UNIVERSITA' DEGLI STUDI DI NAPOLI
"FEDERICO II"
FACOLTA' DI INGEGNERIA

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

TESI DI LAUREA

STRUMENTI PER L'IMPLEMENTAZIONE AUTOMATICA DI VINCOLI IN APPLICAZIONI A COMPONENTI

RELATORI

CH.MO PROF.
D'AMBROSIO
ANTONIO D'ACIERNO
CH.MO PROF.
GIUSEPPE DE PIETRO
CH.MO PROF.
ANTONIO CORONATO

CANDIDATO

DIEGO

MATR. 41/1418

ANNO ACCADEMICO 2002/2003

“A mamma Angela e papà Antonio”
Con affetto…
... grazie !

Ringraziamenti

I miei più sentiti ringraziamenti vanno ai Professori Antonio Coronato, Antonio D'Acierno e Giuseppe De Pietro, per la loro disponibilità e per il paziente e scrupoloso contributo alla fase di stesura e correzione del presente lavoro.

INDICE

CAPITOLO III

IL LINGUAGGIO SDL E LA PROGETTAZIONE DEI TOOLS

CAPITOLO IV

METODOLOGIA PROPOSTA PER LO SVILUPPO DI APPLICAZIONI A COMPONENTI

CAPITOLO V

CASO DI STUDIO

CAPITOLO I
I SISTEMI DISTRIBUITI

1.1 Introduzione

Alla fine degli anni ’80, la diffusione delle reti di calcolatori e dei sistemi distribuiti hanno fatto emergere una nuova dimensione dell’industria del software su larga scala: lo sviluppo del software attraverso l’integrazione d’applicazioni.
La diffusione delle reti ha fatto crescere per aziende e organizzazioni complesse, l’esigenza di un nuovo approccio allo sviluppo dei propri sistemi informativi per adeguarsi alla diversità e ai cambiamenti delle tecnologie hardware, software e di rete; un approccio basato non soltanto o principalmente sulla riprogettazione dei sistemi, bensì sulla loro integrazione.
In ambiente di rete esistono vari fattori che aumentano la complessità dell’attività di sviluppo software:

- le applicazioni sono distribuite su una rete di calcolatori di caratteristiche differenti, dotati di hardware e di sistemi operativi diversi e spesso non interoperanti;
- tipicamente occorre adoperare diversi linguaggi di programmazione;
- i dati sono distribuiti su più nodi d’elaborazione, memorizzati in archivi o con sistemi di gestione di basi dati (DBMS) diversi, e devono essere in generale condivisi da applicazioni locali e remote;
- le problematiche di sicurezza, tolleranza ai guasti, contesa e condivisione delle risorse si presentano ben più complesse che per i sistemi centralizzati, anche in ambiente di rete omogeneo; l’eterogeneità aggiunge ulteriore complessità.

Tali fattori hanno fatto sì che l'elaborazione elettronica delle informazioni abbia assunto come paradigma generale quello d'elaborazione in ambiente distribuito eterogeneo. L' **interoperabilità** tra sistemi informativi sviluppati in momenti, con linguaggi e con tecniche diverse, e operanti su piattaforme eterogenee, è diventata una problematica centrale delle tecnologie dell'informazione.

Tale interoperabilità può assumere diverse forme, dalla possibilità di scambiare dati fra le applicazioni, alla **cooperazione applicativa** che prevede l'integrazione delle funzionalità delle diverse tipologie di applicazioni (aziendali, web, automazione di ufficio ecc).

Occorre anche considerare che progetti di applicazioni su larga scala richiedono tempi elevati, che si misurano in mesi e di frequente in anni: in simili periodi, con la rapidità con cui evolvono le piattaforme hardware, i sistemi operativi, il software di base, assume una fondamentale importanza la capacità di essere predisposti ai cambiamenti e di gestire, se non anticipare, l'evoluzione tecnologica. A tal fine, lo sviluppo di sistemi complessi richiede sempre più di poter:

- utilizzare applicazioni prefabbricate di terze parti (componenti Commercial Off-The-Shelf o COTS);
- riutilizzare applicazioni esistenti (sistemi ereditati o legacy);
- integrare le applicazioni aziendali con quelle di ufficio;
- costruire architetture complesse che integrino le tecnologie del web, dei sistemi di gestione di basi di dati (DBMS) e transazionali.

In effetti, sistemi complessi sono raramente sviluppati ex novo; tipicamente essi evolvono a partire da sistemi esistenti già funzionanti. L'integrazione di applicazioni, siano essi sistemi di nuovo sviluppo, sistemi ereditati, componenti COTS, è da considerarsi lo scenario usuale per analisti, architetti software e programmatori, nel contesto dello sviluppo di sistemi informativi su larga scala.

Elemento centrale delle moderne tecnologie per l'integrazione è la presenza di un'infrastruttura software d'interconnessione, ormai comunemente indicata con il

termine *middleware*, neologismo inglese con il quale si vuole evidenziare la sua caratteristica di trovarsi sopra il livello delle apparecchiature hardware e dei sistemi operativi, e di offrire servizi di cooperazione applicativa per le varie tipologie di applicazioni(Figura 1.1).

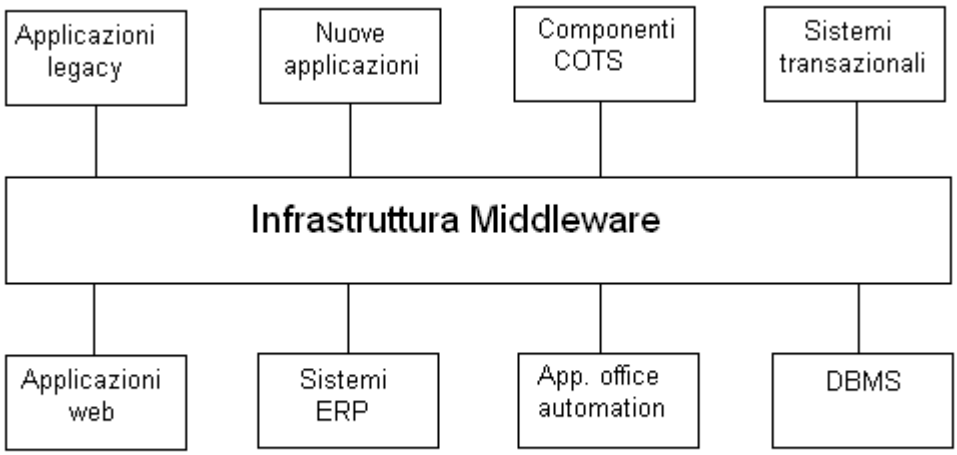


Figura 1.1 – *il middleware come infrastruttura d'integrazione*

1.2 Il middleware

Il termine *middleware* nasce una decina di anni fa per indicare, in maniera del tutto generale, uno strato software (programma, protocollo o altro) che si interpone tra il sistema operativo e le applicazioni. Il termine si è poi evoluto nel tempo assieme agli stessi concetti di applicazione e sistema operativo, assumendo però varie connotazioni e/o interpretazioni a seconda dell'implementazione effettiva di un *middleware*. Al giorno d'oggi le infrastrutture *middleware* sono impiegate maggiormente per la creazione di sistemi distribuiti sulla rete, col

compito principale di garantire la trasparenza dell'eterogeneità dei sistemi coinvolti.
Lo strato *middleware* offre ai programmatori di applicazioni distribuite librerie di funzioni, o *middleware API* (*Application Programming Interface*), in grado di mascherare i problemi dovuti all'eterogeneità dei sistemi su rete.
Il livello *middleware* può mascherare le eterogeneità mediante meccanismi di:

• **trasparenza del sistema operativo:** operando al di sopra del sistema operativo, i servizi forniti dalle API *middleware* possono essere definiti in maniera indipendente da esso, consentendo la portabilità delle applicazioni tra diverse piattaforme;

• **trasparenza del linguaggio di programmazione:** la comunicazione tra componenti sviluppati in linguaggi diversi pone problemi connessi alle differenze tra i tipi di dato supportati ed alla diversità tra i meccanismi di scambio parametri nell'invocazione di sottoprogrammi. Il livello *middleware* può consentire l'interoperabilità, definendo un sistema di tipi intermedio e regole non ambigue di corrispondenza con i tipi dei linguaggi più diffusi;

• **trasparenza della locazione:** le risorse (dati e servizi) dovrebbero essere accessibili a livello logico, senza conoscerne l'effettiva locazione fisica. E' lo strato *middleware* che deve farsi carico della localizzazione su rete del processo partner in una comunicazione, e del trasporto dei dati (*location transparency*);

• **trasparenza della migrazione:** dati e servizi possono venir rilocati durante il loro ciclo di vita, se un componente migra, il *middleware* può consentire l'accesso a componenti mobili sulla rete, in maniera trasparente ai moduli clienti (*migration transparency)*;

• **trasparenza ai guasti:** un'elaborazione distribuita può fallire anche solo in maniera parziale per guasti che si verificano nei singoli nodi o nel sottosistema di comunicazione. Occorrono tecniche complesse per poter gestire uno stato globale consistente della computazione. Lo strato *middleware* può offrire al

programmatore meccanismi ad alto livello per mascherare i guasti (*failure transparency*);

• **trasparenza della replicazione:** la replicazione di componenti è una delle tecniche più comuni per realizzare politiche di tolleranza ai guasti, ma può essere utilizzata anche per migliorare le prestazioni, attraverso un miglior bilanciamento del carico di elaborazione. L'esistenza di più copie di un componente dovrebbe però essere trasparente ai suoi clienti (*replication transparency)*;

• **trasparenza delle implementazioni commerciali:** alcune tecnologie *middleware*, tra cui CORBA, si presentano in realtà come specifiche di riferimento promosse da organizzazioni di produttori, spesso poi recepite dagli enti di standardizzazione. Le varie implementazioni commerciali sono quindi realizzate in maniera conforme allo standard, eventualmente differenziandosi tra loro per aspetti secondari o per servizi aggiuntivi fuori standard. In questo modo è resa possibile l'interoperabilità tra applicazioni basate su realizzazioni commerciali diverse dello stesso tipo di *middleware*. Nell'ultimo decennio sono state realizzate numerose tipologie di piattaforme *middleware*, sia in forma prototipale in ambito di ricerca, sia in forma commerciale.

1.3 Modelli di middleware

Nell'ultimo decennio si sono diffuse diverse tipologie di piattaforme *middleware*. Le varie tipologie differiscono tra loro in maniera sostanziale per funzionalità offerte, meccanismi di comunicazione, standard, modelli di programmazione e ambito applicativo. L'intero settore è tuttora in continua evoluzione, per cui è difficile definire delle rigide categorie di appartenenza delle varie piattaforme tecnologiche disponibili sul mercato. Una prima classificazione macroscopica è quella basta sul contesto d'uso per cui si distingue fra:

- *middleware* di comunicazione tra applicazioni
- *middleware* per l'accesso ai dati
- *middleware* per l'integrazione di servizi base di rete

Esistono altre tipologie di *middleware* che non rientrano rigidamente in una di tali categorie, e possono essere definite *middleware* di tipo verticale, orientati a specifici settori applicativi come le piattaforme per il settore finanziario.
Una seconda classificazione è quella basata sui meccanismi di comunicazione. In quest'ottica si distingue tra:

- *middleware* per comunicazioni sincrone
- *middleware* per comunicazioni asincrone
- *middleware* per la gestione di transazioni distribuite

Un'ulteriore classificazione si basa sui modelli di programmazione sia per l'interazione inter-applicazione sia per quella intra-applicazione. I modelli principali sono:

- modello di chiamata di procedura remota (*Remote Procedure Call* o RPC)
- modello orientato ai messaggi (*Message-Oriented Middleware* o MOM)
- modello transazionale (*Transaction Processing* o TP)
- modello dello spazio delle tuple (*Tuple-Spaces* o TS)
- modello di accesso remoto ai dati (*Remote Data Access* o RDA)
- modello a oggetti distribuiti (*Distribuited Objects Middleware* o DOM)
- modello a componenti (*Component Model* o CM)

1.3.1 Il modello di chiamata di procedura remota

Il modello RPC estende la programmazione sequenziale al contesto distribuito mediante il paradigma cliente-servente. Un programma sequenziale è tipicamente articolato in un insieme di procedure con un unico flusso di controllo. Nel modello RPC, il flusso di controllo logico resta unico, ma viene eseguito in ambiente distribuito.

La procedura remota è eseguita nel contesto di un processo su un altro elaboratore connesso in rete. Lo scambio dei parametri fra programma chiamante e procedura chiamata avviene come in una tradizionale invocazione di sottoprogramma; il programmatore invoca la procedura remota con le stesse modalità di una locale, e resta dunque isolato dai dettagli delle diverse piattaforme su cui sono in esecuzione i processi cliente e quelli serventi di un'applicazione.

Per fornire tale livello di astrazione, il funzionamento del modello RPC si basa sulla presenza di speciali procedure d'interfaccia dette *stub*, una dal lato cliente e una dal lato servente. Tali procedure implementano in maniera trasparente al programmatore la comunicazione su rete per lo scambio dei parametri tra il programma chiamante (cliente) e la procedura chiamata (servente); a tal fine essi accedono direttamente al livello di trasporto dei protocolli di rete. Le procedure stub sono compilate e collegate con le rispettive parti dell'applicazione.

Il meccanismo RPC è illustrato in Figura 1.2. Lo *stub* cliente:

- preleva i parametri di scambio dal chiamante;
- li impacchetta in un apposito messaggio che affida al software di rete affinché sia trasmesso alla macchina dove risiede il servente;
- attende il messaggio di risposta che indica il completamento della procedura;
- interpreta il messaggio e ne estrae i valori dei parametri d'uscita;
- restituisce i parametri al programma chiamante che riprende il controllo

lo *stub* servente:

- è in attesa che il software di rete consegni un messaggio di richiesta di esecuzione della procedura;
- quando arriva una richiesta, interpreta il messaggio, prelevandone i parametri d'ingresso;
- fornisce i parametri alla procedura chiamata, cui cede il controllo;
- riprende il controllo al termine della procedura e inserisce i parametri di uscita in un messaggio di risposta al cliente;
- invia il messaggio di risposta tramite il software di rete e si pone nuovamente in attesa di richiesta

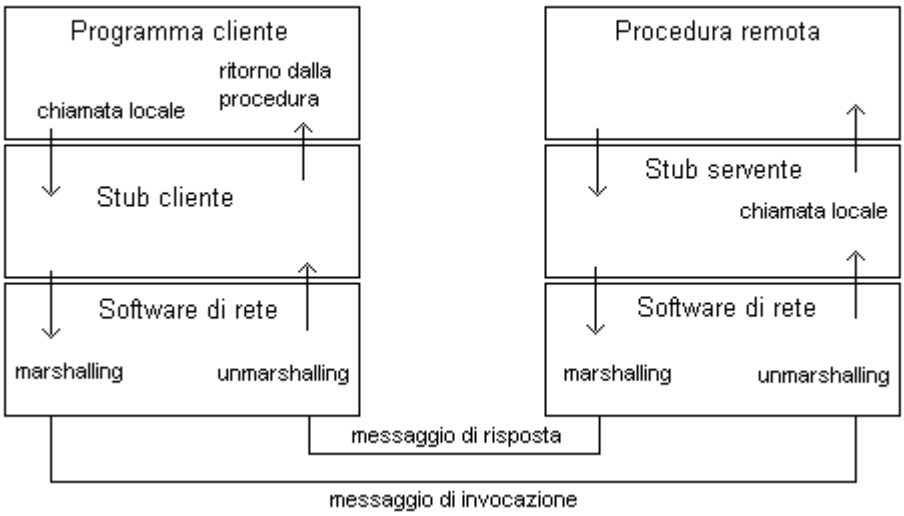


Figura 1.2 – *il funzionamento del meccanismo RPC*

Le operazioni svolte dagli *stub* per l'inserimento e l'estrazione dei parametri della chiamata di procedura prendono il nome rispettivamente di *marshalling* e *unmarshalling*. In particolare, il *marshalling* dei parametri consiste in:

- una conversione del formato dei dati, per tenere conto delle differenze di rappresentazione sulle macchine cliente e servente
- una serializzazione dei dati che vengono trasformati in sequenze di byte e impacchettati in un formato comprensibile a cliente e servente

Per garantire la trasparenza della posizione, è solitamente previsto un processo *binder*, responsabile di gestire un elenco delle procedure RPC disponibili sul nodo: ai serventi esso offre primitive di registrazione nell'elenco, mentre ai clienti offre primitive per richiedere la lista dei servizi. È previsto inoltre un processo che si occupa di attivare dinamicamente il servente: tale processo gestisce un elenco di servizi e delle relative immagini eseguibili e si occupa di avviare l'esecuzione di un processo corrispondente all'immagine del servizio richiesto.
Le applicazioni basate sul modello RPC sono sviluppate solitamente con l'aiuto di strumenti ausiliari per la generazione degli *stub*, infatti, la struttura di questi ultimi è esattamente definita, una volta che sono noti i parametri della RPC, il formato dei messaggi e le procedure del software di rete da invocare per la comunicazione. È quindi possibile automatizzare la produzione del codice per le procedure *stub*, ciò è fatto medianti appositi generatori di codice, che, in effetti traducono la specifica dell'interfaccia tra modulo chiamante e modulo servente, generando gli stub. La specifica di tale interfaccia è redatta dal programmatore mediante un linguaggio di alto livello, che è denominato linguaggio di definizione delle interfacce o IDL (interface definition language).

1.3.2 Il modello orientato ai messaggi

I sistemi middleware orientati allo scambio dei messaggi o MOM (message oriented middleware), si basano sull'astrazione di una coda di messaggi tra processi interoperanti su rete (Figura 1.3), generalizzazione del concetto di *mailbox*, tipico dei sistemi operativi.

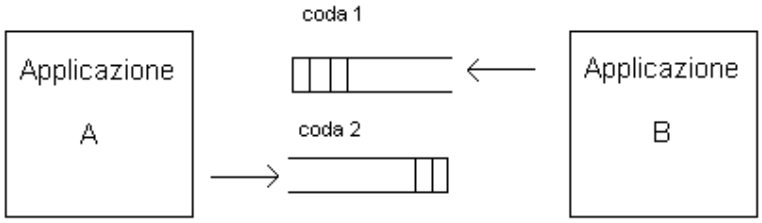


Figura 1.3 - il *funzionamento del meccanismo MOM*

Nel modello MOM la comunicazione non è di tipo cliente-servente, nel senso che l'interazione non avviene per sua natura tra un'entità in attesa passiva di richieste ed entità che intraprendono la comunicazione, ma è tra pari (*peer-to-peer*), di tipo produttore-consumatore, e prevalentemente asincrona, sebbene alcuni sistemi consentano anche uno scambio di messaggi di tipo sincrono.

I sistemi MOM adottano spesso un modello *publish-subscribe*, in cui i processi produttori pubblicano messaggi differenziati per tipo (*publish*) e i consumatori possono dichiararsi interessati ai messaggi in base al loro tipo (*subscribe*), al fine di ricevere dal *middleware* la notifica di una presenza nella coda di messaggi d'interesse. In questo modo è possibile una comunicazione una-a-molti, non supportata dal modello RPC.

Le piattaforme MOM hanno trovato una certa applicazione in ambito *Enterprise Application Integration*, ove sono spesso preferite al modello RPC, infatti, integrare un numero elevato di applicazione comporta, nel modello RPC, troppi punti di interconnessione e renderebbe proibitiva la fase di manutenzione. L'impiego di MOM, invece, consente di disaccoppiare clienti e serventi, e il numero di connessioni varia linearmente con il numero dei componenti da integrare.

I servizi delle piattaforme MOM comprendono in generale primitive di:

- creazione e cancellazione di code temporanee e persistenti

- immissione di messaggi in coda. I messaggi possono essere indirizzati a un unico destinatario o a più destinatari
- assegnazione priorità ai messaggi
- prelievo dei messaggi da una coda
- condivisione di una coda tra più applicazioni

1.3.3 Il modello transazionale

Il modello transazionale è in uso da decenni in applicazioni critiche di accesso a basi di dati, quali quelle in ambito bancario o assicurativo, o nei sistemi di prenotazione delle compagnie aeree. Una transazione è una sequenza di elaborazione che, pur essendo eseguita in concorrenza con altre e in un ambiente dove sono possibili malfunzionamenti, gode di ben precise proprietà atte a garantirne la correttezza, note come proprietà ACID (atomicità, Consistenza, Isolamento, Persistenza).

Tale tecnologia, nota anche come TP, è nata per lo sviluppo di applicazioni affidabili di accesso ad archivi centralizzati su sistemi di tipo mainframe. Con lo sviluppo dei sistemi di gestione di basi di dati (DBMS) tale tecnologia è stata incorporata nel motore di tali sistemi DBMS.

La tecnologia TP è stata successivamente estesa agli ambienti distribuiti ed appare come la migliore soluzione per garantire la integrità transazionale in applicazioni critiche in ambiente eterogeneo; per tale motivo anche altre categorie di middleware tendono a incorporare funzioni di tipo TP; un esempio n'è il *Transaction Service* di CORBA.

I servizi offerti da un *middleware* TP comprendono:

- assegnazione di priorità alle richieste, schedulazione e dispatching ai DBMS
- gestione di transazioni distribuite tra DBMS eterogenei
- gestione di transazioni multiprocesso
- bilanciamento del carico
- ripristino da malfunzionamenti

1.3.4 Il modello dello spazio delle tuple

Il modello dello spazio delle tuple è nato a metà degli anni 80 all'università di Yale negli Usa nell'ambito di LINDA, uno dei primi sistemi classificabili come *middleware*.

In questo modello la comunicazione tra applicazioni ha luogo attraverso un'astrazione costituita da uno spazio virtuale di memoria condiviso (*virtual shared memory*): le applicazioni possono sincronizzarsi e scambiarsi dati inserendo e prelevando strutture dati (le tuple) dallo spazio virtuale (Figura 1.4) mediante operazioni di scrittura (write), lettura (read) e rimozione (take). LINDA è considerato il capostipite di una famiglia di linguaggi di coordinazione per applicazioni parallele o distribuiti, diffusisi negli anni '90, prima in ambiti di ricerca e successivamente anche in piattaforme commerciali. Due esempi d'implementazioni commerciali di questo modello sono *TSpaces* di IBM e JavaSpaces della Sun.

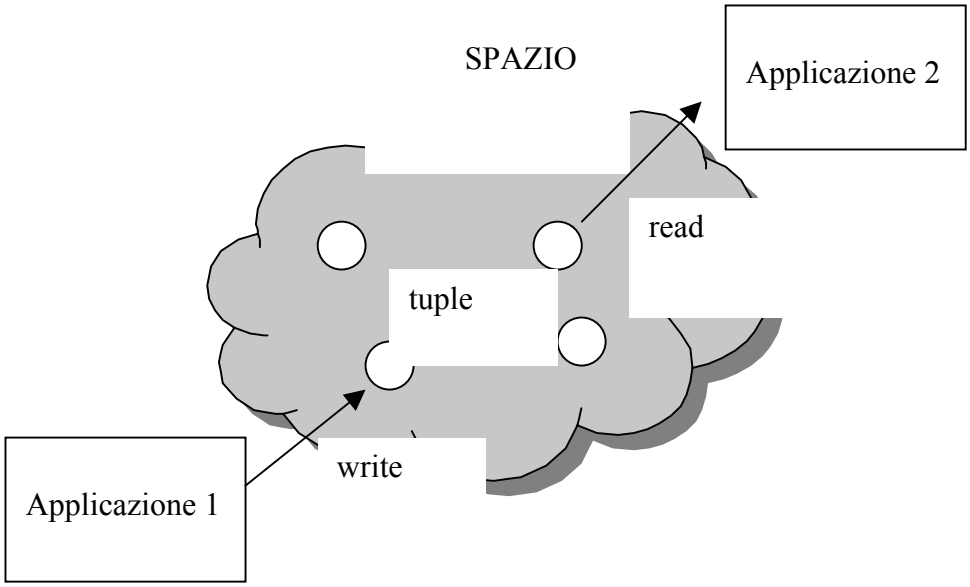


Figura 1.4- *Il modello dello spazio delle tuple*

Lo spazio delle tuple può essere considerato uno spazio di memoria associativa, in quanto l'accesso alle tuple avviene per contenuto e tipo e non per indirizzo: un'operazione di lettura richiede di fornire una tupla-modello (pattern), che viene confrontata con le tuple presenti nello spazio. La lettura restituisce una o tutte le tuple che corrispondono esattamente al pattern specificato.

Una caratteristica che ha fato tornare recentemente di attualità il modello dello spazio delle tuple è che in esso la comunicazione è anonima (due entità interagenti non devono avere un riferimento l'una all'altra) e con accoppiamento lasco (le entità interagiscono indirettamente inserendo e prelevando tuple dallo spazio virtuale). In effetti, nei modelli di middleware basati sul paradigma cliente-servente (RPC, TP, RDA, DOM), un cliente deve possedere un riferimento al servizio remoto per effettuare la richiesta, fatto che comunque avviene in maniera trasparente al programmatore grazie ai meccanismi di binding dinamico. Tuttavia, moderne applicazioni che coinvolgono apparati mobili e di limitata potenza elaborativa, come i dispositivi di elettronica di consumo, quali telefoni cellulari e Personali digital assistant, richiedono maggiore flessibilità e adattatività alle

caratteristiche dei terminali utente. Per tali applicazioni si va affermando un modello architetturale orientato ai servizi denominato *SOA* (*service-oriented architecture*), per il quale appaiono particolarmente adatti middleware basati sul modello dello spazio delle tuple.

1.3.5 Il modello di accesso remoto ai dati

Con l'espressione *middleware* di accesso remoto ai dati (*Remote Data Access o RDA*) ci si riferisce a quelle piattaforme, spesso anche denominate genericamente *database middleware*, che offrono servizi per lo sviluppo di applicazioni di accesso a basi di dati remote, basate sul paradigma cliente servente. Esse forniscono servizi di accesso a archivi o a sistemi DBMS di vario tipo (relazionale, gerarchico, a oggetti), con diversi livelli di trasparenza per mascherare la distribuzione dei dati sui nodi di rete e le eterogeneità tra i sistemi.
Questa tipologia di middleware adopera il linguaggio standard SQL (*Structured Query Language*) per l'interrogazione e la manipolazione logica di dati remoti, indipendentemente dal sistema con cui vengono gestiti fisicamente. Le interfacce di programmazione offerte sono di tipo proprietario, oppure basate su standard *de facto*, quali per esempio la specifica *Open DataBase Connectivity* (ODBC).
Le applicazioni distribuite basate sull'approccio RDA sono realizzate tipicamente con architetture a due o a tre livelli. In entrambi i casi il cliente non accede direttamente ai dati remoti, ma il *middleware* RDA fa da intermediario, gestendo da un lato la concorrenza tra più clienti, dall'altro facendo da gateway verso i gestori fisici dei dati.
Nel caso di architetture a più livelli, il funzionamento può essere così schematizzato (Figura 1.5):

- l'applicazione cliente prepara una interrogazione SQL che deve essere eseguita sul gestore remoto dei dati, e la trasmette al middleware

- il middleware sul nodo cliente impacchetta la richiesta in un messaggio, che inoltra al sistema servente;
- il middleware sul nodo servente riceve il messaggio, estrae l'istruzione sql e la invia al dbms o al gestore dei dati
- il dbms esegue il comando e invia la risposta completa al middleware
- il middleware sul nodo servente si occupa di inoltrare i dati all'omologo componente sul cliente, eventualmente suddividendoli in più messaggi
- l'applicazione cliente preleva i dati della risposta dal middleware locale

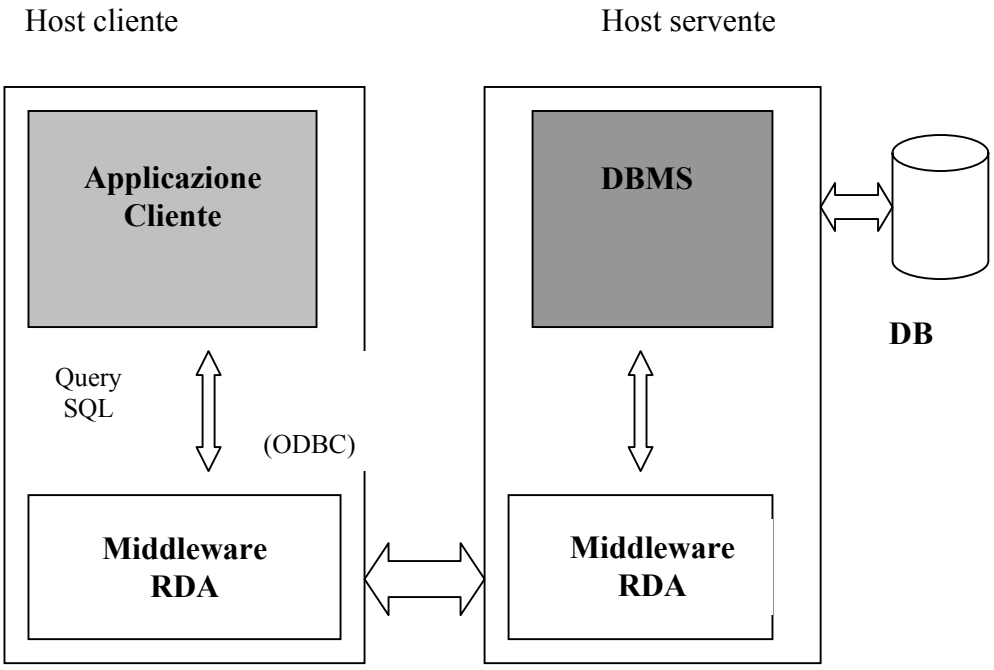


Figura 1.5 - *Funzionamento del modello RDA*

In linea di principio la presenza dello strato aggiuntivo rispetto all'accesso diretto al DBMS remoto introduce un sovraccarico. I vantaggi della presenza dello strato middleware appaiono tuttavia chiari se si pensa che un'interrogazione SQL produce in genere una quantità di dati variabile: la dimensione della risposta non è

perciò nota a priori (a differenza delle chiamate di procedura remota). L'applicazione cliente non accede direttamente al DBMS ma preleva i record risultanti dall'interrogazione direttamente dall'interfaccia RDA locale. Il DBMS è quindi sollevato dall'onere di gestire sessione applicatrice.

1.3.6 Il modello ad oggetti distribuiti

Il modello ad oggetti distribuiti nasce dall'integrazione di due paradigmi di programmazione:

- il paradigma dell'orientamento agli oggetti
- il paradigma cliente-servente

Integrando i due paradigmi, il modello estende le tecniche di sviluppo ad oggetti al contesto della programmazione su rete. Le piattaforme *middleware* che si basano su tale modello, sono solitamente classificate come *Distribuited Object Middleware* (DOM).
Il paradigma dell'orientamento agli oggetti (object orientation O-O) è ampiamente utilizzato in molte fasi dello sviluppo software, dall'analisi alla progettazione alla programmazione, per la sua capacità di ben supportare molti principi fondamentali dell'ingegneria del software, quali l'astrazione, l'incapsulamento, l'information hiding, la modularità, la coesione intra-modulo, il disaccoppiamento tra moduli, il riuso, lo sviluppo incrementale e l'estensibilità. Elemento cruciale delle tecniche O-O è la modellazione di un sistema come un insieme di oggetti, ciascuno dotato di un proprio stato, accessibile solo attraverso l'invocazione di operazioni sull'oggetto stesso.
Il paradigma cliente-servente è probabilmente il più diffuso, per la sua semplicità, nello sviluppo di applicazioni distribuite. Il paradigma classifica i componenti di un sistema software distribuito in due tipologie:

- moduli servente, dotati di un comportamento reattivo alle richieste che provengono da altri componenti
- moduli cliente, che prendono l'iniziativa della comunicazione

Dal punto di vista concettuale, i due paradigmi si fondono in maniera naturale. Un oggetto può, infatti, essere visto come un servente, inteso come entità passiva, che accetta richieste di operazioni per l'accesso al proprio stato interno; all'invocazione di un'operazione su un oggetto-servente può essere attribuita la semantica di richiesta di un servizio. Il paradigma ad oggetti è intrinsecamente di tipo cliente-servente.

In termini concreti, esistono profonde differenze tra il concetto di oggetto proprio delle tecniche di programmazione O-O e quello proprio dei sistemi ad oggetti distribuiti. Nel primo caso, gli oggetti tipicamente appartengono a classi, che rappresentano la specifica delle proprietà e responsabilità degli oggetti, e che sono offerte sotto forma di librerie software. Gli oggetti distribuiti costituiscono invece moduli eseguibili autonomamente, e accessibili attraverso la rete mediante interfacce.

A differenza dunque della programmazione in un linguaggio ad oggetti, in un sistema distribuito gli oggetti non sono in esecuzioni in uno stesso ambiente (processo), ma sui vari nodi di rete, e l'invocazione di un'operazione assume il carattere di una richiesta remota. In questo senso, la semantica di un'invocazione remota è del tutto analoga a quella di una chiamata di procedura remota. Compito delle piattaforme DOM è fornire meccanismi analoghi a quelli delle piattaforme RPC, ma con primitive che corrispondono alle astrazioni del paradigma ad oggetti. Le tecnologie DOM aggiungono quindi alla comunicazione cliente-servente di tipo RPC i benefici della tecniche O-O.

Le piattaforme DOM si presentano sovente sotto forma di *Object Request Brokers* (ORB). Un ORB è una sorta di *bus* software per l'interazione tra oggetti distribuiti su sistemi eterogenei; la comunicazione con un oggetto-servente è mediata dal

bus, e avviene tramite messaggi di richiesta e risposta scambiati da apposite procedure *stub*, come nel modello RPC (Figura 1.6). Esempi di ORB sono Microsoft DCOM e OMG CORBA.

Le piattaforme DOM mutuano da quelle RPC molte caratteristiche, quali la presenza di un linguaggio IDL, di procedure *stub*, e di meccanismi di binding dinamico. L'ORB fa da intermediario nelle interazioni fra oggetti su rete, garantendo proprietà di trasparenza dei sistemi operativi, della locazione e migrazione degli oggetti, e consentendo al programmatore di invocare metodi su oggetti senza conoscere la loro effettiva posizione nella rete, né le caratteristiche della macchina su cui risiedono. La trasparenza del linguaggio di programmazione è ottenuta tramite il linguaggio di descrizione delle interfacce, e mediante compilatori che svolgono il ruolo di generatori delle procedure *stub*. La trasparenza delle implementazioni commerciali è garantita solo dalle tecnologie non proprietarie, conformi a standard internazionali. È questo il caso di CORBA, che non rappresenta una specifica tecnologia, ma un modello di riferimento standard per implementazioni commerciali di piattaforme DOM.

Un modello ad oggetti distribuiti non basato sulla notazione di ORB è quello della tecnologia *Remote Method Invocation* (RMI) di Sun Microsystem. Questo middleware è specifico per la piattaforma Java, e fa riferimento ad un ambiente di elaborazione omogeneo costituito da macchine virtuali Java (*Java Virtual Machine* o JVM).

