I fondamenti ontologici dei linguaggi di programmazione orientati agli oggetti: i casi delle relazioni e dei ruoli

Matteo Baldoni[1], Guido Boella[1], e Leendert van der Torre[2]

[1] Dipartimento di Informatica - Università di Torino - Italy
baldoni@di.unito.it http://www.di.unito.it/~baldoni
guido@di.unito.it http://www.di.unito.it/~guido

[2] University of Luxembourg - Luxembourg
leendert@vandertorre.com http://agamemnon.uni.lu/ILIAS/vandertorre

Abstract. In this paper we consider the ontology behind Object Oriented programming languages. We show that two conceptual primitives are missing: relations and roles. We simulate relations in a programming language by means of objects. To introduce roles we provide an ontological definition of roles and use it to extend the Java language.

Sommario. In questo articolo esaminiamo l'ontologia sottostante i linguaggi di programmazione orientati agli oggetti. Mostriamo che due primitive concettuali mancano all'appello: le relazioni e i ruoli. Introduciamo le relazioni simulandole per mezzo di oggetti. Per introdurre i ruoli partiamo da una definizione ontologica di ruolo e la utilizziamo per estendere il linguaggio Java.

1 Introduzione

Una ontologia nell'ambito dell'informatica è definita da Gruber (1993) come una specifica di una concettualizzazione: una rappresentazione precisa in un linguaggio formale dei concetti di un dominio e delle primitive usate per definire tali concetti.

I linguaggi usati dall'informatica per scrivere programmi, dal Fortran a Java, dal Lisp al C++, hanno tutti alle loro spalle una propria concettualizzazione della computazione che vogliono realizzare. Tale concettualizzazione è riflessa nei costrutti che permettono ai programmatori di raggiungere i loro obiettivi: variabili, assegnazione di variabili, cicli, condizioni, strutture dati, eccezioni, ecc. Diversi linguaggi di programmazione usano spesso concettualizzazioni della computazione molto divergenti.

I primi linguaggi storicamente sviluppatisi, detti imperativi (quali FORTRAN, COBOL, Pascal, C), fanno riferimento ad una concettualizzazione che ricalca il funzionamento delle architetture di Von Neumann dei computer sottostanti: una memoria contenente i dati e i programmi su cui opera un processore. I costrutti principali permettono al processore di modificare i valori delle variabili contenuti in memoria e di saltare da un punto all'altro del programma da eseguire contenuto in memoria.

Una possibile alternativa è quella dei linguaggi cosidetti funzionali (quali Lisp, Scheme, ML), che basano la loro concettualizzazione sul concetto di funzione. I loro costrutti principali sono proprio ispirati al modello della definizione di una funzione,

http://www.dif.unige.it/epi/networks

***Networks* 6**: 79-89, 2006
 http://www.swif.uniba.it/lei/ai/networks/

dell'applicazione di valori ad una funzione, e della riduzione dell'espressione risultante ad un nuovo valore come risultato.

Il paradigma di programmazione che ha avuto però più successo negli ultimi anni è stato la cosidetta *object orientation* (termine traducibile in programmazione orientata agli oggetti, OO nel seguito, si veda anche Booch (1988)). Tale paradigma introduce una nuova primitiva concettuale estranea alle tradizioni precedenti: quella dell'oggetto come insieme di proprietà e procedure che operano su queste proprietà. La programmazione imperativa è invece più orientata alla gestione del flusso di esecuzione, quella funzionale sulla nozione di procedura che manipola dei dati.

L'idea base dell'OO è che valori complessi composti da diversi attributi debbano essere strettamente associati alle operazioni che li manipolano. L'unione di dati e procedure relative è chiamato oggetto. L'interazione con un oggetto non avviene come nella programmazione imperativa per mezzo della manipolazione diretta degli attributi di un certo dato, ma solo attraverso le operazioni consentite dall'oggetto a cui il dato appartiene e definite al suo interno. L'implementazione di una operazione è specificata all'interno di un oggetto perché solo da quella posizione è possibile accedere agli attributi privati dell'oggetto. Le possibili operazioni effettuabili su un oggetto sono specificate da una interfaccia. Questa strutturazione raggiunge il principio della astrazione dei dati: la possibilità di manipolare un dato con una operazione deve essere separata dal modo in cui il dato è realizzato per mezzo di attributi e operazioni. Solo in questo modo si può lasciare la libertà di cambiare l'implementazione interna di una struttura dati e delle sue operazioni senza cambiare le possibilità di interazione che offre.

