molte altre svariate definizioni di metodologia e metodo. Una tra le più interessanti per la *metodologia* è quella proposta in [GFMMP91] *A methodology is a collection of methods covering and connecting different stages in a process. The purpose of a methodology is to prescribe a certain coherent approach for solving a problem in the context of a software process by preselecting and putting in relation a number of methods. A methodology has two important components: one that describes the process elements of the approach, and a second that focuses on the work products and their documentation.*

In accordo a questa definizione in [CCZ05] si definisce un *metodo* come *a way of performing some kind of activity within a process, in order to properly produce a specific output (i.e., an model or a document) starting from a specific input (again, an artefact or a document). Any phases of a process, to be successfully applicable, should be complemented by some methodological guidelines (including the identification of the techniques and tools to be used, and the definition of how artifacts have be produced) that could help the involved stakeholders in accomplishing their work according to some defined best practices.*

Queste definizioni appena proposte chiariscono l'ambiguità tra metodo e metodologia, ma allo stesso modo finiscono per creare ambiguità tra processo e metodologia. Tra i due termini possiamo trovare qualche elemento comune:

- entrambi si focalizzano su cosa dobbiamo fare nelle differenti attività necessarie per costruire un sistema software,
- il processo di sviluppo è più concentrato sul processo globale che include tutte le fasi, il loro ordine e schedule, la metodologia invece si concentra molto più sulle specifiche tecniche da usare e sul lavoro che deve essere prodotto.

Possiamo quindi affermare che una metodologia si concentra esplicitamente su come portare a termine una attività o un passo di una specifica fase del processo, mentre il processo si occupa anche di aspetti di gestione più generali come per esempio trovare risposta alla domande riguardo a "chi, quando e quanto costa".

Concludendo, possiamo affermare che come per i processi di sviluppo **non esiste una metodologia ideale**. Differenti metodologie hanno differenti aree di applicazione: le metodologie orientate agli oggetti sono tipicamente molto appropriate per sistemi interattivi, ma lo sono meno per sistemi con stringenti requisiti real-time.

Breve storia delle metodologie

Lo sviluppo dei linguaggi di programmazione ad alto livello segna il primo affrancamento della costruzione del software dai dettagli operativi di uno specifico elaboratore.

Il movimento della **programmazione strutturate** iniziato attorno al 1970 introduce i primi elementi di organizzazione entro i programmi, portando alla definitiva eliminazione di alcune istruzioni di basso livello, tra cui il salto incondizionato (**goto**).

Il movimento dello **Structural Design** promosso da Edward Yourdon e Larry Constantine [YC78] propone criteri come **(un)coupling** e **cohesion** per dare maggiore struttura e qualità al progetto, introducendo nel 1978 il formalismo delle **Structured Charts**.

Mellier Page-Jones propone nel 1988 un insieme di strategie di mapping [PageJones88] per trasformare in modo sistematico modelli di structured design da modelli di structured analysis. Queste tecniche sono riprese oggi da **MDA**.

Tom De Marco [DeMarco78] e Edward Yourdon [Yourdon88] diffondono la **Structural Analysis** basata sul principio di separare la specifica del problema dalla specifica della soluzione. I modelli dell'analisi e del progetto vengono espressi in termini di **Data Flow Diagrams (DFD)** e **State Diagrams** con notazioni oggi assimilate da **UML** (tranne i **DFD**),

Nel 1985 Ward e Mellor introducono [WM85] l'approccio **outside-in** alla modellazione, in alternativa al **top down**, enfatizzando che l'analisi e il progetto di un sistema software deve partire dalla comprensione/modellazione dell'ambiente in cui il sistema deve operare. I modelli sono espressi in termini di **terminators**, **events** ed **event-response**; a questi concetti corrispondono oggi l'idea di **actor** (per **terminators**) e **use case** (per **event-response**) introdotte da Ivar Jacobson nella sua **OOSE (Object Oriented Software Engineering)** [JCJ092].

L'avvento e la diffusione del paradigma ad oggetti convinse poi molti che per l'organizzazione dei sistemi l'idea di oggetto fosse un punto di partenza più solido e stabile che non l'idea di funzione. Booch [Booch86], Meyer [Meyer88] ed altri nel 1986-

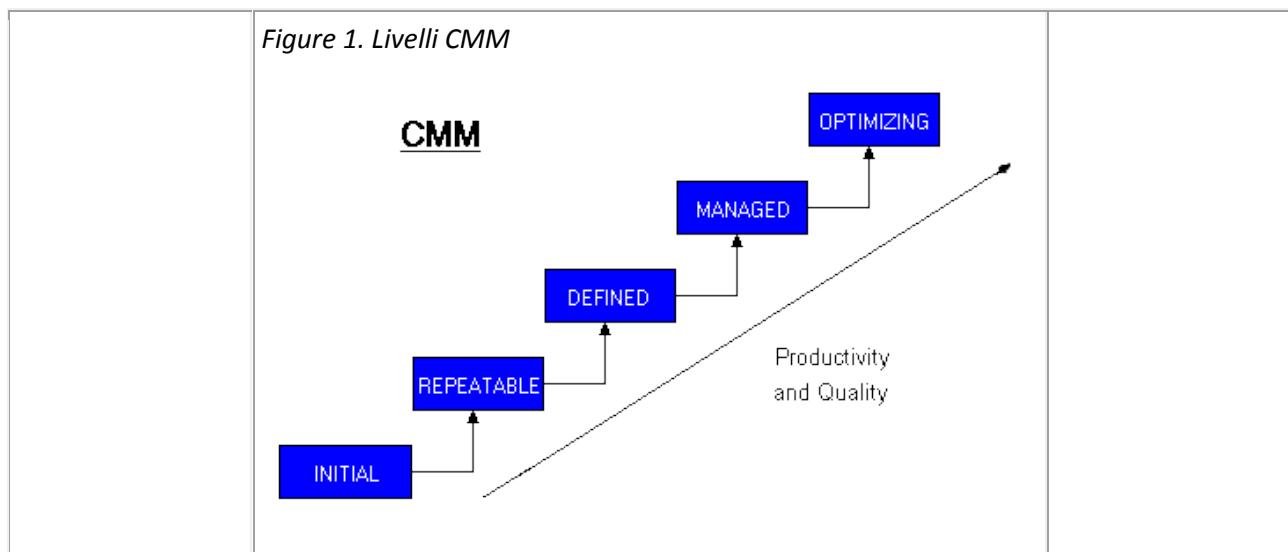
1988 introdussero le prime forme di **OOD** (*Object Oriented Design*) e **OOA** (*Object Oriented Analysis*). Meyer sostenne il principio della **programmazione per contratto** [Meyer92] basata su interfacce, pre-, post-condizioni ed invarianti traducendo le sue idee nel linguaggio **Eiffel**. Queste idee si ritrovano anche in linguaggi di specifica come **Z** [PST96], **VDM** [Jones91] e in **OCL** [WK99].

James Rumbaugh fu capofila di coloro che introdussero nel 1991 [RBPEL91] la **Object Management Technique (OMT)** e l'associato insieme di notazioni per dare supporto alla costruzione di modelli di analisi e di progetto. Il vantaggio dell'uso di notazioni rigorose per i modelli fu sottolineato dal metodo **Syntropy** introdotto nel 1994 da Steve Cocks e John Daniels [CD94].

Il termine **software engineering** potrebbe essere stato coniato nel 1968 in un workshop NATO (Naur e Randell, 1969).

Capability Maturity Model

All'inizio degli anni 90 il **Software Engineering Institute (SEI)** ha sviluppato, su sollecitazione del Dipartimento della Difesa (DoD) U.S.A un sistema di classificazione delle capacità di una organizzazione in relazione al processo di produzione del software. Il sistema **SEI**, denominato **Capability Maturity Model (CMM)** è articolato in 5 livelli:



Livello 1 (Initial)

L'organizzazione è capace di costruire prodotti software ma:

1. non è riconoscibile alcun processo
2. la qualità del prodotto e del progetto dipende totalmente dalle persone che vi lavorano
3. al termine di un progetto non vi è alcuna memoria del costo, del piano di lavoro o della metrica usata per stabilire la qualità
4. il successo di un nuovo progetto non è correlato al successo dei progetti precedenti a meno di non impegnare le stesse persone

Livello 2 (Repeatable)

L'organizzazione è capace di costruire prodotti software tenendo traccia dei costi e dei tempi di lavoro in modo che risulta possibile predire il costo e il piano di lavoro per progetti simili. Le predizioni sono tuttavia ancora legate alle persone che hanno lavorato nel progetto.

Livello 3 (Defined)

L'organizzazione è capace di costruire prodotti software seguendo un processo standard che mitiga la dipendenza del risultato da specifiche persone. E' necessario prevedere un addestramento esplicito del personale. La capacità predittiva è ancora legata ai progetti sviluppati in passato

Livello 4 (Managed)

L'organizzazione è capace di costruire prodotti software impostando una fase di predizione dei costi e del piano di lavoro basandosi su una classificazione dei compiti e dei componenti e su metriche di misura dei loro costi e tempi di sviluppo. Questo livello non è rappresenta ancora la capacità massima in quanto mancano procedure per la gestione delle modifiche a livello di paradigmi, metodologie e strumenti.

Livello 5 (Optimized)

L'organizzazione segue un processo che include parti di meta-processo volte a valutare il processo stesso al fine di introdurre elementi di adattamento e miglioramento.

Processi di produzione del software

Costruire software sognifica realizzare un **prodotto**, sulla base di un **progetto**, adottando uno specifico **processo di sviluppo** svolto dal lavoro cooperativo di più **persone** che si pongono precisi obiettivi, tra cui:

- **Comprendere la natura e lo scopo del prodotto.**
- **Definire e analizzare i requisiti.**
- **Selezionare il processo di sviluppo, le tecnologie disponibili e definire un piano di lavoro.**
- **Progettare e costruire il prodotto.**
- **Collaudare il prodotto.**
- **Distribuire e manutenere il prodotto.**

Un **processo** di sviluppo software è un insieme ordinato di passi che coinvolgono tutte quelle attività, vincoli e risorse richiesti per produrre uno specifico output che soddisfa una serie di requisiti iniziali. Tipicamente un processo di sviluppo è composto da differenti fasi / stadi messi in relazione tra loro: ogni fase identifica la porzione di lavoro che deve essere fatta nel contesto del processo, le risorse che devono essere impiegate e i vincoli a cui si deve obbedire durante l'esecuzione della fase. Le diverse fasi sono tipicamente composte da un insieme di attività che possono a loro volta essere concepite come composizione di unità di lavoro atomiche (**task, step**).

Un processo di sviluppo del software si presenta quindi come un insieme di politiche, strutture organizzazionali, tecnologie, procedure e deliverable necessari per concepire,

sviluppare, mettere in esecuzione e manutenere un prodotto software. A tal fine sono utilizzati diversi contributi e concetti:

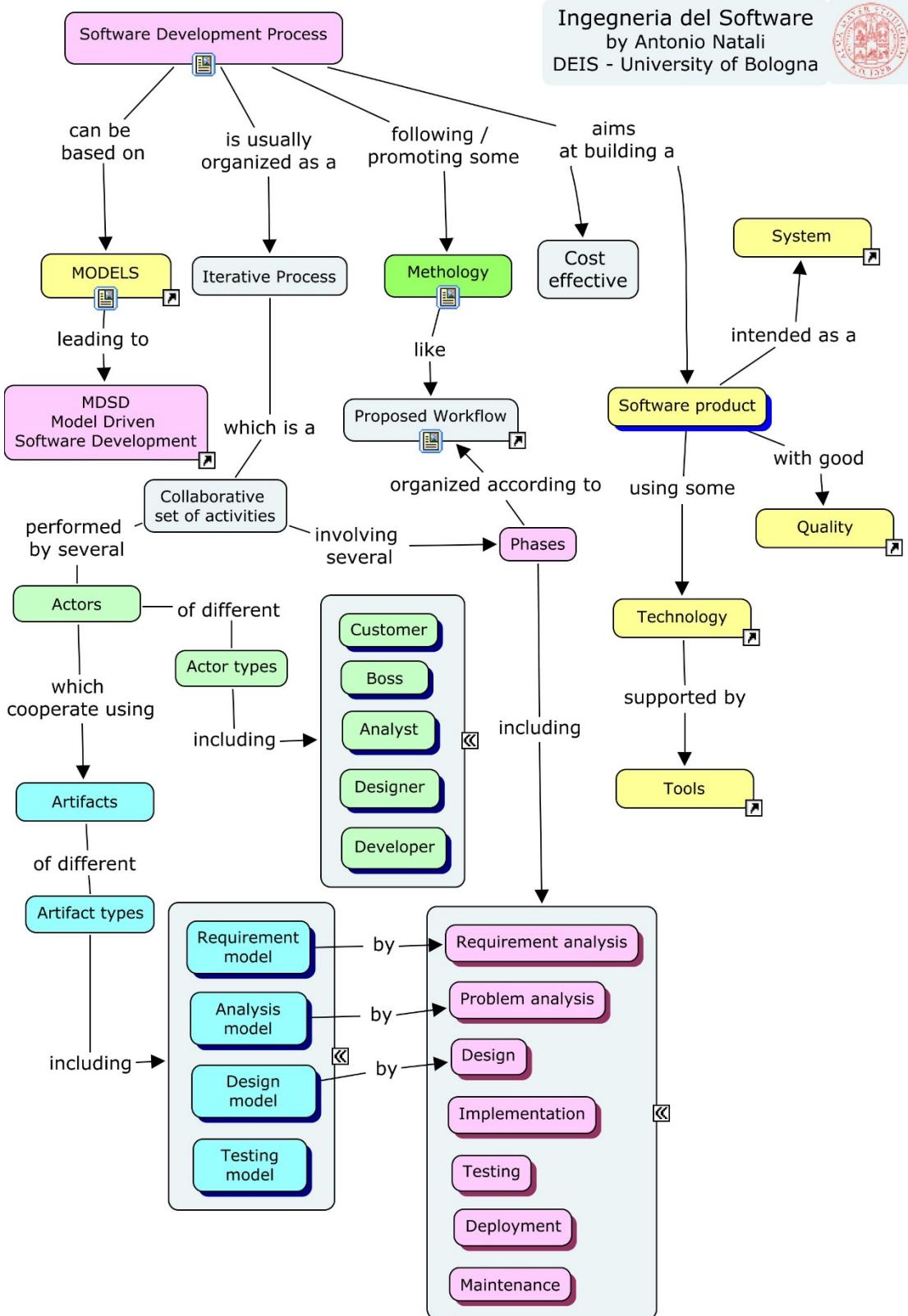
- **Tecnologie per lo sviluppo**: rappresentano il supporto tecnologico al processo. Per portare a termine il processo di sviluppo sono infatti necessari tool, infrastrutture e ambienti di sviluppo appropriati.
- **Metodi e tecniche di sviluppo**: rappresentano linee gida su come usare la tecnologia e compiere le attività dello sviluppo. Il supporto metodologico essenziale all'impiego corretto delle tecnologie.
- **Comportamento organizzazionale**: rappresenta la scienza per l'organizzazione delle risorse umane coinvolte. La gestione ottimale delle risorse umane infatti gioca un ruolo cruciale nel successo del processo di sviluppo.
- **Marketing ed economia**: lo sviluppo software non è uno sforzo autocontenuto. Come per ogni altri prodotto, il software deve soddisfare le esigenze del utilizzatore in specifici mercati.

Un processo di sviluppo può essere basato su un insieme di principi che nel loro complesso determinano una **metodologia** di riferimento, anche se è più comune intendere con il termine **metodologia** una collezione di metodi che determinano e connettono le diverse fasi del processo e per **metodo** la modalità di eseguire un'attività nell'ambito di un processo.

Sommerville [Sommerville07] identifica quattro attività fondamentali comuni ad ogni processo di sviluppo del software:

- **Specificazione** - è il processo di comprensione e definizione dei servizi richiesti dal sistema e dell'identificazione dei vincoli sulle operazioni del sistema e sul suo sviluppo.
- **Progetto e Implementazione** - è il processo di descrizione della struttura del software che deve essere implementato, dei dati che sono parte del sistema, delle interfacce tra i componenti del sistema e qualche volta degli algoritmi che devono essere usati. Successivamente questa specifica è convertita in un sistema eseguibile.
- **Validazione** - è il processo che intende mostrare che il sistema è conforme alla sua specifica iniziale e che il sistema incontra le aspettative degli utilizzatori.
- **Evoluzione** - è il processo che modifica il software in risposta ai cambiamenti dei requisti degli utilizzatori.

Reference map: Processi



Qualita' di processo e di prodotto

Un buon processo dovrebbe essere caratterizzato da un insieme di proprietà quali:

- **Effectiveness:** il processo deve aiutare a costruire il prodotto giusto, che corrisponda alle esigenze dell'utente o del committente.

- **Maintainability**: esprimere in modo esplicito le scelte di progetto e di realizzazione, in modo che sia possibile rimediare ad errori o introdurre modifiche con relativa facilità.
- **Predictability**: dare la possibilità di pianificare il lavoro e valutare i costi di produzione.
- **Repeatability**: un buon processo dovrebbe poter essere riapplicato nella costruzione di altri sistemi, senza costringere un team di lavoro a riorganizzarsi partendo da zero.
- **Quality**: permettere la costruzione di prodotti di qualità.
- **Improvement**: un buon processo dovrebbe permettere di identificare i margini di miglioramento del processo stesso.
- **Tracking**: permettere ai diversi attori, inclusi i committenti, di seguire lo stato del processo.

Per quanto riguarda il prodotto, una possibile definizione di qualità può essere quella data dal British Standards Institute: *The totality of features and characteristics of a product or service that bear on its ability to satisfy a need.*

Costruire software con caratteristiche di qualità non significa solo produrre codice che "funziona" ma anche dare risposta a due domande fondamentali:

- è stata costruita la cosa giusta? (**validazione**)
- il prodotto è stato costruito in modo giusto? (**verifica**)

All'obiettivo della verifica si associa spesso una valutazione sulla capacità del prodotto di soddisfare alcuni importanti requisiti. Un "buon codice" ad esempio dovrebbe:

- esprimere in modo chiaro il modo con cui soddisfa i requisiti funzionali;
- controllare i propri ingressi e reagire in modo predicable a ingressi considerati illegali;
- essere stato ispezionato e validato da persone competenti diverse dalla persona che lo ha scritto;
- essere stato collaudato in modo sistematico ed esaustivo in modi diversi e indipendenti;
- essere associato ad una adeguata documentazione;
- avere ragionevole cognizione del proprio tasso di fallimento (**defect rate**) ;
- essere **estendibile**, cioè dotato della capacità di essere potenziato per fornire nuove funzionalità;
- essere **evolvibile**, cioè adattabile a requisiti (ragionevolmente) diversi;
- essere **portatile**, cioè applicabile a diversi ambienti e piattaforme;
- essere **generale**, cioè applicabile a uno spettro relativamente ampio di situazioni.

La responsabilità primaria per produrre un artefatto in qualità è ovviamente legata alla persona che lo crea. Per evitare autoreferenzialità, molte organizzazioni affiancano alla diverse fasi del processo di produzione un **processo di revisione** affidandolo a persone diverse.

Il processo di revisione può avvenire a prodotto costruito o in itinere e secondo approcci a **black box** o a **white box**. Ad esempio il collaudo finale di un'applicazione per verificare il soddisfacimento dei requisiti funzionali è un tipico processo a black-box che astrae dalla organizzazione interna del sistema. Viceversa l'**'ispezione** di un progetto da parte di un gruppo di esperti è un processo a **white box** che permette al progettista di conoscere eventuali difetti e ripararli una volta individuati.

Processi a cascata

I **processi di sviluppo a cascata** sono caratterizzati da una netta separazione tra la fase di **costruzione** dalla fase di **analisi** e **progettazione**; processi di questo tipo sono comunemente applicati dall'ingegneria edile, meccanica, aerospaziale, etc.

Questo tipo di processi si è rivelato spesso inadeguato per il software, in quanto un prodotto software si presenta esso stesso più come un **progetto** che come un sistema fisicamente riconoscibile; i vincoli indotti dalle leggi fisiche sono per lo più relativi alle risorse umane e monetarie e il materiale stesso della costruzione è costruito da prodotti software (linguaggi, strumenti, piattaforme operative, etc) che evolvono con sorprendente rapidità.

Ma il punto più rilevante è che nella costruzione del software **i requisiti sono spesso non stabili**, sia perché le attese del committente cambiano a causa della flessibilità intrinseca al software, sia per la natura di molti domini applicativi. A ciò si affianca la continua evoluzione dei supporti operativi.

In un contesto di questo genere, seguire il modello a cascata potrebbe dilazionare nel tempo il momento in cui affrontare i **rischi** connessi al progetto. Scoprire in fase di progetto avanzato o in fase di collaudo che un requisito è stato mal definito o del tutto ignorato o che esiste un modo migliore per utilizzare il supporto operativo di riferimento è spesso troppo tardi ed è la ragione di molti fallimenti.

Il modello a cascata può essere adottato con successo solo in casi particolari o perseguito, come propone l'approccio **MDA** (si veda [Architecture model-driven](#)), una generazione automatica del codice a partire da modelli.

In ogni caso, per sistemi complessi e/o di grandi dimensioni risulta necessario prevedere la retroazione di una fase sulle precedenti, in relazione sia ad approfondimenti nella comprensione del problema, sia a cambiamenti del punto di vista e dell'esperienza dei progettisti o della evoluzione della tecnologia.

L'alternativa al processo di sviluppo a cascata è un **processo iterativo** in cui la dinamica con cui si svolgono i vari task prevede una esecuzione ripetuta dei task stessi nell'ambito di varie fasi, al termine di ciascuna delle quali corrisponde una qualche versione più o meno dettagliata del sistema software da costruire.

Processi iterativi a spirale

Con l'introduzione di metodologie di sviluppo iterative ed adattative, la costruzione del software non è più vista come un atto creativo istantaneo (un "big bang") quanto un processo di crescita, tipico di un organismo in evoluzione.

L'immagine che normalmente si evoca è quella di una **spirale** [Boehm88] in cui ogni voluta rappresenta un raffinamento e/o completamento del sistema rispetto alla voluta precedente.

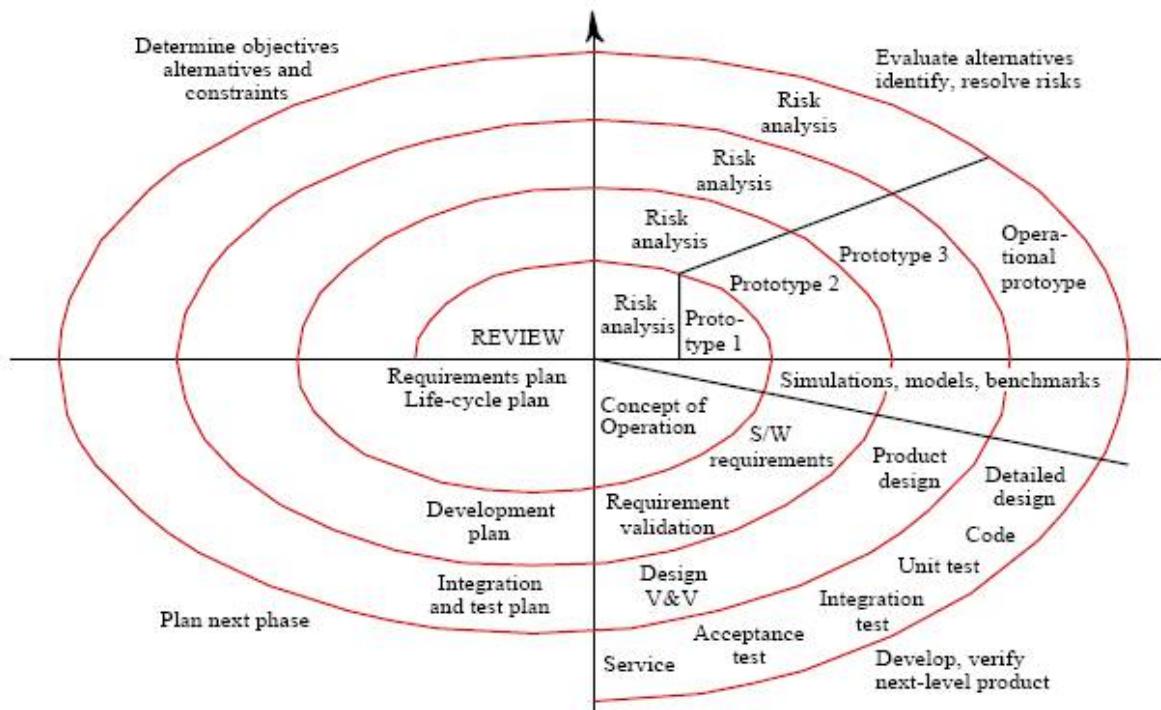


Figure 1. Processo a spirale

Lo sviluppo a spirale deve essere sostenuto da metodologie di progettazione orientate al cambiamento e alla costruzione di sistemi modificabili. Le proposte in questo senso - quali **rapid prototyping**, **evolutional delivery** [Gilb88] poi divenuto **incremental model** [Sommerville07], **synchronize and stabilize** [CS96], **evolutionary development** [Larman03] - hanno numerose caratteristiche in comune:

- focalizzano l'attenzione sull'articolazione di sistemi in **parti riusabili**, organizzate secondo principi di **astrazione**, **località** ed **incapsulamento** della informazione; pertanto vi è uno stretto collegamento con concetti e principi tipici del **paradigma ad oggetti**;
- mirano a collegare senza discontinuità (**seamless**) il passaggio dalle fasi di analisi e definizione dei requisiti alle fasi di progetto, costruzione e collaudo;
- puntano a sostenere **processi di sviluppo incrementali**, individuando metodologie e meccanismi per sostenere la **coerenza** e la **consistenza** del processo e il lavoro collaborativo tra più persone.

Riguardo all'ultimo punto vi sono due principali scuole di pensiero, la scuola che ritiene che il miglior collante sia costituito da artefatti costituiti da modelli espressi in linguaggi standard come **UML** e la scuola che ritiene che il miglior collante sia il codice stesso.

Processi basati su modelli

La scuola promossa da **Grady Booch**, **James Rumbaugh** e **Ivar Jacobson**, che ha condotto alla definizione e allo sviluppo di **UML** [RJB97] sostiene la necessità di pervenire al codice finale attraverso la specifica di un insieme di **modelli** del sistema.

Nel processo di produzione del software infatti, il lavoro singolo e cooperativo dei diversi attori (analisti, progettisti, sviluppatori, etc) si basa su un insieme di **conoscenze** che spesso rimangono implicite all'interno del processo. Uno degli scopi dei **processi model**

based è rendere esplicite queste conoscenze attraverso la costruzione di artefatti in forma di **diagrammi** espressi con notazioni formali, cioè attraverso l'uso di un linguaggio con una ben precisa sintassi e semantica.

Lo scopo dei diagrammi è rappresentare **modelli del sistema** volti a descrivere, in modo conciso e preciso, conoscenze sul problema utili anche per individuare rischi e scelte progettuali. L'uso dei modelli è motivato principalmente dalla difficoltà della mente umana di impostare ragionamenti efficaci in presenza di dettagli troppo minuti. I linguaggi per la descrizione dei modelli rappresentano quindi il culmine della continua evoluzione verso livelli di astrazione più elevati rispetto alle macchine che ha caratterizzato la storia dei linguaggi di programmazione.

Il concetto di modello può riguardare anche il processo stesso di produzione del software: un **modello del processo di sviluppo** definisce il tipo e l'organizzazione delle fasi in cui esso si articola e il tipo di interazione tra gli attori del processo.

Processi agili

La scuola promossa da **Kent Beck**, nota come **Extreme Programming (XP)** [Beck00], è l'espressione più nota di uno stile di pensiero per cui il miglior modello di un sistema software rimane comunque nel codice. La progettazione viene vista come parte integrante del processo di programmazione: il progetto si modifica al modificarsi del codice e occorre utilizzare metodologie che abilitano e sfruttano questo fatto, garantendo pieno controllo sui cambiamenti e sui rischi.

Alla base della metodologia **XP**, Beck pone la iterazione di quattro pratiche fondamentali: **codifica, collaudo, interazione e progettazione** la cui applicazione non è ovviamente esclusa dagli approcci model-driven.

La pratiche dell'eXtreme Programming



Le pratiche proposte da Beck [Beck00] possono essere riassunte come segue:

Planning game

Determina l'ambito (**scope**) della iterazione corrente, affrontando per primi i task ritenuti prioritari.

E' richiesta una forte interazione con il committente per concordare le priorità e le date di delivery, sulla base delle stime degli analisti e degli sviluppatori, tenendo conto dei rischi e della organizzazione del personale.

Le feature del sistema da sviluppare sono descritte e spiegate in **index cards**.

Small releases

Si tenta di usare la **regola di Pareto** per ottenere l'80% del valore atteso dal committente (**business value**) con il 20% di sforzo, ponendo il progetto in produzione il prima possibile.

Questa attività è fortemente correlata con la precedente per determinare le priorità e per mantenere il progetto quanto più semplice possibile.

Simple design

Mira a mantenere il progetto semplice evitando di affrontare problemi relativi a potenziali bisogni futuri. Procedendo in stretta coordinazione con **Small releases** si cerca di produrre rapidamente un sistema funzionante per ricevere feedback dal committente senza cercare di prevedere quello che potrebbe volere.

Testing

Si cerca di impostare un sistema automatizzato di collaudo, in modo da assicurarsi che il sistema possa continuare a "funzionare" anche in conseguenza di modifiche architettoniche o refactoring (**"A feature does not exist unless a test validates that it functions."**).

Continuous integration

Si cerca di automatizzare li processi di produzione, distribuzione e deploy in modo da poter ottenere una versione funzionante dell'**intero sistema** entro poche ore ed almeno una volta al giorno.

Refactoring

Introduce modifiche nel codice con l'obiettivo di mantenerlo semplice, evitando duplicazioni. Nel contesto di una continua interazione con il committente, la necessità di refactoring può essere dovuta alla modifica o aggiunta di features e anche alla necessità di reimpostare l'archettura del sistema.

Pair-programming

Uno stesso task viene assegnato a una coppia e non a una singola persona, al fine di aumentare la qualità sfruttando la collaborazione e il senso critico.

Collective ownership

Chiunque può apportare una modifica al sistema, una volta che questa sia stata collaudata e che la vecchia versione continui ad essere disponibile. La condivisione delle "proprietà" del sistema implica la condivisone della **responsabilità**, mitigando la criticità dovuta a persone o ruoli insostituibili riguardo a parti ritenute complicate o critiche.

40-hours week

Il tempo dedicato al lavoro deve essere limitato per evitare che persone sovraccaricate di lavoro introducano più errori che contributi positivi.

On-site customer

Il committente dovrebbe essere sempre reperibile quando sia necessario chiarire le **features stories**.

Metaphor

Tra committente e sviluppatori deve essere stabilito un linguaggio comune per esprimere senza fraintendimenti le funzionalità richieste.

Coding standard

Stabilisce stili, pratiche comuni e convenzioni seguite da tutto il team di lavoro.

Quale processo?

Un'importante considerazione da fare è che **non esiste un processo ideale**. Differenti tipi di sistemi necessitano di differenti tipi di processi di sviluppo: ad esempio, per il software real-time di controllo di un aereomobile, il progetto deve essere completamente specificato prima che lo sviluppo possa cominciare, mentre in un sistema di e-commerce la specifica può anche non essere del tutto completa quando inizia lo sviluppo del sistema.

La definizione di un processo non può prescindere dalle caratteristiche dell'organizzazione in cui esso avviene e dalla competenza ed esperienza degli attori coinvolti. Nel 1968 Melvin Conway enunciò una famosa "legge": "**Any piece of software reflects the organizational structure that produced it**" proprio per sottolineare che esiste uno stretto rapporto tra l'organizzazione dello sviluppo e l'architettura dei sistemi che un'organizzazione produce. Difficile quindi pensare che non vi sia un impatto anche sui processi di sviluppo stessi. Di conseguenza le generiche attività di un processo possono essere organizzate in modi differenti e descritte a differenti livelli di dettaglio per i differenti tipi di sistemi. L'uso di un processo inappropriate può ridurre la qualità o l'utilità del prodotto software che deve essere sviluppato o migliorato.

Al fine di supportare lo sviluppatore nell'adozione del processo più adeguato sono stati sviluppati pattern e modelli (**Software Process Models**) che forniscono una rappresentazione semplificata di un processo software da una specifica prospettiva. Ricordiamo al proposito i pattern relativi a metodologie agili [Bergin05], XP [BJ05], model.driven [Evans03], [Völter06], test-driven [Meszaros07] e i pattern languages proposti per la integrazione tra un software development process e un'organizzazione [CH04].

Un modello di processo prescrive le fasi attorno a quali un processo dovrebbe essere organizzato, l'ordine di esecuzione delle attività, i modi in cui le diverse fasi di lavoro devono interagire e coordinarsi tra loro, etc. In altre parole, un modello di processo definisce un **template** attorno a cui organizzare e dettagliare un vero processo sulla base di astrazioni utili a definire ed organizzare le fasi di produzione e gli artefatti prodotti in ciascuna fase.

Dai linguaggi di programmazione ai modelli

L'elaboratore elettronico può simulare il comportamento di ogni altra macchina (per elaborare informazione); ciò rende il progettista del software un "**costruttore di nuovi mondi**", la cui organizzazione interna e le cui leggi di funzionamento possono essere molto diverse da quelle del mondo che ha reso possibile e che supporta quella stessa costruzione.

I "mondi virtuali" possono essere organizzati secondo leggi anche molto diverse tra loro, ma devono comunque obbedire a quei principi fondamentali che permettono ad un essere umano di affermare che il risultato produce un **sistema** e non una semplice aggregato di elementi: occorre garantire **corenze e consistenza** e il soddisfacimento di proprietà strutturali e comportamentali che garantiscono efficienza e uso corretto delle risorse permettendo anche la modifica (l'evoluzione) di quanto realizzato.

Tradizionalmente l'obiettivo di descrivere la struttura e il comportamento di un sistema software è stato raggiunto attraverso l'uso di linguaggi di programmazione. Oggi la descrizione di un sistema attraverso uno o più **modelli** (si veda Il concetto di modello) intesi come **viste del sistema** lungo una specifica dimensione (si veda Tre dimensioni fondamentali) permette di regionare sulle proprietà del prodotto software da realizzare senza essere condizionati dalle caratteristiche della piattaforma operativa e fornisce un filo conduttore che facilita l'integrazione e la gestione controllata delle varie fasi dello sviluppo.

Prima di addentrarci nei dettagli di questa sempre più importante transizione dai linguaggi di programmazione ai modelli, può essere conveniente richiamare in breve alcune tappe salienti che hanno storicamente condotto a questo punto.

Linguaggi e paradigmi computazionali

Alla base di ogni comunicazione occorre un linguaggio. Come il linguaggio ha svolto un ruolo fondamentale nello sviluppo della cultura e nella evoluzione della specie umana, così esso assume un ruolo altrettanto importante nello sviluppo dei sistemi di elaborazione e nelle comunicazioni uomo-macchina e macchina-macchina. Grazie alle proprie capacità espressive, il linguaggio costituisce un potente stimolo per suggerire concetti e modi di soluzione appropriati per problemi di un ampio spettro di domini applicativi. Al tempo stesso però, esso può rappresentare un impedimento, quando le sue metafore non siano adeguate o divengano obsolete.

Un linguaggio per esprimere computazioni meccanizzabili stabilisce un profondo rapporto con il concetto di automa per la risoluzione di problemi che conduce al quesito fondazionale dell'informatica teorica: **quali elementi occorre introdurre per denotare processi computazionali meccanizzabili e quali proprietà di conseguenza possono essere associate a tali processi in virtù dei sistemi di denotazione adottati?**

La risposta a questo quesito trova le sue radici nella logica-matematica ed è un risultato di studi compiuti a partire dagli anni 1920-30. Per il moderno ingegnere del software questa parte della storia dell'informatica costituisce un ottimo esempio del rapporto tra analisi, progetto e realizzazione nel contesto di un'impresa che mirava a costruire una "macchina" capace di costituire un **elaboratore "universale"** di informazione.

Sintassi (astratta) e semantica

La descrizione dei linguaggi viene effettuata introducendo notazioni formali diverse per la descrizione della struttura delle frasi (**sintassi**) e per la descrizione del significato di una frase (**semantica**). La struttura delle frasi lecite di un linguaggio può essere descritta attraverso un dispositivo formale denominato **grammatica**. Pur se meno diffusi per via della loro relativa maggior complessità, sono disponibili anche vari metodi formali per la descrizione della semantica di un linguaggio.

La **sintassi astratta** di un linguaggio fornisce una descrizione indipendente da ogni notazione concreta specificando il linguaggio in termini delle strutture che lo formano. Tecnicamente la sintassi astratta di un linguaggio è istanziata dall'analizzatore sintattico del linguaggio (**parser**) come forma interna del codice sorgente. Ad esempio un documento scritto in sintassi concreta **XML** [XML] viene rappresentato in memoria da un **DOM** [DOM] (**Document Object Model**) la cui organizzazione strutturale costituisce la sintassi astratta di **XML**.

La **semantica statica** determina i criteri di corretta formazione delle strutture di un linguaggio. Ad esempio una regola di molti linguaggi di programmazione di alto livello "tipati" è che ogni variabile deve essere dichiarata prima di essere usata; questa regola non può essere specificata dalla sintassi (né astratta né concreta). Poichè la regola viene controllata dal compilatore si parla di semantica anche se concettualmente la semantica statica appartiene più al campo sintattico.

La **semantica dinamica** determina il significato dei costrutti del linguaggio.

Macchine per elaborare informazione

Per affrontare il "problema dell'elaborazione dell'informazione" gli studiosi hanno inizialmente focalizzato l'attenzione su **cosa deve fare** una macchina per elaborare e solo in un secondo tempo su **come lo deve fare**. In particolare il concetto di **automa** (per elaborare informazione) è stato precisato come un **dispositivo concettuale** che stabilisce una precisa **relazione** tra un dato di ingresso e un dato di uscita, soddisfacendo i seguenti vincoli di realizzabilità fisica:

- se l'automa è fatto di parti, queste sono in **numero finito**;
- l'ingresso e l'uscita sono denotabili attraverso un **insieme finito** di simboli.

Il punto chiave del progetto del comportamento di un elaboratore consiste nel dotare una unità centrale di elaborazione (**CPU**) della capacità di interpretare una sequenza di istruzioni efficiente e computazionalmente completo. La memoria è una sequenza lineare di celle destinata a contenere la rappresentazione binaria delle istruzioni e dei dati su cui esse agiscono.

In questo modo non si considera alcun aspetto "fisico" o tecnologico specifico. L'automa potrebbe essere realizzato da un insieme di dispositivi elettronici digitali, oppure da dispositivi meccanici o biologici. L'obiettivo è di astrarre dai singoli, specifici casi concreti enucleando le caratteristiche ritenute essenziali.

La storia dell'informatica ha quindi avuto inizio impostando un processo di progettazione indipendente dalle tecnologie, che ha prodotto come risultato la definizione di **modelli matematici** caratterizzati da precise proprietà. Questi modelli, detti **sistemi formali** hanno non solo definito il concetto stesso di computabilità ma sono stati alla base dello sviluppo tecnologico che caratterizza l'era dell'informazione (per un approfondimento si veda [VRH04]).

Macchine astratte

Tra i più noti sistemi formali introdotti per caratterizzare il concetto di computabilità vi è una gerarchia di macchine astratte che parte dagli automi a stati finiti (**ASF**) e termina alla macchina di Turing (**TM**). Automi a capacità computazionale intermedia sono gli **automi a stati finiti con stack** e la **macchina di Turing a nastro limitato** [Brady77] (si veda

anche [Educational_abstract_machines](#)). In questa gerarchia, ogni livello caratterizza la capacità di risolvere classi diverse di problemi, attribuendo alla **TM** la capacità più elevata.

La macchina di Minsky (MM)

L'insieme delle istruzioni necessarie a formare un sistema computazionalmente completo è sorprendentemente ridotto. L'insieme che segue costituisce un esempio basato sulla ipotesi che l'unità di elaborazione possa fare riferimento ad un numero illimitato ma finito di celle di memoria, ognuna delle quali può contenere un numero intero arbitrariamente grande. Ogni istruzione è contenuta in una cella di memoria associata in modo univoco ad un nome o etichetta (**label**); le istruzioni appartengono al seguente insieme (**instruction-set**):

- **ZERO** *cell* {poni 0 nella cella specificata dall'indirizzo **cell**}
- **INC** *cell* {incrementa di 1 il contenuto della cella specificata}
- **SUBJZ** *cell* *label* {se il contenuto di **cell** vale 0 salta alla istruzione di etichetta **label**, altrimenti decrementa di 1 il contenuto di **cell** e prosegui in sequenza}
- **HALT** {ferma la macchina}

Minsky [Minsky] ha dimostrato (si veda anche [Counter machine](#)) che un automa di questo genere è Turing-equivalente. Ogni problema (computabile) può quindi essere risolto con questo ridottissimo insieme di istruzioni, definito in modo da riflettere lo stile imperativo tipico dei sistemi di Von Neumann. In particolare, si possono evidenziare le due categorie fondamentali di istruzioni di ogni elaboratore elettronico:

- istruzioni di manipolazione di dati (**ZERO**, **INC**, **SUBJZ**)
- istruzioni di controllo del flusso del progamma (**SUBJZ**)
- **Funzioni e procedure**
 - I linguaggi di programmazione più diffusi ([Cobol](#), [FORTRAN](#), [C](#), [C++](#), [C#](#), [Java](#), [Lisp](#), [Prolog](#), [CLR](#), [ByteCode](#), etc) sono tutti **computazionalmente completi**, cioè dotati della capacità tipica del formalismo della **TM** di poter esprimere la soluzione ad ogni problema computabile (che viene definito tale proprio perchè è possibile definire una **TM** che lo risolve).
 - Ciò che differenzia i linguaggi e che segna la storia stessa della evoluzione dei linguaggi è la capacità di esprimere elementi utili alla organizzazione del software.
 - Già i primi linguaggi di programmazione ([FORTRAN](#) e [LISP](#)) nascono con l'intento di fornire costrutti utili ad elevare il livello di astrazione e a promuovere la modularità del software. Due di questi costrutti risultano ancora oggi fondamentali: la procedura e la funzione.
 - Le **procedure** e le **funzioni** costituiscono una prima, elementare, ma tuttora fondamentale forma di "componente software riusabile" all'interno di una proto-infrastruttura costituite dal programma principale. Sul piano semantico questi costrutti catturano l'idea di un automa risolutore che può anche riusare (copie di) sè stesso per fornire la soluzione ad un problema, dando pieno supporto a schemi di ragionamento ricorsivo e iterativo.
 - Ne scaturisce il classico modello organizzativo delle computazioni noto come "read-eval-print":

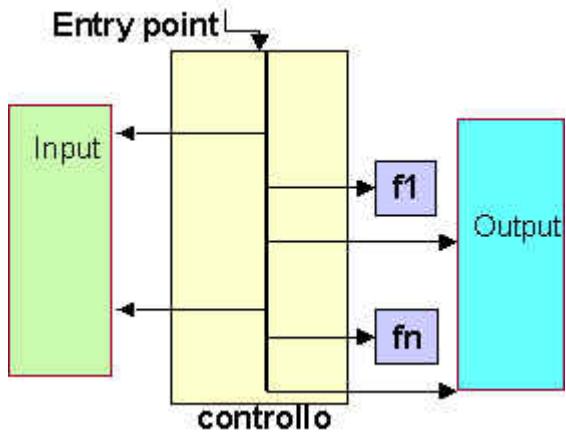


Figure 1. Read-Eval-Print

- Ad esempio, un sistema software concepito con l'ottica del linguaggio C ha come elemento centrale il componente (programma) che esprime il flusso di controllo con cui sono attivate le azioni di elaborazione espresse da funzioni; le interazioni tra funzioni avvengono mediante invocazioni (con trasferimento di controllo oltre che di dati), mentre le interazioni con l'utente attraverso i dispositivi di I/O.
- L'evoluzione introdotta negli anni 70 dalla **programmazione strutturata** ha promosso l'eliminazione dai programmi del salto incondizionato (**goto**) a favore di costrutti (**for**, **while-do**) capaci di catturare l'essenza del ragionamento iterativo, di disciplinare la progettazione e costruzione di cicli e di aumentare l'efficienza rispetto a iterazioni realizzate da chiamate ricorsive di funzioni in assenza della ottimizzazione relativa all **tail-recursion**.

Moduli e information hiding

A partire dal 1972 e dai contributi di Parnas [Parnas72], i linguaggi di programmazione tendono a introdurre nuovi costrutti linguistici per meglio catturare la semantica delle astrazioni di dato, separando in modo radicale l'aspetto della rappresentazione concreta dall'aspetto legato all'accesso ai dati. Si diffondono cioè costrutti capaci di supportare l'incapsulamento dell'informazione, cioè la proprietà di rendere inaccessibile la rappresentazione concreta dei dati (implementazione) e di permetterne la manipolazione solo attraverso le operazioni di un insieme di operazioni di interfaccia.

L'incapsulamento (anche detto **information hiding**) costituisce un concetto complementare all'astrazione. Mentre l'astrazione si focalizza sul funzionamento osservabile di un oggetto e su cosa fa un oggetto, l'incapsulamento si focalizza sull'implementazione e permette di modificare un sistema software con uno sforzo limitato. Qualunque sia l'implementazione scelta per una nuova categoria di dati, essa è inessenziale agli occhi dei clienti, una volta che assicuri il rispetto del contratto tra il cliente e un oggetto di quella categoria stabilito dall'interfaccia.

Il **modulo** nasce come un costrutto linguistico che permette di raggruppare dati, funzioni e procedure in una singola unità sintattica e che permette l'incapsulamento dell'informazione. Attraverso l'uso di un modulo è possibile costruire una barriera di astrazione intorno alla rappresentazione concreta di dati ed operazioni. Se ben definito, un modulo ha la possibilità di: garantire l'uso consistente e corretto di risorse computazionali, cioè il soddisfacimento di invarianti e di ottenere indipendenza dalla rappresentazione.

Mentre le funzioni possono essere viste come **meccanismi di astrazione rispetto alle espressioni** e le procedure come **meccanismi di astrazione rispetto ai comandi**, i

moduli **non** costituiscono invece un meccanismo di astrazione, ma solo un contenitore di informazione senza alcuna precisa semantica. In particolare il modulo in quanto tale non riflette e non suggerisce l'idea di astrazione di dato e un programmatore poco accorto potrebbe usare i moduli in modo incoerente e non ricevere alcuna indicazione di "errore" da parte del compilatore.

Un costrutto linguistico più direttamente correlato all'idea di tipo di dato inteso come unione di dati e operazioni applicabili si di essi è il costrutto **class**, che nasce nel [Simula67](#) per specifica la struttura e il comportamento di entità computazionali, detti **oggetti**, che costituiscono una unione di dati e comportamento, caratterizzata da una precisa semantica e specifiche proprietà.

Impostare un sistema sulla base di una Architettura ad oggetti (si veda Lo stile architettonicale), induce a spostare l'attenzione dal funzionamento interno di ciascun oggetto al modo con cui gli oggetti sono relazionati tra loro.

Il paradigma ad oggetti

L'introduzione di linguaggi orientati agli oggetti ha segnato la storia dell'informatica promuovendo un rapporto tra struttura e funzionamento più articolato della semplice coppia istruzioni-dati ed offrendo uno spazio concettuale molto più ricco che caratterizza (e condiziona) il modo con cui si analizzano, si progettano e si costruiscono oggi i sistemi software.

Il termine **oggetto** inizia a fare la sua comparsa nei primi anni 70, in modo indipendente in settori diversi dell'informatica, per catturare l'idea di un ente che abbina una parte dati ad un insieme di operazioni:

- nelle architetture dei calcolatori per dare supporto ai sistemi operativi;
- nei linguaggi per sostenere metodologie di programmazione modulare e incrementale;
- in intelligenza artificiale, nella forma di frames per la rappresentazione della conoscenza;
- nei sistemi informativi come base dei modelli semantici.

Impostare il progetto e la costruzione di un sistema software partendo dallo stile computazionale ad oggetti, permette di modulare nel modo più conveniente il rapporto tra la parte puramente algoritmica e la parte strutturale ed architettonicale del sistema. Infatti, anche il neofita è indotto, fin dalle prime fasi, a superare la tradizionale concezione di [programma=sequenza di istruzioni](#) che opera su una collezione di dati a favore di un sistema costituito da una [rete \(grafo\) di oggetti](#), tra loro interagenti attraverso un insieme di interfacce accuratamente progettate per garantire indipendenza dalla struttura interna.

Lo spazio concettuale di riferimento è molto più articolato della semplice coppia istruzioni(procedure)-dati; tra i suoi elementi più importanti, che formano parte integrante dei più diffusi linguaggi di programmazione come C++, Java, C#, vi sono infatti i seguenti:

- **oggetto**: un aggregato di dati ed operazioni, istanza di una classe;
- **classe**: descrizione della struttura e del comportamento di un oggetto. E' un costrutto che permette la creazione di oggetti. In taluni linguaggi la classe è essa stessa un oggetto, descritto da una metaclasses;
- **metodo**: un servizio reso da un oggetto;

- **interfaccia**: l'insieme dei metodi ed altre proprietà visibili di un oggetto;
- **messaggio**: la richiesta ad un oggetto di eseguire un metodo;
- **ereditarietà**: una relazione tra classi che permette il riuso delle specifiche e del codice;
- **delegazione**: una relazione tra oggetti che permette ad un oggetto di svolgere lavoro per conto di un altro;
- **tassonomia** (gerarchia di classi): una struttura dati ad albero che rappresenta la relazione di ereditarietà;
- **polimorfismo**: la possibilità di porre dietro ad una singola interfaccia diverse implementazioni;
- **binding dinamico**: il meccanismo che stabilisce il valore associato ad un identificatore solo a tempo di esecuzione.

Questo nuovo spazio concettuale:

- promuove metodologie basate sull'astrazione ed una chiara distinzione tra la fase di dichiarazione (di struttura e proprietà) e la fase operazionale di funzionamento;
- realizza un bilanciamento tra diversi paradigmi computazionali (dichiarativo, operazionale e interattivo);
- sussume i meccanismi legati ai principi dell'information hiding e ai concetti dell'astrazione di dato supportati dai linguaggi precedenti;
- propone nuovi meccanismi e forme di astrazione (polimorfismo, ereditarietà) utili sia per la progettazione sia per lo sviluppo e per la manutenzione del software (progettazione incrementale, riusabilità);
- informa senza discontinuità (**seamless**) tutte le fasi della produzione del software (specificazione dei requisiti ed analisi, progetto, implementazione, testing, manutenzione);
- permette di esprimere sia computazioni sequenziali sia computazioni concorrenti fornendo armi per aggredire l'intrinseca complessità dello sviluppo di applicazioni distribuite.

Nell'Aprile 1999 undici compagnie hanno fondato l'**Object Management Group** (OMG, www.omg.org) che oggi comprende circa 800 organizzazioni, con l'intento di fornire linee-guida e di definire specifiche di object management per framework comuni di supporto allo sviluppo di applicazioni.

Oltre gli oggetti

Lo sviluppo delle applicazioni distribuite in Internet ha "messo alle corde" il paradigma ad oggetti, che rivela mancanze su importanti concetti come l'autonomia, l'orientamento ai compiti da svolgere, la collocazione in un ambiente e la capacità di interagire in modo flessibile. Ad esempio, approcci basati sugli oggetti non catturano l'idea che l'interazione possa essere basata sulla negoziazione e non forniscono nessun supporto su come mantenere un bilanciamento tra il comportamento reattivo e quello proattivo in situazioni complesse e dinamiche.

L'adozione del paradigma ad oggetti induce a costruire le applicazioni adottando una prospettiva orientata alle funzionalità e porta o alla costruzione di architetture statiche o alla necessità di adottare complesse infrastrutture per la gestione delle dinamiche, delle riconfigurazioni e per il supporto della negoziazione di risorse e compiti.

L'ulteriore passo evolutivo potrebbe essere costituito dal **paradigma a componenti** e/o dal **paradigma ad agenti**, di cui si discuterà tra poco.

Dagli oggetti a UML

Gli oggetti risultano utili per modellare entità del mondo reale (persone, clienti, veicoli, etc.) funzioni e processi complessi (automi, centri di servizio) o artefatti (figure grafiche, stack, buffer, etc.). Gli oggetti divengono anche elementi fondanti per la risoluzione di problemi interattivi (si pensi ad esempio al problema della prenotazione dei posti aerei) che non hanno specifiche complete, né una definizione precisa di correttezza o di complessità. Come i sistemi software in generale, gli oggetti richiedono inoltre la specifica di un contratto che si prolunga nel tempo.

L'interpretazione di ereditarietà come concetto legato alla classificazione delle entità di un sistema apre nuove dimensioni progettuali in relazione alla descrizione del dominio di un problema in termini di gerarchie (tassonomie) di entità e concetti. Inoltre essa permette la specifica di tipi parziali, attraverso la introduzione di **classi astratte**, cioè di classi non completamente specificate che affidano il completamento delle specifiche strutturali e comportamentali alle proprie sottoclassi, promuovendo in tal modo metodologie di progetto e costruzione orientate al cambiamento.

I linguaggi ad oggetti promuovono dunque un forte spostamento dal piano dei meccanismi al piano dei concetti e progressivamente si comprende che costrutti nati per migliorare pragmaticamente la qualità del prodotto software e per promuovere la riusabilità del codice possono svolgere un ruolo ancor più fondamentale nella fase di progetto dei sistemi.

Tra le numerose metodologie che vengono proposte per il progetto basato su oggetti, quali Catalysis, Shaere/Mellor, Fusion, quelle proposte da Grady Booch (OOSE, Object Oriented Software Engineering), James Rumbaugh (OMT, Object Modeling Technique) e Ivar Jacobson (Objectory) iniziano nel 1994 un processo di reciproca integrazione, che culmina nella definizione della proposta unificata **Rational Unified Process (RUP)** e di un linguaggio grafico per la descrizione di modelli concettuali: l'**Unified Modeling Language (UML)**.

I termini dello spazio concettuale scaturito dalla programmazione ad oggetti riflessi in **UML** costituiscono oggi gli elementi di base con cui esprimere l'organizzazione e il funzionamento dei sistemi software. Adottando questi elementi sia per l'analisi sia per il progetto, l'ingegnere del software affronta un problema in termini di reti (grafi) di oggetti, tra loro interagenti attraverso interfacce accuratamente progettate per garantire indipendenza dalla struttura interna.

Componenti software

Il modello di programmazione ad oggetti promuove la progettazione e lo sviluppo di sistemi software modulari e permette di superare il tradizionale approccio monolitico alla costruzione del software migliorando la adattabilità, scalabilità e manutenibilità dei sistemi. Tuttavia, fin dal 1994 alcuni autori hanno rilevato che:

Object orientation has failed but component software is succeeding (Udell)

Il concetto di oggetto non è infatti di per sé sufficiente a garantire processi di progettazione e di sviluppo in cui il sistema finale possa venire ottenuto come composizione di parti ad alta riusabilità già definite e disponibili. Nonostante i termini componente e oggetto siano spesso usati in modo intercambiabile, il concetto di componente software mira a denotare un ente che può essere prodotto acquisito e

distribuito in modo indipendente e che può interagire con altri componenti per formare un sistema funzionante.

Parlare di componenti software significa allargare lo spazio concettuale di riferimento, introducendo caratteristiche e proprietà diverse da quelle che caratterizzano gli oggetti. In generale si può affermare che gli oggetti formano un mezzo particolarmente appropriato per costruire componenti. Come già avvenuto per gli oggetti, non esiste ancora una definizione di componente software universalmente accettata.

Nel 1996 la European Conference of Object Oriented Programming (ECOOP) coniò la seguente definizione: *A software component is a unit of composition with contractually specified interface and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*

Con questa definizione, un componente software viene inteso come una entità a sé stante e interconnnettibile che permette la costruzione di sistemi per aggregazione di parti costruite in modo indipendente da fornitori diversi. Dunque, in generale, un oggetto non può dirsi automaticamente un componente.

La definizione pone anche in evidenza il ruolo della interfaccia come unica possibile "via di accesso" ad un componente e i requisiti che devono essere rispettati dall'ambiente esterno al componente (contesto) affinche il componente possa esistere e funzionare. Per realizzare un sistema come aggregazione di componenti occorre definire e realizzare in primo luogo una infrastruttura e un insieme di regole atte a garantire la interazione e il coordinamento delle parti.

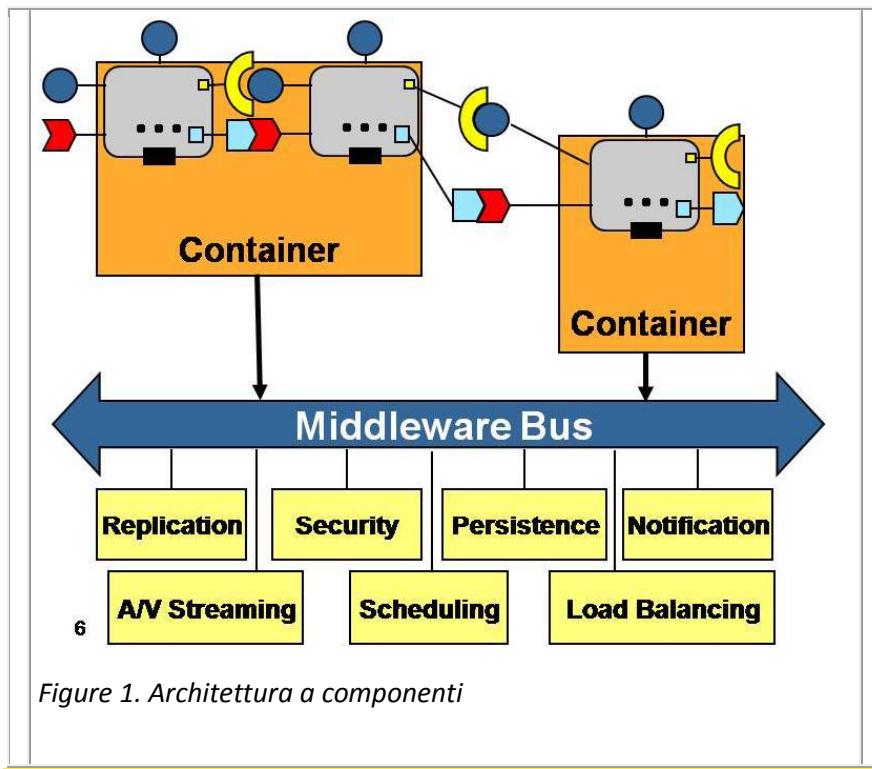


Figure 1. Architettura a componenti

Inversione del controllo

Una delle più importanti relazioni tra oggetti è costituita dalla relazione di dipendenza e in particolare dalla dipendenza di un oggetto composto dalle sue parti componenti o da altri oggetti che intende utilizzare.

L'idea di base della ***dependency injection*** è che il "popolamento" di un oggetto con una appropriata implementazione delle parti da cui dipende venga affettuata da un altro oggetto "assemblatore". Spesso si parla di ***inversion of control*** (IoC) in quanto non è più un oggetto che si preoccupa della costruzione delle parti da cui dipende, ma un oggetto esterno.

Sono state individuate tre forme di dependency injection:

- ***Constructor injection***, detta anche ***IoC di tipo 3***: la iniezione da parte dell'assemblatore avviene avvalendosi del costruttore di un oggetto;
- ***Setter injection***, detta anche ***IoC di tipo 2***: la iniziazione avviene tramite modificatori (detti anche ***setter methods***, si veda [Comportamento](#)) dell'oggetto.
- ***Interface injection***, detta anche ***IoC di tipo 1***: la iniziazione avviene definendo ed utilizzando interfacce per l'iniezione.

L'inversione del controllo è uno dei meccanismi più usati nelle infrastrutture a componenti, di cui si discuterà nella sezione [I componenti software](#).

Il paradigma ad agenti

Gli agenti e i Sistemi Multi-agente (**MAS**) si sono dimostrati una tecnologia in grado di far fronte alla complessità degli odierni scenari **ICT**. Diverse esperienze industriali hanno già mostrato apprezzamento per l'impiego delle tecnologie ad agenti nei processi manifatturieri [Bussmann98], nei Web services, nei marketplace virtuali basati sul Web [Kephart02] e nella gestione delle reti distribuite [Timon03]. Diversi altri studi suggeriscono l'adozione delle tecnologie ad agenti e dei **MAS** come tecnologia abilitante per diversi futuri scenari come per esempio il pervasive computing, il Grid computing e il semantic Web.