- CORBA

Lo standard *Common Object Request Broker Architecture* (CORBA) è stato definito nel 1991 dall'OMG (Object Management Group e fa parte di un più ampio modello architetturale denominato *Object Management Architecture* (OMA).

La è la specifica di più alto livello dell'OMG. Essa prevede quattro ambiti di standardizzazione per servizi di supporto alle applicazioni:

- l'object request broker o ORB;
- i servizi (CORBA*services*);
- i servizi di alto livello (CORBA*facilities*);
- i domini (CORBA*domains*)

Il termine CORBA si riferisce alla specifica dell'ORB ma è frequentemente usato per indicare tutta l'architettura OMA.

I quattro elementi OMA sono spesso rappresentati gerarchicamente come mostrato in Figura 1.6, per evidenziare il livello crescente di astrazione dei servizi offerti. Alla base della piramide c'è l'ORB, che fornisce i meccanismi di comunicazione fondamentali. Il livello dei CORBA*services* contiene i servizi agli oggetti, mentre quello delle CORBA*facilities* contiene i servizi alle applicazioni, detti di tipo "orizzontali", utili cioè per le diverse categorie di applicazioni. Il livello dei CORBA*domains* contiene servizi specializzati per domini applicativi. In cima alla piramide sono i *business object*, termine con cui si indicano componenti riusabili di livello applicativo.

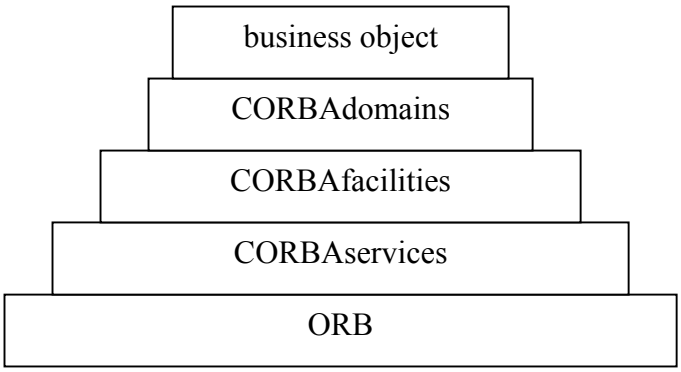


Figura 1.6 –*Livelli di astrazione dei componenti di OMA*

Un'applicazione distribuita CORBA consiste di oggetti serventi, entità passive che offrono servizi, e di clienti, entità attive che accedono ai loro servizi. Ogni oggetto servente consiste in due parti distinte: l'**interfaccia** pubblica, specificata in un linguaggio apposito, e l'**implementazione**, redatta in un linguaggio di programmazione, che resta privata per l'oggetto stesso. Il cliente interagisce con il servente sulla base dei servizi pubblici di interfaccia, ignorandone i dettagli implementativi.

Per la descrizione delle interfacce, lo standard CORBA definisce il linguaggio IDL. Per lo sviluppo dei clienti e dei serventi, lo standard definisce un insieme d'interfacce di base, che costituiscono il cosiddetto nucleo dell'*object request broker*, la cui struttura è mostrata in Figura 1.7.

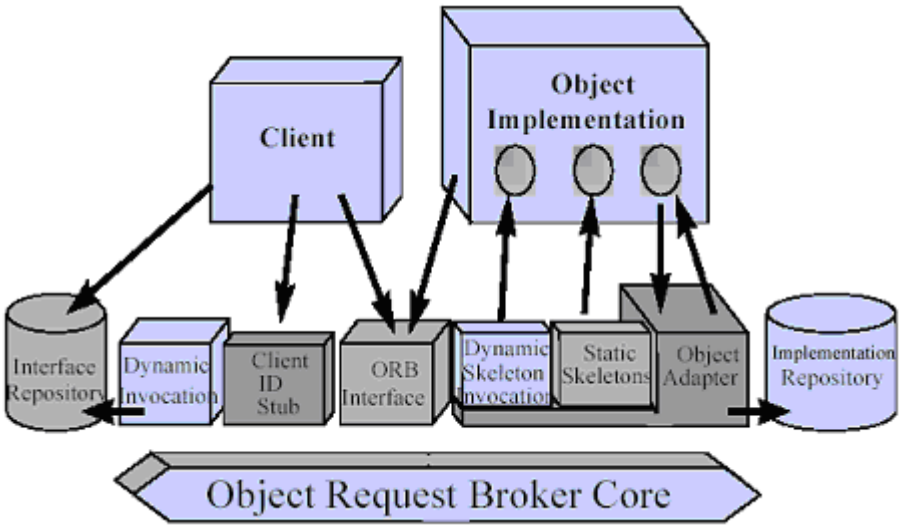


Figura 1.7 - *Architettura CORBA*

- L'OMG *Interface Definition Language* (IDL) utilizzato per circoscrivere all'interno di un oggetto le differenze legate al linguaggio di programmazione. L'oggetto CORBA corrisponde ad una definizione d'interfaccia descritta in IDL. Un compilatore IDL riceve la descrizione dell'interfaccia ed esegue una corrispondenza tra l'IDL ed il linguaggio in cui è realmente implementato l'oggetto. La compilazione di un'interfaccia produce almeno due interfacce: lo

stub (lato client) e lo *skeleton* (lato server). Lo *stub* è l'interfaccia attraverso cui un client può richiedere servizi ad un server. Esso contiene tutte le definizioni delle operazioni dell'interfaccia IDL dell'oggetto in forma di definizioni di metodo del linguaggio di programmazione utilizzato. Dal punto di vista del client, rappresenta la parte dell'oggetto CORBA con cui esso interagisce. Il client esegue, di fatto, un'invocazione locale allo stub, che "impacchetta" (marshalling) i dati dell'invocazione in un messaggio che è consegnato all'ORB e da questo inviato al server. Di conseguenza allo stub del client è associato a tempo di esecuzione un riferimento remoto all'oggetto (Object Reference), che è il mezzo utilizzato dal client per invocare operazioni sull'oggetto. Per questa ragione lo stub è spesso chiamato proxy. Lo *skeleton* è l'equivalente dello stub dal lato server. Esso rappresenta lo scheletro dell'oggetto ed è la struttura di partenza per l'implementazione dell'oggetto. Infatti, la classe che implementa l'oggetto (detta *Servant*) deve ereditare lo skeleton e aggiungere la business logic alle operazioni dichiarate in esso. Lo skeleton si differenzia dallo stub poiché coopera con un *Object Adapter* per le operazioni che riguardano l'attivazione dell'oggetto. Lo *skeleton* riceve la richiesta dall'Object Adapter, ne estrae i dati (unmarsahlling) e li passa all'implementazione dell'oggetto.

- *L'Object Request Broker* (ORB) utilizzato per nascondere la locazione fisica a due oggetti interagenti, consentendo pertanto una serie di operazioni (invocazione di metodo, attivazione ecc.) a prescindere dalla locazione degli oggetti all'interno dell'ORB. Questa funzionalità è realizzata attraverso una serie d'interfacce sopra un *core ORB* come l'Object Adapter, la Dynamic Invocation Interface (DII), l'Implementation Repository e l'Interface Repository (vedi Figura 1.7). La specifica del componente ORB dell'OMA è *CORBA* (Common Object Request Broker Architecture).
- *General Inter-ORB protocol* (GIOP) utilizzato per nascondere la locazione fisica a due oggetti interagenti che risiedono in due ORB distinti. GIOP è un protocollo astratto che specifica un insieme di tipi di messaggi utilizzabili tra

un client ed un server, una sintassi standard nel trasferimento dati delle interfacce IDL ed un formato standard per ogni messaggio utilizzabile. GIOP fornisce la sintassi per la comunicazione inter-ORB ma non la semantica. IIOP aggiunge le informazioni di cui ha bisogno un ORB, ovvero la semantica, per comunicare attraverso un meccanismo di trasporto con connessione ed affidabile come TCP.

- I *CORBA Object Service* (COS) supportano funzioni base per usare ed implementare oggetti per eseguire delle operazioni sul sistema. Questi servizi sono forniti d'interfaccia specificata dall'IDL.

I servizi CORBA aggiungono funzionalità all'ORB. Le operazioni messe a disposizione dal servizio CORBA sono definite da un'interfaccia IDL. Tale interfaccia è stata standardizzata dall'OMG. Attualmente, i servizi standard sono divisi in tre categorie servizi legati ai sistemi distribuiti, alle basi di dati e di tipo generale.

Servizi legati ai sistemi distribuiti. Di seguito è riportata una descrizione sommaria del naming e del trading service:

- **Naming service**. Permette (i) ad un client di trovare un oggetto attraverso il suo nome e (ii) ad un server di registrare i suoi oggetti dandogli un nome.
- **Trading Service**. Il trading service è un servizio di locazione di oggetti, come il Naming Service, ma lavora ad un livello più astratto. Nel Naming Service si cerca un oggetto a seguito di un nome. Nel Trading Service si fanno delle richieste per avere una lista di oggetti che soddisfano certe caratteristiche o proprietà (inserite nella richiesta). Il Trading Service contiene dei record chiamati *offer*, formati da un campo *tipo di servizio*, *insieme di proprietà del servizio* e *riferimento all'oggetto*. Un server che vuole inserire un record relativo ad un suo oggetto nel Trading Service deve inviare la struttura dati offer opportunamente riempita. Questa operazione viene effettuata attraverso una export().
- **Event Service.** Aggiunge a CORBA la possibilità di utilizzare un paradigma asincrono simile al publish/subscribe per le comunicazioni tra gli oggetti.

- **Notification Service**. Estensione dell'Event Service, più flessibile e affidabile. In particolare il Notification Service definisce un modello dati per i messaggi, permette di effettuare delle sottoscrizioni basate su query e quindi di filtrare i messaggi su un canale e aggiunge il supporto per la qualità del servizio nella trasmissione delle notifiche.

Servizi di tipo generale. Di seguito è riportata una descrizione sommaria del Licensing e del Life Cycle Service:

- **Licensing Service**. Questo servizio controlla i diritti (di licenza) di utilizzo di un client di un determinato servizio CORBA, servizio applicativo o della piattaforma stessa.
- **Life Cycle Service (LFS).** L'interfaccia di LFS include una serie di operazioni per copiare, muovere, creare e distruggere un oggetto all'interno dell'ORB. Il componente centrale di un LFS è l'object factory ovvero un oggetto capace di creare altri oggetti. Un client che fa una richiesta di creazione di un oggetto avrà restituito un riferimento all'oggetto creato.

Servizi legati alle basi di dati. Questi servizi nascono dalla necessità di integrazione tra ORB e DBMS.

- **Transaction Service (OTS).** CORBA mettere a disposizione un servizio per la gestione delle transazioni in modo che queste ultime soddisfino la proprietà **ACID** (atomicità, consistenza, isolamento e durabilità) tipica delle transazioni nelle basi di dati. Il servizio OTS è caratterizzato da tre oggetti residenti in OTS: *terminator, coordinator e control* che gestiscono le fasi di terminazione, coordinamento e di controllo delle operazioni di una transazione. Inoltre per creare una transazione, un client ha a disposizione un *TransactionFactory* e l'oggetto modificato avrà un corrispondente *resource object*, legato alla transazione che sta effettuando la modifica, che coopererà con l'oggetto *coordinator* di OTS per il commit della transazione.
- **Externalization Service**. Questo servizio permette ad oggetti di essere immagazzinati in una sequenza di byte per essere successivamente ricostruiti. Per esempio un oggetto può essere immagazzinato in una sequenza di byte su

un disco rimovibile e quindi caricato su un'altra macchina. Le operazioni di questo servizio sono due: *externalize* e *internalize*. La prima scrive l'oggetto in una sequenza di byte includendo lo stato corrente al momento della "esternalizzazione", la seconda ricostruisce l'oggetto da una data sequenza di byte includendo lo stato "esternalizzato".

- **Relationship Service**.[8] All'interno di una piattaforma distribuita, gli oggetti stabiliscono delle relazioni. A volte queste sono rese esplicite dall'uso di database relazionali o database ad oggetti. Nel caso in cui non si abbiano a disposizione questi strumenti e si vogliano comunque esplicitare le relazioni, l'OMG ha introdotto la possibilità di definire relazioni all'interno dell'IDL attraverso un servizio apposito. Quindi, queste relazioni non fanno parte della definizione dell'oggetto ma sono definite nell'interfaccia di questo servizio.
- **Concurrency Control Service**. Questo servizio permette a client o transazioni di mettere lock sugli oggetti in lettura ed in scrittura, esclusivi e condivisi. Insieme al transaction service. Questo servizio permette di implementare algoritmi complessi di controllo della concorrenza tra le transazioni all'interno dell'ORB.
- **Query Service.** Questo servizio permette di interrogare collezioni di oggetti. Esso riceve della stringhe di caratteri che rappresentano l'interrogazione in un dato linguaggio. L'OMG ha per ora specificato un servizio di query in grado di interpretare interrogazioni in SQL e QQL.
- **Persistence Service.** Questo servizio mette a disposizione un insieme di strumenti per il salvataggio su memoria stabile dei dati che rappresentano lo stato di un oggetto. Questo stato può essere ad esempio reinstallato a seguito di un guasto che causa la morte dell'oggetto.

1.3.7 Il modello a componenti

Elemento centrale del modello a componenti è il concetto di componente software.

Un componente può essere definito come un'unità software dotata d'interfacce, le cui dipendenze da altre unità sono esplicite nella sua specifica, che può essere rilasciata e eseguita autonomamente e che può essere soggetta a composizione con altre unità per costruire unità più complesse.

Tale definizione cattura le proprietà essenziali di un modulo software eseguibile, inteso come unità di composizione dotata di una chiara separazione tra le proprietà e funzionalità visibili esternamente (interfaccia), e la loro realizzazione nascosta all'interno; essa evidenzia inoltre il requisito di un componente di essere efficacemente riutilizzabile.

Le tecniche d'ingegneria del software a componenti (*Component Based Software Enginnering* CBSE) estendono la nozione del riuso rispetto alle tecniche ad oggetti. Nella CBSE, il riuso non è inteso solo in termini di codice sorgente, classi e procedure, ma d'intere applicazioni. In tal senso il modello a componenti può essere considerato un'evoluzione del modello ad oggetti.

Nel modello ad oggetti distribuiti la logica delle interazioni tra oggetti è nascosta al loro interno, e le dipendenze tra loro non sono esplicite. Le connessioni tra oggetti sono strettamente legate alla logica del programmatore, e difficilmente quindi possono essere analizzate e configurate esternamente all'applicazione e dinamicamente. Ciò determina livelli di coesione interna e di accoppiamento tra oggetti inadeguati, ai fini della loro effettiva riusabilità, manutenibilità e sostituibilità. Il modello a componenti si propone di superare questo limite.

L'obiettivo dello sviluppo a componenti (o component-based) è l'integrazione di applicazioni esistenti, applicazioni sviluppate ex-novo e componenti COTS (component off-the-shelf) dando particolare enfasi a due aspetti: riuso e riduzione delle dipendenze fra le applicazioni in modo da facilitarne il cambiamento. In questa ottica, progettare sistemi compont-based può esere visto come un

problema di gestione delle dipendenze fra i componenti stessi; bisogna assicurare che componenti software, sviluppati in momenti differenti e da differenti organizzazioni interagiscano con successo.
Per raggiungere questo obiettivo, è utile pensare a un componente come un software che fornisce servizi, e ne richiede altri. In tale ottica si può fare una analogia tra un componente e una azienda. Le aziende forniscono servizi ai loro clienti e richiedono servizi ad altre aziende. Come vengono gestiti dalle aziende queste relazioni con i clienti e con i loro fornitori di servizi? Attraverso i contratti.
Un contratto è una descrizione non ambigua dei dettagli che costituiscono un accordo fra più parti. Esso specifica le responsabilità che ogni entità coinvolta nel contratto deve rispettare per accedere ai servizi offerti dalle altre entità. Analogamente, i rapporti fra i componenti di un sistema component-based possono essere regolati da contratti; è cosi nata la tecnica del "***Design by Contracts***" per progettare grossi sistemi come un insieme di componenti software cooperanti [9][10]. Si distinguono due tipi differenti di contratto:

Utilizzo- il contratto che regola i rapporti fra un componente e il suo cliente (cioè chi utilizza i servizi del componente)

Realizzazione- il contratto che deve rispettare l'implementatore

Un *contratto di utilizzo* descrive la relazione fra il cliente di un componente e i servizi che offre il componente, questa descrizione è fatta specificando una *interfaccia*.
La specifica di una interfaccia include:

- **Operazioni** – una lista di operazioni che l'interfaccia fornisce, incluse la loro signatures e definizione

- **Modello delle informazioni** – la definizione astratta di ogni informazione o stato che è mantenuto tra le richieste dei client e il componente che supporta le operazioni

Ogni operazione è trattata come una sorta di micro contratto in cui sono definiti gli input, gli output, le relazioni tra essi e le condizioni sotto le quali sono applicate. La ragione per raggruppare le operazioni in unità contrattuali più grandi (interfacce) è perché il loro utilizzo necessita l'esistenza di altre operazioni e perché esse sono usate raramente da sole.
Della stessa importanza della signature, per una operazione, sono le precondizioni e le postcondizioni:

- **Precondizione** – definizione della condizione in cui la Postcondizione sarà applicata
- **Postcondizione** – definizione degli effetti della operazione sui suoi parametri e sul modello delle informazioni

La prima è rivolta al cliente per assicurare che egli invochi l'operazione nelle condizione specificate. Conformandosi a questo contratto, egli accetta che se la precondizione non è sodisfatta il comportamento atteso non sarà rispettato. Da parte sua il componente promette di soddisfare le promesse fatte al cliente , mediante la postcondizione, se la precondizione è rispettata.
Nella specifica di una interfaccia è anche possibile esplicitare degli **invarianti** su quello che è stato prima chiamato modello delle informazioni, nonché delle **guardie** che possono essere viste come dei constraints che interassano il comportamento in genere del componente e non solo le operazioni, come fanno precondizioni e postcondizioni, o solo le informazioni, come gli invarianti.
Un *contratto di utilizzo* è un contratto valido a tempo di esecuzione, invece, un *contratto di realizzazione* è un contratto che ha valore in fase di progetto cioè

deve essere rispettato dal progettista del componente. Il contratto di realizzazione fa parte, così come il contratto di utilizzo, della specifica del componente.

Mentre una *interfaccia* definisce un insieme di servizi offerti al cliente, il *contratto di realizzazione* serve per definire dei vincoli sulla implementazione o sulla distribuzione. Ad esempio è possibile specificare che la realizzazione di un certo servizio deve avvenire attraverso l'interazione con un determinato componente.

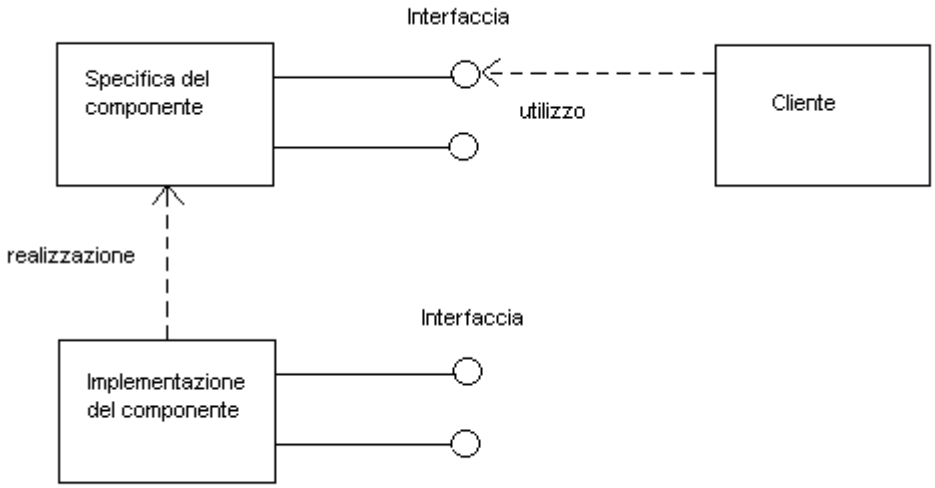


Figura 1.8: *contratto di utilizzo e contratto di realizzazione*

La specifica di un componente si compone di più parti, non tutte di interesse per il cliente. La definizione di quali interfacce deve offrire il componente è solo la parte di specifica che riguarda l'utente, una interfaccia non specifica come l'implementazione della interfaccia deve interagire con gli altri componenti per adempiere alle proprie responsabilità; ciò è definito nella specifica stessa del componente . In altre parole la specifica di una intrefaccia potrebbe implicare delle interazioni con altri componenti ma non descrive come queste avvengono, ciò è fatto in altre parti della specifica.

La ragione per tenere separate realizzazione e utilizzo è facilitare il cambiamento: un cambiamento nella realizzazione non implica un cambiamento nell'utilizzo e quindi non interessa il cliente. Questo è importante perché fornisce la possibilità

di cambiare la specifica che riguarda l'implementazione senza che debba essere modificato il modo in cui il cliente usa i servizi offerti dall'interfaccia

Essendo parti di software eseguibile e autoconsistente, i componenti incapsulano caratteristiche aggiuntive rispetto agli oggetti, quali meccanismi d'installazione, configurazione, attivazione, disattivazione, sicurezza e persistenza. Nelle piattaforme DOM tali caratteristiche sono supportate da servizi middleware mediante apposite API utilizzabili dal programmatore. Nelle piattaforme a componenti, invece, tali caratteristiche costituiscono un insieme di servizi di sistema predefiniti, e lo sviluppatore ha il solo compito di dichiarare le strategie con cui associare i servizi ad un componente in fase di programmazione o, più tipicamente, di distribuzione (deployment).

Le piattaforme a componenti sono solitamente basate sul concetto di contenitore (container), che consente ai progettisti di concentrarsi prevalentemente sulla logica dell'applicazione, separandola dalla logica dei servizi di sistema di cui un componente necessita. Un contenitore è un ambiente di esecuzione di componenti, in grado di ospitarli, gestirne il ciclo di vita, e gestire i servizi ad essi necessari.

Le tipologie d'interazioni tra componenti sono classificate e descritte in un insieme d'interfacce predefinite incluse nei contenitori. Ciò assicura che componenti sviluppati in maniera indipendente interagiscano in modo prevedibile e possano essere istanziati in ambienti standard di esecuzioni (i contenitori). Anche i contenitori sono classificati in un insieme di tipi standard, ciascuno dei quali adatto per specifici ruoli assunti dai componenti all'interno di una architettura software. Ogni tipo di contenitore mette a disposizione dei componenti ospitati le interfacce adeguate per il loro ruolo.

Un esempio di middleware a componenti è la tecnologia Java 2 Enterprise Edition (J2EE)di Sun MicroSystem.

1.3.8 I middleware di nuova generazione

Nuove tecnologie *middleware* si stanno affacciando sul mercato. In particolare, sono considerati *middleware* di nuova generazione le tecnologie *Web Services* e *Reflective Middleware*.

I *Web Services* trovano il loro maggior utilizzo nella rete Internet. rappresentano delle interfacce applicative disponibili su Internet, definite in un linguaggio standard W3C, il WSDL (*Web Services Description Language*). Essi sono resi disponibili e catalogati attraverso dei *repository* pubblici su Internet (UDDI, *Universal Description Discovery and Integration*) ed invocati utilizzando un protocollo di *middleware* leggero denominato SOAP (*Simple Object Access Protocol*). SOAP è un *middleware* che permette di fare sia chiamate RPC che *messaging*, descrive i messaggi in formato XML e li veicola su Internet via HTTP, riuscendo così a passare senza problemi attraverso i *firewall*.

La Figura 1.9 descrive lo scenario tipico d'invocazione di un *Web Service*: esso viene prima ricercato nei *repository* UDDI, la sua interfaccia viene recuperata dal *Web Service consumer* in formato WSDL e poi attraverso una chiamata SOAP invocato. Il *Web Service Consumer* ed il *Web Service Provider* devono perciò condividere soltanto la descrizione del servizio nel formato standard WSDL. L'unico vincolo all'utilizzo di un *Web Service* è l'accesso ad Internet e la disponibilità di un supporto SOAP per invocarli.

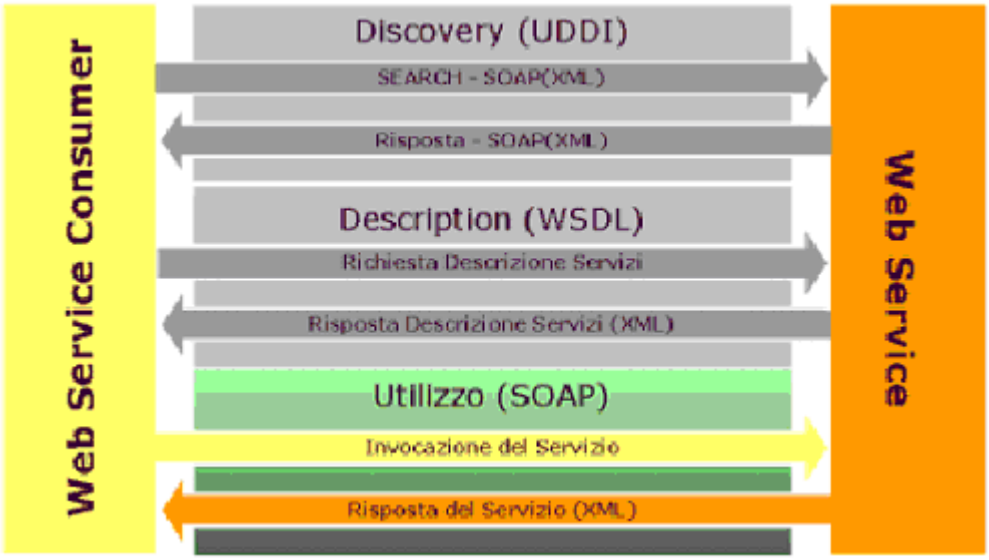


Figura 1.9- *Scenario di invocazione di un Web Service*

I *Web Services* più che a sostituire i *middleware* di prima generazione aspirano a risolvere in maniera definitiva i problemi di interoperabilità. In particolare, i *Web Services* rappresentano un *integration middleware*, cioè un *middleware* attraverso il quale integrare tra loro differenti *platform middleware* di prima generazione, i quali continuano ad essere lo strumento più utilizzato per la realizzazione di servizi distribuiti in ambienti classici.

Sono, però, una tecnologia ancora poco matura e non riescono a risolvere completamente i problemi d'interoperabilità, inoltre puntano a risolvere l'integrazione tra piattaforme preesistenti, essendo non idonee ad essere utilizzate in ambienti di *pervasive computing*.

La seconda tecnologia *middleware* di nuova generazione è costituita dai *Reflective Middleware*.

Il *Reflective Middleware* utilizza delle tecniche derivate dal *computational reflective* per aggiungere flessibilità al *middleware*.

Il *computational reflective* rappresenta la possibilità da parte di un sistema software di eseguire computazioni (meta-computazioni) in relazione a delle proprie strategie computazionali.

In particolare, il concetto di *reflection* si riferisce alla capacità di un sistema di esaminare il proprio comportamento e di modificarlo dinamicamente nell'arco dell'elaborazione. Un sistema *reflective* è in grado di fornire una rappresentazione del proprio comportamento, attraverso operazioni d'ispezione e di adattarlo in base a queste ultime.

Un *Reflective Middleware* si può definire come un "sistema *middleware* che fornisce un'ispezione ed un adattamento del proprio comportamento attraverso un'appropriata strategia computazionale".

Per le loro caratteristiche, i *Reflective Middleware* offrono un'elevata flessibilità in ambienti dinamici. Ciò nonostante, essa è una tecnologia poco matura e ancora in fase sperimentale.

CAPITOLO II

MODELLAZIONE E IMPLEMENTAZIONE DI BUSINESS RULES IN APPLICAZIONI A COMPONENTI

2.1 Introduzione

La tecnica del "design by contract" vista nel Capitolo 1, è molto usata nella progettazione di sistemi a componenti. Tale tecnica prevede la specifica nei contratti di utilizzo e di realizzazione di particolari requisiti chiamati *constraints* (invarianti, precondizioni, postcondizioni e guardie). Purtroppo le piattaforme *middleware* per lo sviluppo di applicazioni a componenti non mettono a disposizione del progettista strumenti per l'implementazione automatica di tali constraints. In questo capitolo saranno introdotte le *business rules*, che possono essere specificate attraverso *constraints*, e ne sarà mostrata una classificazione e un linguaggio per la modellazione. Successivamente, saranno analizzati i limiti degli strumenti attuali per la progettazione di sistemi a componenti. Infine, saranno proposti dei *patterns* per l'implementazione di *business rules*.

2.2 Le business rules

Le *business rules* (BRs) sono dichiarazioni di politiche o condizioni che devono essere soddisfatte [11].
Le BRs possono quindi esprimere i requisiti di un sistema software. In particolare, le BRs possono essere requisiti funzionali di alto livello specificabili nella fase di analisi come casi d'uso[3]. Le BRs possono essere anche rappresentate in altri modi, infatti, i casi d'uso sono user friendly ma non sono una rappresentazione particolarmente utile per la fase di implementazione [12],[13]; una migliore rappresentazione è attraverso *contratti* e *constraints* (vincoli sui dati o sui processi di una organizzazione).
Le BRs possono essere classificate in diversi modi, uno di questi riflette il modo in cui di solito gli analisti le specificano e prevede la divisione delle business rules inizialmente in due categorie: *contraint rules* e *derivation rules*.
Con riferimento alle tecniche di OOA, si può dire che un *constraint rules* specifica le politiche o le condizioni che vincolano la struttura o il comportamento di un oggetto; le *derivation rules* specificano politiche o condizioni per dedurre o elaborare concetti da altri concetti.
Questa prima classificazione può essere ulteriormente raffinata, come mostrato in Figura 2.1.

- Stimulus/Response rules

Le rules di tipo stimulus/response vincolano il compoprtamento di operazioni del dominio applicativo specificando clausole di tipo "**WHEN** (Evento)" ed "**IF (**condizione**)"** che devono essere soddisfatte per potere eseguire la suddetta operazione.

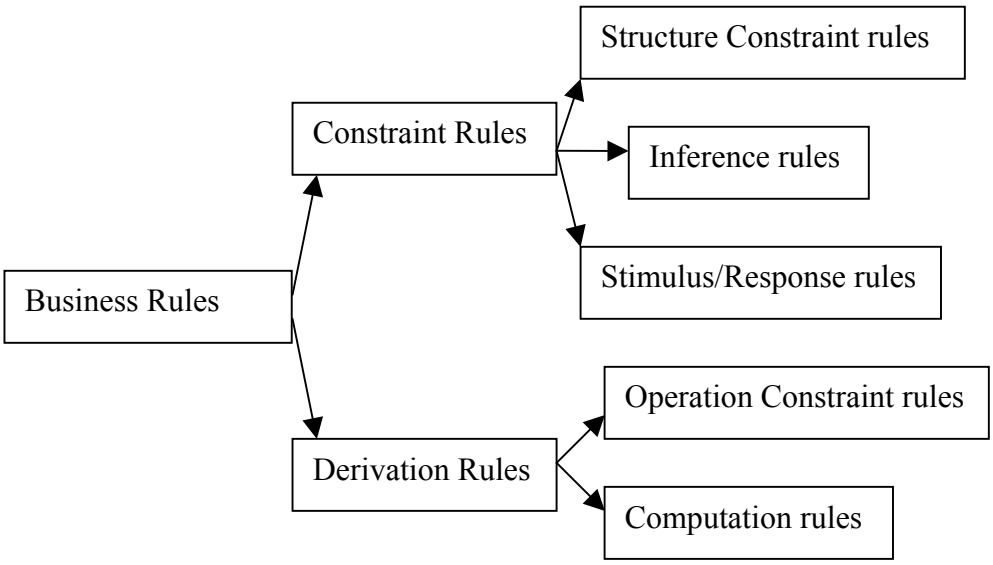


Figura 2.1- *classificazione delle business rules*

Tali rules hanno senso con riferimento a un particolare evento, in altre parole, il ruolo della clausola IF ha senso solo quando l'evento specificato nella clausola WHEN si verifica.
Un esempio di rule di tipo stimulus/response è:

WHEN *un libro è richiesto da un utente*
IF *la copia del libro è disponibile*
THEN *dai la copia all'utente*
Di questa categoria di constraints sono spesso usati quelli che vengono chiamati *guardie***.** Una guardia forza l'esecuzione di una operazione del componente ogni volta che una determinata condizione viene verificata.

- Operation Constraint rules

Le rules di questo tipo specificano quelle condizioni che devono essere verificate prima e dopo l'esecuzione di una operazione per assicurare che l'operazione sia corretta. Queste condizioni sono completamente indipendenti dall'evento che provoca l'invocazione della operazione.
Le condizioni che devono essere verificate affinchè l'operazione sia eseguita sono dette *precondizioni* e sono contenute in clausole del tipo **ONLY IF** (*condizione*). Ad esempio una precondizione è:

Promuovi l'impiegato al ruolo di manager
***ONLY IF** l'impiegato non è un manager*

Le condizioni che invece devono essere soddisfatte dall'esecuzione dell' operazione vengono chiamate *postcondizioni* e sono contenute in clausole di tipo **IS CORRECT**
ONLY IF (*condizione*). Ad esempio una postcondizione è:

*La promozione di un impiegato a manager **IS CORRECT***
***ONLY IF** questo impiegato è un manager*

Bertrand Meyer è stato uno dei primi ad ad asserire che queste paricolari business rules possono essere viste come parti di un contratto che lega un metodo e chi lo invoca, in altre parole è come se l'operazione dicesse: "se tu mi invochi con la precondizione soddisfatta, prometto di lasciare l'oggetto nello stato che è specificato nella postcondizione" [14].