La programmazione OO è ispirata al modo con cui interagiamo con gli oggetti nel mondo. Ad es., possiamo compiere una stessa operazione, come schiacciare un pulsante, per accendere diversi tipi di dispositivi. Per poter interagire con questi oggetti non è necessario sapere i dettagli di come avviene l'accensione (si pensi a tutte le operazioni necessarie all'accensione di un computer che sono per lo più ignorate dall'utente). Per interagire con tali dispositivi l'importante è sapere che esiste una operazione di accensione per metterli in funzione e come invocarla.

Nonostante il successo del paradigma OO, quali siano le specifiche della sua concettualizzazione della computazione non è ancora del tutto chiaro. In altre parole non esiste una ontologia precisa che specifichi la concettualizzazione sottostante l'OO. A dimostrazione dei problemi che possono emergere dalla mancanza di una relazione esplicita fra ontologia e linguaggi di programmazione in questo articolo esamineremo due casi esemplari: la mancanza dei concetti di relazione e ruolo accanto a quello di oggetto nei linguaggi di programmazione più diffusi.

Mentre i linguaggi di rappresentazione, quali UML, OWL, ecc., sono basati sulla distinzione fra entità (o oggetti) e relazioni, nei linguaggi di programmazione questa distinzione non è presente. Faremo vedere come è possibile simulare le relazioni all'interno dell'ontologia dell'OO. Nel secondo caso invece, anche nell'ambito delle ontologie fondazionali non è chiaro cosa sia un ruolo. Per Steimann (2000) la dualità fra oggetti e relazioni è connaturata al modo di pensare degli esseri umani, ma esiste evidenza del fatto che tale dualità deve essere complementata da una terza nozione ugualmente importante: quella di ruolo. In questo articolo faremo vedere come partendo da una definizione di ruolo sviluppata in ambito ontologico e condivisa sotto molti aspetti

da diversi autori, si possa introdurre i ruoli in un linguaggio di programmazione OO.

2 Le relazioni in OO

2.1 Gli oggetti e le loro relazioni

Molti autori, tra i quali Rumbaugh (1987), hanno affermato la necessità di introdurre la nozione di relazione come cittadina di prima categoria nei linguaggi di programmazione. Per Rumbaugh la nozione di relazione è complementare a quella di oggetto e altrettanto importante.

Si consideri ad esempio un dominio quale l'università, composto da oggetti quali gli studenti, i professori, i corsi, gli esami. Ognuna di queste categorie è collegata da relazioni con le altre: gli studenti frequentano i corsi, i professori li insegnano, gli studenti sostengono gli esami, ecc.

Ad una prima analisi le relazioni potrebbero essere modellate per mezzo degli attributi degli oggetti: come uno studente ha un numero di matricola, così frequenta un corso. Questa soluzione ha delle ovvie limitazioni: uno studente può frequentare anche più di un corso e ad un corso potrebbe essere a sua volta collegato un certo numero di studenti. Inoltre, se consideriamo il caso degli esami, uno studente che sostiene un esame acquisisce degli attributi aggiuntivi, quali la votazione ottenuta all'esame, che non esistono al di fuori di tale relazione. Analogamente è possibile associare ad una relazione alcune operazioni specifiche che non esistono indipendentemente da tale relazione, nel nostro caso, ad es., rispondere ad una domanda all'interno di un esame. Se si riducessero, come proposto sopra, le relazioni ad attributi di oggetti, gli attributi di una relazione e le sue funzionalità verrebbero divise arbitrariamente fra i possibili partecipanti alla relazione (ad es., studenti ed esami), anche se questi magari non entreranno mai a far parte di tale relazione.

Esistono anche ragioni di altro tipo per motivare l'introduzione in un linguaggio di programmazione di un costrutto esplicito per le relazioni. In particolare, i metodi di analisi di un problema e linguaggi per il design di sistemi informatici, quali ad es. UML (Unified Modelling Language, si veda Rumbaugh *e altri* (2004)), utilizzano tale nozione. Ad es., in UML le relazioni sono chiamate associazioni. Dato che i linguaggi di programmazione usati per implementare un sistema informatico non hanno una nozione esplicita di relazione, si perde il collegamento fra i concetti usati nell'analisi e nella descrizione del sistema da una parte e quelli usati nella sua implementazione dall'altra.