Nella comunità scientifica si sta diffondendo la convinzione che i **MAS** siano più di una semplice tecnologia, e che rappresentino invece un nuovo paradigma general-purpose per lo sviluppo del software [Jennings01], [Zambonelli03]. La computazione basata sugli agenti promuove la progettazione e lo sviluppo di applicazioni in termini di entità software autonome (agenti), situate in un ambiente che portano a termine i loro obiettivi attraverso l'interazione con le altre entità autonome e con l'ambiente adottando protocolli e linguaggi di comunicazione di alto livello. Queste caratteristiche ovviamente rendono i **MAS** adatti ad affrontare lo sviluppo dei moderni scenari:

- l'autonomia delle entità dell'applicazione riflette la natura intrinsecamente decentralizzata dei moderni sistemi software distribuiti e può essere considerata come la naturale estensione delle nozioni di modularità e encapsulamento per i sistemi;
- il modo flessibile in cui gli agenti operano e interagiscono (sia gli uni con gli altri che con l'ambiente) è adatto a scenari dinamici e impredicibili dove il software andrà a lavorare;
- il concetto di ***agency*** fornisce una visione unificata dei risultati provenienti dall'Intelligenza Artificiale rendendo gli agenti e i **MAS** le entità più adatte a cui attribuire un comportamento "intelligente" [Zambonelli04].

Negli ultimi anni, insieme alla maggiore accettazione delle tecnologie ad agenti come nuovo paradigma dell'ingegnerizzazione del software, sono cresciuti gli sforzi nella ricerca per l'identificazione e la definizione di nuovi modelli e tecniche per lo sviluppo dei sistemi complessi per mezzo delle tecnologie **MAS**. Queste ricerche propongono una molteplicità di metafore e approcci formali per lo sviluppo di nuove metodologie e tecniche di modellazione particolarmente adatti per il paradigma orientato agli agenti.

Gli agenti

Al momento esiste ancora un didattito insoluto nella comunità scientifica riguardo alla definizione di "agente". Una caratterizzazione molto utile comunque risulta la seguente [Jennings99]: *an agent is an encapsulated computer system that is situated in some environment, and that is capable of flexible, autonomous action in that environment in order to meet its design objectives*

Questo significa che gli agenti sono:

- entità ben definibili risolutrici di problemi con "contorni" e interfacce ben definite,
- situati in un particolare ambiente: essi ricevono input relativi allo stato dell'ambiente attraverso sensori e attuano sull'ambiente attraverso specifici attuatori,
- progettati per adempiere ad uno specifico ruolo: essi hanno particolari obiettivi da portare a termine, che possono essere rappresentati sia implicitamente che esplicitamente all'interno degli agenti,
- autonomi: essi controllano il proprio stato interno e il proprio comportamento,
- capaci di esibire un comportamento flessibile di risoluzione dei problemi: essi devono essere
 - **reattivi**: capaci di rispondere in un tempo limitato ai cambiamenti che avvengono nell'ambiente al fine di portare a termine i propri obiettivi per i quali sono stati progettati
 - **proattivi**: capaci di adottare opportunisticamente nuovi goal e prendere l'iniziativa al fine di soddisfare gli obiettivi per i quali sono stati progettati.

Quando si adotta una visione del mondo orientata agli agenti diventa subito ovvio che un singolo agente è insufficiente, infatti molti problemi richiedono l'impiego di diversi agenti: per rappresentare la natura decentralizzata del problema, il controllo distribuito, le prospettive multiple o gli interessi in competizione. Inoltre, gli agenti dovranno interagire gli uni con gli altri sia per portare a termine obiettivi individuali che sociali o anche per gestire le dipendenze che nascono dall'essere situati nello stesso ambiente.

Agenti vs. oggetti

Anche se ci sono certe somiglianze tra gli approcci orientati agli oggetti e quelli orientati agli agenti (entrambi adottano il principio dell'information hiding e riconoscono l'importanza cruciale della modellazione dello spazio delle interazioni) ci sono anche svariate importanti differenze [Jennings01]. Primo, gli oggetti sono generalmente passivi: necessitano di ricevere un messaggio (o una chiamata a metodo) per ottenere il controllo dell'esecuzione, gli agenti invece sono entità attive. Secondo, anche se gli oggetti incapsulano lo stato e la realizzazione del comportamento essi non incapsulano l'attivazione del comportamento (scelta delle azioni): l'oggetto può così invocare ogni metodo pubblico di ogni altro oggetto e una volta invocato il metodo le azioni associate ad esso possono essere eseguite, è invece parte della natura degli agenti la scelta del tipo di azione da eseguire e se eseguire tale azione.

In più, il paradigma orientato agli oggetti fallisce nel fornire una adeguato insieme di concetti e meccanismi per modellare i sistemi complessi: per questo tipo di sistemi come ricorda Booch [Booch94] *le classi, gli oggetti e i moduli forniscono essenziali ma insufficienti astrazioni*. Gli oggetti individuali presentano comportamenti a grana troppo fine e l'invocazione di metodi è un meccanismo troppo primitivo per descrivere il tipo di interazioni che avvengono nei sistemi complessi. Il riconoscimento di queste mancanze ha portato allo sviluppo di più potenti meccanismi di astrazione a supporto della

progettazione dei sistemi complessi come per esempio i design pattern, i framework applicativi e le tecnologie a componenti.

Anche se questi sono stati indubbiamente passi avanti si sono presto rivelati non del tutto adatti nel soddisfacimento delle caratteristiche necessarie per lo sviluppo dei sistemi complessi: la focalizzazione sulle funzionalità generiche e i rigidi pattern di interazione si scontrano con la natura impredicibile e dinamica dei sistemi complessi. Inoltre, gli approcci orientati agli oggetti forniscono solo un supporto minimale per specificare e gestire le relazioni organizzazionali.

Riassumendo, la visione tradizionale dei sistemi orientati agli oggetti e quella orientata agli agenti hanno almeno tre differenze cruciali [Wooldridge00]:

- gli agenti incorporano una forte nozione di autonomia rispetto agli oggetti, in particolare essi decidono autonomamente se eseguire o meno azioni in risposta a richieste provenienti da altri agenti
- gli agenti sono capaci di esibire un comportamento (reattivo, proattivo, sociale,...) flessibile mentre gli oggetti tipicamente forniscono un comportamento rigido
- i sistemi multi-agente sono intrinsecamente multi-thread, quindi ogni agente ha il proprio flusso di controllo, nei sistemi ad oggetti questo non è sempre il caso in quanto tipicamente il flusso controllo passa da un oggetto ad un altro.

Il linguaggio UML

Il linguaggio **UML** fornisce oggi lo standard notazionale per la rappresentazione dei modelli di un sistema software. **UML** nasce nel 1997 dal lavoro congiunto di Booch, Rumbaugh e Jacobson attingendo da concetti relativi alla metodologia di Booch, all'**OMT** di Rumbaugh e all'**OOSE** di Jacobson. (si veda [Breve storia delle metodologie](#)).

La specifica di UML nella versione 2.2.1 è organizzata in due volumi: **UML 2.1.1: Infrastructure** e **UML 2.1.1: Superstructure**; il primo volume definisce i costrutti fondamentali del linguaggio, mentre il secondo definisce i costrutti a livello utente.

La specifica UML2.2.1 è definita usando un approccio basato sul concetto di **metamodello** piuttosto che sull'idea classica di grammatica; in altre parole il linguaggio è definito da un modello che descrive gli elementi che lo caratterizzano. Ciò non preclude la possibilità che UML venga descritto in futuro anche con linguaggi formali a base matematica come **Object-Z** o **VDM** (si veda Unified Modeling Language: Infrastructure 2.1.1 pg. 6).

Il metamodello di UML è costituito da **Meta Object Facility** (si veda [UML2.0](#)) un linguaggio definito da **OMG**, che ha introdotto anche il linguaggio **QVT** (**Query, Views and Transformation**) come linguaggio per standardizzare le trasformazioni tra modelli.

E' molto importante sottolineare che:

- UML è un linguaggio (o una famiglia di linguaggi) e non una metodologia.
- UML propone un ricco insieme di elementi a livello utente; tuttavia è alquanto informale sul modo di utilizzare al meglio i vari elementi. Ad esempio gli **state diagrams** possono essere usati per descrivere **use cases**, sottosistemi ed oggetti; ciò implica che per comprendere un diagramma un lettore deve conoscere il contesto in cui esso è collocato.
- Non tutti gli elementi di UML sono associati ad una semantica operazionale. E' il caso ad esempio degli elementi **component** e **deployment**.

- Un modello UML può essere introdotto con diverse finalità: per *descrivere un concetto*, per *descrivere una specifica* o per *descrivere una implementazione*. Nel caso si voglia definire un modello a livello concettuale, occorre districarsi tra le molteplici notazioni che UML fornisce per selezionare solo quelle utili a produrre un artefatto utile e comprensibile a quel livello.

Il concetto di modello

Nel linguaggio comune il termine **modello** è spesso usato per denotare un'astrazione di qualcosa che esiste nella realtà, come ad esempio il modello che posa per un artista, una riproduzione in miniatura, un esempio di modo di svolgere un'attività, una forma da cui ricavare vestiti, un ideale da seguire, etc.. Alcuni (tra cui gli ingegneri) intendono per **modello** un sistema matematico o fisico che ubbidisce a specifici vincoli e che può essere utilizzato per descrivere e comprendere un sistema (fisico, biologico, sociale, etc.) attraverso relazioni di analogia.

Nel contesto dei processi di costruzione del software, il termine **modello** va primariamente inteso come *un insieme di concetti e proprietà volti a catturare aspetti essenziali di un sistema, collocandosi in un preciso spazio concettuale*. Per l'ingegnere del software quindi un modello costituisce una visione semplificata di un sistema che rende il sistema stesso più accessibile alla comprensione e alla valutazione e facilita il trasferimento di informazione e la collaborazione tra persone, soprattutto quando è espresso in forma visuale.

Nel concepire un modello come visione semplificata di un sistema software si assume che il sistema abbia già una sua esistenza concreta. In alcune fasi di lavoro (in particolare nella fase di analisi) *il sistema è il modello*; un raffinamento o una variazione del modello corrisponde in questo caso ad una variazione del sistema.

La produzione esplicita di modelli si rivela utile in quanto i diversi attori di un processo di produzione di software (committenti, analisti, progettisti, utenti, etc) operano a diversi livelli di astrazione. Definendo opportuni modelli del sistema da realizzare, in ogni fase del processo di produzione l'attenzione può essere focalizzata sugli aspetti rilevanti in quella fase, utilizzando una forma di comunicazione comprensibile ad attori diversi. Per garantire coesione e interoperabilità, si cerca di individuare regole di corrispondenza e di trasformazione automatica tra modelli (si veda [Architecture model-driven](#)).

UML2.0

L'Unified Modeling Language (**UML**) è un linguaggio la cui sintassi e semantica non viene descritta usando le convenzionali notazioni (**BNF**, **EBNF**, etc.) usate per i linguaggi testuali. Il meccanismo usato per descrivere **UML** prende il nome di metamodellazione (**metamodeling**) in quanto **UML** è definito da un modello.

Il modello definito per esprimere modelli si dice **metamodello** e descrive di fatto un linguaggio, spesso limitandosi alla sintassi astratta e alla semantica senza escludere però che possa introdurre anche la definizione di una sintassi concreta.

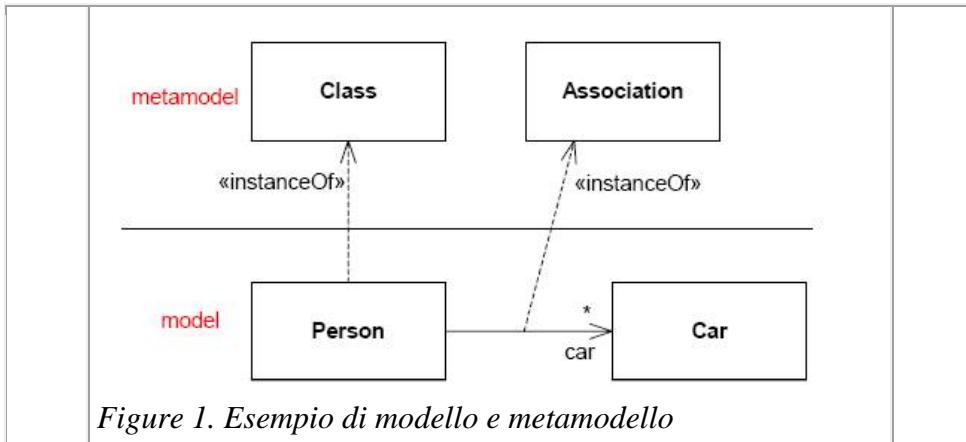


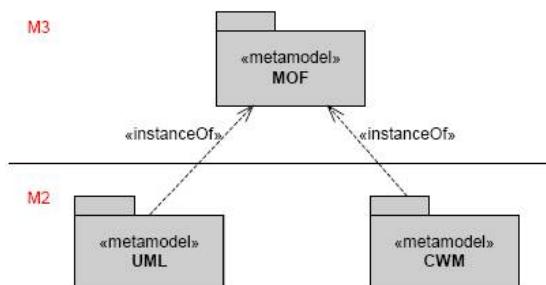
Figure 1. Esempio di modello e metamodello

Ai fini pratici, un linguaggio e il modello che lo definisce possono essere considerati equivalenti. Un metamodello può ovviamente avere un meta-metamodello e così via, senza limite prestabilito.

La specifica di UML nella versione 2.2.1 è organizzata in due volumi: [UML 2.1.1: Infrastructure](#) e [UML 2.1.1: Superstructure](#); il primo volume definisce i costrutti fondamentali del linguaggio, mentre il secondo definisce i costrutti a livello utente.

I livelli di UML2

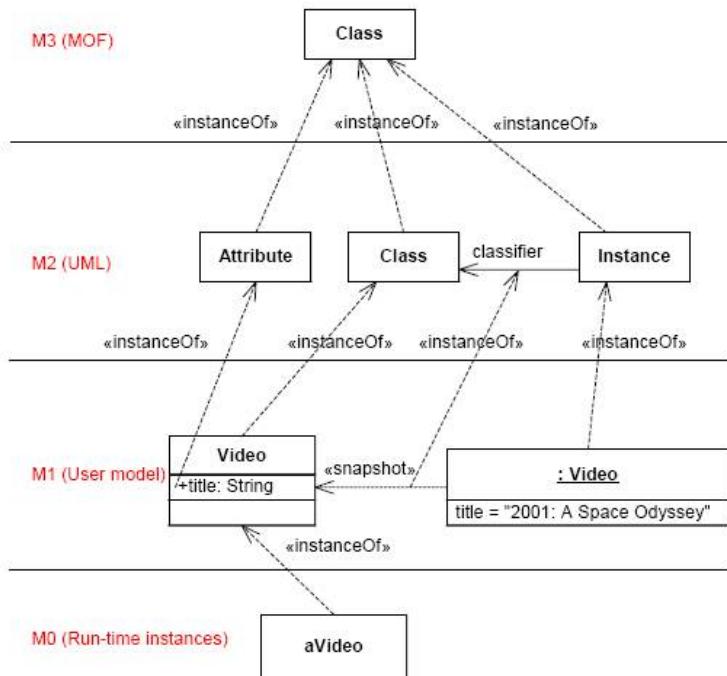
[OMG](#) ha organizzato l'architettura logica di modellazione in [UML](#) su una struttura a quattro livelli:



- **Livello M0:** il livello del [*sistema finale*](#);
- **Livello M1:** il livello del [*modello*](#) che descrive il sistema;
- **Livello M2:** il livello del [*metamodello*](#) che descrive gli elementi con cui esprimere un modello.
- **Livello M3:** il livello di [*meta-metamodello*](#) che descrive gli elementi con cui esprimere un meta-modello.

Il livello M3 di [UML](#) è il [*meta-metamodel layer*](#) di [OMG](#) denominato [*Meta-Object Facility*](#) (MOF). MOF è definito in termini di sè stesso. La spiegazione relativa a queste gerarchie di meta-livelli è fornita nella Section 7.6 della Infrastructure specification insieme al seguente esempio (figura 7.8 pag 19):

Figure 2. Esempio di livelli UML



L'esempio mostra che l'elemento **Video** definito dall'utente è una istanza dell'elemento **Class** definito in **UML2**, il quale a sua volta è una istanza dell'elemento **Class** definito in **MOF**.

I concetti chiave utilizzati nella modellazione sono il **Classificatore**, e le sue successive **istanze** nei diversi livelli che sono la **Classe** e l'**Oggetto** come istanza della classe (di cui la **Class** risulta essere il **classifier**). Un altro concetto chiave è la possibilità di navigare da una istanza al suo "meta-oggetto" (il suo **classifier**). Questo concetto fondamentale può essere usato per gestire qualsiasi numero di livelli (chiamati "metalivelli") per mezzo della **Reflection** che permette di muoversi trasversalmente ed in modo ricorsivo in ogni numero di metalivelli.

Meta Object Facility

MOF fornisce un framework concettuale per la gestione dei meta-dati e per la costruzione di un insieme di servizi interoperabili per lo sviluppo e di modelli e di sistemi **metadata-driven**. Esempi di questi sistemi sono tools di sviluppo, sistemi di data warehouse, metadata repositories etc. Inoltre differenti tecnologie standardizzate da OMG, tra cui UML,CWM, SPEM, XMI, e vari UML profiles, usano **MOF** le tecnologie derivate da **MOF** (tra cui in particolare XMI e JMI) per metadata-driven interchange e metadata manipulation. Grazie a **MOF** i modelli possono essere:

- esportati da una applicazione
- importati in un'altra
- trasportati attraverso la rete
- memorizzati in un repository e successivamente recuperati
- rappresentati in differenti formati
- trasformati
- usati per generare codice applicativo

MOF permette la portabilità delle rappresentazioni dei modelli, definendo come i modelli possano essere scambiati usando rappresentazioni **XML** nel formato **XML Metadata**

Interchange (XMI). XMI [XMI] è un mapping di MOF in XML che, dalla versione 2.0, include anche la serializzazione di informazioni relative al layout dei diagrammi. Inoltre MOF definisce anche interfacce riflessive (MOF::Reflection) che permettono introspezione non solo per MOF stesso, ma per ogni altro metamodello definito come istanza di MOF.

Profili

Gli elementi di UML permettono di esprimere un insieme di concetti general purpose. L'OMG, nell'ambito di MDA, affronta anche il problema di definire metamodelli per specifici domini, in forma di profili, denominati **core models**. Sono esempi di core models UML profile for EAI, relativo alle *Enterprise Application Integration* o UML profile for EDOC per sistemi distribuiti d'impresa.

Il concetto di profilo è stato introdotto in UML per permettere l'adattamento o l'estensione di UML verso specifiche esigenze professionali o tecniche. I profili si basano su tre tipi di artefatti: **stereotipi**, **tagged values** e **vincoli** (**constraints**). Nell'ambito di un profilo lo **stereotype** introduce un nuovo elemento, mentre un **tagged value** permette di introdurre proprietà aggiuntive.

Un profilo può essere visto come un linguaggio concreto nell'ambito della famiglia di linguaggi costituita da UML.

Un profilo dipende sempre da un reference metamodel, che può essere UML o un altro profilo. Un profilo non può modificare o rimuovere i vincoli del suo reference metamodel, ma solo restringerli ulteriormente. I vincoli sono espressi in OCL; ovviamente un vincolo definito in OCL in un modello di livello N riguarda le istanze del modello, cioè il livello N-1. I vincoli definiti a livello della definizione di uno stereotipo sono intesi che debbano essere applicati a tutte le classi in cui lo stereotipo è applicato.

Il concetto di profilo è definito in modo formale in UML2.0 (il che non è per UML1) che introduce anche notazioni specifiche, tra cui un simbolo per denotare l'**estensione**, un concetto del tutto nuovo che non deve essere confuso con l'associazione, la dipendenza o l'ereditarietà.

UML come famiglia di linguaggi

Il concetto di sintassi astratta e di semantica statica (si veda [Sintassi \(astratta\) e semantica](#)) sono al centro della definizione di UML attraverso costrutti quali **class**, **attribute**, **operation**, **association**. La sintassi concreta è espressa in forma grafica da rettangoli, frecce, etc. e in forma testuale da XMI [XMI]. Il linguaggio OCL (si veda [xref href="..//ocl/OCL.dita"/>](#)) permette di esprimere espressioni che fanno riferimento alla sintassi astratta di UML; tali espressioni sono usate per definire formalmente la semantica statica dei profili UML (si veda [Profili](#)).

Naturalmente per descrivere la sintassi astratta di un linguaggio occorre un linguaggio. Nel caso di UML questo linguaggio è MOF (si veda [Meta Object Facility](#)) cui è associata una sola sintassi concreta.

L'iniziativa MDA (si veda [Architecture model-driven](#)) si focalizza su UML e MOF come linguaggio per esprimere meta-modelli specificamente rivolti a un particolare dominio applicativo. Per questo motivo MOF costituisce un meta-metamodello, cioè un modo per esprimere un meta-modello. Ma MOF può essere considerato anche un linguaggio con cui

esprimere un **domain specific language** (DSL) la cui sintassi concreta dovrebbe essere basata (nella raccomandazione MDA) su profili UML.

Considerando UML come una **famiglia di linguaggi** piuttosto che un singolo linguaggio, un profilo può essere visto come un linguaggio concreto nell'ambito di tale famiglia. Nella proposta MDA ogni DSL è predisposto per usare come base UML e espressioni OCL per specificare la sua semantica statica.

Azioni e comportamenti

MOF è definito formalmente in quanto è associato a una precisa semantica. La semantica statica dei metamodelli viene espressa tramite espressioni OCL. La definizione formale della semantica dinamica costituisce ancora un problema aperto.

Action semantics

Per permettere la definizione di comportamenti, UML2 definisce la sintassi astratta (senza alcuna sintassi concreta) di elementi utili a formulare una **action semantics**.

Questi elementi, la cui semantica è espressa in linguaggio naturale, sono: le usuali operazioni aritmetico-logiche, feature tipiche di linguaggi sequenziali quali if, for, switch, la creazione-distruzione di istanze, la navigazione lungo associazioni tra classi, la generazione di istanze di associazioni (links), la generazione di segnali, la definizione di funzioni con parametri di input-output, timers e infine variabili (detti instance handle) ed operazioni di assegnamento e lettura di valori, anche i forme aggregate (sets, bags, sequences).

L'action semantics non contiene costrutti strutturali come classi e relazioni: i relativi elementi hanno senso solo in relazione ad altri elementi di un modello, quali ad esempio le operazioni di una classe o le azioni di una statemachine.

iUML è la versione realizzata da Kennedy Carter [www.kc.com] di xUML [MB02] una versione di UML privata degli elementi "semantically weak" (in pratica core UML) e aumentata dall'action semantics espressa dall'**Action Specification Language** (ASL) descritto in [MB02] capitolo 10. iUML fornisce una sintassi concreta per l'action semantics, nel contesto di una linea di ricerca volta a rendere eseguibili i modelli UML definiti in base ai soli elementi cui può essere associata una semantica di esecuzione.

Il comportamento

Il comportamento può essere implicito nel significato degli elementi di un metamodello e può essere vincolato da frasi OCL che possono portare alla generazione (anche automatica) di codice di controllo/collaudo.

In generale, una grande parte del comportamento del sistema può essere comportamento definito dall'architettura.

Per esprimere il comportamento UML offre **activity diagrams** e **state diagrams** e permette di introdurre notazioni in forma di **description tags** per indicare alternative (ad es. comportamento sincrono / asincrono). A questo approccio descrittivo / configurativo si contrappone un approccio creativo in cui il comportamento è espresso da sistemi formali, quali ad esempio **state charts** [HP88]. Il vantaggio di questo approccio è che è chiaro come passare dalla specifica alla realizzazione, anche in modo automatizzato attraverso

tools. Il limite è costituito dal fatto che il formalismo potrebbe essere non adeguato alle esigenze di specifica di comportamento in un dato dominio.

Nel caso non si trovi un formalismo adeguato, si potrebbe pensare di definire un nuovo linguaggio; il problema è che non è facile specificare la semantica di questo nuovo linguaggio, passo necessario cui ricavare l'implementazione (engine del formalismo). Un ulteriore modo è fare ricorso a un linguaggio computazionalmente completo, ad esempio un linguaggio basato sulla action semantics.

La specifica di un comportamento deve essere sempre associata a un elemento strutturale. Questo fatto è la ragione per cui si introducono spesso "behavioral wrappers", cioè sottotipi di elementi strutturali dedicati a svolgere un dato comportamento espresso con un certo formalismo. L'implementazione del comportamento può essere basato su interpreti, capaci ad esempio di leggere ed eseguire una state machine descritta in [XML](#).

L'Object Constraint Language

Un vincolo ([constraint](#)) è una regola che restringe l'insieme dei valori assumibili da attributi e relazioni in un modello.

L'[Object constraint Language](#) (OCL) è stato introdotto per esprimere vincoli sul modelli UML, sia a livello [M1](#) che [M2](#). In OCL i vincoli sono denotati da espressioni dichiarative "pure" (cioè prive di effetti collaterali) e "tipate" basate sugli attributi e sulle relazioni definite in un modello.

Per la definizione formale di OCL si veda [ocl-definition.pdf](#).

Per esempi d'uso di OCL si veda xref href="..//lab/agentModel.dita"/>

Il ruolo dell'architettura

La moderna costruzione del software riconosce all'[architettura del sistema](#) un ruolo strategico, nonostante il termine [architettura](#) sia tra i vocaboli più sovraccarichi di significato.

Normalmente, si parla di [architettura di un sistema](#) quando ci si vuole riferire all'insieme delle macro-parti in cui il sistema si articola, includendo le loro responsabilità, relazioni e interconnessioni. Per molti il termine [architettura](#) potrebbe però evocare l'immagine di uno schema in cui compare una rete di blocchi e linee di connessione; questa visione andrebbe meglio indicata col termine [mappa](#) o "topologia". Per altri l'[architettura](#) evoca l'idea di uno schema concettuale di soluzione riferito a un certo dominio applicativo, come ad esempio nella frase [architetture web](#); in questo caso sarebbe più appropriato utilizzare il termine [framework](#).

L'[Open Group Architectural Framework](#) definisce architettura "*a set of elements depicted in an architectural model and a specification of how these elements are connected to meet the overall requirements of an information system*".

In [BCK03] si dice che "*the software architecture of a program or computing system is the structure or structures of the system, which comprises software components, the externally-visible properties of these components and the relationships among them*".

La IEEE Computer society definisce (nel 2000) l'architettura "*the fundamental organization of a system embodied in its components their relationships to each other and to the environment, and the principles guiding its design and evaluation*".

Tra le altre accezioni possibili, una delle più curiose, su cui vale la pena di riflettere, è quella per cui *l'architettura è ciò che rimane di un sistema quando non si può più togliere nulla, continuando a comprenderne la struttura e il funzionamento*.

Le prime esperienze collettive nello studio delle architetture software possono essere fatte risalire al workshop OOPSLA del 1981 guidato da Bruce Anderson che mirava allo sviluppo di un "architecture handbook". A questo periodo può anche essere fatto risalire l'idea di *pattern* culminata nella pubblicazione nel 1995 dell'ormai famoso testo sui *Design Pattern* [GHJV95] della così detta GoF (*Gang-of-Four*: Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides). Da allora si sono susseguiti molte altre conferenze e lavori. I riferimenti più noti sono i cinque testi sulle *Patten Software Architectures* ([POSA1], [POSA2], [POSA3], [POSA4], [POSA5]) e i convegni PLoP (*Pattern Languages of Programming*).

Lo stile architetturale

Tra le molte possibili interpretazioni del termine *architettura*, vi è l'idea che parlare di architettura significhi fare riferimento ad un insieme di regole e schemi (*pattern*) che riducono i gradi di libertà di costruzione di un sistema. Questa interpretazione è piuttosto astratta, ma risulta di crescente importanza nella moderna produzione del software, tanto che nel seguito, per richiamarla, parleremo di *Architettura* di un sistema.

Il ruolo primario di un'Architettura è quello di fornire un riferimento nella progettazione e una guida per il progettista, mitigando, con le proprie regole e pattern, il rischio di una "creatività inconcludente".

L'insieme di principi, regole e pattern di un'Architettura concorre a formare uno spazio concettuale che viene anche indicato con il termine *stile architetturale*.

Lo *stile architetturale* definisce una famiglia di sistemi software in termini della loro organizzazione strutturale (e comportamentale). Come tale lo stile architetturale definisce una tecnica di progettazione / costruzione indipendente da ogni specifico contesto applicativo.

Architecture model-driven

Il processo noto come *Model Driven Architecture* (MDA) si basa sull'idea di modellare diversi aspetti e diversi livelli di astrazione di un sistema, sfruttando regole di corrispondenza tra modelli per garantire coesione e interoperabilità.

La *MDA-guide* caratterizza l'*MDA* attraverso una precisa sequenza di fasi:

1. specifica di un sistema software in modo indipendente dalla piattaforma di supporto;
2. specifica di un insieme di piattaforme di supporto o selezione di specifiche di piattaforme già esistenti;
3. selezione di una particolare piattaforma;
4. trasformazione della specifica del sistema specifica nelle specifica relativa alla piattaforma selezionata.

La guida denota con il termine **PIM** (*platform independent model*) un modello che descrive un sistema in modo indipendente da ogni piattaforma operativa e con il termine **PSM** (*platform specific model*) un modello che descrive un sistema con piena conoscenza di una particolare piattaforma operativa. Un **PSI** (*platform specific implementation*) è una possibile espressione di un PSM.

L'attenzione di MDA si focalizza sui linguaggi di descrizione dei modelli e nella introduzione di **regole di trasformazione** capaci di trasformare un **PIM** in uno o più **PSM** attraverso l'uso di appositi tools.

Per rendere eseguibile le trasformazioni tra modelli da parte di una macchina è essenziale che i linguaggi usati per definire i modelli abbiano una definizione formale. questo obiettivo viene oggi ottenuto derivando i linguaggi di descrizione dei modelli dal linguaggio **MOF** (si veda [Meta Object Facility](#)) definito da [OMG](#), che ha introdotto anche il linguaggio **QVT** (*Query, Views and Transformation*) [QVT] come linguaggio per standardizzare le trasformazioni.

Un altro punto importante è che MDA fonda la definizione dei linguaggi sul concetto di **metamodello** piuttosto che sull'idea classica di grammatica; in altre parole un linguaggio è definito da un modello che descrive gli elementi che lo formano (si veda [xref href="..//uml/introUml.dita"/>](#))

Dimensioni

Sia nella fase di analisi che in quella di progetto, la descrizione di un sistema software può avvenire focalizzando l'attenzione su almeno tre diversi punti di vista:

1. l'organizzazione del sistema in parti (**struttura**);
2. il modo in cui le diverse parti scambiano informazione implicita o esplicita tra loro (**interazione**);
3. il funzionamento del tutto e di ogni singola parte (**comportamento**).

Questi punti di vista costituiscono tre indispensabili dimensioni in cui articolare lo spazio della descrizione del sistema, qualunque sia il linguaggio utilizzato per esprimere questa descrizione.

Costrutti per esprimere strutture (di dati e di controllo), forme di comportamento e meccanismi di interazione sono presenti in tutti i linguaggi di programmazione. Un punto importante consiste nel capire fino a che punto i costrutti di un linguaggio debbano influenzare il progettista (se non lo stesso analista). Fino alla fine degli anni 90 il linguaggio di programmazione è stato il veicolo principale per introdurre nuovi concetti sia sul piano computazionale sia sul piano della organizzazione del software.

L'avvento della programmazione ad oggetti sembra avere segnato il culmine di questo processo; un motivo può certo essere il raggiungimento di una sufficiente maturità nella capacità espressiva in ciascuna delle dimensioni citate. Tuttavia, il motivo principale della relativa (apparente) stagnazione nello sviluppo di nuovi linguaggi può essere ricondotto all'idea che un linguaggio non deve essere necessariamente accompagnato da una sintassi concreta ma può essere sufficiente definire una **sintassi astratta** utilizzando un meta-linguaggio come ad esempio **MOF**(si veda [Meta Object Facility](#)) unitamente alla semantica del linguaggio e a un framework ([oo](#)) di supporto.

Questa idea è sviluppata oggi con riferimento ai *domain specific languages* che approfondiremo più avanti (si veda xref href="..//lab/agentModel.dita"/>). Nel seguito di questo capitolo cercheremo invece di approfondire l'uso dei modelli e di UML nel processo di costruzione del software.

Struttura

UML affida la descrizione della parte statica di un sistema ai seguenti elementi strutturali:

- class, interface, collaboration, use case, active class, component, node

UML2.0 introduce nuovi concetti per descrivere l'architettura interna di un elemento:

- part, connector, port

ed *elementi di raggruppamento*:

- package, subsystem

La definizione strutturale riguarda in primo luogo l'architettura complessiva del sistema e poi, ricorsivamente, definisce la struttura di ogni sottosistema e di ogni elemento del sistema.

Per quanto riguarda l'architettura complessiva, vedremo come l'uso dei pattern languages (si veda xref href="..//pattern/PatternLanguages.dita"/>) possa risultare strategico per individuare lo *stile architettonico* (si veda xref href="StileArchitetture.dita"/>) più appropriato all'applicazione in esame.

Per impostare in modo sistematico la definizione a livello strutturale di un elemento può essere conveniente, sia in fase di analisi sia in fase di progetto, cercare di dare risposta ad alcune domande quali:

- l'elemento è *atonico o composto*? Nel caso sia composto quali sono le parti che lo formano?
- l'elemento è *dotato di stato modificabile*? In caso affermativo, quali sono le operazioni di modifica dello stato? (si veda la sezione sul comportamento)
- quali sono le *proprietà* dell'elemento, cioè quali *attributi* lo caratterizzano ?
- da quali altri elementi *dipende* e secondo quale forma di dipendenza?

Si noti che un elemento composto implica la definizione ricorsiva della struttura di ogni parte e la definizione di operazioni denominate *selettori*. Notiamo anche che l'individuazione di una struttura composta porta spesso alla individuazione di un insieme di *operazioni primitive* sulla base delle quali costruire ogni altra operazione di manipolazione/gestione dell'elemento. Riprenderemo queste osservazioni nella prossima sezione.

Interazione

Per descrivere l'interazione UML introduce event, interaction.

UML2.0 introduce una nuova forma di descrizione ad alto livello di interazioni: l'*Interaction Overview Diagram*. Questo diagramma combina elementi degli *activity diagrams* e dei *sequence diagrams* per mostrare il flusso di un sistema in esecuzione.

Le interazioni possono essere *sincrone* o *asincrone* e riguardare informazioni o *stream* di dati. In questo secondo caso esse possono essere anche *isocrone*.

- In una **interazione asincrona** la comunicazione è "bufferizzata" senza alcuna limitazione sulle dimensioni del buffer. L'emittente non deve attendere alcuna informazione di ritorno anche quando invia informazione ad uno specifico destinatario. Il ricevente attende solo quando il buffer è vuoto. Nel caso di stream, non vi sono vincoli di tempo per la ricezione.
- In una **interazione sincrona** la comunicazione avviene senza l'uso di alcun buffer. L'emittente e il destinatario scambiano informazione unificando concettualmente le proprie attività. Nel caso di stream, il destinatario si aspetta di ricevere i dati con un ritardo (**delay**) che non supera un massimo prefissato.
- Una **interazione isocrona** riguarda solo stream; il destinatario si aspetta di ricevere i dati con un delay compreso tra un minimo e un massimo.

Per impostare in modo sistematico la dimensione interazione è opportuno chiarire le diverse forme che questa può assumere. Nel seguito faremo riferimento alla seguente terminologia:

- **Evento (event)**: informazione emessa (più o meno consapevolmente) in modo asincrono da una sorgente senza alcuna particolare nozione di ricevente e senza alcuna aspettativa da parte dell'emittente.
- **Segnale (signal)**: informazione inviata in modo asincrono a $N (N \geq 1)$ destinatari, noti o meno all'emittente, con l'aspettativa che venga ricevuta da qualcuno, al fine di eseguire un'azione che potrebbe portare vantaggio all'emittente e/o al sistema nel suo complesso.
- **Messaggio (message)**: informazione inviata in modo asincrono a $N (N \geq 1)$ specifici destinatari, noti alla emittente, con l'aspettativa che questi lo ricevano e lo elaborino, senza attesa di una risposta esplicita.
- **Invito (invitation)**: messaggio inviato a $N (N \geq 1)$ destinatari, con l'aspettativa che almeno uno lo riceva e invii al mittente un messaggio di conferma.
- **Richiesta (request)**: messaggio inviato a uno specifico receiver; il contenuto del messaggio rappresenta la richiesta di esecuzione di una attività, con aspettativa da parte del sender che questa attività si concluda con una risposta pertinente alla richiesta.
- **Conferma (reply, acknowledgment)**: messaggio inviato da un receiver al sender di un invito; il contenuto del messaggio rappresenta un riconoscimento di avvenuta ricezione.
- **Risposta (response)**: messaggio inviato da un receiver al sender di una richiesta; il contenuto del messaggio rappresenta la risposta alla richiesta.
- **Risultato (result)**: messaggio inviato dal receiver di una richiesta ad uno o più destinatari, noti e meno; il contenuto del messaggio rappresenta la risposta alla richiesta.

Le interazioni vengono spesso suddivise secondo quattro pattern [POSA3]; con riferimento alla terminologia precedente

- **Fire and forget**: il caso di invio di eventi, segnali, messaggi.
- **Sync with server**: il caso di invio di invitation.
- **Poll objects**: il sender invia una request delegando ad un oggetto (**poll object**) la responsabilità di ricevere la risposta. Il sender usa il poll object per verificare ed acquisire la disponibilità della risposta.
- **Result callback**: il sender invia una request specificando un oggetto (**callback object**) che implementa un metodo che verrà invocato dal supporto non appena il receiver invierà la risposta.

Comportamento

Per descrivere il comportamento di un elemento, UML propone due tipi di diagrammi: **activity diagram**, **state diagram**.

UML2.0 introduce nuovi concetti quali: **composition**, **references**, **exceptions**, **loops** e **alternatives**.

I comportamenti sono di solito connessi alle operazioni associate ad un elemento; queste possono essere inquadrate in diverse categorie, tra cui;

- **primitive**: operazioni sulla base delle quali può essere realizzata ogni altra operazione relativa a quel componente;
- **proprietà**: forniscono un attributo o una proprietà ;
- **predicati**: forniscono un valore di verità su attributi e/o stato;
- **selettori**: forniscono (accesso a) una parte costituente;
- **operatori relazionali**: operazioni che decidono la verità di una relazione tra il componente e un altro componente;
- **modificatori**: modificano lo stato interno e assumono spesso la forma di **setter methods**;
- **convertitori**: convertono la rappresentazione interna in altra rappresentazione; in particolare ne forniscono la rappresentazione esterna (di solito in forma di stringa).

Notiamo che se un elemento è composto, esso potrebbe fornire non solo selettori, ma anche operazioni (detta **iniettori**) che permettono di attribuire in modo dinamico ad un elemento una sua parte costitutiva, come completamento della fase di costruzione dell'elemento. Queste operazioni sono particolarmente presenti adottando schemi di inversione del controllo (si veda [xref href="..//paradigmi/InversioneControllo.dita"/>](#)).

Un componente privo di modificatori può essere considerato "puramente funzionale". Un componente privo di selettori può essere considerato "atomico".

Per individuare le **operazioni primitive** di un elemento composto può essere utile osservare che disporre di operazioni per disgregare una struttura negli elementi componenti e per ricomporla è tutto ciò che occorre per poter operare con essa disponendo di un linguaggio general-purpose computazionalmente completo. Infatti, una volta disgregata la struttura sarà possibile ricreare la stessa struttura oppure una struttura diversa, vuoi per una diversa composizione dei suoi elementi, vuoi per l'introduzione di nuovi elementi.

Interazione in UML 2.0

Nella progettazione di un sistema software la dimensione dell'interazione gioca un ruolo sempre più critico per il corretto sviluppo del sistema. In particolare la modellazione delle interazioni viene utilizzata in diverse situazioni. Per esempio possono essere usate da un team di progettisti per raggiungere una prima visione di alto livello condivisa sulle comunicazioni in un sistema software, oppure in fase più avanzata per progettare al meglio i protocolli di comunicazione. La modellazione delle interazioni può essere anche ampiamente utilizzata in fase di test del sistema, dove le effettive comunicazioni tra le entità possono essere tracciate e confrontate con le versioni iniziali dei modelli dell'interazione per vedere se i requisiti iniziali di funzionamento sono stati rispettati.

Data l'importanza strategica della modellazione delle interazioni, è bene cercare di capire come il linguaggio UML 2.0 possa essere utilizzato per modellare al meglio le comunicazioni in un sistema. L'obiettivo di questo studio non è solo capire come UML 2.0

possa modellare le interazioni, ma è anche quello di comprendere quali siano i punti di forza di questo strumento e quali le sue limitazioni.

Per fare questo illustreremo dapprima i diagrammi messi a disposizione da UML 2.0, dedicando particolare attenzione ai **Diagrammi di sequenza** e poi illustreremo con alcuni esempi se sia possibile o meno mappare sui diagrammi di sequenza l'**Interaction Specific Language** presentato nel capitolo X.

UML 2.0 e Interazione

UML 2.0 mette a disposizione il package **Interaction** per descrivere i concetti necessari per esprimere le interazioni in base al loro scopo. E' possibile esprimere un'interazione attraverso diversi tipi di diagrammi, ognuno dei quali fornisce capacità di rappresentazione differenti che lo rendono più adatto a per certi tipi di situazioni:

- **Diagrammi di Sequenza**: si concentrano sullo scambio di messaggi tra Lifeline.
- **Diagrammi di Comunicazione**: mostrano le interazioni dal un punto di vista architettonale dove gli archi tra le Lifeline sono decorati con la descrizione dei messaggi e dalla loro sequenza.
- **Diagrammi di Interaction Overview**: mostrano l'interazione attraverso un'overview del flusso di controllo nel sistema. Tali diagrammi hanno elementi notazionali che li rendono spesso simili ai diagrammi delle attività, occorre però fare attenzione in quanto la semantica di tali elementi è differente tra i due diagrammi.
- **Diagrammi Temporali**: rappresentano l'evoluzione di una entità nel tempo.

Nel seguito verrà illustrato in dettaglio la struttura del diagramma di sequenza, per i dettagli relativi agli altri tipi di diagrammi si invita il lettore a fare riferimento alla specifica SuperStructure di UML 2.0 capitolo 14.

Diagramma di Sequenza

Il tipo più comune di diagramma di interazione è il **Diagramma di sequenza** che si concentra sullo scambio di **messaggi** tra un numero di **Lifeline**.

Nella parte restante di questa sezione presenteremo la struttura dei diagrammi di sequenza, tralasciando volutamente la descrizione dei Messaggi e dei Combined fragment (novità introdotta in UML 2.0) che rappresentano il cuore dei diagrammi e saranno illustrati nelle sezioni successive. Il capitolo si concluderà con la presentazione di alcuni esempi di modellazione che metteranno in luce punti di forza e di debolezza dei diagrammi di sequenza.

Partecipanti

In un diagramma di sequenza, i partecipanti solitamente sono istanze di classi UML caratterizzate da un nome. La loro vita, cioè il tempo che intercorre dalla loro creazione alla loro distruzione, è rappresentata da una **Lifeline**, cioè una linea tratteggiata verticale ed etichettata, in modo che sia possibile comprendere a quale componente del sistema si riferisce.

In alcuni casi il partecipante non è un'entità semplice, ma composta: è possibile modellare la comunicazione fra più sottosistemi, assegnando una lifeline ad ognuno di essi. Il dettaglio di ciascuna parte viene poi definito utilizzando altri diagrammi, che

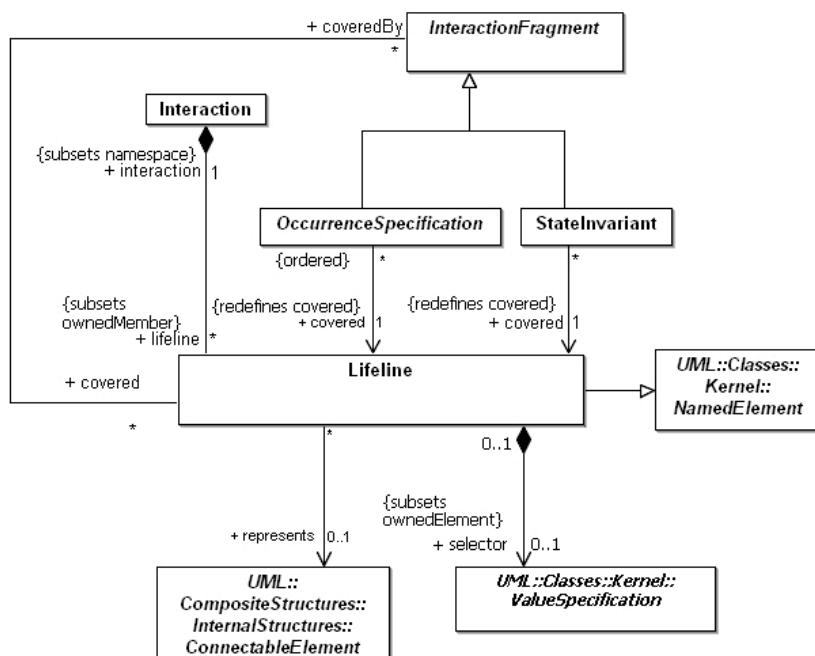
mostrano come ogni elemento costituente il gruppo concorre alla gestione dello stimolo ricevuto; questa pratica è definita decomposizione.

Dalla specifica UML SuperStructure (pag. 490) la definizione di **Lifeline** risulta essere la seguente:

A lifeline represents an individual participant in the Interaction. While Parts and StructuralFeatures may have multiplicity greater than 1, Lifelines represent only one interacting entity. Lifeline is a specialization of NamedElement. If the referenced ConnectableElement is multivalued (i.e. has a multiplicity > 1), then the Lifeline may have an expression (the "selector") that specifies which particular part is represented by this Lifeline. If the selector is omitted, this means that an arbitrary representative of the multivalued ConnectableElement is chosen.

Il meta-modello della Lifeline è riportato nella figura sottostante.

Figure 1. Meta-modello Lifeline (tratto da UML SuperStructure pag. 461)



L'ordine in cui delle OccurrenceSpecification (cioè l'invio e la ricezione di eventi) avvengono lungo la Lifeline rappresenta esattamente l'ordine in cui tali eventi di verificano. D'altra parte la distanza (in termini grafici) tra due eventi non ha rilevanza dal punto di vista semantico. La semantica di una Lifeline (all'interno di una Interazione) è la semantica dell'Interazione nella quale vengono selezionati solo gli OccurrenceSpecification relativi alla Lifeline in esame.

Dal punto di vista notazionale, una Lifeline è rappresentata da un rettangolo che costituisce la "testa" seguito da una linea verticale (che può essere tratteggiata) che rappresenta il tempo di vita del partecipante. Le informazioni che identificano il partecipante sono mostrate all'interno del rettangolo. E' interessante notare che nella sezione della notazione, viene indicato espressamente che il "rettangolino" che viene apposto sulla Lifeline rappresenta l'attivazione di un metodo (chiamata nella specifica ExecutionSpecification). In particolare la specifica afferma che: "To depict method activations we apply a thin grey or white rectangle that covers the Lifeline line." Tale

precisazione lega ovviamente questi diagrammi strettamente al paradigma object-oriented.

State invariant

In alcuni casi, per poter ritenere valida un'interazione, è necessario specificare che alcuni vincoli devono essere soddisfatti. Allo scopo UML 2.0 fornisce la possibilità di inserire, in una Lifeline, una **Invariant**. Se il comportamento dell'entità in questione è descritto anche da uno **Statechart** è possibile usare gli stati del diagramma degli stati per definire le invarianti da rispettare; in tal caso si avrà una State Invariant.

Riferimenti ad altri diagrammi

Spesso i diagrammi di sequenza possono assumere una certa complessità, si può quindi rivelare molto utile poter definire tali comportamenti più articolati come composizione di nuclei di interazione più semplici, oppure, se una sequenza di eventi ricorre spesso, potrebbe essere utile definirla una volta e richiamarla dove necessario.

Per questa ragione, oltre alla decomposizione vista in precedenza nella sezione relativa alle Lifeline, UML 2.0 permette di inserire riferimenti ad altri diagrammi e passare loro degli argomenti; ovviamente ha senso sfruttare quest'ultima opzione solo se il diagramma accetta dei parametri sui quali calibrare l'evoluzione del sistema. Questi riferimenti prendono il nome di **InteractionOccurrences** o di **InteractionUse**, mentre i punti di connessione tra i due diagrammi prendono il nome di **Gate**.

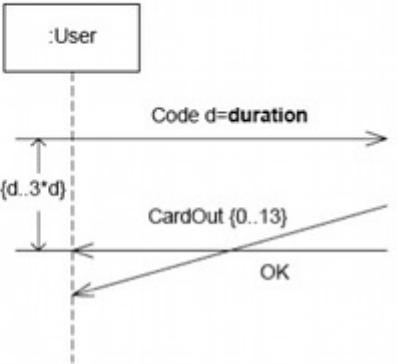
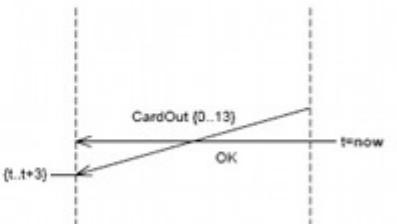
Un Gate rappresenta un punto di interconnessione che mette in relazione un messaggio al di fuori del frammento di interazione, con uno all'interno del frammento.

Vincoli temporali

Per modellare sistemi real-time, o comunque qualsiasi altra tipologia di sistema in cui la temporizzazione è critica, è necessario specificare un istante in cui un messaggio deve essere inviato, oppure quanto tempo deve intercorrere fra un'interazione ed un'altra. Grazie, rispettivamente, a **Time Constraint** e **Duration Constraint** è possibile definire questo genere di vincoli.

La notazione utilizzata per esprimere tali vincoli è riportata nella Figura sottostante.

Figure 2. Notazione per Vincoli temporali (tratto da UML Superstructure pag. 507)

Node Type	Notation	Reference
DurationConstraint Duration Observation	 <pre> sequenceDiagram participant User as :User User->>User: Code d=duration activate User User-->>User: CardOut {0..13} deactivate User activate User User-->>User: OK deactivate User </pre>	See Figure 14.26 on page 513.
TimeConstraint TimeObservation	 <pre> sequenceDiagram activate User User-->>User: CardOut {0..13} deactivate User activate User User-->>User: t=now deactivate User </pre>	See Figure 14.26 on page 513.

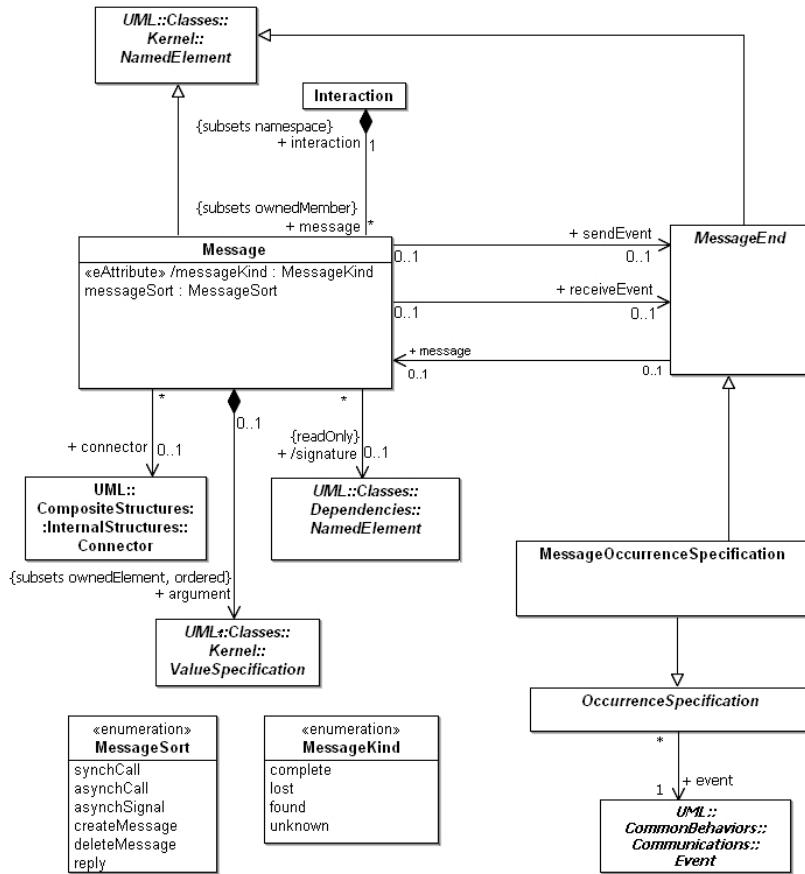
Specifiche dei Messaggi in UML 2.0

Dalla specifica UML SuperStructure (pag. 483) la definizione di messaggio risulta essere la seguente:

"A Message is a NamedElement that defines one specific kind of communication in an Interaction. A communication can be, for example, raising a signal, invoking an Operation, creating or destroying an Instance. The Message specifies not only the kind of communication given by the dispatching ExecutionSpecification, but also the sender and the receiver. A Message associates normally two OccurrenceSpecifications - one sending OccurrenceSpecification and one receiving OccurrenceSpecification".

Il meta-modello del Messaggio è riportato nella figura sottostante.

Figure 1. Meta-modello Messaggio (tratto da UML SuperStructure pag. 462)



Un messaggio rappresenta quindi un'interazione realizzata come comunicazione fra Lifeline, una rappresentante il mittente e l'altra rappresentante il destinatario. Questa interazione può consistere nella creazione o distruzione di un'istanza, nell'invocazione di un'operazione, o nell'emissione di un segnale. Dal meta-modello del messaggio riportato nella figura, è molto interessante notare che un primo modo di classificare i messaggi riguarda gli eventi di invio e ricezione che viene espresso attraverso l'attributo "**messageKind**" che è un enumerativo. In particolare:

- Se sono specificati entrambi allora è un **complete message**; la semantica è rappresentata quindi dall'occorrenza della coppia di eventi **<sendEvent, receiveEvent>**.
- Se il destinatario non è stato specificato allora è un **lost message**; in questo caso è noto solo l'evento di invio del messaggio e quindi è possibile immaginare che il messaggio non sia mai giunto a destinazione.
- Se il mittente non è stato specificato allora è un **found message**; in questo caso è noto solo l'evento di ricezione del messaggio mentre la sorgente rimane sconosciuta. Tale può essere usata per rappresentare messaggi la cui sorgente è fuori dallo scope di interesse, esempi di questo tipo possono essere il rumore oppure attività che non si vuole descrivere in dettaglio.
- Nel caso non sia noto né il destinatario né il mittente allora è un **unknown message**.

Una seconda classificazione deriva invece dal tipo di comunicazione rappresentato dai messaggi. Tale classificazione viene espressa dall'attributo "**messageSort**" di tipo enumerativo come si può vedere nella figura del meta-modello. In particolare:

- **call**: è un complete message associato all'invocazione di un'operazione che può avvenire in modo sincrono (**synchCall** valore di default) o asincrono (**asynchCall**), il cui nome corrisponde a quello dell'operazione. Se la call è sincrona, il chiamante attende il completamento del comportamento richiamato, mentre se la call è asincrona il chiamante continua la sua esecuzione senza aspettare nessun valore di ritorno (cfr SuperStructure pagg. 243-244, elemento "CallAction").
- **signal**: rappresenta un puro scambio d'informazioni generato da un evento asincrono. Il mittente non aspetta nessun valore di ritorno, nel caso però questo sia presente esso verrà semplicemente ignorato e non trasmesso al mittente.

Un invio asincrono implica la prosecuzione del mittente senza la certezza che il messaggio sia effettivamente giunto a destinazione; al contrario il mittente non prosegue finchè il destinatario non è disponibile alla ricezione nel caso sincrono. Se una comunicazione è asincrona può accadere che l'ordine in cui i messaggi arrivano al ricevente differisca da quello con cui sono stati inviati.

Un messaggio può quindi essere inteso sia come un'operazione di call (con conseguente inizio di una attività di esecuzione), sia come l'invio o la ricezione di un segnale. Quando un messaggio rappresenta l'invocazione di una operazione, gli argomenti del messaggio rappresentano gli argomenti della CallOperationAction sulla Lifeline del mittente e gli argomenti dell'occorrenza della CallEvent sulla Lifeline del ricevente. Inoltre in questo caso tipicamente sarà presente un **replay message** dalla Lifeline del chiamato alla Lifeline del chiamante prima che quest'ultimo possa procedere con l'esecuzione. Quando un messaggio, invece, rappresenta un segnale, gli argomenti del messaggio sono gli argomenti della SendAction sulla Lifeline del mittente e sulla Lifeline del ricevente gli argomenti sono disponibili in SignalEvent.

In base all'evento che generano, se non è un CallEvent o un SignalEvent, i messaggi possono essere:

- **create message**: è un complete message che causa la creazione di un'istanza;
- **delete message**: è un complete message che causa la distruzione di un'istanza;
- **reply message**: è un complete message che rappresenta una risposta ad una interazione precedente.

Un messaggio può avere degli argomenti associati ad esso: se abbiamo una call questi devono corrispondere alla signature dell'operazione, altrimenti non ci sono vincoli; il primo degli argomenti è definito **return-value**, cioè cosa ci aspettiamo in risposta.

Come possiamo notare dal meta-modello le classificazioni non si traducono in una specializzazione del concetto di messaggio. Non possiamo parlare di call e signal come tipi, perchè quello che cambia è come viene realizzata la comunicazione.

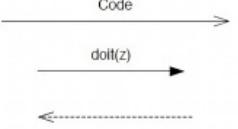
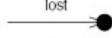
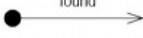
Rappresentazione dei messaggi

Un messaggio è rappresentato con una freccia che parte dal mittente e indica il destinatario come mostrato nella figura sottostante:

- se la punta della freccia è aperta allora il messaggio è asincrono, se invece è piena il messaggio è sincrono;
- se è rappresentato con una freccia tratteggiata con la punta aperta allora è un replay message;
- se è rappresentato con una freccia tratteggiata con la punta aperta che indica l'oggetto creato allora è un create message;

- se la freccia punta ad un cerchio nero allora rappresenta un lost message, se parte da un cerchio nero rappresenta un found message.

Figure 2. Notazione dei Messaggi (tratto da UML SuperStructure pag. 507)

NODE TYPE	NOTATION	REFERENCE
Message		Messages come in different variants depending on what kind of Message they convey. Here we show an asynchronous message, a call and a reply. These are all <i>complete</i> messages.
Lost Message		Lost messages are messages with known sender, but the reception of the message does not happen.
Found Message		Found messages are messages with known receiver, but the sending of the message is not described within the specification.
GeneralOrdering		

Combined Fragment

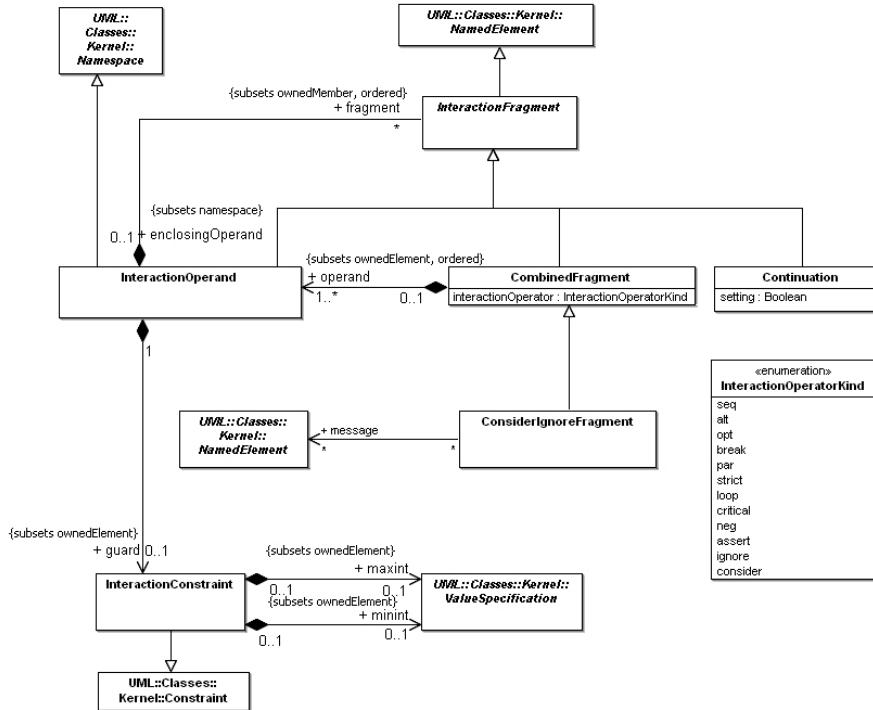
La specifica di UML 2.0 permette di esprimere comportamenti più complessi rispetto a quanto era possibile fare con UML 1.x, ad esempio è ora possibile rappresentare l'esecuzione atomica di una serie di interazioni, oppure che un messaggio deve essere inviato solo in determinate condizioni. A tale scopo UML 2.0 mette a disposizione gli *Interaction Fragment*. Inseriti in un *Combined Fragment*, cioè un contenitore atto a delimitare l'area d'interesse, essi servono per spiegare che una certa catena di eventi, racchiusa in uno o più operandi, si verificherà in base alla semantica dell'operatore associato.

Dalla specifica UML SuperStructure (pag. 467) la definizione di *Combined fragment* risulta essere la seguente:

A combined fragment defines an expression of interaction fragments. A combined fragment is defined by an interaction operator and corresponding interaction operands. Through the use of CombinedFragments the user will be able to describe a number of traces in a compact and concise manner. CombinedFragment is a specialization of InteractionFragment.