- Structure Constraint Rules

Le structure constraint rules specificano politiche o condizioni sulla struttura degli oggetti e sulle loro relazioni, attraverso clausole di tipo "**IT MUST ALWAYS HOLD THAT** (condizione)", che non possono essere violate. Un constraint di questo tipo può vincolare la struttura di un oggetto in molti modi ad esempio può vincolare il valore di un attributo:

IT MUST ALWAYS HOLD THAT

il salario di un impiegato non può essere maggiore del salario del suo manager

oppure può vincolarne la cardinalità:

IT MUST ALWAYS HOLD THAT

Il numero dei membri della suprema corte di giustizia americana non deve essere maggiore di 9

o può vincolare la molteplicità di una relazione, ad esempio in una relazione cliente-ordine:

IT MUST ALWAYS HOLD THAT

un cliente non può fare più di 7 ordini

Come si vede le structure-constraint rules non fanno riferimento ad operazioni perché devono essere valide in ogni circostanza operativa, cioè, quando la stato di un oggetto cambia, la structure constraint rule deve essere verificata.
Questi particolari business-rules sono anche chiamati *invarianti.*

- Inference Rules

Le inference rules specificano che se certi concetti sono veri, allora una certa conclusione può essere dedotta. Queste rules sono generalmene associati a sistemi esperti. Un esempio di inferece rule è:

IF *un oggetto è un Impiegato*
THEN *questo oggetto è una Persona*

- Computation Rules

Le inference rule permettono deduzioni semplicemente accedendo a concetti esistenti. Invece, le computation rules derivano i propri risultati tramite l'esecuzione di algoritmi, ad esempio:

L'oggetto Donna ***IS COMPUTED AS FOLLOWS***
l' intersezione di tutte gli oggetti Umani Femmina e Umani Adulti

Nell'ambito dei sistemi software tradizionali le business rules più utilizzate nella fase di specifica sono i constraint

2.3 Modellazione di business rules in OCL

Uno dei maggiori vantaggi derivanti dall'utilizzo delle BRs per la specifica dei requisiti è la grande chiarezza esplicativa. Ciò ha spinto l' OMG ad aggiungere dei meccanismi in UML per potere arricchire i diagrammi con le BRs.

Un diagramma UML come il *class diagram*, non è fine abbastanza per evidenziare tutti gli aspetti di una attività di specifica ed è per tale motivo che l'integrazione delle business rules con i diagrammi risulta particolarmente vantaggiosa. Il meccanismo usato in UML per aggiungere business rules a un diagramma è il *constraint*.

Un *constraint* si rappresenta graficamente come una stringa racchiusa fra parentesi e posta vicino all'elemento a cui si riferisce, sia esso una classe o una relazione.

L'espressione di una business rules attraverso un *constraint* può inizialmente avvenire mediante il linguaggio naturale ma nel momento in cui bisogna essere precisi, sia perché si deve passare da una fase in cui i modelli fatti del dominio applicativo hanno scopo puramente esplicativo a una fase in cui i modelli sono gli artifatti di base per passare a una fase di progetto, sia perché si vuole utilizzare qualche strumento automatico di sviluppo che a partire dai modelli del dominio applicativo generi del codice, bisogna eliminare le ambiguità di tale linguaggio. Attualmente, l'organismo di standardizzazione OMG (object mangment group) sta proponendo l'OCL (object constraint language).Questo linguaggio ha interessanti proprietà: i)è un linguaggio formale molto semplice da usare in quanto non si basa su costrutti complessi; ii) permette la produzione di modelli testuali.

OCL può essere usato per diversi scopi:

- per specificare invarianti su attributi di strutture

- per specificare pre- e post condizioni su operazioni e metodi
- come linguaggio di navigazione per le relationship
- per specificare Business Rules in genere

OCL è puramente esplicativo, di conseguenza, una espressione non ha nessun effetto sul sistema software che si sta modellando, ciò significa che lo stato del sistema non cambierà a causa della valutazione di una espressione sebbene questa sia usata per specificare un cambiamento nello stato (come, ad esempio, in una postcondizione).
OCL è *tipizzato* quindi le espressioni devono rispettare innanzitutto le regole sui tipi di dato; ogni *Classifier* definito in UML rappresenta un distinto tipo OCL ma sono previsti anche un insieme di altri tipi predefiniti.

2.3.1 Parole chiave

OCL possiede un certo numero di parole riservate. Ciò significa che queste parole non possono trovarsi in qualsiasi posizione all'interno di una espressione. La lista delle keyword è mostrata in seguito:

If	implies
then	endpackage
else	package
endif	context
not	def
Let	inv
Or	pre

and	post
xor	in

- context

La keyword *context* introduce il contesto per l'espressione OCL. Ad esempio, con riferimento al class diagram in Figura 2.2 una espressione OCL riguardante la classe Company deve essere preceduta da :

context Company …..

….. espressione …..

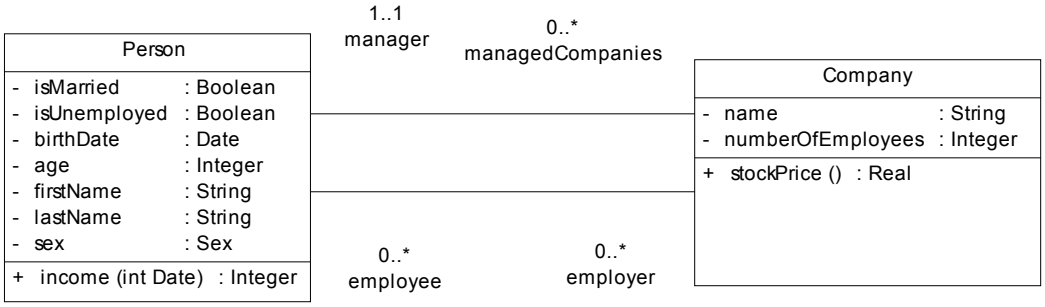


Figura 2.2- *class diagram di riferimento*

- self

Ogni espressione OCL si riferisce a un certo contesto, la keyword ***self*** serve per riferirsi alla particolare istanza del contesto specificato. Per esempio, con riferimento alla Figura 2.2, se il contesto è Company, allora self si riferisce alla istanza di Company:

context Company ….

…..**self**.name…..

- package – endpackage

La specifica del contesto è precisa quando il package, a cui appartiene il *Classifier* coinvolto nella espressione, è deducibile facilmente dalle condizioni del diagramma UML. Invece, se si vuole specificare esplicitamente il package del *classifier* si possono usare le keywords *package* e *endpackage* :

package Package::SubPackage

context X1 …..
……

context X2 ….
……..

endpackage

- Inv

Per specificare un invariante si usa la keyword *inv***.** Con riferimento alla Figura 2.2, se si vuole specificare un invariante per vincolare *number of employees* a un valore maggiore di 50 si può usare la seguente espressione OCL:

context Company

inv: self.numberOfEmployees > 50

- ***Pre*** e ***Post***

Una espressione OCL può specificare una precondizione o una postcondizione utilizzando rispettivamente le keyword ***pre:*** e ***post:***. Con ***self*** , in questo caso, si

specifica una istanza del *Classifier* che mostra l'operazione o il metodo a cui si rivolgono pre e post.
In questo tipo di espressioni, in cui c'è il coinvolgimento di operazioni, devono essere specificati il contesto nonché la signature della operazione questo per potere fare riferimento, all'interno della espressione, ai parametri della operazione stessa.
Con la parola riservata *result* si può indicare il risultato della operazione.
Un esempio di postcondizione, con riferimento alla Figura 2.2 è:

context Person::income(d : date) : Integer
post: *result* = 5000

2.3.2 Tipi predefiniti

In OCL, è disponibile un numero di tipi di dato predefiniti utilizzabili dall'analista in qualsiasi momento. I tipi predefiniti e alcune delle operazioni definite su di essi sono mostrati nella Tabella 2.1.

Tipo	Operazioni
Boolean	and, or, xor, not, implies, if-then-else
Integer	*, +, -, / , abs()
Real	*, +, -, /, floor()
String	toUpper(), concat()
Collection	Size(),includes(OclAny):Boolean, excludes(OclAny):Boolean, isEmpty():Boolean, Sum():T, isUnique():Boolean
Set	union(set2: Set(T)):Set(T) , intersection(set2:Set(T)):Set(T), including(T):Set(t), excluding(T):Set(t),

	select(OclExpression):Set(T)
Bag	union(bag2: Bag(T)):Bag(T) , intersection(bag2:Bag(T)):Bag(T), including(T):Bag(t), excluding(T):Bag(t), select(OclExpression):Bag(T)
Sequence	Count(T):Integer, union(Sequence(T)):Sequence(T), append(T):Sequence(T), prepend(T):Sequence(T), at(Integer):T

Tabella 2.1 : *tipi predefiniti e operazioni principali*

Il tipo *Collection* fornisce un grande numero di operazioni per permettere all'analista di manipolare collezioni di dati di un tipo generico T.
Il tipo Collection è un tipo astratto i cui sottotipi rappresentano collezioni di dati concrete. OCL prevede tra sotto tipi di *Collection*: *Set, Sequence e Bag*.
Il tipo *Set* rappresenta l'insieme matematio, esso non contiene duplicati.
Il tipo *Bag* è come il *Set* ma può contenere duplicati. Le operazioni definite sono essenzialmente equivalenti a quelle definite sul tipo Set.
Il tipo *Sequence* è come un *Bag* ma con elementi ordinati per posizione.
Su questi tipi sono definite operazioni algebriche di intersezione, unione, più operazioni per la gestione di raccolte di dati come operazioni per la ricerca di elementi, l'aggiunta ecc.
La specifica di uno di questi tipi può avvenire con espressioni del tipo:
Set { 1, 2 , 4 , 88 }
Sequence { 'mela' , 'arancia', 'fragola' }
Bag { 1 , 2, 4, 2 }

I tipi *Set*, *Bag* e *Sequence* sono utilizzati sopreattuto quando le espressioni OCL riguardano le associazioni e la loro navigazione. Partendo da uno specifico

oggetto, è possibile navigare una associazione sul class diagram per riferirsi ad altri oggetti e alle loro proprietà, per fare ciò nelle espressioni si usa il ruolo che ha l'oggetto dall'altro lato della associazione. Il valore di questa espressione è un Set di oggetti dello stesso tipo dell'oggetto all'altro lato della associazione.
Ad esempio, facendo riferimento alla Figura 2.2, si può scrivere:

context Company
inv: self.manager.isUnemployed = false
inv: self.employee → **notEmpty()**

Nel primo invariante self.manager è un oggetto di tipo Person perché la molteplicità della associazione è 1. Nel secondo invariante self.employee sarà un Set di Persons perché la molteplicità è 0..*; su questo Set si possono applicare le operazioni predefinite, in questo caso notEmpty():Boolean.
Quando la associazione sul class diagram è segnata con { ordered } il risultato della navigazione è di tipo Sequence.
Le operazioni sui sottotipi di *Collection* (*Set*, *Bag*, *Sequence*) sono invocate tramite il simbolo →.
Ai tipi predefiniti mostrati in tabella se ne aggiungono altri 3 : ***OclType***, ***OclAny***, ***OclExpression*.**

- OclType
Tutti i tipi definiti in UML e quelli predefiniti in OCL appartengono a un tipo. Questo tipo è una istanza del tipo OCL chiamato *OclType*. L'accesso a questo tipo permette all'analista un limitato accesso al meta-livello del modello, ciò può essere utile per necessita di modellazioni molto avanzate.
Una istanza del tipo *OclType* viene chiamata semplicemente *type*, questa istanza ha delle proprietà alcune delle quali sono:

type.name(): String -- il nome di type

type.attributes(): Set (String) -- il set di nomi degli attributi di type
type.associationEnds(): Set(String) – il set di nomi dei Classifier associati a type

- OclAny

Il tipo *OclAny* è il supertipo di tutti i tipi del modello UML e di quelli predefiniti da OCL. Le proprietà di un tipo *OclAny* sono disponibili in ogni oggetto nelle espressioni OCL.
Una istanza del tipo *OclAny* viene chiamato *object*, alcune delle sue proprietà sono:

object = (object2: OclAny) : Boolean -- vero se object e lo stesso di object2
object.oclIsKindOf(type : OclType) : Boolean – vero se type è uno dei tipi di object

- OclExpression

Ogni espressione OCL è un oggetto nel contesto di OCL. Il tipo di questo oggetto è *OclExpression*. Questo tipo e le sue proprietà servono per definire la semantica dell'espressione stessa.

2.4 La progettazione di sistemi component-based e le business rules: i limiti degli strumenti attuali

Le tecnologie per lo sviluppo e la distribuzione di oggetti distribuiti cooperanti come CORBA, DCOM, e più recentemente Web Services, prevedono la definizione delle interfacce dei componenti. In particolare, CORBA, che è stato scelto per questa tesi come piattaforma per lo sviluppo di componenti, prevede che la definizione delle interfacce avvenga attraverso un linguaggio apposito chiamato IDL (*interface definition language*). Purtroppo, IDL non permette ai progettisti di formalizzare, nella definizione delle interfacce, aspetti quali *business rules* e *relationships* come invece è previsto nel *design by contract*. Di conseguenza, negli skeleton generati automaticamente dal compilatore IDL, non c'è traccia di questi aspetti così che la loro implementazione è completamente a carico dei programmatori i quali hanno due possibili scelte: o modificare gli skeleton o tenere conto di questi aspetti nella realizzazione dei componenti (cioè arricchendo le operazioni previste nelle interfacce e aggiungendone altre) basandosi sugli artifatti prodotti dalla fase di modellazione. Quindi la possibilità di avere strumenti e metodologie per implementare automaticamente le proprietà modellate, ad esempio in UML, faciliterebbe notevolmente la fase di sviluppo oltre a ridurre la possibilità di commettere errori.

2.5 Patterns per l'implementazione di business rules

Per l'implementazione delle business rules sui componenti si possono seguire dei *patterns*[22].

In particolare, in questo paragrafo, sono presentati dei possibili *patterns* per implementare dei constraints che non prevedono la navigazione fra componenti.

L'implementazione è riferita a componenti sviluppati in C++, inoltre si è supposto che per ogni attributo di un componente sia prevista una operazione *set* per l'aggiornamento.

2.5.1 Invariante

Un invariante è una condizione che deve essere sempre soddisfatta. Questa, generalmente, vincola il valore di un attributo, quindi può essere implementata modificando l'operazione set relativa a quell'attributo in modo che venga esaminata la condizione specificata prima che il valore sia aggiornato.
Ad esempio, supponiamo di avere un componente specificato come in Figura 2.3:

Figura 2.3 *specifica del componente*

per un invariante formalizzato nel modo seguente:

context C -- componente C
inv: self. Attribute_1 > 0 -- invariante sull'attributo Attribute_1

l'implementazione prevede:

1. definizione di una nuova operazione, chiamata *Inv()*, per verificare che il valore da dare all'attributo verifichi la condizione

2. modifica dell'operazione set in modo che l'aggiornamento sia effettuato solo se la condizione specificata dall'invariante è verificata
3.

quindi l'implementazione del componente subisce le modifiche mostrate in Figura 2.4.

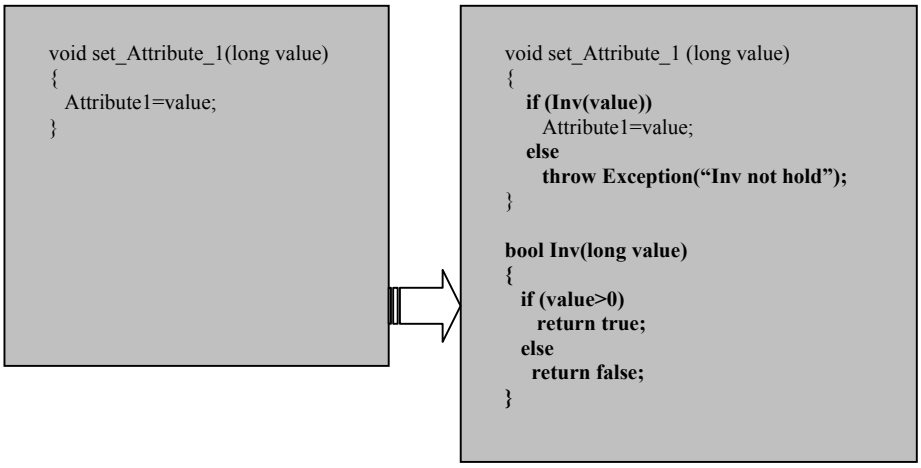


Figura 2.4 -*implementazione di un invariante*

i cambiamenti fatti nella specifica del componente sono mostrati in Figura 2.5, mentre il flusso di controllo dell' operazione di set, modificato per tenere conto della verifica dell'invariante, è mostrato in Figura 2.6.

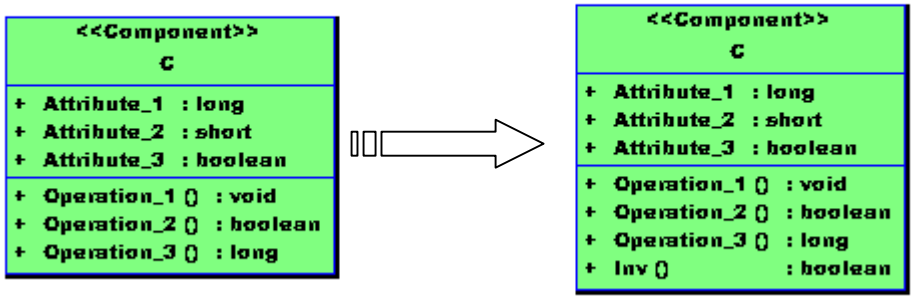


Figura 2.5 - *modifica della specifica*

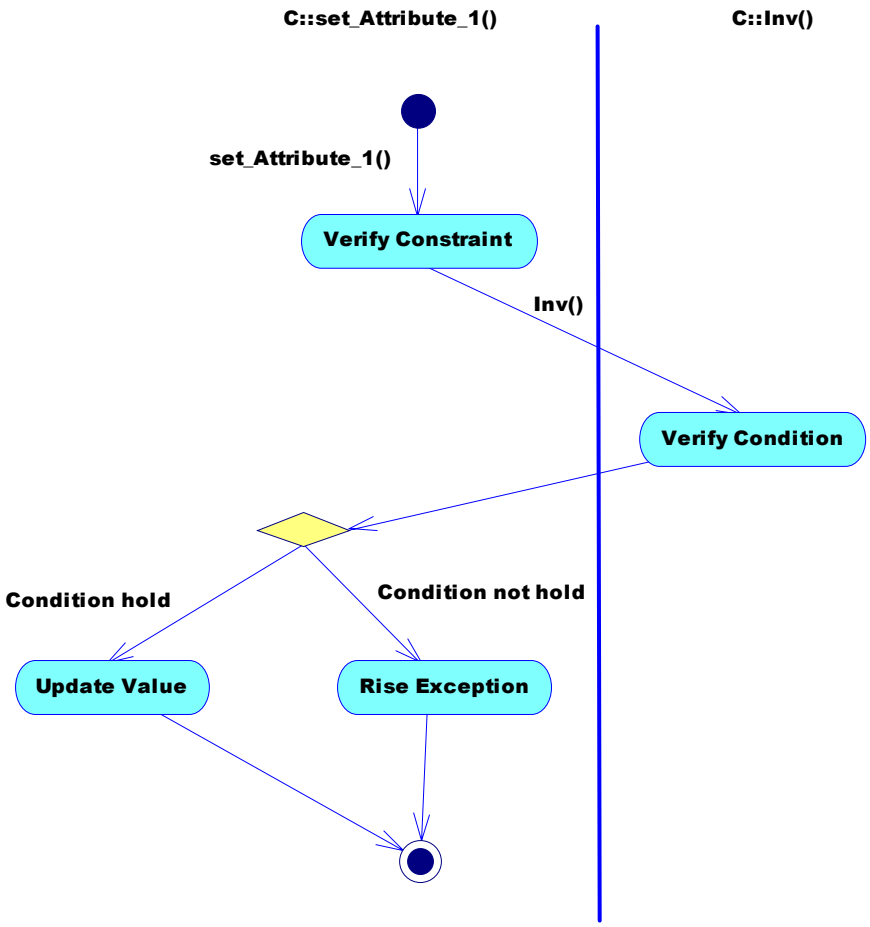


Figura 2.6 – *flusso di controllo della operazione set_Attribute_1(long value)*

come si vede, dopo l'invocazione del metodo *set* viene chiamata l'operazione *Inv* per verificare che il nuovo valore immesso verifichi la condizione; se questà è verifica l'attributo viene aggiornato altrimenti viene lanciata una eccezione per segnalare che l'operazione non è andata a buon fine per la violazione dell'invariante.

2.5.2 Precondizione

Una precondizione è una condizione che deve essere soddisfatta prima di eseguire una operazione e, tipicamente, dipende dal valore di uno o più attributi del componente. Il pattern per l'implementazione della precondizione è simile a quello seguito per gli invarianti, infatti, anche in questo caso, prima di eseguire l'operazione bisogna valutare una condizione, quindi bisogna aggiungere un metodo per la verifica della condizione e modificare il metodo a cui si riferisce la precondizione in modo da eseguire inizialmente la verifica della condizione.
Ad esempio, sempre con riferimento al componente specificato in precedenza, per la seguente precondizione:

context A::Operation_2()
pre: ((**self.**Attribute_2==true) and (**self.**Attribute_1>0))

l' implementazione prevede:

1. la aggiunta al componente di una operazione per la verifica della condizione
2. la modifica della operazione Operation_2() per tenere conto della verifica della precondizione

quindi l'implementazione del componente subisce le modifiche mostrate in Figura 2.7.

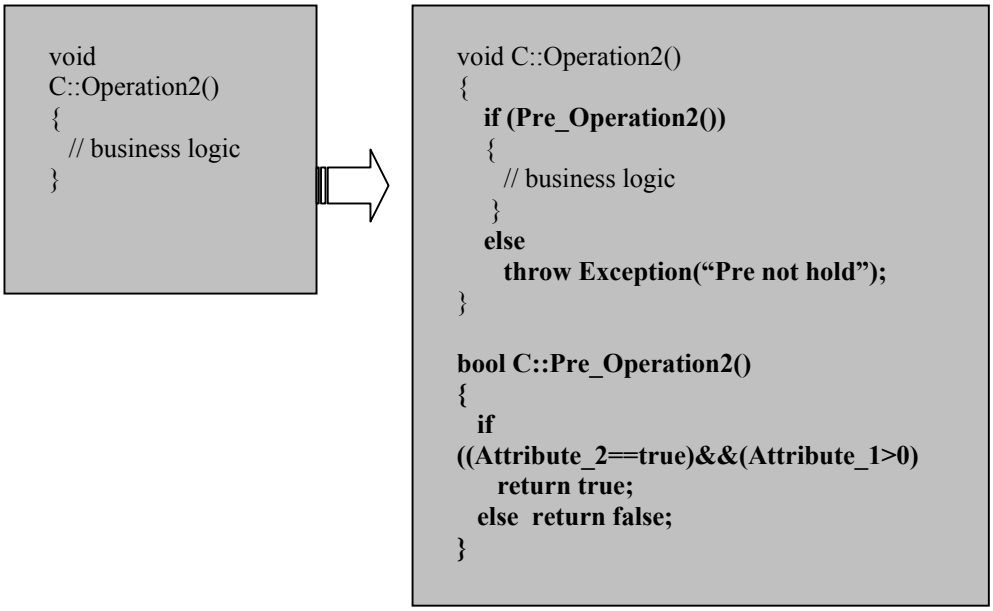


Figura 2.7- *implementazione della precondizione*

Di conseguenza, la specifica del componente subisce le modifiche mostrate in Figura 2.8, mentre il flusso di controllo della generica operazione di set diventa quello di Figura 2.9.

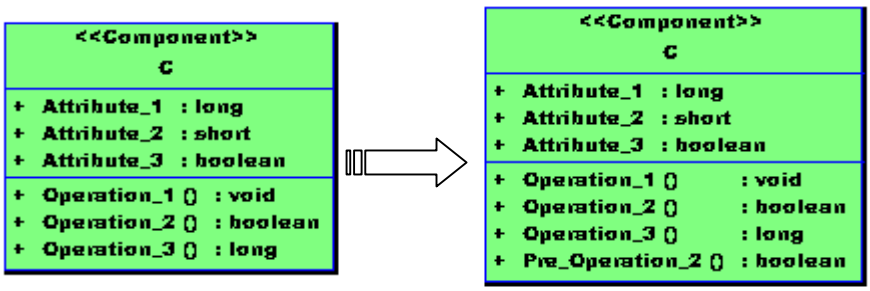


Figura 2.8 *-specifica del componente*

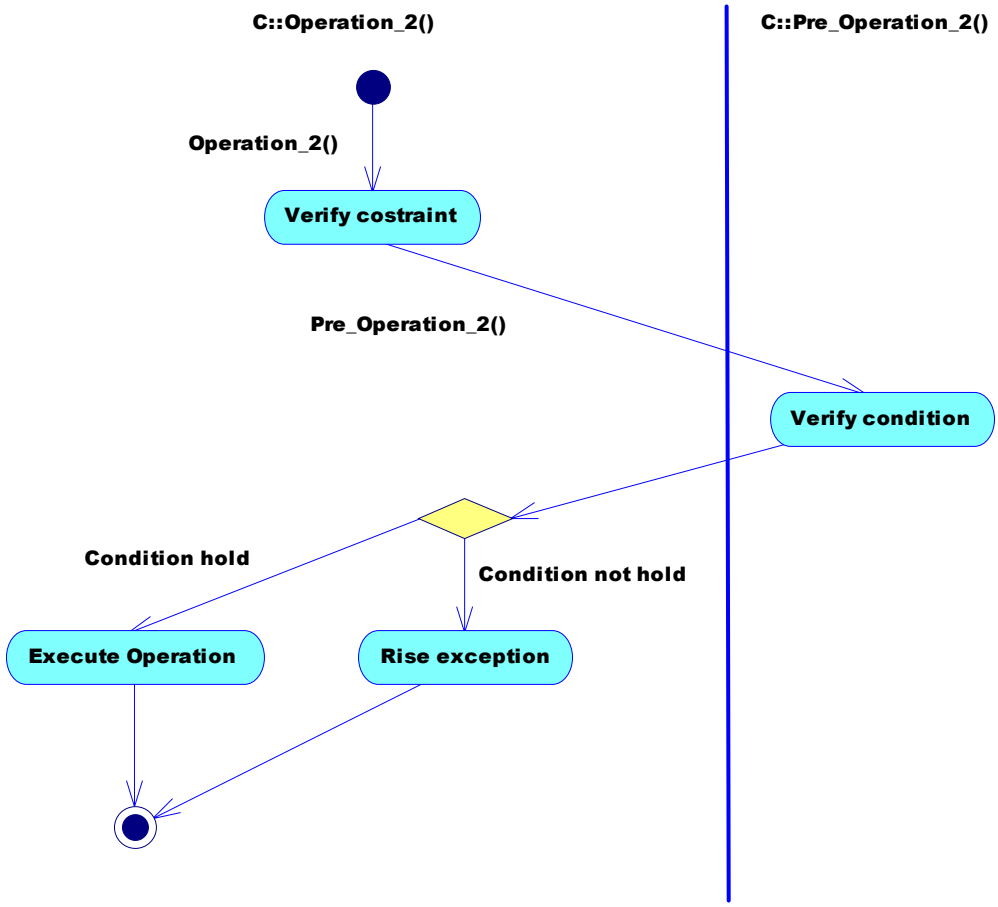


Figura 2.9- *flusso di controllo della operazione Operation_2()*

2.5.3 Postcondizione

Una postcondizione su un metodo definisce, in genere, il valore di ritorno di tale metodo o le modifiche che questo metodo apporta sugli attributi del componente che lo implementa. In sostanza una postcondizione definisce la business logic di un metodo quindi la sua implemenntazione consiste semplicemente nel "riempire" il metodo con la postcondizione. Ad esempio, per la seguente postcondizione:

context C::Operation_3()

post: **if** (**self.**Attribute_3 > 0)

then

result self. Attribute_1***self.**Attribute_3

else

result (-1)***self.**Attribute_1***self.**Attribute_3

endif

l'implementazione può avvenire come mostrato in Figura 2.10

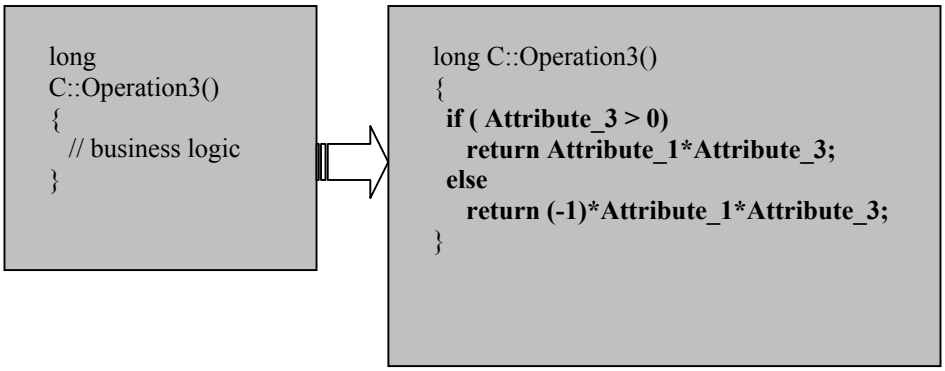


Figura 2.10 – *implementazione della postcondizione*

quindi la specifica del componente subisce la modifica mostrata in Figura 2.11.

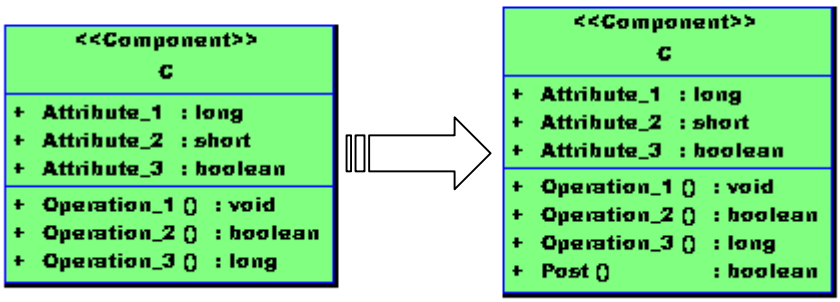


figura 2.11- *specifica del componente*

2.5.4 Guardia

Una guardia forza l'esecuzione di una operazione del componente ogni volta che una determinata condizione viene verificata. Generalmente, tale condizione è composta da predicati che coinvolgono gli attributi del componente, quindi, l'implementazione di una guardia su un componente deve prevedere il controllo della condizione ogni volta che uno degli attributi presenti in essa viene modifcato.
Ad esempio, con riferimento al componente specificato in Figura 2.3, data la seguente guardia:

context C
guard: if ((**self.**Attribute_1 < 0) **or** (**self.**Attribute_2 < 0))
then
execute self. Operation_1()
endif

l' implementazione consiste in:

1. definire una nuova operazione per verificare la condizione
2. modificare l'operazione di set per gli attributi coinvolti nella condizione in modo da controllare la condizione ogni volta che essi vengono modificati

quindi l'implementazione del componente cambia come mostrato in Figura 2.11.

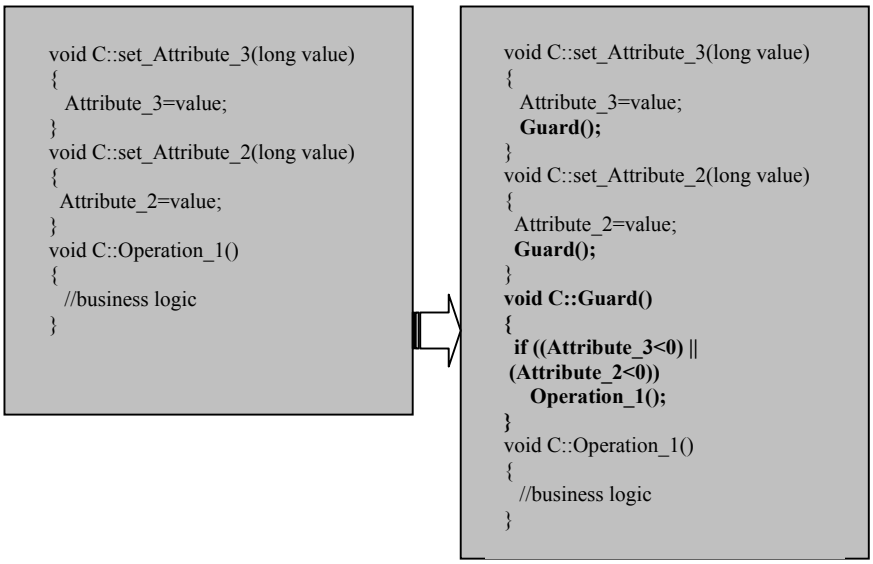


Figura 2.11- *implementazione della guardia*

Di conseguenza, la specifica del componente subisce le modifiche mostrate in Figura 2.12, mentre il flusso di controllo della generica operazione di set diventa quello di Figura 2.13.

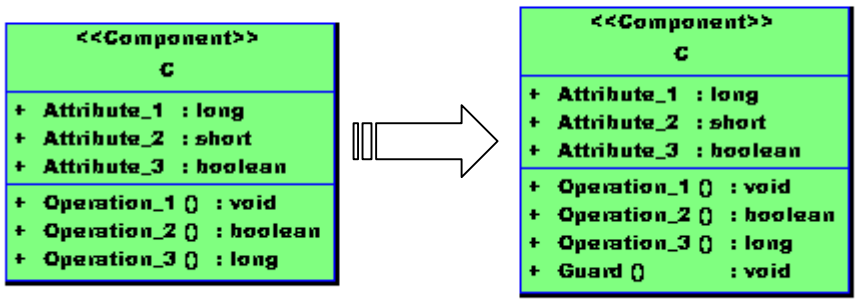


Figura 2.12 – *specifica del componente*

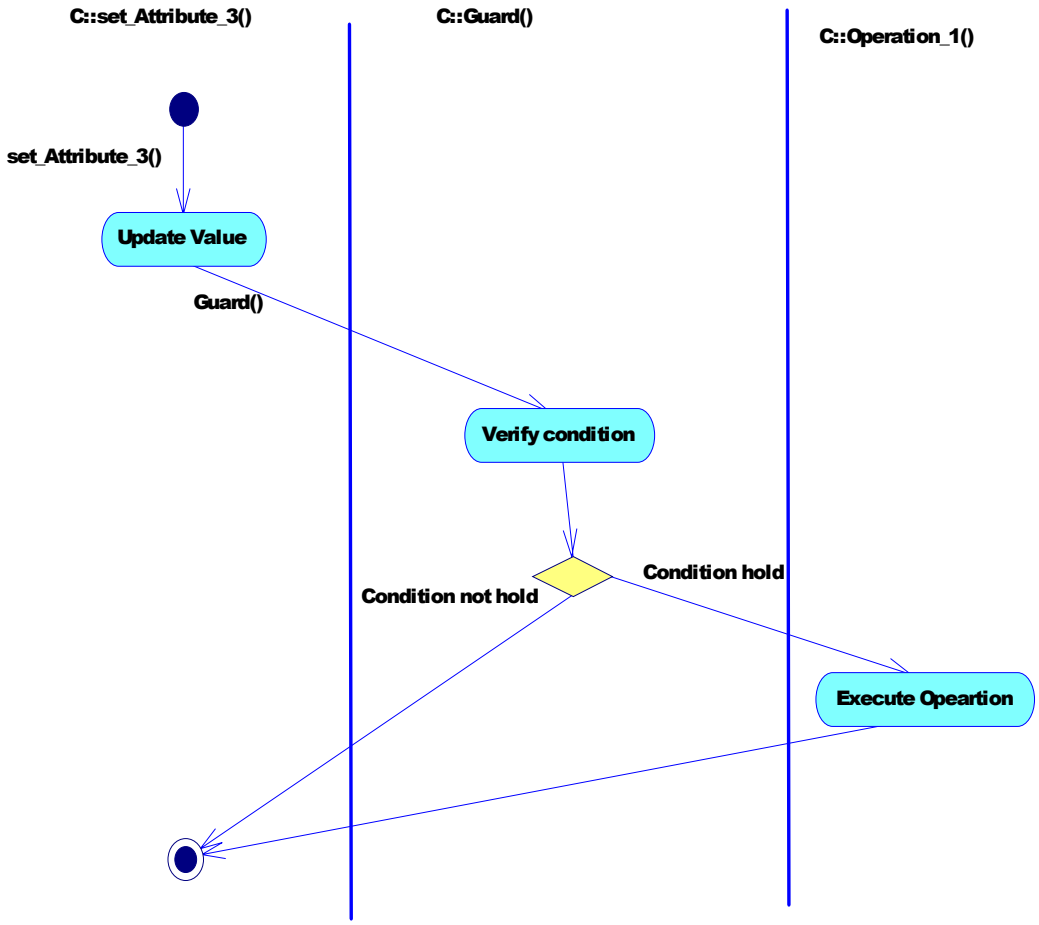


Figura 2.13 – *flusso di controllo della operazione*

I patterns esaminati in precedenza saranno usati come riferimento durante il progetto dei tools (Capitolo 3) per l'implementazione automatica dei constraints.