2.2 Un esempio: relationship design pattern

Non esiste ad oggi un linguaggio di riferimento che includa le relazioni come costrutto di base. Uno dei motivi principali è che le relazioni possono essere comunque simulate in OO con i costrutti esistenti, ottenendo almeno in parte gli stessi obiettivi, il lavoro di Noble (1997) ne è un esempio. La simulazione delle relazioni avviene tramite l'operazione conosciuta nell'ambito delle ontologie e più in generale della rappresentazione della conoscenza come "*reificazione*". Ovvero, letteralmente, le relazioni vengono viste come insiemi di oggetti rappresentanti l'estensione della relazione (cioè le coppie di elementi collegati dalla relazione).

```
class Student {
  String name;
  int number;
  HashSet<Course> attends;
  int totalWorkload() {
    int total = 0;
    for(Course c : attends) {
      total = total +
              c.workload;}
    return total;
  }
}
```

```
class Course {
  String code;
  String title;
  int workload;
  HashSet<Student> attendees;
  HashSet<Course> prerequisites;
  void enrol(Student s) {
    if(s.totalWorkload() < 40) {
      attendees.add(s);
      s.attends.add(this);}
  }
  void withdraw(Student s) {
    attendees.remove(s);
    s.attends.remove(this);
  }
}
```

Figura 1. Esempio di relazione studenti e corsi.

In Figura 1 è mostrata la soluzione senza reificazione: studenti e corsi sono legati indirettamente da attributi distribuiti nelle rispettive classi (`attends` e `attendees` rispettivamente). La relazione può anche essere specificata tra gli elementi di una stessa classe come rappresentato da `prerequisites`.

Per ottenere la reificazione è necessario definire una classe che rappresenta, non la relazione direttamente, ma come creare le istanze della relazione reificata. In Figura 2 introduciamo quindi una classe `Attends` le cui istanze contengono i riferimenti ai partecipanti alla relazione. In questo modo, le informazioni riguardanti ogni coppia di elementi collegati sono presenti in un tutto coeso, e gli attributi e le operazioni proprie di una relazione possono essere collocate in un unico posto. Infine, la riusabilità, leggibilità e modificabilità del programma sono incrementati, dato che la classe rappresentante la relazione permette di fattorizzare tutti gli elementi rilevanti.

La soluzione che è stata usata nell'esempio può essere generalizzata in uno schema da utilizzare per introdurre le relazioni in un programma, un cosidetto *design pattern* come introdotto da Gamma *e altri* (1995). È quindi possibile, simulare le relazioni con gli oggetti, anche se questo comporta un lavoro aggiuntivo per il programmatore e facilita l'introduzione di inconsistenze nel programma.

Ma se consideriamo gli esempi visti sopra da un punto di vista ontologico troviamo che molte delle entità che partecipano ad una relazione hanno un carattere particolare. Ad es., studenti e professori non sono due categorie che rappresentano cosidetti tipi naturali, cioè quelle categorie che determinano permanentemente l'essenza di una entità, come invece la categoria persona (si veda Guarino e Welty (2002)). Sono piuttosto dei ruoli, delle categorizzazioni temporanee di entità in un certo contesto. Come vedremo nella prossima sezione, una analisi ontologica dei ruoli può aiutare a costruire un nuovo

```
class Student {
  String name;
  int number;
}
class Course {
  String code;
  String title;
}
```

```
class Attends {
  Course attended;
  Student attendee;
  Attendes(Student s, Course c) {
    attended = c;
    attendee = s;
  }
}
```

Figura 2. Esempio di relazione reificata studenti e corsi.

linguaggio di programmazione.

3 I ruoli in OO

3.1 Una definizione ontologica di ruolo

Nonostante siano state fatte varie proposte su come introdurre i ruoli nei linguaggi di programmazione, si vedano i lavori di Herrmann (2002); Kristensen e Osterbye (1996); Papazoglou e Kramer (1997); Albano *e altri* (1993); Gottlob *e altri* (1996), c'è scarso consenso su quali siano le proprietà dei ruoli e il loro uso. Una possibile ragione per tale divergenza è il fatto che la nozione di ruolo appartiene al senso comune e, in quanto tale, ha dei contorni imprecisi: ognuno ha delle personali intuizioni a riguardo del suo significato, ma parzialmente diverse da quelle degli altri.

Per evitare di dover ricorrere esclusivamente all'intuizione e per prescindere da necessità pratiche immediate, in questo articolo introduciamo la nozione di ruolo in un linguaggio di programmazione che sia ben fondata. Per questo partiamo da come i ruoli sono definiti nelle ontologie, dove si ha un maggiore consenso su cosa siano i ruoli, oltre che una maggiore precisione nel definirne il significato.