Il meta-modello del Combined fragment è riportato nella figura sottostante.

Figure 1. Meta-modello Combined fragment (tratto da UML SuperStructure pag. 464)



Gli operatori a disposizione sono:

- **Loop**: specifica che quello che è racchiuso nell'operando sarà eseguito ciclicamente finchè la guardia, una condizione booleana associata all'operando stesso, sarà verificata;
- **Alternatives (alt)**: indica che sarà eseguito il contenuto di uno solo degli operandi, quello la cui guardia risulta verificata;
- **Optional (opt)**: indica che l'esecuzione del contenuto dell'operando sarà eseguita solo se la guardia è verificata;
- **Break (break)**: ha la stessa semantica di opt, con la differenza che in seguito l'interazione sarà terminata;
- **Critical** : specifica un Atomic Execution Block;
- **Parallel (par)**: specifica che il contenuto del primo operando può essere eseguito in parallelo a quello del secondo;
- **Weak Sequencing (seq)**: specifica che il risultato complessivo può essere una qualsiasi combinazione delle interazioni contenute negli operandi, purchè: (1) l'ordinamento stabilito in ciascun operando sia mantenuto nel complesso; (3) eventi che riguardano gli stessi destinatari devono rispettare anche l'ordine degli operandi, cioè i messaggi del primo operando hanno precedenza su quelli del secondo; (3) eventi che riguardano destinatari differenti non hanno vincoli di precedenza vicendevole;
- **Strict Sequencing (strict)**: indica che il contenuto deve essere eseguito nell'ordine in cui è specificato, anche rispetto agli operandi;
- **Ignore**: indica che alcuni messaggi, importanti ai fini del funzionamento del sistema, non sono stati rappresentati, perchè non utili ai fini della comprensione dell'interazione;
- **Consider**: è complementare ad ignore;
- **Negative (neg)**: racchiude una sequenza di eventi che non deve mai verificarsi;
- **Assertion (assert)**: racchiude quella che è considerata l'unica sequenza di eventi valida. Di solito è associata all'utilizzo di uno State Invariant come rinforzo.

Rappresentazione dei Combined fragment

Un Combined fragment è rappresentato attraverso un rettangolo (vedi figura sottostante). L'operatore è inserito in un pentagono nell'angolo in alto a sinistra. E' possibile inserire più di un operatore nel pentagono. Questo rappresenta una abbreviazione di rappresentazione per Combined fragment innestati. Per esempio se inseriamo "loop strict" nel pentagono ha il medesimo significato di due Combined fragment innestati, il più esterno con operando "loop" e il più interno con operando "strict". Gli operandi di un combined fragment sono mostrati attraverso l'ausilio di linee tratteggiate per dividere il diagramma in diverse regioni corrispondenti agli operandi. Per i dettagli della rappresentazione di ciascun operando si rimanda il lettore alla specifica UML SuperStructure pag.471 e seguenti.

Figure 2. Notazione dei Combined fragment (tratto da UML SuperStructure pag. 506)

NODE TYPE	NOTATION	REFERENCE
Frame		The notation shows a rectangular frame around the diagram with a name in a compartment in the upper left corner.
CombinedFragment		

Esempi di Interazione con i diagrammi di sequenza

Dopo aver analizzato in dettaglio la struttura dei diagrammi di sequenza, ci occupiamo ora di capire quale sia l'effettivo grado di espressività che è possibile raggiungere e quali siano le limitazioni di questi diagrammi. Prima di addentrarci nella nostra indagine è bene fare qualche precisazione.

1. **Entità:** anche se UML è nato nell'ambito object-oriented, i sistemi software non sono più costituiti solamente di oggetti che interagiscono tra loro attraverso il meccanismo dell'invocazione di metodo. Oggigiorno si ha sempre più a che fare con sistemi eterogenei e distribuiti, dove l'interazione non può essere rincondotta semplicemente alla mera invocazione di metodo, non si hanno più solo oggetti, ma anche entità più complesse come i componenti e gli agenti intelligenti. Nel seguito quindi parleremo di entità che interagiscono in modo da astrarre dallo specifico tipo di entità che interagiscono tra loro.
2. **Tool:** per le nostre indagini sarà necessario disporre di un tool di modellazione. Da un lato il tool ci permetterà di provare "sul campo" le potenzialità dei diagrammi di sequenza, dall'altro però l'uso di un tool comporta dei vincoli (o dei gradi di libertà) dovuti al grado di aderenza del tool allo standard UML 2.0. Va anche tenuto in considerazione che spesso i progettisti software raramente si preoccupano di fare riferimento alle specifiche ufficiali pubblicate dall'OMG, al limite fanno riferimento a manuali a loro disposizione o alla guida del tool di modellazione, che non sempre seguono alla lettera le direttive della specifica UML 2.0. Ci preoccuperemo quindi anche di evidenziare gli errori che questa condotta

può indurre. In particolare sono stati usati due differenti tool per testare i diagrammi di sequenza: MagicDraw UML 16.5 e Jude 5.4 entrambi nella versione Community.

La restante parte di questa sezione è organizzata come segue: verrà prima mostrato qualche semplice esempio di interazione elementare realizzata attraverso i diagrammi di sequenza, poi verrà brevemente richiamato l'Interaction Specific Language (ISL) di alto livello che è già stato presentato nel Capitolo X, ed infine verrà presentato come sia possibile mappare questo ISL attraverso i diagrammi di sequenza. Quest'ultima parte sarà divisa in due sottosezioni ciascuna della quali affronta rispettivamente le problematiche delle comunicazioni sincrone e asincrone.

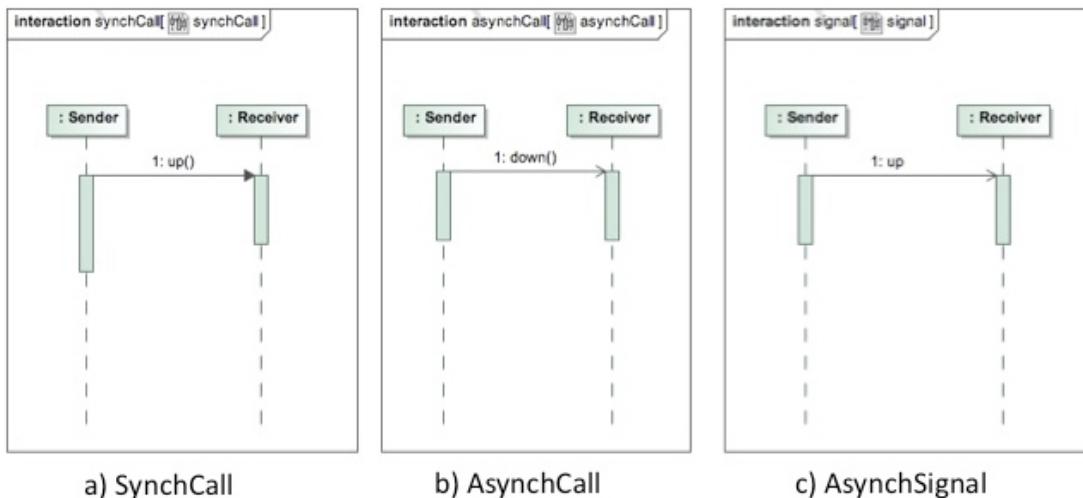
Interazioni di base e diagrammi di sequenza

In questa sezione definiremo la struttura di un semplice sistema campione, al fine di osservare quanti tipi di interazioni elementari si possono esprimere attraverso i diagrammi di sequenza di UML 2.0. A tale fine si è deciso di concentrarsi solamente sui messaggi, tralasciando State Invariant, Combined Fragment e vincoli temporali, in quanto i messaggi rappresentano il blocco di costruzione più elementare per modellare le comunicazioni e vogliamo testare proprio le loro potenzialità.

Come sistema sceglieremo un semplice *Producer/Consumer*: lo chiameremo Sender-Receiver, poiché avremo un'entità Sender che ricoprirà il ruolo di Producer, inviando messaggi ad una seconda entità, Receiver, che li processerà e, in base al contenuto, incrementerà o decrementerà un contatore. Sostanzialmente quindi i messaggi che possono essere scambiati sono di due tipi: *up e down*.

Supponiamo, come primo caso, che Sender debba semplicemente comunicare a Receiver di incrementare (o dualmente di decrementare) il contatore. Le possibili interazioni singole che possiamo rappresentare sono illustrate nella seguente figura.

Figure 1. Interazione Singola



I tre casi rappresentati illustrano rispettivamente:

- (a) Sender richiede l'invocazione di `up()` a Receiver e si assicura che il messaggio sia arrivato prima di proseguire (comunicazione sincrona);
- (b) Sender invia una richiesta per l'invocazione di `down()` a Receiver e poi continua la sua esecuzione (comunicazione asincrona);

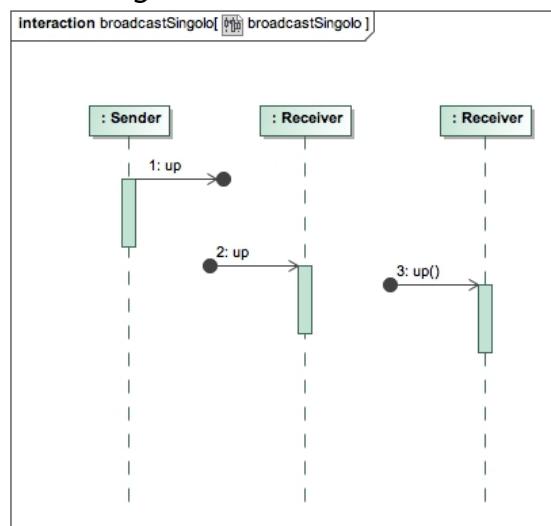
- (c) Sender invia un segnale up a Receiver e poi continua la sua esecuzione.

Per quanto riguarda il caso a) della comunicazione sincrona possiamo supporre che sia il supporto alla comunicazione a fornire una risposta di avvenuta ricezione al Sender, in quanto come fatto notare nella sezione relativa ai messaggi una SynchCall blocca il chiamante sino al completamento del chiamato.

I diagrammi mostrati nella figura sono stati realizzati con MagicDraw, in quanto Jude permette di rappresentare solo messaggi sincroni e asincroni, senza specificare se gli asincroni siano call o signal. Studiando il comportamento del tool però, sembrerebbe che vengano gestiti gli AsynchSignal in quanto non è possibile associare ad un messaggio asincrono un messaggio di replay. Questo mette in luce quanto affermato già in precedenza, cioè che i tool non sempre rispettano le direttive OMG.

Come è facile notare dalla figura precedente gli scenari considerati fanno riferimento a comunicazioni singole di tipo 1-1, in quanto le tipologie di messaggi da noi utilizzati rappresentavano messaggi completi, cioè messaggi in cui sono noti sia l'evento di invio che quello di ricezione, quindi sia mittente che destinatario. Tali tipi di messaggi non consentono la modellazione di comunicazioni 1-n. Se volessimo rappresentare scenari di interazione singola 1-n potremmo sfruttare i messaggi lost e found come mostrato nella figura sottostante.

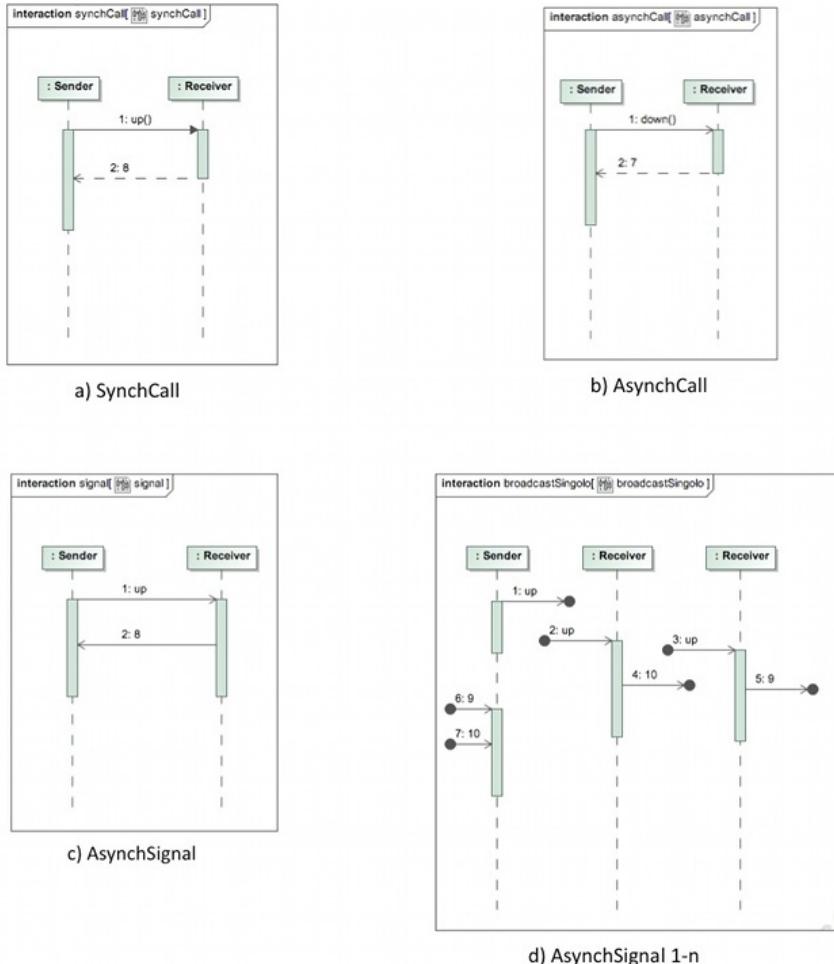
Figure 2. Interazione Singola 1-n



Dalla figura emerge chiaramente che questo tipo di rappresentazione è un po' una forzatura, in quanto vengono utilizzati messaggi che di per sé hanno una specifica semantica in una accezione non sempre immediata da capire specialmente se si pensa alla possibilità di impiegare generatori di codice automatici nei quali non c'è posto per semantiche di comunicazioni ambigue.

Passiamo ora ad analizzare uno scenario un più complesso nel quale il Receiver comunicherà al Sender il valore aggiornato del contatore. Le possibili interazioni che possiamo rappresentare sono illustrate nella seguente figura realizzata con MagicDraw.

Figure 3. Interazione con risposta



Osservando i diagrammi possiamo dire che:

- (a) Sender richiede l'invocazione di `up()` a Receiver e si mette in attesa di una risposta che viene inviata attraverso un `replay message`.
- (b) Sender invia una richiesta per l'invocazione di `down()` a Receiver, prosegue e poi aspetta la risposta inviata attraverso un `replay message`.
- (c) Sender invia un segnale `up` a Receiver, prosegue e poi aspetta una risposta. In questo caso però la risposta sarà inviata attraverso un segnale asincrono in quanto la specifica UML 2.0 non consente di associare a un `AsynchSignal` un `replay message`, cosa che invece sembra possibile fare con MagicDraw.
- (d) abbiamo una situazione particolare che mette in luce la limitatezza di UML nella gestione delle interazioni 1-n: osservando il diagramma possiamo vedere che Sender invia un segnale `up` attraverso l'ausilio di un `lost message`, per poi aspettare le risposte. Le due entità Receiver percepiscono il segnale attraverso dei `found message`, elaborano ed inviano la risposta sfruttando di nuovo dei `lost message`, senza indirizzare specificatamente le risposte a Sender.

Lo scenario d) mette in luce diversi problemi. Il primo riguarda l'identità del mittente della risposta. Se l'identità del Receiver è rilevante ai fini dell'elaborazione di Sender, allora non è possibile sfruttare lo scenario d). Infatti, come si può vedere in figura, le risposte che arrivano al Sender non portano traccia del mittente, quindi il Sender non sarà mai in grado di attribuire con esattezza ad ogni Receiver il rispettivo valore del contatore. Inoltre, come si può vedere in figura, le risposte arrivano in un ordine differente rispetto a quello in cui sono state generate, ma questo lo si riesce a capire solo guardando il

contenuto del messaggio. Questo fatto non è sorprendente, se pensiamo a comunicazioni che avvengono in sistemi distribuiti dove la rete gioca un ruolo strategico, non si ha mai certezza sui tempi di consegna dei messaggi, né sull'effettiva consegna.

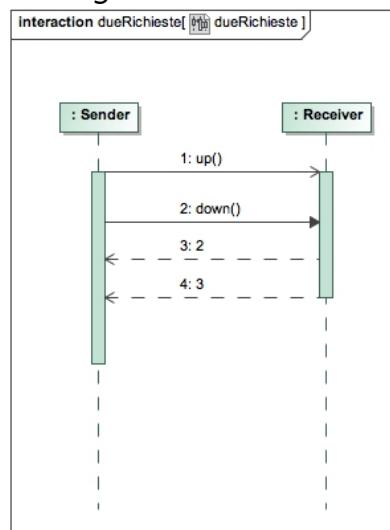
Possiamo affermare con certezza che la maggior parte delle comunicazioni che avvengono nei sistemi software sono ormai di tipo distribuito, questo introduce sia problematiche di sicurezza delle comunicazioni, sia problematiche relative alla gestione della perdita o dei ritardi nella consegna dei messaggi.

Per quanto riguarda le problematiche di sicurezza delle comunicazioni attualmente i diagrammi di sequenza non offrono nessun meccanismo per esprimere se una comunicazione debba avvenire o meno in modo sicuro. Di recente è stato proposto un profilo UML, chiamato [UMLsec](http://www.jurjens.de/jan/umlsec/index.html) (<http://www.jurjens.de/jan/umlsec/index.html>) per permettere di modellare sistemi sicuri attraverso UML. Tale profilo pare non sia stato ancora riconosciuto dall'OMG, in quanto esso non risulta tra l'elenco dei profili riportati nel sito ufficiale dell'organizzazione. I diagrammi di sequenza però non sono stati modificati nella loro struttura, ciò che è stato fatto per specificare che una comunicazione è sicura si riconduce semplicemente allo scambio di più messaggi all'interno di un qualsiasi protocollo, i primi dei quali rappresentano lo scambio di chiavi di sessione con cui cifrare i messaggi successivi, che usano una incomprensibile "etichettatura". Questo ovviamente rende molto disagevole l'utilizzo di tali diagrammi e contribuisce ad aumentarne inutilmente la complessità di lettura.

Per quanto riguarda invece la gestione della perdita o dei ritardi nella consegna dei messaggi, UML 2.0 (nella sua versione originale, lo studio di tutti i possibili profili esula dalla nostra indagine) pare non offrire nessun supporto di base per la rappresentazione di questo evento a meno di non ricorrere all'uso di Combined fragment e Interaction use, che però ovviamente aumentano la complessità di lettura dei diagrammi.

Negli scenari presentati fino ad ora abbiamo considerato condizioni ideali. Consideriamo adesso il caso di un sistema non molto reattivo che costringe Sender ad inviare due richieste successive, una di up e una di down prima di riuscire ad ottenere una risposta, come mostrato nella figura sottostante.

Figure 4. Interazione ambigua



A questo punto è spontaneo chiedersi a quale messaggio si riferisce ciascuna delle risposte. Anche supponendo, come è abitudine, che un reply message si riferisca alla call

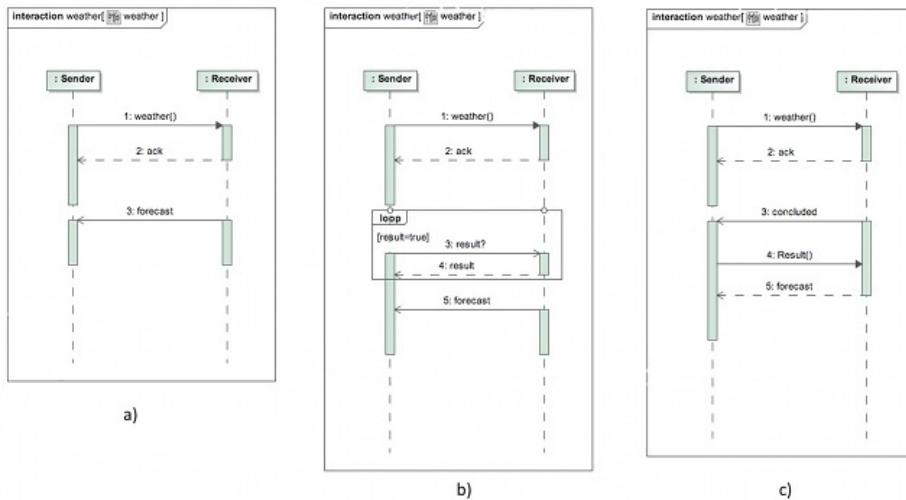
o alla signal che lo precede, la questione non si risolve: infatti, in questo caso, risulterebbe che sono entrambi relativi al messaggio (2), il che è impossibile. Inoltre, nello scenario mostrato in figura, la supposizione che l'ordine delle risposte rispetti l'ordine dei messaggi di richiesta è disattesa. Infatti, guardando il contenuto delle risposte è facile immaginare che il primo dei replay message fa riferimento alla seconda richiesta. Occorre soffermarsi un istante su questo aspetto. Per un umano che deve effettivamente implementare il codice relativo al diagramma in figura è ancora abbastanza semplice riuscire ad interpretare la semantica basandosi sul contenuto del messaggio, tale pratica non sembra comunque corretta in generale in quanto non è sempre possibile associare la risposta alla rispettiva richiesta guardando il contenuto della risposta. Si supponga per esempio che le risposte fossero di tipo booleano, in questo caso non ci sarebbe stato alcun modo di associare risposte con richieste. Se poi pensassimo di utilizzare generatori di codice automatici l'associazione richiesta/risposta basata sul contenuto della risposta diventa assolutamente non praticabile. Un generatore automatico non è in grado di interpretare la semantica della risposta e non potrà mai tradurre in codice effettivo un diagramma ambiguo come quello in figura.

Vediamo quindi da questo semplice esempio una delle limitazioni di UML 2.0: in un diagramma di sequenza basta introdurre un minimo di complessità che diventa impossibile associare in modo univoco un messaggio di replay (o un messaggio che rappresenta una risposta nel caso di AsynchSignal) all'effettivo messaggio di richiesta. Questa ovviamente non è una limitazione da poco, le comunicazioni nei sistemi sono tipicamente basate su protocolli di interazione che possono essere anche molto complessi e non poter associare "nativamente" in modo univoco ciascuna risposta alla sua richiesta causa non pochi problemi nell'interpretazione di un diagramma.

Teniamo a sottolineare che abbiamo usato la parola "nativamente" in quanto è sempre possibile costruire una "sovrastruttura" ad hoc di associazione delle risposte alle richieste, ma questa ovviamente non rappresenta uno standard come invece è la specifica UML, e quindi si avrà la necessità che ogni membro del team di progetto e sviluppo del sistema comprenda (allo stesso modo) la versione modificata dei diagrammi di sequenza. Questo ovviamente va ad impattare sul processo di sviluppo del sistema che diventa più complicato da capire e da replicare al di fuori del team di sviluppo.

Un'altra discussione può scaturire da un ulteriore esempio. Supponiamo di avere a che fare con un Receiver più complesso, in particolare tale Receiver è in grado di gestire messaggi del tipo "Elabora una previsione meteo per i prossimi 5 giorni" (*weather*). L'elaborazione necessaria per gestire la richiesta richiede abbastanza tempo in quanto fare una previsione meteo richiede una simulazione, e quindi la risposta non potrà essere istantanea. Quindi per questo tipo di interazione può essere opportuno pensare di comunicare al mittente la presa in carico del messaggio ed in un secondo tempo fornire il risultato dell'elaborazione. Quest'ultimo caso è un esempio di **protocollo parzialmente specificato**, in quanto, una volta dato conferma, la seconda parte dell'interazione può svolgersi in qualsiasi modo vogliamo come riportato nella figura sottostante:

Figure 5. Protocollo parzialmente specificato



In particolare:

- a) Receiver può inviare un messaggio con il risultato a Sender quando è pronto;
- b) Sender può interrogare a **polling** Receiver fino a che non ottiene il risultato, in questo caso abbiamo dovuto fare ricorso ad un Combined fragment in quanto non era possibile specificare un loop di attesa utilizzando solo gli strumenti di base dei diagrammi di sequenza;
- c) Receiver può comunicare la fine dell'elaborazione, e poi aspettare che Sender lo interroghi per fornire il risultato.

Per rappresentare in un diagramma la prima parte di questa interazione ricadiamo comunque nella stessa rappresentazione utilizzata per il Sender-Receiver mostrato negli esempi precedenti, mentre per la parte relativa alla risposta abbiamo un problema: come è possibile capire cosa significa il diagramma? Come possiamo distinguere una risposta da una conferma se sono rappresentate nel medesimo modo? Di nuovo ricadiamo nel problema che l'unico strumento che abbiamo a disposizione per distinguere le due differenti situazioni è l'interpretazione semantica del diagramma, quindi non abbiamo nemmeno la certezza che due umani possano interpretare esattamente allo stesso modo tale diagramma.

Ci rendiamo conto da queste osservazioni che la modellazione di questo scenario di comunicazione non ha una soluzione banale. UML 2.0 offre la possibilità di modellare l'interazione con un approccio **low-level**, utile per poterlo applicare in ogni situazione, ma fragile per gli aspetti presentati e quindi difficilmente applicabile per automatizzare l'implementazione. Come già sottolineato in precedenza, un essere umano è in grado di osservare un diagramma ed agire per risolvere problemi semantici, un generatore di codice invece ha bisogno di concetti e pattern definiti e sufficientemente stabili da essere utilizzati con qualsiasi tecnologia, stile architetturale e paradigma di programmazione. Sostanzialmente si deve avere la possibilità di esprimere l'interazione ad un più alto livello di astrazione, in modo da semplificare il lavoro del progettista.

Interaction Specific Language

Nella sezione precedente abbiamo messo in luce diversi problemi legati in generale alla capacità espressiva dei diagrammi di sequenza di UML 2.0 ed abbiamo sottolineato come sia necessario mettere a disposizione di analisti e progettisti un "vocabolario" sia per esprimere l'interazione ad un livello di astrazione che semplifichi il lavoro degli

sviluppatori, sia per fornire una semantica non ambigua per l'interazione. Di fatto la frase (ma anche la sua rappresentazione in un diagramma di sequenza) "A manda un messaggio a B" lascia aperte molte domande tra cui in particolare:

- che tipo di comunicazione hanno?
- quali sono le aspettative che hanno A e B?

Supponiamo anche di specificare che il messaggio sia sincrono, supposizione che risponde alla prima domanda ma non ancora alla seconda. Non sappiamo infatti se A necessiti una risposta da B, se la risposta è un *ack* oppure il risultato. Non sappiamo se A rimanga in attesa, oppure continua la sua elaborazione e la risposta arriverà quando sarà pronta. Come vediamo questo semplice esempio rientra nella categoria dei protocolli parzialmente specificati introdotti alla fine della sezione precedente.

Riportiamo quindi brevemente la descrizione dell'Interaction Specific Language presentato in precedenza:

- **Dispatch**: messaggio che viene inoltrato da un soggetto (dispatcher) ad uno specifico destinatario, nella speranza che questi lo riceva e lo serva. Nel caso la comunicazione non si concluda con successo il mittente può ricevere un *fault message*, offrendogli la possibilità di reagire come crede.
- **Signal**: messaggio di natura asincrona, emesso da un soggetto (sorgente) e affidato al mezzo di trasmissione. In conseguenza uno o più soggetti, se interessati, sono in grado di percepirla senza che il comportamento di uno influenzi gli altri. La comunicazione realizzata è, almeno concettualmente, broadcast.
- **Event**: messaggio che non rappresenta un intento comunicativo, ma un accadimento. Può essere sollevato da un soggetto (sorgente) senza nessuna aspettativa: esso infatti lascia una traccia nell'ambiente che può essere percepita dagli altri soggetti, oppure no.
- **Request**: rappresenta una richiesta che un soggetto (richiedente) fa ad uno o più soggetti, aspettandosi di ricevere almeno una ed al più tante risposte quante sono le richieste inviate. La relativa Response viene recapitata al richiedente sperando che egli la acquisisca.
- **Invitation**: messaggio che modella una domanda posta a determinati soggetti (uno o più), aspettandosi di ricevere una conferma (*Acknowledgment*) per ciascun destinatario che accetti l'invito; almeno una.
- **Response e Acknowledgement**: possono essere usati solo in relazione a Request e Invitation, altrimenti non hanno senso.

Comunicazione Sincrona

In questa sezione viene mostrato come e se è possibile mappare l'ISL presentato in precedenza attraverso i diagrammi di sequenza UML 2.0 utilizzando in particolare messaggi di tipo SynchCall. Vedremo negli esempi che con questo tipo di messaggi avremo sempre il problema di capire chi fornisce la risposta che "sblocca" il mittente del messaggio.

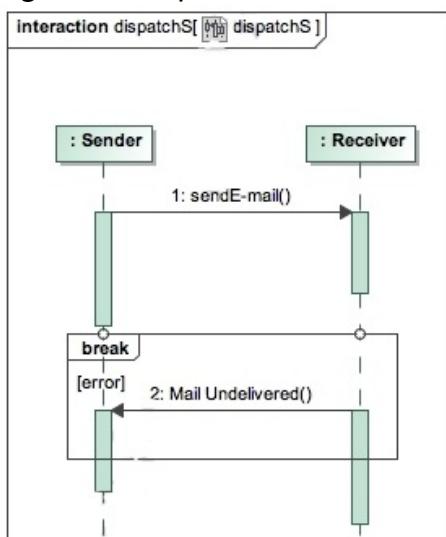
Al fine di illustrare il mapping useremo esempi tratti dalle comunicazioni con cui abbiamo a che fare nella vita quotidiana.

Dispatch

Un classico esempio di comunicazione di tipo "dispatch" è rappresentato dall'invio di un messaggio di posta elettronica. Di fatto, se pensiamo solamente al protocollo utilizzato per l'invio di una e-mail (prescindendo quindi dalle aspettative di chi invia il messaggio) questo presuppone semplicemente che dopo l'invio della e-mail al mittente venga segnalato solamente se ci sono stati dei problemi. Il diagramma di interazione di questo scenario è riportato nella figura sottostante.

Da notare che come accennato nella sezione degli esempi di interazione si suppone che sia il supporto alla comunicazione a dare un "acknowledgement" al Sender in modo che il suo flusso di esecuzione possa riprendere. Questa supposizione ovviamente cambia la semantica del diagramma di sequenza in quanto tale messaggio non è indicato nel diagramma per evitare di aumentare la complessità del diagramma stesso introducendo anche il supporto alla comunicazione. Ovviamente questa nuova entità dovrebbe essere introdotta nel diagramma nel caso in cui il relativo codice dovesse essere realizzato da uno sviluppatore non consci della "convenzione" adottata.

Figure 1. Dispatch Sincrono



Per realizzare questa semplice comunicazione abbiamo dovuto fare ricorso al Combined fragment "break" che permette di esprimere un comportamento opzionale e successivamente fa terminare l'interazione, in alternativa si poteva utilizzare il Combined fragment "opt". Per indicare il messaggio di fault "Mail Undelivered" è stato usata una SynchCall anche se si ritiene non essere la soluzione migliore, in quanto di nuovo il supporto alla comunicazione dovrebbe fornire l'"acknowledgement" per sbloccare l'esecuzione del Receiver.

Da notare che l'utilizzo di messaggi SynchCall per rappresentare una comunicazione di tipo dispatch non sembra essere la soluzione migliore a meno di far emergere l'entità "supporto alla comunicazione" e far apparire i messaggi di presa in consegna dell'e-mail. Però di nuovo qui nascono diversi problemi:

- "è davvero quello che si vuole rappresentare?". Questa domanda sorge spontanea in quanto non sempre si vuole mostrare l'entità "supporto alla comunicazione", nella fase di progettazione potrebbe essere molto utile indicarla, ma se siamo nella fase di analisi, nella quale vogliamo solo indicare interazioni ad alto livello, un

diagramma come quello riportato in figura è ambiguo e può portare ad una comprensione errata da parte del progettista.

- "come rappresento la presa in carico?". Questa è un'altra questione interessante in quanto dalla specifica UML si evince che una SynchCall necessita di un replay message per sbloccare l'attesa del chiamante, ma se il supporto invia un replay message, come potremmo poi rappresentare una eventuale risposta a questa e-mail? di nuovo con un replay message? ma poi come lo associo la risposta alla relativa domanda?

Di nuovo le limitazioni dei diagrammi di sequenza di UML non consentono una completa e non ambigua rappresentazione di questo semplice tipo di comunicazione.

Signal ed Event

Un esempio di comunicazione di tipo "signal" è rappresentato dall'invio di un sms. Se pensiamo al meccanismo sottostante e prescindiamo dalle aspettative del mittente l'invio di un sms rappresenta l'invio di un segnale ad uno o più destinatari. Allo stesso modo un esempio di comunicazione di tipo "event" potrebbe essere rappresentato da un allarme che suona e produce una perturbazione dell'ambiente circostante che può essere percepita o meno dalle altre entità nell'ambiente.

Ovviamente questi due tipi di comunicazione che per loro natura sono asincroni non sono facilmente mappabili attraverso l'utilizzo di call sincrone. Inoltre, come abbiamo già visto in precedenza l'invio di segnali broadcast comporta l'utilizzo di "lost" e "found" message, rendendo così estremamente ambigui i diagrammi che ne risultano. Non ci dilungheremo oltre nello spiegare tutte le problematiche relative alla realizzazione di signal ed event attraverso SynchCall in quanto tali problematiche sono già state trattate ampiamente nei paragrafi precedenti.

Request/Response

Questo tipo di comunicazione è quello che trova maggiori esempi nelle interazioni che abbiamo nel quotidiano. L'esempio più immediato che viene in mente è quello di una telefonata che viene mappato molto facilmente da una SynchCall essendo la telefonata per sua natura sincrona. Questo però è poco rappresentativo, in quanto la maggior parte delle comunicazioni request/response non avviene in questo modo. Riprendiamo invece l'esempio della posta elettronica (tralasciando la questione del messaggio di fault già trattata nella sezione relativa a "dispatch"), stavolta supponendo che nelle aspettative del mittente ci sia il desiderio di ricevere una e-mail di risposta.

Il diagramma di sequenza che viene più immediato in questo caso è quello di una SynchCall con il relativo messaggio di replay. Tale diagramma però appare subito il meno realistico, in quanto non è detto che la risposta sia immediata e quindi non possiamo pensare di lasciare il Sender in attesa di una risposta che potrebbe anche non arrivare mai per problemi di varia natura (rete, ricevente non legge la posta, ecc.). Dovremo quindi pensare di utilizzare il "trucco" che il supporto invii al Sender un messaggio di presa in carico del messaggio e poi al Sender in un qualche modo arriverà la risposta alla e-mail spedita.

Anche in questo caso vediamo emergere diverse problematiche già ampiamente illustrate. La prima riguarda la rappresentazione sia del messaggio della presa in carico sia della risposta, come si fa a distinguerle se vengono rappresentate dalla stessa freccia? La seconda problematica riguarda l'associazione tra richiesta e risposta nel caso che quest'ultima non sia istantanea e il Sender abbia avviato altre comunicazioni. L'ultima

problematica è relativa ai protocolli parzialmente specificati, categoria di cui questo esempio fa parte. Abbiamo infatti il problema di stabilire come la risposta possa essere consegnata al Sender e come abbiano già ampiamente discusso nel caso della richiesta "weather" vi sono più soluzioni ciascuna delle quali presenta diverse problematiche.

Le cose si complicano ulteriormente se pensiamo a request/response di tipo 1-n. In questo caso non possiamo usare lo schema con messaggi "lost" e "found" in quanto il Sender avrà sicuramente necessità di conoscere l'identità di chi ha mandato la response a meno di non introdurre nel contenuto del messaggio una indicazione dell'identità del mittente. Il caso si complica ulteriormente se dovessimo rappresentare più Sender nello stesso diagramma, in questo caso avremo la necessità di indicare anche l'identità del destinatario della response oltre a quella del mittente. Questo ovviamente da un lato ci consentirebbe di risolvere il problema, dall'altro però ci porta ad una soluzione che "mescola" l'atto di comunicazione con l'informazione veicolata, cose che per quanto possibile occorrerebbe tenere distinte. E' bene ricordare che aggiungere informazioni relative all'atto fisico della comunicazione all'interno del messaggio causa diversi problemi in caso di generazione automatica del codice. Il contenuto del messaggio non dovrebbe nemmeno essere processato dal generatore automatico, se così non fosse ci troveremo ad avere generatori automatici creati ad hoc per ogni team di sviluppo (supponendo che all'interno dello stesso team si adottino sempre le stesse convenzioni) e questo ridurrebbe drasticamente le potenzialità del loro riuso. L'unica soluzione al problema dell'invio broadcast risulterebbe quella rappresentare sul diagramma tante interazioni singole quanti sono i diversi destinatari e attendere le relative risposte.

Invitation/Acknowledgement

Un esempio di questo tipo di comunicazione è dato dalle fasi iniziali dell'avvio di una chat nelle quali una entità invita altre entità a prendere parte alla discussione e si mette in attesa di ricevere le risposte agli inviti. Anche se la semantica della comunicazione è differente rispetto a quella che abbiamo nel caso di request/response, all'atto pratico della realizzazione di questa comunicazione attraverso i diagrammi di sequenza è facile vedere che si incontrano le stesse problematiche di "implementazione" attraverso SynchCall.

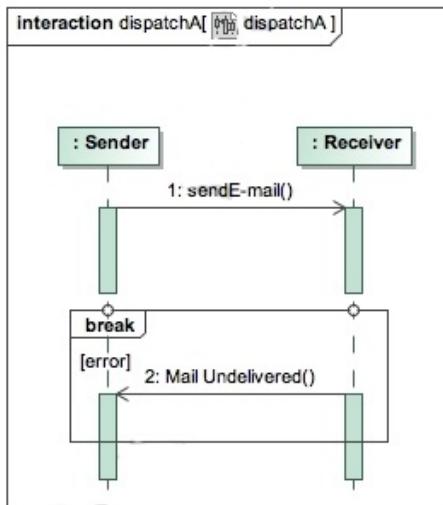
Comunicazione Asincrona

In questa sezione viene mostrato come e se è possibile mappare l'ISL presentato in precedenza attraverso i diagrammi di sequenza UML 2.0 utilizzando in particolare messaggi di tipo AsynchCall e AsynchSignal. Verranno ripresi gli esempi illustrati nella sezione relativa ai messaggi SynchCall.

Dispatch

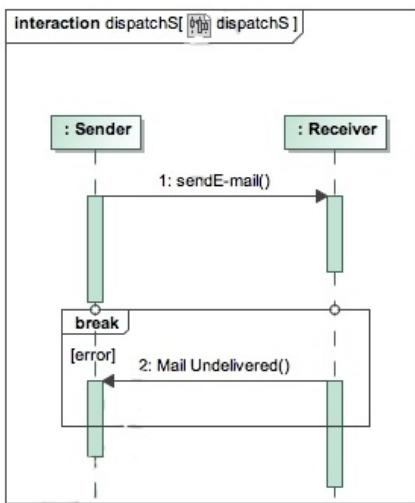
Come esempio di comunicazione dispatch è stato introdotto lo scenario di invio di un messaggio di posta elettronica. Nella figura sottostante è riportato il diagramma di sequenza rappresentante tale scenario realizzato attraverso messaggi AsynchCall.

Figure 1. Dispatch Asincrono



Come per il caso sincrono, per rappresentare l'eventuale invio di un messaggio di fault è stato utilizzato il Combined fragment "break". Nella figura sottostante, invece, è rappresentato il diagramma di sequenza relativo all'invio della posta elettronica realizzato mediante AsynchSignal. Come si può vedere tale diagramma non differisce molto dal precedente. Inoltre l'utilizzo di call asincrone e signal ha il vantaggio di non dover delegare al supporto di comunicazione il compito di inviare un messaggio al Sender per sbloccare il suo flusso di esecuzione.

Figure 2. Dispatch con Segnale



Confrontando i diagrammi con quello realizzato nel caso di interazione sincrona possiamo dire che sia AsynchCall che AsynchSignal permettono di modellare in modo abbastanza soddisfacente questo tipo di comunicazione.

Signal ed Event

Questi due tipi di comunicazione (rispettivamente invio sms e allarme) da una parte vengono mappati abbastanza bene da AsynchCall ed in modo soddisfacente da AsynchSignal, in quanto entrambi i tipi di messaggi supportano al meglio la natura asincrona delle comunicazioni. Dall'altra parte però essi sono affetti dalle stesse problematiche delle SynchCall relative all'invio broadcast che non è supportato da UML 2.0. Anche in questo caso bisognerà fare ricorso all'utilizzo di messaggi "lost" e "found", con la conseguente abiguità che ne deriva.

Request/Response

Riprendiamo l'esempio dell'invio della posta elettronica proposto nella sezione relativa ai messaggi sincroni. Utilizzando messaggi di tipo AsynchCall il Sender non ha più la necessità di attendere un messaggio di presa in carico per continuare la sua esecuzione e questo ovviamente offre un vantaggio rispetto all'uso di una call sincrona. Il problema che rimane però anche nel caso asincrono è quello di stabilire l'associazione tra la richiesta effettuata e il replay messaggio che giungerà dal Receiver, che come abbiamo detto più volte rappresenta una delle limitazioni più grandi dei diagrammi di sequenza.

Le cose non migliorano adottando AsynchSignal poiché in questo caso non si possono nemmeno adottare messaggi di replay per mandare la risposta al Sender in quanto la specifica UML 2.0 non lo consente (anche se ciò è consentito da MagicDraw). Quindi in questo caso sarà necessario inviare la risposta attraverso un altro AsynchSignal che diventa ancora più difficile da correlare alla richiesta iniziale.

Comunque c'è da notare che l'utilizzo dei due tipi di messaggi asincroni non risolve il problema del protocollo parzialmente specificato in quanto il Sender ha diverse possibilità di azione per il recupero della risposta. Inoltre, per quanto riguarda l'invio broadcast si hanno le stesse problematiche già discusse per il caso sincrono.

Invitation/Acknowledgement

Come fatto notare nel caso sincrono, la realizzazione di questo tipo di comunicazione non si discosta molto dalla realizzazione della comunicazione request/response. Possiamo quindi affermare che la modellazione di questa comunicazione attraverso messaggi e segnali asincroni risolve solo una delle problematiche illustrate nel caso sincrono (la presa in consegna del supporto) e lascia aperte tutte le altre problematiche di rappresentazione.

Conclusioni

In questo capitolo sono stati ampiamente illustrati i diagrammi di sequenza così come riportato nel documento di specifica e sono stati fatti diversi esempi che hanno messo in evidenza la limitatezza del potere espressivo di tali diagrammi. Le ragioni di tali limitatezze sono da ricercare nell'origine totalmente orientata agli oggetti e ai sistemi concentrati di UML. Di fatto gli attuali diagrammi di sequenza non sono in grado di modellare scenari di interazione distribuita tra entità eterogenee (non solo oggetti).

I meccanismi di modellazione dell'interazione sono troppo a basso livello e non consentono di rappresentare in modo consono scenari di interazione 1-n. Questo ha permesso di mostrare che non è possibile realizzare in modo soddisfacente un mapping tra l'Interaction Specific Language di alto livello necessario per rendere chiara e non ambigua la semantica della comunicazione e i diagrammi di sequenza.

I (design) pattern

Nonostante il ruolo strategico che l'Architettura riveste nel denotare e conferire ad un sistema le sue caratteristiche essenziali, l'architettura di un sistema software spesso non costituisce il risultato di una fase esplicita di analisi e di progettazione, ma scaturisce in modo implicito dai meccanismi di combinazione relativi al linguaggio o allo stile di programmazione adottato per la codifica.

Ovviamente questo modo di procedere è molto pericoloso, soprattutto quando le metafore del linguaggio di codifica non sono pertinenti al problema. Si pensi ad esempio al caso di un programmatore che debba affrontare la costruzione di una applicazione di rete disponendo del solo linguaggio C.

Per colmare il divario tra il livello di progettazione e il livello di codifica si è diffuso l'utilizzo di un potente tipo di "strumento logico": il **pattern**.

L'ideatore riconosciuto del concetto di **pattern** è l'architetto, **Christopher Alexander**[Alexander79] che, di fronte alla necessità di dover risolvere ricorrentemente tipologie di problemi simili, ha proposto l'idea di catturare la "**struttura di una comprovata soluzione efficace**" attraverso una descrizione, detta appunto pattern. Un pattern per Alexander stabilisce una relazione tra un **contesto**, un insieme di **forze** che caratterizza quel contesto e una **configurazione** che permette di trovare un equilibrio tra quelle forze, anche se contrastanti tra loro. Una configurazione specifica i partecipanti al pattern, le loro responsabilità e le loro interazioni; il pattern spiega come e perchè le forze sono state bilanciate in quel modo.

Come già accaduto per il concetto di oggetto, anche per il concetto di **pattern** sono state proposte molte visioni e definizioni. La definizione di [POSA1] pag. 8 dice: **A pattern for software architecture describes a particular recurring design problem that arises in specific design contexts, and presents a well proven generic scheme for its solution. The solution scheme is specified by describing its constituent components, their responsibilities, and the way they collaborate.**

Un pattern fornisce uno schema di soluzione (non una soluzione immediatamente disponibile) per un problema ricorrente adottata in uno o più sistemi reali e individuato attraverso un'opera di astrazione. Martin Fowler definisce (in **Analysis Patterns**, pg.8) un pattern come **an idea that has been useful in one practical context and will probably be useful in others**. Egli ribadisce anche (pg.12) che **patterns are suggestions, not prescriptions**.

Attraverso una accurata forma descrittiva (si veda [POSA5], cap 3, pag. 91-116 e cap. 12, pag. 325-346) i pattern ([POSA5] pag. 8-12):

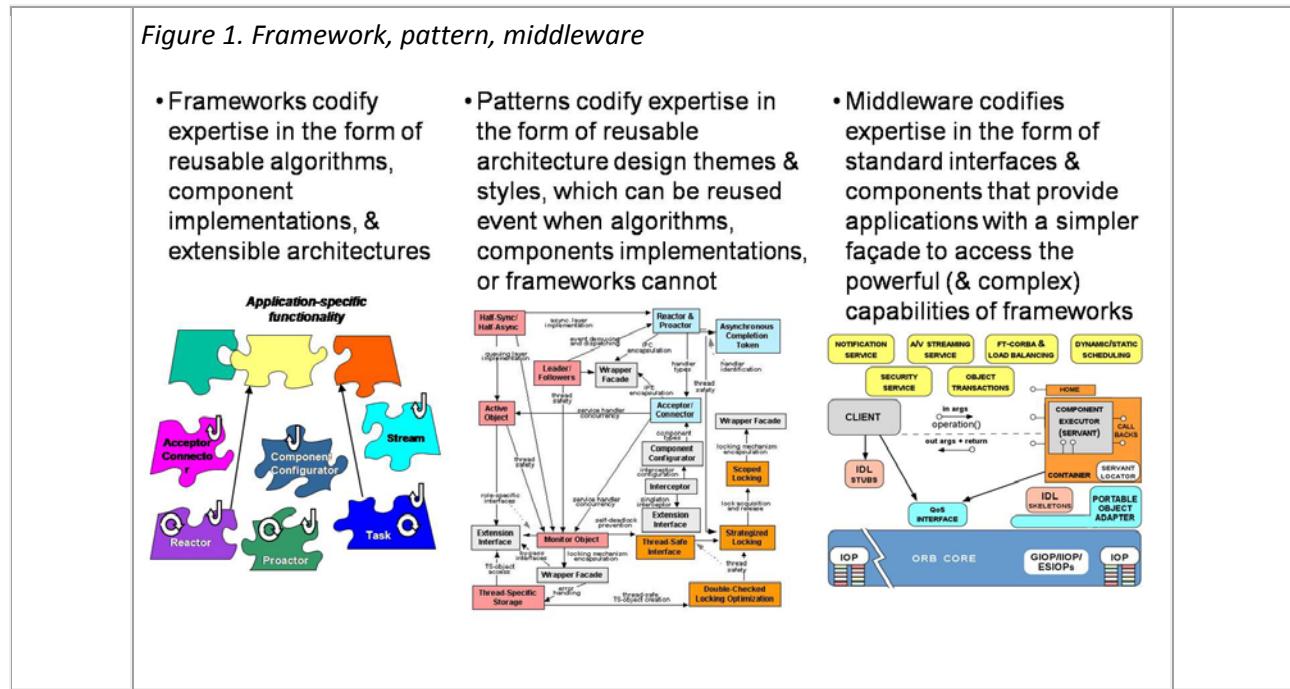
- documentano "best practices" di progettazione ritenute efficaci dagli esperti;
- catturano l'esperienza di progettazione in una forma che risulta indipendente dai dettagli di un singolo progetto, di un paradigma implementativo e da un linguaggio di programmazione;
- identificano e specificano astrazioni a un livello superiore a quello di singoli oggetti;
- forniscono un vocabolario comune e un'visione condivisa di concetti di progettazione;
- costituiscono un modo per documentare le architetture software;
- supportano la costruzione di software caratterizzato da ben precise proprietà.

Va anche sottolineato che ([POSA5] pag. 13-23):

- un pattern non è l'espressione di un qualsiasi progetto di successo;
- un pattern non è un "blueprint" da cui ricavare immediatamente codice;
- un pattern non può essere applicato in modo meccanico; la sua integrazione in un sistema specifico richiede l'impegno della mente umana;

- talvolta i pattern sono accompagnati da diagrammi UML; questa pratica è tuttavia fuorviante, in quanto induce a pensare a un pattern come uno schema immediatamente utilizzabile;
- un pattern non promuove nuove idee;
- l'uso di un pattern non garantisce di per sé la produzione di software corretto, estendibile ed efficiente;
- i pattern non sono componenti software;
- l'uso dei pattern non è in disaccordo con l'idea di *refactoring*, ma può avvenire in modo complementare ad essa;
- i pattern da soli non sono in grado di risolvere la crisi del software.

Inizialmente introdotto nei processi software come *pattern di progettazione* con specifico riferimento al paradigma ad oggetti, l'idea di pattern si è molto sviluppata, fino a giungere a proporre il concetto di *pattern language* (si veda [Pattern languages](#)). I *pattern (languages)* fungono oggi da veri e propri depositi di conoscenza sulle tecniche più consolidate per la soluzione di problemi non riconducibili a livello di algoritmi. La figura che segue (di Douglas C. Schmidt) può infine essere utile per chiarire la differenza tra pattern, framework e middleware.



Il ruolo dei pattern

L'uso dei pattern risulta particolarmente utile nelle fasi di analisi del problema e di impostazione dell'architettura (logica) della soluzione.

Un pattern ha in primo luogo un ruolo "didattico" in quanto spiega la logica con cui un problema può essere risolto, usando uno "stile letterario" di descrizione. In secondo luogo, un pattern ha un ruolo "sociale" in quanto promuove una tecnologia collaborativa in cui la comunità ricostruisce e codifica un'esperienza condivisa, gettando le premesse per nuove azioni di successo. Infine un pattern ha un ruolo "generativo" in quanto è anche in grado di promuovere una soluzione efficace ad un problema.

Il punto più importante nella descrizione di un pattern però non è la parte relativa alla "soluzione" ma è l'insieme delle forze e il contesto che determinano il problema e il modo in cui il pattern affronta queste forze. Nella visione dell'architetto Alexander [Alexander79] le **forze** sono i vincoli, i requisiti, le proprietà desiderate che "rendono un problema" il problema di cui il pattern si occupa; il **contesto** è la situazione che conduce al problema e che delinea i casi in cui il pattern può essere applicato con successo.

Un pattern affronta dunque uno specifico **problema** e pone in relazione un **contesto**, un sistema di **forze** che si manifestano ripetutamente in quel contesto e una specifica configurazione software (**soluzione**) che permette a queste forze di bilanciarsi. (definizione di Jim Coplien, 2004). Un pattern definisce *un possibile spazio di soluzioni* e non un singolo punto; da esso si possono ricavare molte realizzazioni concrete diverse, ciascuna delle quali deve tenere conto dei requisiti della specifica applicazione. Nell'ambito di un processo di produzione del software un pattern può essere interpretato come un *micro-processo* che produce specifici artefatti.

Un **pattern architetturale** esprime l'organizzazione strutturale e comportamentale di un sistema software. Definisce un insieme predefinito di sottosistemi, specificando le loro responsabilità, le loro relazioni strutturali e le regole (protocolli) d'uso. Un **pattern di progettazione** fornisce invece uno schema utile a raffinare i sottosistemi di un sistema software, specificando una struttura di interazione tra componenti che risolve un problema generale di progettazione nell'ambito di un particolare contesto. Come tale esso rappresenta uno schema per raffinare la struttura e il comportamento dei sottosistemi di un sistema software. E' importante osservare che l'applicazione di un design pattern non incide sulla struttura fondamentale del sistema. Si dice invece **idioma** un pattern di basso livello, solitamente legato a un linguaggio di programmazione, che descrive come realizzare un dato schema di relazioni tra componenti usando le features del linguaggio

La categorizzazione di un pattern come architetturale o di progetto può dipendere dal punto di vista; ad esempio il pattern *interpreter* [GHJV95] viene proposto come un design pattern, ma potrebbe anche essere visto come l'elemento centrale di un sistema software.

Vari autori hanno osservato che alcuni problemi possono essere meglio affrontati attraverso **compound patterns**, con l'avvertenza che la composizione di un insieme di pattern risolve l'insieme di forze specifico del problema e non una combinazione dei problemi relativi a ciascun pattern componente. Ad esempio il compound pattern *Bureaucracy* di Dirk Riehle include i pattern [GHJV95] *Composite, Mediator, Chain of Responsibility* e *Observer*. E' stato anche osservato che un problema di tipo generale può essere risolto da una **famiglia di pattern**; ciascun elemento della famiglia definisce il problema in modo più specifico o risolve le forze in modo diverso. Si può anche dare il caso che un particolare dominio applicativo possa essere associato ad un **sistema di pattern** in cui ciascun pattern risolve uno specifico problema in quel dominio, facendo anche riferimento ad altri pattern del sistema per la sua implementazione.

In generale quindi, i pattern sono utili ciascuno individualmente; essi però possono risultare ancora più utili se combinati in un insieme coerente costruito tenendo conto che *ogni pattern dipende dai pattern più semplici che contiene e dai pattern più ampi entro cui è contenuto* ([Alexander79], pag 312).

La lunga riflessione sul concetto di pattern che si è sviluppata a partire dagli anni 90 ha quindi indotto a porre maggiore attenzione alle **relazioni** tra i diversi pattern e ha

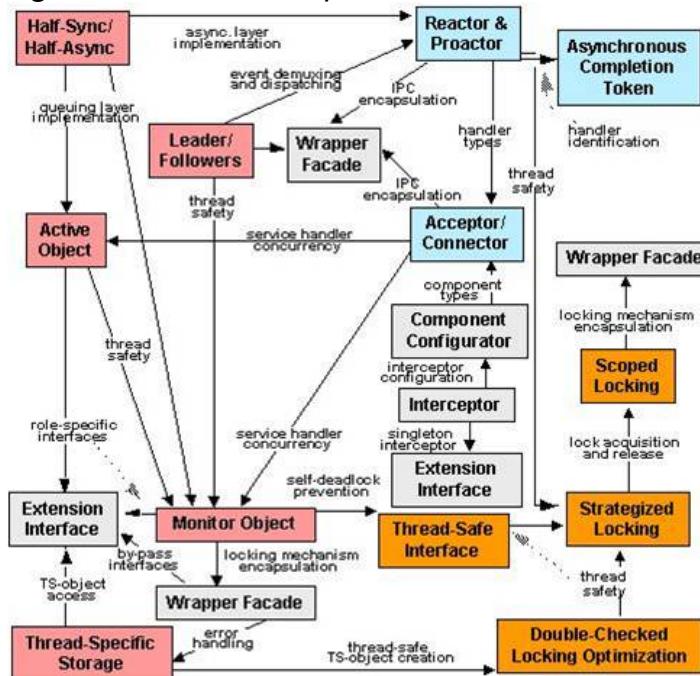
promosso l'idea che via sia maggior valore negli insiemi di pattern piuttosto che nei singoli pattern presi isolatamente. Non si tratta solo di definire collezioni o cataloghi di pattern secondo un qualche schema logico di aggregazione; l'obiettivo è individuare famiglie e/o **insiemi di pattern che nel loro complesso possano caratterizzare uno stile architettonico** (si veda [Insiemi di pattern](#)).

Insiemi di pattern

Per vari autori, una collezione di pattern che si appoggiano l'uno sull'altro permette di costruire sistemi che risultano superiori alla somma delle loro parti. Ad esempio Michael Beedle, in *Reengineering the Application Development Process* lega l'uso di insiemi di pattern all'idea di sistemi capaci di mostrare comportamenti emergenti ed auto-organizzanti: *"spontaneously recurring patterns of dense local interaction between entities, resulting in dynamic, self-organizing systems that are adaptive, open, and capable of multi-scale effects. In other words, pattern languages provide a dynamic process for the orderly resolution of problems within their domain which indirectly leads to the resolution of a much broader problem. The patterns and rules in a pattern language combine to form an architectural style. In this manner, pattern languages guide system analysts, architects, designers, and implementors to produce workable systems that solve common organizational and development problems at all levels of scale and diversity."*

In generale va detto che non si tratta solo di definire collezioni o cataloghi di pattern secondo un qualche schema logico di aggregazione; l'obiettivo è individuare famiglie e/o **insiemi di pattern che nel loro complesso possano caratterizzare uno stile architettonico**. La figura che segue mostra le relazioni tra i pattern presentati in [POSA2]

Figure 1. Il sistema di pattern POSA2



L'organizzazione di pattern in insiemi coerenti può avvenire in vari modi. In particolare ([POSA5] cap 9, pag. 248-250):

- Si definisce un catalogo di pattern secondo un qualche criterio; in questo modo si evidenziano insiemi detti **pattern collections** ([POSA5] cap 8, pag. 209-242). I **Design Patterns** descritti nel testo della GoF [GFJV95] sono suddivisi in tre categorie: pattern *creazionali, strutturali e comportamentali*.
- Un insieme di pattern può essere esplorato cercando famiglie di pattern che possano risolvere specifici problemi; in questo modo si evidenziano insiemi detti **pattern complements** ([POSA5] cap 5, pag. 135-164).
- Individuare i pattern per il progetto e la realizzazione dei quali può essere utile fare ricorso ad altri pattern; in questo modo si evidenziano insiemi detti **pattern compounds** ([POSA5] cap 6, pag. 165-182).
- Trattare lo sviluppo di un sistema software in modo narrativo, evidenziando la sequanza logica di applicazione di pattern, e descrivendo i criteri con cui i pattern sono stati progressivamente selezionati, guidando di fatto il progetto e lo sviluppo del sistema; in questo modo si evidenziano insiemi detti **pattern stories** ([POSA5] cap 7, pag. 183-190);
- Rimuovere la parte narrativa specifica da una pattern story, ponendo in evidenza un insieme "riusabile" detto **pattern sequence** ([POSA5] cap 7, pag. 191-208).

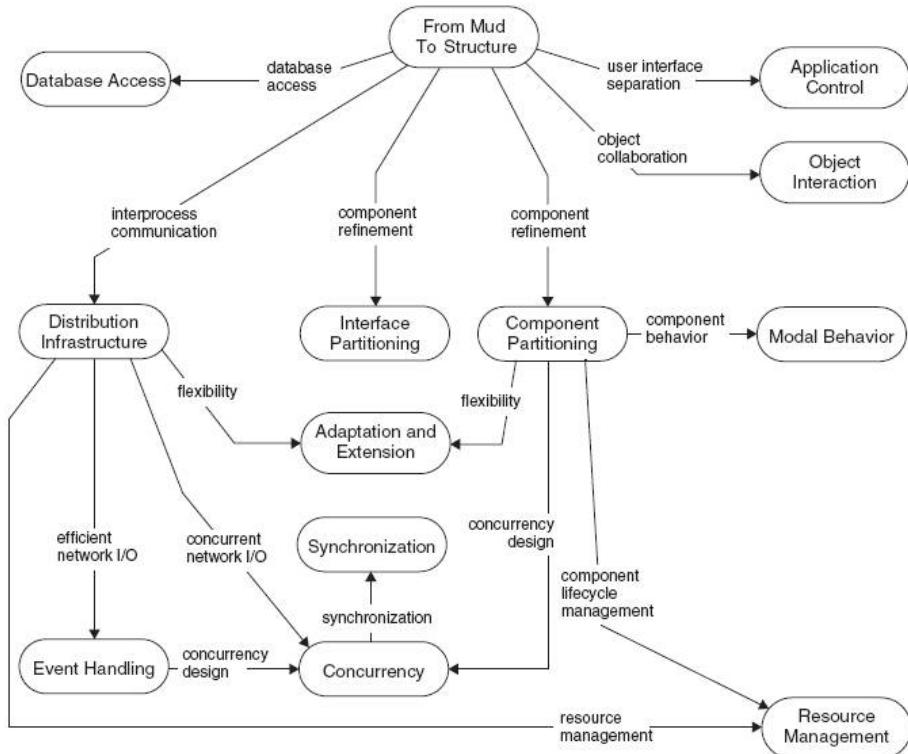
Un insieme di pattern forma un sistema quando è completo da due punti di vista:

- **Morfologicamente**: i pattern si collegano l'uno all'altro a formare una struttura priva di discontinuità.
- **Funzionalmente**: ogni nuova forza (introdotta da un pattern) è risolta dal sistema di pattern.