CAPITOLO III
IL LINGUAGGIO SDL E LA PROGETTAZIONE DEI TOOLS

3.1 Introduzione

Il problema dello sviluppo di applicazioni a componenti con CORBA, evidenziato nei capitoli precedenti, è che il linguaggio di definizione delle interfacce IDL non premette di specificare proprietà quali *business rules* e *relationships* modellate, in fase di analisi, mediante diagrammi UML. Di conseguenza, nel codice generato automaticamente dai compilatori IDL non c'è traccia di queste proprietà che devono, pertanto, essere implementate dagli sviluppatori, causando un aumento dei tempi di sviluppo e del rischio di errori. Nasce quindi la necessità utilizzare dei tools capaci di implementare automaticamente tali proprietà a partire dalla loro specifica.

In questo capitolo verrà mostrato inizialmente un linguaggio testuale per la specifica di *business rules* e *relationship*, successivamente sarà mostrato lo sviluppo di un tool chiamato CoCoGen (component constraint generator) che, a partire da tale specifica, modifica il codice generato dai compilatori IDL per le piattaforme di sviluppo basate su CORBA.

Infine, verrà mostrato lo sviluppo di un tool chiamato CoCoMod (component constraint generator) per la progettazione visuale di applicazioni a componenti.

3.2 Il linguaggio SDL

Per implementare automaticamente business rules e le relationships nelle applicazioni a componenti c'è bisogno innanzitutto di potere esprimere tali proprietà. I modelli UML non soddisfano questo bisogno perché sono dei modelli visuali e quindi difficilmente gestibili da un tool automatico. Per questo motivo si preferiscono modelli testuali. Sebbene alcuni software di progettazione forniscano anche descrizioni testuali dei modelli visuali, la soluzione di usare queste descrizioni è stata scartata perché vincolerebbe il tool di implementazione a un determintato software di progettazione.
Per ottenere modelli testuali ci sono due possibilità:

1. estendere IDL per potere esprimere anche business rules e relationship;
2. definire un nuovo linguaggio testuale;

In questo studio si è scelto di definire un nuovo linguaggio chiamato *Structure Definition Language* (SDL). La possibilità di estendere IDL è stata scartata in quanto avrebbe richiesto la modifica di un linguaggio standard e avrebbe cambiato il suo obiettivo principale: specificare l'interfaccia dei componenti in maniera indipendente dal linguaggio di implementazionee dalla struttura del componente. Il linguaggio SDL deve colmare il gap semantico esistente fra i modelli visuali e IDL.
Come detto nel Capitolo 2, UML è stato recentemente esteso, attraverso l'OCL, per potere formalizzare alcuni tipi di constraints. In particolare, OCL permette di formalizzare alcuni constraints come invarianti, precondizioni e postcondizioni che sono fondamentali per esprimere una business rules; in realtà sarebbe necessario anche potere esprimere le guardie. IDL, invece, è un linguaggio per la definizione delle interfacce offerte da un componente e permette soltanto di esprimere operazioni, attributi e ereditarietà del componente.

In Figura 3.1 è mostrato il gap semantico fra IDL e la coppia <UML,OCL>.

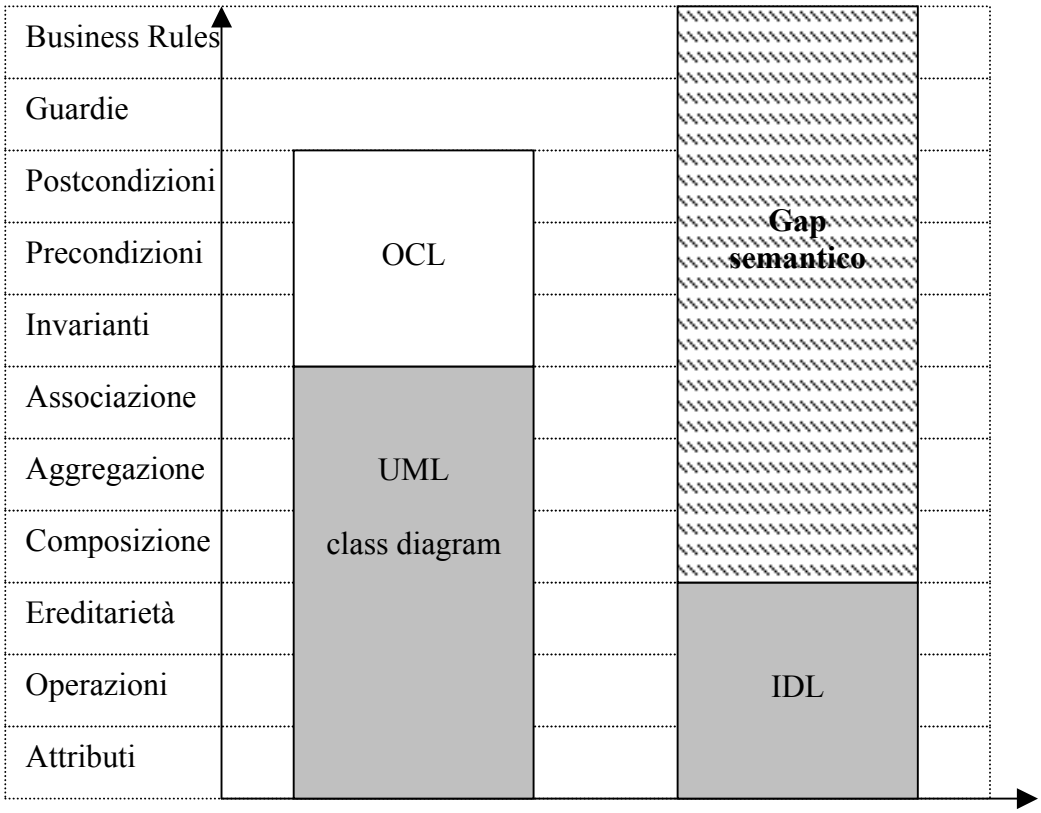


Figura 3.1 – *SDL structure definition language*

Il nuovo linguaggio di modellazione SDL, deve permettere di formalizzare quelle proprietà esprimibili mediante la coppia <UML,OCL> e che non è possibile specificare in IDL. Inoltre, deve permettere di modellare le guardie.

OCL ha due qualità fondamentali: essere formale, ma semplice, e essere testuale; per tale motivo SDL lo incorpora ne eredita lo stile sintattico.

Come si vede dalla Figura 3.2, SDL colma il gap semantico evidenziato in precedenza.

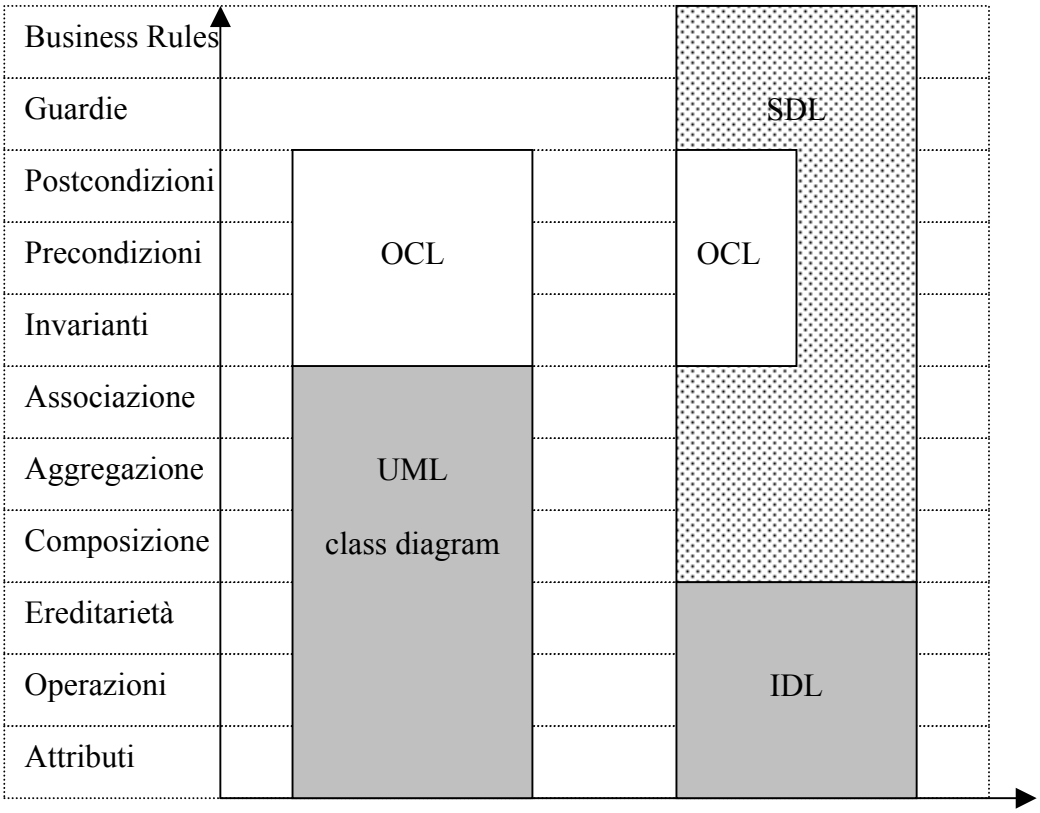


Figura 3.2 – *La semantica di SDL*

In Tabella 3.1 sono mostrate alcune delle parole chiave di SDL con il rispettivo significato e l'indicazione di quando fanno parte dello standard OCL o meno.

Parola chiave	**Descrizione**	**OCL**
Context	indica la classe o l'operazione a cui si riferisce la business rule	*
inv:	indica un invariante	*
pre:	indica una precondizione	*

post:	indica una postcondizione	*
guard:	indica una guardia	
Association	indica una associazione fra due classi	
Toward	indica l'end-point di una associazione	
Aggregation	indica una aggregazione fra due classi	
Aggregate	indica quali sono gli oggetti aggregati in una associazione	
Aggregatedto	indica qual è l'oggetto aggregato in una aggregazione	
Compositon	indica una composizione fra classi	
Compose	indica qual è l'oggetto composto in una composizione	
Composed	indica quali sono i componenti in una composizione	
Multiplicity	indica la molteplicità di una relazione	
Self	indica l'istanza corrente di una classe	*
Execute	indica una operazione da eseguire	
@pre	indica il valore precedente	*

	di una variabile	

Tabella 3.1 – *le parole chiave di SDL*

Per meglio comprendere la sintassi di SDL e la sua semantica è possibile riferirsi al modello di Figura 3.3 e esaminare il corrispondente modello SDL riportato in seguito.

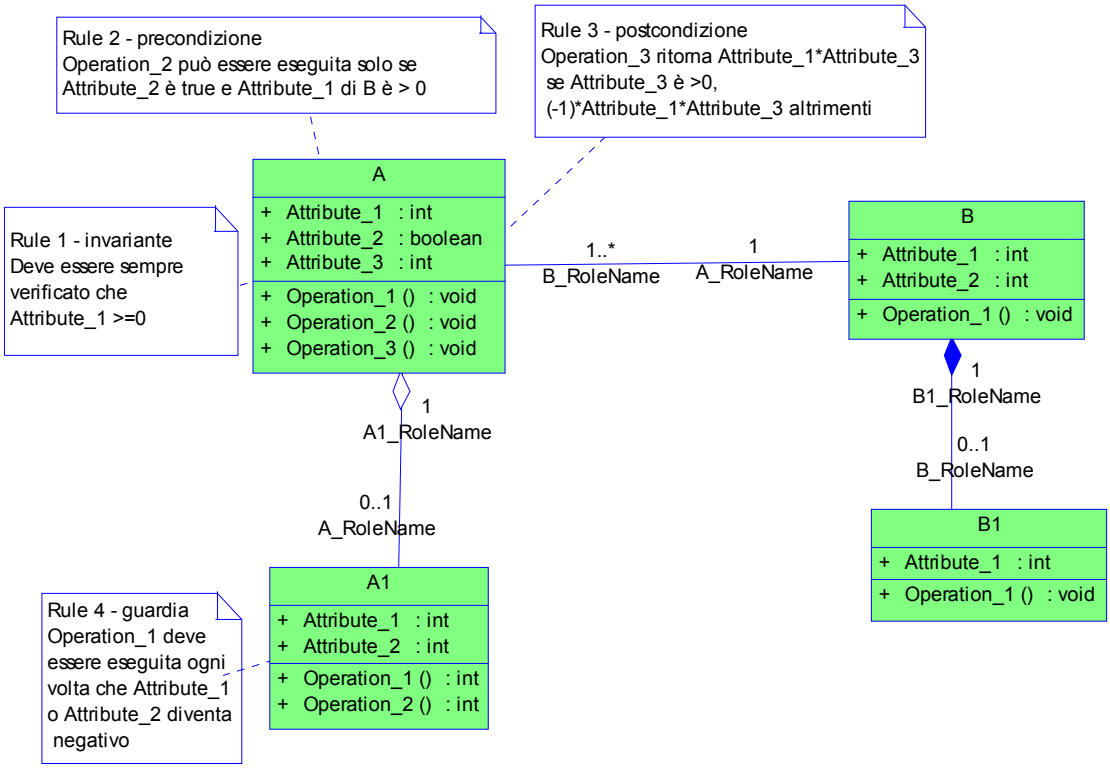


Figura 3.3 – UML class diagram

--Modello SDL

context A
inv: self.Attribute_1 > 0
association: A_RoleName **multiplicity** 1 **toward** B
aggregation: A_RoleName **multiplicity** 0 to 1 **aggregate** A1

context A::Operation_2()
pre: ((**self.**Attribute_2==true) and (**self.**B.Attribute_1>0))

context A::Operation_3()

post: if (**self.**Attribute_3>0)
then
result **self.**Attribute_1***self.**Attribute_3
else
result (-1)***self.**Attribute_1***self.**Attribute_3
endif

context A1
guard: if (**self.**Attribute_1<0) or (**self**.Attribute_2<0)
then **execute self.**Operation_1()
endif
aggregation: A1_RoleName **multiplicity** 1 **aggregatedt**o A

context B
association: B_RoleName **multiplicity** 1 to many **toward** A
composition: B_RoleName **multiplicity** 0 to 1 **composed** by B1

context B1
composition: B1_RoleName **multiplicity** 1 **compose** B

3.3 Sviluppo del tool CoCoGen

3.3.1 Fase di analisi

Il tool CoCoGen deve essere in grado di implementare business rules e relationships nel codice generato automaticamente dai compilatori IDL per le piattaforme middleware aderenti allo standard CORBA come Java IDL[5][21], TAO[17], ORBIX2000™[16], Inprise Visibroker[5] etc. Le business rules e le relationships devono essere espresse mediante un modello SDL. Quindi, il processo che deve seguire il progettista prima di eseguire il tool è quello mostrato in Figura 3.4.

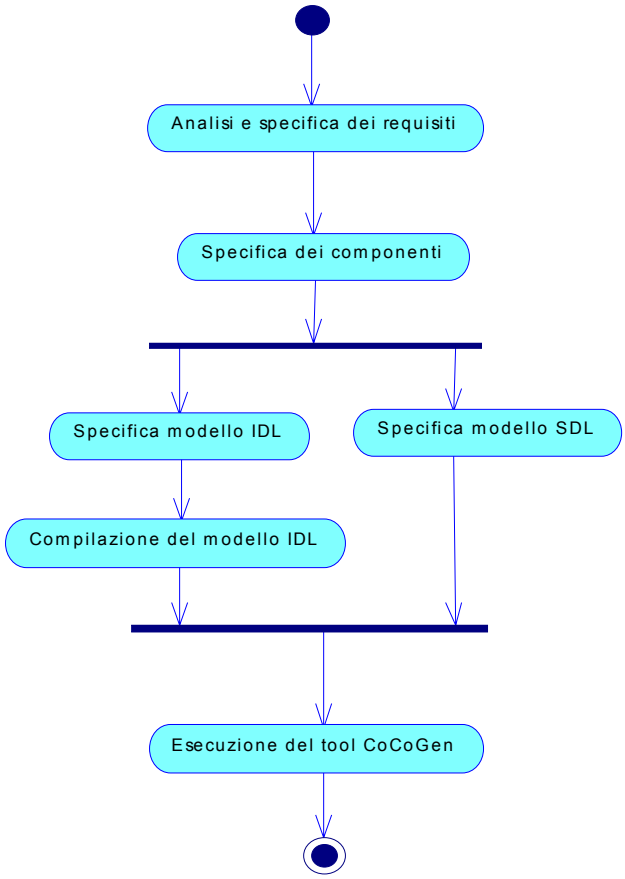


Figura 3.4 – *Attività del progettista*

In Figura 3.5 è mostrato il diagramma dei casi d'uso e di seguito sono riportate le loro descrizioni.

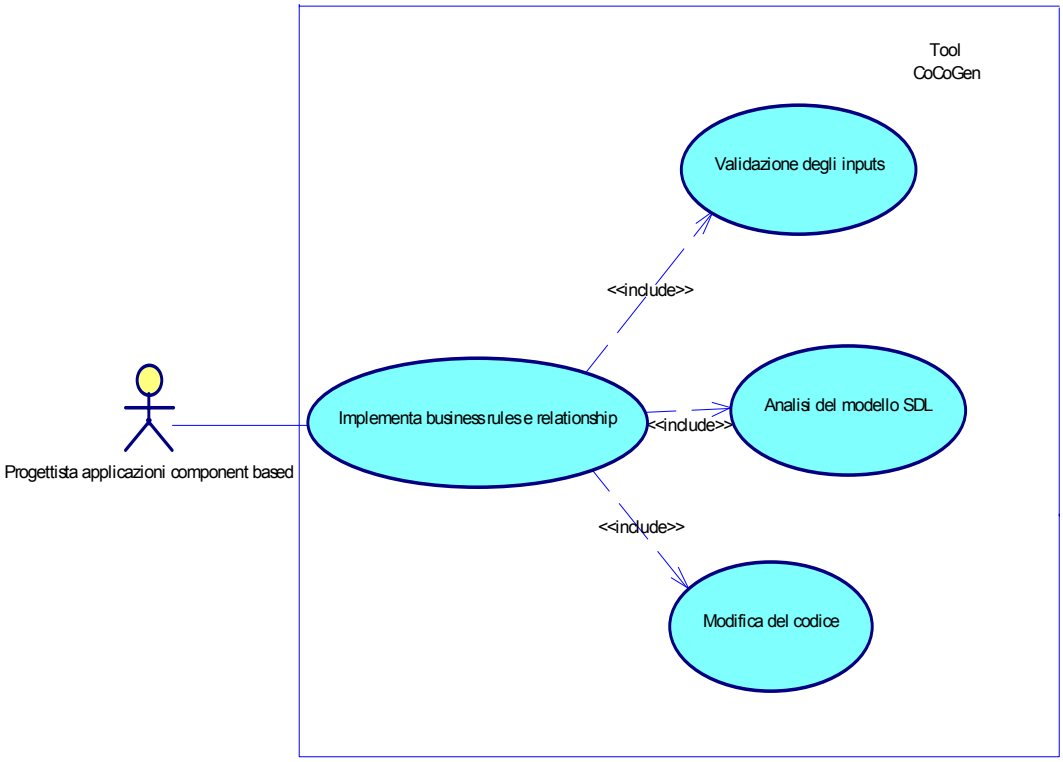


Figura 3.5 – *diagramma dei casi d'uso*

Nome **Implementa business rules e relationships**
Iniziatore **Progettista di applicazioni component based**
Obiettivo **Implementare business rules e relationship nel codice generato da un compilatore IDL**

Scenario di successo principale

1. il progettista inserisce i files contenenti il codice da modificare, il file contenente il modello SDL e un identificativo della piattaforma middleware
2. *include* **Validazione degli inputs**
3. *include* **Analisi del modello SDL**
4. *include* **Modifica il codice**
5. il tool comunica l'esito della implementazione

Nome **Validazione degli Inputs**
Iniziatore incluso nel caso d'uso **Implementa business rules e relationships**
Obiettivo **verificare l'esistenza dei files di input e la consistenza dell'identificativo di piattaforma immesso**

Scenario di successo principale

1. il tool valida gli inputs

Estensioni

1. gli inputs non sono validi
    a. fine caso d'uso

Nome **Analisi del modello SDL**
Iniziatore incluso nel caso d'uso **Implementa business rules e relationships**
Obiettivo **Individuare nel modello SDL business rules e relationships**

Scenario di successo principale

1. il tool esamina il file SDL e individua business rules e relationships

Estensioni

1. si verifica un errore durante l'analisi del file SDL
    a. fine caso d'uso

Nome **Modifica codice**
Iniziatore incluso nel caso d'uso **Implementa business rules e relationships**
Obiettivo **modificare il codice prodotto dai compilatori IDL per implementare le business rules e le relationship trovate con l'analisi del file SDL**

Scenario di successo principale

1. il tool implementa le business rules e le relationships nel codice generato dai compilatori IDL

Estensioni

1. si verifica un errore nella implementazione
    a. fine caso d'uso

3.3.2 Fase di progettazione

Per questo tool, è stato adottato un ciclo di vita del tipo a *sviluppo e consegna incrementale* in cui il processo di produzione software è partizionamento: ogni partizione definisce un insieme di ben definiti comportamenti del sistema e procede dalla definizione dei requisiti all'implementazione, etc. del software che riproduce tali comportamenti. I prodotti di ogni nuova partizione vengono aggiunti ed integrati con l'esistente e costituiscono un incremento del sistema[18].
In questa prima partizione ci si è occupati della implementazione dei constraints (invarianti, precondizioni, postcondizioni, guardie) su piattaforme Java _idl e TAO.
Per la progettazione di questa prima partizione del tool, sono stati esaminati tutti i casi d'uso e per ciascuno di essi è stata individuata una *collaboration* (insieme di classi, interfacce e altri elementi che lavorano insieme per fornire un comportamento cooperativo che è maggiore della somma di tutte le sue parti[2]) che lo realizza.
Per la realizzazione del caso d'uso **Implementa business rules e relationships** serve innanzitutto un elemento capace di ricevere gli input dall'utente (progettista di applicazioni component-based) e coordinare l'esecuzione degli altri casi d'uso **Validazione degli inputs**, **Analisi del modello SDL**, **Modifica del codice**. Tale elemento coordinatore può essere rappresentato dalla classe mostrata in Figura 3.6.

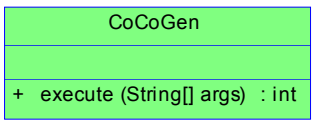


Figura 3.6 – *elemento coordinatore*

La classe riportata in Figura 3.6, è dotata di un unico metodo *execute* che ha come parametro di ingresso gli inputs del progettista ossia:

- i file da modificare

- il file contente il modello SDL
- un numero che identifica la piattaforma di sviluppo (1→ Java_IDL; 2→ Tao)

il parametro di output è un intero che rappresenta l'esito della implementazione dei constraints.

Per la realizzazione del caso d'uso **Validazione degli inputs** può essere sufficiente una classe come quella mostrata in Figura 3.7.

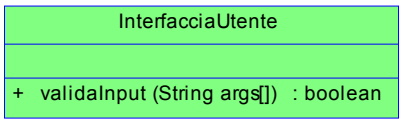


Figura 3.7 – *Classe InterfacciaUtente che realizza il caso d'uso* **Validazione degli inputs**

La classe *InterfacciaUtente* rappresenta l'entità preposta alla validazione degli inputs immessi dall'utente, per tale motivo è fornita di una apposita funzione la cui signature è:

boolean validaInput(String args[])

tale funzione ha come parametro di ingresso un vettore di stringhe rappresentante gli inputs dell'utente del tool, essa restituisce un valore booleano per comunicare l'esito della validazione (positivo = true, negativo=false).

Tale classe è in relazione con la classe *CoCoGen* che come detto ha la funzione di coordinare tutto il processo di implementazione di business rules e relationships; tale relazione è mostrata in Figura 3.8.

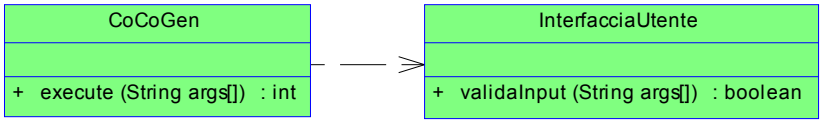


Figura 3.8 – *Dipendenza di CoCoGen da InterfacciaUtente*

Le necessarie interazioni fra gli oggetti di queste due classi, per raggiungere l'obiettivo del caso d'uso in esame, sono mostrate in Figura 3.9.

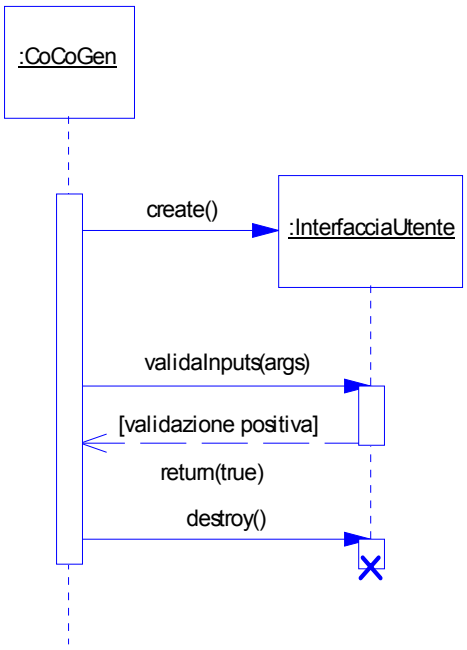


Figura 3.9 – *interazione* CoCoGen – *InterfacciaUtente*

Il caso d'uso **Analisi del modello SDL** ha come obiettivo l'analisi del file contente il modello SDL per identificare tutti i constraints contenutivi, la struttura atta a memorizzare questi constraints sarà utilizzata successivamente dalle entità preposte alla implementazione dei constraints nel codice.

Per la realizzazione di questo caso d'uso è necessaria la collaborazione mostrata in Figura 3.10.

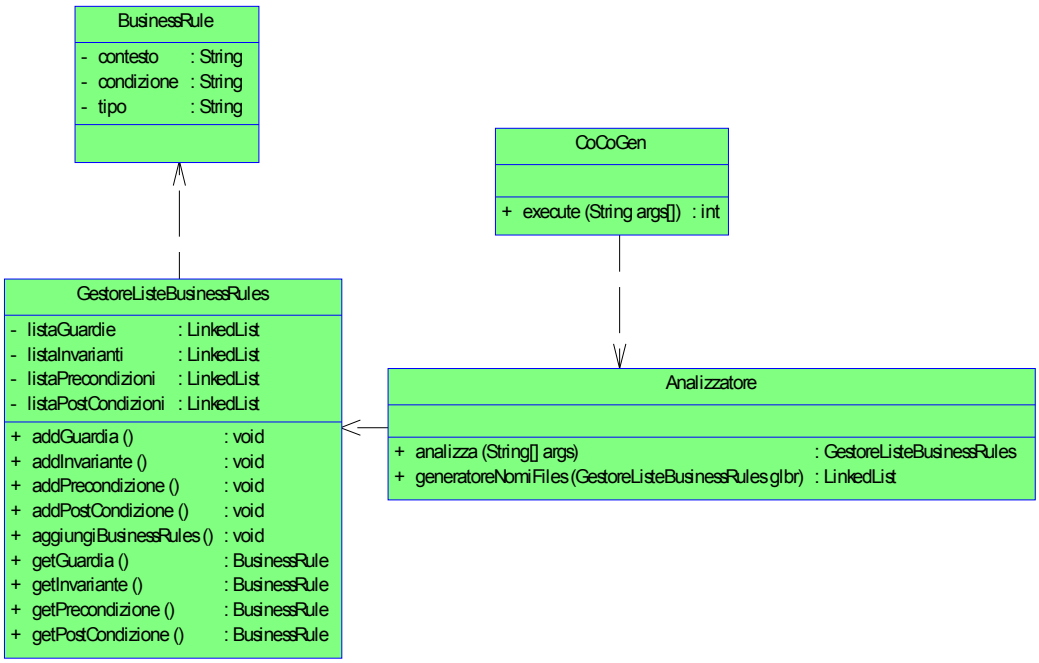


Figura 3.10 – *collaboration per il caso d'uso* **Analisi del modello SDL**

Classe ***BusinessRule***- tale classe rappresenta un generico constraint il quale, come visto in OCL e in SDL, è caratterizzato dal *contesto* a cui si riferisce, dalla *condizione* che costituisce la business rule vera e propria e da un *tipo* che ne permette una classificazione (invariante, precondizione, postcondizione, guardia).

Classe ***GestoreListeBusinessRules*** – rappresenta la struttura atta a memorizzare tutti constraints individuati dall'oggetto della classe ***Analizzatore***.

Classe ***Analizzatore*** – rappresenta l'entità preposta all'analisi del file SDL per individuare i constraints contenutivi; tale entità deve memorizzare i constraints individuati in un oggetto della classe ***GestoreListaBusinessRules***.

Per l'analisi del file SDL, tale classe è fornita di un metodo *analizza* la cui signature è:

GestoreListeBusinessRules analizza*(String args[])*

il parametro di ingresso è un vettore di stringhe che rappresentano gli inputs dell'utente del tool, fra cui il nome del file SDL. Il parametro di uscita è un oggetto di tipo ***GestoreListaBusinessRules*** che contiene i constraints individuati.

Le interazioni previste fra gli oggetti di queste classi, per realizzare il caso d'uso in esame, sono mostrate in Figura 3.11.

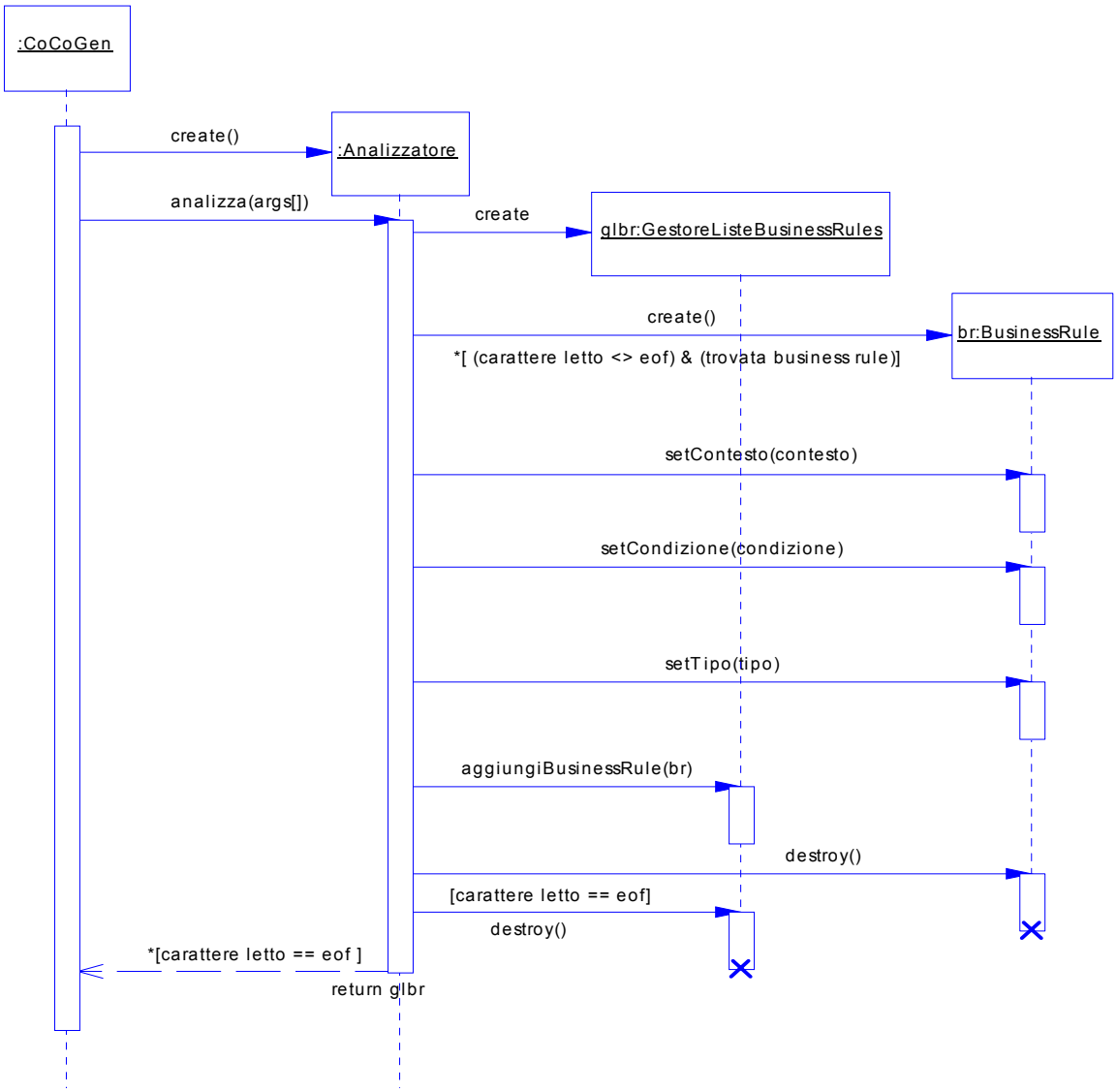


Figura 3.11- *sequence diagram per il caso d'uso* **Analisi del modello SDL**

Il caso d'uso **Modifica codice** ha come obiettivo l'implementazione dei constraints individuati nel modello SDL. Come detto in precedenza, i constraints contenuti nel file SDL vengono individuati dall' ***Analizzatore*** e memorizzati nella struttura ***GestoreListeBusinessRules;*** per l'implementazione, questi constraints devono essere letti da uno ***Dispatcher*** che ne comanda l'implementazione a un ***Implementatore*** .