La definizione proposta da Boella e van der Torre (2004, 2005) si basa sulla metafora dell'organizzazione. Le organizzazioni, così come più in generale le istituzioni, non sono oggetti come gli altri oggetti materiali. Le istituzioni, in quanto appartenenti alla realtà sociale, hanno proprietà diverse. Non possono essere, ad esempio, manipolati dall'esterno (si può, invece, accendere una radio, riempire un bicchiere). Più precisamente, possono essere manipolate solo seguendo le regole da loro stesse specificate: le opportunità di interazione che sono offerte dall'istituzione stessa solo a chi gioca un ruolo in esse.

La metafora organizzativa va oltre al dominio delle istituzioni, e può essere usata in altri ambiti: se è utile, ogni oggetto può essere visto come una istituzione o una organizzazione strutturata in ruoli. In questo modo i ruoli possono essere usati per distinguere diversi modi di interazione con lo stesso oggetto che dipendono dal tipo di entità che vuole manipolare l'oggetto. In questo modo viene promossa la modularità del sistema, una proprietà necessaria in caso di applicazioni complesse.

Per fare un esempio, supponiamo di modellare un oggetto di tipo (o appartenente alla classe, per usare la terminologia OO) stampante (`Printer`) come una istituzione or-

ganizzata in due ruoli distinti che offrono diversi insiemi di operazioni (o, tecnicamente, metodi): un ruolo di utente normale (`User`) e uno di amministratore (`SuperUser`).

I due ruoli offrono delle operazioni (cioè sono dei contenitori di metodi come le classi). Alcuni di questi metodi sono comuni ai due ruoli, ma possono avere funzionamenti diversi, altri sono specifici di un ruolo. Non esiste tuttavia un modo di manipolare la stampante senza passare per questi ruoli. Chi gioca il ruolo di utente può mandare in stampa (cioè invocare il metodo `print` del ruolo) dei lavori fino ad un limite massimo di pagine stampate. Questo significa che per ogni ruolo di tipo utente si devono contare separatamente il numero di pagine stampate e mantenere tale conteggio fra una stampa e l'altra. Il numero di pagine stampate è un nuovo attributo che si aggiunge a quelli propri di chi sta giocando il ruolo di utente ed esiste solo fino a quando il ruolo viene mantenuto. Un ruolo, o meglio una istanza di ruolo associata a chi gioca un ruolo in una data istituzione, modella quindi lo stato dell'interazione fra chi ricopre il ruolo e l'istituzione. Anche il ruolo amministratore offre il metodo di stampa, ma con un funzionamento diverso: non ci sono limiti di pagine stampate. Inoltre, un amministratore può, ad esempio, cancellare i lavori in coda che attendono di essere stampati. Quindi, il ruolo amministratore non solo riesce ad accedere allo stato privato e alle operazioni dell'istituzione ma anche a quelli degli altri ruoli della stessa istituzione.

Un ruolo non è solo una classe qualunque di cui possono essere create delle istanze. Un'istanza di un ruolo può accedere ai valori e alle operazioni private di un'altra classe: l'istituzione a cui il ruolo appartiene. Ad es., un utente può stampare perché accede alle procedure private di stampa della stampante. Il metodo `print` che fa sì che la stampante stampi effettivamente è un metodo privato che non può essere quindi invocato direttamente. Solo attraverso la mediazione di un ruolo si può arrivare a stampare, con le peculiarità proprie del tipo di ruolo.

La possibilità offerta da un ruolo di accedere allo stato e operazioni private di un altro oggetto appartenente ad un'altra classe sembra violare un principio base della programmazione OO: lo stato di un oggetto è accessibile solo allo stesso oggetto. È possibile dare ai ruoli questa possibilità solo se chi definisce un tipo di ruolo è lo stesso programmatore che definisce la classe a cui il ruolo appartiene: ad es., la classe utente deve essere definita da chi definisce la classe stampante. In un linguaggio OO come Java questo fatto è espresso dall'inclusione di una classe nell'altra: la classe utente è definita dentro la classe stampante, o in termini tecnici, è una *inner class*.

Per giocare il ruolo di utente un oggetto deve avere un account (deve avere cioè le proprietà specificate da una interfaccia `Accounted`), il cui nome viene stampato sulle pagine che l'utente manda in stampa. Si noti che `Accounted` non è una classe di oggetti ma solo una specifica che può essere soddisfatta da diversi tipi di oggetti.