Nel seguito faremo riferimento in particolare al sistema di pattern definito in [POSA4] che amplia quanto definito in [POSA2] includendo 114 pattern suddivisi in 13 aree di problemi:

1. from mud to structure
2. distribution infrastructure
3. event demultiplexing and dispatching
4. interface partitioning
5. component partitioning
6. application control
7. concurrency
8. synchronization
9. object interaction
10. adaptation and extension
11. modal behaviour
12. resource management
13. database access

A pag. 40 di [POSA4] viene presentato un diagramma che illustra le relazioni logiche tra le



aree:

Figure 2. Aree POSA4

A pag.45 gli autori presentano una figura che illustra in modo sintetico il modo con cui viene descritta una problem area:

Figure 3. Struttura della descrizione di un'area POSA4

Name of the problem area →

Event Demultiplexing and Dispatching

Distributed computing is ultimately event-driven, even when middleware platforms offer applications with a more sophisticated communication model, such as request/response operations or asynchronous messaging. There are a number of challenges that differentiate event-driven software from software with a 'self-directed' flow of control [PLoPD1]:

- *Asynchronous arrival of events.* Behavior in event-driven software is triggered largely by external or internal events that can arrive asynchronously. Most events must be handled promptly, even if the application is under heavy workload, or while it is executing long-duration services. If not, response time will suffer, and hardware devices with real-time constraints will fail or corrupt data.
- *Simultaneous arrival of multiple events.* Event-driven software typically receives events from multiple independent event sources, such as I/O ports, sensors [...].
- [More event handling challenges]

The four event-handling patterns in our pattern language for distributed computing help to fill this gap. They provide efficient, extensible, and reusable solutions to key event demultiplexing and dispatching problems in event-driven software:

- The REACTOR pattern (259) [POSA2] allows event-driven software to demultiplex and dispatch service requests that are delivered to an application from one or more clients.
- The PROACTOR pattern (262) [POSA2] allows event-driven software to demultiplex and dispatch service requests triggered by the completion of asynchronous operations efficiently. [...].
- The ACCEPTOR-CONNECTOR pattern (265) [POSA2] [...].
- [More pattern abstracts]

A diagram that outlines the integration of the patterns 'from' this problem area into the pattern language

A pattern that uses a pattern 'from' this problem area

A pattern that 'belongs' to this problem area

A concrete uses-relationship between two patterns labeled with its purpose

A pattern that is used by a pattern 'from' this problem area. Those patterns that are external to the book are set in *italics*.

A discussion on the patterns from this problem area →

The following diagram illustrates how Reactor and Proactor integrate into our pattern language

The Reactor and Proactor patterns define event demultiplexing and dispatching infrastructures that can be used by event-driven applications to detect, demultiplex, dispatch, and process events they receive from the network. Although both patterns resolve essentially the same problem in a similar context, and also use similar patterns to implement their solutions, the concrete event-handling infrastructures they suggest are distinct, due to the orthogonal nature of each pattern is exposed. [More discussion].

Notiamo che gli autori non utilizzano, intenzionalmente, alcuna notazione di quelle oggi in uso (in particolare quella connessa a UML), dicendo che (pag. 47): **"One reason we do not use popular modeling notations is to avoid the fallacy of 'false concreteness,' which often leads readers to think that what is in the diagram is the only way to implement a pattern. Instead, we provide a solution sketch, not a concrete specification with classes, objects, and relationships between them. Our notation therefore mixes many aspects: role specification, role organization, role collaboration, pseudo-interfaces, and pseudo-code, whatever appears appropriate to show a particular pattern."**

Pattern languages

Una generica idea di insieme o network di pattern risulta però insoddisfacente se si vuole individuare un concetto che possa supportare l'applicazione dei pattern per creare in modo olistico sistemi software per specifici domini e/o per affrontare in modo sistematico aspetti rilevanti e strategici quali comunicazione, interazione, distribuzione, fault-tolerance, life-time management di oggetti/componenti, etc.

Il concetto attualmente preso a riferimento per una "holistic, systematic, and constructive support for developing software with patterns" ([POSA5] pag. 244) è quello di **pattern language** definito in [POSA5] pag. 260 come "*a network of tightly interwoven patterns that defines a process for systematically resolving a set of related and interdependent software development problems*".

Il concetto di linguaggio va qui inteso in senso lato. Con riferimento alla usuale interpretazione di linguaggio, i singoli pattern possono essere visti come gli elementi terminali (**words**) mentre le regole grammaticali sono più difficili da individuare; in [POSA5] (pag 281) si lega l'idea di grammatica alle pattern sequences ammissibili e si discutono (pag 282-284) anche i possibili formalismi per esprimere tali sequenze, tra cui la "railroad notation" usata per definire la sintassi del linguaggio Pascal [JW75].

Il punto importante da comprendere è che (POSA5] pag. 260) così come un singolo pattern è molto più che "a solution that arises within a specific context", "*a pattern language is much more than a network of tightly interwoven patterns*". L'idea di fondo consiste nel privilegiare la visione di insieme (il tutto) rispetto ai casi particolari (le parti).

Le principali forze di un pattern language sono riconducibili non solo alle unioni delle forze dei pattern costituenti, ma anche il complesso dei pattern può risolvere le forze globali che si presentano nel dominio. In questa visione un pattern language è il mezzo per definire uno o più **stili architetturali** (si veda [Lo stile architettonico](#)).

Un pattern language ha in sè l'idea che ogni pattern del linguaggio debba essere applicato all'interno di una specifica sequenza. Per raggiungere questo fine, il contesto di un pattern che fa parte di un pattern language descrive i pattern che devono essere introdotti prima che esso possa essere realizzato con successo. Inoltre ogni pattern descrive un **resulting context** che descrive quali altri pattern possono essere usati per la sua implementazione o per procedere nel processo di raffinamento dei problemi. L'insieme dei contesti di tutti i pattern di un pattern language forma quella che [POSA5] chiama (pag. 278) la **topologia** dei pattern, una struttura parzialmente ordinata con un unico punto di ingresso.

Idealmente si potrebbe pensare di definire due pattern languages diversi per uno stesso dominio adottando uno stesso insieme di pattern ma modificando la topologia. I due linguaggi sarebbero caratterizzati da due diversi stili architetturali proprio in virtù del diverso modo di organizzare la topologia dei contesti.

Pertanto, un pattern language non definisce solo un insieme di pattern logicamente correlati, ma fornisce anche linee guida su come affrontare i problemi di un dominio e su quali soluzioni concrete avanzare in termini di progetto o implementazione in quanto fornisce indicazioni sull'ordine logico (in termini di **pattern sequences**) con cui affrontare i problemi, le possibili vie per una costruzione efficace del tutto. Il "tutto" potenzialmente generabile da un buon pattern language dovrebbe possedere una qualità spesso denotata come **QWAN**: *quality without a name* riconducibile ad una universale idea di bellezza ed ordine. (si veda anche [Gelernter88] *Machine Beauty: Elegance and the Heart of Technology*).

Un software design pattern language è sempre definito con specifico riferimento a un dato dominio, con l'obiettivo di rappresentare e trasferire conoscenze e metodi che sono stati ripetutamente applicati con successo per creare e supportare architetture software sostenibili in quel dominio e non per rappresentare idee che **potrebbero** risultare utili.

Ad esempio il pattern language descritto in [POSA4] riguarda i sistemi distribuiti e mira a catturare le diverse situazioni del software distribuito senza compromettere fattori di qualità come scalabilità e performance. Il testo presenta 114 pattern suddivisi in 13 aree di problemi: ***from mud to structure, distribution infrastructure, event demultiplexing and dispatching, interface partitioning, component partitioning, application control, concurrency, synchronization, object interaction, adpatation and extension, modal behaviour, resource management, database access.***

Altri pattern language sono stati proposti ([POSA5] pag. 370) ad esempio per progettare i componenti di un server [VSW02], realizzare comunicazioni remote in ambiente distribuito [VKZ04], realizzare applicazioni sicure [SFHBS06] o web services [SNL05], per gestire processi di sviluppo con metodi agili o model-driven quali aspect oriented software development [Schmidt06] e model driven software development [SV05], fino a interessare campi diversi dall'Informatin Technology (IT) come nel caso del [pedagogical patterns project](#).

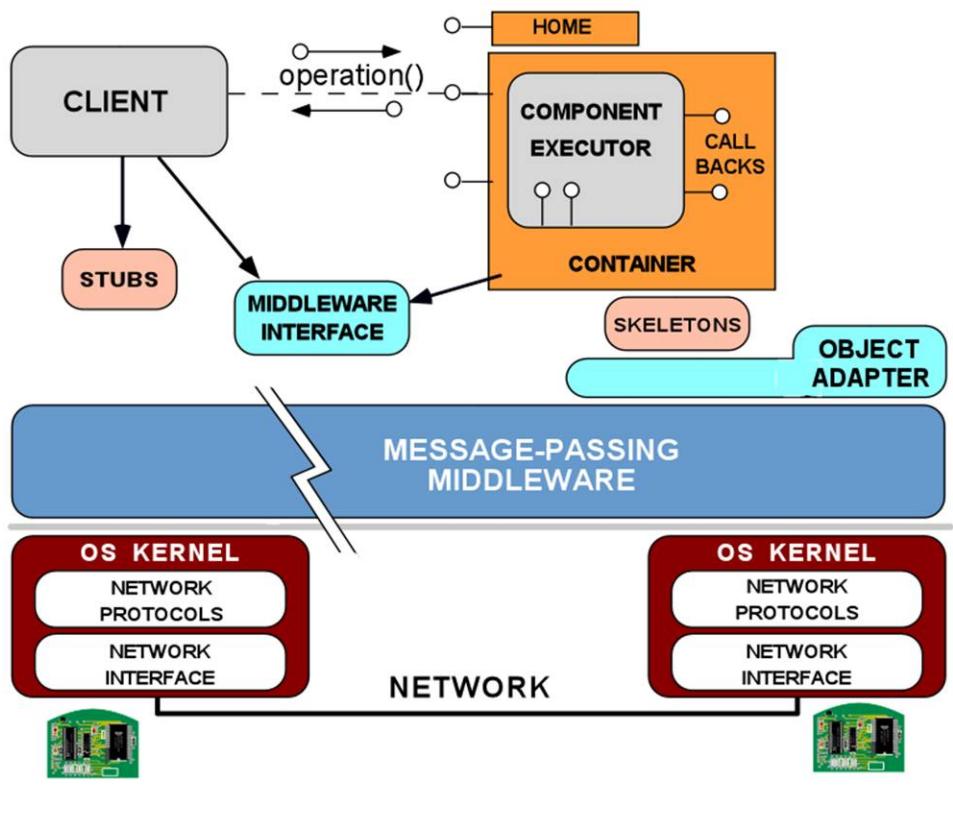
Molti importanti domini di interesse IT sono ancora scoperti o non pienamente affrontati, tra cui ([POSA5] pag. 377-280) le service oriented architectures (SOA), distributed real time and embedded systems, mobile and pervasive systems, collaborative working e collective intelligence. Il progetto di Booch per la creazione di un [Handbook of Software Architecture](#) ha oggi in elenco circa 2000 pattern.

Piattaforme operative

Ogni prodotto software è intrinsecamente legato a una piattaforma di esecuzione, capace di comprendere ed eseguire il linguaggio in cui il software è espresso. La piattaforma di esecuzione è quasi sempre un ulteriore prodotto software (ad esempio J2EE [J2EE], .NET [.NET] che si basa a sua volta su una gerarchia di prodotti software (JVM [JVM], CLR [CLR], il sistema operativo, etc.) fino a giungere alla piattaforma fisica costituita dall'hardware dell'elaboratore.

In questa organizzazione a livelli, ogni livello supporta una collezione diversa di astrazioni la cui granularità tende a crescere dal basso verso l'alto mentre l'applicabilità (**scope**) tende a decrescere. Ad esempio, la piattaforma J2EE fornisce astrazioni per lo sviluppo di business applications ed ha scope più ristretto rispetto a Java J2SE [J2SE] su cui si poggia.

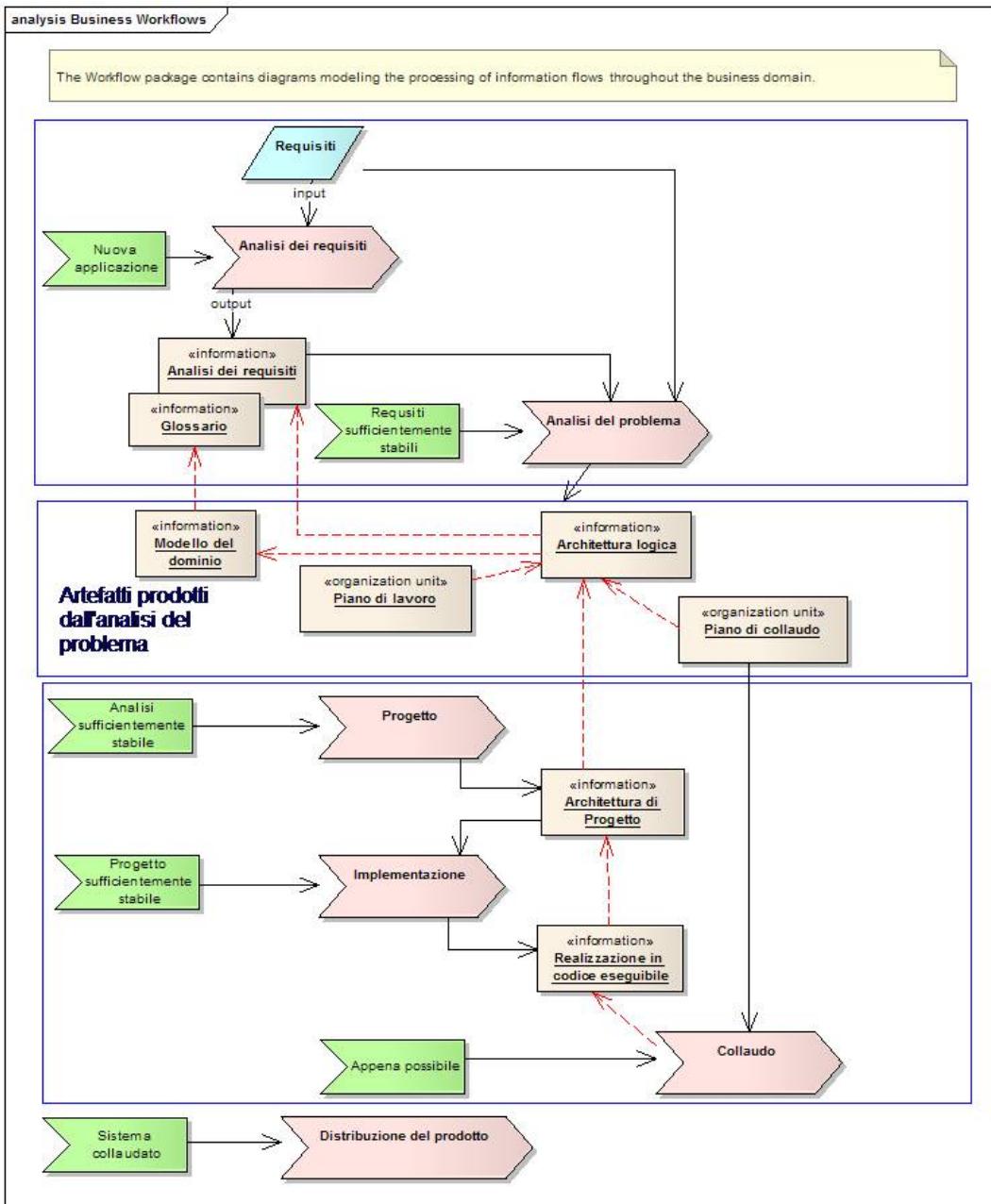
Figure 1. Gerarchia di



Le piattaforme di alto livello permettono di affrontare la risoluzione di un problema fissando un punto di partenza più vicino al dominio applicativo, riducendo l'**abstraction gap** con il livello di astrazione del problema. Un'astrazione supportata da una piattaforma può essere vista come una soluzione preconfezionata a una famiglia di sotto-problemi comuni e correlati e può essere usata assieme ad altre astrazioni per risolvere in modo completo o parziale altri problemi nel dominio.

Un percorso di riferimento

In un processo di produzione che mira a costruire un prodotto software a partire dai requisiti di un committente, gli elementi logici che svolgono un ruolo strategico possono essere riassunti attraverso l'activity diagram UML che segue.



La richiesta di sviluppo di una nuova applicazione innesca la fase di analisi dei requisiti che, in stretta cooperazione con il committente, produce alcuni artefatti essenziali tra cui un **modello dei casi d'uso (use cases)** con relativi **scenari**, un **glossario** dei termini usati nel documento che descrive i requisiti e un primo **modello del dominio** indipendente dalle tecnologie realizzative e, per quanto possibile, anche dalla specifica applicazione in esame.

Non appena l'analisi dei requisiti raggiunge uno stadio sufficientemente stabile, si passa ad enucleare i principali problemi posti dall'applicazione. Dall'analisi dei problemi si cerca di capire se siamo in un contesto già affrontato da altri, iniziando anche a tenere conto di **requisiti non funzionali** (si veda [Requisiti non funzionali](#)). Un possibile punto di riferimento è la **pattern community**, che potrebbe avere già proposto uno o più **pattern languages** (si veda [Pattern languages](#)) relativi al problema in esame. Nel caso esista un pattern language affine, un'attenta lettura del pattern language aiuterà a comprendere meglio il problema ed ad approfondire l'analisi grazie alle esplicita enunciazione del **contesto** e delle **forze** in gioco. Nel caso in cui non si trovi alcun pattern language di

riferimento, la fase di analisi del problema dovrà richiedere più tempo, attenzione e risorse umane.

La fase di analisi produce vari artefatti che trovano la loro sintesi nella definizione di una **architettura logica** che può essere vista come la specifica di un insieme di **vincoli** (strutturali, comportamentali e di interazione) imposti dal problema e/o dal dominio applicativo; tali vincoli devono essere individuati tenendo conto dei requisiti, dell'ambiente e del contesto socio-economico in cui si inserisce l'applicazione, evitando di introdurre vincoli dettati da specifiche tecnologie realizzative (a meno che queste non siano esplicitamente menzionate nei requisiti).

Esempio: una ditta viene incaricata del progetto del ponte sullo stretto di Messina, Dopo avere analizzato le caratteristiche del territorio e del tratto di mare da attraversare, la ditta potrebbe concludere che l'architettura (logica) del ponte dovrà essere a una sola campata; ciò in quanto si è valutato troppo rischioso (per via delle correnti, venti, etc) o troppo antieconomico (per la profondità, etc). pensare di installare pilastri di supporto in mezzo al mare.

Gli artefatti (modelli) scaturiti dall'analisi, e in particolare l'architettura logica, costituiscono gli elementi fondamentali per una accurata **analisi dei rischi** e una prima **pianificazione del lavoro**, nel quadro di un processo di tipo iterativo. Da essi si anche ricavare un primo insieme di **piani di collaudo** utili ad agevolare la integrazione di sistema e a supportare **processi test-driven**.

Dalla fase di analisi del problema il responsabile del progetto ricava informazioni essenziali per organizzare i tempi e i modi delle fasi successive del processo di sviluppo, tenendo conto delle risorse disponibili. Se l'architettura logica del sistema mostra che questo può essere articolato in più sottosistemi, allora il project leader può attivare, anche in parallelo, diverse linee di progettazione e sviluppo. Poichè è essenziale che venga garantita la possibilità di integrazione dei diversi sottosistemi nell'unico sistema finale, il project leader cura con la massima attenzione le modalità di interazione tra i sottosistemi e definisce precisi vincoli (se possibile in forma di **piani di collaudo**) per ciascun sottosistema.

La fase di progettazione (di un sottosistema) mira non solo a individuare e descrivere una soluzione al problema (**what**), ma soprattutto a descrivere i motivi (**why**) che determinano la soluzione proposta.

La fase di progettazione dovrebbe procedere dal generale al particolare, sviluppando per prime le parti più critiche individuate dall'analisi. I pattern languages possono essere usati per impostare un processo di sviluppo incrementale il cui risultato consiste in una specifica **pattern sequence** [POSA5]. La progettazione non procede necessariamente a cascata, ma può dare luogo a refactoring (di progetto) e può produrre anche retroazioni sulla fasi precedenti. Al progressivo sviluppo dell'architettura del sistema corrisponde un progressivo raffinamento dei piani di collaudo.

La progettazione può essere notevolmente influenzata (ed anche agevolata) dall'uso di piattaforme operative sviluppate da terze parti in forma di **framework** o di **product-line architectures** (si veda [Sviluppo basato su framework](#)). Idealmente però la fase di progettazione mira a dare risposte a problemi posti dall'applicazione; essa dovrebbe avvenire quindi in modo il più possibile platform-independent, così che la realizzazione possa poi avvenire su piattaforme operative diverse.

Il codice di implementazione dovrebbe essere il più possibile allineato con la soluzione logica scaturita dalla fase di progetto. Questo obiettivo viene facilmente raggiunto nel caso in cui sia possibile **generare il codice** in modo automatico dai modelli che rappresentano il progetto. Nel caso in cui non sia possibile adottare approcci generativi, la **tracciabilità** (si veda Requisiti) deve essere mantenuta in modo esplicito dagli sviluppatori. La produzione del codice avviene comunque in un contesto vincolato dai piani di collaudo definiti nelle fasi precedenti.

Impostazione generale

In sintesi si può affermare che non si tratta solo di costruire un prodotto

... ma di porre attenzione al **processo** di produzione ...

... costruendo **artefatti** (prevalentemente in forma di modelli UML) ...

... con particolare riguardo agli artefatti di analisi e di progetto ...

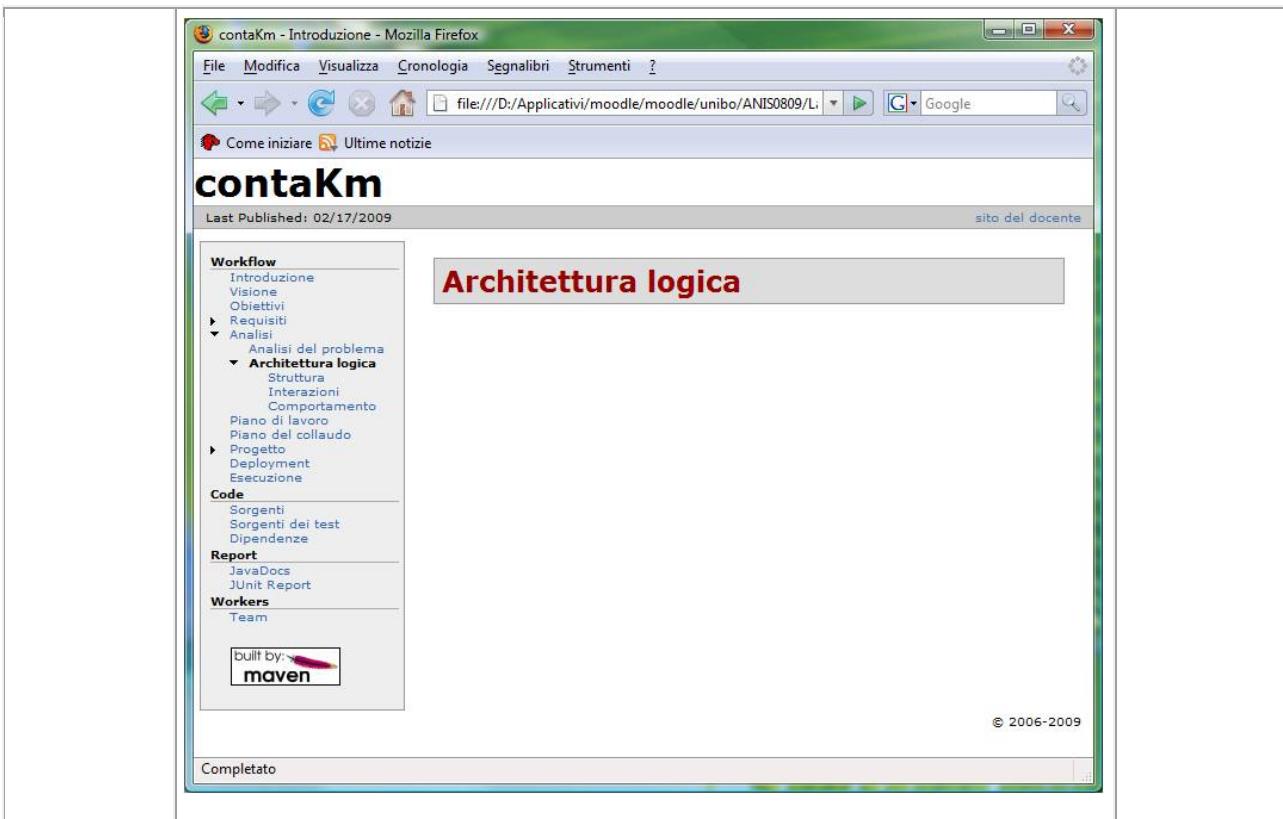
... al fine di promuovere e sostenere un'organizzazione cooperativa ed evolutiva del lavoro.

L'analisi dei requisiti, l'analisi del problema e la progettazione richiedono infatti forti **interazioni tra persone**, in particolare colleghi di lavoro (analisti, progettisti, etc) ed utenti finali.

L'introduzione di adeguati modelli espressi in **UML** può risultare fondamentale per il miglioramento della comunicazione e della comprensione reciproca; i modelli di progetto possono inoltre già costituire la specifica finale per il sistema, nel caso in cui sia possibile la generazione automatica di (parte del) codice su specifiche piattaforme.

Per gestire al meglio il processo di produzione può essere conveniente introdurre uno schema di riferimento con il supporto di un **sito dell'applicazione** in cui mantenere una rappresentazione aggiornata degli artefatti. Scopo del sito è promuovere lo scambio di conoscenza tra i diversi attori (analisti, progetti, realizzatori, etc) coinvolti nel processo. Un esempio di come si può presentare un sito di processo realizzato con il framework **Maven** è ripostato nella figura che segue:

Figure 1. Sito di progetto



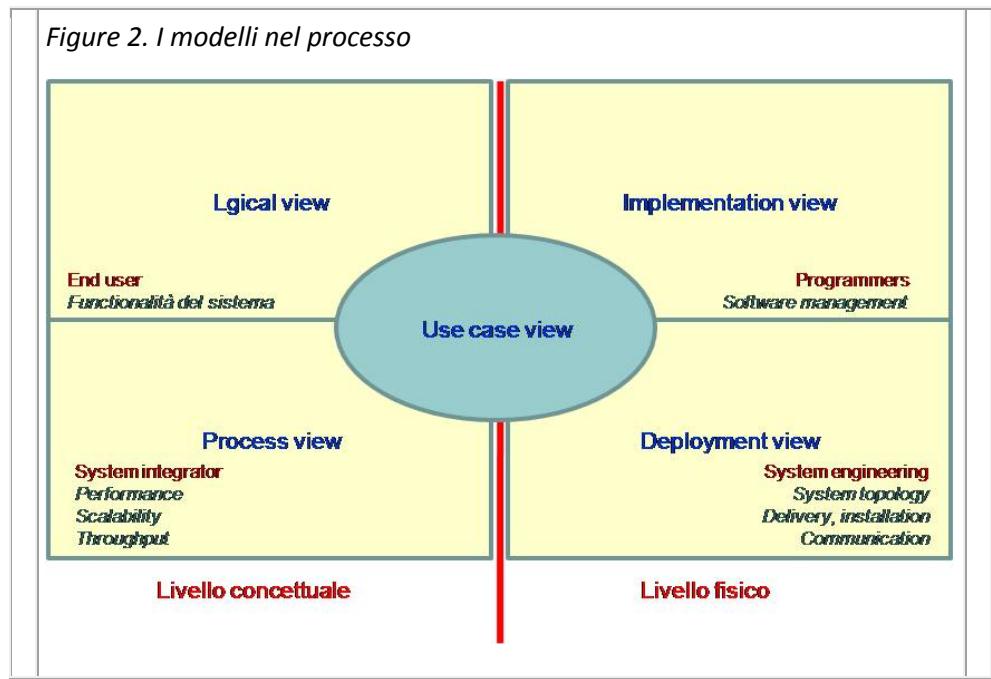
Questo sito può essere generato da un archetipo che fornisce uno schema di riferimento iniziale per la descrizione e l'organizzazione degli artefatti nel contesto di un processo di tipo iterativo (si veda Processi iterativi a spirale).

Come criterio di comportamento generale si può partire dalla massima "**non c'è codice senza progetto, non c'è progetto senza analisi del problema, non c'è problema senza requisiti**" e procedere secondo il seguente workflow:

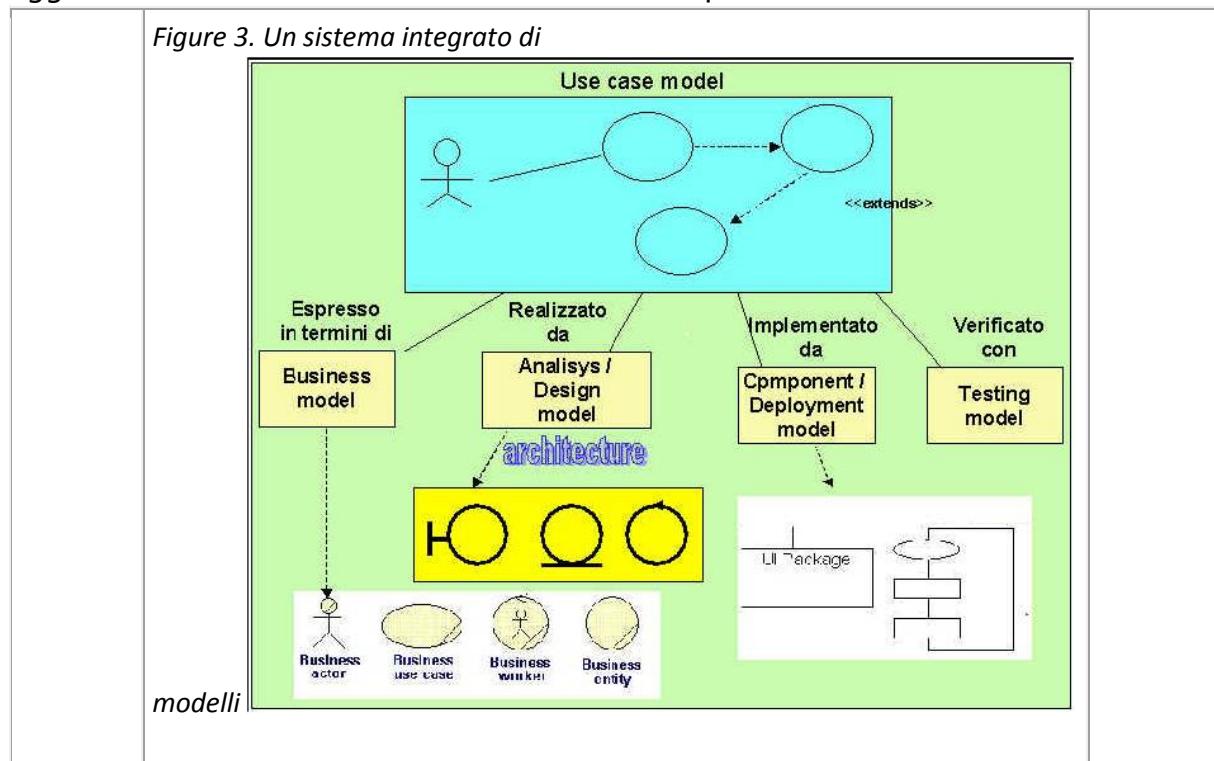
1. Si imposta la fasi di analisi del problema con l'obiettivo di definire un'architettura logica che tenga conto dei requisiti e dei vincoli imposti dal problema e dal dominio applicativo.
2. Facendo riferimento all'architettura logica si definisce un primo piano di collaudo.
3. Facendo riferimento all'architettura logica si individuano la parti più rischiose e/o più costose da realizzare e si pianificano in modo esplicito i tempi di progettazione e sviluppo, sulla base delle risorse umane disponibili e dei tempi di consegna concordati.
4. Si imposta una prima iterazione di progetto con l'intento di costruire un prototipo che soddisfi il piano di collaudo, limitando l'attenzione ai requisiti essenziali e agli scenari primari definiti in tali requisiti.
5. Si mostra il prototipo al committente e si interagisce con il committente stesso per assestare i requisiti. Nel caso risulti necessario, si innesca una nuova iterazione sull'analisi del problema e si ridefinisce il piano di lavoro.
6. Si imposta una seconda iterazione, tenendo conto anche dei requisiti e degli scenari secondari non affrontati nella prima iterazione. Una volta realizzato e collaudato il nuovo prototipo, si torna a interagire con il committente iniziando, se sarà il caso, nuove iterazioni fino alla "copertura" di tutti i requisiti.
7. Si costruisce la versione di distribuzione del prodotto e si procede alla eventuale installazione.

L'archetipo del sito cui faremo riferimento introduce un insieme di voci che verranno discusse nelle sezioni che seguono.

In questa sede osserviamo che, ponendo al centro dell'attenzione i requisiti, le viste del sistema espresse attraverso artefatti di vario tipo, inclusi modelli UML, possono porre in luce aspetti diversi, concettuali o fisici, in relazione ai diversi attori coinvolti nel processo. La figura che segue esprime una possibile articolazione di riferimento:



L'insieme dei modelli che formano gli artefatti costruiti nelle varie fasi del processo deve formare una descrizione completa, consistente e non troppo ridondante (di una vista) del sistema. La transizione tra i diversi modelli deve essere continua, cioè deve avvenire quanto più possibile "senza cuciture" (in modo **seamless**) facendo in modo che un oggetto in un modello abbia il suo o i suoi corrispettivi in un altro.



Il concetto di transizione seamless tra modelli è alla base della proprietà di tracciabilità (**traceability**), per cui, in qualsiasi direzione si percorra la sequenza di modelli generati, deve essere possibile mappare uno o più oggetti in un modello in uno o più oggetti in un altro.

Modello del dominio

Concludiamo queste note di carattere generale osservando che la definizione di un **sistema software** implica spesso la definizione di una collezione di **domini** e di **dipendenze tra domini**. Il termine **dominio** denota un mondo relativamente autonomo, reale o astratto, costituito da un insieme di **entità** che operano in accordo a precise regole, politiche e vincoli. Categorie di domini frequentemente usate sono:

- **Dominio di applicazione**: rappresenta il sistema dal punto di vista dell'utente.
- **Dominio di architettura**: rappresenta le strategie globali di progetto.
- **Dominio di servizio**: fornisce servizi generici di supporto al dominio applicativo.
- **Dominio di implementazione**: rappresenta componenti o piattaforme software pre-esistenti o legacy.

Un dominio viene rappresentato in UML da un **package**. Un dominio può fare assunzioni sulla esistenza di altri domini, caratterizzati da precise proprietà.

Visione

L'artefatto relativo alla visione può essere un documento ([html](#), [word](#), [xml](#), etc) che descrive le motivazioni generali (sociali, economiche, personali, etc) che hanno indotto una persona, un gruppo o un'azienda ad impegnarsi nello sviluppo dell'applicazione, evidenziando i benefici che si prevede scaturiscano dalla applicazione stessa.

Esempio: lo sviluppo di un'applicazione basata su dispositivi mobili per il monitoraggio di temperatura, pressione, etc di un paziente e per l'invio di tali dati via connessioni wireless a centri specializzati può promuovere l'assistenza di persone sole, anziani ed evitare lunghi e costosi ricoveri fuori dalle loro abitazioni.

Obiettivi

L'artefatto relativo agli obiettivi può essere un documento che illustra gli obiettivi concreti che si vogliono raggiungere grazie alla costruzione dell'applicazione o in conseguenza dei suoi utilizzi o degli sviluppi futuri.

Esempio: l'applicazione "monitoraggio dati di un paziente" intende produrre dispositivi mobili a basso costo capaci di emettere-ricevere dati in modo interoperabile con i web server o i web services di centri specializzati di gestione dei pazienti.

Collegato agli obiettivi vi può essere anche un **modello del business**, per esprimere il contesto in cui deve operare il sistema e per definire un vocabolario di termini e concetti di riferimento utili nella fase di analisi dei requisiti.

Requisiti

La costruzione di un sistema software implica il soddisfacimento di due categorie di requisiti: i **requisiti funzionali**, direttamente connessi al problema o all'applicazione e **requisiti non funzionali** che mirano a conseguire proprietà generali e strategiche.

La specifica dei requisiti costituisce un documento di interesse sia del committente sia del progettista. La versione dei requisiti di interesse del committente viene detta **Customer requirements** (**C-requirements**); la versione di interesse del progettista-sviluppatore viene detta **Developer requirements** (**D-requirements**).

I requisiti devono essere redatti in modo che risultino comprensibili e tali da definire in modo univoco **cosa** il sistema deve fare. A tal fine è desiderabile che i requisiti siano:

<ul style="list-style-type: none">• Chiari• Corretti• Completati• Concisi	<ul style="list-style-type: none">• Precisi• Non ambigui• Consistenti	<ul style="list-style-type: none">• Tacciaibili• Modificabili• Verificabili (collaudabili)• Realizzabili
--	---	---

I requisiti vengono normalmente espressi da documenti scritti in linguaggio naturale. In particolari contesti applicativi (ad esempio nei sistemi **real time**) questa descrizione **deve** essere completata da specifiche formali. Alla base delle specifiche formali vi sono notazioni matematiche che permettono di focalizzare l'attenzione sullo stato di un (sotto)sistema prima e dopo la esecuzione di una azione. Un noto linguaggio di specifica formale è il linguaggio **Z** [Z].

Tracciabilità

Si dice **traceability** la capacità di porre in corrispondenza ciascun requisito con le corrispondenti parti di analisi, progetto e di implementazione. La tracciabilità da un requisito alla implementazione si dice **forward traceability** mentre la tracciabilità dalla implementazione ai requisiti si dice **backward traceability**.

Più in particolare, la **forward traceability** può riguardare la tracciabilità tra **D-requirement** ed implementazione mentre la **backward traceability** può riguardare la tracciabilità tra **D-requirements** e **C-requirements**.

La matrice che segue:

Requirement ID	Class1	Class2	Class3	Validated by
2314	op1(...)	op2(...)	op3(...)	Unit test 22
2315	op1(...)	op4(...)	op5(...)	Unit test 37

rappresenta un esempio di **matrice di tracciabilità**, che correla ciascun requisito a un insieme di operazioni definite entro classi. L'ultima colonna riporta la unità di collaudo del requisito.

La tabella mostra che una modifica al requisito 2314 potrebbe implicare modifiche in più operazioni e che la modifica a una di queste (op1 di Class1) potrebbe ripercuotersi sul requisito 2315.

Al procedere dello sviluppo, il documento dei requisiti (**D-requirements**) dovrebbe essere mantenuto consistente con il progetto e la implementazione. Questo obiettivo, spesso

difficile da ottenere, può essere facilitato dalla generazione automatica di codice, tra cui i piani di collaudo (si veda [Piano di collaudo](#)).

Standard documentali IEEE per i requisiti

Lo standard [IEEE 830-1993](#) (che è anche uno standard [ANSI](#)) invita a strutturare un documento dei requisiti come segue:

1. **Introduzione**
 - a. Purpose
 - b. Scope
 - c. Definitions, acronyms and abbreviations
 - d. References
 - e. Overview
2. **Overall description**
 - a. **Product perspective**
 - i. System interfaces
 - ii. User interfaces
 - iii. Hardware interfaces
 - iv. Software interfaces
 - v. Communications interfaces
 - vi. Memory constraints
 - vii. Operations
 - viii. Site adaption requirements
 - b. Product functions
 - c. User characteristics
 - d. Constraints
 - e. Assumptions and dependencies
 - f. Apportioning and requirements
3. **Specific requirements**
4. **Supporting information**

Il capitolo 2 ([Overall description](#)) può essere considerato relativo alla descrizione dei [C-requirements](#), mentre il capitolo 3 ([Specific requirements](#)) può essere considerato relativo alla descrizione dei [D-requirements](#). La sezione 2.a.7 ([Operations](#)) può essere fatta corrispondere alla descrizione degli [use cases](#).

Analisi dei requisiti

La lettura dei requisiti può essere condotta in diversi modi, ciascuno correlato a diverso punto di vista "organizzativo":

- cercando di definire un [modello dei casi d'uso](#) basato sulle relazioni di generalizzazione, inclusione, estensione;
- cercando di definire le [features](#) con cui l'applicazione si mostra ad un osservatore esterno, articolando ciascuna [feature](#) in una gerarchia di funzioni;
- cercando di organizzare i requisiti in classi;
- cercando di associare i requisiti ai diversi [stati](#) in cui l'applicazione si può trovare. Per ciascun stato si determinano gli [eventi](#) che possono colpire l'applicazione mentre si trova in quello stato.

Un modo per impostare l'analisi è rivolgere particolare attenzione ai [sostantivi](#) e ai [verbi](#) che compaiono nel testo. Da una sistematica analisi dei [sostantivi](#) è possibile infatti formalizzare la conoscenza sul dominio applicativo e favorire la produzione di un

modello del dominio basato su insiemi di astrazioni (**entità del dominio**) caratterizzati da proprietà, politiche e vincoli comuni. Dalla analisi dei **verbi** è possibile individuare l'insieme delle **azioni** che il sistema dovrà compiere.

Sia le entità che le azioni possono essere rappresentate da diagrammi **UML**. A tal fine può essere utile distinguere tra classi che modellano:

- **Entità concettuali**, spesso correlate a cose tangibili e a **nomi** nel documento dei requisiti.
- **Ruoli** classi che rappresentano ruoli giocati da cose, persone od organizzazioni. Si manifestano spesso come **sottotipi**.
- **Episodi (incidents)**, che rappresentano qualcosa che accade in qualche istante specifico di tempo, presente, passato o futuro.
- **Interazioni**, che rappresentano operazioni (**transazioni**) o contratti che coinvolgono due o più classi. Le interazioni possono essere **asincrone** o **sincrone**
- **Specifiche** che rappresentano qualità comuni possedute da oggetti di un'altra classe

Artefatti relativi ai requisiti

Gli artefatti devono definire i requisiti funzionali e non funzionali che l'applicazione è chiamata a soddisfare. I requisiti vengono normalmente espressi da documenti scritti in linguaggio naturale. In particolari contesti applicativi (ad esempio nei **sistemi real time**) questa descrizione deve essere completata da specifiche formali.

Una pratica ormai diffusa è definire

- un **glossario** che riporta una spiegazione/definizione dei termini usati nel testo che descrive i requisiti;
- una traduzione dei requisiti in modelli UML dei **casi d'uso** (si veda xref href="#">);
- documenti volti a descrivere **scenari** (si veda xref href="#">) che illustrano le attività connesse ad un caso d'uso con dati reali.

Con questi artefatti l'analista dei requisiti cerca di chiarire il significato delle richieste del committente, ponendosi anche dal punto di vista dell'utilizzatore finale dell'applicazione, in modo da evitare ambiguità, incomprensioni e omissioni.

Un importante artefatto che può accompagnare il modello dei requisiti è costituito da una prima versione del **modello del dominio**, che sarà poi sviluppata nella fase dell'analisi del problema.

Modellazione del dominio applicativo

In generale, se si decide di organizzare i requisiti in classi, queste classi non sono in alcun modo pensate per essere utilizzabili in fase di progetto e di implementazione. Tuttavia, individuare classi che possano essere mantenute anche nelle fasi di progetto e di implementazione promuove la correlazione tra requisiti, progetto e implementazione e quindi la tracciabilità.

La raccolta e l'analisi dei requisiti mira ad una corretta **identificazione** e modellazione del dominio applicativo e delle funzioni che l'applicazione deve svolgere, cioè la specifica di **cosa** il sistema deve fare. Nella modellazione di un dominio è di fondamentale importanza usare esclusivamente il **vocabolario** terminologico di quel dominio.

Il **vocabolario** di un dominio è una lista di definizioni dei termini usati nel definire i requisiti di una applicazione in quel dominio

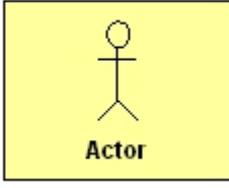
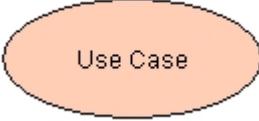
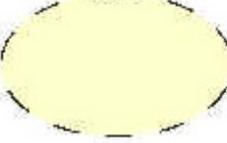
La definizione del vocabolario di dominio è uno dei fattori chiave per migliorare la comunicazione tra i diversi attori del processo di costruzione, in particolare tra analisti e progettisti. In particolare, ciascuna entità di dominio che è possibile evincere dalla analisi dei requisiti può essere espressa da una classe **UML** e posta in relazione logica con altre entità del dominio sfruttando le diverse relazioni di **associazione** che **UML** permette di esprimere.

La modellazione del dominio applicativo costituisce spesso uno dei cardini nel processo di sviluppo.

Casi d'uso

Gli **use cases** sono stati proposti da Jacobson [JCO92] come un modo sistematico e naturale per la raccolta di **C-requirements** attraverso la specifica della interazione tra il sistema e gli agenti esterni (detti **attori**) ad esso interessati. L'intento è descrivere le funzionalità che **danno valore** al sistema e non le funzionalità relative alla sua organizzazione interna.

Un attore (**Actor**) identifica il **ruolo** che una entità esterna assume quando interagisce direttamente con il sistema.

Figure 1. Actor	Figure 2. Use case	Figure 3. Use case realization
		

Un **Actor** modella agenti anche di natura molto diversa tra loro, quali persone, sistemi elettronici, sistemi meccanici, etc. che possono avere una qualche interazione con il sistema. Dal momento che il loro comportamento è definito esternamente al sistema in costruzione, non necessitano di alcuna ulteriore specifica.

L'ovale rappresenta in **UML** un caso d'uso (use case) che a sua volta specifica una **interazione** tra il sistema e uno o più **actor**, unitamente alle funzionalità che il sistema deve eseguire, agli occhi dei suoi utilizzatori. Un caso d'uso rappresenta una sequenza di azioni eseguita dal sistema per produrre un risultato osservabile che abbia valore agli occhi di un actor; questa azione può essere decomposta in altri use cases.

L'ovale tratteggiato rappresenta una collezione di oggetti che interagiscono per realizzare un use case o un'operazione. Questa icona viene anche usata per rappresentare un **pattern** o una **collaborazione parametrizzata**, cioè uno schema i cui partecipanti sono astratti e sono sostituiti da enti concreti al momento dell'utilizzo.

Gli use case sono utili non solo per la comprensione dei requisiti, ma anche per:

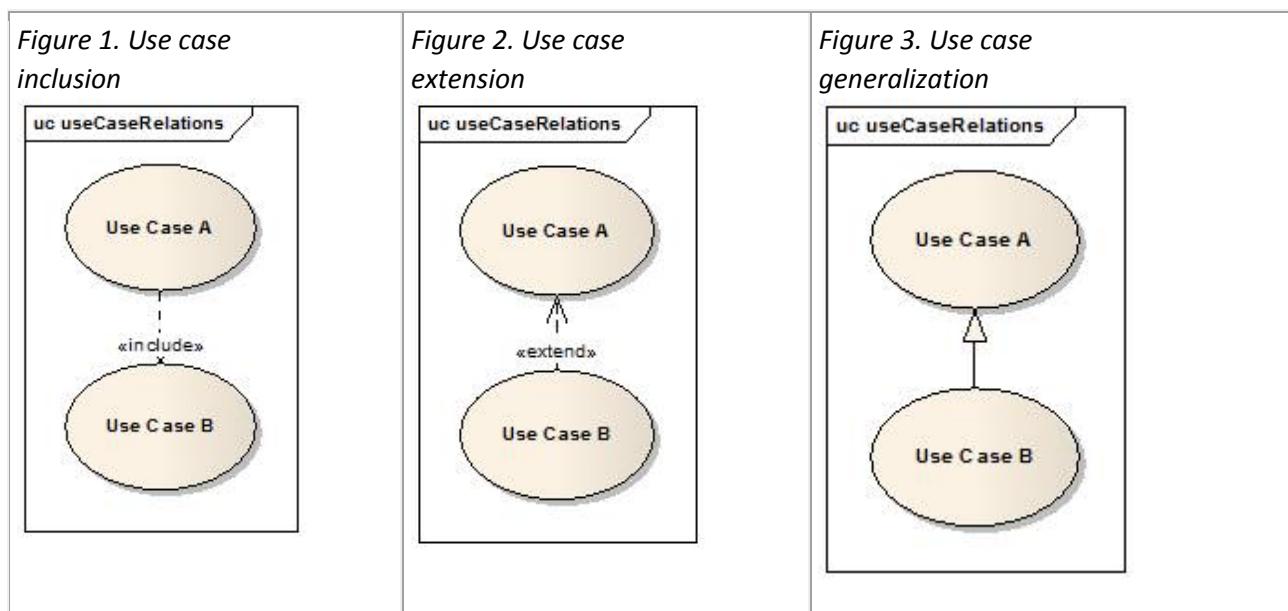
- guidare la individuazione di un modello del dominio;
- promuovere la corrispondenza (tracciabilità) tra requisiti, progetto e implementazione;
- impostare tecniche di **rapid prototyping**;
- individuare e specificare piani di collaudo.

Ad ogni use case è di solito associata, una documentazione supplementare che può essere sia semplicemente testuale sia espressa in termini di diagrammi UML di attività (*State Machine Diagrams*, *Activity Diagrams*, *Sequence Diagrams*, etc).

Tra i diversi tipi di diagrammi delle attività, i *sequence diagram* richiedono una decomposizione ad oggetti, che deve essere impostata rimanendo sempre dal punto di vista dell'utente, evitando di introdurre concetti che riguardano la progettazione. A tal fine, le classi che compaiono nelle *swimlines* sono rappresentative degli *attori* che iniziano azioni distinte e delle *classi-chiave* del dominio.

Relazioni tra use cases

UML prevede notazioni per organizzare i requisiti (in particolare *D-requirements*) espressi in forma di use cases.



La relazione di inclusione (stereotipo `<<include>>`) da uno use case A ad uno use case B indica che all'interno del flusso principale di A viene **obbligatoriamente** inserito il flusso di B.

La relazione di estensione (stereotipo `<<extend>>`) da uno use case B ad uno use case C, indica che all'interno del flusso principale di uno use case C può essere **optionalmente** inserito, sotto determinate condizioni, il comportamento di uno use case B. Questo concetto non va confuso con il concetto legato all'ereditarietà della programmazione ad oggetti.

La relazione di generalizzazione tra uno use case C ed uno use case A indica che C è una specializzazione di A.

Scenari

La esecuzione delle attività connesse ad un caso d'suo **con dati reali** viene detta **scenario**. Per descrivere uno scenario può essere utilizzato un **template** come quello che segue:

Campo	Descrizione

ID(Nome)	Identificatore univoco del caso d'uso
Descrizione	Descrever lo scenario
Attori	Describe gli attori coinvolti nello scenario.
Precondizioni	Descrizione delle proprietà rilevanti richieste dal sistema prima della esecuzione del caso d'uso.
Scenario principale	Descrizione del comportamento del sistema in situazioni standard.
Scenari alternativi	Descrizione del comportamento del sistema in situazioni particolari o eccezionali.
Postcondizioni	Descrizione delle proprietà che devono essere vere quando le attività connesse al caso d'uso sono state completate.

Uno scenario prefigura uno o più situazioni di **collaudo** ciascuna delle quali può essere formalizzata da un **test case** espresso in termini delle entità di dominio.

Requisiti non funzionali

Oltre ai requisiti non funzionali di tipo generale connessi ad ogni "buon codice" (si veda Qualita' di processo e di prodotto) tipici requisiti non funzionali sono:

Efficienza (performance)	Specifica i vincoli sul tempo di esecuzione, sull'uso di risorse di memoria (primaria e secondaria) etc.
Affidabilità (reliability)	Specifica il livello di fallimento accettabile
Robustezza (availability)	Specifica il periodo di tempo in cui l'applicazione deve essere operativa per gli utenti
Scalabilità (scalability)	TODO
Persistenza (persistence)	Specifica quali dati del sistema devono essere mantenuti in modo persistente, ad esempio per permettere il riavvio del sistema in caso di shutdown intenzionale o non atteso
Portabilità (portability)	Specifica su quali diverse piattaforme operative il sistema deve poter essere eseguito
Configurabilità (configurability)	Specifica se il sistema può avere diverse configurazioni e se queste possono essere definite in modo statico (off-line) o dinamico (senza fermare il sistema)
Generalità (generality)	Specifica se il sistema deve fornire una soluzione di tipo generale nel dominio, in modo che possa essere riusato in altre applicazioni
Usabilità (usability)	Specifica come il software supporta le attività di un utente umano. Include affordance (il costo di apprendimento della una user interface) e accessibility .(misura dell'insieme di utenti che possono interagire in modo efficace con il sistema)

Supportabilità (<i>supportability</i>)	Specifica il costo di mantenimento del software dopo che è stato distribuito al consumatore.
Gestione degli errori	Specifica come l'applicazione deve rispondere ad errori del suo ambiente o della applicazione stessa
Interoperabilità	Specifica come l'applicazione comunica con il suo ambiente o con altre applicazioni
Requisiti inversi	Specifica quello che l'applicazione non deve fare (per chiarire possibili fraintendimenti)

Analisi

L'analisi del problema parte sempre da una attenta lettura del testo che descrive i requisiti (si veda Requisiti) e degli scenari in cui essi sono articolati.

Un punto importante da ricordare è che l'analisi **non** si pone come obiettivo la descrizione di quali proprietà strutturali e comportamentali debba possedere il **sistema che risolve** il problema: questo è l'obiettivo della fase di progetto. Piuttosto la fase di analisi mira alla definizione della Architettura Logica implicata dai requisiti e dalle problematiche individuate.

Analisi del problema

Dalla lettura degli scenari definiti nella fase di analisi dei requisiti (si veda Analisi dei requisiti) si può individuare un primo insieme di problemi; si procede quindi ad effettuare l'**analisi dei problemi** avendo come obiettivo la definizione di una **architettura logica** del sistema, basata sul **modello del dominio**, che trova in questa fase la sua completa definizione.

Artefatti dell'analisi

Il **modello dell'analisi** è costituito da un insieme di diagrammi UML costruiti per definire una struttura concettuale del sistema robusta e modificabile. Con questo modello l'analista del problema esprime fatti il più possibile "oggettivi" sul problema (**non sulla soluzione**) focalizzando l'attenzione su **sottosistemi, ruoli e responsabilità** insiti negli scenari prospettati durante la descrizione dei requisiti.

Idealmente, se l'analisi fosse svolta in modo contemporaneo da più gruppi di analisti diversi non comunicanti, i modelli proposti da ciascun gruppo dovrebbero risultare "semanticamente equivalenti"; in caso contrario il problema sarebbe mal definito e ogni fase successiva sarebbe compromessa fin dall'inizio.

Un artefatto di sintesi dell'analisi è costituito dall'**architettura logica** del sistema, un modello UML che descrive la struttura (insieme delle parti), il comportamento atteso (dal tutto e da ciascuna parte) e le interazioni così come possono essere dedotte dai requisiti, dalle caratteristiche del problema e del dominio applicativo, senza alcun riferimento alle tecnologie realizzative.

Questo modello costituisce il punto di riferimento per l'analisi dei rischi, la pianificazione del lavoro, la pianificazione del collaudo e per la selezione (o lo sviluppo) delle tecnologie da utilizzare nella fase di progetto.

Architettura Logica

L'analisi del problema può porre in evidenza un insieme di **vincoli** e di **proprietà** desiderate o desiderabili che limitano di fatto lo spazio delle soluzioni. In altre parole, se l'analisi del problema riesce ad identificare le **forze** rilevanti in gioco e a stabilire come esse interagiscono e confliggono tra loro, allora è possibile anche delineare i **trade-offs** che ne possono emergere e specificare i tratti salienti di una **architettura logica** in grado di fornire un limite alla creatività inconcludente e costituire un buon punto di partenza per la risoluzione del problema stesso.

L'architettura logica di un sistema costituisce quindi un modello del sistema ispirato dai requisiti funzionali e dalle forze in gioco nel dominio applicativo o nella specifica applicazione e mira ad identificare i macro sottosistemi in cui il problema stesso suggerisce di articolare il sistema risolvente. L'architettura logica è il più possibile indipendente da ogni ipotesi sull'ambiente di implementazione.

Un modo per valutare la qualità di una architettura logica e la coerenza con i requisiti è dare risposta alle seguenti domande:

- E' possibile addentrarsi nei dettagli dell'architettura procedendo a livelli di astrazione via via decrescenti o siamo di fronte a un ammasso non organizzato di classi?
- Le dipendenze tra le parti sono state impostate a livello logico o riflettono (erroneamente) una visione implementativa?
- Se nel modello compaiono entità denotate da termini **non definiti** nel glossario costruito dall'analista dei requisiti, quale è la motivazione della loro presenza? Sono elementi realmente necessari a questo livello o siamo di fronte ad una prematura anticipazione di elementi di progettazione?
- Se nel modello **non compaiono** entità corrispondenti a termini definiti nel glossario, quale è la motivazione della loro mancanza? Siamo di fronte a una dimenticanza o vi sono ragioni reali per non includete questi elementi?

Schemi architetturali

Jacobson [JCJO92] propone una metodologia di costruzione dell'architettura di un sistema basata sulla partizione sistematica degli **use cases** in oggetti di tre categorie, ottenute articolando lo spazio concettuale in tre dimensioni: **informazione**, **presentazione** e **controllo**. A ciascuna di queste dimensioni corrisponde una specifica classe di oggetti, ciascuna rappresentata da uno stereotipo, cui corrisponde una specifica icona, ora adottata anche nel **Rational Unified Process (RUP)**.

<p>Figure 1. Informazione</p> 	<p>Entity</p> <p>E' la dimensione relativa alle entità del sistema cui corrisponde l'insieme degli oggetti che includono funzionalità relative alle informazioni che caratterizzano il sistema; questi oggetti sono denominati oggetti entità e costituiscono gran parte del modello del dominio (si veda Analisi dei requisiti).</p>
<p>Figure 2. Presentazione</p> 	<p>Boundary</p> <p>E' la dimensione relativa alle funzionalità che dipendono dall'ambiente esterno cui corrisponde l'insieme degli oggetti che incapsulano l'interfaccia del sistema verso il mondo esterno; questi oggetti sono denominati oggetti interfaccia.</p>

Figure 3. Controllo

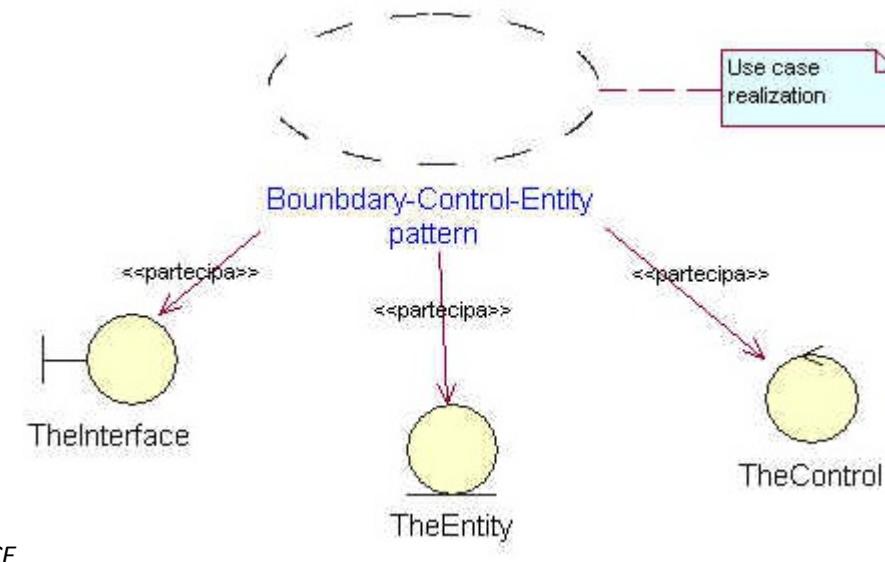


Control

E' la dimensione relativa agli enti che incapsulano il controllo cui corrisponde l'insieme degli oggetti includono funzionalità non incapsulabili negli oggetti delle categorie precedenti. Il loro compito, tipicamente, è di fare da collante tra gli oggetti interfaccia e gli oggetti entità; questi oggetti sono denominati **oggetti controllo**.

La proposta di impostare l'architettura di un sistema software distinguendo tra **boundary objects**, **control objects** ed **entity objects** costituisce un solido punto di partenza anche per il disegno dell'architettura logica di molte applicazioni. Infatti l'analista tende ad affrontare la complessità dei problemi partizionando i problemi stessi in sotto-problemi; è quindi del tutto logico che un analista eviti di associare alle entità di un dominio sia le funzionalità di una specifica applicazione sia le funzionalità tipiche della interazione con l'utente. Ne consegue un classico approccio all'analisi rappresentabile in **UML**, da un diagramma come quello che segue:

Figure 4.