In Figura 3.12 sono mostrate le strutture della *collaboration* che realizza il caso d'uso in esame.

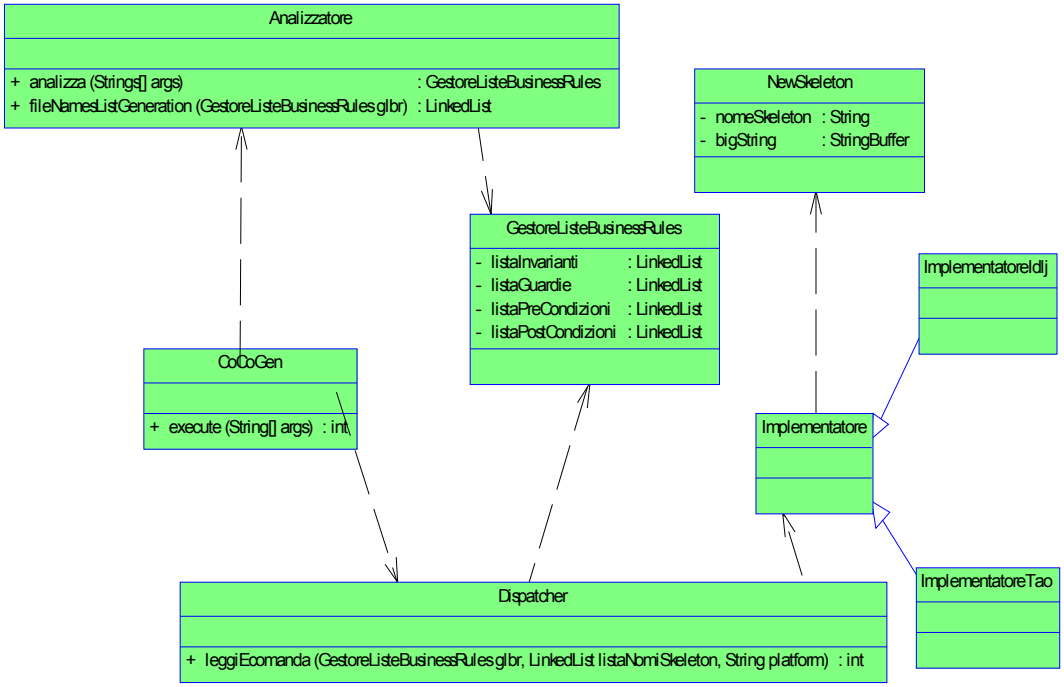


Figura 3.12 – *collaboration per il caso d'uso* **Modifica Codice**

Classe ***Dispatcher*** – questa classe rappresenta lo shedulatore che ha il compito di leggere i constraints memeorizzati nel ***GestoreListeBusinessRules*** e di comandare all'***Implementatore*** l'implementazione di tali constraints.

Classe ***NewSkeleton*** – i file che devono essere modificati sono sequenziali quindi non si può accere a una specifica posizione senza non avere acceduto a tutte le posizioni precedenti a partire dall'inizio. Ciò rende difficoltose le operazioni di modifica sul file. Per risolvere questo problema è possibile trasformare il file in una grande stringa, modificare quest'ultima per implementare i constraints e ,infine, riscrivere il file sostituendo il suo contenuto con la stringa che lo rappresenta.

La classe NewSkeleton (Figura 3.13) rappresenta tale stringa ed ha come variabili membro, oltre che stringa stessa (*bigString*), anche il nome del file da cui è ottenuta (*nomeSkeleton*). I suoi metodi servono per la creazione della stringa a partire dal file (*creaBigString*) e per la scrittura della stringa sul file (*scriviSuFile*).

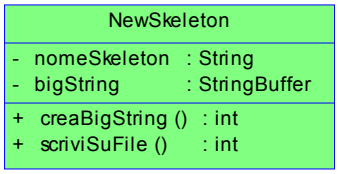


Figura 3.13 – *classe NewSkeleton*

Classe ***Implementatore*** – tale classe rappresenta l'entità preposta alla implemetazione dei constraints nel codice prodotto dai compilatori. Tale classe si specializza in due sottoclassi ***ImplementatoreIdlj*** e ***ImplementatoreTao*** adibite rispettivamente alla implementazione su Java_Idl e TAO. I compilatori IDL possono generare diverse quantità di codice a seconda dell'utilizzo o meno di determinate opzioni di compilazione: alcuni compilatori generano solo gli *skeleton* altri generano anche le classi relative *Servant* (che è la classe che effettivamente implementa il componente). Di conseguenza, esistono due possibilità per la implementazione dei constraints: 1) modificare gli *skeleton*; 2) modificare le classi relative al *Servant*.

Per la piattaforma Java_Idl è stata scelta l'implementazione mediante modifica degli *skeleton*, mentre per la piattaforma TAO è stata scelta l'implementazione

mediante modifica delle classi relative al *Servant*. I patterns usati per l'implementazione sono simili a quelli mostrati nel capitolo 2.

I dettagli delle classi ImplementatoreIdlj e ImplementatoreTao sono mostrati rispettivamente in Figura 3.14 e Figura 3.15. In esse sono presenti delle utility che servono per la modifica della condizione della business rules e sono usati dai metodi di implementazione. In Figura 3.16 è mostrato l'algoritmo usato da tutte le operazioni di implementazione, tale algoritmo vale per entrambe le piattaforme.

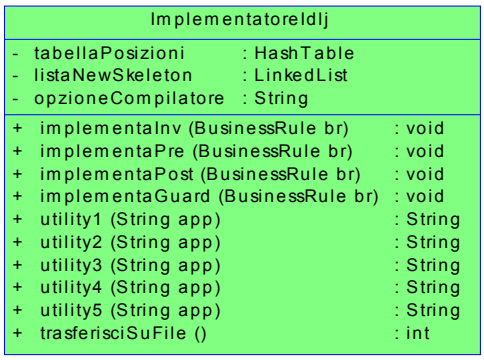


Figura 3.14 *Classe ImplementatoreIdlj*

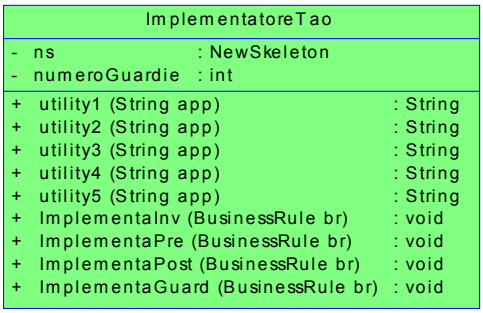


Figura 3.15 *Classe ImplementatoreTao*

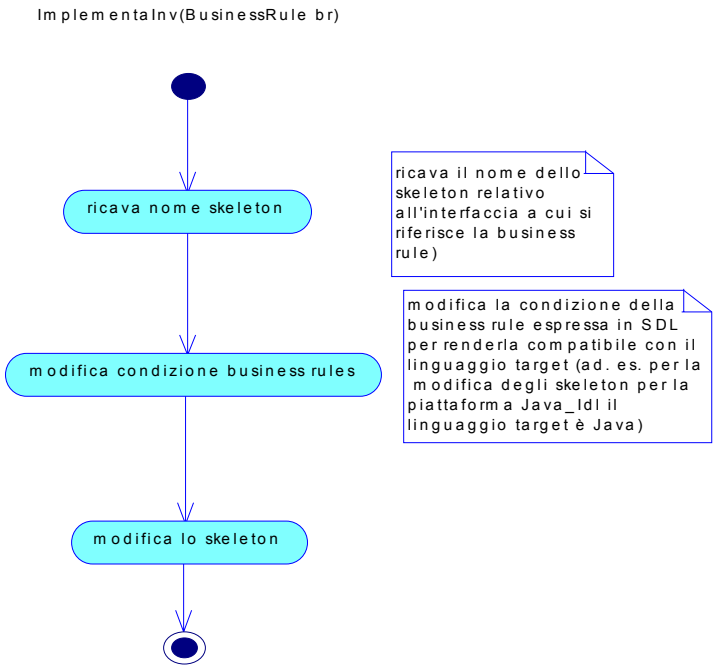


Figura 3.16 – *Algoritmo di implementazione*

In Figura 3.17 sono mostrate le interazioni fra le strutture della collaboration di Figura 12 necessarie per realizzare il caso d'uso.

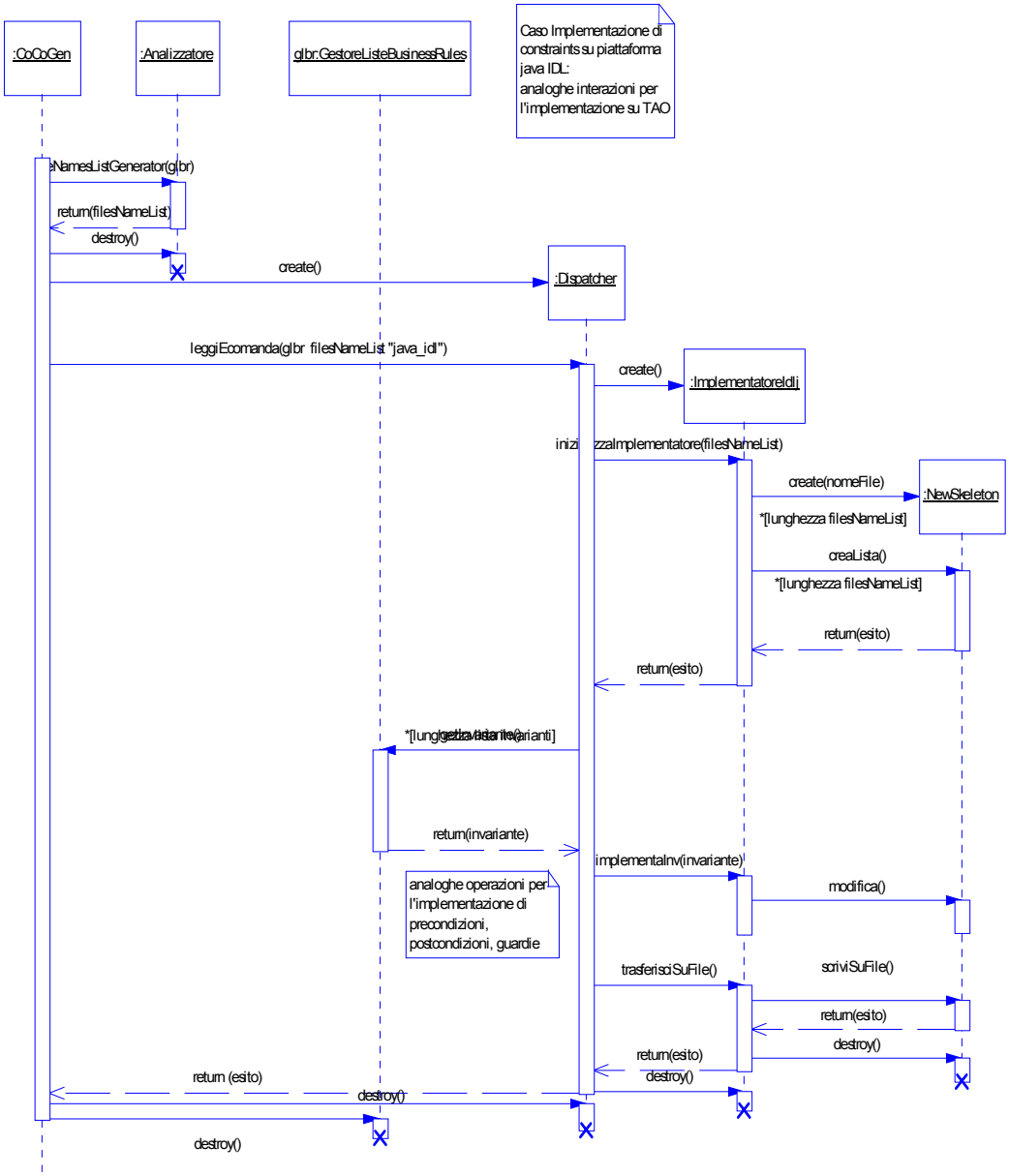


Figura 3.17 – *Interazioni fra le strutture della collaboration*

3.3.3 Fase d'implementazione

Per l'implementazione del tool CoCoGen è stato usato il linguaggio Java [19][20][21] nella versione J2SDK1.4.1_02.Il codice relativo all'implementazione dei tools non è riportato per motivi di brevità ma è comunque presente nel cd-rom allegato a questo studio.

3.4 Sviluppo del tool CoCoMod

L'idea di sviluppare un tool per la modellazione delle interfacce dei componenti nasce dall'esigenza di automatizzare alcune operazioni che sono a monte dell'utilizzo del tool CoCoGen. Come si vede dalla Figura 3.18, prima di potere eseguire il tool CoCoGen il progettista deve specificare i modelli SDL e IDL, e compilare il modello IDL. Ciò, oltre a comportare un aumento dei tempi di sviluppo, comporta un aumento della possibilità di compiere errori.

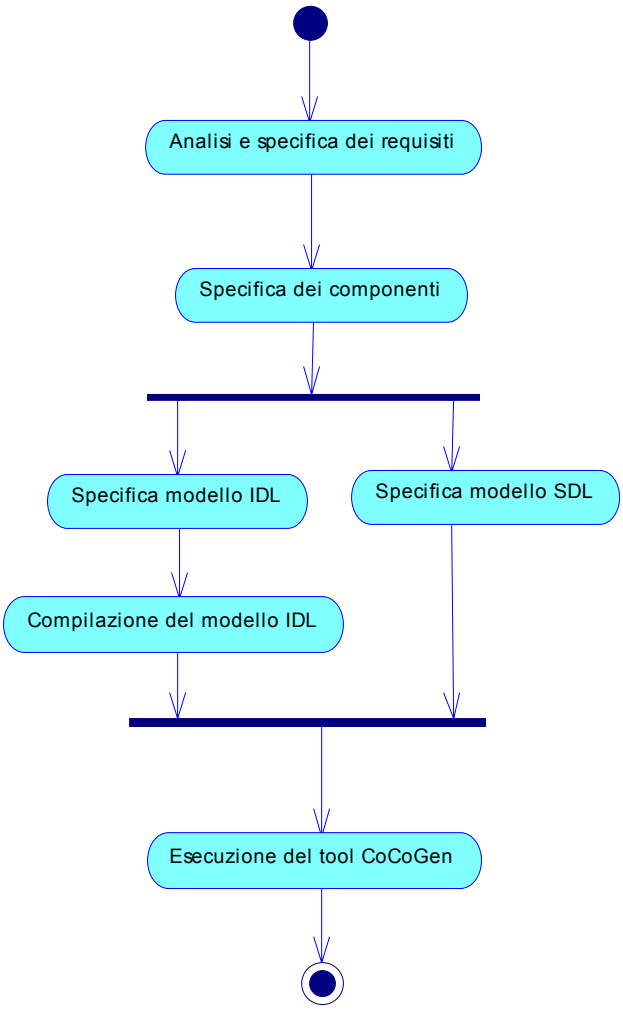


Figura 3.18 – *Attività del progettista*

Quindi, la disponibilità di un tool che fornisce un ambiente grafico per:

- specificare interfacce e constraints
- generare automaticamente i modelli SDL e IDL
- compilare il modello IDL
- eseguire CoCoGen

facilita ulteriormente lo sviluppo di applicazioni distribuite.

3.4.1 Fase di analisi

Il tool CoCoMod deve offrire una "area di progetto" in cui il progettista può disegnare le interfacce dei componenti e quindi definire tutte quelle proprietà come costanti, eccezioni, tipi di dato, ereditarietà, attributi, operazioni etc. Il progettista deve anche potere definire un modulo a cui le interfacce appartengono, analogo al costrutto *module* offerto da IDL. La novità rispetto a quello che può essere fatto anche con IDL è che, relativamente a una interfaccia, è possibile definire dei constraints e in particolare:

- invarianti e guardie sugli attributi
- precondizioni e postcondizioni sulle sue operazioni

Il tool deve essere anche in grado di generare , a partire da ciò che stato definito nell'area di progetto (moduli, interfacce, etc), i modelli IDL e SDL.
Infine, deve essere offerta al progettista la possibilità di compilare il modello IDL e di eseguire il tool CoCoGen per implementare i constraints eventualmente specificati.

Dizionario
Elemento= figura rappresentante una interfaccia o un modulo (analogo del costrutto *module* di IDL)
Area di progetto=parte dello schermo in cui il progettista può inserire elementi
Progetto= insieme di moduli e interfacce (e delle loro proprietà) specificati dal progettista. Le proprietà di moduli e interfacce sono quelle definite dallo standard IDL (costanti,eccezioni, typedef, attributi, operazioni, etc.)

Il diagramma dei casi d'uso del tool CoCoMod è mostrato in Figura 3.19.

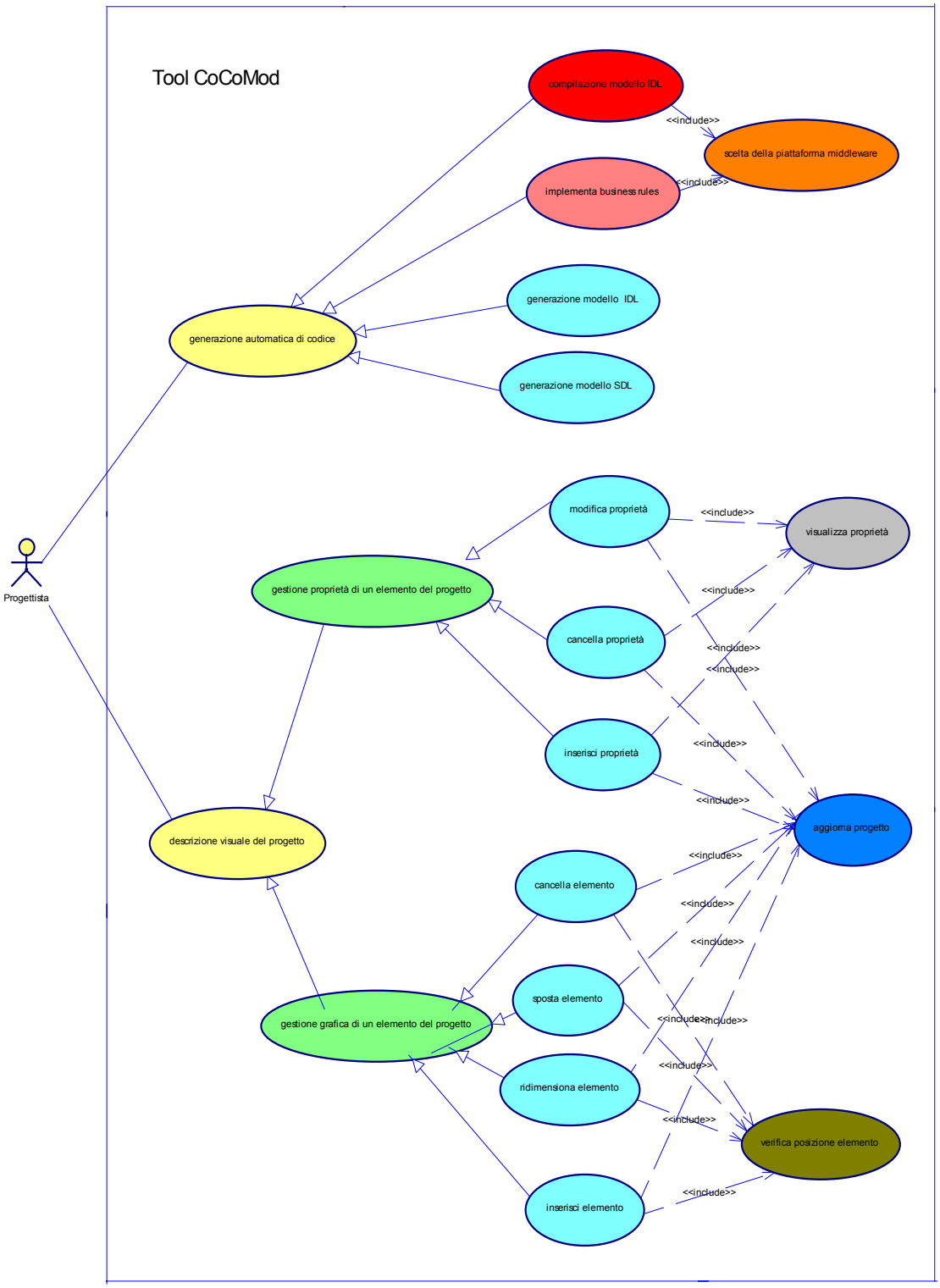


Figura 3.19 – *Diagramma dei casi d'uso del tool CoCoMod*

I casi d'uso relativi agli elementi di progetto (inserisci elemento, ridimensiona elemento, sposta elemento, cancella elemento) possono essere ulteriormente specificati per distinguere i casi in cui l'elemento è una interfaccia o un modulo.
Di seguito sono riportate le descrizioni dei casi d'uso.

Nome **inserisci elemento**
Iniziatore **progettista**
Obiettivo **inserire un elemento nel progetto**

Scenario principale di successo

1. il progettista sceglie l'elemento da inserire nell'area di progetto
2. il progettista sceglie il punto in cui posizionare l'elemento
3. *include* **verifica posizione elemento**
4. *include* **aggiorna progetto**

il caso d'uso **cancella elemento** è analogo al precedente ma non prevede il passo 3.

Nome **ridimensiona elemento**
Iniziatore **progettista**
Obiettivo **ridimensionare un elemento**

Scenario principale di successo

1. il progettista seleziona l'elemento da ridimensionare
2. il progettista modifica le dimensioni dell'elemento
3. *include* **verifica posizione elemento**
4. *include* **aggiorna progetto**

il caso d'uso **sposta elemento** è analogo al precedente.

Nome **verifica posizione elemento**
Iniziatore incluso nei casi d'uso **inserisci elemento**, **ridimensiona elemento**, **sposta elemento**
Obiettivo verificare che non ci siano intersezioni fra le figure nell'area di progetto

Scenario principale di successo

1. il tool verifica la assenza di intersezioni

Estensioni

1. il tool verifica la presenza di intersezioni
    a. fine caso d'uso

Nome **aggiorna progetto**
Iniziatore **incluso in tutti i casi d'uso che specializzano il caso d'uso descrizione visuale del progetto**
Obiettivo **ridisegnare l'area di progetto**

Scenario principale di successo

1. ridisegna l'area di progetto

Nome **modifica proprietà elemento**
Iniziatore **progettista**
Obiettivo **modificare una proprietà dell'elemento**

Scenario principale di successo

1. il progettista seleziona l'elemento
2. *include* **visualizza proprietà**
3. il progettista sceglie la proprietà da modificare
4. il progettista modifica la proprietà
5. *include* **aggiorna progetto**

i casi d'uso **cancella proprietà** e **inserisci proprietà** sono analoghi al precedente.

Nome **visualizza proprietà**
Iniziatore **incluso nei casi d'uso che specializzano il caso d'uso gestione proprietà elemento del progetto**
Obiettivo **mostrare al progettista le proprietà di un elemento (costanti, eccezioni, operazioni, attributi, constraints)**

Scenario principale di successo

1. il tool mostra le proprietà dell'elemento

Nome **generazione del modello IDL**
Iniziatore **progettista**
Obiettivo **generare il modello IDL delle interfacce e dei moduli disegnati nell'area di progetto**

Scenario principale

1. il progettista decide di generare il modello IDL
2. il tool chiede dove salvare il file contente il modello IDL
3. il progettista sceglie dove salvare il file
4. il tool genera il modello e lo salva nel file selezionato

Estensioni

4. si verifica un errore
    a. il tool segnala che è accaduto un errore
    b. fine caso d'uso

il caso d'uso **generazione del modello SDL** è analogo al precedente.

Nome **compilazione modello IDL**
Iniziatore **progettista**
Obiettivo **compilare il modello IDL del progetto**

Scenario principale di successo

1. il progettista vuole compilare il modello IDL
2. *include* **scelta della piattaforma**
3. il tool manda in esecuzione il compilatore per la piattaforma scelta

Estensioni

4. si verifica un errore di compilazione
    a. il tool comunica che è avvenuto un errore
    b. fine caso d'uso

Nome **implementa business rules**
Iniziatore **progettista**
Obiettivo **eseguire il tool CoCoGen per implementare i constraints specificati**

Scenario principale di successo

1. il progettista vuole eseguire il tool CoCoGen
2. il tool esegue CoCoGen

Estensioni

2. si verifica un errore nella esecuzione di CoCoGen
    a. il tool comunica che è avvenuto un errore
    b. fine caso d'uso

Nome **scelta della piattaforme middleware**
Iniziatore incluso nei casi d'uso **compilazione modello IDL** e **implementazione business rules**
Obiettivo **permettere al progettista di scegliere la piattaforma su cui devono essere implementati i constraints**

Scenario principale di successo

1. il tool mostra le piattaforme middleware supportate
2. il progettista sceglie la piattaforma

3.4.2 Fase di progetto

Per la realizzazione del caso d'uso **descrizione visuale del progetto** è necessario innanzitutto modellare la struttura di un progetto (Figura 3.20), termine con cui viene indicato l'insieme dei moduli, delle interfacce e delle loro proprietà.
Come si vede un progetto si compone di un certo numero di elementi (Moduli, Interfacce e Relationships) e di un certo numero proprietà (Direttive, Costanti, Typedef, Eccezioni).
Ogni elemento a sua volta si compone di proprietà (Attributi,Operazioni,Guardie, Invarianti, Eccezioni, Costanti, Typedef).
Di seguito è riportata una breve descrizione di alcune classi modellate in Figura 3.20

classe ***DesignElement*** – tale classe rappresenta il generico elemento del progetto, fra i suoi attributi ce ne sono due per tenere conto dell'aspetto visuale del progetto e sono:

- dimension che è la dimensione della figura rappresentante l'elemento
- position che è la posizione nella area di progetto della figura rappresentante l'elemento

classe ***Directive*** – tale classe rappresenta il meccanismo fornito da IDL per l'inclusione testuale di un file.

Classe ***Exception*** – tale classe rappresenta una generica eccezione sollevabile da una operazione e può avere visibilità a livello dell'intero progetto, del singolo modulo o della singola interfaccia a seconda di dove viene definita.

Classi ***PostCondition*** e ***PreCondition*** – tali classi rappresentano rispettivamente una postcondizione e una precondizione associata a una deteminata operazione (rappresentata dalla classe ***Operation***)

Classe ***Invariant*** – tale classe rappresenta un invariante su un attributo (rappresentato dalla classe ***Attributo***)

Classe ***Guard*** – tale classe rappresenta una guardia definita su una interfaccia.

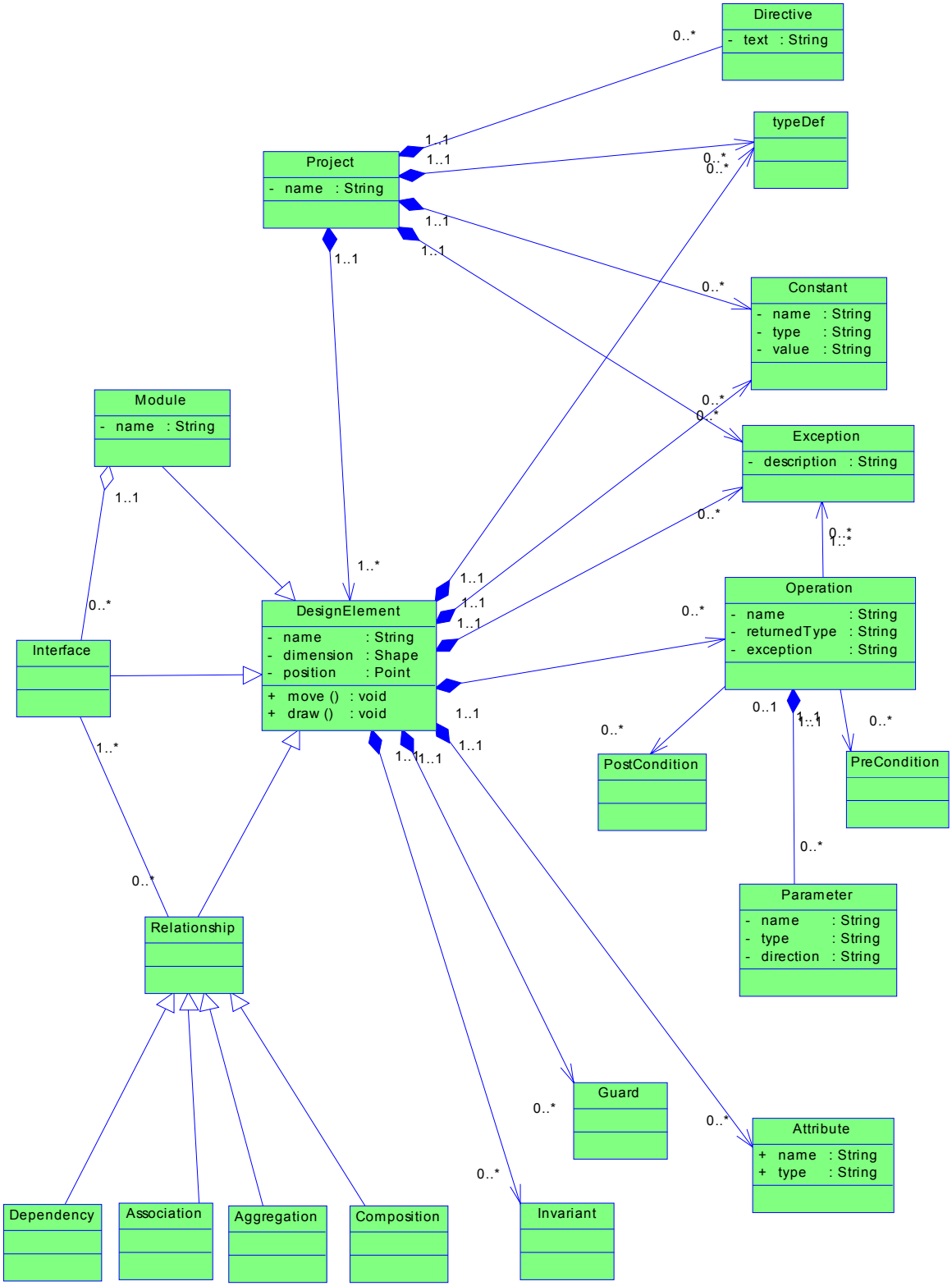


Figura 3.20 – *Struttura di un "progetto"*

Prima di pensare alla realizzazione dei casi d'uso è necessario innanzitutto pensare a una entità che dia al progettista accesso ai servizi di progettazione visuale (indicati nei casi d'uso che specializzano **gestione grafica elemento di progetto**) e di generazione del codice (indicati nei casi d'uso che specializzano **generazione automatica codice**) offerti dal tool CoCoMod.
Per tale motivo è stata introdotta la classe ***InterfacciaUtente***(Figura 3.21).

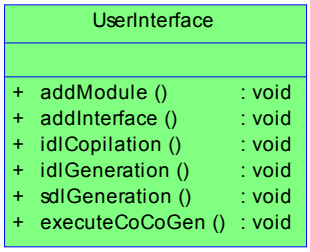


Figura 3.21 – *classe UserInterface*

Per la realizzazione del caso d'uso **gestione grafica elemento di progetto** e di quelli che lo specializzano (**cancella elemento**, **sposta elemento**, **ridimensiona elemento**, **inserisci elemento**) si è pensato all'introduzione di una entità adibita alla gestione della grafica del progetto, tale entità è rappresentata dalla classe ***GraphicEditorManager*** rappresentata in Figura 3.22.
Inoltre, per la realizzazione del caso d'uso **verifica posizione elemento** si pensato di introdurre nella classe Project i seguenti metodi:

- *verifyModuleArea(Area area)* per verificare se la figura rappresentante un modulo interseca altre figure
- *verifyInterfaceArea(Area area)* per verificare se la figura rapresentante una interfaccia interseca altre figure
- *verifyPointInModule(Point point)* per verificare se un punto dell'area di progetto selezionato dal progettista è in incluso nella figura rappresentante un modulo

- *verifyPointInInterface (Point point)* per verificare se un punto dell'area di progetto selezionato dal progettista è incluso nella figura rappresentante una interfaccia
- *findInclusion(Area area)* per verificare se la figura rappresentante una interfaccia è inclusa nella figura rappresentante un modulo

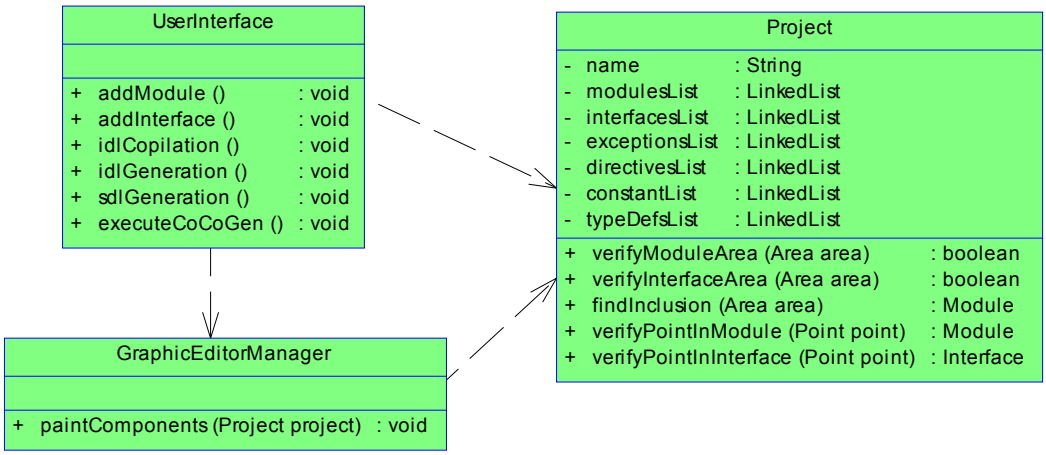


Figura 3.22 – *relazioni fra le classi Project, GraphicEditorManager, UserInterface*

In Figura 3.23 sono mostrate le interazioni fra le classi Project, GraphicEditorManager e UserInterface per realizzare il caso d'uso ***inserisci elemento*** in cui elemento è inteso come interfaccia

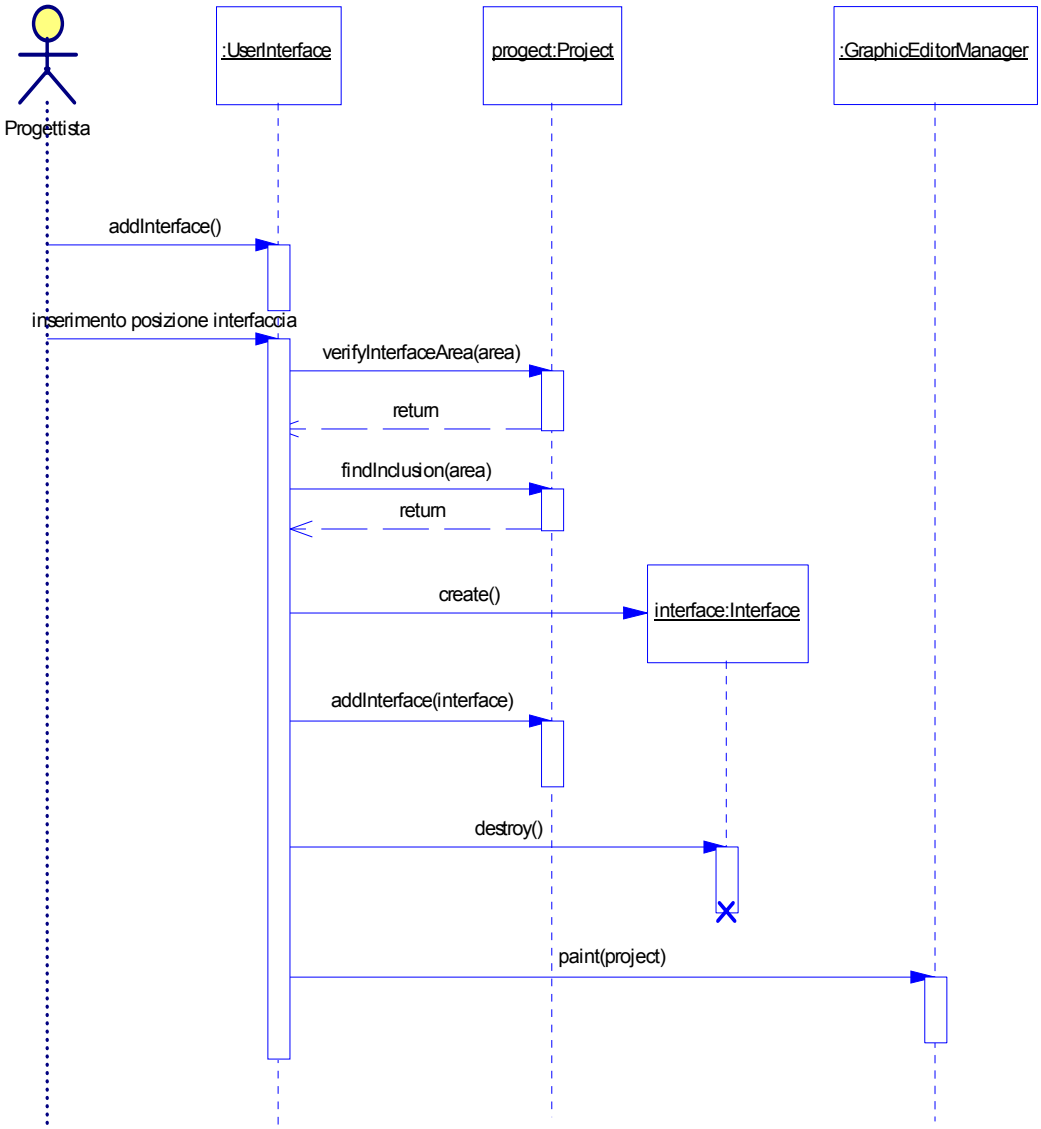


Figura 3.23 – *Interazioni per l'aggiunta di una interfaccia*

Interazioni del tutto simili sono necessarie per le altre operazioni riguardanti l'aspetto visuale del progetto.