Infine, un ruolo quale utente può essere giocato solo nel momento in cui esistono altri due oggetti: una istanza della classe stampante e una istanza che rispetti il vincolo `Accounted`. Sono necessarie cioè una istanza della istituzione e una di chi gioca il ruolo. Solo dopo essere entrato nel ruolo utente un oggetto `Accounted` può stampare.

Questo esempio mostra le caratteristiche che hanno i ruoli nel nostro modello ontologico:

– *Fondazione*: un'istanza di ruolo deve essere sempre associata ad un'istanza dell'istituzione a cui appartiene, oltre ad essere associata ad un'istanza di chi gioca tale

ruolo.
- *Dipendenza definizionale*: la definizione di un ruolo dipende dalla definizione dell'istituzione a cui appartiene.
- *Poteri istituzionali*: le operazioni definite nel ruolo possono accedere allo stato dell'istituzione e degli altri ruoli: sono dei poteri istituzionali.
- *Prerequisiti*: per giocare un ruolo è necessario soddisfare dei requisiti, cioè offrire delle operazioni che sono utilizzate nella realizzazione delle operazioni del ruolo.

Molte di queste caratteristiche sono considerate anche da altri autori nel campo delle ontologie e della rappresentazione della conoscenza. Ad es., Masolo *e altri* (2004) considerano una nozione più debole di fondazione (il ruolo dipende da chi lo gioca) e di dipendenza definizionale (la definizione del ruolo menziona l'istituzione a cui appartiene).

Loebe (2005) considera ruoli che dipendono da contesti; Viganò e Colombetti (2006) considerano i poteri come elementi definitori dei ruoli.

Inoltre, come hanno notato Guarino e Welty (2002), a differenza dei tipi naturali, come persona, i ruoli mancano di rigidità: un oggetto può cominciare a giocare un ruolo e poi abbandonarlo senza perdere la sua identità. Al contrario una persona non può cessare di essere tale.

Infine, Steimann (2000) nota che un ruolo può essere giocato da tipi naturali differenti. Ad es., un ruolo come acquirente può essere giocato da entità di tipo persona o di tipo organizzazione, due classi che non sono correlate fra di loro e non sono sussunte da un tipo superiore. Il ruolo deve solo specificare le proprietà richieste a chi gioca il ruolo, e non un tipo. La nozione di prerequisiti rispecchia questa visione.

Quanto ha notato Steimann impedisce di modellare i ruoli come specializzazioni dinamiche di un tipo come proposta da Albano *e altri* (1993); Gottlob *e altri* (1996). Se acquirente fosse un sottotipo di persona non potrebbe essere allo stesso tempo un sottotipo di organizzazione dato che sono due tipi disgiunti.

La nostra definizione di ruolo, quindi, è indipendente dai linguaggi di programmazione ed è in gran parte condivisa con altri approcci in ambito ontologico.

3.2 powerJava

Baldoni *e altri* (2006b,c) introducono i ruoli accanto agli oggetti per estendere il linguaggio OO Java utilizzando la definizione discussa nella sezione precedente. Il nuovo linguaggio si chiama powerJava in quanto nella nostra definizione ontologica i ruoli sono dotati di poteri. Java è esteso con:

1. Un costrutto che definisce il ruolo, con il suo nome, i suoi prerequisiti e le operazioni che si possono invocare su di esso.
2. L'implementazione di un ruolo come una classe che ne rispecchia la definizione, che è inclusa nella definizione della classe che rappresenta l'istituzione.
3. Il modo in cui un oggetto gioca un ruolo e ne può invocare i poteri.

La Figura 3 mostra per mezzo dell'esempio discusso sopra l'uso dei ruoli in powerJava. Prima di tutto un ruolo è specificato come una sorta di interfaccia (`role` - a destra) che elenca i prerequisiti di chi può giocare quel ruolo (`playedby`) e quali operazioni

```
class Printer {
  private int printedTotal;

  private void print(String name)
    {...}


  definerole User {

    private int printed;

    public void print(){ ...
      printed = printed + pages;
      Printer.print(that.getName());
    }
  }
}
```

```
role User playedby Accounted
   { void print();
     int getPrinted(); }

interface Accounted
   { String getName();
     String getLogin();}

jack = new AccountedPerson();
laser1 = new Printer();
laser1.new User(jack);
laser1.new SuperUser(jack);
((laser1.User)jack).print();
```

Figura 3. A role `User` inside a `Printer`.

sono acquisite nel momento in cui si gioca il ruolo. Secondariamente (a sinistra) un ruolo è implementato all'interno di un oggetto come una sorta di classe interna (inner class) che realizza le specifiche della definizione di ruolo omonima (`definerole`). La classe interna implementa tutti i metodi richiesti nella specifica del ruolo come se implementasse un'interfaccia. Tali metodi possono fare riferimento all'oggetto che gioca il ruolo per mezzo della variabile speciale `that` che permette di invocarne i metodi specificati nei prerequisiti.