La conseguenza è che l'architettura di un sistema software risulta quasi fisiologicamente articolata in una sequenza di livelli (**layer**) verticali che viene mantenuta anche in fase di progetto e realizzazione.

A questa articolazione verticale si accompagna spesso una articolazione in layer orizzontali tipica delle piattaforme (si veda [Piattaforme operative](#)).

Layers

L'architettura di un sistema software risulta quasi fisiologicamente articolata in una sequenza di **layer** verticali ed orizzontali che parte dalla fase di analisi e viene spesso mantenuta anche in fase di progetto e di realizzazione.

Layer verticali

Per quanto riguarda la ripartizione verticale è ormai scontato separare la parte che realizza le entità dell'applicazione (**dominio**) dalla parte che realizza l'interazione con l'utente (**logica di presentazione**), introducendo una parte centrale di connessione (**control**), che nelle applicazioni di rete viene spesso denominata **middleware**.

Più specificatamente:

- il livello delle **entità** forma il (modello del) dominio applicativo;
- il livello di **presentazione** comprende le parti che realizzano l'interfaccia utente. Per aumentare la riusabilità delle parti, questo livello è progettato e costruito astraendo quanto più possibile dai dettagli degli specifici dispositivi di I/O;
- il livello di **applicazione** comprende le parti che provvedono ad elaborare l'informazione di ingresso, a produrre i risultati attesi e a presentare le informazioni in uscita.

Livelli orizzontali

Ognuno dei livelli orizzontali definisce e realizza funzionalità (in forma di **API**) e servizi che vengono visti ed usati come enti primitivi dal livello superiore.

Una classica ripartizione orizzontale è ad esempio:

- livello applicativo (**top**);
- servizi di base;
- piattaforma di supporto;
- sistema operativo (**bottom**).

Analisi dei rischi

L'analisi dei rischi può essere relativa agli effetti di un sistema o al processo di produzione.

Per quanto riguarda il processo, osserviamo che la definizione di un'architettura logica del sistema è indispensabile se si vuole porre in luce le parti più critiche per la progettazione e per lo sviluppo, in relazione alle risorse, alle competenze, alle tecnologie disponibili, etc. e per fornire utili elementi su cui impostare il piano di lavoro.

Per quanto riguarda gli effetti del sistema, il discorso richiederebbe un trattato a sé stante; in questa sede ci limitiamo a fissare qualche punto di riferimento.

Sistemi critici

I fallimenti del software sono relativamente comuni, spesso causano inconvenienti ma nessun danno serio a lungo termine; in taluni casi tuttavia il fallimento di un sistema software può portare a perdite economiche, danni fisici o minacce per la vita.

I sistemi tecnici o socio-tecnici da cui dipendono persone o aziende sono chiamati **sistemi critici**. Se questi sistemi non forniscono i loro servizi come ci si aspetta possono verificarsi seri problemi e importanti perdite. Ci sono tre tipi principali di sistemi critici:

1. Sistemi **safety-critical**: i fallimenti possono provocare incidenti, perdita di vite umane o seri danni ambientali.
2. Sistemi **mission-critical**: i malfunzionamenti possono causare il fallimento di alcune attività a obiettivi diretti.
3. Sistemi **business-critical**: i fallimenti possono portare a costi molto alti per le aziende che li usano.

La proprietà più importante di un sistema critico è la sua **fidatezza**. Ci sono quattro dimensioni di fidatezza:

1. Disponibilità (**availability**): è la probabilità che un sistema sia attivo, funzionante e in grado di fornire servizi utili in ogni momento.
2. Affidabilità (**reliability**): è la probabilità, in un determinato periodo di tempo, che il sistema fornisca correttamente i servizi come atteso dall'utente.
3. Sicurezza (**safety**): è una valutazione di quante possibilità ci sono che il sistema causi danni a persone o al loro ambiente.
4. Protezione (**security**): è una valutazione di quante possibilità ci sono che il sistema resista alle intrusioni accidentali e non.

Oltre a queste quattro principali caratteristiche della fidatezza ve ne sono diverse altre, tra cui in particolare la **tolleranza all'errore** la quale sancisce che il sistema deve essere progettato in modo da evitare e tollerare errori di immissione. Quando l'utente commette degli errori, il sistema dovrebbe, per quanto possibile, individuarli e correggerli automaticamente oppure chiedere all'utente di reinserire i dati.

Ci sono diverse ragioni per le quali la fidatezza è importante:

- I sistemi non affidabili, non sicuri e non protetti sono rifiutati dagli utenti.
- I costi di un fallimento del sistema potrebbero essere enormi.
- I sistemi inaffidabili possono causare perdita di informazioni.

Piano di collaudo

Al termine dell'analisi dei requisiti e dell'analisi del problema, i modelli che definiscono il dominio e l'architettura logica del sistema dovrebbero dare sufficienti informazioni su **cosa** le varie (macro)parti del sistema debbano fare senza specificare ancora molti dettagli del loro comportamento interno. Il "cosa fare" di una parte dovrà comprendere anche le forme di interazione con le altre parti.

Lo scopo del **piano di collaudo** è cercare di precisare il comportamento atteso da parte di una entità prima ancora di iniziare il progetto e la realizzazione. Focalizzando l'attenzione sulle interfacce delle entità e sulle interazioni è possibile impostare scenari in cui specificare in modo già piuttosto dettagliato la "risposta" di una parte ad uno "stimolo" di un'altra parte.

Lo sforzo di definire nel modo più preciso possibile un piano di collaudo di un sistema prima ancora di averne iniziato la fase di progettazione viene ricompensato da una miglior comprensione dei requisiti, da un approfondimento nella comprensione dei problemi e da una più precisa definizione dell'insieme delle funzionalità (operazioni) che ciascuna parte deve fornire alle altre per una effettiva integrazione nel "tutto" che costituirà il sistema da costruire.

In attesa di individuare modi per esprimere in modo organico, sistematico ed efficace la **semantica** delle entità che costituiscono un sistema, la definizione di piani di collaudo può

introdurre elementi utili a comprendere il significato delle entità e specificarne nel modo più chiaro possibile il comportamento atteso.

Un piano di collaudo va dunque concepito ed impostato da un punto di vista logico, cercando di individuare categorie di comportamenti e punti critici. In molti casi tuttavia può anche risultare possibile definire in modo precoce piani di collaudo concretamente eseguibili, avvolendosi di strumenti del tipo **JUnit** che sono ormai diffusi in tutti gli ambienti di programmazione.

Lo sforzo di definire un piano di collaudo concretamente eseguibile su una data piattaforma implementativa è una pratica tipica dei processi agili in quanto promuove uno sviluppo molto più controllato, sicuro e consapevole del codice poiché il progettista e l'implementatore possono verificare subito in modo concreto la correttezza (o meglio la non scorrettezza rispetto ai collaudi) di quanto sviluppato. Per definire collaudi eseguibili prima di disporre del codice implementativo, risulta spesso utile fare il riferimento ai pattern **Interface** e **Factory**.

Il tema della pianificazione del collaudo verrà approfondito nel seguito con riferimento a casi concreti (si veda ad esempio [Esempio di sviluppo model based](#)). Anticipiamo in questa sede una breve introduzione a **Junit** in quanto portatore di concetti e meccanismi ormai indispensabili in ogni processo di produzione del software.

JUnit

<="" a="" style="margin-top: 1em;">>Junit (<http://www.junit.org/index.htm>) è uno strumento ideato da **Beck** e **Gamma** come supporto al concetto di **collaudo continuo** nell'ambito di processi di sviluppo agili e incrementali. Esso introduce alcuni concetti e termini ormai divenuti comuni in tutti gli ambienti di sviluppo:

Test fixture	Insieme delle risorse richieste da un collaudo per poter operare.
Test case	Associa un fixture ad un insieme di attività di collaudo (tests).
Test suite	Una collezione di Test Cases tra loro collegati.

Una classe di collaudo deve essere definita come specializzazione di una classe predefinita (in Java la classe **junit.framework.TestCase**) ed internamente organizzata secondo la struttura che segue:

```
public class XXX extends junit.framework.TestCase{  
    /* -----  
     * Costruttore (non indispensabile ma opportuno)  
     * ----- */  
    public XXX(String name) {  
        super(name);  
    }  
  
    /* -----  
     * Operazione opzionale di inizializzazione  
     * ----- */  
    protected void setUp() throws Exception { super.setUp();... }  
  
    /* -----  
     * Operazione opzionale per definire  
     * una suite di collaudo  
     * ----- */  
    public static Test suite() {  
        return new TestSuite(XXX.class);  
    }  
}
```

```

}

/* -----
 * Operazione di collaudo
----- */
public void testYYY() {
    ...
    assertTrue( comment, booleanExpression );
}

/* -----
 * Operazione opzionale di finalizzazione
----- */
protected void tearDown() throws Exception{ super.tearDown();... }

/* =====
   Il main trasmette la classe di collaudo a un tool
   specializzato per l'esecuzione di operazioni di testing
   ===== */
public static void main(String args[]) {
    junit.textui.TestRunner.run(XXX.class);
} //main

} //myTest

```

La tabella che segue riporta il significato e il ruolo degli elementi principali:

<code>void testYYY()</code>	Operazione di collaudo (priva di argomenti) eseguita in modo automatico.
<code>void setUp()</code>	Metodo invocato prima di ogni operazione di collaudo per definire dati ed oggetti globali, che costituiscono la test fixture .
<code>void tearDown()</code>	Metodo invocato alla fine di ogni operazione di collaudo per ripulire le fixture da informazioni non eliminabili in modo automatico.
<code>junit.textui.TestRunner</code>	Strumento che automatizza l'esecuzione delle operazioni di collaudo, generando un rapporto.

La classe adibita al collaudo eredita da `junit.framework.TestCase` la capacità di individuare ed eseguire automaticamente le operazioni di collaudo in essa definite con signature della forma

```
public void testYYY(){...}
ove YYY denota una qualsiasi stringa definita dall'utente.
```

La classe eredita inoltre la capacità di visualizzare automaticamente i messaggi di spiegazione relativi ad operazioni di asserzione quali:

<code>assertTrue(b)</code>	Il test ha successo se il valore booleano b vale true .
<code>assertFalse(b)</code>	Il test ha successo se il valore booleano b vale false .
<code>assertNull(obj)</code>	Il test ha successo se obj ha valore null .
<code>assertNotNull(obj)</code>	Il test ha successo se obj ha valore not null .
<code>assertTrue(b)</code>	Il test ha successo se il valore booleano b vale true .
<code>assertEquals(x,y)</code>	Il test ha successo se x e y denotano valori primitivi uguali o denotano oggetti uguali (secondo il metodo equals).

<code>assertSame(ox,oy)</code>	Il test ha successo se <code>ox</code> e <code>oy</code> denotano lo stesso oggetto.
<code>assertNotSame(ox,oy)</code>	Il test ha successo se <code>ox</code> e <code>oy</code> non denotano lo stesso oggetto.
<code>fail()</code>	Il test fallisce.

Piano di lavoro

I requisiti possono essere classificati in tre categorie:

- essenziali
- desiderabili
- opzionali

L'ideale (regola di Pareto) è che l'80% dei risultati di un'applicazione possa essere raggiunto dal soddisfacimento del 20% dei requisiti.

Il piano di lavoro articola lo sviluppo in una successione di **prototipi**, il cui rilascio, nei tempi stabiliti, rende più concreta l'interazione con il committente e concorre a precisare e stabilizzare i requisiti funzionali.

La fase di analisi (dei requisiti e del problema) è essenziale per la definizione del piano di lavoro e per la distribuzione dei compiti tra più persone, tenendo conto delle parti da costruire (si veda Architettura Logica) e dell'analisi dei rischi.

Progetto

Lo scopo della fase di progettazione consiste nel raffinamento dell'architettura logica del sistema, considerando tutti gli aspetti vincolanti che sono stati trascurati nelle fasi precedenti. La fase di progettazione deve mirare non solo a individuare e descrivere una soluzione al problema (**what**), ma soprattutto a descrivere i motivi (**why**) che hanno determinato questa soluzione.

L'architettura del sistema definita dal progetto dovrebbe scaturire come punto di equilibrio tra le forze in gioco nel problema e quindi dovrebbe risultare, come già l'architettura logica definita dall'analisi, ancora il più possibile indipendente dalle tecnologie realizzative. La fase di progettazione dovrebbe procedere dal generale al particolare, sviluppando per primi i sottosistemi più critici individuati dall'analisi.

La progettazione non procede necessariamente a cascata, ma può dare luogo a refactoring e può produrre anche retroazioni sulla fasi precedenti. Al progressivo sviluppo dell'architettura del sistema corrisponde un progressivo raffinamento dei piani di collaudo.

Progettazione

Per pervenire in modo sistematico ad un modello di progetto che sia già sensibile al livello di codifica, Jacobson [JCJO92] propone di articolare gli oggetti dell'analisi in blocchi (assimilabili ai moduli o ai package dei linguaggi di programmazione) e di porre particolare attenzione sulla interazione tra i blocchi.

A questo fine è utile l'uso di **interaction diagrams** con cui descrivere gli **use case** in termini di sequenze di stimoli tra blocchi. Poiché per stimolo si intende ogni elemento

che permette la comunicazione tra blocchi, l'obiettivo di questa fase è definire le **comunicazione esterne** tra i blocchi.

Tra le scelte da effettuare nella fase di progetto vi è anche (a meno non sia un requisito) la individuazione del linguaggio di programmazione, della piattaforma operativa di supporto e degli strumenti ritenuti utili alla costruzione del sistema.

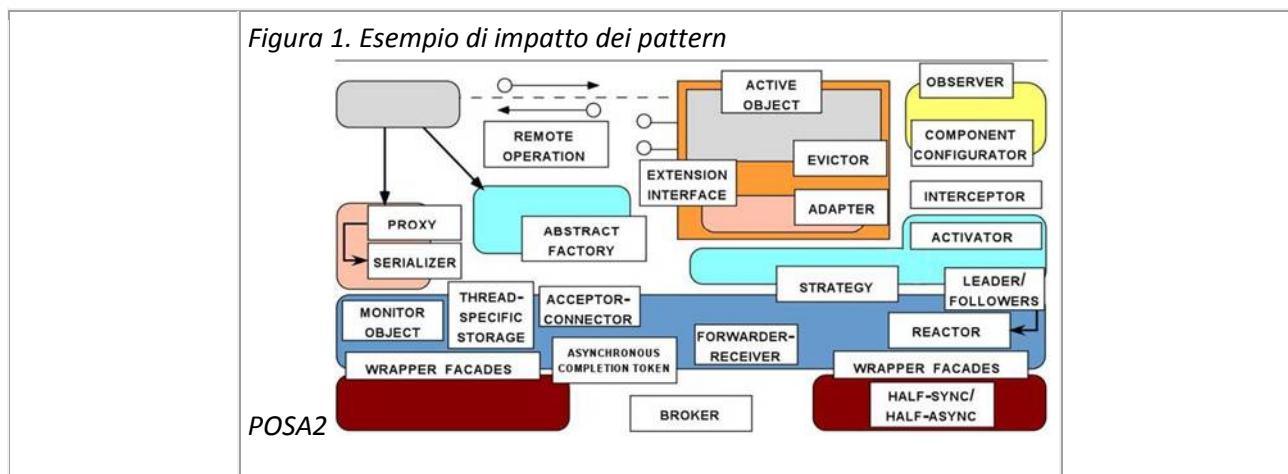
Artefatti di progetto

L'insieme degli artefatti relativi al progetto è costituito da un insieme di diagrammi UML che definisce la organizzazione logica della soluzione, cioè l'architettura di un sistema che soddisfa tutti i requisiti.

L'**architettura di progetto** può essere ottenuta in modo sistematico da una progressiva trasformazione dell'architettura logica in forme più articolate che preservano i vincoli stabiliti al livello precedente. Nel caso siano disponibili generatori / configuratori automatici di codice, la forma finale dell'architettura di progetto costituisce "il sistema". In caso contrario essa costituisce l'artefatto chiave con cui validare la soluzione prima di passare alla costosa fase di implementazione/codifica.

Un ruolo strategico nel processo di trasformazione dell'architettura logica del sistema in una architettura di progetto capace di soddisfare i requisiti, è costituito dall'uso dai pattern e dai pattern languages che promuove la motivazione sistematica delle scelte che progressivamente portano al raffinamento dell'architettura, fino alla sua forma finale.

La figura che segue mostra una panoramica di impiego dei pattern POSA2 nel contesto di una piattaforma a livelli (si veda Piattaforme operative).



Macro-parts of a software system

Se supponiamo di analizzare una moderna applicazione software "ben fatta" è assai probabile che essa si presenti logicamente organizzata in tre macro-parti:

- una parte che potrebbe essere condivisa in tutte le applicazioni future (**generic part**);
- una parte organizzata secondo alcuni schemi ben riconoscibili (ad esempio in forma di pattern) riusabili in diversi contesti (**schematic part**);
- una parte del tutto specifica per l'applicazione e non generalizzabile in alcun modo (**specific part**).

Ciascuna delle parti individuate potrebbe (dopo un refactoring del codice, se necessario) essere incapsulata in un sottosistema; la generic part potrebbe essere inserita nel sottosistema **piattaforma**, la schematic part in un sottosistema che si appoggia alla piattaforma e la specific part in un sottosistema che si appoggia ai due precedenti.

Questa articolazione può costituire un importante punto di riferimento sia per progetti di tipo **bottom-up**, che partono da ciò che disponibile (linguaggio di programmazione, piattaforma, framework, etc) per giungere (con processi di sintesi) al sistema voluto, sia per progetti di tipo **top-down** che partono dal problema da ricolvere e cercano di decomporlo progressivamente (con processi di analisi) fino a giungere a elementi terminali disponibili (istruzioni, componenti, etc.).

Le sezioni che seguono sono destinate ad approfondire entrambi i tipi di progettazione.

Sviluppo bottom-up

Un'astrazione cattura conoscenza acquisita dall'esperienza su problemi ricorrenti, rendendo tale conoscenza riusabile come possibile punto di arrivo durante la fase di decomposizione di un problema. Di solito una nuova astrazione viene introdotta come una pratica isolata; successivamente viene riconosciuta come pattern e quindi incapsulata in uno strumento di produzione o in una piattaforma. Come ultimo passo un'astrazione viene accolta entro un linguaggio.

Tecnologie come **Java-J2EE** o **C#-.NET** possono essere caratterizzate dal tipo di allineamento piattaforma-linguaggio. Nel caso in cui il linguaggio sia più astratto della piattaforma, le astrazioni del linguaggio devono essere supportate e gestite da tools quali compilatori, debugger, generatori, etc. Nel caso in cui la piattaforma sia più astratta del linguaggio, l'utente deve farsi carico diretto dei concetti non incapsulati nel linguaggio, con conseguente aumento di complessità. Un caso classico è il concetto di **evento** assente in Java ma presente fin dalla Java Virtual Machine (**JVM**).

Spesso le moderne piattaforme permettono di utilizzare le astrazioni nelle forme promosse dal paradigma ad oggetti quali classi, oggetti, componenti software, servizi, etc. Questo fatto induce a impostare la progettazione (in tutto o in parte) non tanto come un processo (analitico) di decomposizione/raffinamento (**top-down**) quanto come un processo (sintetico) di composizione (**bottom-up**) che perviene alla soluzione di un problema attraverso una aggregazione di feature rese disponibili dalla piattaforma.

Nel loro complesso queste feature formano un nuovo **spazio di soluzione** per i problemi. La complessità viene ridotta in quanto una piattaforma presenta un numero di feature più limitato rispetto ai livelli sottostanti e quindi un minore insieme di combinazioni possibili. L'adattamento alle esigenze di una specifica applicazione è reso più agevole dal fatto che i componenti sono spesso progettati e realizzati in modo da essere estendibili e adattabili attraverso la esposizione di informazione via metadati usabili offline o a run time.

Naturalmente tutto questo ha un prezzo: la piattaforma incapsula e spesso congela decisioni di progetto che sarebbero state oggetto di una decisione esplicita in un processo top-down.

Il punto importante quindi è riconoscere da un lato la necessità pratica all'uso di piattaforme di alto livello per lo sviluppo industriale del software; da un altro lato occorre però anche riconoscere come sia necessario continuare a pensare in termini di astrazione senza focalizzarsi sullo spazio di soluzione fornito da una piattaforma. Solo così infatti si

può pensare di creare nuovi domini di soluzione in grado di affrontare in modo sistematico nuovi problemi, rimpiazzando i domini di soluzione connessi alle tecnologie esistenti e quindi aprire la via alla **innovazione di prodotto e di processo**.

Piattaforme e framework

Nella pratica comune, l'uso di una piattaforma operativa viene quasi sempre deciso **prima** di iniziare la costruzione dell'applicazione, con inevitabili conseguenze sul processo di produzione del software, tanto che si può parlare di un **platform-based design**. Abbiamo infatti già osservato (si veda [Sviluppo bottom-up](#)) come una piattaforma non solo fattorizzi elementi comuni a tutte le applicazioni, ma fornisca anche un insieme di astrazioni che permettono di ridurre l'**abstraction gap** tra il livello operativo e il dominio del problema.

Sul piano progettuale-operativo, il contributo principale di una piattaforma consiste nella possibilità di impostare lo sviluppo del software in termini di assemblaggio di componenti configurabili (**development by assembly**). Il termine "componente" è usato qui in senso lato, per indicare unità modulari, capaci di promuovere il riuso e di essere adattabili / estendibili in relazione alle esigenze di una specifica applicazione.

Un'applicazione sviluppata partendo da una delle piattaforme doperative oggi più diffuse mostra una precisa macro-organizzazione logica che viene spesso riflessa a livello di codice. Ad esempio, nel caso di una applicazione J2EE:

- l'applicazione è organizzata in layer orizzontali e verticali;
- i processi applicativi sono realizzati da componenti distribuiti;
- i componenti usano interfacce per esporre metodi invocabili localmente o in modo remoto e metodi per gestire l'identità, individuare factory e istanze, accedere ai metadati e gestire errori;
- le interfacce sono implementate attraverso pattern quali ([GHJV95] [POSA2])**factory, proxy, facade, adapter, delegate, event, command, exception, metadata**;
- i componenti potrebbero utilizzare connection factories, connections e adapter per interagire con componenti esterni;
- i componenti potrebbero utilizzare descrizioni di Web Services, gestori di risorse, servizi di discovery, schemi di documenti e documenti per interagire con altri processi applicativi;
- i componenti potrebbero utilizzare servizi di sistema per inizializzare e registrare istanze e per gestire aspetti legati allo stato, distribuzione, concorrenza, transazioni, naming, security;
- i componenti potrebbero usare **server pages, forms, scripts, servlet** per definire la parte di presentazione all'utente;
- la descrizione dei componenti sarebbe espressa in termini di molteplici artefatti, quali codice Java, file JXML, AR, WAR, EAR, etc

Una piattaforma permette l'adattamento allo specifico problema tramite meccanismi di tipo white-box o black-box. I meccanismi di tipo **white-box** si basano principalmente sulla **ereditarietà** tra classi e sui pattern [GHJV95] quali **Factory Method** e **Template Method**. Questo approccio richiede una conoscenza profonda della struttura delle classi della piattaforma e può condurre a inconsistenza. I meccanismi di tipo **black-box** nascondono il codice sorgente, si basano sulla associazione e implicano spesso l'uso di pattern quali **Decorator** e **Visitor**.

Le piattaforme stanno decisamente evolvendo [JF88] da framework basati su white-box a framework basati su black-box che sono più flessibili (in quanto le associazioni sono modificabili a tempo di esecuzione), richiedono solo la conoscenza di specifiche, encapsulano l'implementazione e portano a maggior stabilità. Rimanendo nell'alveo del paradigma ad oggetti, per framework si può intendere quanto riportato in [JF88]: "*a set of classes that embodies an abstract design for solutions to a family of related problems*". In generale per framework si può intendere un sistema software che cattura l'esperienza di risoluzione di problemi generali ricorrenti in forma di algoritmi riusabili, componenti software ed architetture estendibili.

Un framework pone a fattor comune le parti riusabili di un progetto e di una implementazione, separando la *generic part* dalla *specific part* di un'applicazione. Le parti che si prevedano cambino più spesso sono encapsulabili adottando il pattern *PluggableObjects* [RJ96]. Un framework propone/impone uno *stile architetturale* (si veda Lo stile architetturale) introducendo meccanismi volti a permettere variazioni e presenta un insieme di aspetti ricorrenti quali: *definition, discovery, composability, adaptability, deployment, distribution*.

Piattaforme come J2EE e .NET realizzano la previsione di Gamma che la crescente importanza dei framework avrebbe condotto ad impostare applicazioni di grandi dimensioni come layer di framework interagenti, ciascuno dedicato a qualche rilevante aspetto dell'applicazione (GUI, persistence, etc).

In un processo di sviluppo di tipo top-down, un framework specializzato potrebbe scaturire dall'analisi (del dominio) del problema [RJ96]; una ditta potrebbe valutare conveniente investire nel suo sviluppo al fine di creare, secondo un adatto stile architetturale, il supporto ad una possibile famiglia di applicazioni, prevedendo la possibilità di riuso ed estensione.

Application framework

Supponendo di applicare ripetutamente il principio della separazione con riferimento a molte applicazioni in un dato dominio, si giunge alla individuazione non tanto di un framework general purpose quale J2EE o .NET quanto ad un *application framework*. Oltre ai vantaggi che scaturiscono da una organizzazione a componenti, l'uso di un application framework risulta vantaggioso in quanto propone e supporta un insieme di *astrazioni riusabili* per una intera famiglia di prodotti.

Lo sviluppo di un application framework scaturisce nella *fase di consolidamento* di un'applicazione [Foote92] quando si effettua un refactoring del software per migliorare l'organizzazione senza modificare il funzionamento (osservabile). Questo può essere fatto da uno stesso team che sviluppa più applicazioni in successione o da team diversi che sviluppano in parallelo [RJ96]. Il secondo approccio è più oneroso sia in tempo che in risorse, ma permette di esercitare prospettive diverse, migliorando la generalità delle astrazioni riusabili che si possono individuare.

L'interesse a definire una *famiglia di prodotti* può nascere da parte di *System Integrators* (SI) nel portare una stessa applicazione a più clienti, o da parte da *Independent Software Vendors* (ISV) nello sviluppo di diverse applicazioni in specifici domini come ad esempio CRM (*Customer Relationship Management*) o da parte di organizzazioni IT ad esempio nel predisporre versioni multiple di un'applicazione attraverso maintenance ed enhancement.

Un application framework tuttavia non può essere definito una volta per tutte; il dominio applicativo presenterà inevitabilmente nuove forze o nuove prospettive per cui occorre rimettere mano alla parte architettonale comune. Inoltre vi possono essere retroazioni (feedback) sul framework da parte di coloro che sviluppano le applicazioni che possono indurre a un refactoring del framework stesso.

D'altra parte un application framework deve essere quanto più stabile possibile in quanto mira a supportare l'unione dei requisiti di un intero dominio applicativo. Per ottenere questo obiettivo può essere opportuno usare *metodi agili* (si veda [Processi agili](#)) che permettano un continuo refactoring preservando l'integrità architettonale del framework e un collaudo continuo al variare dei requisiti supportati (per non compromettere l'integrità delle applicazioni). Nella progettazione e sviluppo di un framework i pattern (languages) svolgono un ruolo molto importante per individuare le alternative di progettazione che permettono di raggiungere specifiche proprietà, prima tra tutte la riusabilità, e per ridurre i costi dei cambiamenti minimizzando i loro effetti.

Framework-based development

La possibilità di disporre di un application framework modifica ulteriormente il processo di produzione del software in quanto lo sviluppo di un'applicazione nel dominio del framework avviene di fatto in modo molto più vincolato rispetto all'uso di una piattaforma general purpose: il framework, imponendo un preciso stile architettonale, ha di fatto già "risolto" un importante insieme delle forze che caratterizzano quel dominio, riducendo la complessità del problema agli occhi sia dell'analista sia del progettista, che possono ora meglio concentrarsi sulla "business logic" del problema.

Una parte rilevante della complessità della costruzione del software si traduce ora nella produzione/gestione dei diversi artefatti relativi ai punti di flessibilità (*variation points*) previsti dal framework per permettere l'adattamento ai requisiti della specifica applicazione. Questa complessità viene ridotta dall'uso di appositi tools; altri tools possono aiutare nella produzione di artefatti di deployment che possono essere assemblati, configurati e packaged per produrre *executables* da replicare, installare, registrare, etc. su piattaforme target quali applications servers, database servers, messaging systems, transaction monitors, web services, etc.

Sviluppo top-down

Lo sviluppo top-down nasce dalla convinzione che le proprietà di un sistema non siano riducibili alle proprietà delle parti constituenti (lo slogan può essere che *il tutto è più della somma delle parti*).

I suoi sostenitori insistono dunque nella necessità di caratterizzare nel modo più preciso possibile le proprietà del sistema nel suo complesso in modo che le parti non immediatamente riducibili agli elementi realizzativi concretamente disponibili possano essere progettate e costruite avendo sempre ben chiaro il quadro globale.

Progetti a crescita incrementale

L'uso dei pattern languages viene associato a un processo evolutivo, incrementale e ricorsivo di *piecemeal growth* che, partendo da una visione del "tutto", applica gradualmente i pattern per trasformare una prima situazione e/o architettura di base, in cui si manifesta un particolare problema, in una nuova situazione e/o architettura in cui il

problema è risolto, ripetendo il procedimento per i nuovi problemi che si possono porre nella nuova situazione.

Questo tipo di processo non è compositivo né a cascata ([POSA5] pg. 301) e condivide molti aspetti dei processi agili [CoHa04], in cui il sistema evolve attraverso forme intermedie stabili e logicamente integrate: ogni struttura è trasformata preservando la visione e le proprietà della struttura più ampia di cui prende il posto.

Nell'ottica ***piecemeal growth*** lo schema logico introdotto in Un percorso di riferimento può essere riletto attraverso una sequenza di domande-risposte.

1. Una volta compresi i requisiti ci si può chiedere: ***come organizzare le funzionalità richieste in insiemi coerenti in modo che ciascun insieme possa essere sviluppato e modificato in modo indipendente dagli altri?*** Una tipica risposta consiste nella definizione di un'***architettura logica di base*** organizzata in layer orizzontali applicando il pattern ***Layers*** (pg. 185).

Figura 1. Pattern

Layers

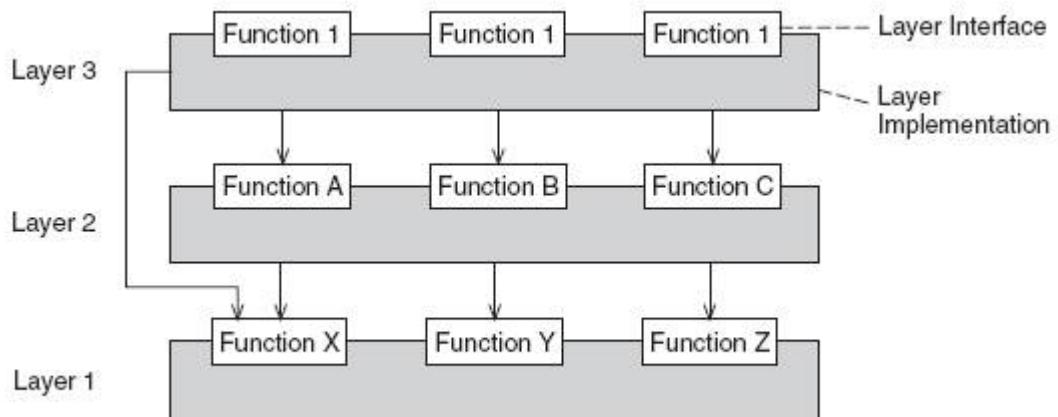
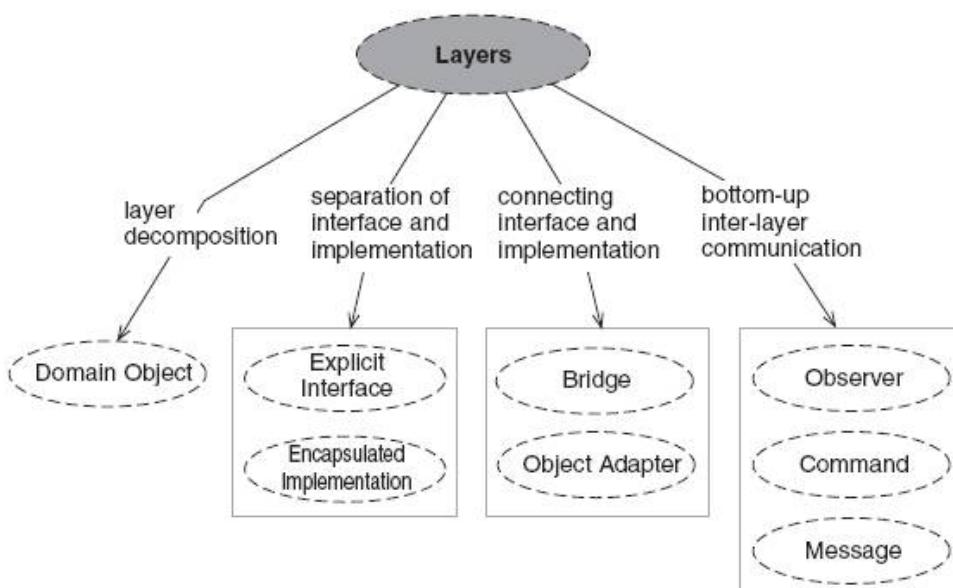


Figura 2. Relazioni del pattern Layers



E' frequente individuare i seguenti livelli orizzontali (POSA4 pg. 67):

- a. **Presentazione**: ha la responsabilità di contenere le interfacce di accesso alle operazioni del sistema e il supporto all'utente.
- b. **Processi di business**: ha la responsabilità di realizzare le funzionalità dell'applicazione.
- c. **Oggetti di business**: ha la responsabilità di definire e gestire le entità del dominio applicativo, sulle quali operano i processi di business del livello superiore.
- d. **Infrastruttura**: ha la responsabilità di realizzare funzionalità indipendenti dal dominio; da essa può dipendere il soddisfacimento di molti requisiti non funzionali.
- e. **Accesso**: ha la responsabilità di gestire gli accessi a supporti esterni al sistema, in primo luogo ai database.

Va ricordato che questa organizzazione è puramente logica e potrebbe ben rappresentare l'architettura logica complessiva del sistema finale; nulla vieta che l'architettura finale del sistema sia poi realizzata diversamente.

2. Subito dopo si può avanzare una seconda domanda di carattere generale: **come raffinare ciascun layer in parti cui assegnare specifiche responsabilità applicative?** Una tipica risposta consiste nell'applicare il pattern **Domain Model** per definire un insieme di parti auto-contenute ciascuna associata a precise responsabilità funzionali; queste parti corrispondono in pratica agli oggetti di business derivati/derivabili dall'analisi.

Figura 3. Pattern Domain Model

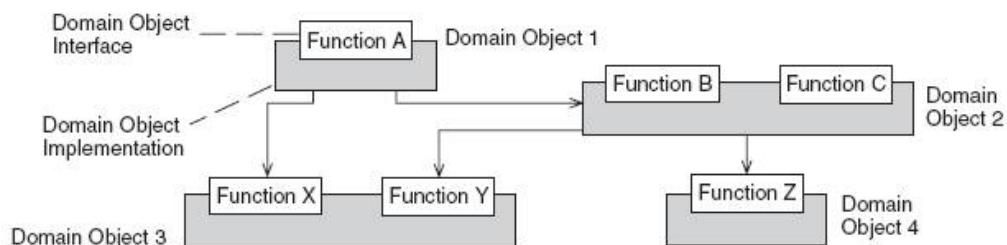
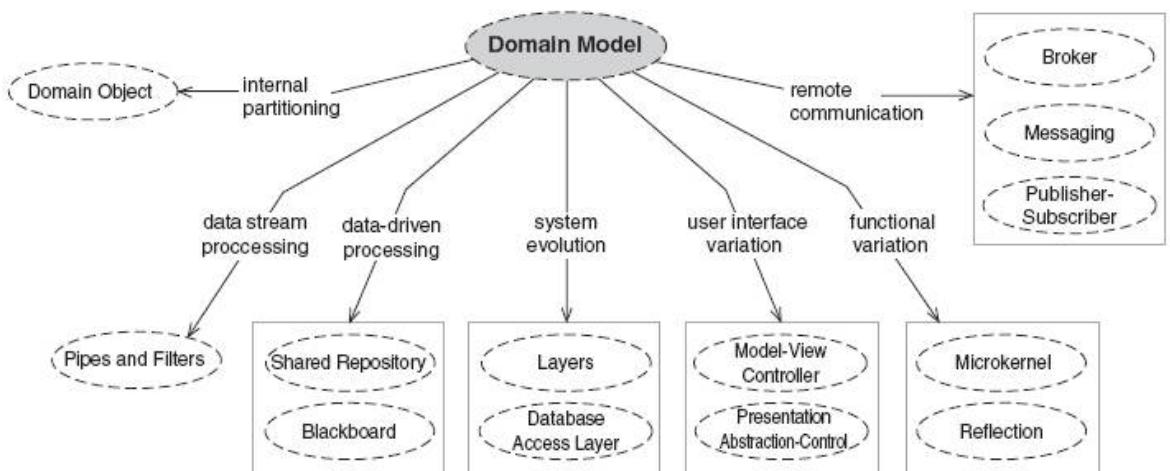


Figura 4. Relazioni del pattern Domain Model



Ciascuno dei layer orizzontali viene quindi articolato attraverso una decomposizione verticale in diversi **domain objects**.

3. L'obiettivo di definire un'architettura logica del problema astraendo da ogni dettaglio implementativo implica che si debba anche dare risposta ad un'ulteriore domanda: **come evitare che una parte debba dipendere in modo diretto dalla realizzazione di un'altra?**. La risposta consiste nel separare, per ogni layer e domain object, la specifica delle funzionalità offerte dalla loro realizzazione, adottando i pattern **Explicit Interface** (pg. 281) e **Encapsulated Implementation** (pg. 313).

Figura 5. Pattern Explicit Interface

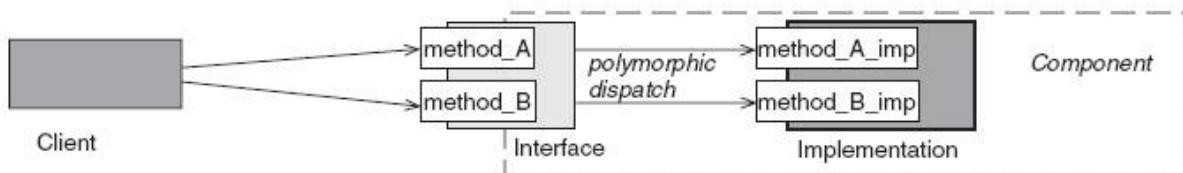


Figura 6. Pattern Encapsulated Implementation

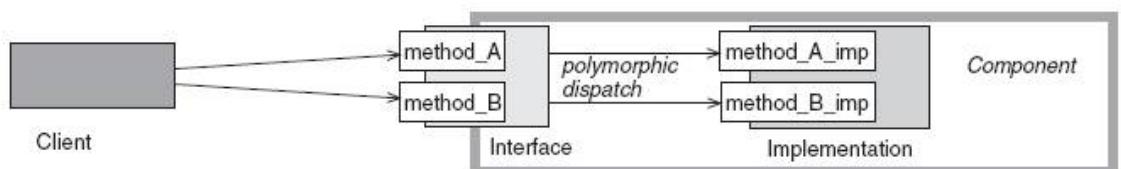
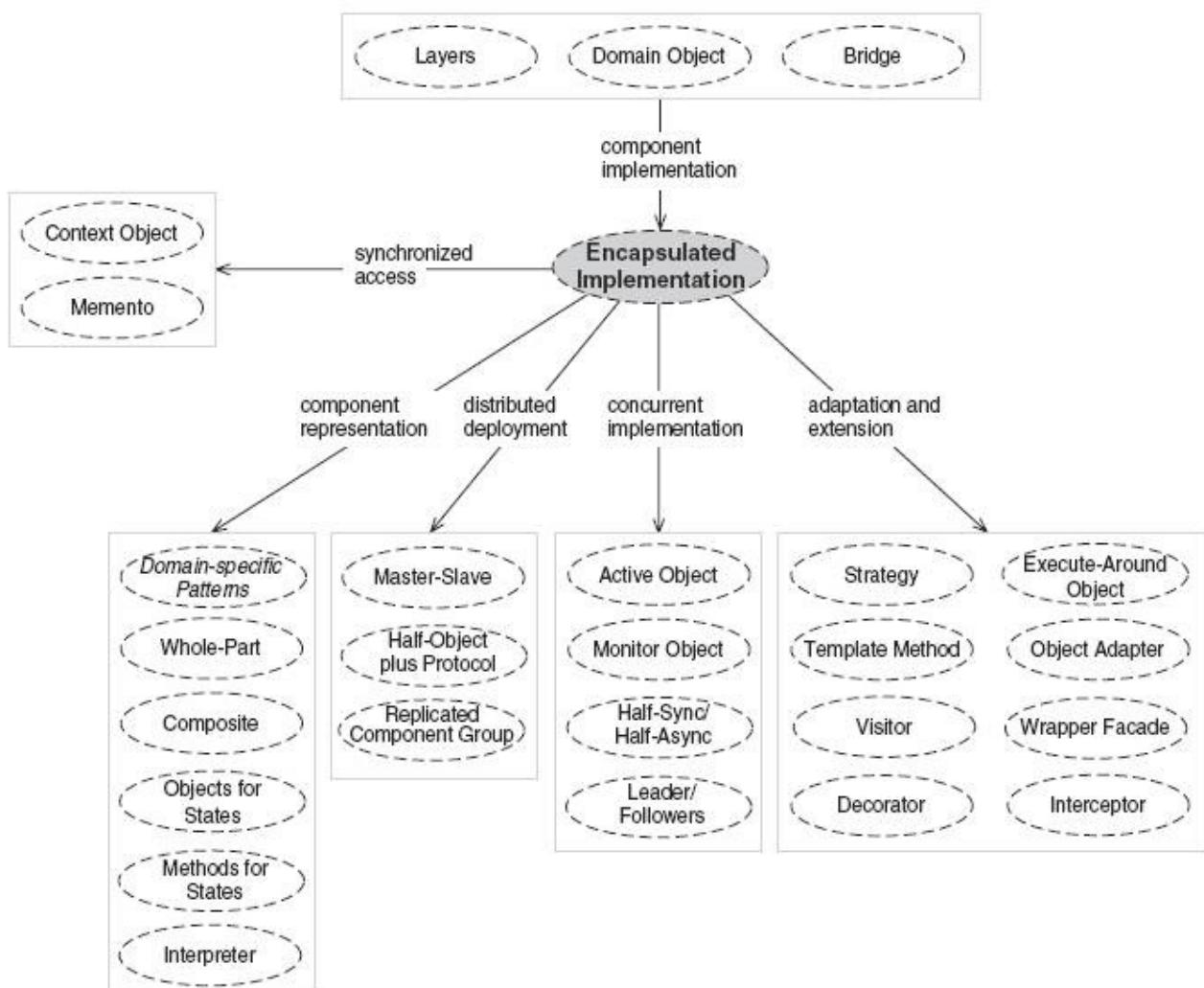
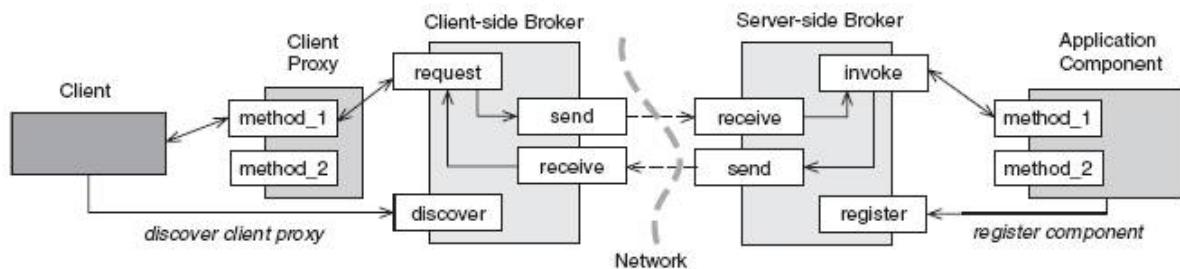


Figura 7. Relazioni Encapsulated Implementation



4. A livello di architettura logica è anche opportuno porsi il seguente problema: **come evitare che un domain object debba affrontare in modo diretto dettagli sulle modalità di comunicazione** con gli altri domain objects? Una tipica risposta è fare riferimento al pattern **Broker** (pg. 237).

Figura 8. Pattern Broker



Tutta la fase di progettazione può essere affrontata in modo sistematico cercando di trasformare l'architettura logica in una architettura di progetto seguendo ancora uno schema domanda-risposta. Una tipica domanda può essere ad esempio: **come evitare overhead e inefficienze senza distruggere il sistema logico a layer?** Questo caso è molto frequente in tutti i sistemi in cui sia necessario mantenere allineato un domain object del presentation layer (ad esempio una vista) con lo stato di un oggetto di un livello inferiore (ad esempio del business object layer). Una tipica risposta è applicare pattern che si basano sul principio della **inversione del controllo** quali **Model View Controller (MVC)** o **Presentation-Abstraction.Control (PAC)**.

Figura 9. Pattern MVC

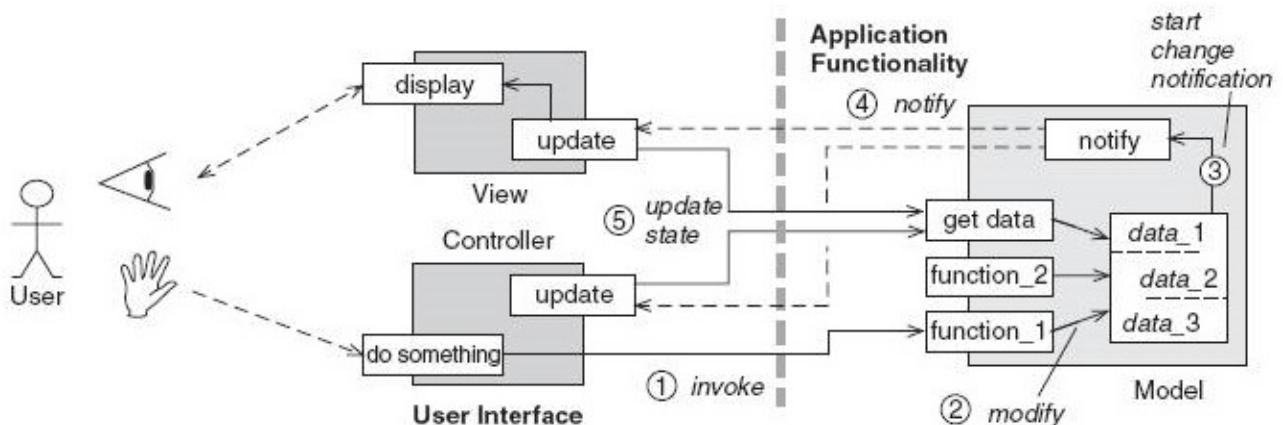
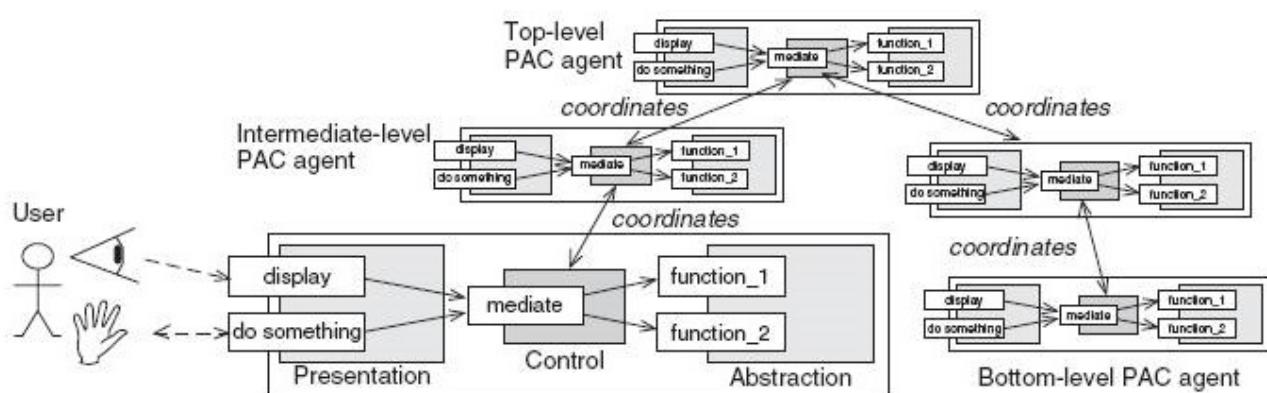


Figura 10. Pattern PAC



Implementazione

La realizzazione (implementazione) può essere agevolata dall'uso di piattaforme operative sviluppate da terze parti in forma di **framework** (si veda [Sviluppo basato su framework](#)) o di **product-line architectures**.

Il codice di implementazione dovrebbe essere il più possibile allineato con la soluzione scaturita dalla fase di progetto. Questo obiettivo viene facilmente raggiunto nel caso in cui sia possibile **generare il codice** in modo automatico dai modelli che rappresentano il progetto.

Nel caso in cui non sia possibile adottare approcci generativi, la **tracciabilità** deve essere mantenuta in modo esplicito dagli sviluppatori. La produzione del codice avviene comunque in un contesto vincolato dai piani di collaudo definiti nelle fasi precedenti.

Artefatti

Il principale artefatto è qui costituito dal codice, che specializza il modello di progetto con riferimento a uno specifico linguaggio di programmazione e a uno specifico ambiente operativo di supporto.

Collaudo

Il collaudo inizia non appena il progetto produce informazioni sufficienti a trasformare le specifiche parziali del piano di collaudo (si veda [Piano di collaudo](#) in una specifica di collaudo vera e propria e non appena è disponibile codice di implementazione).

Si dice **functional testing** il collaudo relativo alle funzionalità che si suppone il sistema debba compiere, mentre si dice **structural testing** il collaudo basato sulla struttura del codice.

Una classificazione:

- Unit Testing: testing single units of work
- Integration Testing: testing how different units of work interact
- Functional Testing: testing subsystems (usually on a boundary API)
- Stress/Load Testing: testing the system performance
- (User) Acceptance Testing: testing the system as a user

Un'altra classificazione:

- White box testing: testing with knowledge of the target source code
- Black box testing: testing on the target public API without knowledge of the target source code

Note:

- I tipi di test e i confini tra di essi sono spesso motivo di dibattito
- Gli unit test sono sempre a white box, gli altri tipi di test possono essere a black box
- In linea di principio, per realizzare uno unit test di un'unità di lavoro che dipende da altre unità di lavoro è necessario sostituire le unità di lavoro da cui la prima dipende con degli **Stub** o utilizzare i **Mock Objects** (altrimenti è necessario considerare il test almeno come un integration test)
- Un indicatore relativo al testing è il **Test Coverage**; in caso di white box testing il Test Coverage può essere dedotto dalla percentuale di righe di codice applicativo eseguite dalla suite di test; in caso di black box testing esso può essere dedotto dal numero di metodi eseguiti sulle unità testate

Myers [Myers04] denomina **failure** la situazione in cui il software opera in contrasto con le specifiche e **fault** l'elemento del software che causa una failure.

Un functional testing potrebbe dare risultato positivo anche se lo stato interno del sistema è erroneo. D'altra parte un piano di collaudo basato su testing strutturale può distogliere l'attenzione su cosa investigare. (il **code coverage** non sempre garantisce di raggiungere **functional coverage**). La **structural coverage** può essere usata come una misura di adeguatezza di collaudi funzionali.

Marick propone [Marick94] la seguente metodologia:

- I test funzionali sono generati dai requisiti o dalle specifiche e dal progetto, con l'intento di individuare le situazioni di fallimento.
- La copertura strutturale è esaminata solo dopo che i test funzionali sono stati tutti soddisfatti.
- Nel caso la coperatura strutturale sia da completare, il collaudatore non si lascia guidare dal sistema software, ma cerca di generare nuove situazioni a livello funzionale.

Organizzare il collaudo

Il collaudo è una fase molto impegnativa, il cui approfondimento richiede un testo specializzato. Come quadro di riferimento generale può essere utile tenere presenti i punti indicati nella tabella che segue.

Cosa si propone il collaudo	<ul style="list-style-type: none"> • Individuare la presenza di difetti in un tempo pianificato e a costi minimi. • Cercare di scoprire che i modi di un sistema non sono quanto è stato specificato debbano essere. La garanzia di assenza di errore richiede tecniche complementari quali ispezione del codice e prove formali di correttezza. • Lo scopo del testing del software è trovare failures così che, una volta che il software ha fallito un collaudo, si possano trovare ed eliminare i faults responsabili della failure.
Cosa collaudo	<ul style="list-style-type: none"> • Quello che costruisco: funzioni, moduli (classi), sottosistemi (package) più integrazione di sottosistemi, integrazione di sistema • System testing: sollecito con ingressi nel dominio dell'utenza • Unit testing: sollecito le parti per individuare le responsabilità delle failures e correggerle.
Come collaudo	<ul style="list-style-type: none"> • Scelgo tra black box, white box, gray box. • Distinguo tra testing di unità, di oggetti e di sottosistemi • Partiziono gli ingressi in classi di equivalenza • Determino le risposte attese o i test oracle. Indago le boundary conditions tra le classi di equivalenza • Mi assicuro che tutti gli statement siano percorsi • Mi assicuro che tutte le alternative (branch) siano percorse (decision coverage) • Genero sequenze di ingressi utili a sollecitare sottosistemi • Genero input casuali
Quando collaudo	<ul style="list-style-type: none"> • Appena possibile, in modo continuo. Per agire il più precocemente possibile posso usare stubs cioè componenti chiamati vuoti, oppure che simulano le risposte anche chiedendole all'utente.

Come pianifico	<ul style="list-style-type: none"> • Decido la filosofia del testing • Decido la forma di documentazione • Determino la estensione del testing • Decido quando e come acquisire i dati di input • Stimo le risorse richieste
----------------	---

Utili sono anche le raccomandazioni di Humphrey [Humphrey89] per quanto riguarda il collaudo delle funzioni.

1. Verifica le operazioni con valori normali degli argomenti (black box)
2. Verifica le operazioni con valori degli argomenti ai limiti (black box)
3. Verifica le operazioni con valori degli argomenti fuori dal range (black box)
4. Assicurare che tutti le istruzioni siano eseguite (statement coverage)
5. Controllare tutti i percorsi, su entrambe le linee decisionali (branch coverage)
6. Controllare l'uso di tutti gli oggetti chiamati
7. Verificare la gestione di tutte le strutture di dati
8. Verificare la gestione di tutti i files
9. Controllare la normale terminazione di tutti i cicli (in cooperazione con *prove di correttezza*)
10. Controllare la terminazione anomala di tutti i cicli
11. Controllare la normale terminazione di tutte le ricorsioni
12. Verificare la gestione delle condizioni di errore
13. Controllare i vincoli di tempo e le sincronizzazioni
14. Verificare tutte le dipendenze dall'hardware

Deployment

La costruire di risorse (file eseguibili, componenti software, etc) da distribuire/installare.dipende fortemente dalla piattaforma operativa utilizzata.

Artefatti

Gli artefatti sono rappresentati dal packaging per la distribuzione e installazione del sistema.

Maintenance

La gestione del prodotto dovrebbe risultare agevolata dalla disponibilità di artefatti capaci di rendere conto delle parti responsabili di una funzionalità da modificare e delle relazioni si queste parti con il resto del sistema.

Artefatti

Gli artefatti sono rappresentati dai supporti disponibili per la modifica (dinamica) della configurazione del sistema.

Esempio di sviluppo model based

Una ditta che produce sistemi di automazione industriale ha incaricato la nostra software house di progettare e costruire il software relativo alla centrale di controllo di un impianto industriale. Una caratteristica importante della centrale è che il suo

funzionamento è legato (in un modo che qui non interessa approfondire) ad una espressione aritmetica formata in accordo alle regole grammaticali che seguono:

E ::= T		E + T		E - T
T ::= F		T * F		T / F
F ::= N		(E)		S
N ::= D		N D		
S ::= sN				
D ::= 0		1		2
		3		4
		5		6
		7		8
		9		

E,T,F,N,D denotano simboli non terminali, E denota lo scopo della grammatica e + - * / () 1 2 3

4 5 6 7 8 9 s denotano simboli terminali. I simboli + - * / rappresentano gli usuali operatori di somma, sottrazione, prodotto, divisione e quoziente.

Il linguaggio generato dal metasimbolo N denota la rappresentazione in base 10 di un numero naturale. Una frase generata dal metasimbolo S denota una "variabile" che rappresenta un valore numerico (reale) dato da un particolare sensore innestato sull'impianto. Ad esempio, in un dato momento del suo funzionamento, la centrale potrebbe dipendere dall'espressione che segue:

s3 * (s1 - s2) / 10 * 3

in cui s1, s3, s2 sono variabili relative a tre diversi sensori. Al mutare del valore di un sensore, il valore della espressione cambia e il risultato influenza in qualche modo il comportamento della centrale di controllo.

La software house incarica quindi il nostro gruppo di lavoro di impostare un (sotto)sistema software per il riconoscimento e la valutazione di espressioni espresse dalla grammatica data.

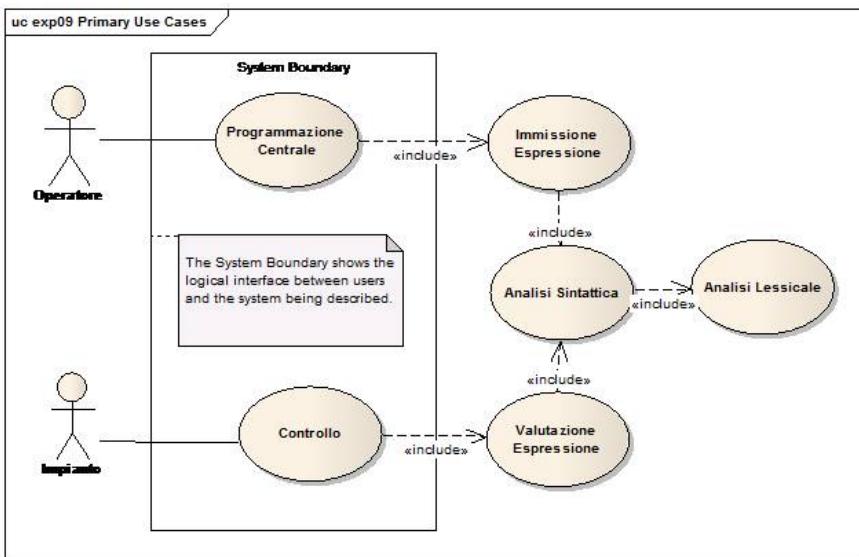
Espressioni: analisi dei requisiti

La fase di analisi dei requisiti (si veda [Analisi dei requisiti](#)) richiede forti interazioni con il committente al fine di chiarire il significato dei vocaboli usati e quello che si vuole che il sistema faccia.

Use cases

Dalla interazione con il committente si è capito che questi desidera che siano realizzati sia un analizzatore lessicale ([lexer](#)) sia un analizzatore sintattico ([parser](#)) relativi al linguaggio generato dalla grammatica. L'analisi lessicale l'analisi sintattica di una frase costituiscono quindi due requisiti funzionali ([C-requirements](#), si veda [Requisiti](#)) centrali, che il seguente modello dei [casi d'uso](#) (si veda [Casi d'uso](#)) colloca all'interno di altre funzionalità desiderate.

Figura 1. Casi d'uso



Glossario

Il significato dei principali termini usati nei requisiti è, in prima battuta, espresso da seguente glossario:

Termino	Significato
Espressione aritmetica	Un modo per denotare un numero (reale) attraverso una stringa in cui compaiono rappresentazione in base 10 di numeri naturali, simboli di operazione e variabili (che denotano valori reali). La stringa deve essere scritta in accordo alle regole grammaticali date.
Scopo della grammatica	Simbolo non terminale scelto come simbolo iniziale di riscrittura per la generazione di rappresentazioni sintatticamente corrette di una espressione.
Simboli non terminali	Detti anche metasimboli o variabili sintattiche , sono simboli che possono essere riscritti.
Simboli terminali	Simboli che fanno parte del linguaggio descritto dalla grammatica: non possono essere riscritti. Formano un insieme disgiunto rispetto ai simboli non terminali.
Token	Elemento del linguaggio prodotto dall'analizzatore lessicale.
Analizzatore lessicale (lexer)	Automa che analizza una stringa con riferimento ad una specifica grammatica e individua elementi lessicali (terminali semplici o composti) del linguaggio quali numeri, nomi di variabili ed operatori restituendoli come oggetti denominati token .
Analizzatore sintattico (parser)	Automa che analizza una sequenza di token con riferimento ad una specifica grammatica e decide se tale sequenza è ben formata (sintatticamente). In caso positivo restituisce una rappresentazione interna della frase relativa alla sequenza in forma di albero (binario).