Per la realizzazione dei casi d'uso che specializzano **generazione automatica codice** è stata introdotta una entità il cui compito è fornire i metodi per la

generazione dei modelli IDL e SDL, la compilazione del modello IDL e l'esecuzione del tool CoCoGen. Tale entità è rappresentata con la classe ***CodeGenerator*** mostrata in Figura 3.24.

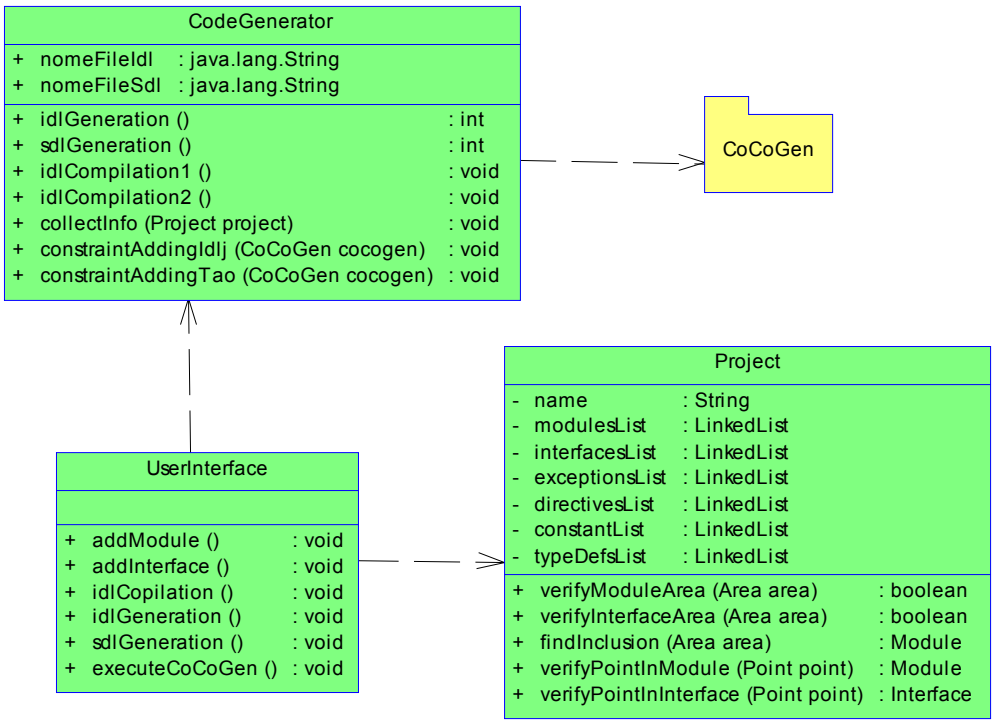


Figura 3.24 – *Relazioni fra le classi UserInterface, Project, CodeGenerator*

In Figura 3.25 sono riportate le interazioni necessarie per realizzare il caso d'uso **generazione modello IDL.** Per gli altri casi d'uso che specializzano **generazione automatica codice** sono previste iterazioni del tutto analoghe.

La parte di progetto relativa alla realizzazione dei casi d'uso che specializzano ***gestione proprietà elemento*** non è qui riportata perchè poco significativa. Infatti lo strumento di sviluppo utilizzato (Borland JBuilder 8) fornisce strumenti visuali per l'utilizzo di form, tabelle, finestre etc. e meccanismi che automatizzano completamente la loro gestione.

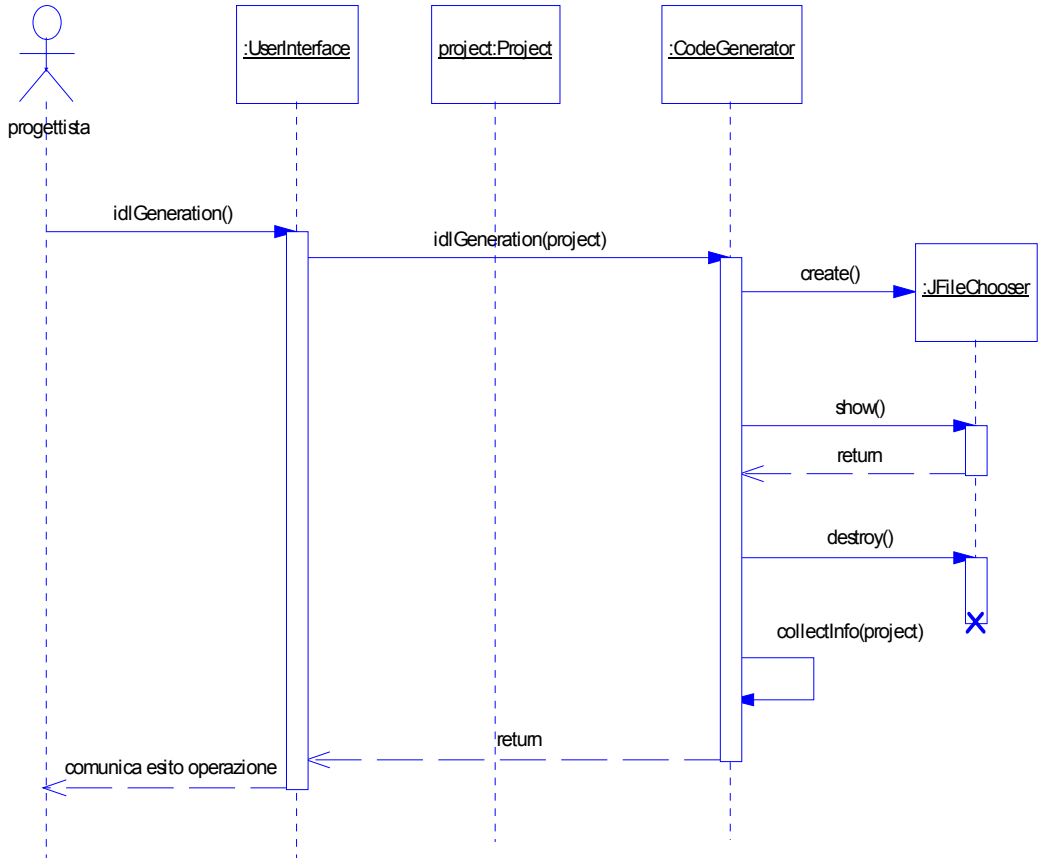


Figura 3.25 – *Interazioni per la realizzazione di* **generazione modello IDL**

Per gli altri casi d'uso che specializzano **generazione automatica codice** sono previste iterazioni del tutto analoghe.

La parte di progetto relativa alla realizzazione dei casi d'uso che specializzano ***gestione proprietà elemento*** non è qui riportata perchè poco significativa. Infatti lo strumento di sviluppo utilizzato (Borland JBuilder 8) fornisce strumenti visuali per l'utilizzo di form, tabelle, finestre etc. e meccanismi che automatizzano completamente la loro gestione.

3.4.3 Fase d'implementazione

Per l'implementazione del tool CoCoGen è stato usato il linguaggio Java [19][20][21] nella versione J2SDK1.4.1_02.
Per l'implementazione del tool CoCoMod è stato usato l'ambiente di sviluppo visuale JBuilder 8
Il codice relativo all'implementazione dei tools non è riportato per motivi di brevità ma è comunque presente nel cd-rom allegato a questo studio.

CAPITOLO IV
METODOLOGIA PROPOSTA PER LO SVILUPPO DI APPLICAZIONI A COMPONENTI

4.1 Introduzione

In questo capitolo sono mostrate le modifiche da apportare al ciclo di sviluppo di applicazioni distribuite per implementare business rules e relationships. Le fasi che costituiscono tale ciclo di sviluppo, non subiscono cambiamenti sostanziali perché le modifiche da apportare non sono di tipo metodologico ma riguardano solo gli strumenti usati in queste fasi per la produzione di deliverables.

4.2 Il processo di sviluppo di applicazioni component-based

Il processo completo di sviluppo di applicazioni a componenti usato come riferimento è mostrato in Figura 4.1[15]. I rettangoli rappresentano *workflow* vale a dire sequenze di attività che producono un risultato di valore osservabile. Le linee sottili rappresentano il flusso di *deliverables* che trasporta informazioni tra i *workflow*.

Il workflow di ***analisi dei requisiti*** riceve in input i requisiti del dominio applicativo e produce una rappresentazione di questoultimo (*business concept model*), e il diagramma dei casi d'uso.

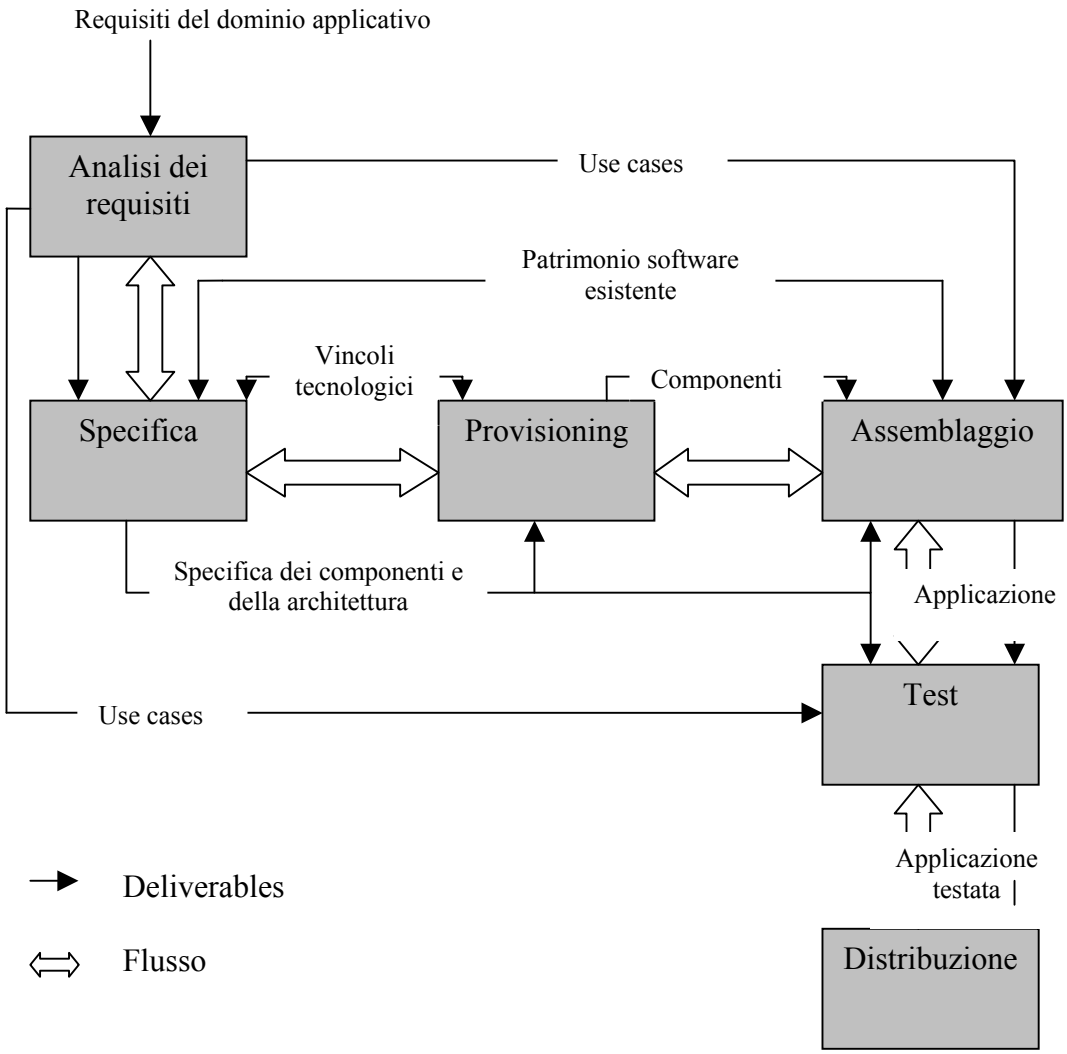


Figura 4.1 – *Il processo di sviluppo*

Il workflow di ***specifica*** riceve in input dal workflow di analisi dei requisiti i casi d'uso e un *business concept model*, inoltre sono usate informazioni circa il patrimonio software esistente, come sistemi legacy, librerie, databases e su vincoli tecnologici come l'uso di particolari architetture o strumenti. I deliverables prodotti sono: la *specifica dei componenti* e *l'architettura dei componenti*.

La specifica dei componenti include la specifica delle interfacce che essi supportano, mentre l'architettura dei componenti mostra come i componenti interagiscono fra loro.

Questi deliverables sono usati: nel workflow di ***provisionig***, per determinare quali componenti devono essere creati e quali comprati; nel workflow di ***assemblaggio***, per indicare la corretta integrazione dei componenti, e nel workflow di ***test***, come input per la creazione dei casi di test.

Il workflow di ***provisioning*** assicura che i componenti necessari siano resi disponibili o mediante acquisto, riuso, modifica di componenti esistenti, o mediante creazione ex-novo. In questo workflow è incluso anche il testing dei singoli component per assicurarsi della loro integrabilità.

Il workflow di ***assemblaggio*** mette insieme i componenti fra loro e li integra nel patrimonio software.

I principali deliverables del workflow di ***analisi dei requisiti*** sono il *business concept model* e i *casi d'uso*. Il *business concept model* è un modello concettuale del dominio applicativo che necessita di essere compreso e accettato. Il suo scopo principale è di creare un vocabolario comune per le persone coinvolte nel progetto.

Un *caso d'uso* è un modo per specificare certi aspetti dei requisiti funzionali del sistema. Esso descrive le interazioni fra un attore (che può essere un utente, un dispositivo o un altro sistema software) e il sistema, quindi aiuta nella definizione del contesto di utilizzo del sistema all'interno del dominio applicativo.

I principali deliverables del workflow di ***specifica*** sono il *business type model*, la *specifica delle interfacce*, la *specifica dei componenti* e la *architettura dei componenti*.

Il *business type model* è un deliverable di specifica intermedio che non costituisce un input per il workflow di ***provisioning***. Il suo scopo è di formalizzare il *business concept model* per definire la conoscenza che ha il sistema software del dominio applicativo che supporta. Questo modello è la base per l'identificazione iniziale delle interfacce.

La *specifica delle interfacce* è un deliverable contenente un insieme di specifiche d'interfacce. Ogni interfaccia è un contratto con un cliente di un componente. Ogni specifica definisce i dettagli di tale contratto ossia quali sono le operazioni fornite, quali sono le loro signature e che effetto ha la l'esecuzione di un'operazione sui parametri della sua signature e sullo stato del componente.

La *specifica dei componenti* è un deliverable contenente un insieme di specifiche di componenti. La specifica di un componente definisce le interfacce che esso supporta, come le specifica di queste interfacce si relazionano fra loro, e quali interfacce sono usate dal componente. Come già anticipato nel Capitolo 1 parlando di "*Design by Contract*", mentre la specifica di un'interfaccia rappresenta un *contratto di utilizzo* che lega il cliente e il componente, la specifica di un componente mette insieme tutti i contratti di utilizzo relativi alle interfacce che supporta per definire un singolo *contratto di realizzazione* che lega il componente al suo sviluppatore.

Infine, la *architettura dei componenti* è il deliverable contenente la descrizione di come le specifiche dei componenti si integrano in una data configurazione, in altre parole l'architettura mostra quali devono essere le relazioni (strutturali e funzionali) fra i componenti per "assemblare" un sistema che soddisfa i requisiti.

4.3 Il workflow di specifica

Il workflow di ***specifica*** è mostrato in Figura 4.2.

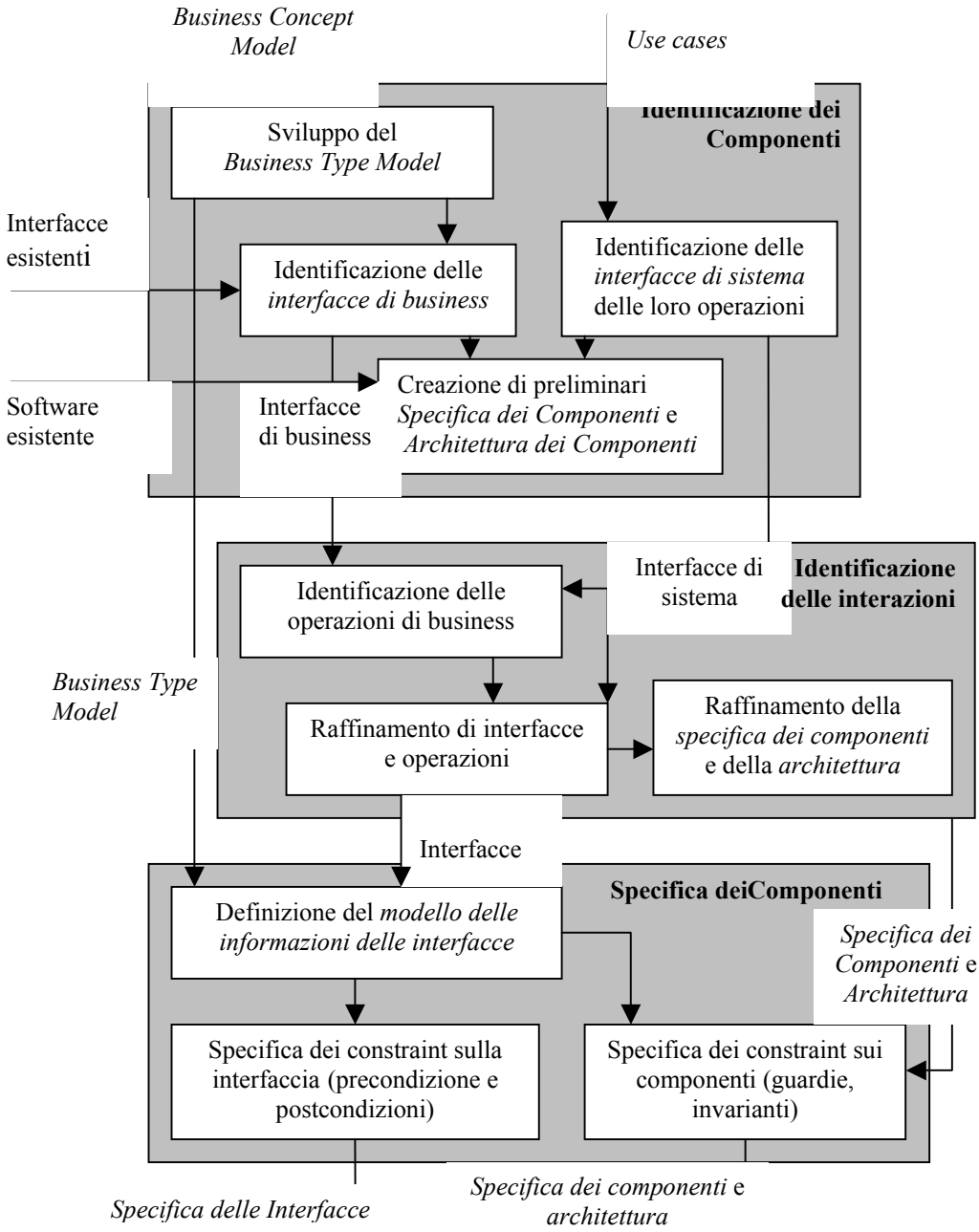


Figura 4.2 – *Le tre fasi del workflow di specifica*

4.3.1 Identificazione dei componenti

La fase di identificazione dei componenti riceve in input dal workflow di ***analisi dei requisiti*** il *business concept model* e i *casi d'uso*. Questa fase assume che l'applicazione ha un'architettura a livelli del tipo mostrato in Figura 4.3

Figura 4.3 – *Architettura della applicazione*

I livelli della applicazione sono i seguenti:

- Interfaccia utente – offre le funzionalità per la presentazione delle informazioni all'utente e per la acquisizione dei suoi input.
- Dialogo con l'utente –offre le funzionalità per la gestione del dialogo con l'utente in una sessione.
- Servizi di sistema – questo livello fornisce l'accesso ai servizi del sistema. Questo livello agisce come interfaccia per i livelli superiori fornendo un contesto in cui i servizi di business più generici sono usati per implementare i servizi dello specifico sistema.

- Servizi di business – questo livello è il core della applicazione e si occupa della gestione delle informazioni, delle regole e dei processi riguardanti un determinato dominio applicativo.

L'obiettivo di questa fase è di identificare, a partire dai deliverables di input, un insieme iniziale di interfacce di sistema e di interfacce di business, che dovranno essere offerte dai componenti del sistema, e organizzare queste interfacce in una preliminare architettura dei componenti. Il *business type model* è un deliverable dal quale si ricava l'insieme iniziale delle interfacce di business ed è usato in un secondo momento per lo sviluppo del *modello delle informazioni delle interfacce*.
In questa fase bisogna anche tenere conto di tutti quegli elementi che possono rappresentare dei vincoli di progetto come, ad esempio, la necessità di usare dei componenti o delle librerie già esistenti o il doversi attenere ad un certo tipo di architettura.

4.3.2 Identificazione delle interazioni

Questa fase esamina come ciascun'operazione del sistema è realizzata mediante l'interazione fra i componenti; è quindi previsto l'utilizzo di modelli per la descrizione delle interazioni come *collaboration diagram*, *sequence diagram* e così via. In questa fase vengono anche prese decisioni riguardanti la gestione dei riferimenti fra componenti in modo da minimizzare le dipendenze e andare così incontro alle caratteristiche di riusabilità e sostituibilità, ad esempio possono essere fatte delle scelte per raggruppare o fattorizzare operazioni e interfacce.
In sostanza, nella fase d'identificazione delle interazioni emergono tutti i dettagli dell'architettura dei componenti, con una chiara comprensione delle dipendenze

4.3.3 Specifica dei componenti

È la fase finale del workflow di ***specifica*** in cui si descrivono dettagliatamente le operazioni e i constraint. Per una data interfaccia ciò significa, innanzitutto, definire gli stati possibili del componente mediante un *modello delle informazioni dell'interfaccia* insieme agli invarianti e le guardie imposte su di esso, quindi specificare mediante precondizioni e postcondizioni come le operazioni possono agire su tale modello.

In questa fase vengono anche specificati i constraint imposti su un particolare componente e che sono indipendenti dalle interfacce come ad esempio il fatto che il componente, per eseguire un'operazione, debba interagire con un determinato componente.

Questa specifica dettagliata delle operazioni e dei constraints non modifica l'architettura dei componenti, infatti, in questa fase, l'architettura deve essere già definita in maniera stabile.

4.4. Il workflow di provisioning

Dopo avere specificato le interfacce, i componenti e l'architettura mediante la quale questi ultimi sono organizzati, è necessario provvedere alla fornitura dei componenti che rispettano tali specifiche; ciò può avvenire in diversi modi: acquisto presso dei component-market, riuso e/o modifica di altri componenti, creazione ex-novo.

Oltre alla specifica, questi componenti devono essere compatibili con quella che si chiama *target tecnology* cioè l'ambiente di runtime dei componenti. L'ambiente di runtime è un ambiente di oggetti distribuiti su cui è definito un insieme di regole, che i componenti devono rispettare per operare in questo ambiente, e un

insieme di servizi offerti per il supporto alle transazioni, alla sicurezza, alla concorrenza e così via.

Come già discusso nei capitoli precedenti, per la creazione ex-novo di un componente, facendo riferimento allo standard CORBA, è necessario innanzitutto descrivere la specifica delle interfacce dei componenti mediante IDL ed è a questo punto che il grosso lavoro svolto nel workflow di ***specifica*** per la produzione di deliverables dettagliati e accurati, generalmente mediante modelli UML, è drasticamente ridimensionato. IDL, infatti, non permette descrivere constraints, business rules, relationship a causa della sua semantica piuttosto povera. Di conseguenza nel codice generato automaticamente dai compilatori[1] non c'è traccia di queste proprietà e la loro implementazione ricade completamente sugli sviluppatori che quindi, oltre alla *business logic* delle operazioni, devono implementare anche le relationship (associazioni, aggregazioni, composizioni) e i constraint (invarianti, precondizioni, postcondizioni, guardie).

La nuova metodologia, possibile grazie all'utilizzo combinato di SDL (structure definition language) e del tool CO.CO.GEN. (component constraint generator) permette allo sviluppatore di concentrarsi esclusivamente sull'implementazione della *business logic* delle operazioni offerte dalle interfacce sollevandolo quindi dal compito di implementare constraint e relationship[2] con tutto ciò che ne deriva in termini di tempo di sviluppo, e quindi costi, e possibilità di errore.

---

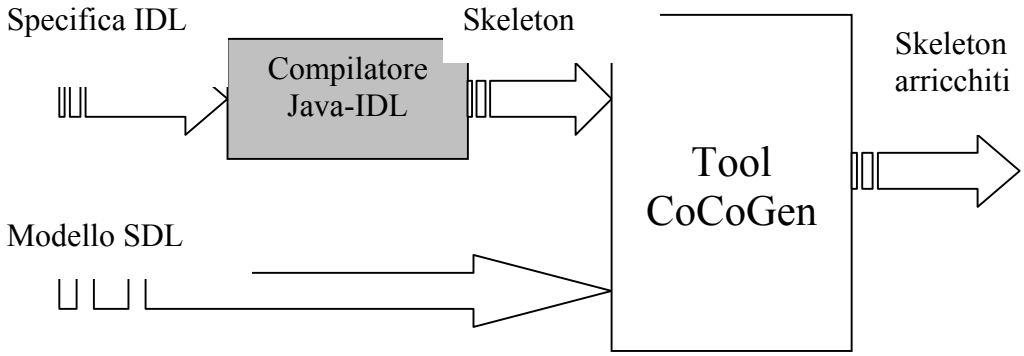
[1] I compilatori IDL possono generare diverse quantità di codice mediante la scelta di alcune opzioni. Alcuni compilatori generano soltanto gli *skeleton* altri possono anche generare le strutture del *servant* che non è altro che la classe che implementa il componente.

[2] Attualmente, il tool implementa automaticamente invarianti, precondizioni, postcondizioni, guardie ma sono allo studio soluzioni per l'implementazione di relationship (aggregazioni, associazioni, composizioni).

4.5 Le modifiche al ciclo di sviluppo

La disponibilità di tools per l'implementazione automatica di constraints in applicazioni a componenti ci permette superare i problemi evidenziati nel paragrafo precedente senza stravolgere il ciclo di sviluppo ma semplicemente apportando delle modifiche ai workflows di ***specifica*** e ***provisioning*** .

Per capire quali sono le modifiche da apportare al ciclo di sviluppo di applicazioni component-based, bisogna richiamare il comportamento esterno del tool CoCoGen progettato nel Capitolo 3 (Figura 4.4). Data specifica dei componenti di un'applicazione espressa in SDL e data la specifica IDL delle interfacce supportate dai componenti, il tool è in grado di interpretare il modello testuale SDL della architettura e andare ad implementare i constraints descritti in tale modello (invarianti, precondizioni, postcondizioni e guardie) nel codice generato automaticamente dal compilatore IDL[3].



[3] Il tool CO.CO.GEN. è stato progettato per implementare constraints su applicazioni a componenti sviluppate su middleware aderenti allo standard CORBA, attualmente le piattaforme supportate dal tool sono Java IDL della SUN MICROSYSTEM e TAO (The ACE Orb) sviluppato in ambiente universitario.
Per Java IDL l'implementazione dei constraints avviene modificando opportunamente gli *skeleton*, invece per TAO l'implementazione avviene aggiungendo del codice nei metodi di quella che è chiamata classe "servant".
La scelta di seguire due tecniche diverse è stata fatta nell'ottica di potenziare il tool per implementare relationship; così facendo in fase di progetto si potranno valutare le possibilità offerte dai due approcci diversi.

Figura 4.4- *Comportamento esterno del tool CoCoGen*

Si capisce quindi che nel workflow di ***specifica***, la descrizione della architettura dei componenti deve essere fatta mediante il linguaggio SDL e non mediante modelli visuali UML come si fa di solito; ciò non rappresenta una limitazione perché nel Capitolo 3 si è detto che UML e SDL hanno uguale potenza espressiva per quanto riguarda la rappresentazione di modelli strutturali.

Per quanto riguarda il workflow di ***provisioning***, dopo la compilazione della specifica IDL si deve usare il tool CoCoGen per implementare le business rules nel codice generato dal compilatore.

Le fasi di descrizione della architettura mediante SDL, specifica delle interfacce in IDL, compilazione del codice IDL, esecuzione del tool CoCoGen, possono essere automatizzate mediante il tool CoCoMod (component constraint modeler), il cui funzionamento è mostrato in Figura 4.5, che permette la modellazione visuale dell'applicazione, basandosi su una notazione UML, nonché la generazione automatica del modello testuale SDL e della specifica IDL, la compilazione della specifica IDL e l'implementazione dei constraints eseguendo il tool CoCoGen.

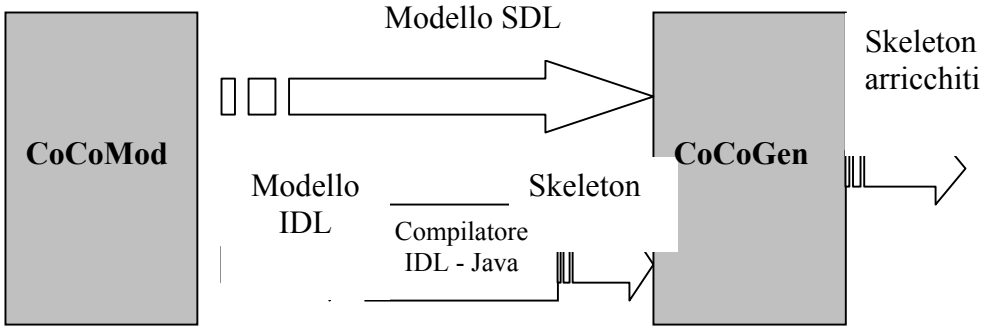


Figura 4.5 – Funzionamento del tool CoCoMod

Tutto ciò rappresenta un'ulteriore semplificazione dello sviluppo di applicazioni a componenti in quanto riduce al minimo la possibilità di errori dovuti alla scrittura del modello SDL e della specifica IDL, in particolare con questo approccio il progettista non è tenuto a conoscere la sintassi di IDL ma solo quella di SDL per la specifica dei constraints.
Le modifiche da apportare al ciclo di sviluppo sono sintetizzate nella Tabella 4.1.

Workflow	Modifiche
Analisi dei requisiti	Nessuna modifica
Specifica	La descrizione della architettura dei componenti deve essere fatta mediante il linguaggio SDL (scritta manualmente o automaticamente eseguendo il tool CoCoMod)
Provisioning	Dopo la compilazione della specifica delle interfacce deve essere eseguito il tool CoCoGen (queste due attività possono essere automatizzate eseguendo il tool CoCoMod)
Assemblaggio	Nessuna modifica
Test	Nessuna modifica
Distribuzione	Nessuna modifica

Tabella 4.1 – *modifiche alla processo di sviluppo*

CAPITOLO V
CASO DI STUDIO

5.1 Introduzione

In questo capitolo è presentato un caso di studio al fine di mostrare l'utilizzo degli strumenti proposti (SDL, CoCoGen e CoCoMod) per l'implementazione di business rules e relationships in una semplice applicazione a componenti. Il ciclo di sviluppo utilizzato sarà conforme a quello mostrato nel capitolo precedente. Naturalmente, data la semplicità della applicazione e lo scopo puramente esplicativo di questo capitolo saranno considerati, con riferimento alla Figura 4.1, solo i workflows di ***analisi dei requisiti***, ***specifica*** e ***provisioning***.

5.2 Workflow di *analisi dei requisiti*

In questo caso di studio, abbiamo considerato un sistema che offre funzioni per la catalogazione, il restauro, il prestito, l'acquisizione, di reperti posseduti da un museo e funzioni per la pianificazione di eventi organizzati dal museo stesso.
I reperti sono classificati per età, nome, stile e tipo. Una scheda di catalogazione che include la descrizione tecnica e una fotografia è associata ad ogni reperto. I reperti sono periodicamente esaminati per stabilire se devono essere restaurati o meno. Le verifiche e i restauri sono eseguiti in laboratori specializzati. Ogni volta che un reperto è verificato o restaurato, un timeout è programmato per stabilire quanto aspettare prima di effettuare la prossima operazione di verifica. Questo timeout deve essere compreso in un intervallo stabilito (ad esempio da 6 a 24 mesi), per ogni tipo di reperto, in base a generici criteri.

I musei possono prestare e/o chiedere in prestito ad altri musei dei reperti per organizzare eventi speciali. I reperti possono essere prestati solo se essi sono disponibili (cioè non in verifica o in restauro). Di conseguenza, i reperti possono essere verificati o restaurati solo se essi sono disponibili e il timeout è scaduto.
In Figura 5.1 è riportato un *class diagram* rappresentante quello il *business concept model* che è un modello concettuale del dominio applicativo il cui scopo è di creare un vocabolario comune per le persone coinvolte nel progetto.
In questo modello ci sono tre classi chiamate rispettivamente *Reperto*, *Museo* e *SchedaReperto*. *Reperto* rappresenta il generico reperto, il quale è posseduto da un museo (rappresentato dalla classe *Museo*). *SchedaReperto* rappresenta la generica scheda di catalogazione associata al reperto.

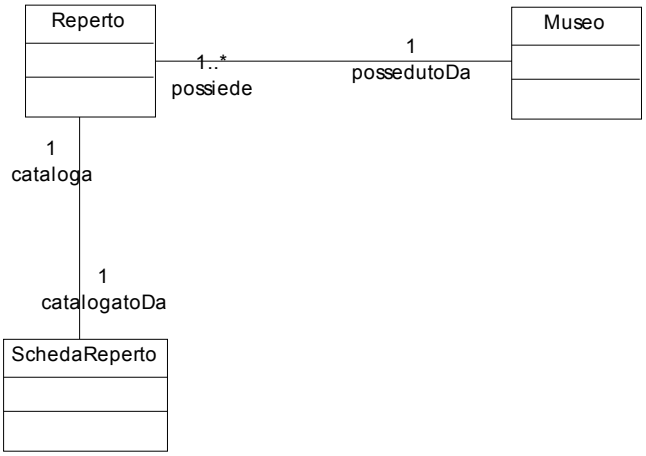


Figura 5.1 – *business concept model*

Dopo avere rappresentato il dominio applicativo mediante il *business concept model* è necessario capire chiaramente quali sono le funzioni e le responsabilità del sistema software, si deve cioè procedere con la *definizione del contesto software*. Per definire tale contesto bisogna decidere, dal punto di vista degli utenti, come deve lavorare il sistema e cosa significa per loro la sua creazione.

I casi d'uso sono il principale meccanismo UML per la definizione del contesto software. Un caso d'uso specifica il comportamento del sistema o di una parte di esso descrivendo un insieme di sequenze di azione che un sistema esegue per raggiungere un obiettivo osservabile ad un attore (utente, dispositivo o altro sistema software). Il diagramma dei casi d'uso è mostrato in Figura 5.2.

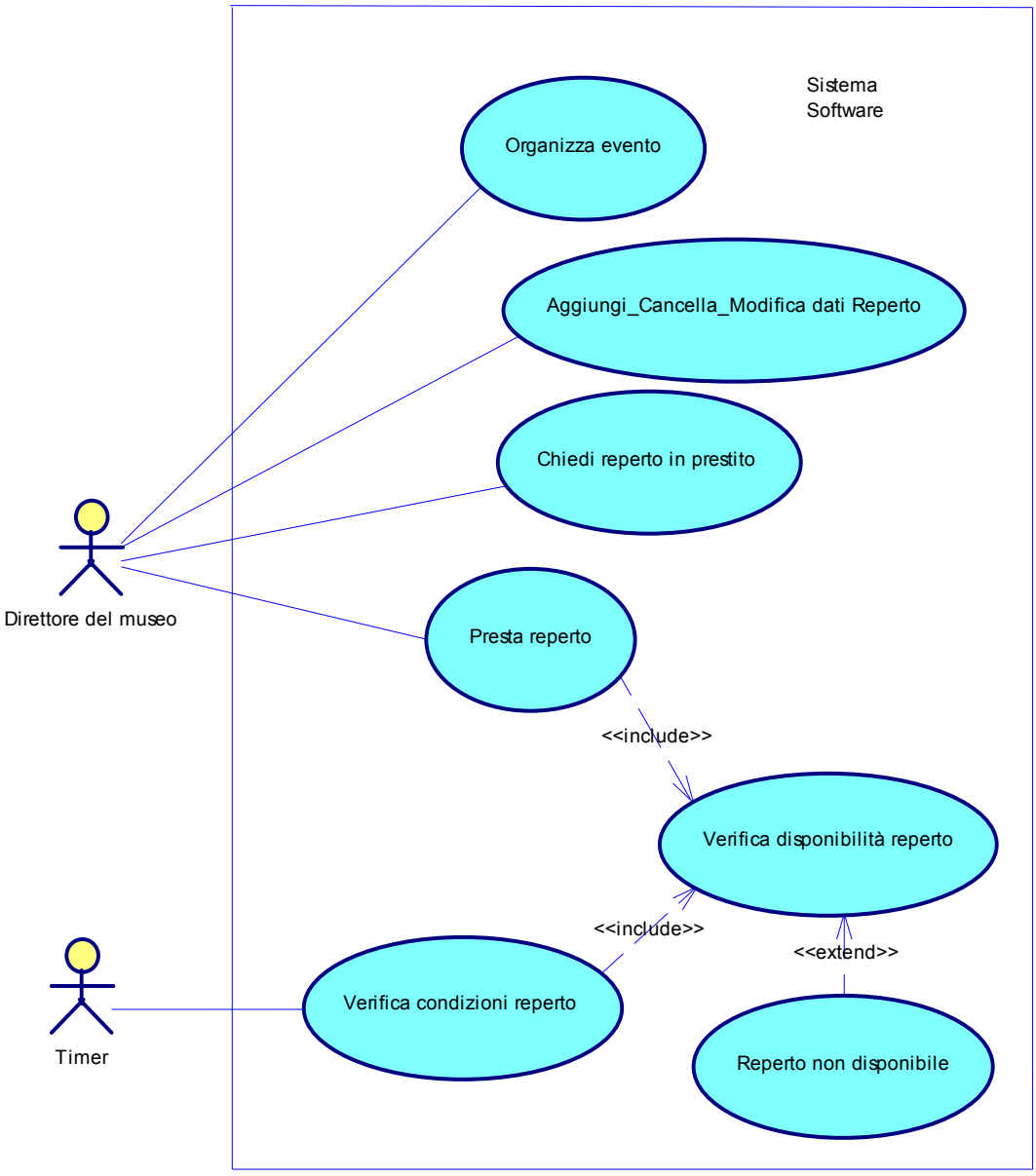


Figura 5.2 – *Diagramma dei casi d'uso*

Di seguito sono riportate le descrizioni di alcuni dei casi d'uso.

Nome ***Organizza evento***
Iniziatore **Direttore del museo**
Obiettivo **organizzare un evento**

Scenario principale di successo

1. il Direttore del museo chiede di organizzare un evento
2. il Sistema fornisce una serie di date disponibili
3. il Direttore sceglie le date in cui si svolgerà l'evento
4. il Sistema aggiorna le date disponibili

Estensioni

3. il Direttore rifiuta le date disponibili
    a. fine caso d'uso

Nome ***Presta reperto***
Iniziatore **Direttore del museo**
Obiettivo **prestare un reperto ad un altro museo**

Scenario principale di successo

1. il Direttore del museo chiede di prestare un reperto ad un altro museo
2. il Direttore sceglie il reperto da prestare
3. **include *Verifica disponibilità reperto***
4. il Sistema procede al prestito del reperto

Nome ***Verifica stato di un reperto***
Iniziatore **Timer**
Obiettivo **verificare lo stato di un reperto**

Scenario principale di successo

1. il Timer segnala un reperto da verificare
2. **include *Verifica disponibilità del reperto***
3. il Sistema ordina il restauro del reperto
4. il Sistema aggiorna il Timer

Nome ***Chiedi reperto in prestito***
Iniziatore **Direttore del museo**
Obiettivo **ottenere un reperto in prestito**

Scenario principale di successo

1. il Direttore decide di chiedere a un museo un reperto in prestito
2. il Direttore seleziona il museo e il reperto
3. il Sistema inoltra la richiesta
4. il Sistema conferma il prestito

Estensioni

5. il reperto non è disponibile
   a. fine caso d'uso

Nome ***Verifica disponibilità reperto***
Iniziatore **incluso nei casi d'uso *Presta Reperto* e *Verifica Reperto***
Obiettivo **verificare la disponibilità di un reperto**

Scenario principale di successo

1. il Sistema verifica la disponibilità di un reperto

Estensioni

1. il reperto non è disponibile
    a. il sistema chiede la restituzione del prestito

Una volta prodotti i casi d'uso e il business concept model, il workflow di ***analisi dei requisiti*** si può considerare finito ed è possibile passare al workflow di ***specifica***.

5.3 Workflow di *specifica*

Come visto nel Capitolo 4, il workflow di specifica può essere diviso in tre fasi: identificazione dei componenti, identificazione delle interazioni e specifica dei componenti. In questo caso di studio seguiremo questa suddivisione.

5.3.1 Identificazione dei componenti

L'obiettivo di questa fase è individuare un insieme iniziale d'interfacce di sistema e d'interfacce di business e da queste definire una prima architettura dei componenti.

Questi deliverables devono essere ricavati da quelli prodotti dal workflow di ***analisi dei requisiti*** e cioè i *casi d'uso* e il *business concept model*.

Le interfacce di sistema devono offrire quelle funzionalità specificate nei casi d'uso, perciò, in generale, di definisce un'interfaccia per ogni caso d'uso e poi, per ogni suo step, ci si accerta se ci sono o meno responsabilità del sistema che devono essere modellate, se così, si rappresentano queste responsabilità come una o più operazioni per l'interfaccia.

Per il caso di studio in esame è possibile raggruppare le funzioni di sistema in tre interfacce come mostrato in Figura 5.3.

L'interfaccia di sistema *IDatiReperto* offre le operazioni per la gestione dei dati del reperto (nome, tipo, stile, età) e quindi le funzionalità che sono state genericamente raggruppate nel caso d'uso *Aggiungi_Cancella_ModificaDati Reperto*

L'interfaccia di sistema *IOperazioniMuseoMuseo* offre le operazioni necessarie per l'organizzazione di un evento e per richiedere in prestito un reperto individuate nei casi d'uso *Organizza evento* e *Chiedi reperto in prestito*.

L'interfaccia di sistema *IStatoReperto* offre le operazioni per la catalogazione, il restauro, il prestito, la restituzione e la verifica di un reperto individuate dai casi d'uso *Presta Reperto*, *Verifica Reperto*, *Verifica disponibilità del reperto*.

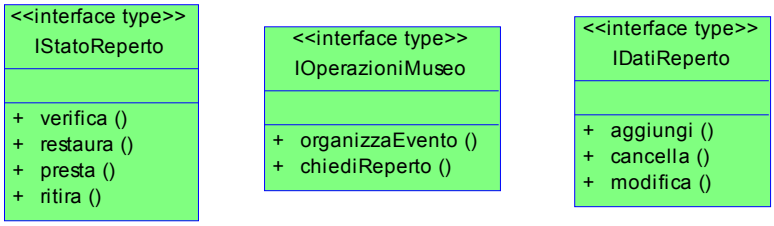


Figura 5.3 – *Interfacce di sistema*

Per l'individuazione delle interfacce di business è necessario innanzitutto creare il *business type model* a partire dal *business concept model*. Il *business type model* è una formalizzazione del *business concept model* il cui scopo è definire la conoscenza che ha il sistema software del dominio applicativo che supporta.

Per il caso in esame questi due deliverables coincidono in quanto il sistema software deve

supportare tutte le attività del dominio applicativo; è possibile però aggiungere al *business type model* le informazioni che riguardano le singole entità rappresentate e che sono state omesse nel *business concept model* come mostrato in figura 5.4.

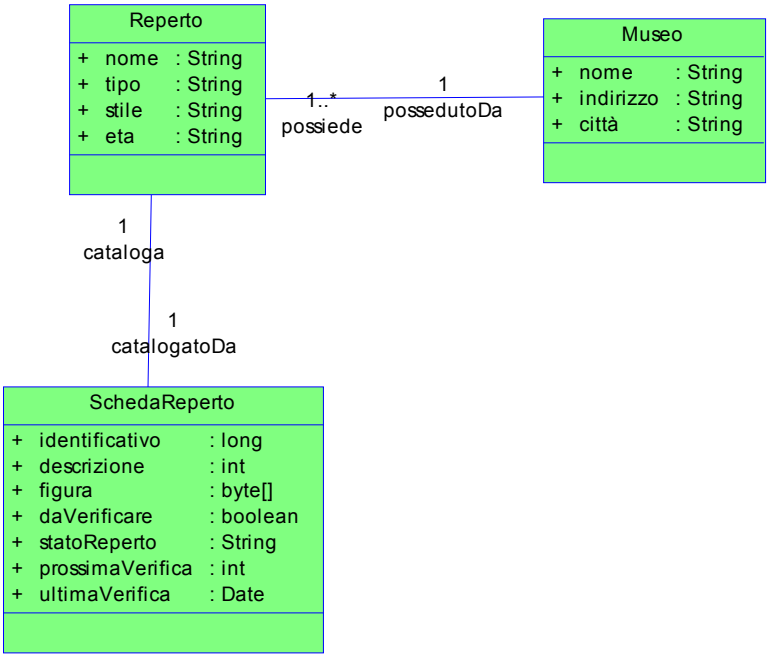


Figura 5.4 – *business type model*

Il business type model rappresenta il *core* delle informazioni del dominio applicativo ed è compito delle interfacce di business offrire le operazioni per la gestione di tali informazioni. Di conseguenza, per il caso in esame, è naturale pensare a tre interfacce di business *IGestioneReperti*, *IGestioneMuseo*,

IGestioneSchedeReperti ognuna adibita alla gestione di un tipo d'informazione presente nel *business type model.*

In realtà, poiché l'interfaccia IGestioneMuseo deve gestire una sola istanza del tipo *Museo* si può pensare di accorpare le sue funzionalità nella interfaccia *IOperazioniMuseo.* In Figura 5.5 sono mostrate le interfacce di business e le responabilità che hanno tali interfacce sul *business type model.*

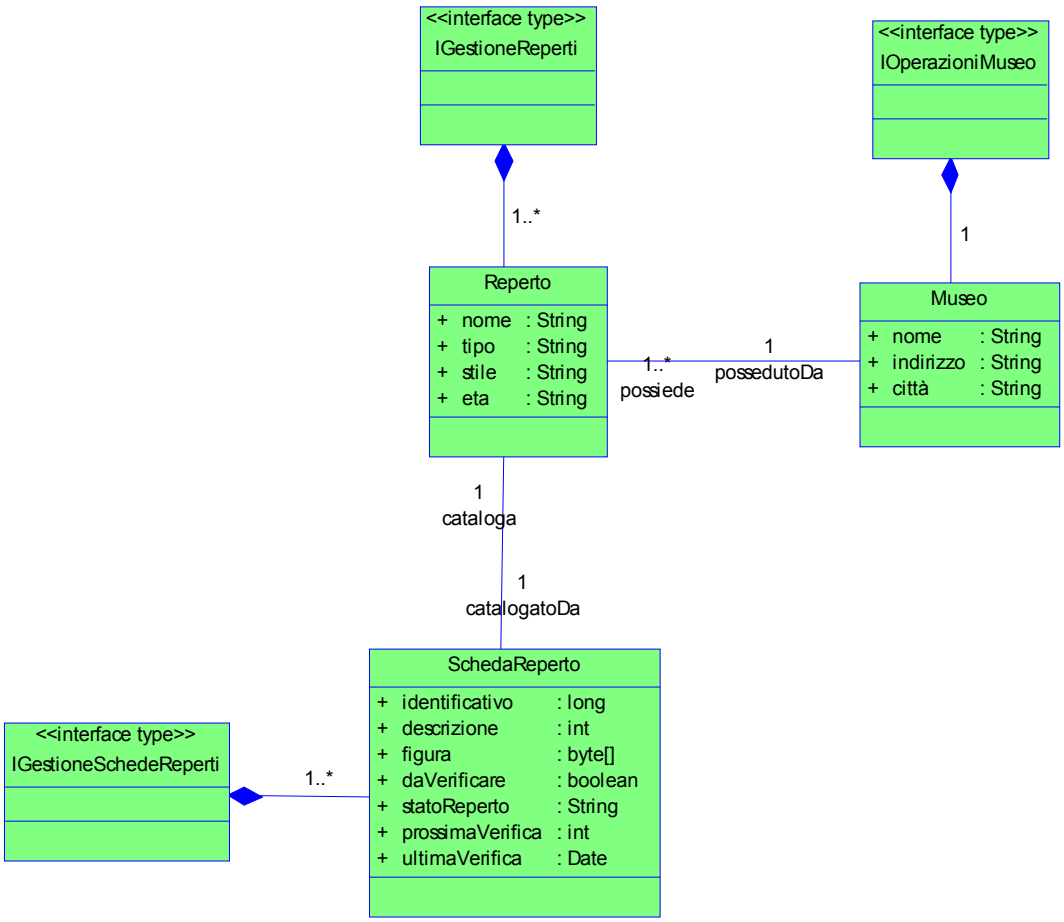


Figura 5.5 – *Responsabilità delle interfacce di business*

Ora che sono state individuate le interfacce di business e di sistema è necessario specificare un insieme di componenti che provvederanno alla loro implementazione. In questo caso è possibile creare un componente per ogni interfaccia poiché i concetti rappresentati nelle interfacce hanno differenti

lifetimes e che le interazioni necessarie fra interfacce di sistema e quelle di business per l'implementazione delle operazioni non coinvolgono grandi quantità di dati. In figura 5.6, sono mostrate le dipendenze fra i componenti e le interfacce: la relazione che lega un'interfaccia al componente che la "offre" è caratterizzata dallo stereotipo <<offers>>, i componenti di sistema dipendono dalle interfacce di business perché queste offrono le funzionalità per accedere ai dati su cui agiscono le operazioni delle interfacce di sistema.

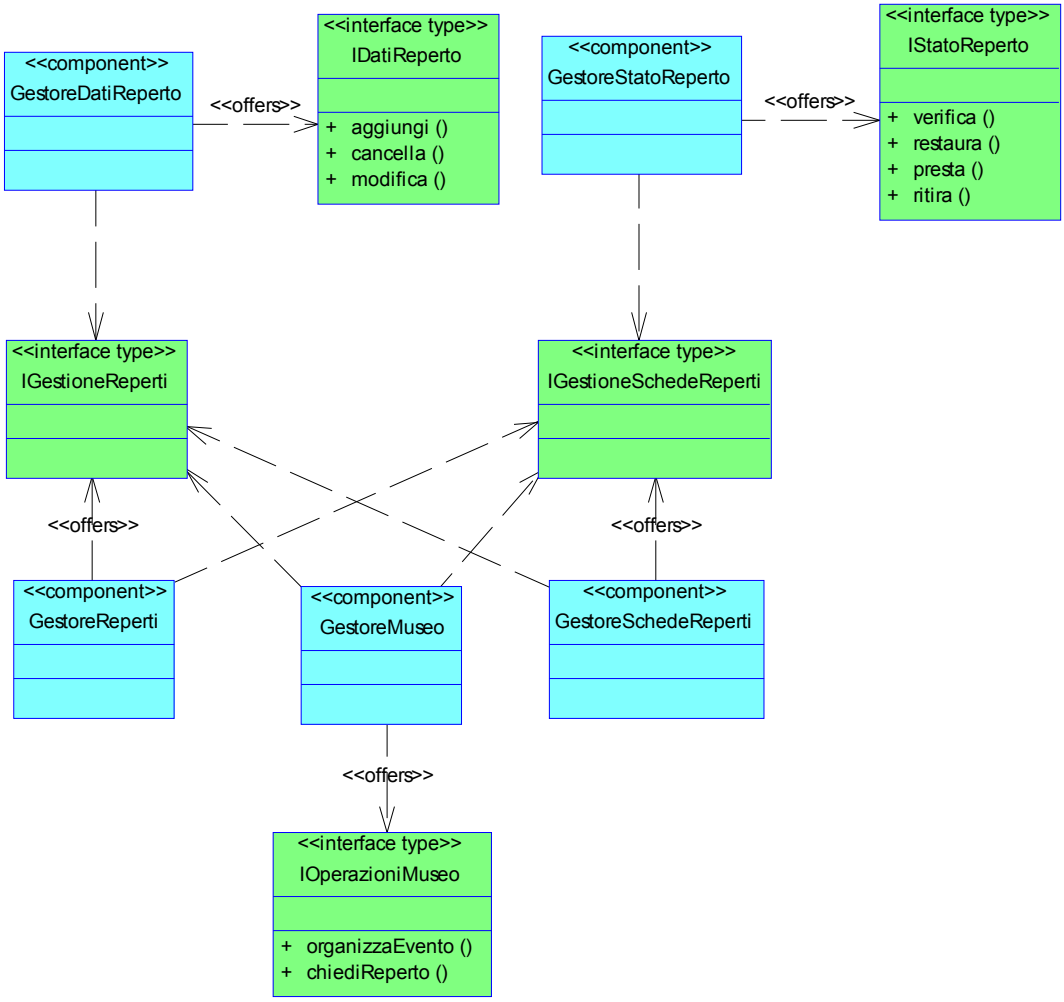


Figura 5.6 *–Relazioni fra componenti e interfacce*

5.3.2 Identificazione delle interazioni

Gli scopi della individuazione delle interazioni sono due: 1) mettere in evidenza le dipendenze i componenti e le interfacce per valutare l'architettura progettata riguardo aspetti come riusabilità, sostituibilità e gestione dei riferimenti fra i componenti (per gli obiettivi di questo capitolo questa valutazione non sarà fatta); 2) aggiungere alle interfacce operazioni non ricavabili dai casi d'uso.
Le interazioni principali sono:

- Componente di sistema *GestoreDatiReperto* – Interfaccia di business *IGestioneReperti* per l'accesso ai dati di un singolo reperto gestiti dal componente di business (devono quindi essere previste operazioni per il prelievo e l'inserimento dei dati di un reperto nella interfaccia di business *IgestioneReperti*)
- Componente di sistema *GestoreStatoReperto* – Interfaccia di business *IgestioneSchedeReperti* per l'accesso alla scheda di catalogazione di un reperto gestita dal componente di business (devono quindi essere previste operazioni per il prelievo e l'inserimento di una scheda di catalogazione nella interfaccia di business *IgestioneReperti*)
- Componente di business *GestoreReperti* – Interfaccia di business *IGestioneSchedeReperti* per la gestione dei riferimenti reperto-scheda
- Componente di business *GestoreSchedeReperti*- Interfaccia di business *IGestioneReperti* per la gestione dei riferimenti scheda-reperto
- Componente di sistema GestoreMuseo – interfacce di business IGestioneReperti e IGestioneSchedeReperti per l'implementazione ulteriori funzionalità

Ora che sono state individuati interfacce, componenti e architettura, è possibile passare alla terza fase del workflow di ***specifica*** e cioè alla fase di *specifica dei componenti.* Ricordiamo che in questa fase dobbiamo definire, per ogni

interfaccia, il *modello delle informazioni dell'interfaccia* con relativi constraints (invarianti e guardie), la *signature* delle operazioni con relativi constraints (precondizioni e postcondizioni) e i constraints sul componente che la implementa (relationship).

Come visto nel Capitolo 4, per l'implementazione delle business rules e delle relationship, l'attività di ***specifica*** va fatta in previsione dell'utilizzo del tool CoCoGen e quindi le interfacce vanno descritte in IDL mentre le business rules, i constraint e le relationship vanno espressi con un modello testuale SDL. In alternativa si è detto che è possibile automatizzare la scrittura di tali modelli, la compilazione di quello IDL e l'esecuzione del tool CoCoGen mediante l'utilizzo del tool CoCoMod, per tale motivo in questo caso di studio verrà usato CoCoMod.

Dopo avere mandato in esecuzione il tool, è possibile scegliere se aprire un progetto già esistente o crearne uno nuovo. Nel nostro caso creiamo un nuovo progetto scegliendo il comando *New* dal menù *File* (Figura 5.7).

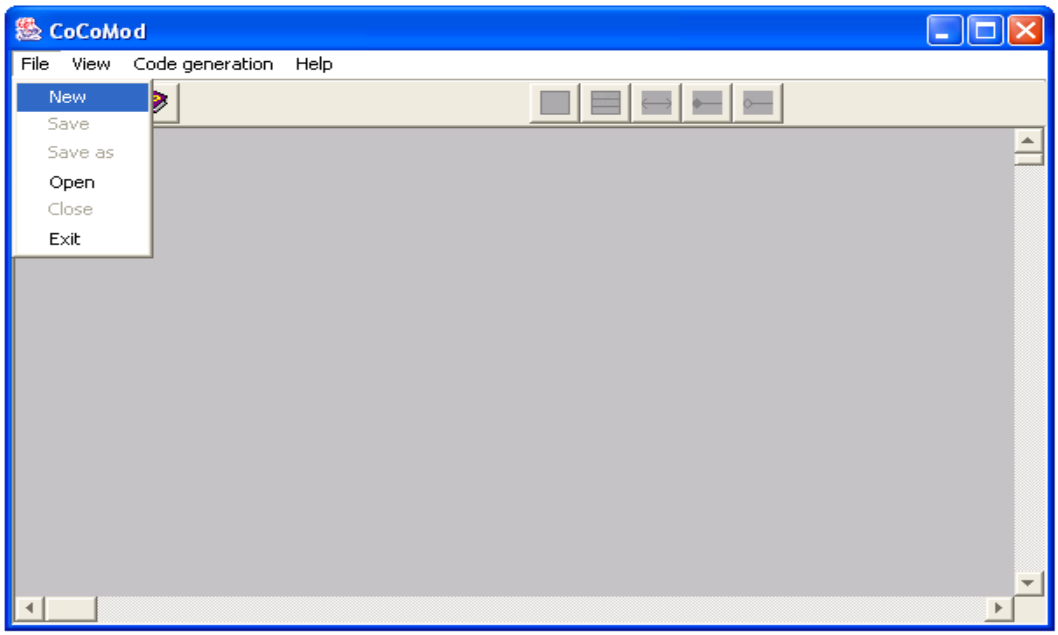


Figura 5.7 – *Creazione di un nuovo progetto*

Una volta creato il nuovo progetto, il tool abilita un'area di disegno per la progettazione visuale di applicazioni a componenti (Figura 5.8).

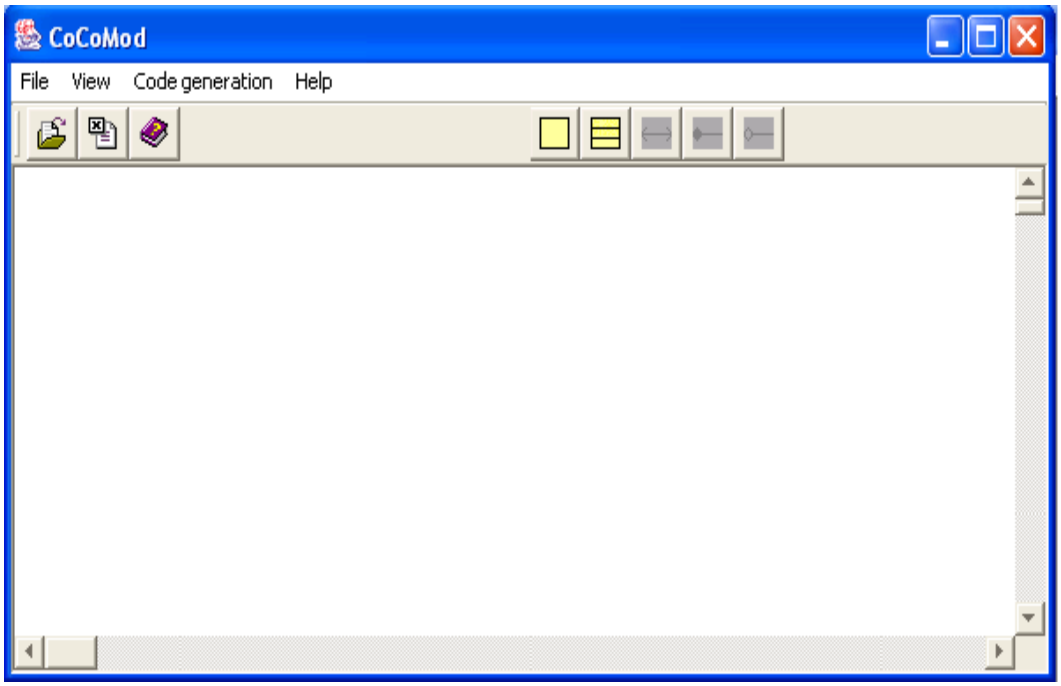


Figura 5.8 – *area di progetto visuale*

Una volta abilitata l'area di progetto è possibile inserirvi degli elementi. Attualmente, è abilitato l'inserimento d'interfacce e moduli: questi ultimi hanno la stessa funzione della dichiarazione *module* utilizzata in IDL per raggruppare definizioni di costanti, tipi, eccezioni e interfacce in un'unica unità logica.

Per inserire un modulo o un'interfaccia basta premere con il puntatore del mouse sull'icona corrispondente sulla barra degli strumenti e fare lo stesso nella posizione desiderata sull'area di progetto.

Per l'applicazione in esame bisogna inserire le cinque interfacce *IOperazioniMuseo*, *IGestioneSchedeReperti*, *IGestioneReperti*, *IStatoReperto*, *IDatiReperto* (eventualmente all'interno di un modulo).

In figura 5.9 è mostrato l'inserimento delle interfacce all'interno di un modulo.

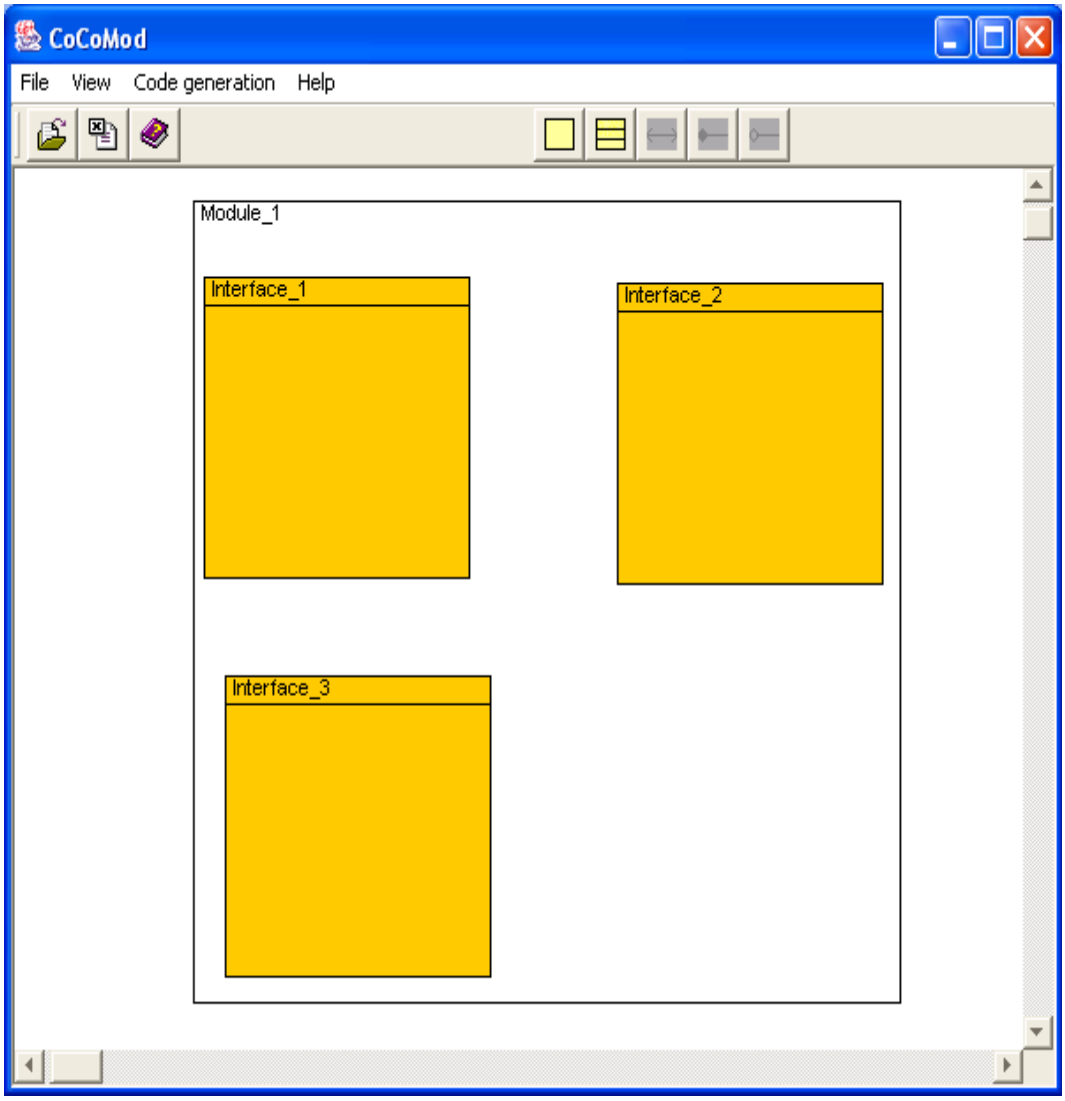


Figura 5.9 – *inserimento delle interfacce in un modulo*

Come si vede dalla Figura 5.9, il tool assegna a moduli e interfacce dei nomi di default, per assegnare dei nomi diversi basta accedere al *form delle proprietà* dell'elemento considerato, posizionandovi il puntatore del mouse e premendo il

tasto destro, e modificare il nome nella scheda *General* come mostrato in Figura 5.10.