Valutazione	Attività che porta alla individuazione di un valore numerico equivalente alla espressione, cioè sostituibile ad essa.
--------------------	---

A questo possiamo aggiungere il significato tradizionalmente attribuito alle parti di una espressione generate dai diversi simboli terminali:

Nome	Significato
Espressione	La frase generata dal metasimbolo E.
Termine	La frase generata dal metasimbolo T.
Fattore	La frase generata dal metasimbolo F.
Variabile	La frase generata dal metasimbolo S.

Scenari

Limitandoci al momento alle sole funzionalità relative al sottoproblema affidatoci, quello che si vuole che il sistema faccia viene descritto dagli scenari che seguono.

(si veda [Scenari](#))

Immissione espressione

Campo	Descrizione
ID (Nome)	UC1 - Immisione frase
Descrizione	Fornire una nuova espressione di controllo alla centrale.
Attori	Operatore.
Precondizioni	La centrale non ha segnalato malfunzionamenti
Scenario principale	L'operatore utilizza un dispositivo di ingresso per scrivere una frase e per inviarla alla centrale. L'operatore quindi attende dal sistema un messaggio che segnala il successo o il fallimento dell'operazione, in relazione all'esito dell'analisi sintattica della frase. Nel caso di esito negativo l'operatore riscrive la frase cercando di correggere gli errori.
Scenari alternativi	L'operatore non riceve alcun messaggio di risposta entro un dato intervallo di tempo (ad esempio 60 secondi); in tal caso l'operatore ripete l'operazione di invio della frase.

Postcondizioni	Nel caso la frase inserita sia corretta, la centrale di controllo ha in memoria la nuova espressione.
-----------------------	---

Analisi sintattica

Campo	Descrizione
ID (Nome)	UC2 - Analisi sintattica
Descrizione	Controlla che una frase sia scritta in accordo alle regole grammaticali date; in caso positivo produce una rappresentazione interna ad albero.
Attori	Operatore.
Precondizioni	E' disponibile una frase da analizzare o un parte di essa.
Scenario principale	Disponendo della frase completa, l'analizzatore invoca un lexer per ottenere la sequenza di token corrispondente alla frase data; quindi esso verifica che la sequenza sia in accordo alle regole grammaticali. In caso positivo esso genera una rappresentazione interna ad albero (binario). In caso negativo segnala un errore.
Scenari alternativi	L'analizzatore riceve la frase via rete; esso potrebbe quindi disporre solo di un prefisso della frase da analizzare. In tal caso l'analizzatore può decidere di iniziare comunque l'analisi, col vantaggio di poter individuare e segnalare eventuali errori non appena questi si presentano.
Postcondizioni	Nel caso l'analizzatore lessicale segnali un errore, la sequenza di token non viene scandita e il segnale di errore viene propagato. Nel caso la frase di ingresso sia corretta, la sequenza di token risulta completamente consumata. Nel caso la frase di ingresso sia sintatticamente errata, la sequenza risulta consumata fino al primo errore rilevato.

Analisi lessicale

Campo	Descrizione
ID (Nome)	UC3 - Analisi lessicale
Descrizione	Controlla che gli elementi lessicali di una frase siano scritti in accordo alle regole date; in caso positivo produce una sequenza di token. In caso negativo segnala un errore.
Attori	Operatore

Precondizioni	E' disponibile una frase da analizzare o un parte di essa.
Scenario principale	Disponendo della frase completa, l'analizzatore individua gli elementi lessicali. Nel caso la frase sia lessicalmente corretta, esso genera una sequenza di elementi in forma di token. In caso contrario segnala un errore.
Scenari alternativi	L'analizzatore riceve la frase via rete; esso potrebbe quindi disporre solo di un prefisso della frase da analizzare. In tal caso l'analizzatore può decidere di iniziare comunque l'analisi, col vantaggio di poter individuare e segnalare eventuali errori non appena questi si presentano.
Postcondizioni	Nel caso la frase di ingresso sia corretta, essa risulta completamente scandita. Nel caso la frase di ingresso sia errata, essa risulta scandita fino al primo errore rilevato.

Valutazione

Campo	Descrizione
ID (Nome)	UC4 - Valutazione
Descrizione	Dato l'albero (binario) che fornisce la rappresentazione interna di una espressione, determina il valore numerico dell'espressione in base alle regole semantiche del linguaggio.
Attori	Controllo
Precondizioni	L'albero (binario) di ingresso non è vuoto.
Scenario principale	<p>Il valutatore vista l'albero agendo come segue:</p> <ul style="list-style-type: none"> • nel caso il nodo sia un operando ne legge il valore; • nel caso il nodo sia un operatore binario, applica all'operatore gli operandi ottenuti valutando i suoi sottoalberi di destra e di sinistra. <p>Al termine del processo restituisce il valore dell'espressione costituito da un numero reale oppure restituisce un segnale di errore nel caso di operazioni semanticamente scorrette (ad esempio una divisione per 0).</p>
Scenari alternativi	True
Postcondizioni	True

Dagli scenari si deduce che il committente propende per una **analisi top-down** di una frase e l'analista del problema dovrà considerare con attenzione il modo con cui la frase viene progressivamente analizzata.

Espressioni: modellazione del dominio

Il dominio del problema è quello dei numeri e dei modi per denotarli, sia in notazione posizionale con riferimento a una base, sia attraverso espressioni. Gli elementi fondamentali del dominio sono quindi costituiti dalle espressioni e dagli elementi lessicali (quali operatori, numeri, parentesi) con cui queste si esprimono.

La modellazione che segue avviene tenendo conto delle tre punti di vista citati in precedenza (si veda [Dimensioni](#)). Poiché il modello del dominio riguarda entità entità assimilabili a **dati**, i modelli relativi alla interazione e al comportamento non saranno espressi in UML in quanto riconducibili a pattern tipici dei [tipi dato \(astratto\)](#).

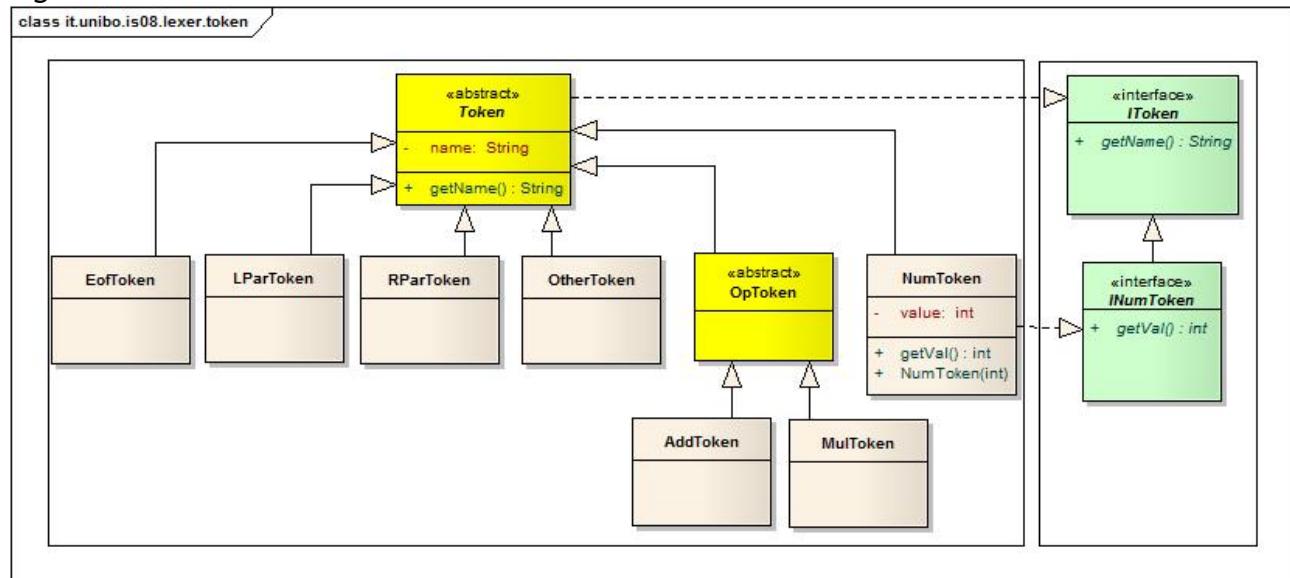
Si privilegerà invece l'idea di impostare **piani di collaudo** (si veda [Piano di collaudo](#)) con l'intento di introdurre elementi utili a comprendere la **semantica** delle entità e specificarne nel modo più chiaro possibile il comportamento atteso.

Modello dei token

Token: struttura

I **token** possono essere rappresentati attraverso il seguente modello UML:

Figura 1. Modello dei token



Il modello introduce classi per rappresentare gli elementi lessicali relativi alla grammatica data:

- **LParToken**: rappresenta una parentesi aperta.
- **RParToken**: rappresenta una parentesi chiusa.
- **OpToken**: rappresenta un operatore binario.
- **AddToken**: rappresenta un operatore di somma o sottrazione.
- **MulToken**: rappresenta un operatore di prodotto o divisione.
- **NumToken**: rappresenta un numero naturale.

Ogni token è una entità atomica dotata dell'attributo **name** che verrà usato per distinguere i token di una stessa classe. Ad esempio il simbolo **+** può essere rappresentato da

un `AddToken` con `name` uguale a `"+"`. I token che rappresentano numeri naturali (`NumToken`) posseggono anche l'attributo `value` che ne denota il valore.

Oltre agli elementi lessicali relativi alla grammatica data il modello introduce anche

- `EofToken`: rappresenta la fine della frase.
- `OtherToken`: rappresenta un elemento non riconosciuto.

Ad esempio la frase

`27 + 3 ?!- 5)`

viene rappresentata dalla seguente sequenza di token:

`NumToken AddToken NumToken OtherToken OtherToken AddToken NumToken RPatToken`

Un token rappresenta un oggetto nel senso più classico del termine: è cioè un ente che nasconde la propria rappresentazione interna e che può essere utilizzato avvelendosi solo di operazioni definite in una opportuna interfaccia (`IToken` o `INumToken`). Questo implica che le dimensioni interazione e comportamento sono riconducibili all'idea di oggetto come ente passivo e dotato di stato (modificabile o meno).

Token: interazione

La interazione con un token avviene invocando una operazione resa disponibile dalla interfaccia. Il meccanismo di supporto e la chiamata di procedura/funzione, che prevede un trasferimento di controllo e di eventuali argomenti dal chiamante e la eventuale restituzione di un valore.

Token: comportamento

Un token è un oggetto atomico il cui stato non è modificabile. Il comportamento di ogni operazione di un token è quindi riconducibile al comportamento di una funzione priva di effetti collaterali che restituisce il valore di una specifica proprietà. Spesso a queste funzioni viene dato un nome del tipo `getXXX` ove `XXX` è il nome della proprietà.

E' anche possibile aggiungere una indicazione relativa al modo con cui costruire un token.

Token: costruzione

Il costruttore di una entità atomica quale un token è una operazione che ha tanti argomenti quante le proprietà che il token deve assumere. Ad esempio la frase `Java`

`IToken op0 = new MulToken("/");`
esprime la dichiarazione di una variabile `op0` come simbolo atto a denotare un oggetto di tipo `IToken` e la inizializzazione di tale variabile al riferimento ad un oggetto di classe `MulToken` che rappresenta un operatore di divisione.

Al momento escludiamo dal modello la presenza di operazioni del tipo `setXXX` che sono spesso introdotte per inizializzare il valore di una proprietà di un oggetto creato con un costruttore privo di argomenti. Operazioni di questo tipo sono introdotte in modo sistematico accanto a funzioni `getXXX`, quando si vogliono applicare tecniche di inversione di controllo (si veda [Inversione del controllo](#)) per la risoluzione di dipendenze tra oggetti. La loro presenza è però molto pericolosa in quanto rende dinamicamente modificabile un oggetto che dovrebbe rappresentare una costante.

Modello delle espressioni

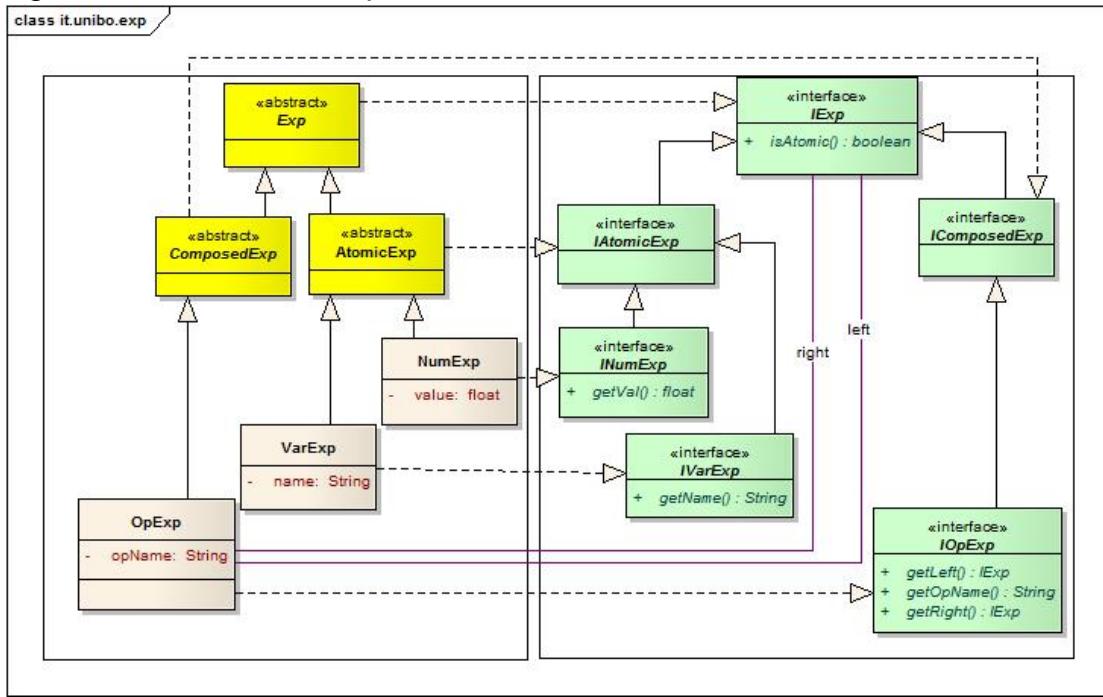
Espressioni: struttura

Un modello logico delle espressioni può essere impostato attraverso una accurata analisi delle regole grammaticali date. Queste infatti possono produrre:

- espressioni atomiche formate da singoli fattori (**F**) costituiti da numeri o da variabili; esempi di espressione di questo tipo sono "237", "(21)", "((413))", "s12", "(s313)".
- espressioni composte formate da più fattori e da operatori binari; un esempio di espressione di questo tipo è "2 + 3*(5 - 4)".

Su queste basi il modello delle espressioni può essere definito facendo riferimento al xref href="../pattern/Composite.dita"/>:

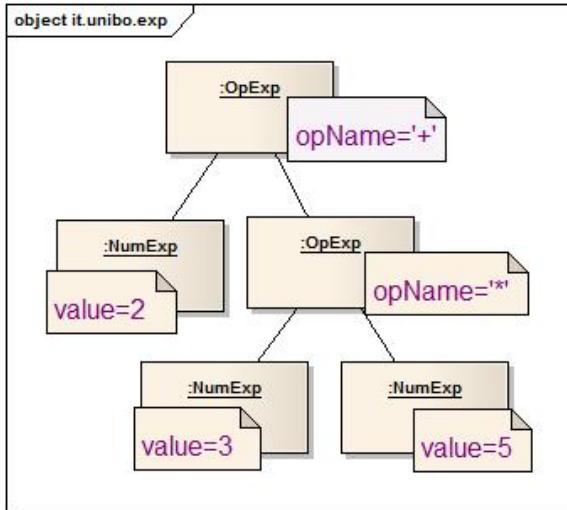
Figura 1. Modello delle espressioni



La classe (astratta) **Exp** rappresenta l'entità alla radice del pattern **Composite**; la classe **NumExp** rappresenta una foglia, mentre la classe **OpExp** rappresenta un elemento composto da due entità selezionabili attraverso le associazioni **left** e **right**.

Il modello delle espressioni costituisce un punto di riferimento per la costruzione di una rappresentazione ad oggetti delle espressioni da parte di un analizzatore sintattico (parser). Poiché la rappresentazione interna prodotta dal parser è detta **Abstract Parse Tree** (APT), gli elementi (nodi) di questo albero saranno istanze delle classi definite dal modello. Ad esempio all'espressione $2 + 3 * 5$ corrisponderà il seguente APT:

Figura 2. APT di $2+3*5$



Si noti che la classe `NumExp` è definita in modo da denotare un valore di tipo `float` per tenere conto del tipo dei valori denotabili da una espressione e non solo del tipo di valori (interi naturali) che possono essere direttamente espressi in essa. In altre parole, il risultato dalla valutazione di una espressione potrà essere un oggetto di tipo `NumExp`.

Una espressione rappresenta un oggetto nel senso più classico del termine: è cioè un ente che nasconde la propria rappresentazione interna e che può essere utilizzato avvolendosi solo di operazioni definite in una opportuna interfaccia (`IExpn` o `INumExp`, etc.). Questo implica che le dimensioni interazione e comportamento sono riconducibili all'idea di oggetto come ente passivo e dotato di stato (modificabile o meno).

Espressioni: interazione

La interazione con una espressione avviene invocando una operazione resa disponibile dalla interfaccia. Il meccanismo di supporto e la chiamata di procedura/funzione, che prevede un trasferimento di controllo e di eventuali argomenti dal chiamante e la eventuale restituzione di un valore.

Espressioni: comportamento

Una espressione è un oggetto atomico o composto il cui stato non è modificabile. Il comportamento di ogni operazione di un token è quindi riconducibile al comportamento di una funzione priva di effetti collaterali. Nel caso di espressioni composte alcune funzioni possono fungere da selettori ciascuno dei quali restituisce il valore relativo ad una parte costituente. Spesso a queste funzioni viene dato un nome del tipo `getXXX` ove `XXX` è il nome della proprietà o della parte.

E' anche possibile aggiungere una indicazione relativa al modo con cui costruire una espressione.

Espressioni: costruzione

Il costruttore di una espressione atomica è una operazione che ha tanti argomenti quante le proprietà che il token deve assumere. Il costruttore di una espressione composta è una operazione che ha tanti argomenti quante le parti costituenti. Compito del costruttore è "assemblare" le parti date in modo da formare un "organismo" capace di assicurare il soddisfacimento di proprietà globali esprimibili come *invarianti* (di classe).

Le operazioni di costruzione relative alle classi concrete possono essere duqne impostate come segue:

NumExp	Per costruire una espressione atomica NumExp basta disporre di un valore intero.
VarExp	Per costruire una espressione atomica VarExp occorre disporre di una stringa che denota il nome della variabile.
OpExp	Per costruire una espressione composta OpExp occorre disporre di una tripla di valori: una stringa che denota l'operatore e due oggetti di tipo IExp che denotano gli operandi.

Ad esempio la frase **Java**

`IExp e1 = new OpExp("/", new NumExp(3), new NumExp(2));`
esprime la dichiarazione di una variabile **e1** come simbolo atto a denotare un oggetto di tipo **IExp** e la inizializzazione di tale variabile al riferimento ad un oggetto di classe **OpExp** che rappresenta una operazione di divisione con operandi **3** e **2**.

Al momento escludiamo dal modello la presenza di operazioni del tipo **setXXX** che sono spesso introdotte per inizializzare il valore di una proprietà di un oggetto creato con un costruttore privo di argomenti. Operazioni di questo tipo sono introdotte in modo sistematico accanto a funzioni **getXXX**, quando si vogliono applicare tecniche di inversione di controllo (si veda [Inversione del controllo](#)) per la risoluzione di dipendenze tra oggetti. La loro presenza è però molto pericolosa in quanto rende dinamicamente modificabile un oggetto che dovrebbe rappresentare una costante.

Espressioni: piano di collaudo

Per meglio definire il comportamento atteso di token ed espressioni, impostiamo un insieme di piano di collaudo (si veda [Piano di collaudo](#)) facendo riferimento ad alcuni criteri di carattere generale relativi ai tipi di dato (astratto).

Piano di collaudo dei token

Un token è una entità atomica non modificabile; un piano di collaudo può essere impostato facendo riferimento al seguente criterio:

i predicati di una entità atomica non modificabile devono restituire valori coerenti con le proprietà attribuite all'entità al momento della costruzione.

Di qui e da quanto detto nel [Modello dei token](#) risulta immediato impostare operazioni di collaudo come le seguenti:

```
public final void testOpToken(){
    IToken fixture = new MultToken("*");
    assertEquals("testOpToken", fixture.getName(), "*");
}

public final void testNumToken(){
    IToken fixture = new NumToken(12);
    assertTrue("testNumToken", fixture.getVal()==12 );
}
```

Piano di collaudo delle espressioni

Una espressione ([Exp](#)) è una entità atomica o composta non modificabile; il piano di collaudo può essere impostato con riferimento al criterio già introdotto per i token più il seguente:

i selettori di una entità composta devono restituire valori coerenti con le parti specificate al momento della costruzione. Di qui e da quanto detto nel [Modello delle espressioni](#) risulta immediato impostare operazioni di collaudo come le seguenti:

```
public class OpExpTest extends junit.framework.TestCase {  
    private IComposedExp fixture;  
    ...  
    protected void setUp() throws Exception{  
        super.setUp();  
        fixture = new OpExp( "+", new NumExp(10), new NumExp(20) );  
    }  
  
    public final void testGetLeft(){  
        assertTrue("testGetLeft", fixture.getLeft().isAtomic());  
    }  
  
    public final void testGetOpName(){  
        assertTrue("testGetOpName", fixture.getOpName().equals("+"));  
    }  
  
    public final void testGetRight(){  
        assertTrue("testGetRight", fixture.getRight().isAtomic() &&  
                  ((INumExp)fixture.getRight()).getVal == 20.0 );  
    }  
}
```

Il codice di collaudo è opportuno venga incluso in un package distinto da quello dell'applicazione; si può pensare di introdurre a questo punto la seguente suite [JUnit](#):

```
import junit.framework.Test;  
import junit.framework.TestSuite;  
import junit.textui.TestRunner;  
  
public class ExpAllTest extends TestSuite{  
  
    public ExpAllTest(String name) {  
        super(name);  
    }  
  
    public static Test suite() {  
        TestSuite suite = new ExpAllTest("Expr Tests");  
        suite.addTestSuite(AtomicExpTest.class);  
        suite.addTestSuite(OpExpTest.class);  
        suite.addTestSuite(ExpTest.class);  
        return suite;  
    }  
  
    public static void main(String[] args) {  
        TestRunner.run(suite());  
    }  
}
```

In questa suite è stato anche introdotto un main che permette l'attivazione del collaudo come una normale applicazione Java sotto il controllo della utility [junit.textui.TestRunner](#) che può essere anche lanciata (posizionandosi nella directory del progetto che contiene la directory [bin](#) del bytecode) attraverso un comando del tipo:

```
java -cp ./bin;C:/repo/junit/junit/3.8.1/junit-3.8.1.jar
```

```
junit.swingui.TestRunner
java -cp ./bin;C:/repo/junit/junit/3.8.1/junit-3.8.1.jar
      junit.textui.TestRunner it.unibo.exp.exp.test.ExpAllTest
```

Espressioni: sintassi astratta

Il modello delle espressioni sviluppato fino a questo punto fornisce la sintassi astratta delle espressioni generate dalla grammatica data, che può essere utilizzata per costruire applicazioni anche in assenza di un parser e di un lexer. Ad esempio l'espressione:

2 * 3 - (4 + 1)

può essere costruita come segue:

```
IExp e1 = new OpExp( "*", new NumExp(2), new NumExp(3) );
IExp e2 = new OpExp( "+", new NumExp(4), new NumExp(1) );
IExp exp = new OpExp( "-", e1, e2 );
```

Sulla base delle interfacce definite è anche possibile impostare algoritmi di riscrittura e di valutazione delle espressioni senza averne completato l'implementazione. Le sezioni che seguono presenteranno qualche esempio in proposito.

Riscrittura in forma polacca

Un'espressione come 2 * 3 - (4 + 1) è scritta in **forma infissa**, cioè in una sintassi concreta in cui gli operatori binari compaiono tra i due rispettivi operandi. Un'espressione può tuttavia essere scritta anche in forma (polacca) prefissa, in cui gli operatori precedono gli operandi (- 2 3 * 4 1 +) o (polacca) postfissa, in cui gli operatori seguono gli operandi (2 3 * 4 1 + -).

Data un'espressione in forma interna ad albero binario, la riscrittura in forma polacca può essere effettuata impostando una visita all'albero stesso; ad esempio:

```
public String visitPostfissa(IExp exp){
    if( exp.isAtomic() ) return ""+((IAtomicExp)exp).getVal();
    else {
        String ls = visitPostfissa(
            ((IComposedExp)exp).getLeft() );
        String rs = visitPostfissa(
            ((IComposedExp)exp).getRight() );
        return ls + " " + " " + rs + " "+
            ((IComposedExp)exp).getOpName();
    }
}
```

Allo stesso modo si può impostare la valutazione di un'espressione:

```
public int eval(IExp exp){
int v=0;
    if( exp.isAtomic() ) return ((IAtomicExp)exp).getVal();
    else {
        int v1 = eval( ((IComposedExp)exp).getLeft() );
        int v2 = eval( ((IComposedExp)exp).getRight() );
        char op =
            ((IComposedExp)exp).getOpName().charAt(0);
        switch( op ){
            case '+' : v = v1 + v2;break;
            case '-' : v = v1 - v2;break;
            case '*' : v = v1 * v2;break;
            case '/' : v = v1 / v2;break;
        }
        return v;
    }
}
```

```
}
```

Il pattern Visitor

Osservando il codice precedente vediamo che siamo chiamati ad eseguire operazioni specifiche in funzione del tipo di elemento che forma un oggetto di tipo `Exp`. Per evitare l'uso del controllo esplicito di tipo e del type-casting sarebbe necessario introdurre nuove operazioni nelle classi `NumExp` e `OpExp` per supportare le esigenze di visita e di valutazione. D'altra parte le necessità che possono sorgere da parte di chi utilizza esprssioni possono essere molteplici; ci si chiede quindi se sia possibile estendere le operazioni relative alla gerarchia di classi di radice `Exp` senza dover modificare ogni volta l'insieme di operazioni definite da ciascuna classe.

La risposta a questa domanda è fornita dal xref href="..//pattern/Visitor.dita"/> che realizza l'idea di **double dispatch** per cui l'operazione eseguita da un oggetto dipende non solo dal nome della richiesta e dal tipo dell'oggetto ma anche dal tipo di un secondo oggetto. Questo secondo oggetto è detto **visitor** ed incapsula la logica dell'operazione da eseguire.

Il meccanismo è il seguente: ogni classe della gerarchia definisce una operazione (di nome `accept`) che riceve come argomento di ingresso un visitor; tale operazione delega a questo visitor la responsabilità di eseguire un'operazione (di nome standard `visit`) sull'oggetto corrente trasferito come argomento secondo il seguente schema di comportamento:

```
public void accept( Visitor v){  
    v.visit(this);
```

Nel caso del nostro modello delle espressioni un **Visitor** potrebbe realizzare la seguente interfaccia:

```
public interface IExpVisitor {  
    public void visit( IAtomicExp e );  
    public void visit( IComposedExp e );  
    public Object doJob();  
}
```

Il metodo `dojob` restituisce l'oggetto che rappresenta il risultato dell'applicazione dell'operazione realizzata dal visitor. Il modello di dominio deve essere esteso come segue:

```
public interface IExp {  
    public void accept(IExpVisitor e);  
}
```

Su queste basi possiamo pianificare il seguente piano di collaudo:

```
private IExp fixture ;  
protected void setUp() {  
//        String frase = "2 * 3 - (4 + 1 )";  
    IExp e1 = new OpExp( "*",  
                        new NumExp(2), new NumExp(3) );  
    IExp e2 = new OpExp( "+",  
                        new NumExp(4), new NumExp(1) );  
    fixture = new OpExp( "-", e1, e2 );  
} //setUp
```

```

public void testPolacca() {
    IExpVisitor visitor = new RewriteVisitor();
    String resultExpected = "2 3 * 4 1 + -";
    String result = visitor.doJob(fixture).toString();
    assertTrue("testPolacca", result.equals(resultExpected));
}//testPolacca

public void testEval() {
    IExpVisitor visitor = new EvalVisitor();
    String resultExpected = "1";
    fixture.accept(visitor);
    String result = visitor.doJob(fixture).toString();
    assertTrue("testEval", result.equals(resultExpected));
}//testEval

```

Il visitor che realizza la riscrittura in forma polacca postfissa può essere definito come segue:

```

package it.unibo.exp.expVisitor;
import java.util.Stack;

public class RewriteVisitor implements IExpVisitor{
Stack<String> stack = new Stack<String>();
    public void visit(IAtomicExp e) {
        stack.push(" "+e.getVal());
    }

    public void visit(IComposedExp e) {
        e.getLeft().accept(this);
        String leftSon = (String) stack.pop();
        e.getRight().accept(this);
        String rightSon = (String) stack.pop();
        stack.push(
            leftSon+ " " + rightSon + " " + e.getOpName());
    }

    public String doJob(IExp e){
        e.accept(this);
        if( !stack.empty()) return stack.pop();
        else return null;
    }
}

```

La definizione del visitor che effettua la valutazione dell'espressione viene lasciata come esercizio al lettore.

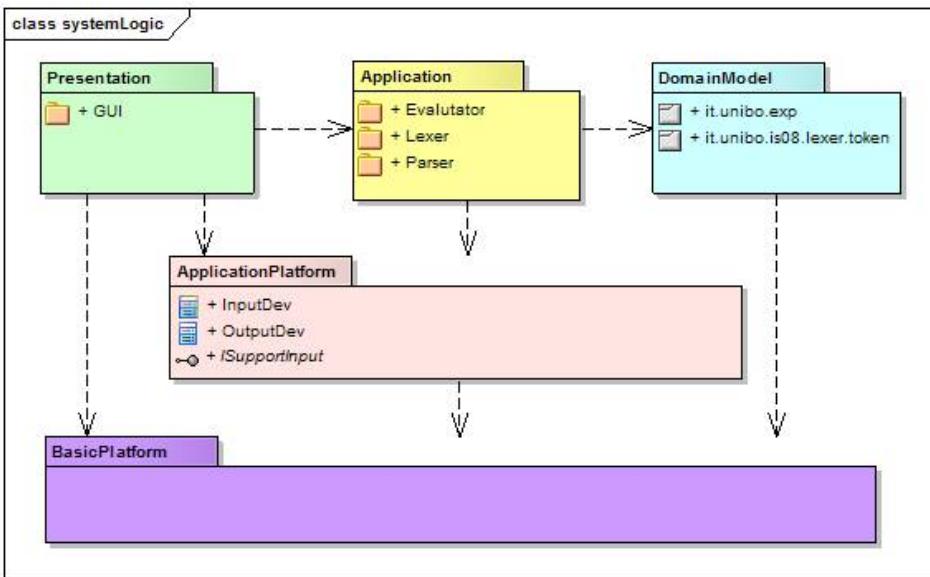
Analisi del problema

L'analisi del problema è fortunatamente nota, essendo il concetto di espressione aritmetica oggetto dello studio della matematica e dell'informatica teorica. Per quanto riguarda il processo di costruzione del software, ricordiamo che obiettivi fondamentali dell'analisi sono:

Un sistema per il riconoscimento e la valutazione di espressioni aritmetiche può essere basato sulla architettura logica rappresentata nella figura che segue:

Architettura logica

Figura 1. Architettura Logica del sistema



Questo diagramma strutturale dice che l'analista ha:

- Riconosciuto il fatto che il sistema richiede la definizione di tre sottosistemi (**three tier**):
 - il sottosistema che definisce il modello dei dati del dominio.
 - il sottosistema di presentazione che si occupa della interazione (in ingresso e in uscita) con l'utente (**interfaccia grafica**);
 - il sottosistema di elaborazione che realizza le attività che danno valore all'applicazione (**riconoscimento e valutazione di frasi**). Questo sottosistema comprende anche entità capaci di configurare (**configurator**) il sistema e di gestire le interazioni (**controller**) tra il sottosistema di presentazione e quello di elaborazione.
- La separazione delle attività interne di elaborazione dai dispositivi di I/O.
- L'articolazione del sottosistema di riconoscimento e valutazione in tre parti: analizzatore lessicale (**Lexer**) analizzatore sintattico (**Parser**) e uno o più valutatori (**Evaluator**).

Naturalmente questa architettura va intesa come una prima decomposizione del sistema, legata ai requisiti funzionali e non funzionali che si vogliono raggiungere. Essa lascia ancora indefinite molte parti, quali ad esempio l'effettivo rapporto tra **Lexer**, **Parser** ed **Evaluator** tra il controller e gli altri componenti, etc..

Per quanto riguarda il sottosistema applicativo, l'analista può osservare che la frase da riconoscere e valutare può essere resa disponibile in diverse forme: su file, come stream di dati via rete, come campo di un database, etc. Per evitare dipendenze dirette da queste diverse tecnologie di supporto alle frasi, l'analista indica come opportuno pensare alla frase da analizzare come il contenuto informativo di un **dispositivo logico** di ingresso capace di permettere l'accesso sequenziale ai singoli caratteri. Questo dispositivo deve essere logicamente reso disponibile al **Lexer** attraverso un'interfaccia quale ad esempio:

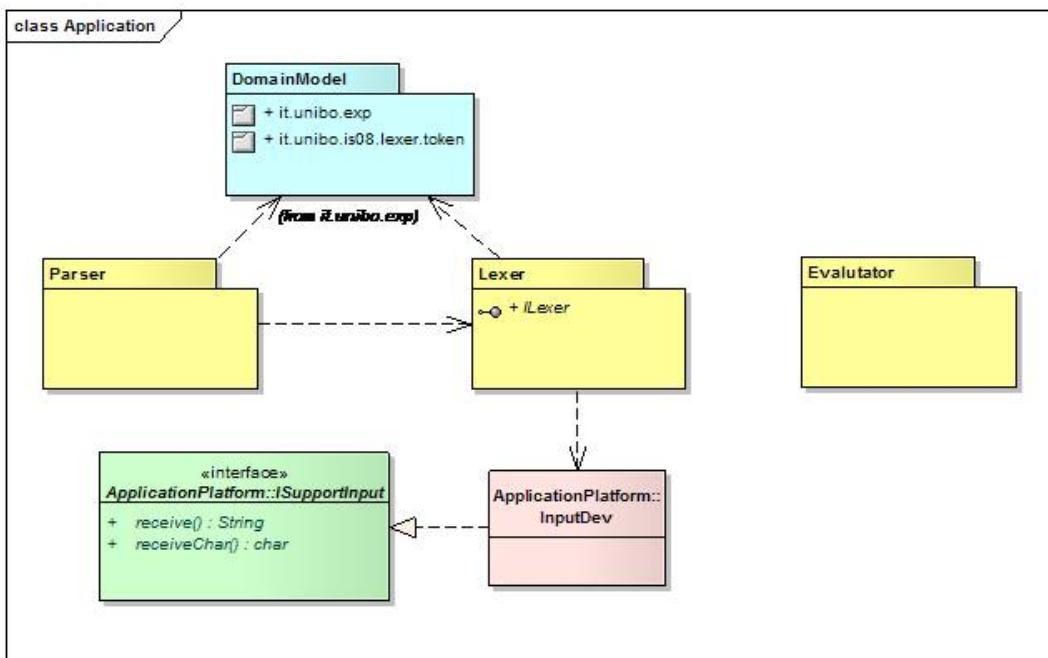
```

public interface ISupportInput {
    public String receive() throws Exception;
    public char receiveChar() throws Exception;
}

```

Sulla base di queste considerazioni la parte strutturale dell'architettura logica della parte applicativa può essere rappresentata come segue:

Figura 2. Architettura Logica della parte applicativa



L'analista ha detto che, sul piano logico, il **Parser** dipende dal **Lexer**; per essere più precisi ha detto qualcosa del tipo: **il (sottosistema) Parser deve avvelarsi di informazioni prodotte dal (sottosistema) Lexer**. Non ha detto in alcun modo che, per ottenere queste informazioni, il (sottosistema) **Parser** deve "chiamare" (una operazione de) il **Lexer**; l'informazione potrebbe essere anche "iniettata" dal **Lexer** (o da altro sottosistema) entro il **Parser**.

Per quanto riguarda la parte di valutazione delle espressioni L'analista può osservare che un valutatore può essere posto in esecuzione in due modi diversi:

- procedendo passo-passo sotto la guida del **Parser**;
- agendo, al termine della attività del **Parser**, sull'**APT** da questi prodotto.

La figura mostra che l'analista non pone vincoli in merito a questo punto.

Piano di lavoro

La fase di analisi del problema è di strategica importanza all'interno del processo di produzione in quanto fornisce informazioni fondamentali per organizzare il lavoro, per identificare e valutare i rischi e per comprendere quali e quante risorse umane e tecnologiche siano necessarie per affrontare il progetto e la realizzazione del sistema.

Nel caso specifico l'analista è fortunato, in quanto la letteratura informatica riporta approfondite analisi sul tema del riconoscimento dei linguaggi. La struttura della grammatica permette di prefigurare la necessità di impostare il progetto e la realizzazione di un **PDA** per il sottosistema) **Parser** e (come di norma accade sempre) il progetto e la realizzazione di un **ASF** per il sottosistema) **Lexer**.

Progetto

Lo scopo della fase di progetto è raffinare l'architettura logica in modo da definire un insieme di parti (funzioni, classi, oggetti, componenti) e un insieme di relazioni tra queste parti in modo che il tutto possa essere realizzato con un conveniente rapporto costo-prestazioni avendo come riferimento una specifica tecnologia (Java, COM, C++, Web, etc).

Una possibile successione di attività da svolgere in questa fase è:

- Utilizzo dei pattern architetturali e di progetto per definizione l'architettura del sistema tenendo conto in modo sistematico delle forze in gioco
- Raffinamento dei piani di collaudo per ciascun componente e per ciascun sottosistema, prima separatamente e poi in modo integrato.
- Individuazione della tecnologia o delle tecnologie di riferimento, in relazione ai requisiti funzionali e non funzionali.
- Impostazione di un primo prototipo, da presentare rapidamente al committente il fine di procedere con più sicurezza alla revisione e/o conferma dei requisiti.

La fase di progettazione sarà discussa in modo approfondito nelle sezioni a seguire.

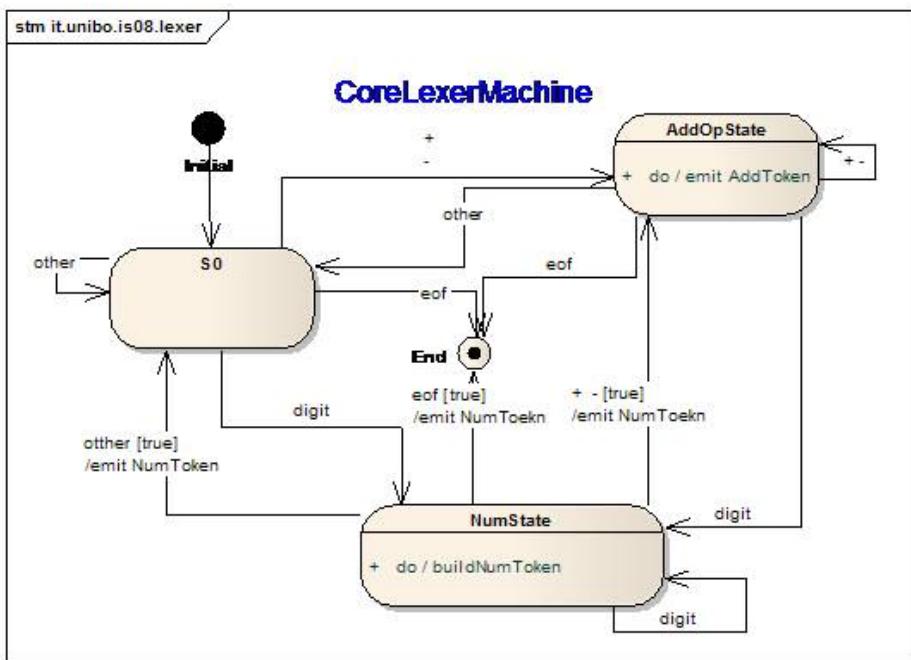
Lexer: struttura e comportamento

Per impostare il progetto della struttura e del comportamento di un analizzatore lessicale iniziamo dalla visione classica del lexer come automa a stati finiti ([ASF](#)) e quindi facciamo riferimento alla prima interfaccia d'uso introdotta:

```
public interface ILexer {  
    public Expr0Token next();  
}
```

La struttura della grammatica potrebbe portare alla definizione di un [ASF](#) come quello rappresentato dallo state diagram [UML](#) che segue:

Figura 1. Il lexer come ASF



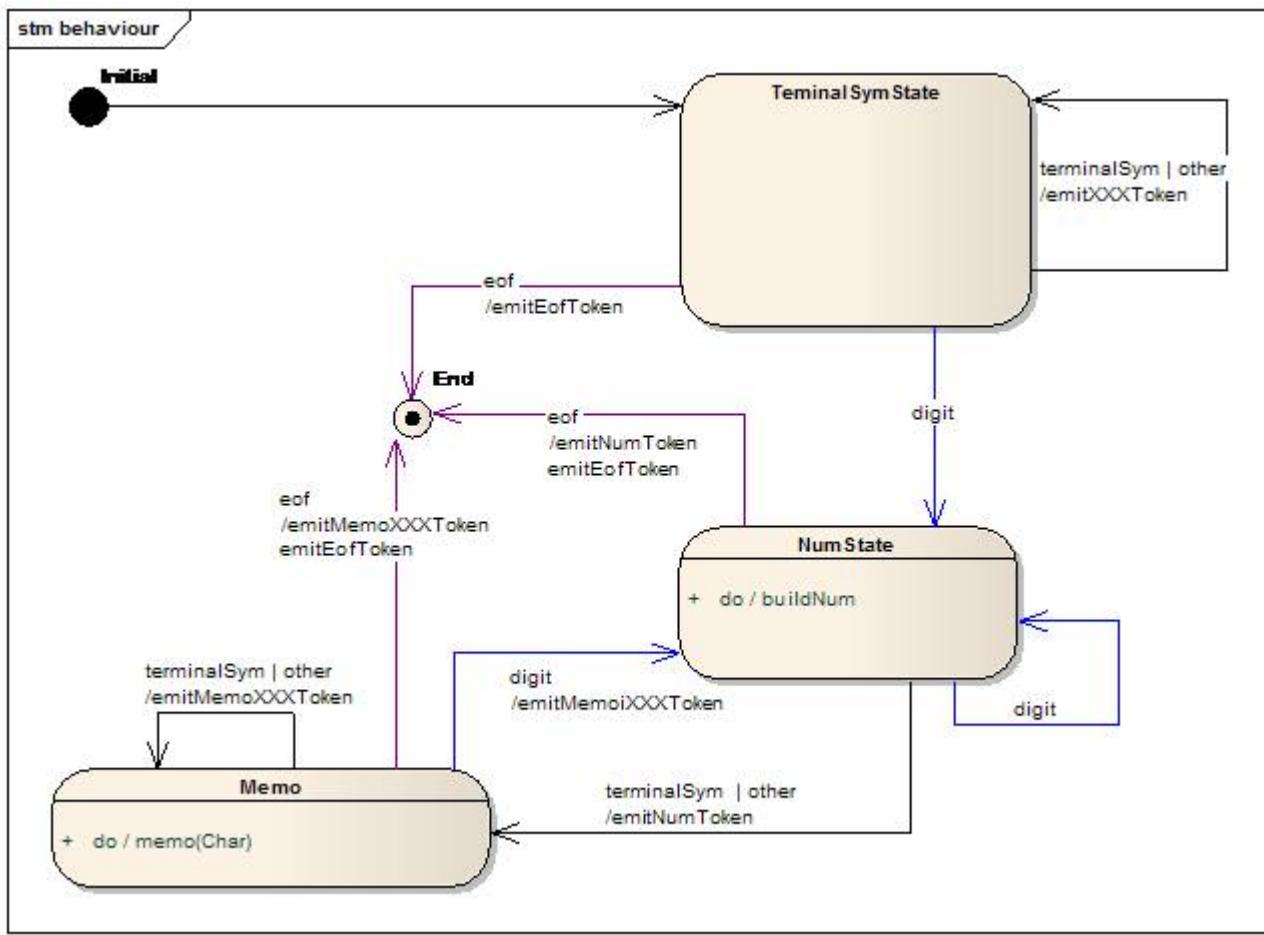
Il diagramma utilizza il termine **digit** per indicare una cifra decimale e il termine **other** per denotare qualunque simbolo (carattere) non compreso tra i simboli terminali della grammatica.

Il diagramma non mostra tutti gli stati: quelli mancanti, relativi al riconoscimento di operatori moltiplicativi e delle parentesi, sono simili allo stato **AddOpState** relativo al riconoscimento degli operatori "+" o "-".

Questo automa emette le proprie uscite sia in corrispondenza agli stati, sia in corrispondenza a transizioni; esso è quindi un automa sia di Moore sia di Mealy. Si noti che al verificarsi della transizione da **NumState** a **AddOpState** con ingresso "+" o "-" l'automa emette due uscite: un **NumToken** durante la transizione e un **AddToken** appena raggiunto il nuovo stato.

La versione mostrata nel diagramma che segue riduce il numero degli stati fattorizzando nello stato **TerminalSymState** la gestione dei simboli terminali (indicati da **terminalSym**) e dedicando lo stato **NumState** alla gestione dei **digit**.

Figura 2. Il lexer come ASF di Mealy



L'automa è ora impostato come un ASF di Mealy che emette sempre solo un token tranne in alcune transizioni verso lo stato finale **End**, in cui possono essere emessi due **token**, il secondo dei quali è sempre di classe **EofToken**.

Lo stato **Memo** rappresenta una memoria interna all'automa che tiene traccia di un token già riconosciuto ma non ancora emesso in uscita.

Dal modello dell'ASF al codice

L'ASF riconoscitore/emettitore di token rappresenta la soluzione (logica, astratta) al problema del riconoscimento lessicale. Questo automa può essere tradotto in codice in modo sistematico, creando una corrispondenza biunivoca tra gli elementi dell'automa e diverse parti del codice. Vi possono essere al proposito diverse strategie; nel seguito useremo i seguenti criteri:

1. Definizione di funzioni per la categorizzazione degli ingressi:

```

2. boolean isDigit( char n ){
3.     return (n >= '0' && n <= '9');
4. }
5.
6. boolean isEof( char n ){
7.     return ( n == 65535 );
8. }
9.
10. boolean isTerminal( char n ){
11.     return (isBraket(n) || isAddOp(n) || isMulOp(n) );
12. }
13.
  
```

```

14. boolean isAddOp( char n ){
15.     return (n == '+' || n == '-');
16. }
17.
18. boolean isMulOp( char n ){
19.     return (n == '*' || n == '/');
20. }
21.
22. boolean isBraket( char n ){
23.     return (n == '(' || n == ')');
24. }

25. Definizione dell'insieme degli stati mediante un tipo enumerativo:
26. enum State {
27.     TeminalSymState, NumState, MemoState, EndState, ErrorState };
28. Definizione dello stato corrente (con inizializzazione):
29. State curState = State.TeminalSymState;
30. Definizione della funzione caratteristica di stato (state function) dell'automa:
31. void sfn(char n){
32.     switch (curState) {
33.         case TeminalSymState: TeminalSymState(n); break;
34.         case NumState: NumState(n); break;
35.         case MemoState: MemoState(n); break;
36.         case EndState: ErrorState(n); break;
37.         case ErrorState: ErrorState(n); break;
38.         default: ErrorState(n);
39.     }//switch
40. }

```

La ***state function*** **sfn** è definita ipotizzando che il simbolo di ingresso dell'automa sia rappresentato da un singolo carattere. Il costrutto **switch** delega i dettagli del comportamento relativo a ciascun stato a una funzione specifica per quello stato; in questo modo è facile modificare il codice per aggiungere / rimuovere stati.

Riportiamo qui di seguito la funzione relativa allo stato **TeminalSymState** sottolineando come la struttura interna di questa funzione rifletta le informazioni contenute nello state diagram; in particolare essa invoca una diversa funzione di transizione (si veda il punto 6) per ciascuna categoria di ingressi. Le funzioni relative agli altri stati sono definibili in modo analogo.

```

void TeminalSymState(char n){
    if ( isEof(n) ){
        transitionToEndState( n );
        return;
    }
    if( isDigit(n) ){
        transitionToNumState( n );
        return;
    }
    if( isTerminal(n) ){
        transitionMealy(State.TeminalSymState, n);
        return;
    }
    transitionMealy(State.TeminalSymState,n);
}

```

41. Definizione della funzione caratteristica di uscita (***machine function***) dell'automa:
42. public void mfn(State newState, char n){

```

43. switch (newState) {
44. case TeminalSymState:
45.     if( curState == State.MemoState)
46.         emitTheToken( memoCh );
47.         emitTheToken( n );
48.         break;
49. case NumState: if( curState == State.MemoState)
50.         emitTheToken( memoCh );
51.         break;
52. case MemoState: if( curState == State.NumState )
53.         emitNumToken();
54.         else emitTheToken( memoCh );
55.         break;
56. case EndState: if( curState == State.NumState )
57.         emitNumToken();
58.         else
59.             if( curState == State.MemoState)
60.                 emitTheToken( memoCh );
61.                 emitToken( new EofToken( ) );
62.                 break;
63. case ErrorState: if( curState == State.NumState )
64.             emitNumToken();
65.             else
66.                 if( curState == State.MemoState)
67.                     emitTheToken( memoCh );
68.                     emitToken( new ErrorToken( ) );
69.                     break;
70. default:
71.     if( curState == State.MemoState)
72.         emitToken( new ErrorToken( ) );
73.         emitToken(new ErrorToken());
74. } //switch
75. }
76.
77. void emitTheToken( char n ){
78. Expr0Token token;
79. switch (n) {
80.     case '(': token = new LParToken( ); break;
81.     case ')': token = new RParToken( ); break;
82.     case '+': token = new AddToken( "+" ); break;
83.     case '-': token = new AddToken( "-" ); break;
84.     case '*': token = new MulToken( "*" ); break;
85.     case '/': token = new MulToken( "/" ); break;
86.     default: token = new OtherToken( );
87. } //switch
88. emitToken( token );
89. }

```

La funzione `emitToken` incapsula i dettagli in cui avviene l'emissione verso il mondo esterno di un `token`

90. Definizione di tante funzioni quante i diversi tipi di transizione dell'automa:

```

91.
92. void transitionToEndState( char n ){
93.     mfn( State.EndState, n );
94.     curState = State.EndState;
95. }
96.
97. void transitionMealy( State newState, char n){
98.     mfn( newState,n );
99.     curState = newState;

```

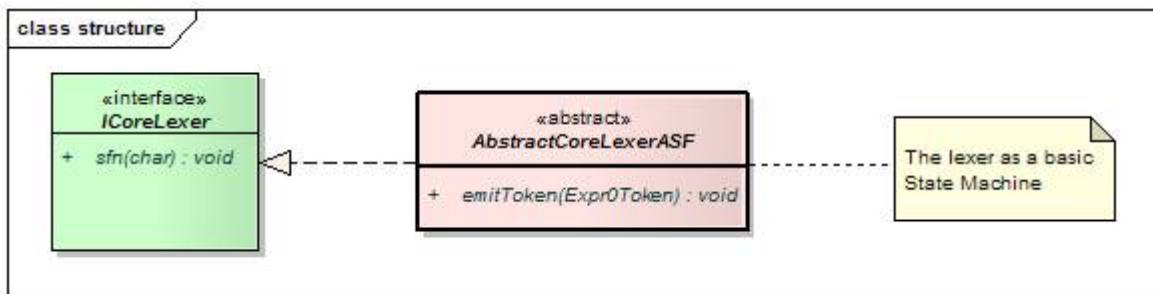
```

100. }
101.
102. void transitionToNumState( char n ){
103.   curNum = curNum*10+(n-'0');
104.   mfn( State.NumState, n );
105.   curState = State.NumState;
106. }
107.
108. void transitionToMemoState( char n ){
109.   memoCh = n;
110.   mfn( State.MemoState, n );
111.   curState = State.MemoState;
112. }
113.

```

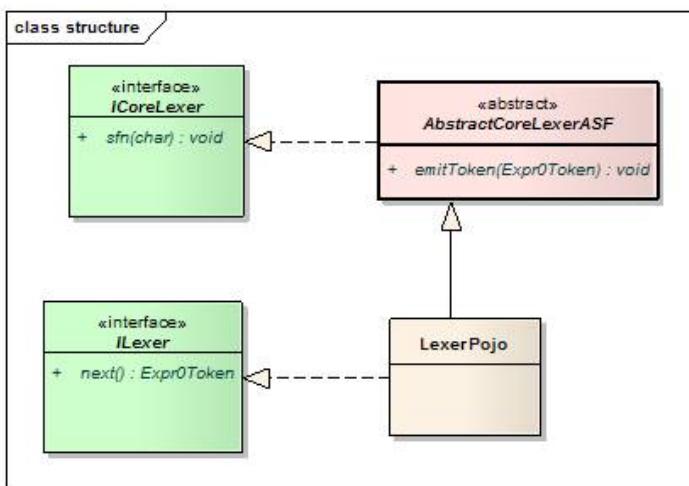
Per amore di chiarezza e modularità possiamo incapsulare il codice precedente in una classe, come rappresentato nel diagramma UML che segue:

Figura 3. Lexer core-machine



La classe astratta **AbstractCoreLexerASF** implementa l'operazione **sfn** lasciando non specificata l'operazione **emitToken**. E' stato cioè applicato il pattern xref href="../pattern/TemplateMethod.dita"/> : [GHJV95] per evitare di cablare entro questa "macchina" i dettagli relativi alla emissione delle uscite; questi dettagli dovranno essere definiti da una versione specializzata della classe; ad esempio:

Figura 4. Lexer come specializzazione della core-machine



LexerPojo è una specializzazione della classe **AbstractCoreLexerASF** che realizza in questo caso un **POJO** cioè un **Plain Old Java Object**.

Interazioni con i dispositivi

Per evitare di dover modificare il codice al variare della sorgente-dati, è opportuno incapsulare in un oggetto di supporto la dipendenza dal dispositivo di ingresso della classe che implementa l'interfaccia **ILexer**. Al momento possiamo vicolare il supporto al "contratto" rappresentato dalla seguente interfaccia (su cui torneremo in seguito):

```
public interface ISupportInput {  
    public String receive() throws Exception;  
    public char receiveChar() throws Exception;  
}
```

Le informazioni di ingresso possono essere disponibili in diverse forme: su stringa, su file, come stream di dati via rete, etc. Per ciascuna di queste forme si può introdurre una classe di implementazione dell'interfaccia **ISupportInput**; ad esempio, nel caso in cui l'ingresso sia disponibile in forma di stringa:

```
public class StringInputSupport implements ISupportInput {  
    public StringInputSupport( String sentence ){  
        ...//TODO  
    }  
    //TODO  
}
```

Per evitare la dipendenza del codice dalla classe di implementazione dei dispositivi è opportuno l'uso del pattern [xref href="..//pattern/FactoryMethod.dita">xref href="..//pattern/FactoryMethod.dita"/>](#) :

```
public class SupportIOFactory {  
  
    public static ISupportInput createStringSupport(  
                                String sentence){  
        return new StringInputSupport( sentence );  
    }  
  
    public static ISupportInput createFileSupport(  
                                String fileName ){  
        return null; //TODO  
    }  
}
```

Il piano di collaudo mostra l'uso della factory e del dispositivo:

```
public final void testMoreChar(){  
    String str = "1 2 +-      H";  
    fixture = SupportIOFactory.createStringSupport(str);  
    try {  
        for( int i=0; i<str.length();i++){  
            char curCh = fixture.receiveChar();  
            assertEquals("testMoreChar", curCh, str.charAt(i) );  
        }  
    } catch (Exception e) {  
        fail("testMoreChar " + e);  
    }  
}
```

Layers

Il diagramma **UML** che segue mostra le relazione logica tra il lexer e il supporto di ingresso:

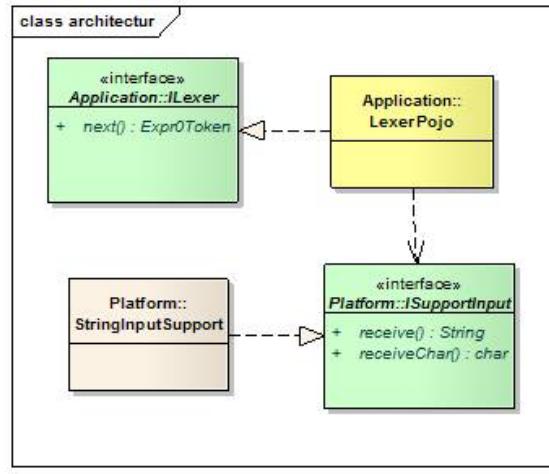


Figura 5. Dipendenza dai dispositivi di input

Poichè i supporti di ingresso sono logicamente distinti dal codice applicativo (in questo caso costituito dal lexer) e possono costituire una risorsa riusabile in altre applicazioni, può essere conveniente incapsulare le classi di implementazione in un ambiente separato da quello che incapsula il codice del lexer. Il modo più semplice per ottenere lo scopo è avvalersi dell'entità **package**; ad esempio:

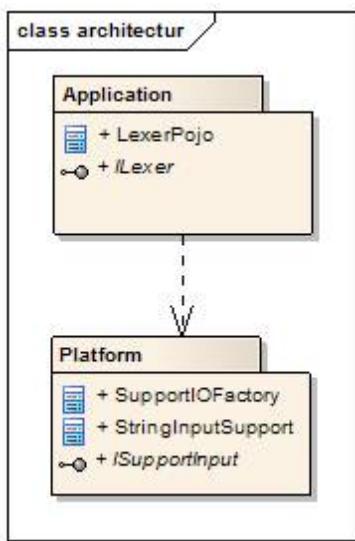


Figura 6. I layer

Questo diagramma UML:

- rappresenta una tipica struttura a livelli, in cui i package più in alto rappresenta il layer applicativo e quello più in basso la "piattaforma" di supporto;
- include la definizione della interfaccia **ISupportInput** a livello piattaforma.

Il secondo punto solleva una questione importante: quale debba essere il livello "che comanda". Nel caso attuale è evidente che la piattaforma di supporto costituisce una sorta di implementazione per esigenze espresse dal layer applicativo. Tuttavia, il fatto che sia il package **Platform** a definire ed esporre il contratto rappresentato dall'interfaccia **ISupportInput** può indurre a pensare che debba essere il livello applicativo a doversi adeguare al modo con cui la piattaforma definisce l'uso dei suoi servizi e non la piattaforma a doversi adeguare alle esigenze del livello applicativo. D'altra parte, spostare la definizione dell'interfaccia **ISupportInput** nel package **Application** renderebbe il package **Platform** incapace di esporre i propri "servizi" in forma astratta

Sciogliere un dilemma di questo tipo (si veda xref href="..//paradigmi/componentiInterfacce.dita"/>) costituisce una decisione molto importante per il progettista (se non per l'analista stesso) nel caso in cui i vari layer siano veri e propri componenti software.

Analisi sintattica

Il primo prototipo di analizzatore sintattico può essere costruito seguendo la stessa sequenza di azioni impostata per il prototipo del lexer. L'interfaccia del *Parser* può essere così definita:

```
public interface IParser{
    /**
     * Effettua l'analisi sintattica della stringa s,
     * restituendo l'Abstract Parse Tree dell'espressione.
     */
    public Exp parse( String frase );
} // IParser
```

Il piano di collaudo avrà una forma del tipo che segue:

```
public class TestParser extends TestCase {
    private IParser parser;

    public TestParser(String name) {
        super(name);
    }

    protected void setUp() {
        parser = ParserFactory.create();
    } //setUp

    public static Test suite() {
        TestSuite ts = new TestSuite();
        ts.addTest( new TestParser("testFraseOk") );
        ts.addTest( new TestParser("testFraseKo") );
        return ts;
    }

    public void testFraseOk() {
        String frase = "5 - 3 - 2" ;
        IExp resultExpected =
            new OpExp('*',
                      new OpExp('+',
                                new NumExp(1), new NumExp(2)),
                      new NumExp(3));
        IExp result = parser.parse( frase );
        checkExp( result, resultExpected );
    } //testFraseOk

    public void testFraseKo() {
        String frase = "(1+2)*";
        String resultExpected = null;
        IExp result = parser.parse( frase );
        checkExp( result, resultExpected );
    } //testFraseKo
} //TestParser
```

Durante la fase di pianificazione del collaudo emergono alcuni casi notevoli:

- l'analisi della frase 5-2-3a cosa restituisce?

- l'analisi della frase $(1+2)^*3$ - cosa restituisce?
- l'analisi della frase $1+2$ cosa restituisce?

Per rispondere osserviamo che un parser può essere impostato secondo diversi pattern di progettazione. Uno dei più diffusi e semplici da realizzare si fonda su schemi di **analisi ricorsiva discendente** e prevede l'introduzione di tante procedure di analisi sintattica quante le produzioni grammaticali. In questo modo si stabilisce (come già avvenuto nel progetto del lexer) una corrispondenza sistematica tra la struttura della grammatica che definisce il linguaggio e la struttura del codice del riconoscitore, agevolando la modifica del prototipo in caso di modifica della grammatica.