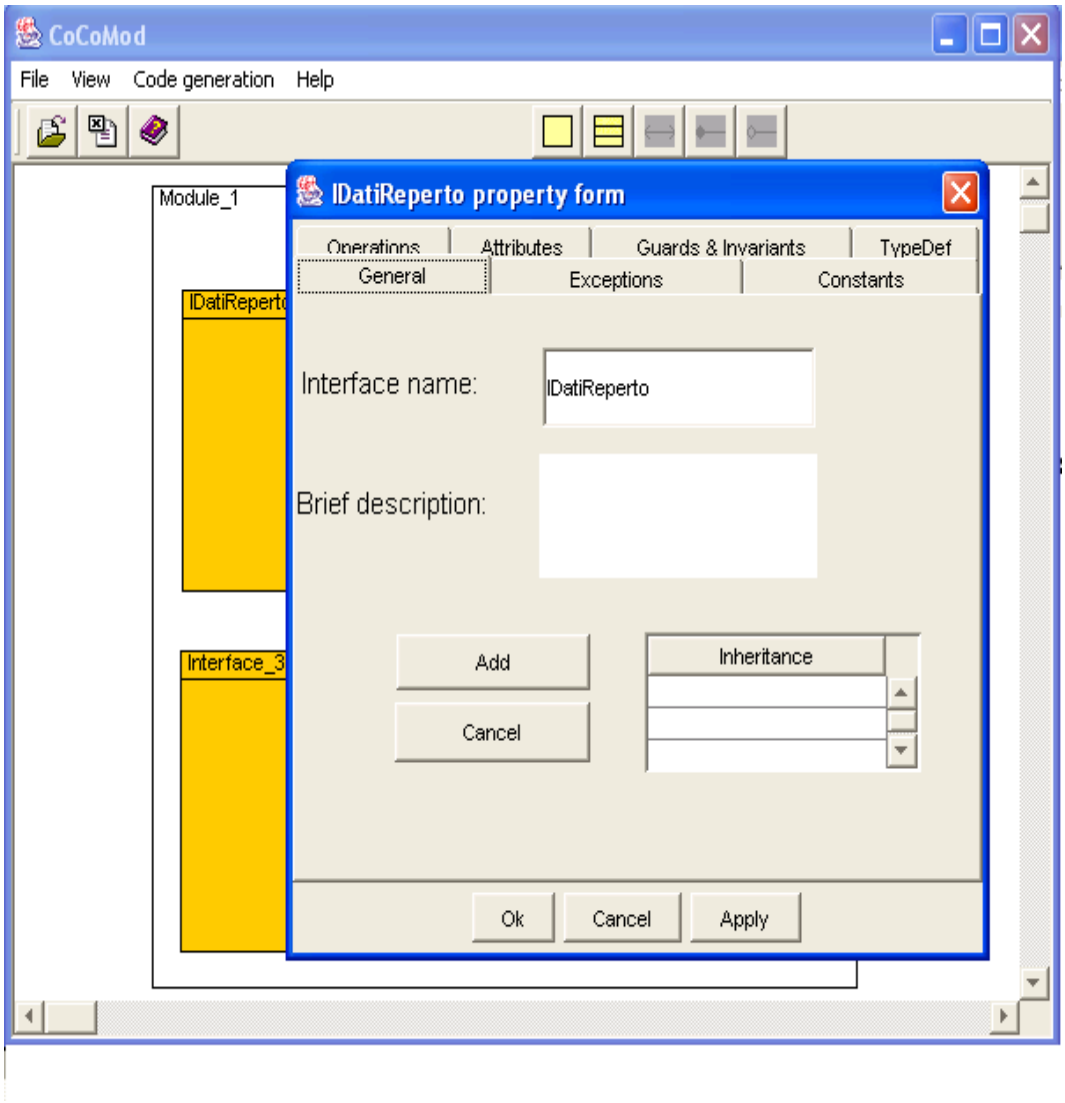


Figura 5.10 – *form delle proprietà*

Una volta inserite le interfacce, bisogna specificare tutte le loro proprietà come costanti, eccezioni, tipi di dati, ma anche il *modello delle informazioni delle interfacce* (definito attraverso gli attributi), con relativi invarianti e guardie, e le operazioni offerte, con relative precondizioni e postcondizioni; ciò può essere

fatto specificando tali proprietà nel *form delle proprietà* di ciascuna interfaccia (Figura 5.11).

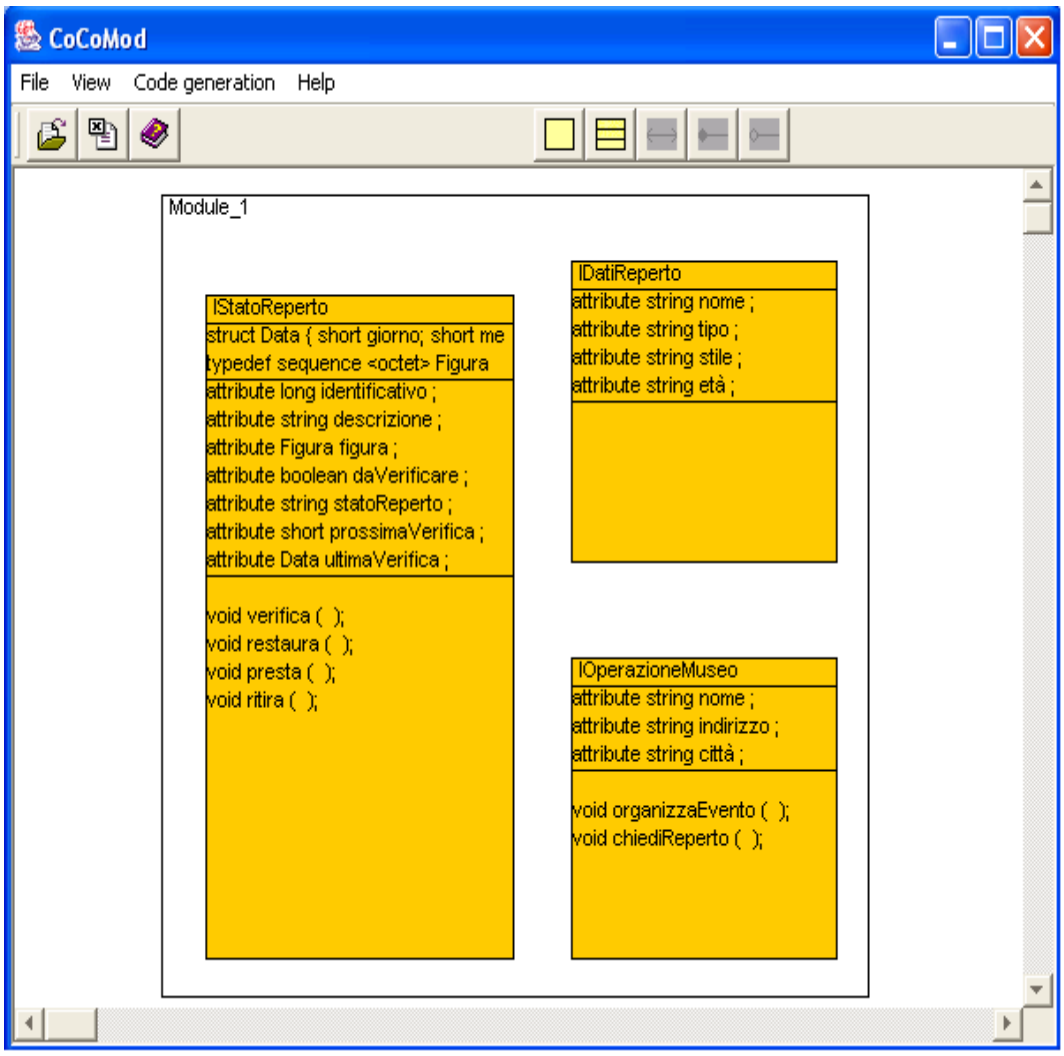


Figura 5.11 – *specifica delle interfacce*

Per quanto riguarda l'interfaccia *IStatoReperto* i constraint da specificare sono:

context IStatoReperto --invariante sull'intervallo fra due verifiche
inv: ((self.prossimaVerifica >= 6) and (self.prossimaVerifica<=24))

context IStatoReperto -- guardia per la verifica del reperto

guard: if ((self.daVerificare == true) and (self.stato==”disponibile”))

then

execute self.verifica()

endif

context IStatoReperto::verifica() -- precondizione per la verifica

pre: ((self.stato == “disponibile”) and (self.daVerificare == true))

context IStatoReperto::verifica() -- postcondizione per la verifica

post: self.stato = “in verifica”

context IStatoReperto::restaura() --precondizione per il restauro

pre: (self.stato ==”disponibile”)

context IStatoReperto::restaura() --postcondizione per il restauro

post: self.stato = “sottoposto a restauro”

context IStatoReperto::presta() -- precondizione per il prestito

pre: (self.stato==”disponibile”)

context IStatoReperto::presta() --postcondizione per il prestito

post: (self.stato==”prestato”)

context IStatoReperto::ritira() --postcondizione per il ritiro di un reperto prestato

post: self.stato=”disponibile”

In Figura 5.12 è mostrato l’inserimento della precondizione e della postcondizione per l’operazione *restaura()*.

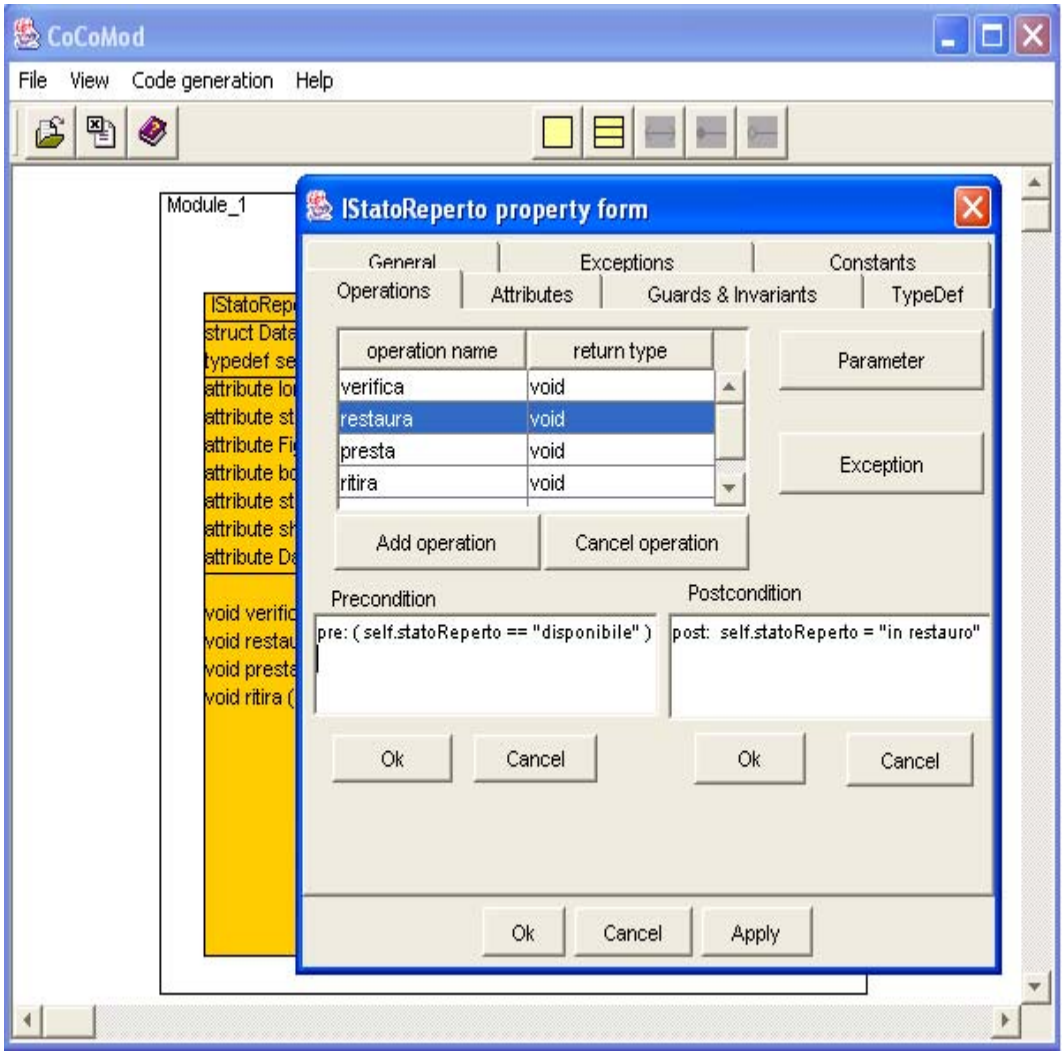


Figura 5.12 – *inserimento di precondizioni e postcondizioni*

Una volta completata la specifica delle interfacce è il workflow di ***specifica*** può considerarsi concluso. È necessario, a questo punto implementare i componenti passare cioè al workflow di ***provisionig***. Come visto nel Capitolo 4, prima di procedere con l'implementazione dei componenti è necessario innanzitutto generare i modelli SDL e IDL, poi compilare il modello IDL e infine lanciare il tool CoCoGen. Per fare ciò mediante il tool CoCoMod bisogna scegliere la

piattaforma di runtime dei componenti al fine di indicare al tool quale compilatore IDL lanciare. Tale scelta può essere fatta selezionando il menu *Code Generation* e da qui *Choose a platform* come mostrato in Figura 5.13.

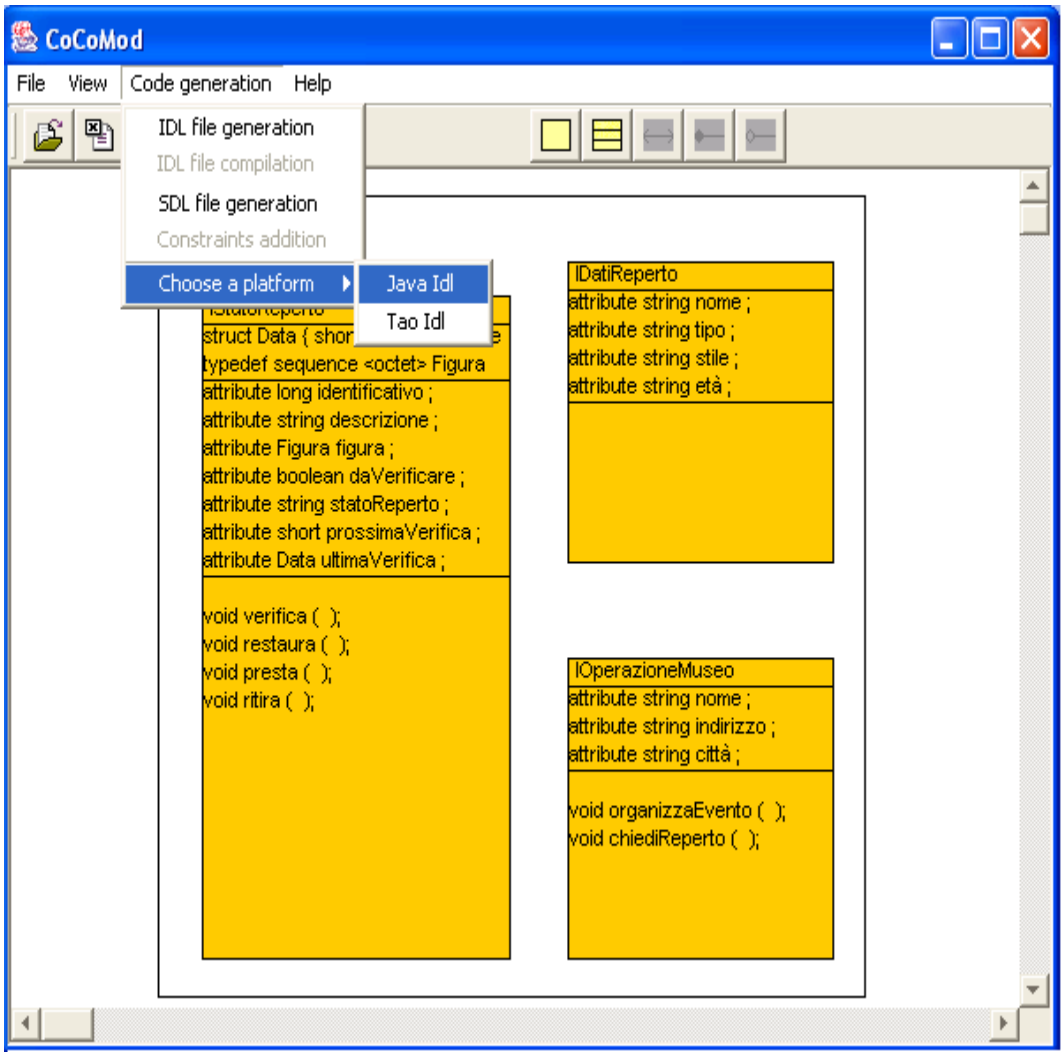


Figura 5.13 – *Scelta della piattaforma*

Scelta la piattaforma si possono generare i modelli IDL e SDL. Ciò può essere fatto selezionando il menu *Code Generation* e scegliendo rispettivamente *IDL file*

generation e *SDL file generation*. I modelli generati possono essere letti scegliendo il menù *View* e scegliendo il modello da visualizzare. Di seguito vengono riporti i modelli SDL (Figura 5.14) e IDL (Figura 5.15) generati dal tool.

```
package Module_1;

context IStatoReperto::verifica (  ): void
post: self.statoReperto = "in verifica"


context IStatoReperto::verifica (  ): void
pre: (( self.statoReperto == "disponibile" ) and (
self.daVerificare == true ))


context IStatoReperto::restaura (  ): void
post:  self.statoReperto = "in restauro"


context IStatoReperto::restaura (  ): void
pre: ( self.statoReperto == "disponibile" )


context IStatoReperto::presta (  ): void
post:  self.statoReperto = "prestato"


context IStatoReperto::presta (  ): void
pre: ( self.statoReperto == "disponibile" )



context IStatoReperto::ritira (  ): void
post: self.statoReperto = "disponibile"


 context IStatoReperto
inv: (( self.prossimaVerifica >= 6 ) and (
self.prossimaVerifica <=24 ))

 context IStatoReperto
guard: if (( self.daVerificare== true ) and (
self.statoReperto == "disponibile" ))
           then
            execute self.verifica()
           endif
endpackage;
```

Figura 5.14 – *Modello SDL*

```
module Module_1 {
 interface IDatiReperto  {
   attribute string nome ;
   attribute string tipo ;
   attribute string stile ;
   attribute string età ;
 };
 interface IOperazioneMuseo  {
   attribute string nome ;
   attribute string indirizzo ;
   attribute string città ;
   void organizzaEvento(  );
   void chiediReperto(  );
 };
interface IStatoReperto  {
   struct Data { short giorno; short mese; long anno;};
   typedef sequence <octet> Figura;
   attribute long identificativo ;
   attribute string descrizione ;
   attribute Figura figura ;
   attribute boolean daVerificare ;
   attribute string statoReperto ;
   attribute short prossimaVerifica ;
   attribute Data ultimaVerifica ;
   void verifica(  );
   void restaura(  );
   void presta(  );
   void ritira(  );
 };
};
```

Figura 5.15 – *Modello IDL*

Una volta generati i modelli è possibile compilare il modello IDL selezionando il menù *Code Generation* e da qui scegliendo *IDL compilation* (Figura 5.16).

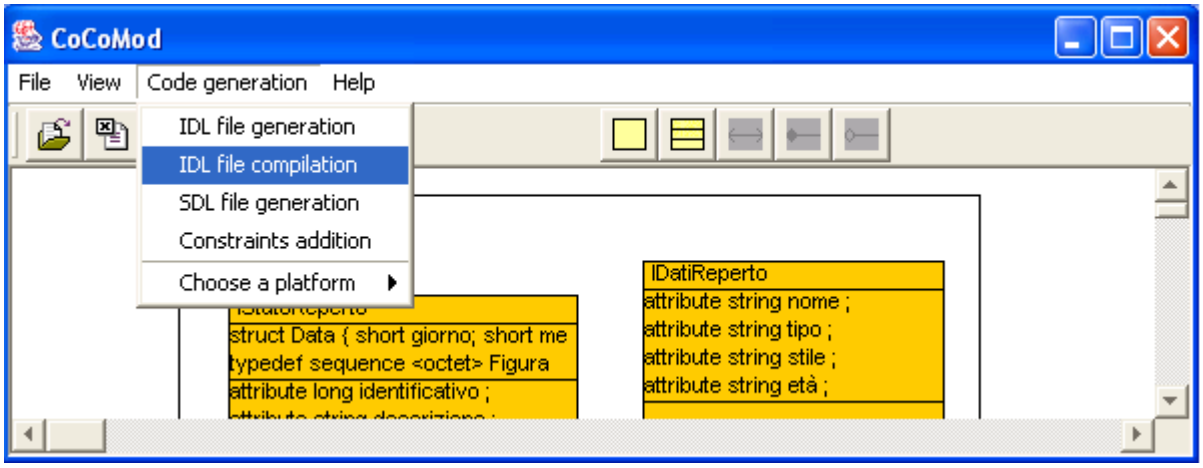


Figura 5.16- *compilazione della specifica IDL*

A questo punto è possibile lanciare il tool CoCoGen per l'implementazione dei constraints presenti nel modello SDL, per lanciare il tool si deve selezionare il menu *Code Generation* e da qui *Constraints Addition* (Figura 5.17).

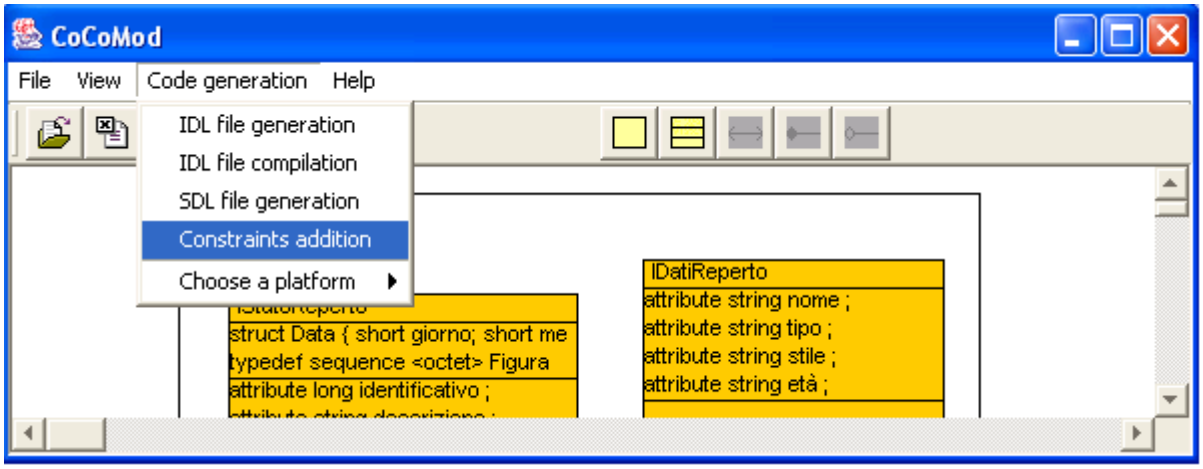


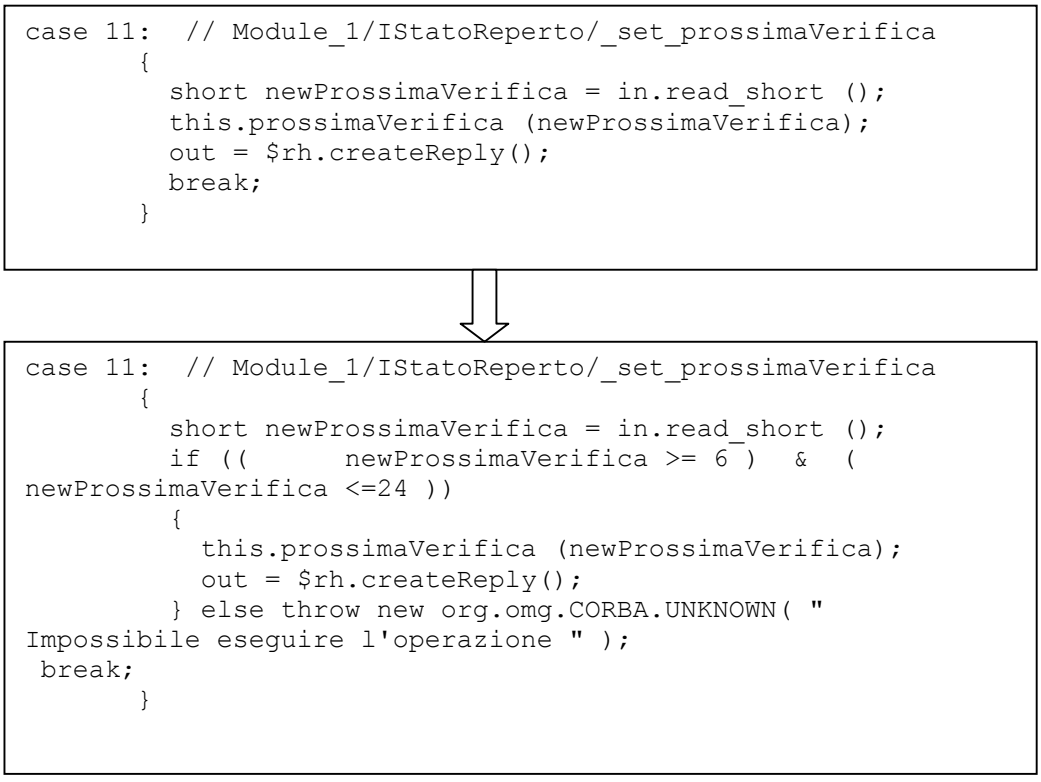
Figura 5.17 – *Implementazione dei constraints*

Una volta implementati i constraints, è possibile fare un confronto fra gli skeleton per l'interfaccia IStatoReperto, che è l'unica su cui sono stati definiti invarianti, guardie, precondizioni e postcondizioni, prima e dopo l'esecuzione di CoCoGen.

Vediamo prima quali sono state le conseguenze dell'implementazione dell'invariante:

context IStatoReperto --invariante sull'intervallo fra due verifiche
inv: ((self.prossimaVerifica >= 6) and (self.prossimaVerifica<=24))

le modifiche sono mostrate in Figura 5.18. Come si può vedere dopo l'implementazione, è aggiunto un controllo sul parametro di input dell'operazione: se il parametro rispetta la condizione allora il valore è aggiornato altrimenti è sollevata un'eccezione.



```
case 11:  // Module_1/IStatoReperto/_set_prossimaVerifica
       {
         short newProssimaVerifica = in.read_short ();
         this.prossimaVerifica (newProssimaVerifica);
         out = $rh.createReply();
         break;
       }
```

```
case 11:  // Module_1/IStatoReperto/_set_prossimaVerifica
       {
         short newProssimaVerifica = in.read_short ();
         if ((      newProssimaVerifica >= 6 )  &  (
newProssimaVerifica <=24 ))
         {
           this.prossimaVerifica (newProssimaVerifica);
           out = $rh.createReply();
         } else throw new org.omg.CORBA.UNKNOWN( "
Impossibile eseguire l'operazione " );
 break;
       }
```

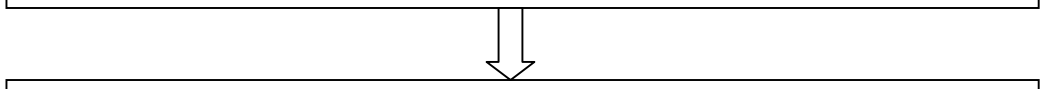
Figura 5.18 – *Implementazione dell'invariante*

In Figura 5.19 sono mostrati gli effetti dell'implementazione della precondizione:

context IStatoReperto::verifica() -- precondizione per la verifica

pre: ((self.stato == "disponibile") and (self.daVerificare == true))

```
case 14:  // Module_1/IStatoReperto/verifica
       {
         this.verifica ();
         out = $rh.createReply();
         break;
       }
```



```
case 14:  // Module_1/IStatoReperto/verifica
       {
       if (( this.statoReperto ().equals(  "disponibile") )
&  (           this.daVerificare ()== true ))
       {
          this.verifica ();
          this.statoReperto ( "in verifica");
          out = $rh.createReply();
       } else throw new org.omg.CORBA.UNKNOWN( " Impossibile
eseguire l'operazione " );
 break;
```

Figura 5.19 – *Implementazione della precondizione*

Come si vede prima dell'esecuzione dell'operazione (this.verifica()) viene valuta la precondizione, se questa condizione è soddisfatta allora l'operazione è eseguita altrimenti è sollevata un'eccezione. Dalla stessa figura si può vedere anche l'effetto della postcondizione:

context IStatoReperto::verifica() -- postcondizione per la verifica

post: self.stato = "in verifica"

in pratica, prima di creare il messaggio di risposta per il cliente (out=$rh.createReply()), è inserita la logica della postcondizione.

Infine, in Figura 5.21 vengono mostrate le conseguenze dell'implementazione della guardia sullo skeleton generato dal compilatore (Figura 5.20).
La guardia ha la seguente espressione:

context IStatoReperto -- guardia per la verifica del reperto

guard: if ((self.daVerificare == true) and (self.stato=="disponibile"))

then execute self.verifica()

endif

```
case 7:  // Module_1/IStatoReperto/_set_daVerificare
       {
         boolean newDaVerificare = in.read_boolean ();
         this.daVerificare (newDaVerificare);
         out = $rh.createReply();
         break;
       }
  .
case 9:  // Module_1/IStatoReperto/_set_statoReperto
       {
         String newStatoReperto = in.read_string ();
         this.statoReperto (newStatoReperto);
         out = $rh.createReply();
         break;
       }
```

Figura 5.20 – *skeleton prima della implementazione*

come si vede dalla Figura 5.21, nell'operazione di *set* degli attributi coinvolti nella condizione della guardia è inserita, dopo l'aggiornamento dell'attributo, la condizione stessa; se tale condizione è verificata viene eseguita l'operazione indicata dalla guardia (in questo caso *verifica()*).

```
case 7:  // Module_1/IStatoReperto/_set_daVerificare
       {
         boolean newDaVerificare = in.read_boolean ();
         this.daVerificare (newDaVerificare);
         if (( this.daVerificare()== true )  &  (
this.statoReperto ().equals(  "disponibile") ))
         {
           if (( this.statoReperto ().equals(  "disponibile")
)  &  ( this.daVerificare ()== true ))
           {
             this.verifica ();
             this.statoReperto ( "in verifica");
             out = $rh.createReply();
           } else throw new org.omg.CORBA.UNKNOWN( "
Impossibile eseguire l'operazione " );
         }else out = $rh.createReply();
         break;
       }
     .
     .
case 9:  // Module_1/IStatoReperto/_set_statoReperto
       {
         String newStatoReperto = in.read_string ();
         this.statoReperto (newStatoReperto);
         if (( this.daVerificare()== true )  &  (
this.statoReperto ().equals(  "disponibile") ))
         {
           if (( this.statoReperto ().equals(  "disponibile")
)  &  ( this.daVerificare ()== true ))
           {
             this.verifica ();
             this.statoReperto ( "in verifica");
             out = $rh.createReply();
           } else throw new org.omg.CORBA.UNKNOWN( "
Impossibile eseguire l'operazione " );
         } else  out = $rh.createReply();
         break;
       }
```

Figura 5.21- *effetto dell'implementazione della guardia*

5.4 Conclusioni

I maggiori contributi forniti da questa tesi sono:

- un linguaggio testuale di modellazione attraverso il quale è possibile riempire il gap semantico esistente tra i modelli visuali e gli attuali modelli testuali; inoltre, tale linguaggio permette di esprimere efficientemente business rules come composizione di invarianti, guardie, precondizioni e postcondizioni;
- un processo di sviluppo per applicazioni distribuite basato sull'utilizzo di un tool capace di modificare il codice generato dai compilatori IDL per implementare i constraints modellati;
- la certezza che i processi di sviluppo e gli strumenti per le applicazioni a componenti possono essere migliorati per ottenere una riduzione dei tempi e dei costi di sviluppo.

I lavori futuri potranno estendere il numero e il tipo di business rules implementabili (come quelle che prevedono la navigazione di relazioni) attraverso la definizione di nuovi patterns implementativi. È anche ipotizzabile l'aumento delle piattaforme supportate come, ad esempio, ORBIX2000™, Imprise Visibroker ecc. Infine, bisognerà analizzare come l'implementazione di una business rules influenza quella delle altre, in scenari più complessi.

BIBLIOGRAFIA

[1] G.T. Heineman e W.T. Councill, "*Component-Based Sotware Engineering*", Addison-Wesley publishing, 2001.

[2] G. Booch, J. Rumbaugh e I. Jacobson, "*The Unified Modeling Language User Guide*", Addison-Wesley publishing, 1999.

[3] P. Chandrasekaran, "*How Use Case Modeling Policies Have Affected the Success of Various Projects (Or How to Improve Usa Case Modeling)*", in the proc. Of Conference on Object Oriented Programming Systems Languages and Applications as Addendum to the 1997 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications.

[4] Medidovic, D. S. Rosenblum, D. F. Redmiles, e J. E. Robbins, "*Modeling Software Architectures in the Unified Modeling Language*", ACM Transactions on Software Engineering and Methodology, Vol 1, N. 1, January 2002, Pages 2-57.

[5] S. Russo and C. Savy and D. Cotroneo and A. Sergio, "*Introduzione a CORBA*", McGraw-Hill publishing, 2002.

[6] OMG, "*The OMG Interface Definition Language*", disponibile sul sito www.omg.org.

[7] OMG, "*OMG-Unified Modeling Language Specifications*", v 1.4, capitolo 6, disponibile sul sito www.omg.org.

[8] OMG, "*Relationship Service Specification*" version 1.0, April 2000, available at http://www.omg.org

[9] R. Michell e J. McKim, "*Design by contract*", TOOLS USA '96 Tutorial

[10] D.F. D'Souza e A.C. Wills, "*Objects, Components, and Framwiorks with UML: The Catalysis Approach*", Addison-Wesley, 1999

[11] J.J. Odell "*Advanced Object-Oriented Analysis and Design Using UML*", Cambridge University Press, 1998.

[12] R.G. Ross, "*The Business Rule Book*", Business Rule Solutions, 2nd Ed., 1997.

[13] R.G. Ross, "*Business Rule Concept*", Business Rule Solutions, 1998.

[14] B. Meyer, "*Object-Oriented Software Construction*", Prentice-Hall, New York, 1988.

[15] J. Cheesman and J. Daniels, "*UML Components – A Simple Process for Specifying Component-Based Software*", Addison-Wesley, 2003

[16] "*ORBIX 2000 Tutorial*", available at http://www.iona.com/docs

[17] http://www.cs.wustl.edu/~schmidt/TAO.html

[18] G. Di Lucca, slide del corso di Ingegneria del Software, A.A. 2002-2003

[19] H. Schildt, "*Fondamenti di Java 2*", McGrawHill, 2003

[20] J. Jaworski, "*Java 1.2 Unleashed*", Apogeo, 1999

[21] http://www.sun.com

[22] A. Coronato, M. Cinquegrani e G. De Pietro, "*Adding Business Rules and Constraints in Component Based Applications*", in proc. of the 2002 International Symposium on Distributed Objects and Applications (DOA 2002), Irvine, California, USA - LNCS 2519