Da questa impostazione scaturisce una tecnica di analisi ricorsiva discendente che consiste nell'associare ad ogni produzione P della grammatica una procedura di riconoscimento con una signature del tipo:

```
IExp ricP( Source )
```

ove $Source$ denota (un dispositivo che rappresenta) la frase di ingresso, e $IExp$ l'albero sintattico che fornisce il risultato dell'analisi. Di solito si assegna alla procedura il compito di determinare il **più lungo prefisso** in $Source$ che costituisce una frase del linguaggio generato dalla produzione P .

Nel seguito le procedure di riconoscimento verranno definite associando all'argomento $Source$ l'analizzatore lessicale ($Lexer$) definito in precedenza. In tal modo il **Parser** potrà avvantaggiarsi del lavoro svolto dal $Lexer$ analizzando una sequenza di **token** e non una sequenza di caratteri.

Con queste premesse:

- l'analisi della frase $5-2-3a$ restituisce l'albero che rappresenta l'espressione $5-2-3$ lasciando come primo token "non consumato" il token relativo al carattere a ;
- l'analisi della frase $(1+2)^*3$ - potrebbe restituire $(1+2)^*3$ lasciando come primo token "non consumato" il token relativo al carattere $-$;
- l'analisi della frase $1+2$ restituisce $null$ in quanto non riesce a trovare il token necessario a completare con successo l'analisi.

Nel caso della frase $(1+2)^*3$ - la tecnica dell'analisi ricorsiva che adotteremo per il primo prototipo indurrà il parser alla consumazione (potremmo dire in modo "eager") del token relativo al carattere $-$ in quanto esso risulta accettabile in quel punto della frase; solo in seguito il parser si potrà accorgere che l'analisi della frase non può essere completata con successo. Nel nostro piano di collaudo prefigureremo quindi che anche per questa frase il parser restituisca il valore $null$ ad indicare la sua incapacità a completare con successo il riconoscimento, anche se un riconoscimento parziale ha avuto luogo.

Riconoscimento dei fattori

Il codice che segue definisce una possibile realizzazione di un metodo dedicato al riconoscimento dei fattori, cioè del linguaggio generato dalle produzioni che hanno come meta-simbolo F :

```
F      := N
F      ::= ( E )
```

Il metodo viene costruito sulla base del seguente ragionamento: **il metodo F deve "consumare" - partendo dal token corrente non ancora analizzato - tutti i token che si possono presentare in base alle regole che riscrivono il meta-simbolo F .**

Ogni frase prodotta dal meta-simbolo **F** ha come token iniziale o un **NumToken** o un **LParToken** relativo alla a (. Nel secondo caso il token **LParToken** deve essere seguito da una sequenza di token generabile dal meta-simbolo **E** a sua volta seguita dal token **RParToken** relativo alla).

In base a questo ragionamento possiamo impostare la struttura del metodo riconoscitore anche se non abbiamo ancora scritto il codice del metodo dedicato al riconoscimento del linguaggio generato dal meta-simbolo **E**. Supponendo che la variabile non-locale **curToken** di tipo **Exp0Token** denoti il primo token non ancora analizzato, il codice può essere scritto come segue:

```
public class Parser implements IParser{

protected ILexer lexer;
protected Exp0Token curToken;

...
    public IExp F() throws Exception {
        IExp myExp = null; //espressione riconosciuta da F
        if( curToken instanceof NumToken ){
            //Produzione F := N
            NumToken num = (NumToken)curToken;
            curToken = lexer.next();
            return new NumExp( num.getVal() );
        }
        if( curToken instanceof LParToken ){
            //Produzione F := ( E )
            curToken = lexer.next();
            myExp = E();
            if( myExp == null ) return null;
            if( curToken instanceof RParToken ){
                curToken = lexer.next();
                return myExp;
            }else return null;
        }
        else return null;
    } //F
}
```

Si noti che, dopo avere riconosciuto un token, il metodo modifica **curToken** in modo che esso referenzi il successivo token da analizzare.

Nel caso il metodo **F** sia chiamato ad operare in un momento in cui il valore di **curToken** non sia né un **NumToken** né un **LParToken**, il valore restituito è **null** in accordo a quanto discusso in fase di pianificazione del collaudo.

Riconoscimento dei termini

In modo analogo a quanto fatto per i fattori è possibile definire in modo sistematico un metodo per il riconoscimento dei termini, cioè del linguaggio generato dalle produzioni che hanno come meta-simbolo **T**:

```
T      :=  F
T      ::=  T * F
T      ::=  T / F
```

In questo caso *il metodo **T** deve "consumare" - partendo dal token corrente non ancora analizzato - tutti i token che si possono presentare in base alle regole che riscrivono il meta-simbolo **T**.* Per mettere in luce quale sia la sequenza di token lecita è opportuno eliminare le ricorsioni sinistre dalle ultime due regole, riscrivendole in notazione **BNF** come segue:

```

T      := F
T      ::= F { * F }
T      ::= F { / F }

```

In questa notazione le riscritture contenute entro i meta-simboli {} possono essere ripetute **0 o più volte**.

```

public class Parser implements IParser{

protected ILexer lexer;
protected Exp0Token curToken;

...
    public IExp T() throws Exception{
        Expr0Token myOp;
        IExp myExp, fact2;
        //Produzione T := F
        myExp = F();
        if( myExp == null ) return null;
        while( curToken instanceof MulToken ) {
            //Produzioni T := F { * F } | F { / F }
            myOp = curToken; //push the op
            curToken = lexer.next();
            fact2 = F();
            if( fact2 == null ) return null;
            myExp = new OpExp( myOp.getName(), myExp, fact2 );
        }//while
        return myExp;
    }
}

```

Si noti che:

- Il ciclo **while** esprime il concetto di ripetizione dell'attività di riconoscimento **0 o più volte**;
- il metodo di riconoscimento **F** viene invocato nella speranza che la sequenza di token sia corretta; nel caso in cui ciò non accada il metodo **F** restituirà **null** come segno di mancato riconoscimento, il che indurrà il metodo **T** a restituire **null** a sua volta;
- il metodo **T**, come ogni altro metodo di riconoscimento lascia nella variabile non locale **curToken** il primo token "non consumato";
- l'assegnamento **myOp = curToken;** alla variabile **myOp** locale alla procedura equivale ad una operazione di **push** (immissione in uno stack) del token che rappresenta l'operatore. Lo stack è in questo caso quello usato dalla macchina virtuale Java per gestire i record di attivazione delle procedure;
- il metodo **T** accumula nella variabile locale **myExp** la parte di frase che riesce a riconoscere applicando un criterio di associatività a sinistra, per cui una frase del tipo **12 / 3 / 2** viene considerata (tenendo conto delle regole ricorsive iniziali) equivalente a **(12 / 3) / 2** anzichè a **12 / (3 / 2)** .

Riconoscimento delle espressioni

Il linguaggio generato dallo scopo **E** della grammatica può venire riconosciuto da un metodo organizzato in modo del tutto analogo a quanto fatto nel caso dei termini:

```

public class Parser implements IParser{
protected ILexer lexer;
protected Exp0Token curToken;

public Parser(String s){
    lexer = LexerFactory.createLexerPojo(s);
}

```

```

public Parser(ILexer lexer){
    this.lexer = lexer;
}

public IExp parse( ) throws Exception{
    curToken = lexer.next();
    return E();
}

public IExp E() throws Exception{
Expr0Token myOp;
IExp myExp,term2;
//Produzione E := T
myExp = T();
if( myExp == null ) return null;
    while( curToken instanceof AddToken ) {
//Produzioni T := T { + T } | T { - T }
    myOp = curToken; //push the op
    curToken = lexer.next();
    term2 = T();
    if( term2 == null ) return null;
    myExp = new OpExp( myOp.getName(), myExp, term2 );
    } //while
    return myExp;
}
...
}

```

Si noti che il codice del **Parser** definisce anche due costruttori:

- un costruttore che riceve dall'esterno il **Lexer** relativo alla frase da analizzare;
- un costruttore che riceve dall'esterno una stringa che rappresenta la frase da analizzare e provvede alla costruzione di un **Lexer** attraverso una classe factory **LexerFactory**.

Il secondo costruttore facilita il collaudo del parser.

Tecnologie

I componenti software

L'attuale tecnologie informatica sta proponendo varie infrastrutture di supporto per componenti software: basta pensare a COM (Microsoft), CORBA (OMG) e Java stesso (Sun). Questa infrastrutture offrono servizi che concorrono a formare i termini di riferimento per un glossario relativo ai componenti:

- acquisition
- configuration
- installation
- composition
- integration
- deployment
- loading
- execution

La situazione complessiva è ancora in rapido divenire, anche a causa della natura di metaprodotto del software, che rende più difficile applicare tecniche e metodologie già

consolidate nell'ingegneria tradizionale o nell'hardware. Tra le principali problematiche si possono elencare quelle che discuteremo nelle sezioni che seguono.

Contratti e interfacce

In un mondo a componenti, un'interfaccia è definita in modo a sè stante e fornisce lo strumento principale con cui i componenti possono interconnettersi, svincolando il progetto e lo sviluppo dei clienti da quello dei servitori.

Concettualmente un'interfaccia costituisce un contratto che deve essere rispettato da ogni ente che vuole partecipare a un'interazione. Questa idea implica la necessità di un formalismo per la specifica del contratto e la garanzia che esista un automa capace di decidere se un dato ente soddisfa un dato contratto o meno. Questa decisione dovrebbe essere presa prima possibile: a compile-time prima che a load-time o a load-time prima che a run-time.

Tutti questi requisiti sono però difficili da soddisfare. Mentre la piena formalizzazione di un'interfaccia e la costruzione di verificatori generali ed efficienti è ancora argomento di ricerca, l'attuale tecnologia riduce un'interfaccia a un elenco di signature di operazioni in cui i dati di ingresso e di uscita sono dichiarati appartendere a specifici tipi.

Dichiarare esplicitamente il tipo di un dato costituisce una versione semplificata di un'idea più generale di contratto espresso da pre- e post-condizioni sulle operazioni. In generale, per rispettare il contratto, un cliente può seguire regole più stringenti di quelle imposte dalle pre-condizioni e un servitore può richiedere condizioni meno stringenti in ingresso ed offrire post-condizioni più rigide in uscita.

Le specifiche di contratti basate sulla sola idea del rispetto di pre- e post-condizioni non possiede però sufficiente capacità espressiva riguardo a molti importanti aspetti. Rimangono inespresse specifiche su protocolli di uso delle operazioni, su vincoli temporali e su requisiti non funzionali, quali ad esempio la complessità computazionale richiesta a un servizio in termini di tempo e memoria.

I componenti possono essere in relazione con un'interfaccia in termini di **export**, **import**, **provides** e **requires**:

- un componente è in relazione di **export** con un'interfaccia se la definisce;
- un componente è in relazione di **import** con un'interfaccia se ne richiede la definizione (da parte di un altro componente);
- un componente è in relazione di **provides** con un'interfaccia se ne fornisce (ad altri componenti attivi nel suo contesto) una implementazione;
- un componente è in relazione di **requires** con un'interfaccia se ne richiede la presenza di un numero più o meno definito (la requires è caratterizzata da **cardinalità**) di implementazioni attive nel suo contesto al fine di eseguire correttamente in esso.

Contratti e osservabilità

Le architetture logiche di molti sistemi (tra cui i sistemi event-driven) ribaltano il normale flusso di controllo, prevedendo che sia un componente a chiamare un cliente (Hollywood principle) con meccanismi di call-back.

In questi casi un componente può esporre parte del proprio stato all'osservazione di un altro componente durante lo svolgimento di un'operazione **OP**. Il cliente che esegue la call-back potrebbe osservare uno stato non consistente, in quanto il contratto stabilito dall'interfaccia di **OP** prevede di norma solo il soddisfacimento di pre-condizioni e di post-condizioni. La situazione si complica ulteriormente in caso di rientranza, cioè nel caso in cui venga invocato un nuovo metodo del componente prima che sia terminata l'esecuzione **OP**.

Sostituzione di componenti

Nei sistemi ad oggetti, un'interfaccia viene normalmente realizzata da una classe. Il meccanismo di late binding fa sì che possano venire dinamicamente connesse tra loro due parti (cliente e servitore) totalmente non consapevoli l'una dell'altra e probabilmente progettate e costruite da persone diverse. E' solo la qualità della specifica connessa all'interfaccia che può tenere insieme queste parti in modo corretto.

In molte situazioni occorre stabilire se un componente **C1** che realizza un'interfaccia **I1** possa essere usato al posto di un componente **C2** che realizza un'interfaccia **I2**. Sul piano teorico questo può accadere se **C1** appartiene a una categoria di oggetti che può essere considerata un sottotipo della categoria cui appartiene **C2**. In generale un'interfaccia **I2** può essere considerata una specializzazione di un'interfaccia **I1** se per ogni operazione con lo stesso nome (**OP**):

- il tipo del valore restituito dall'operazione **OP** definita in **I2** è un sottotipo del valore restituito dall'operazione **OP** definita in **I1**. Questa proprietà prende il nome di **covarianza** e si giustifica per il fatto che il componente sostituente deve avere post-condizioni (di cui il valore di uscita fa parte) più stringenti dell'originale;
- per ciascuno degli argomenti di ingresso di **OP**, il tipo dell'argomento definito in **I1** è un sottotipo del tipo dello stesso argomento definito in **I2**. Questa proprietà prende il nome di **controvarianza** e si giustifica per il fatto che il componente sostituente deve avere precondizioni (di cui i parametri di ingresso fanno parte) più rilassate dell'originale.
- per ciascuno degli argomenti di **OP** che può fungere da parametro di ingresso-uscita, il tipo specificato non deve essere modificato. Questa proprietà prende il nome di **invarianza** e si giustifica per il fatto che un argomento di ingresso-uscita partecipa contemporaneamente alle pre- e alle post-condizioni.

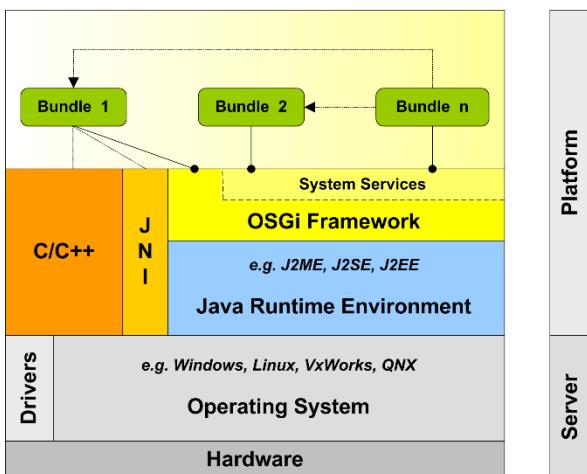
Open Service Gateway

La **Open Service Gateway initiative** (OSGi) o **OSGi Alliance** è un consorzio non-profit fondato nel Marzo 1999 da produttori hardware e software con lo scopo di creare una piattaforma (middleware) che assicuri l'interoperabilità tra applicazioni e servizi forniti attraverso la rete. Essa promuove un ecosistema trasversale di sviluppatori di software e fornisce specifiche, implementazioni di riferimento, test suite e certificazioni.

Inizialmente, le specifiche OSGi erano mirate al campo della domotica (Home-Automation), in particolare alla costruzione di residential gateway efficienti. Successivamente i suoi campi di utilizzo si sono estesi ai più diversi settori, che vanno dalla telefonia mobile alle applicazioni per desktop (l'IDE Eclipse [GBG03] è forse il più famoso). Si contano diverse applicazioni anche nell'ambito dell'industria automobilistica, dell'automazione industriale e nello sviluppo di applicazioni server.

OSGi detta le specifiche di una architettura distribuita nella quale è possibile eseguire in maniera coordinata il deployment e la gestione remota dei servizi: la [OSGi Service Platform](#). Tali specifiche definiscono un ambiente Component-Oriented e standardizzato per i servizi di rete, che è la base per un architettura Service-Oriented. In questa ottica OSGi può essere sinteticamente definito come un [Universal Middleware](#) che permette la creazione di software distribuibile in forma binaria per diverse piattaforme, in diverse industrie, e per diversi scopi.

Figure 1. Layer OSGi

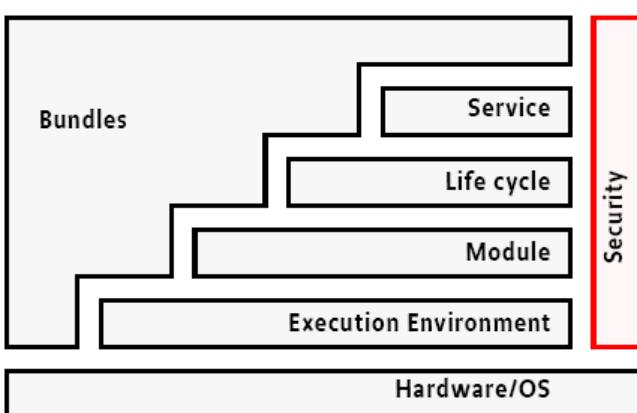


Il Framework OSGi

Il componente principale delle OSGi specifications è rappresentato dal framework OSGi. Esso implementa un modello a componenti dinamico e fornisce un ambiente standardizzato, sicuro, modulare ed estensibile per l'esecuzione di applicazioni.

I mattoni per la costruzione delle applicazioni sono i **bundle** e possono essere installati, attivati, disattivati e disinstallati a run-time, anche in modo remoto e senza bisogno di riavviare il sistema.

Come evidenziato nella figura precedente, il framework si poggia sulla Java Virtual Machine, il che gli conferisce una totale indipendenza dalla piattaforma e la capacità di offrire ai bundle la possibilità di accedere direttamente alle risorse di sistema. Il framework OSGi è costituito da quattro livelli (o *layer*):



- **Execution environment**: rappresenta un ambiente di esecuzione standardizzato per i bundle basato sulle Java specifications. A questo livello vengono gestite le possibili configurazioni dell'ambiente di esecuzione a seconda del sistema operativo e dei dispositivi sui quali verranno eseguite.
- **Module**: fornisce le primitive per la costruzione dei moduli (bundle), definisce la politica di class-loading e di gestione delle dipendenze fra moduli.
- **Life Cycle**: fornisce le primitive per la gestione a run-time dei bundle: installazione, attivazione, disattivazione, aggiornamento e disininstallazione. Utilizza il **Module Layer** per gestire il class-loading.
- **Service Registry**: definisce il **servizio** quale entità fondamentale di composizione per le applicazioni e fornisce un modello di cooperazione fra bundle che tiene in conto del forte dinamismo del sistema.

Esiste anche un altro livello trasversale e interagente con tutti gli altri: il **Security Layer**. Si basa sostanzialmente sul Java2 security model fornendo un sistema per firmare e verificare l'autenticità dei bundle; in più aggiunge i meccanismi e le primitive per una gestione dinamica dei permessi.

Execution Environment

La disponibilità di implementazioni della macchina virtuale Java per diversi ambienti operativi è la chiave della portabilità di Java, proclamata nello slogan **write once, run everywhere**. La **JVM** (Java Virtual Machine) realizza infatti un ambiente di esecuzione omogeneo, che nasconde le specificità del sistema operativo sottostante. Per ovviare alle limitazioni computazionali dei moderni dispositivi (ad esempio cellulari e palmari) Java ha affiancato alla sua edizione standard ([J2SE]) una nuova edizione ridotta e configurabile a seconda del dispositivo su cui viene installata, la **Java 2 Micro Edition** ([J2ME]).

La introduzione di più tipi di macchina virtuale ha riaperto il problema della portabilità del software; OSGi lo affronta consentendo di **specificare** quale ambiente di esecuzione Java utilizzare per una specifica applicazione. OSGi permette di scegliere fra svariate configurazioni e profili quali ad esempio **J2SE**, **CDC**, **CLDC**, **MIDP** etc. Inoltre OSGi ha standardizzato un proprio ambiente di esecuzione basato su**Fundation Profile** [JME-FP] ed una sua variante, **OSGi/Minimum-1.1** [OSGi/Min], che specifica l'ambiente di esecuzione minimo perché il framework OSGi possa funzionare

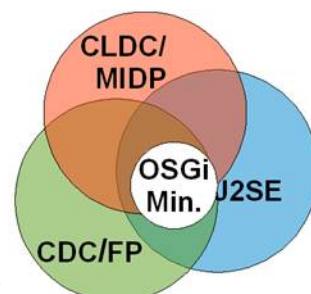


Figure 2. Minimum Execution Environment OSGi

L'ambiente minimale comprende un sottoinsieme delle primitive offerte dagli altri ambienti di esecuzione consentiti e risulta pertanto compatibile con essi. Per ogni componente (**bundle**) è possibile specificare un proprio ambiente di esecuzione, sarà poi il framework a gestire tale richiesta in maniera trasparente.

Il Module Layer

La piattaforma standard Java fornisce solo un supporto limitato al packaging e al deploying delle applicazioni e dei componenti. Ad esempio in Java non esiste un modo per esprimere la dipendenza di un file jar da altri file jar (la voce "Class-path" nel manifest non è un modo affidabile per esprimere tali dipendenze). Pertanto è impossibile stabilire se codice encapsulato in un file jar funzionerà a run-time oppure no, a causa della mancanza di parti contenute in altri file jar. Inoltre tutto il codice contenuto in un file jar è visibile all'esterno; si possono dichiarare nel codice delle classi private o protected, ma in tal modo tali classi risulterebbero non visibili all'interno dello stesso package.

Il Module Layer fornisce una soluzione generica e standardizzata per la gestione della modularizzazione del codice Java.

Dipendenze

Nel Framework OSGi la gestione delle dipendenze e collaborazioni fra bundle avviene a due livelli: dipendenze dei bundle/package ([Deployment Dependency](#)) e dipendenze dei servizi ([Service Dependency](#)). Il [Module Layer](#) si occupa in particolare del primo tipo di dipendenze (si veda [Deployment dependency](#)), mentre il [Service Layer](#) si occupa di fornire il supporto per la gestione del secondo (si veda [Service Dependency](#)).

Il Life Cycle Layer

Il Life Cycle Layer si poggia sul Module Layer e sul Security Layer per fornire i meccanismi per gestire in maniera sicura il ciclo di vita dei bundle. Il Module Layer in parte si occupa di questo aspetto definendo l'architettura di class loading, ma non specifica come un bundle debba essere installato, aggiornato o disinstallato né formalizza sistematicamente le fasi di vita di un bundle. (si veda [Life cycle](#)).

Il Service Layer

Il Service Layer definisce un modello collaborativo e dinamico di tipo [publish, find and bind](#) (pubblica, torva e collega) che consente ai bundle sia di erogare i propri servizi, che di usufruire di quelli messi a disposizione dagli altri bundle, senza che nessuno di loro abbia conoscenza a priori degli altri. Questo livello dà alla piattaforma OSGi i connotati di una prima [Service Oriented Architecture](#) (SOA).

Eventi e Listener

Il framework OSGi genera e gestisce una serie di eventi che descrivono il cambiamenti di stato dell'intero sistema. Esistono tre categorie principali di eventi:

- Bundle-Event
- Framework-Event
- Service-Event

I primi due tipi di eventi sono relativi al Life Cycle Layer e rappresentano rispettivamente: gli eventi relativi al cambiamento di stato di un bundle e quelli relativi alle modifiche del framework, ed agli eventuali errori. L'ultimo tipo di evento è relativo al Service layer; eventi di questo tipo vengono generati in caso di registrazione, cancellazione o modifica delle proprietà di un servizio.

Il framework definisce anche diversi tipi di listener che si occupano di catturare gli eventi e notificarli agli interessati. In particolare i **BundleListener** vengono attivati quando arriva un **BundleEvent**, quindi un cambiamento nello stato di un bundle (ad esempio se viene fatto partire un bundle), i **FrameworkListener** quando arriva un **FrameworkEvent** (che può segnalare errori, warning, refresh, ecc.) ed i **ServiceListener** quando arriva un **Service-Event** (ad esempio al momento della registrazione di un nuovo servizio).

Il bundle e il suo manifest

Il bundle è l'unità fondamentale di progettazione e distribuzione del software nel framework OSGi. Un bundle è costituito da un **jar** (file **Java Archive**) contenente il codice e le risorse (ad esempio file **HTML**, **gif** etc.) necessarie al suo funzionamento nel framework.

Fra le varie classi implementate all'interno di un bundle deve essere presente una classe **Activator** che esegue l'inizializzazione dei parametri del bundle in fase di attivazione (metodo **start**) e il rilascio di risorse (clean up) in fase di disattivazione (**stop**).

Ogni bundle deve avere al suo interno un file manifest (**./META-INF/MANIFEST.MF**) che ne descrive il contenuto e fornisce tutte le informazioni necessarie al framework per la sua corretta installazione e attivazione. Ad esempio nel manifest sono elencate le dipendenze da altre risorse (package, bundle ect.) che devono essere disponibili per il bundle prima che venga attivato.

Il manifest

Di seguito si riporta un esempio delle voci principali che compaiono nel file manifest di un bundle:

La versione del framework

Bundle-ManifestVersion:2

Specifica la versione del framework OSGI adottata (2 per la release 4 di OSGI, 1 per le quelle precedenti).

Il nome simbolico

Bundle-SymbolicName:it.unibo.MyBundle

Il nome simbolico deve essere unico all'interno del sistema (per una istanza del framework) e permette di identificare univocamente ciascun bundle. Ogni bundle è univocamente identificabile all'interno del framework tramite la coppia **SymbolicName-Version**. Specificando la direttiva **singleton** di seguito al nome simbolico si stabilisce che il bundle può esistere nel framework in un'unica versione: in questo caso il nome simbolico da solo è sufficiente per identificare il bundle.

La versione del bundle

Bundle-Version:1.2.0:

La versione, insieme al nome simbolico, costituisce un identificativo unico per i bundle.

Il nome

Bundle-Name:MyBundle

E' il nome assegnato al Bundle (non può contenere caratteri spazio).

La descrizione

Bundle-Description:a bundle description

Fornisce una breve descrizione del bundle.

Il class path

Bundle-ClassPath:..jar/http.jar

Definisce una lista di file **jar** e directory (interne al bundle), separate da virgola, contenenti classi e risorse.

L'activator

Bundle-Activator:it.unibo.env.Activator

Specifica il nome della classe che attiva (**start**) e disattiva (**stop**) il bundle.

La lista dei package importati

Import-Package:org.osgi.util.tracker;version=1.3

Insieme dei package richiesti dal bundle per poter funzionare. La direttiva **version** permette di distinguere anche tra più versioni dello stesso package.

La lista dei package esportati

Export-Package:org.osgi.util.tracker;version=1.3

Insieme dei package esportati dal bundle.

La lista dei package importati dinamicamente

DYNAMICImport-Package:com.acme.plugin.*

Insieme dei package importati dinamicamente quando necessario.

URL per l'aggiornamento

Bundle-UpdateLocation://www.deis.unibo.it/bndl.jar

Specifica l'URL da cui è possibile scaricare gli aggiornamenti.

L'ambiente di esecuzione richiesto

Bundle-RequiredExecutionEnvironment: CDC-1.0/Foundation-1.0

Specifica l'ambiente di esecuzione necessario per il funzionamento del bundle. Per applicazioni pensate per lavorare sulla **J2ME** l'ambiente di esecuzione si specifica con la coppia configurazione-profilo. Ad esempio **CDC1.0/Foundation-1.0** specifica l'ambiente di esecuzione della **J2ME** con **Connected Device Configuration** e profilo **Foundation-1.0**.

Lista dei bundle richiesti

Require-Bundle:it.unibo.deis.myBundle

Contiene un elenco di bundle (denotati dal loro nome simbolico) che devono essere installati sul framework prima che il bundle possa essere posto in esecuzione.

Deployment dependency

Le **Deployment Dependency** possono essere classificate in due diversi tipi:

1. Bundle-to-Package
2. Bundle-to-Bundle

Dipendenze Bundle-to-Package

Un bundle può necessitare del codice contenuto in un altro bundle, per esempio di una classe che utilizzerà per costruire oggetti. Per far questo i bundle dichiarano nel **manifest** i package da importare e quelli da esportare che saranno resi disponibili agli altri componenti. Questo tipo di dipendenza è detto **Bundle-to-Package** e viene automaticamente gestita dal framework OSGi, che non permette l'esecuzione di un bundle finché tutti i package che importa non sono esportati da almeno un altro bundle.

Questa è una dipendenza statica, nel senso che deve essere risolta prima di poter porre in esecuzione l'applicazione, ma lascia, nel senso che non viene creata una relazione biunivoca stretta tra un bundle che esporta e uno che importa, ma tra un bundle che necessita di un package e il package stesso, che può essere fornito da qualsiasi altro bundle.

Dipendenze Bundle-to-Bundle

Il framework OSGi prevede la possibilità di inserire nel manifest l'attributo **Require-Bundle** che crea un legame stretto tra due o più bundle. Questa direttiva impone di poter eseguire il bundle che l'ha inserita nel proprio manifest solo se tutti i bundle richiesti sono in esecuzione.

Questo crea una dipendenza **Bundle-to-Bundle** ancora statica, ma molto più stretta rispetto alla precedente, in quanto relativa non ad un package, che potrebbe essere esportato da qualunque bundle, ma ad uno specifico bundle.

Anche se in molti casi può tornare utile, OSGi consiglia di limitare al minimo questa dipendenza e di usare piuttosto dipendenze di tipo **Bundle-to-Package** che garantiscono una maggior flessibilità ed adattabilità dei componenti.

Dipendere da un bundle piuttosto che da un singolo package ha infatti due inconvenienti: da un lato il rischio di richiedere più package di quanti effettivamente ne servano, ritardando inutilmente l'attivazione; dall'altro il legame stretto che viene a crearsi con uno specifico bundle, il che limita la possibilità di sostituire parti del sistema.

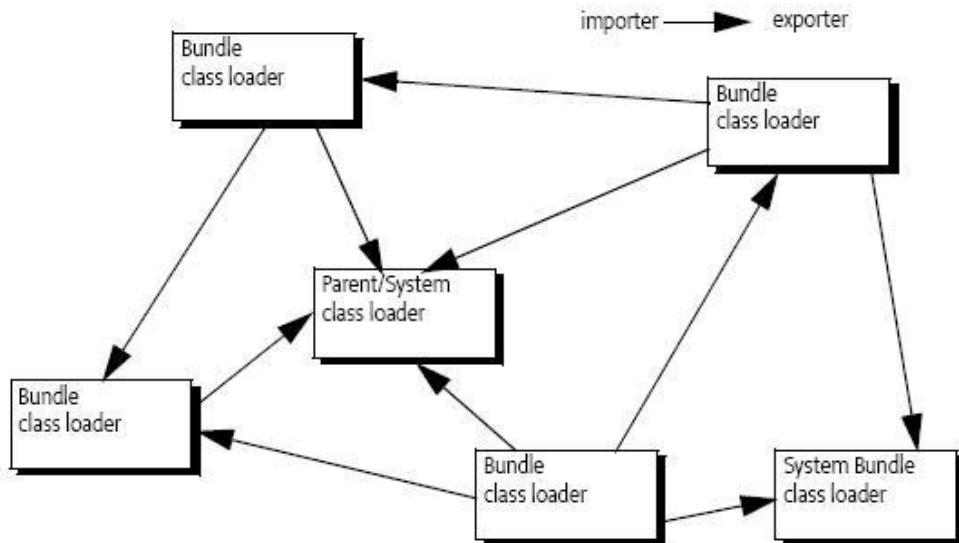
Come la precedente, anche questa dipendenza è gestita automaticamente dal framework.

Class loading

Per capire appieno come il Framework OSGi gestisce la Deployment Dependency è necessario esaminare accuratamente l'architettura di supporto al class-loading che esso fornisce.

Più bundle possono condividere la stessa macchina virtuale Java. All'interno di tale macchina virtuale essi possono nascondere package e classi agli altri bundle oppure condividerli con essi. La strategia adottata da OSGi consiste nel **associare un singolo class loader ad ogni bundle** e nel creare una rete di delegazione gerarchica fra tali class loader sfruttando i meccanismi di supporto al class loading fornito dalla JVM:

Figure 1. Architettura del class-loading OSGi



Il class loader può caricare classi e risorse dal:

- **Boot Class Path:** la boot class path contiene i package che compongono la piattaforma Java.
- **Framework Class Path:** il Framework di solito ha un proprio class loader per caricare le classi che lo implementano.
- **Bundle Space:** il bundle space è composto dal file **jar** associato al bundle, più ogni altro file **jar** ad esso strettamente collegato, come ad esempio i così detti **fragments** (OSGi fornisce un meccanismo che permette ai bundle di essere sintetizzati come somma di più frammenti ognuno situato nel proprio **URL**. Questo è ad esempio il metodo usato per fornire ad un plug-in internazionale il supporto per lingue multiple).

Se un **class-loader** riceve la richiesta di caricare una classe che non è fra quelle sopra elencate potrà richiederla ai class-loader suo parent così come questi ai propri parent e così via. Se si definisce il **class-space** come tutte le classi raggiungibili da un certo class loader associato ad un particolare bundle, allora un **class-space** potrà essere composto dalle classi contenute:

- nel parent class loader (normalmente i package della piattaforma Java contenuti nella boot class path);
- nei package importati, elencati nella voce **Import-Package** del manifest;
- nei bundle richiesti, elencati nella voce **Require-Bundle** del manifest;

- nei package propri del bundle, elencati nella voce **Bundle-ClassPath** del manifest;
- nei fragments allegati al bundle.

Un **class-space** deve essere consistente, nel senso che non potrà contenere classi con lo stesso nome. Al contrario più class-space, nella stessa piattaforma OSGi, possono contenere classi con lo stesso nome. Il Module Layer supporta un modello per cui versioni multiple della stessa classe possono essere presenti nella stessa macchina virtuale.

Prima che ad un bundle venga associato un class-loader è necessario risolvere tutte le dipendenze dichiarate nel suo file manifest. Sarà il framework a farsi carico di tale compito attraverso un processo iterativo il cui risultato sarà una rete di connessioni fra bundle.

Il framework OSGi è un framework dinamico pertanto può accadere che in qualsiasi momento ad un bundle, per il quale il processo di risoluzione delle dipendenze aveva dato esito positivo, vengano a mancare delle risorse necessarie al suo funzionamento. La gestione di questa eventualità viene lasciata al programmatore; l'unico supporto dato dal framework è la voce **DYNAMICIMPORT-PACKAGE** nel file **manifest** che consente di specificare una lista di package alternativi in cui è possibile trovare la risorsa richiesta. Inoltre il framework può ritardare la creazione di un particolare **class-loader** finché non è realmente necessario.

Life cycle

Le fasi di vita di un bundle sono: installazione, risoluzione, attivazione, disattivazione, disinstallazione e aggiornamento, quest'ultima realizzata come combinazione delle precedenti. Ognuna di queste fasi può far cambiare lo stato di un bundle. Un bundle può trovarsi in uno dei seguenti stati:

- **INSTALLED**: il bundle è stato installato con successo.
- **RESOLVED**: tutte le risorse di cui il bundle necessita sono disponibili. Questo stato indica che il bundle è pronto per essere attivato o è stato disattivato.
- **STARTING**: il bundle sta per essere attivato (il metodo **start()** dell'**Activator**, è stato chiamato ma non ha ancora fornito un risultato).
- **ACTIVE**: il bundle è stato attivato con successo ed è in esecuzione.
- **STOPPING**: il bundle sta per essere disattivato (il metodo **stop()** dell'**Activator** è stato chiamato ma non ha ancora fornito un risultato).
- **UNINSTALLED**: il bundle è stato disinstallato. Rappresenta lo stato finale.

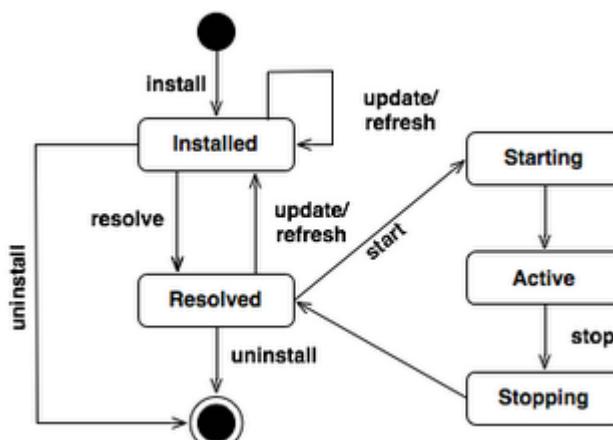


Figure 1. OSGi Life-Cycle

Per aggiungere un bundle ad un'istanza del framework lo si deve innanzitutto installare al suo interno. L'installazione può essere effettuata soltanto da un altro bundle oppure dal framework stesso (ad esempio come comando passato da console) che a sua volta è un bundle ([System-Bundle](#)). L'interfaccia [BundleContext](#) fornita ad ogni bundle dal framework (si veda [Bundle context](#)) consente di installare un bundle tramite il metodo [installBundle\(String URL\)](#).

Se il bundle supera una fase di validazione relativa alla sicurezza viene installato ([INSTALLED](#)) ed il metodo [installBundle](#) produce come risultato un [bundle object](#) (si veda [Bundle object](#)) ovvero la rappresentazione a run-time di un bundle che consente di controllare tutte le restanti fasi di vita del bundle.

La locazione del bundle può essere sia remota che locale; in ogni caso il framework ne esegue il download e ne tiene una copia in una cartella locale. La fase di installazione gode delle seguenti proprietà:

- **persistenza**: un bundle rimane installato nonostante la terminazione del framework o della macchina virtuale Java a meno che non venga esplicitamente disininstallato;
- **atomicità**: la fase di installazione deve installare completamente il bundle o, se l'installazione fallisce, deve lasciare il sistema nello stato consistente precedente il tentativo d'installazione.

Una volta installato il bundle, il framework tenta di risolvere tutte le dipendenze dichiarate nel file [manifest](#) attraverso un processo iterativo di risoluzione. Se tale processo andrà a buon fine il bundle assumerà lo stato [RESOLVED](#) e potrà essere attivato, altrimenti resterà nello stato [INSTALLED](#) fino a che tutte le dipendenze non verranno risolte: in questo caso il bundle non potrà essere messo in esecuzione ma i package che esporta potranno comunque essere utilizzati dagli altri bundle.

Quindi se il bundle passa nello stato [RESOLVED](#) può essere attivato utilizzando il metodo [start\(\)](#) del suo [Activator](#). Tale metodo pone il bundle nello stato [STARTING](#), registra i servizi forniti dal bundle e svolge le operazioni di inizializzazione. Se tutto ciò va a buon fine il bundle passa nello stato [ACTIVE](#) ed è in esecuzione. La terminazione di un bundle può avvenire per mezzo di un altro bundle o del bundle stesso che di fatto può auteterminarsi.

Ad una richiesta di terminazione, il bundle entra nello stato di [STOPPING](#) ed il framework esegue il metodo [stop\(\)](#) dell'[Activator](#). Questo metodo deve eseguire la cosiddetta chiusura dolce, finendo di servire i clienti ancora in attesa nel caso stia erogando un servizio (si veda [Service Dependency](#)), informando il framework che i servizi da lui erogati non sono più disponibili e liberando le risorse allocate dopo l'attivazione. Se tutto va buon fine il bundle viene disattivato e ritorna nello stato [RESOLVED](#). Naturalmente anche in questo stato il bundle non è attivo ma i package da lui esportati restano utilizzabili dagli altri bundle nel framework.

A questo punto il bundle può essere disininstallato (ed eliminato dal framework) tramite il metodo [uninstall\(\)](#) fornito dal [BundleContext](#). In tal caso il bundle viene posto nello stato [UNINSTALLED](#), l'evento viene notificato agli altri bundle del framework e vengono liberate tutte le risorse che erano state destinate a quel bundle.

Oltre alle operazioni citate è possibile eseguire anche l'operazione di [update](#) (o [refresh](#)). Questa operazione ha come risultato l'aggiornamento del codice del bundle (che potrebbe cambiare versione, ma deve mantenere lo stesso nome simbolico). Un bundle

aggiornato rende immediatamente disponibili i package che esporta. Allo stesso tempo i package esportati dalla versione precedente restano disponibili finché il framework non viene riavviato (o non viene chiamato il metodo `refresh-Packages()`). Il comando di `update` (che come gli altri può essere inviato solo da un bundle) può arrivare anche quando il bundle si trova nello stato `ACTIVE`, in tal caso viene automaticamente terminato, aggiornato e fatto ripartire.

Bundle object

Ad ogni bundle installato nel Framework OSGi è associato un oggetto `BundleObject` che è la rappresentazione a run-time del bundle stesso. Tale oggetto può essere usato per gestire il ciclo di vita del bundle (attivazione, disattivazione etc.) e per reperire tutte le informazioni relative al bundle (identità, manifest, stato etc.).

Un bundle all'interno del framework può essere identificato da:

- **Bundle identifier:** un numero intero unico assegnato dal framework per il totale tempo di vita del bundle. Tale numero viene assegnato in maniera crescente man mano che i bundle vengono installati e potrà cambiare solo se il bundle viene aggiornato o se il framework viene riavviato (il `System-Bundle` ha sempre identificativo `0`).
- **Bundle location:** l'URL da cui è stato installato il bundle. All'interno dello stesso framework la locazione del bundle deve essere unica e non varia in seguito ad aggiornamenti.
- **Bundle symbolic name:** un nome assegnato dallo sviluppatore del bundle. Insieme alla versione del bundle rappresenta un identificatore unico.

Vari metodi possono essere invocati tramite quest'oggetto. I più importanti sono elencati di seguito:

- `getBundleId()`: fornisce il Bundle identifier associato al bundle.
- `getLocation()`: fornisce il Bundle location associato al bundle.
- `getSymbolicName()`: fornisce il Bundle Symbolic name associato al bundle.
- `getState()`: fornisce lo stato corrente del bundle.
- `start()`: avvia il bundle.
- `stop()`: disattiva il bundle.
- `uninstall()`: disininstalla il bundle.
- `update()`: aggiorna il bundle.

Bundle context

La relazione fra il framework ed i bundle in esso installati è realizzata tramite l'uso di un particolare oggetto chiamato `BundleContext`. Esso rappresenta il contesto di esecuzione di ogni singolo bundle all'interno della OSGi Service Platform e consente di:

- installare nuovi bundle nel framework (ad esempio tramite il metodo `install-Bundle(String URL)`).
- ottenere informazioni sugli altri bundle installati nel framework (tramite il metodo `getBundles()`).
- accedere all'area di memoria persistente riservata al bundle (tramite il metodo `getDataFile(String filename)` che crea un file in tale area di memoria).
- ottenere i service object relativi ai servizi registrati nel framework (tramite il metodo `getService(ServiceReference reference)`),

- registrare servizi nel framework (tramite il metodo `registerService(String classname, Object service, Dictionary properties)`).
- sottoscrivere o meno la ricezione degli eventi generati all'interno del framework (tramite il metodo `addBundleListener (BundleListener listener)`).

Servizi OSGi

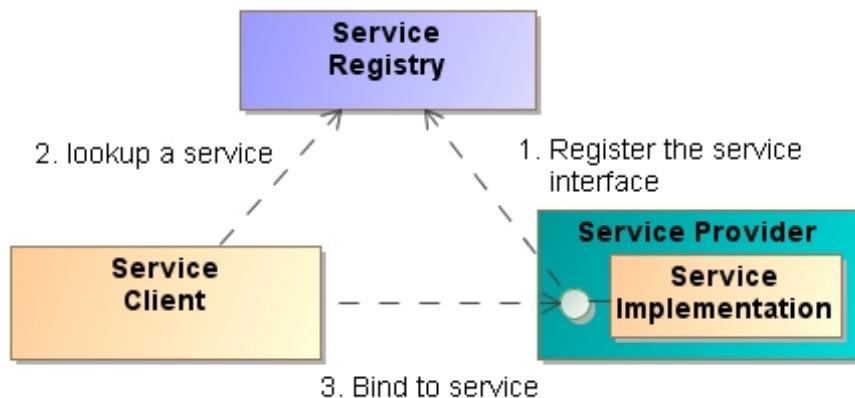
Un servizio OSGi può essere definito come l'implementazione di una funzionalità il cui stato non dipende dal contesto in cui è usato. L'idea di OSGi è quella di separare la specifica del servizio dalla sua realizzazione, per cui un servizio è definito semanticamente da un'interfaccia ed implementato da un oggetto.

L'interfaccia del servizio deve essere specificata in modo platform-independent tramite il costrutto `interface` di Java.

L'oggetto realizzatore del servizio è creato e gestito da un bundle, il quale lo deve registrare su un registro pubblico (il `Service Register` OSGi) in modo da renderlo disponibile agli altri bundle. Le dipendenze tra il servizio e il bundle che lo possiede sono risolte direttamente dal framework: ad esempio se un bundle viene terminato, i servizi che aveva registrato vengono eliminati automaticamente dal `Service Register`.

La figura descrive il modello a servizi OSGi, che si svolge in tre fasi: pubblicazione del servizio da parte di un fornitore, ricerca (e ottenimento) di un servizio da parte del cliente, consumo del servizio.

Figure 1. Pubblicazione, ricerca e acquisizione di un servizio



Processo di pubblicazione di un servizio

Se si vuole creare un bundle che eroghi un servizio, si deve per prima cosa decidere l'interfaccia del servizio (OSGi definisce numerose interfacce standard per i servizi di uso più comune). Il secondo passo è creare una classe che implementi quell'interfaccia: gli oggetti di tale classe costituiranno l'implementazione del servizio. A questo punto è necessario pubblicare il servizio su un registro, in modo che tutti i bundle possano venirne a conoscenza.

Il registro in questione è il `Service Register` OSGi e la registrazione avviene tramite il metodo `registerService` del `BundleContext`

```
ServiceRegistration registerService(
    String clazz, Object service, java.util.Dictionary properties)
```

Il parametro `clazz` fornisce il nome dell'interfaccia del servizio, mentre il parametro `service` è in generale un oggetto e può essere o un'istanza di classe che implementa l'interfaccia del servizio oppure un'istanza di `ServiceFactory`, che viene utilizzata per personalizzare il servizio in base a chi lo richiede. Il parametro `properties` serve per attribuire delle proprietà alla registrazione del servizio. Tali proprietà sono delle coppie `attributo = valore`, che aggiungono delle specifiche alla descrizione del servizio. Il metodo restituisce (se tutto va a buon fine) un oggetto `ServiceRegistration` che può essere usato per cancellare il servizio o per accedere alle sue proprietà (per leggerle o modificarle).

Processo di ricerca di un servizio

Un cliente che voglia accedere ad un servizio deve anzitutto essere a conoscenza dell'interfaccia del servizio richiesto. Nota tale interfaccia, è sufficiente accedere al `ServiceRegister` OSGi per vedere se qualche bundle implementa quel servizio: a tale fine si può utilizzare uno dei metodi messi a disposizione dal `BundleContext`.

Uno di questi metodi è ad esempio

```
ServiceReference[ ] getServiceReferences(  
    String clazz, String filter)
```

in cui il primo parametro fornisce il nome dell'interfaccia Java associata al servizio, mentre il secondo è una stringa rappresentante un filtro di ricerca da applicare alle proprietà del servizio richiesto. Il metodo restituisce un'array di oggetti di tipo `ServiceReference` che soddisfano la ricerca, oppure `null` se non ce ne sono. Un oggetto `ServiceReference` rappresenta un riferimento al servizio e ne incapsula le proprietà e le altre meta-informationi.

Come noto l'interfaccia Java da sola potrebbe non essere sufficiente a caratterizzare il servizio che rappresenta; in questo caso è possibile registrare il servizio con proprietà aggiuntive che ne completino le specifiche. Il filtro potrebbe poi selezionare solo i servizi di interesse, ad esempio quelli erogati in una certa lingua, oppure con certi requisiti non funzionali (come i tempi di risposta, l'efficienza, ecc).

L'oggetto `ServiceReference` può essere memorizzato e passato agli altri bundle senza implicazioni di dipendenza. Si deve però tenere in conto che la sua validità è legata alla presenza della registrazione del servizio sul `ServiceRegister`: se il servizio viene cancellato dal registro il `ServiceReference` ottenuto in precedenza diventa inconsistente.

Processo di consumo del servizio

L'ultima fase è ovviamente il consumo del servizio, cioè la fase in cui il produttore fornisce il servizio e il cliente lo utilizza. Questa fase ha inizio con la richiesta da parte del cliente di ottenere il servizio effettivo, che viene eseguita utilizzando il metodo del `BundleContext`

```
Object getService(ServiceReference ref)
```

Il metodo ha come parametro il `ServiceReference` ottenuto durante la ricerca e restituisce un generico `Object`. La prassi è quella di eseguire un `cast` esplicito da `Object` all'interfaccia Java del servizio così da poterne utilizzare i metodi. In altre parole, il bundle che offre il servizio ne fornisce un'istanza al cliente. Le caratteristiche dell'oggetto servizio dipendono da come è stato registrato, in particolare tutti i clienti potrebbero ottenere la stessa istanza, oppure potrebbero essere forniti servizi personalizzati (si veda [Service Tracker](#)).

Rilascio di un servizio

Quando il bundle richiedente ha terminato di usare il servizio, può eseguire un'operazione di rilascio, allo scopo di migliorare l'efficienza rilasciando le risorse impegnate. Questa operazione si esegue tramite il metodo del `BundleContext`,

```
ungetService(ServiceReference ref)  
in cui il parametro è il ServiceReference ottenuto durante la ricerca del servizio.
```

È importante notare che questo metodo non libera immediatamente le risorse, ma si limita a decrementare un contatore relativo all'uso di un certo servizio da parte di un certo bundle. Quando il contatore raggiunge lo `0`, il bundle fornitore sa che quel bundle non ha più bisogno del servizio e può eventualmente liberare delle risorse o terminarsi. In altri termini l'unico scopo di questa operazione è eliminare la dipendenza tra il bundle che realizza il servizio e quello che lo usa.

Per questa ragione occorre molta cautela, come mostra il seguente stralcio di codice.

```
ServiceReference ref =  
    context.getServiceReference( IService.class.getName() );  
if( ref != null ){  
    IService s1 = IService context.getService( ref );  
    IService s2 = IService context.getService( ref );  
    //primo rilascio del servizio  
    context.ungetService( ref );  
    //secondo rilascio del servizio  
    context.ungetService( ref );  
}
```

Il programma esegue una ricerca per trovare dei servizi di tipo `IService`. Due chiamate successive (`getService`) allo stesso servizio da parte dello stesso bundle, forniscono lo stesso oggetto; in altri termini l'istanza del servizio è unica per ciascun richiedente (è comunque possibile fornire istanze diverse a bundle diversi). Le operazioni di `ungetService` non hanno alcuna influenza apparente sul bundle che richiede il servizio, infatti sia dopo una che dopo due chiamate di questo metodo il risultato sembra essere lo stesso.

In effetti, l'oggetto servizio non può essere distrutto dal garbage collector di Java fino all'annullamento di tutti i suoi riferimenti, per cui rimane ancora utilizzabile. Purtroppo, una volta eseguito il rilascio del servizio, il bundle fornitore del servizio non ha più legami con il bundle richiedente e potrebbe tranquillamente cancellare il servizio dal `Service Register`, terminare o essere disinstallato. Questo può diventare un problema soprattutto se l'oggetto servizio esegue delle callback ad altri oggetti istanziati dal bundle fornitore, o se necessita di risorse private di quel bundle, che potrebbero non essere più disponibili. In altre parole, dopo avere eseguito il rilascio di un servizio, il riferimento a quel servizio è da considerarsi inconsistente e non deve essere più utilizzato. In ogni caso, dopo l'esecuzione del metodo `stop`, le risorse relative ai servizi vengono comunque rilasciate automaticamente.

Service Dependency

Il `Service Layer` si occupa di fornire il supporto per la gestione delle `Service Dependency`. Quest'ultimo tipo di dipendenze può essere a sua volta classificato in due sotto tipi:

- Dipendenza Bundle-to-Service
- Dipendenza Service-to-Service

Dipendenza Bundle-to-Service

Un oggetto di un bundle può richiedere i servizi offerti da un altro bundle. In questo caso si genera una dipendenza di tipo dinamico, in quanto gestita dai bundle a run-time e lascia, in quanto collegata alla sola interfaccia del servizio, che potrebbe essere implementata da qualsiasi bundle. Tale dipendenza è detta **Bundle-to-Service**. Un bundle che vuole utilizzare un servizio deve conoscere solo la sua interfaccia: la ricerca dei fornitori di quel servizio avviene a run-time, senza nessun intervento garantista da parte del framework.

Il vantaggio principale di questa scelta sta nel suo dinamismo: i bundle possono essere messi in esecuzione anche se al momento non sono disponibili tutti i servizi di cui necessitano. Questo è particolarmente utile nel caso di servizi non indispensabili (ad esempio la mancanza del servizio di **log**, per tenere traccia delle operazioni eseguite, potrebbe non perturbare il resto delle operazioni del bundle), e di servizi che vengono attivati solo successivamente alla messa in opera del sistema, come ad esempio l'inserimento di un nuovo dispositivo. Lo svantaggio principale di questo tipo di dipendenze consiste nell'elevato sforzo computazionale richiesto ai bundle che devono occuparsi di gestire le fasi di ricerca dei servizi, la possibilità di non trovarne e soprattutto la possibilità che essi, da un momento all'altro, non siano più disponibili.

Dipendenza Service-to-Service

Questo tipo di dipendenza è un caso particolare del precedente ed è descritto nella figura. Si supponga che un bundle **B1** richieda un servizio **S2** che a run-time è fornito dal bundle **B2**. Il servizio **S2** è implementato da un oggetto **O2** che a sua volta necessita di un servizio **S3** (fornito da un bundle **B3**). In questo caso il bundle **B1** risulta dipendere da entrambi i servizi, in quanto la mancanza di **S3** non gli permetterebbe di ottenere il servizio **S2**. La dipendenza tra due servizi è detta **Service-to-Service**, ed è la più difficile da gestire, in quanto non può essere nota a priori. Ogni bundle ha infatti la visibilità limitata al primo servizio (**S2** nell'esempio) e OSGi non fornisce alcun modo per scoprire se questo dipende a sua volta da altri servizi.

Figure 1. Esempio di dipendenza Service-to-Service.

In un ambiente dinamico come OSGi, in cui in ogni istante può cambiare il numero di bundle (quindi di servizi) in esecuzione, la gestione delle Service Dependency può diventare troppo onerosa, per cui sono state pensate diverse soluzioni a questo problema, tra cui

- Service Tracker
- Service Binder

Service Tracker

OSGi definisce una utility, il **Service Tracker**, allo scopo di tenere traccia di registrazioni, modifiche e cancellazioni di servizi sul **Service Register**.

Tale utility è composta da una classe **ServiceTracker** e un'interfaccia **ServiceTrackerCustomizer**, ed è caratterizzata da leggerezza e semplicità d'uso. Il suo compito è quello di permettere la creazione di una lista di servizi da monitorare, mettersi in ascolto dei relativi **ServiceEvent** e gestire tali eventi in maniera personalizzabile. La classe **ServiceTracker** ha tre diversi costruttori che permettono di specificare come parametri, oltre al **BundleContext** e ad un'implementazione

dell'interfaccia `ServiceTrackerCustomizer`, un filtro, una classe o una `ServiceReference`, in modo da permettere la più vasta scelta di servizi da monitorare. L'utility comincia a monitorare i servizi solo dopo l'esecuzione del metodo `open()`, e termina il compito dopo l'esecuzione del metodo `close()`.

Una volta attivata, essa fornisce metodi per l'acquisizione di servizi già registrati (`getService()`) o per porsi in attesa di servizi che non lo sono ancora (`waitForService(long)`); in quest'ultimo caso occorre impostare un `timeout`.

La personalizzazione del comportamento del `ServiceTracker` avviene creando un implementazione dell'interfaccia `ServiceTrackerCustomizer`, in genere estendendo la classe `ServiceTracker` e ridefinendone i metodi di risposta agli eventi. Così facendo è possibile personalizzare i comportamento in risposta all'arrivo, alla modifica, o alla cancellazione di un nuovo servizio tra quelli impostati nel `ServiceTracker`.

Si noti che la ridefinizione di questi metodi potrebbe essere utile per una selezione più mirata dei servizi. Ad esempio all'arrivo della notifica di registrazione di un nuovo servizio, la ridefinizione del metodo `addingService()` potrebbe analizzare dettagli aggiuntivi su quel servizio (come scoprire il bundle che lo implementa) per decidere se è compatibile con quanto richiesto oppure no.

Service Binder

Un altro approccio al problema della gestione delle dipendenze è quello proposto da H.Cervantes [CH04] con la realizzazione di **Service Binder**. In questo approccio si è cercato di estrarre la gestione delle dipendenze dai bundle e trasferirla ad un ambiente di esecuzione, creando un nuovo livello (di fatto un nuovo framework) sopra al framework OSGi. Il nuovo ambiente è realizzato installando il bundle `ServiceBinder` sul framework OSGi, come mostra la figura.

Figure 1. Environment realizzato dal Service Binder

I componenti in esecuzione in questo nuovo ambiente sono chiamati **Service Component** e hanno al loro interno un descrittore XML che ne elenca le caratteristiche. La figura mostra lo schema logico di questi componenti. Ognuno di essi dichiara l'insieme delle interfacce di servizio che fornisce e che richiede assieme alle sue proprietà. Questi componenti, inoltre, espongono un'interfaccia di controllo (`ControlInterface`) che serve per poter implementare il pattern **Inversion Of Control** (si veda xref href="..../paradigmi/InversioneControllo.dita"/>), cioè per permettere al nuovo ambiente di gestire il life cycle delle istanze.

Figure 2. Schema logico di un Service-Component.

Durante l'esecuzione, una istanza di un **Service Component** implementa i servizi forniti ed è connessa alle altre istanze per creare un'applicazione. Le proprietà dei servizi identificano l'istanza e sono usate quando tali servizi sono pubblicati sul `Service Register` OSGi.

Un esempio di descrittore XML per **Service Binder** può essere il seguente:

```
<?xmlversion="1.0" encoding="UTF??8"?>
<bundle>
  <componentclass="org.simpleclient.impl.ServiceImpl">
    <providesservice="org.simpleclient.interfaces.
```

```

        SimpleClientServiceA" />
<providesservice="org.simpleclient.interfaces.
                    SimpleClientServiceB" />
<propertyname="provider" value="Beanome.org" type="string" />
<requires
    service="org.simpleservice.interfaces.SimpleService"
    filter="(version=*)"
    cardinality="1..n"
    policy="static"
    bind??method="setServiceReference"
    unbind??method="unsetServiceReference"
/
</component>
</bundle>
```

Si notino in particolare le varie proprietà che possono essere specificate per un servizio richiesto. La proprietà **filter** specifica un filtro in accordo con la strategia di ricerca di servizi OSGi, **policy** permette di distinguere tra una politica associativa statica (in cui il servizio non può essere cambiato) o dinamica (in cui un servizio può essere sostituito con un altro).

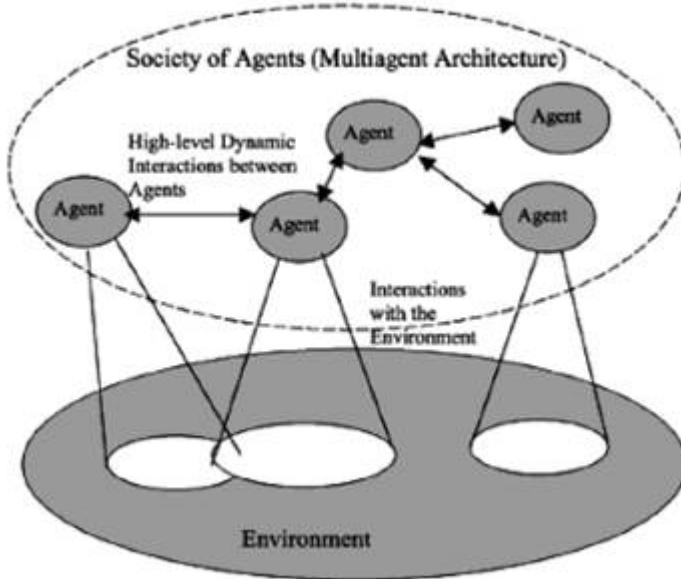
Le ultime due proprietà descrivono i metodi da eseguire durante l'arrivo di un nuovo servizio o la sua dipartita. La proprietà **cardinality** specifica la cardinalità del servizio richiesto (i cui valori possono essere **0..1**, **0..n**, **1..1**, **1..n**). Il primo valore di questa coppia rappresenta l'opzionalità del servizio (**0** se ne può fare a meno, **1** se è indispensabile), mentre il secondo valore rappresenta la molteplicità (**1** una sola istanza di servizio, **n** diverse istanze). Per approfondimenti su questo sistema si veda [CH04b].

Un mondo ad agenti

Se consideriamo la definizione di agente data in precedenza (si veda [Gli agenti](#)) è chiaro che un ***Multi Agent System (MAS)*** non può essere semplicemente ricondotto ad un gruppo di agenti che interagiscono tra loro. La completa modellazione di un **MAS** richiede di porre attenzione sia all'ambiente dove il **MAS** e gli agenti che lo costituiscono vivono, sia sulla società formata da un gruppo di agenti con obiettivi comuni. L'ambiente e la società di agenti saranno analizzati in dettaglio nelle sezioni seguenti che riassumiamo qui brevemente.

La modellazione dell'**ambiente** implica l'identificazione delle sue caratteristiche di base, le risorse che si possono trovare nell'ambiente e il modo in cui gli agenti possono interagire con l'ambiente stesso. La modellazione delle **società** di agenti invece implica l'identificazione di tutte le regole che dovrebbero guidare l'evoluzione attesa del **MAS** e dei vari ruoli che gli agenti possono giocare all'interno della società.

Figure 1. Architettura di un sistema multi-agente



Queste considerazioni portano ad una caratterizzazione molto generale dell'architettura di un MAS riportata nella figura qui sopra (tratta da [Zambonelli04]) e che differisce totalmente dalle architetture proposte negli approcci tradizionali dell'ingegneria del software.

Verso un nuovo paradigma

Le architetture tradizionali promosse dall'approccio orientato agli oggetti e riportate nella figura ([Zambonelli04]) promuovono una visione di entità "funzionali" o "orientate al servizio" che influenzano direttamente il modo in cui i sistemi software vengono progettati architetturalmente.

Figure 1. Architettura di un sistema ad oggetti

Tipicamente la progettazione globale del sistema si basa su di una architettura piuttosto statica che deriva da una decomposizione (e modularizzazione) delle funzionalità e dei dati richiesti dal sistema per portare a termine i suoi obiettivi globali e sulla definizione delle interdipendenze tra questi moduli ([Zambonelli03b]). In particolare ([Shaw96], [Bass03]):

- gli oggetti sono usualmente considerati come fornitori di servizi, responsabili di specifiche porzioni di dati e incaricati di fornire servizi per altri oggetti (il modello "contrattuale" dello sviluppo software promuove esplicitamente questa visione)
- le interazioni tra gli oggetti sono usualmente espressione di interdipendenze: due oggetti interagiscono per accedere a servizi e dati che sono disponibili localmente
- ogni cosa in un sistema tende ad essere modellata in termini di oggetti e ogni distinzione tra attori attivi e risorse passive viene così trascurata.

In altre parole, lo sviluppo orientato agli oggetti da una parte promuove l'incapsulamento dei dati e delle funzionalità oltre che il concetto di interazione orientata alle funzioni, dall'altra parte tende a trascurare la modellazione e l'incapsulamento del controllo dell'esecuzione. Viene assunta qualche sorta di "controllo globale" sull'insieme delle attività del sistema (per esempio la presenza di un singolo flusso di esecuzione o un insieme limitato di flussi di esecuzione controllabili e globalmente sincronizzati).

Assumere e/o imporre un tale tipo di controllo potrebbe non essere realizzabile nei sistemi complessi: piuttosto che incorrere in rischi di perdita di controllo, una soluzione migliore potrebbe essere quella di delegare esplicitamente il controllo dell'esecuzione ai componenti del sistema come avviene per esempio nei MAS. Infatti:

- delegare il controllo a componenti autonomi può essere considerato come una nuova dimensione di modularità e encapsulamento. Quando le entità possono encapsulare il controllo, oltre che i dati e gli algoritmi, essi possono meglio gestire le dinamicità degli ambienti complessi (le contingenze locali possono essere gestite localmente dai componenti) e si possono ridurre le interdipendenze tra i componenti limitando l'esplicito trasferimento del controllo dell'esecuzione. Questo porta ad amplificare la separazione tra le due dimensioni di progettazione: il livello-componente (cioè quello intra-agente) e il livello-sistema (cioè quello inter-agente);
- la dinamicità e l'apertura degli scenari applicativi rende impossibile conoscere a priori tutte le potenziali interdipendenze tra i componenti (per esempio quali i servizi saranno necessari ad un dato punto dell'esecuzione e con quali altri componenti sarà necessario interagire) come richiesto da una prospettiva orientata alle funzioni. I componenti autonomi a cui è stato delegato il controllo possono essere arricchiti con abilità sociali sofisticate, che sono la capacità di prendere decisioni riguardo allo scope e alla natura delle loro interazioni e di iniziare interazioni in modo flessibile (per esempio attraverso la negoziazione dei servizi e delle informazioni);
- per i sistemi complessi una chiara distinzione tra gli attori attivi (autonomi e con il proprio flusso di controllo) del sistema e le risorse passive (oggetti passivi senza controllo autonomo) potrebbe fornire un modello semplificato del problema. Infatti, i componenti software di una applicazione spesso hanno una controparte nel mondo reale che può essere attiva o passiva e di conseguenza è un sistema di questo tipo viene modellato in modo più adatto in termini di entità attive (agenti) e passive (risorse ambientali).

Cercare di arricchire gli approcci più convenzionali con nuove proprietà e caratteristiche per far fronte alle nuove necessità può portare ad introdurre una pericolosa discrepanza tra il livello di astrazione adottato e il livello concettuale nel quale devono essere risolti i problemi applicativi. In parole semplici, oggetti e componenti sono ad un livello di astrazione troppo basso per affrontare la complessità che i sistemi software richiedono, e mancano di importanti concetti come l'autonomia, l'orientamento ai compiti da svolgere, la collocazione in un ambiente e la capacità di interagire in modo flessibile.

Per esempio, approcci basati sugli oggetti o sui componenti non sono in grado di supportare la progettazione di algoritmi di negoziazione per governare le interazioni, e non forniscono nessun supporto su come mantenere un bilanciamento tra il comportamento reattivo e quello proattivo in situazioni complesse e dinamiche. Questo forza a costruire le applicazioni adottando una prospettiva orientata alle funzionalità e porta o alla costruzione di architetture statiche o alla necessità di adottare complesse infrastrutture per la gestione delle dinamiche, delle riconfigurazioni e per il supporto della negoziazione di risorse e compiti.

Riassumendo, la computazione basata sugli agenti promuove un livello di astrazione più adatto per gli scenari moderni e che risulta più appropriato per la costruzione di sistemi flessibili, altamente modulari e robusti qualunque sia la tecnologia veramente utilizzata per costruire gli agenti. Questo ci porta a considerare la computazione basata sugli agenti come un nuovo possibile paradigma di ingegnerizzazione del software.

Società di agenti

In accordo a ricerche recenti (Catelfranchi, [Carabelea05], [Carabelea05]) la socialità presuppone la presenza di agenti. Ad un livello molto base un agente è una entità capace di agire, cioè di produrre qualche effetto causale e qualche cambiamento nel suo ambiente. Ovviamente questa definizione così generale non è sufficiente a giustificare la socialità. Ad un livello più complesso è necessario avere un insieme di agenti. Gli agenti sono entità individuali con abilità sociali, essi hanno una rappresentazione del mondo esterno e una limitata abilità di percepirla e cambiarla, tipicamente fanno affidamento su altri agenti per tutto ciò che ricade al di fuori del loro scope di conoscenza.

Gli agenti possono quindi essere pensati come appartenenti ad una società: il comportamento di un agente è spesso incomprensibile al di fuori della sua struttura sociale. Per esempio il comportamento di un agente che partecipa come compratore ad un'asta è difficile da essere spiegato al di fuori del contesto dell'asta stessa e delle regole che la governano. Allo stesso modo il comportamento di una società di agenti non può essere meramente espresso come semplice somma dei comportamenti degli agenti che la compongono. Infatti le regole che governano un'asta insieme al comportamento dei singoli agenti partecipanti portano ad un comportamento globale che non può essere ridotto alla mera somma dei comportamenti individuali. Le regole sociali quindi "imbrigliano" le interazioni degli agenti e guidano il comportamento globale di una società verso la realizzazione dei suoi obiettivi globali.

Quindi, la metafora più attraente per la modellazione dei MAS sembra essere quella della società umana (o organizzazione umana) in cui [Wooldridge00]:

- Un sistema software è concepito come una istanziazione computazionale di un gruppo (potenzialmente aperto) di individui autonomi che interagiscono tra loro (agenti).
- Ogni agente può essere visto come giocatore di uno o più specifici ruoli: esso ha un insieme di incarichi e obiettivi parziali ben definiti nel contesto dell'intero sistema ed è responsabile del loro compimento che deve portare a termine in modo autonomo. Questi obiettivi parziali possono essere sia altruistici (che contribuiscono ad un qualche obiettivo sociale) o opportunistici (per agenti che badano ai propri interessi).
- Le interazioni non sono solo una mera espressione di interdipendenze e vengono piuttosto viste come mezzo per gli agenti per portare a termine i loro ruoli nel sistema. Inoltre le interazioni sono chiaramente identificate e localizzate nella definizione del ruolo stesso, ed aiutano a caratterizzare la struttura totale della società e la posizione del ruolo in essa.
- L'evoluzione delle attività nella società, derivando dall'esecuzione autonoma degli agenti e dalle loro interazioni, determina il conseguimento
 - degli obiettivi dell'applicazione
 - di un obiettivo globale identificato a priori (come per esempio in un sistema di gestione di workflow gli agenti altruistici contribuiscono alla realizzazione di un progetto cooperativo)
 - di un obiettivo collegato alla soddisfazione di obiettivo individuali (come per esempio in un'asta mediata il cui scopo è quello di soddisfare le necessità di agenti venditori e compratori)
 - di entrambi (come per esempio in una azienda Web che utilizza un marketplace virtuale al fine di migliorare l'efficienza delle vendite).

La metafora organizzazionale -- oltre ad essere naturale per gli sviluppatori umani che sono continuamente immersi in una varietà di insiemi organizzazionali e aperti alla

possibilità di riusare una grande varietà di studi ed esperienze legate alle organizzazione del mondo reale -- appare appropriata per un grande range di sistemi software. Da una parte, alcuni sistemi trattano il controllo e il supporto delle attività di qualche organizzazione del mondo reale (per esempio sistemi di controllo manifatturiero, gestione di workflow e marketplace elettronici). Inoltre una progettazione basata sulle organizzazioni potrebbe ridurre la distanza concettuale tra i sistemi software e i sistemi del mondo reale che questi supportano. Dall'altra parte, altri sistemi software anche se non sono associati ad una pre-esistente organizzazione del mondo reale possono trattare con problemi per i quali le organizzazioni umane potrebbero essere una fruttuosa sorgente di ispirazione visto che hanno già mostrato di produrre soluzioni effettive (per esempio la condivisione delle risorse, l'assegnamento dei task e la negoziazione dei servizi).

Più in generale, ogni volta che un sistema software è complesso abbastanza da giustificare un approccio basato sugli agenti ed ancora richiede un significativo grado di predicitività e affidabilità, la metafora organizzazionale potrebbe essere la più appropriata. Infatti, affidandosi agli agenti che giocano ruoli ben definiti e interagiscono tra loro in rispetto a pattern istituzionali, la metafora organizzazionale promuove un controllo sia ad un micro-livello (il livello degli agenti) che ad un macro-livello (il livello del sistema) sulla progettazione e sulla comprensione del comportamento totale del sistema.

L'ambiente

Tradizionalmente l'ambiente del **MAS** è stato considerato semplicemente come un contesto di deployment dove gli agenti erano immersi, esso per esempio include le infrastrutture di comunicazione, la topologia della rete e le risorse fisiche disponibili. In questo caso l'ambiente del **MAS** viene considerato semplicemente come un output della fase di analisi del sistema ed i progettisti sono in un qualche modo soggetti ad esso e non possono cambiarlo. Oggigiorno però è presente un crescente riconoscimento dell'importanza del ruolo dell'ambiente che viene sempre più spesso visto come una nuova dimensione di progettazione dei **MAS** [Weyns07], [Viroli07]: esso infatti può incapsulare una significativa porzione della complessità del sistema in termini di servizi, meccanismi e responsabilità che possono sgravare gli agenti di qualche compito. Possiamo quindi pensare di dividere concettualmente l'ambiente del **MAS** in due "diversi ambienti":

- **ambiente esterno**: rappresenta il contesto di deployment dove il **MAS** andrà ad eseguire
- **ambiente interno** (ambiente degli agenti): rappresenta una porzione del **MAS** esterna agli agenti che, anche se esterna agli agenti stessi, è soggetta alla progettazione attiva

Ovviamente ai fini della progettazione di un nuovo **MAS** l'ambiente "più interessante" è quello interno perché è possibile progettarlo e quindi crearlo con le risorse e la struttura topologica che meglio possono supportare l'esecuzione degli agenti.

In fase di analisi però va considerato e modellato accuratamente anche l'ambiente esterno la cui conoscenza sarà poi necessaria per la progettazione delle interazioni che gli agenti avranno con le risorse pre-esistenti (che potremmo pensare come legacy system): è infatti impensabile che il nuovo **MAS** vada ad operare in un contesto dove non esista nessuna risorsa, è quindi bene modellare queste risorse nel modo migliore al fine di garantire poi una corretta progettazione dello spazio delle interazioni degli agenti.

Nel seguito parleremo semplicemente di ambiente facendo riferimento all'ambiente intero del **MAS**, che come detto per noi risulta il più interessante dal punto di vista ingegneristico. A tal proposito è possibile riconoscere almeno due ingredienti fondamentali per la progettazione dell'ambiente: le **astrazioni ambientali** che sono entità dell'ambiente che incapsulano una o più funzionalità, e le **astrazioni topologiche** che sono entità che rappresentano la struttura spaziale (fisica e/o logica) dell'ambiente stesso.

Le astrazioni ambientali. L'astrazione di agente non è la sola astrazione a popolare il **MAS** [Viroli07]: l'ambiente del **MAS** può essere visto come una composizione di "blocchi di costruzione" chiamati **astrazioni ambientali**. Una astrazione ambientale è una entità dell'ambiente del **MAS** che incapsula una o più funzionalità o servizi per agli agenti. L'agente percepisce queste risorse nell'ambiente, conosce le funzionalità che esse mettono a disposizione e può interagire con esse al fine di portare a termine i propri obiettivi (sociali o individuali). Dal punto di vista della progettazione, le astrazioni ambientali possono essere viste come i luoghi dove possono essere memorizzate e fatte applicare le regole, le norme e le funzioni che regolamentano il comportamento sociale degli agenti.

Le astrazioni ambientali rappresentano un modo per accomunare le svariate entità che sono state proposte sia a livello concettuale che a livello implementativo da diverse ricerche svolte negli anni passati [Viroli07]. L'idea di "uniformare" queste entità attraverso una nuova astrazione nasce dal fatto che molte entità proposte condividono la stessa idea di astrazione abilitante per l'interazione degli agenti. Molte infrastrutture per agenti possono infatti essere viste come ambienti che incarnano le astrazioni ambientali (abilitanti per le interazioni) per le applicazioni: per esempio possiamo citare gli spazi di tuple di Linda [Gelernter85], [Gelernter92], i centri di tuple di TuCSoN [Omicini99], i co-field di TOTA [Mamei05], etc.

Le astrazioni topologiche. Oltre a considerare le diverse astrazioni che popolano l'ambiente del **MAS**, è opportuno considerare anche la sua struttura fisica o topologia. In particolare possiamo considerare le **astrazioni topologiche** (astrazioni che supportano la topologia come entità di prima classe) come astrazioni per la progettazione del **MAS** [Molesini08]. Come definizione generale, possiamo vedere la topologia come una collezione di insiemi di "quartieri/ vicinanze" (neighbourhood in inglese) che forniscono una nozione strutturata di località per il **MAS**, dal punto di vista locale sino a quello dell'intero sistema.

Quando consideriamo il **MAS** come una composizione di agenti situati e astrazioni ambientali viene naturale pensare la topologia del **MAS** come una collezione di insiemi di agenti e astrazioni ambientali. Questo influisce ovviamente su nozioni come la **visibilità** tra gli agenti (se l'agente può vedere e comunicare con un altro agente), tra gli agenti e le astrazioni ambientali (se l'agente può vedere e interagire con una astrazione ambientale), e la **mobilità** di un agente (quali nuovi "quartieri" un agente può raggiungere).

Questo concetto è molto importante nel **MAS** poichè la visibilità e l'accessibilità di altre entità identificano strettamente quali obiettivi possono essere delegati (a altri agenti) e quali servizi possono essere ottenuti (dalle astrazioni ambientali), infine definiscono la capacità degli agenti di portare a termine i loro obiettivi. In molti casi il concetto di topologia è legato alla struttura fisica del contesto di deployment includendo sia la topologia della rete che quella dell'ambiente fisico come per esempio nel caso di ambienti per sistemi robotici, ma in generale spesso dal punto di vista progettuale è conveniente considerare una nozione virtuale dello spazio.

Agent UML

Agent UML (AUML) [], una estensione del linguaggio UML, nasce per supportare i progettisti dei sistemi basati sugli agenti nei primi anni del 2000. AUML è stato sviluppato nel contesto di FIPA (*Foundation for Intelligent Physical Agent*) che è stata recentemente inglobata dalla IEEE ed ora promuove la standardizzazione delle tecnologie basate sugli agenti e l'interoperabilità tra gli standard ad agenti promossi e le altre tecnologie che attualmente risultano essere standard nel mondo industriale. La filosofia generale dietro allo sviluppo di AUML è quella di estendere/riusare porzioni di UML, e ove questo non sia possibile gli sviluppatori propongono di usare qualche altro formalismo o di creare uno nuovo.

Il lavoro di estensione di UML è iniziato intorno al 2000 e si è concentrato inizialmente sulla definizione dei requisiti necessari al supporto e alla modellazione degli agenti e dei sistemi multi-agente. Successivamente è stata approfonditamente studiata la versione UML1, che all'epoca era lo standard, e sono state fatte le prime proposte di estensione.

Le principali critiche mosse ad UML1 riguardavano l'incapacità di modellare propriamente:

- le *interazioni* tra agenti e i relativi protocolli adottati,
- il concetto di *ruolo* giocato dagli agenti e i cambiamenti di ruolo durante l'esecuzione,
- l'*architettura interna* degli agenti con particolare riferimenti agli agenti intelligenti che prevedono una base di conoscenza ed elaborati meccanismi di pianificazione delle azioni,
- il *contesto sociale* nel quale gli agenti sono immersi e le regole sociali a cui devono sottostare.

Inizialmente il Comitato tecnico di FIPA preposto alla definizione e allo sviluppo di AUML si è concentrato sull'estensione di UML per il supporto alla **modellazione delle interazioni** definendo prima di tutto il concetto di "agent interaction protocol" (AIP) come *a communication pattern, with an allowed sequence of messages between agents having different roles, constraints on the content of the messages, and a semantics that is consistent with the communicative acts (CAs) within a communication pattern.*

I **messaggi** dovevano quindi essere espressi attraverso uno specifico linguaggio di comunicazione per agenti, come per esempio FIPA-ACL (vedi nel seguito) che ne vincolava la semantica, mentre i protocolli dovevano vincolare i parametri, l'ordine e il tipo dei messaggi scambiati.

Attualmente il lavoro del Comitato di programma per lo sviluppo di AUML risulta fermo in quanto parte dei requisiti identificati per la modellazione dei sistemi ad agenti sono stati recepiti dall'OMG e introdotti nella versione UML2 come per esempio alcuni requisiti legati alla modellazione delle interazioni.

Nel seguito presenteremo brevemente la specifica FIPA per la rappresentazione dei protocolli di interazione e la specifica FIPA per il linguaggio di comunicazione.

FIPA Interaction

I diagrammi di sequenza di [Agent UML](#) sono stati adottati inizialmente da [FIPA](#) per esprimere i protocolli di interazione tra agenti. Nei diagrammi vengono considerate due parti principali: il frame che delimita il diagramma di sequenza e il flusso dei messaggi tra i ruoli che viene rappresentato per mezzo dell'insieme delle lifeline e dei messaggi. Da notare che mentre in un diagramma di sequenza di [UML](#) i rettangoli rappresentano oggetti, nel caso dei diagrammi di sequenza di [UML](#) i rettangoli rappresentano agenti o ruoli giocati dagli agenti.

Figure 1. AUML Frame

La keyword **sd** sta per "sequence diagram" e proviene delle specifiche di [UML2](#). Dopo la keyword tipicamente troviamo il nome del protocollo di interazione. Diversamente da quanto avviene in [UML2](#), i parametri non sono seguono il nome del protocollo ma sono posti come commento esterno al diagramma di sequenza (vedi figura sottostante). I parametri sono prefissati attraverso lo stereotipo **parameters**, inoltre in [AUML](#) tali parametri non sono legati ai parametri dei messaggi come avviene in [UML2](#) ma sono invece legati all'istanziazione dello specifico template . Per l'esempio della figura che segue il protocollo tipico dell'asta inglese è stato instanziato in un caso specifico strettamente legato ai parametri associati al diagramma: vendere l'auto ad un certo prezzo e prima di una certa data prefissata.

Figure 2. Frame con parametri

Un template di protocollo viene tipicamente denotato attraverso l'uso dello stereotipo **template** che viene posto prima del nome del protocollo stesso come mostrato in figura. Inoltre i progettisti dei protocolli devono obbligatoriamente specificare 3 tipi di parametri chiave che sono l'ontologia (keyword "ontology"), il linguaggio del contenuto ([CL](#)) e il linguaggio usato nel protocollo ([ACL](#)).

Figure 3. Template di protocollo

L'uso delle **lifelines** in [AUML](#) è diverso rispetto a quello di [UML2](#), in cui le lifeline rappresentano un partecipante individuale nell'interazione. In [AUML](#) invece è possibile rappresentare diversi agenti sulla stessa lifeline. Questo è possibile perché in [AUML](#) una lifeline tipicamente rappresenta un ruolo che può essere giocato da diversi agenti.

Ruoli multipli

[UML](#) definisce un ruolo come *a named set of behaviours possessed by a class or part participating in a particular context*. Una lifeline in una diagramma di sequenza definisce il periodo di tempo durante il quale un ruolo esiste nel contesto della specifica interazione, rappresentato da una linea tratteggiata verticale. Quando una lifetime viene creata il relativo ruolo diventa attivo nel contesto del protocollo e rimane attivo sino alla fine della lifeline.

Un agente però è capace di assumere ruoli multipli durante una interazione ([classificazione multipla](#)), così come è possibile che durante l'interazione l'agente possa cambiare ruolo ([classificazione dinamica](#)). Queste abilità vengono rappresentate mediante una serie di diagrammi di sequenza attraverso una connessione tra il corrente ruolo giocato dall'agente e il successi.

I ruoli multipli invece implicano che gli agenti abbiano diverse partecipazioni nella stessa interazione: se un messaggio è mandato sia ad un ruolo m che ad un ruolo n, un agente

che gioca entrambi i ruoli riceverà entrambi i messaggi, uno per il ruolo m e uno per il ruolo n. La classificazione dinamica si riferisce all'abilità di cambiare ruolo durante l'interazione: se ad uno specifico momento t l'agente ha il ruolo m e cambia il ruolo m da n a t+1 significa che l'agente a t+1 riceverà messaggi solo per il ruolo n.

Se l'agente stava giocando diversi ruoli oltre ad m esso lascia solo il ruolo m. Inoltre la classificazione multipla e quella dinamica sono operazioni atomiche: non è possibile che un messaggio possa essere spedito o ricevuto durante il cambiamento del ruolo. Le dinamiche dei ruoli comunque non sono direttamente catturate nei diagrammi di sequenza, esse vengono propriamente rappresentate nei **diagrammi di ruolo** questo per risolvere le problematiche relative a vincoli che possono sussistere durante il cambiamento di un ruolo.

Infine, uno specifico ruolo può assumere comportamenti differenti a seconda del gruppo in cui si trova per questo motivo i progettisti possono adottare il nome del gruppo come suffisso al nome ruolo per rendere chiara la distinzione. Un gruppo viene visto come un insieme di agenti che sono relazionati tra loro attraverso i ruoli che giocano.

Figure 1. Lifelines

La cardinalità del ruolo è aggiunta alle lifeline per mostrare i numero di agenti che giocano lo stesso ruolo e sono coinvolti nell'interazione. Ai progettisti sono offerte diverse opzioni per rappresentare la cardinalità:

1. il progettista conosce l'esatto numero degli agenti coinvolti nel ruolo
2. il progettista conosce il range e può scriverne i limiti
3. il numero degli agenti non è conosciuto e in questo caso la cardinalità può essere formulata attraverso una formula logica o una condizione

Una lifeline è composta da due elementi: una label presente nel rettangolo in alto della lifeline e una linea tratteggiata verticale ancorata a tale rettangolo. I ruoli sono sempre sottolineati nella label della lifeline. Le label inoltre contengono la cardinalità del ruolo per la quale esistono diversi tipi di formati:

1. n: rappresenta il numero esatto degli agenti che giocheranno il ruolo all'interno dell'interazione
2. **m op i op n** specifica il range dei valori per la cardinalità. Op si riferisce agli operatori tradizionali di confronto
3. una condizione o una formula logica

La classificazione dinamica è rappresentata nel diagramma di sequenza come una linea diretta dal ruolo corrente al nuovo ruolo con lo stereotipo **change role**, mentre per la classificazione multipla gli agenti possono aggiungere nuovi ruoli attraverso una linea diretta verso il nuovo ruolo con lo stereotipo **add role** come mostrato in figura.

Figure 2. Ruoli multipli in AUML

La figura sottostante riporta un riassunto delle possibili notazioni per i diversi tipi di lifeline introdotti.

Figure 3. Notazioni delle possibili lifeline

Le lifeline modellano la presenza degli agenti nelle interazioni attraverso la loro identità o attraverso il loro ruolo nell'interazione. La parte

- a) descrive un ruolo dell'agente;
- b) un agente con il suo identificatore e ruolo;
- c) rappresenta la forma completa della label: l'identificatore dell'agente, il ruolo e il gruppo dell'agente.
- d) descrive un ruolo e il suo gruppo senza far riferimento a nessun agente specifico;
- e) f) e g) considerano la cardinalità dell'agente. In particolare in e) è conosciuto l'esatto numero degli agenti mentre in f) viene considerato il range della cardinalità. Infine in g) la cardinalità di un agente è basata su una condizione.

Messaggi

Nel contesto di **UML2** e **AUML**, un **messaggio** definisce una particolare comunicazione tra due lifeline in un diagramma di sequenza. I mittenti e i destinatari di un messaggio possono essere sulla stessa lifeline o no. Quando la lifeline è la medesima sia per il mittente che per il destinatario occorre tener presente due situazioni:

1. Il mittente vuole ricevere il messaggio così com'è
2. Il mittente vuole essere omesso dalla lista dei riceventi

Queste due differenti situazioni devono essere distinte nella notazione del messaggio (vedi figura). Al messaggio possono altresì essere associati dei vincoli che influiscono sul suo uso. Gli agenti possono usare sia messaggi sincroni che asincroni. Il contenuto del messaggio è dato sopra alla freccia senza nessuna formalizzazione riguardo al contenuto che dipende dal tipo di linguaggio di comunicazione adottato. Visto che le lifeline tipicamente rappresentano ruoli che possono essere giocati da diversi agenti contemporaneamente è importante specificare la cardinalità. La cardinalità dei messaggi segue direttamente da quella relativa ai ruoli già presentata. Tipicamente la cardinalità deve essere specificata sia per i mittenti che per i destinatari ed è omessa solo quando è ovvia. La figura sottostante riporta un riassunto delle possibili notazioni per i diversi tipi di messaggi.

Figure 1. Notazioni dei possibili messaggi

La figura mostra le notazioni adottate nello scambio dei messaggi tra agenti. In particolare:

- a) messaggio asincrono;
- b) messaggio sincrono;
- c) messaggio con cardinalità sia per mittente che per ricevente (esatto numero di agenti coinvolti);
- d) messaggio con cardinalità rappresentata attraverso un range;
- e) messaggio con cardinalità espressa mediante una condizione;
- f) messaggio per un agente specifico;
- g) messaggio asincrono per la stessa lifeline dove il mittente riceve una copia del messaggio;
- h) messaggio asincrono per la stessa lifeline dove il mittente non riceve una copia del messaggio;
- i) messaggio sincrono per la stessa lifeline dove il mittente non riceve una copia del messaggio;

Le figure sottostanti mostrano due esempi di protocolli il primo con messaggi asincroni e il secondo con messaggi sincroni.

Figure 2. Protocollo con messaggi asincroni

Figure 3. Protocollo con messaggi sincroni

Vincoli

I **vincoli** influenzano il modo in cui i messaggi e il percorsi alternativi nei protocolli di interazione vengono usati. Diversi messaggi possono essere inibiti se i vincoli associati ad essi non sono soddisfatti. Tipicamente i vincoli sono usati per scegliere un percorso tra diverse possibili alternative. Due tipi principali di vincoli vengono considerati: vincoli **bloccanti** e **non-bloccanti**.

I vincoli bloccanti implicano che l'interazione rimane bloccata sino a che il vincolo non viene soddisfatto, mentre i non-bloccanti implicano che l'elemento regolato dai vincoli viene eseguito solo se il vincolo è soddisfatto. I vincoli bloccanti possono essere applicati a ruoli, a specifici agenti in un ruolo o a un insieme di ruoli. Se i vincoli bloccanti sono applicati ad un ruolo significa che tutti gli agenti che giocano tale ruolo coinvolti nell'interazione sono bloccati sino a che i vincoli non sono soddisfatti. Gli agenti sono bloccati solo per questo ruolo e possono continuare l'interazione con gli altri ruoli coinvolti.

I vincoli non-bloccanti sono tipicamente usati nelle alternative e nella selezione dei percorsi e possono anche utilizzare un caso di default chiamato **else** che viene usato se nessuno dei vincoli precedenti è soddisfatto. I vincoli possono essere definiti sia come testo in formato libero sia formalmente attraverso l'uso di **OCL**. I vincoli sono riportati vicino all'elemento su cui il vincolo deve essere applicato. Quando i vincoli sono applicati ad alternative essi sono posti sopra ad ogni alternativa. I vincoli bloccanti vengono rappresentati attraverso lo stereotipo **blocking**, mentre quelli non-bloccanti sono rappresentati tra parentesi quadre.

La figura sottostante mostra di protocolli con vincoli bloccanti e non-bloccati.

Figure 1. Protocollo vincoli

Particolari tipi di vincoli sono i **vincoli temporali** che rappresentano il tempo nei diagrammi di sequenza. Esempi sono riportati nelle figure sottostanti

Figure 2. Esempio vincoli temporali

Figure 3. Esempio vincoli temporali

Percorsi alternativi

Un protocollo con un singolo percorso tra lo stato iniziale e quello finale riduce fortemente l'autonoma degli agenti. Risulta quindi opportuno esprimere diverse alternative e lasciare gli agenti liberi di scegliere quella che risulta essere la migliore nel contesto dello stato interno di ciascun agente. Per rappresentare i percorsi alternativi **AUML** si basa sulla combinazione di frammenti (**CombinedFragment**) proposta da **UML2** le scatole che contengono i differenti percorsi dell'interazione dipendono dagli operatori dell'interazione. I diversi tipi di operatori sono:

- **Alternative**: significa che sono disponibili diverse alternative per seguire l'interazione e l'agente ne deve scegliere al massimo una per continuare. Ad ogni alternativa è associata una guardia, di conseguenza l'alternativa che viene scelta è quella cui guardia risulta vera e nessuna alternativa può essere scelta se le guardie

risultano tutte vere. In questo caso il fragment "alternative" non può essere eseguito. Se deve comunque essere eseguita almeno una alternativa il progettista deve prevedere la presenza della scelta "else" che viene eseguita se tutte le guardie risultano essere false

- **Option**: considera solo un percorso nel CombinedFragment. Se le condizioni associate a questo percorso vengono tutte valutate vere allora il percorso viene eseguito, altrimenti nulla accade e l'interazione riprende dopo questo CombinedFragment. Questo operatore può essere rappresentato come un CombinedFragment con un operatore Alternative a due percorsi: il primo contiene l'insieme dei messaggi che devono essere scambiati se le condizioni sono soddisfatte, mentre il secondo è vuoto e corrisponde alle condizioni non soddisfatte.
- **Break**: è definito in UML 2.0 come uno scenario di rottura che ferma l'esecuzione del diagramma di sequenza ed esegue lo scenario nel break CombinedFragment. L'esecuzione interrotta non può essere ripresa. Tale comportamento può essere usato per rappresentare eccezioni che fermano l'esecuzione corrente per eseguire una specifica sequenza di messaggi
- **Parallel**: rappresenta l'esecuzione parallela di percorsi differenti in un ordine. Permette ai progettisti di rappresentare l'invio concorrente di diversi messaggi.
- **Weak Sequencing**: è definito da UML 2.0 e specifica che i messaggi sulla stessa lifeline all'interno del CombinedFragment sono ordinati ma non è possibile fare assunzioni sull'ordine dei messaggi che provengono dalle differenti lifeline nel CombinedFragment. Questo operatore sembra utile nel caso dell'osservazione dei trace dei messaggi quando gli agenti sono distribuiti e non condividono lo stesso clock globale.
- **Strict Sequencing**: raffina il Weak Sequencing e assicura che tutti i messaggi nel CombinedFragment sono ordinati dall'alto al basso.
- **Negative**: descrive l'insieme dei messaggi che sono considerati non validi nell'interazione.
- **Critical Region**: rappresenta la sezione critica dei sistemi distribuiti. Permette di descrivere la sequenza dei messaggi nella regione critica che deve essere spedita automaticamente e non può essere intervallata ad altre sequenze di messaggi. Tipico esempio è dato dalle trasazioni.
- **Ignore**: definisce l'insieme dei messaggi che deve essere ignorato durante l'interazione. Un tipico esempio riguarda l'arrivo di messaggi molto in ritardo che non sono più interessanti nell'attuale stato dell'interazione.
- **Consider**: rappresenta l'opposto di Ignore e rappresenta tutti i messaggi che devono essere considerati nell'interazione.
- **Assertion**: descrive l'unica sequenza dei messaggi che può essere accettata nel CombinedFragment visto lo stato attuale dell'interazione.
- **Loop**: permette di rappresentare l'applicazione multipla di un insieme ordinato di messaggi. I progettisti possono usare lowerbound e upperbound o un'espressione booleana. Finché la condizione è soddisfatta il loop viene eseguito e i messaggi spediti e ricevuti.

Unioni di percorso

Questi operatori rappresentano separatori di percorso mentre le unioni dei percorsi sono effettuate attraverso l'operatore **Continuation** di UML2. Le figure sottostanti presentano diversi esempi di questi operatori.

Figura 1. Esempio di separazione dei percorsi

Figura 2. Continuation

Protocolli combinati

In **AUML** è possibile rappresentare protocolli **combinati** attraverso l'InteractionOccurrence di **UML2**. Un possibile tipo di combinazione è quella interleaved che viene mostrata attraverso un template di protocollo, una volta che il protocollo chiamato è terminato riprende l'esecuzione del protocollo chiamante come mostrato in figura. Il protocollo interno non ha nessuna conoscenza di quello esterno. Il fragment del protocollo interno adotta la keyword "ref".

Figure 1. Protocolli combinati

Talvolta però è necessario che i due protocolli possano interagire tra loro: il protocollo esterno manda un messaggio a quello interno passando diverse informazioni. Alla fine dell'esecuzione del protocollo interno questo potrebbe dover passare ulteriori informazioni al protocollo esterno. Questo tipo di interazione tra protocolli è gestito in **AUML** attraverso i **Gate** di **UML 2.0**. Esistono due tipi di interazione tra protocolli: interazioni bloccanti e interazioni non-bloccanti. Nel primo caso il protocollo esterno attende la terminazione di quello interno, nell'altro il protocollo esterno riprende la sua esecuzione senza attendere il completamento di quello interno. In figura è riportato un esempio.

Figure 2. Esempio di interazione tra protocolli

FIPA ACL

Lo scopo di **FIPA ACL** è quello di fornire una logica conversazionale agli agenti in modo tale da alzare il livello semantico delle comunicazioni tra agenti rispetto a livello di comunicazione supportato dalle altre tecnologie, come per esempio le comunicazioni basate sugli eventi. A tal fine, ogni primitiva di comunicazione in **FIPAACL** (chiamata "atto di comunicazione") possiede una precisa semantica fornita attraverso pre- e post-condizioni espresse attraverso la logica modale del primo ordine. Con questa semantica l'agente è in grado di esprimere le sue attitudini personali (credenze, incertezze, scelte, intenzioni,...) riguardo alla sua conoscenza acquisita piuttosto che semantica della conoscenza stessa.

Un messaggio **FIPA ACL** contiene uno o più parametri il cui numero varia a seconda dello specifico contesto della comunicazione. L'unico parametro che è obbligatorio in tutti i messaggi è il parametro **performative** che identifica l'atto della comunicazione, anche se è buona norma che i messaggi contengano anche parametri come **sender**, **receiver** e **content**. L'elenco completo dei parametri è riportato nella tabella sottostante.

Parametro	Categoria del Parametro	Descrizione
performative	tipo di atto comunicativo	denota il tipo di atto comunicativo
sender	partecipante della comunicazione	denota l'identità del mittente del messaggio, cioè il nome dell'agente
receiver	partecipante della comunicazione	denota il ricevente del messaggio che può essere rappresentato da uno o più agenti nel caso di un multi-cast

reply-to	partecipante della comunicazione	denota a chi devono essere spedite le risposte al messaggio
content	contenuto del messaggio	denota il contenuto del messaggio, cioè l'oggetto dell'azione
language	descrizione del contenuto	specifica il linguaggio in cui il contenuto è espresso
encoding	descrizione del contenuto	specifica il tipo di encoding supportato dal linguaggio adottato
ontology	descrizione del contenuto	specifica quali ontologie sono usate al fine di fornire il giusto significato semantico ai simboli del linguaggio
protocol	controllo della conversazione	specifica il protocollo di interazione che il mittente sta usando
conversation-id	controllo della conversazione	usato per specificare la sequenza dei messaggi all'interno di una conversazione
reply-with	controllo della conversazione	introduce una espressione che dovrà essere usata nella risposta per identificare questo messaggio
in-reply-to	controllo della conversazione	denota una espressione che riferisce a un'azione precedente di cui questo messaggio rappresenta la risposta
reply-by	controllo della conversazione	specifica una espressione di tempo o data che indica l'ultima volta nella quale il mittente ha voluto ricevere una risposta

La tabella sottostante riporta invece i tipi di atti comunicativi ammessi, cioè i valori che può assumere il parametro performative.

Tipo di atto	Descrizione	Risposta a
accept-proposal	rappresenta l'accettazione ad una proposta inviata in precedenza	propose
agree	atto di accettazione all'esecuzione un'azione	request
cancek	permette ad un agente i di informare l'agente j che i non è più interessato che j esegua l'azione che aveva richiesto in precedenza. J è comunque libero di eseguire o meno l'azione	-
cfp	indica un processo di negoziazione attraverso una richiesta di proposte all'esecuzione di una data azione	risposta attraverso una serie di

		proposte che soddisfano le condizioni espresse nel messaggio
confirm	il mittente informa il ricevente che la proposizione sulla quale il ricevente aveva dubbi è vera	-
disconfirm	il mittente informa il ricevente che la proposizione sulla quale il ricevente aveva dubbi è falsa	-
failure	specifica ad un altro agente che una azione è stata tentata, ma il tentativo è fallito	-
inform	il mittente informa il destinatario che una data proposizione è vera	-
inform-if	è una abbreviazione per informare se una proposizione è creduta vera o meno	-
inform-ref	permette al mittente di informare il destinatario riguardo ad un oggetto che il mittente crede corrispondere ad un descrittore	-
not-understood	permette di informare il destinatario che il mittente percepisce che il destinatario ha fatto una qualche azione ma che il mittente non è in grado di capire.	-
propagate	il mittente vuole che il destinatario recuperi gli identificativi degli agenti a cui il messaggio è veramente stato indirizzato e spedisca a questi il messaggio come se fosse stato inviato direttamente dal mittente	-
propose	l'azione di sottoporre una proposta di esecuzione di una particolare azione date certe precondizioni	cfp
proxy	il mittente desidera che il ricevente selezioni gli agenti denotati da un descrittore e che mandi a questi il messaggio contenuto	-
query-if	l'azione di richiedere ad un altro agente se una proposizione è vera o falsa	confirm o disconfirm
query-ref	l'azione di richiedere ad un altro agente riguardo all'oggetto specificato attraverso una espressione referenziale	inform-ref
refuse	rifiuto di eseguire una azione e spiegazione del motivo	request

reject-proposal	rifiuto di eseguire una determinata azione nel contesto di una negoziazione	cfp
request	il mittente richiede al ricevente di eseguire una specifica azione	-
request-when	il mittente informa il ricevente che una data azione deve essere eseguita appena una data precondizione si verifica	-
request-whenever	il mittente informa il ricevente che una data azione deve essere eseguita appena una data precondizione si verifica, quando la precondizione diventa falsa tale azione deve essere rieseguita non appena la precondizione torna ad essere vera	-
subscribe	il mittente richiede di essere notificato ogni volta che l'oggetto denotato cambia stato	-

Model driven software development

Per gestire la fase di riorganizzazione di un framework, la documentazione del framework stesso in termini di modelli ([UML](#)) potrebbe rilevarsi fondamentale per abbattere i tempi e i costi. Ma ancor più fondamentale sarebbe la possibilità di pensare ai modelli come una [**nuova forma di codice sorgente**](#) e di disporre di generatori capaci di ricavare automaticamente dai modelli il codice eseguibile del framework. In questo caso per modificare il framework basterebbe modificare i modelli.

Le trasformazioni possono essere viste in generale come un modo per veicolare best practices e possono essere di tipo [**orizzontale**](#) (refactoring, localizing), [**verticale**](#)(dall'alto al basso) od [**obliquo**](#) (combinazioni orizzontali e verticali).

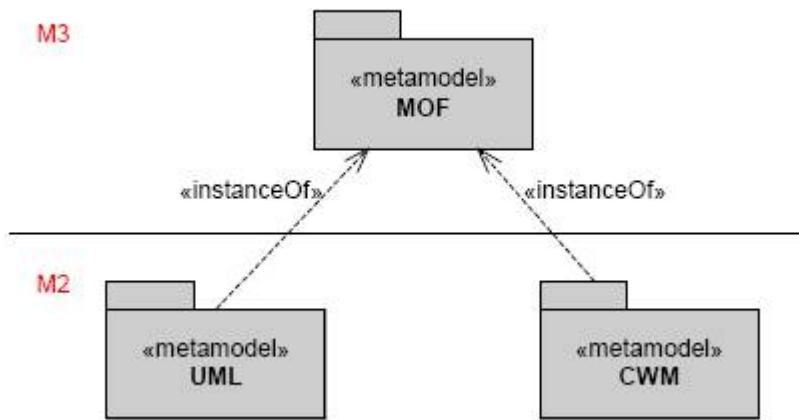
L'approccio [**MDSD**](#) ([**Model Driven Software Development**](#)) [SV05] mira a ridurre la complessità della trasformazione limitando queste alla "parte schematica" e ripetitiva relativa ad una specifica piattaforma. Il punto di partenza di questo approccio è costituito da un [**Domain Specific Modeling Language**](#) (DSL) definito attraverso l'uso di modelli formali che vengono trasformati direttamente nel codice o negli altri artefatti previsti dalla piattaforma. Ontologicamente, [**MDA**](#) è una specializzazione di [**MDSD**](#).

Al di là delle differenze, questi approcci condividono un insieme di concetti, vocaboli e strumenti che permettono di comprendere il ruolo di linguaggi come [UML](#), la loro evoluzione, gli attuali limiti e il ruolo che i modelli possono avere nei moderni processi di sviluppo software, apendo la via a una auspicabile integrazione tra metodi agili e processi di sviluppo model-driven.

UML2

Uno dei maggiori obiettivi della specifica di [**UML2**](#) è stato di allineare architetturalmente [UML](#) e le Meta Object Facility ([MOF](#)) in modo che [MOF](#) possa essere usato come metamodello di [UML](#).

Figure 1. MOF e UML



Dalla versione 2 in avanti quindi, **UML** è uno di tanti possibili linguaggi (metamodelli) definibili in **MOF**. Un altro linguaggio definito via **MOF** è **OCL** (si veda [L'Object Constraint Language](#)) che permette di esprimere vincoli sul modelli, sia a livello **M1** che **M2**. Attraverso **MOF** è possibile definire anche linguaggi non **Object Oriented**.

L'infrastructure di **UML2** è definita nella **InfrastructureLibrary**, la quale, articolata in due package (**Core** e **Profile**) definisce un "reusable metalanguage kernel" and a "metamodel extension mechanism" per **UML** (**Infrastructure specification** pag. 27); essa è usata sia a livello **M2** che **M3** in quanto riusata da **UML** e **MOF**.

UML2 e MDA

L'[AnnexB](#) della **Infrastructure specification** (pg. 201) discute come l'**UML2** sia stato concepito anche con lo scopo di dare supporto ai principali concetti della visione **MDA**.

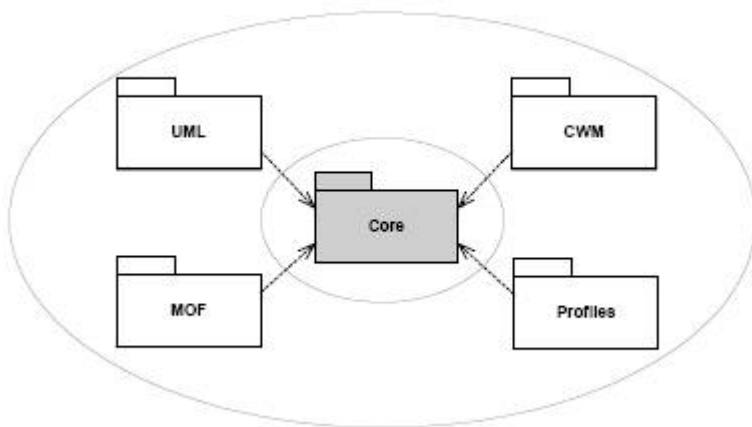
In particolare viene segnalato che **UML2**:

- fornisce una famiglia di linguaggi (si veda [UML come famiglia di linguaggi](#)). La revisione del concetto di **profilo** può promuovere la definizione di **domain specific languages**, di **piattaforme operative** (si veda [Piattaforme operative](#)) e di **metodi** (Unified Process, Agile methods, etc). I tool che implementano **MOF2** permettono di definire nuovi linguaggi come metamodelli (si veda [xref href="..//lab/agentModel.dita"/>](#));
- permette la specifica di un sistema sia da un punto di vista logico (modelli dell'analisi, platform independent models, etc.) sia da un punto di vista fisico (componenti);
- migliora la capacità espressiva per la specifica di piattaforme operative (**PSM**); ne sono esempio i micro-profili **J2EE/EJB** e **.NET/COM** definiti in **Superstructure specification - annex D**;
- supporta la trasformazione di specifiche **PIM** in **PSM** attraverso relazioni definite nella **Superstructure specification** quali **Realization**, **Refine**, **Trace**.

Il Core package

La parte centrale riusabile della **InfrastructureLibrary** è il **Core package**, definito in modo che elementi della modellazione possano essere condivisi tra **MOF** e **UML**.

Figure 1. Il Core package



Ogni model element di **UML** è anche istanza di esattamente un model element in **MOF**. Nel caso di **MOF** le metaclassi della **InfrastructureLibrary** sono usate come sono, mentre nel caso di **UML** a questi elementi sono fornite proprietà addizionali, per tenere conto di requisiti specifici.

Il **Core package** è a sua volta suddiviso (si veda [Riuso in MOF](#)) in altri packages :

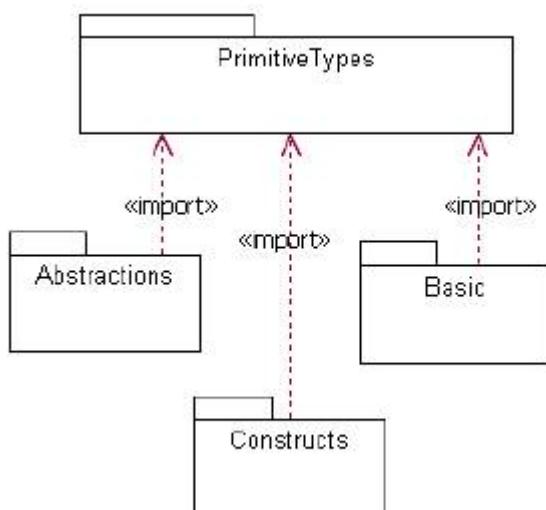


Figure 2. Core packages

- Il package **PrimitiveTypes** contiene alcuni tipi predefiniti di uso comune (**Integer**, **Boolean**, **String**, **UnlimitedNatural**).
- Il package **Abstractions** contiene principalmente metaclassi astratte suddivise in sotto-package (si veda la figura più sotto).
- Il package **Constructs** contiene metaclassi concrete che inducono a object oriented modeling (si veda la figura più sotto).
- Il package **Basic** contiene costrutti-base per la definizione di un linguaggio di modellazione minimale class-based. Questi costrutti (**Tyeps**, **Classes**, **DataTypes**, **Packages**) sono riusati per la definizione di **EMOF** (si veda [EMOF e CMOF](#)) e per la produzione di rappresentazioni **XMI** di **UML**, **MOF** e altri metamodelli basati sulla **InfrastructureLibrary**. Questi costrutti sono un sottoinsieme dei costrutti definiti in **Constructs** che ne fornisce una definizione più complessa che viene riusata in **CMOF** e nella **Superstructure specification**.

Figure 3. Core abstraction packages

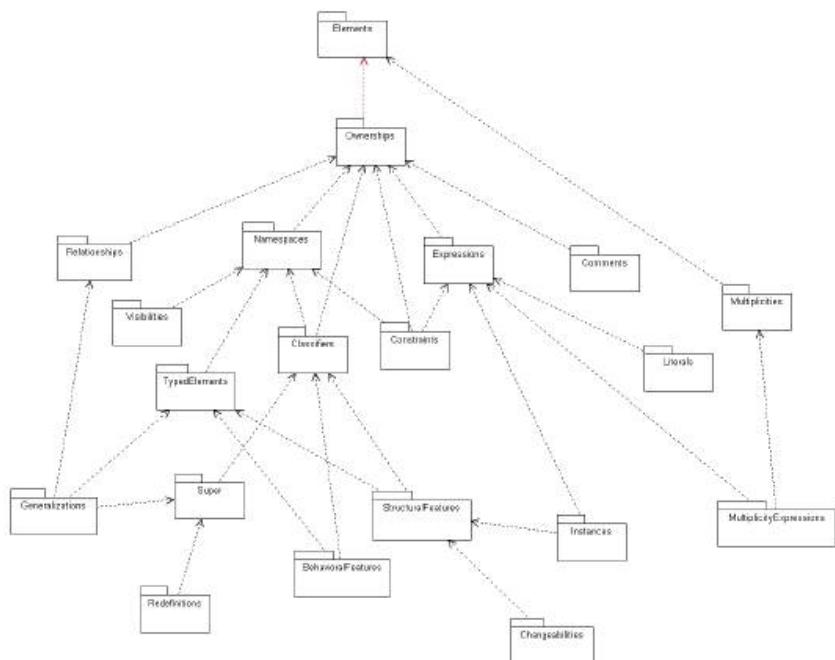
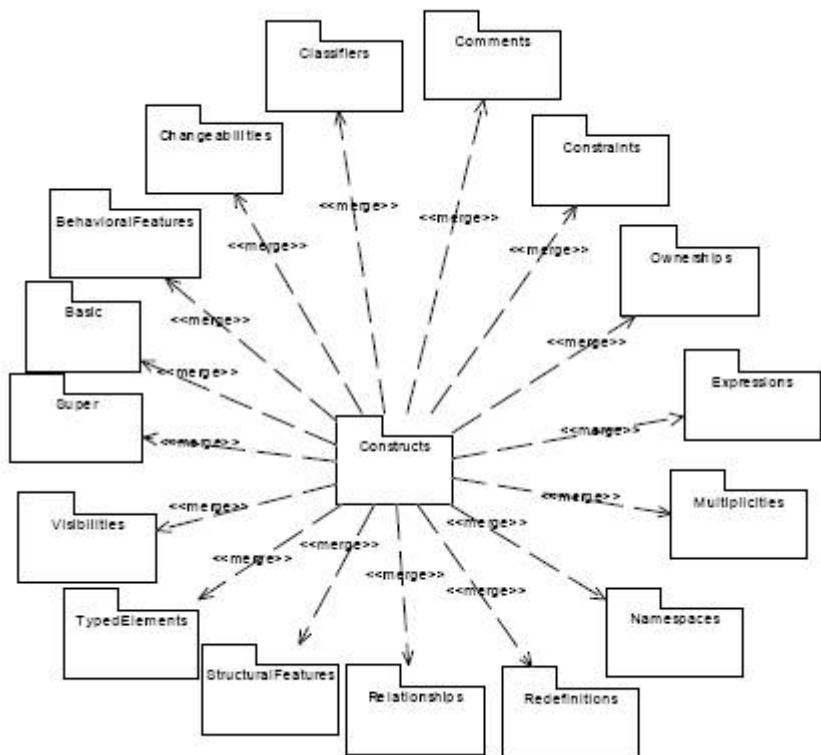


Figure 4. Core constructs packages



Le metaclassi-base

Le definizioni dei costrutti-base sono impostate come riportato dai diagrammi che seguono.

Figure 5. Types

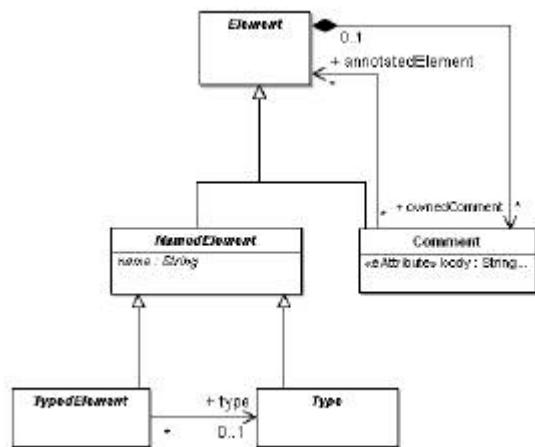


Figure 6. Classes

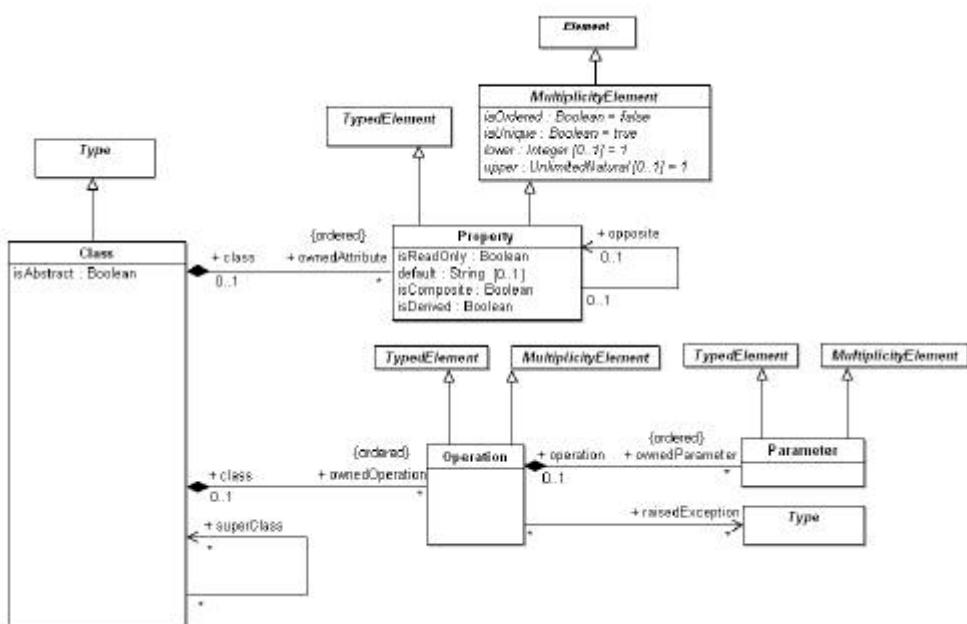


Figure 7. DataTypes

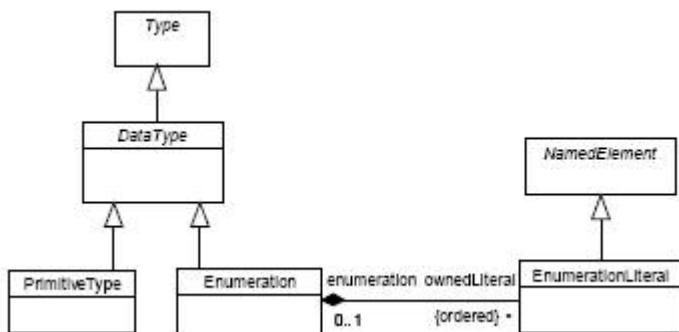
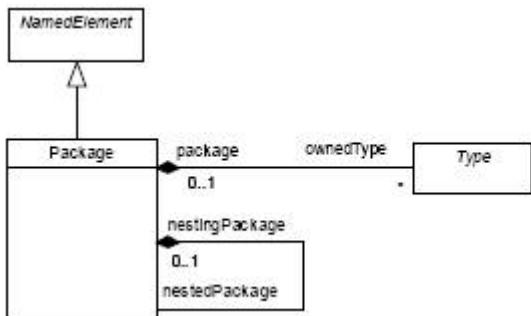


Figure 8. Packages



Riuso in MOF

In MOF l'**importazione** dei package fa sì che gli elementi di modellazione del package importato siano visibili nel package che fa l'importazione. Il **merge** dei package fa sì che il package che esegue il merge venga esteso con le nuove caratteristiche che provengono dal package che "subisce" il merge.

In MOF2.0 è possibile utilizzare tutti i concetti legati alla modellazione delle classi come importing, subclassing, aggiunta di nuove classi, associazione e aggiunta associazioni tra classi esistenti. Questi concetti sono poi utilizzati per definire package aggiuntivi come per esempio **Reflection** (estende un modello con la capacità di autodescriversi) , **Extents** (permette di estendere gli elementi di un modello attraverso l'uso di una coppia nome/valore) e **Identities** (fornisce una estensione che permette di identificare univocamente gli oggetti del meta-modello indipendentemente dal modello dei dati che può essere soggetto a cambiamenti nel tempo) sia in MOF 2.0 che in tutti gli altri modelli MOF2 compliant.

In MOF il riuso avviene attraverso il meccanismo di estensibilità di CMOF.

MOF definisce anche un **meccanismo di estensione** che può essere usato per estendere metamodelli in alternativa o in congiunzione con i **profili** (descritti nel Chapter 13 della Infrastructure spec, **Core: Profiles**). I profili sono definiti come un subset del meccanismo di estensione di MOF.

EMOF e CMOF

Essential MOF (EMOF) è un sottoinsieme di MOF che corrisponde alle facility che è possibile trovare nei linguaggi di programmazione ad oggetti e in XML. EMOF fornisce un framework per mappare semplici modelli MOF in diverse implementazioni come per esempio JMI a XMI. L'obiettivo principale di EMOF è quello di permettere la definizione di meta-modelli usando concetti semplici e allo stesso tempo supportando meccanismi di estensione per la creazione di meta-modelli più sofisticati.

Complete MOF (CMOF) è invece il meta-modello utilizzato per specificare tutti gli altri meta-modelli tra cui quello di UML2. CMOF è costruito a partire da EMOF e dal package **Constructs** del Core di UML2.

Figure 1. EMOF e CMOF

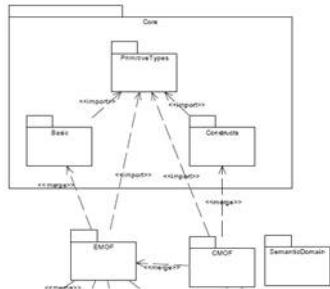
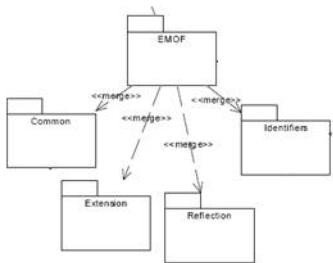
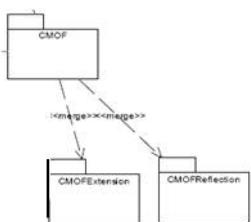


Figure 2. EMOF



Il package **CMOF** riusa la sintassi astratta definita nella **InfrastructureLibrary** per **UML**, **MOF**.

Figure 3. CMOF



Eclipse Ecore

Eclipse [GBG03] fornisce un framework per la generazione model driven denominato **Eclipse Modeling Framework** [BSMEG07] ([EMF](#)). Tale framework si colloca a metà strada tra un approccio **MDA** vero e proprio e un approccio totalmente privo di modelli (code-based) cercando di raggiungere un bilanciamento pragmaticamente utile tra cofica e modellazione.

Il metalinguaggio usato in [EMF](#) per rappresentare i modelli è denominato **Ecore** (si veda http://wiki.eclipse.org/Ecore_Tools) e può essere visto come una implementazione allineata con `xref href="..//uml/EMOF-2003.pdf" format="pdf"/>`.

Ecore introduce i seguenti concetti

- **EClass** per rappresentare una classe, dotata di zero o più attributi e zero o più references.
- **EAttribute** per rappresentare un attributo, dotato di un nome e di un tipo.
- **EReference** per rappresentare un terminale di associazione tra classi, dotato di un nome, di un boolean per indicare contenimento e un referene target type che è una seconda classe.
- **EDataType** per rappresentare il tipo di un attributo.