Al fondo della colonna di destra della figura, è mostrato l'uso dei ruoli in powerJava. Prima di tutto viene creato un oggetto che possa giocare il ruolo `jack`. Questo oggetto appartiene ad una classe che implementa i prerequisiti (`AccountedPerson` è una `Person` che implementa l'interfaccia `Accounted`. Prima che l'oggetto possa giocare il ruolo inoltre deve essere creata anche un'istanza della classe che rappresenta l'istituzione (la `Printer laser1`). Una volta che una `Printer` è creata l'oggetto `jack` può diventare uno `User`. Si noti che lo `User` è creato all'interno dell'oggetto `Printer laser1`(`laser1.new User(jack)`) e che `jack` è un argomento passato al costruttore del ruolo `User`, avendo il tipo richiesto `Accounted`. Inoltre `jack` gioca anche il ruolo di `SuperUser`.

Per poter giocare il ruolo di `User`, `jack` deve prima essere classificato come tale per mezzo di un cosiddetto *role casting* (`(laser1.User) jack`), dato che una `AccountedPerson` non ha il metodo `print`. Si noti che `jack` non viene classificato come un generico `User` ma come un utente della specifica istituzione `Printer laser1`. Una volta che `jack` ha assunto il ruolo `User` può esercitare i suoi poteri, in questo caso stampare (`print`).

Questo metodo è chiamato un potere dato che, in contrasto con i metodi standard, può accedere allo stato di altri oggetti. Nell'esempio il metodo `print()` accede allo stato privato di `Printer` e invoca `Printer.print()`.

4 Conclusioni

In questo articolo discutiamo la relazione che può esistere fra ontologie e linguaggi di programmazione. Ogni paradigma di linguaggi di programmazione riflette una diversa concettualizzazione, basata, ad es., sul concetto di macchina di Von Neumann, di funzione, oppure orientata agli oggetti.

Partendo dalle limitazioni dei linguaggi ad oggetti correnti, esaminiamo come il riferimento ad una analisi ontologica, cioè della specifica della concettualizzazione sottostante al linguaggio di programmazione possa essere usata per estendere un linguaggio di programmazione.

Come primo caso abbiamo riscontrato l'assenza di un costrutto esplicito per rappresentare relazioni in linguaggio OO. Abbiamo esaminato quali problemi porti tale mancanza, e mostrato come, senza dover introdurre nuovi costrutti, si possano simulare le relazioni per mezzo di una operazione ontologica di reificazione.

Infine, ci siamo focalizzati sul problema dei ruoli, una primitiva che dovrebbe essere presente in una ontologia accanto a classi e relazioni, ma che è quasi sempre trascurata. Per introdurre un costrutto per i ruoli in un linguaggio di programmazione abbiamo importato una definizione ontologica del costrutto di ruolo. In questo modo, si ha una definizione precisa di cosa sia un ruolo e congruente con molte altre proposte, e soprattutto indipendente dai problemi di programmazione che si vogliono risolvere.

La connessione esistente fra relazioni e ruoli deve essere ancora approfondita. Spesso i ruoli sono collegati ad una relazione e per questo motivo sono associati a coppie, quali acquirente/venditore, moglie/marito, ecc. D'altra parte quando si partecipa ad una relazione lo si fa attraverso un ruolo (si veda anche l'esempio della Sezione 2.2). Anche nel modello di basi di dati Entity-relationship gli argomenti di una relazione sono etichettati con ruoli.

Infine, in Baldoni *e altri* (2006a) abbiamo mostrato come la nozione di ruolo qui proposta sia spiegabile in termini di quelle che Gibson (1979) chiama *affordances*, nella sua teoria ecologica della percezione. Le *affordances* di un ambiente sono le azioni che esso offre ad una certa specie animale e dipendono dalle capacità di tale specie. Non si tratta di proprietà fisiche oggettive. Ad es. una superficie d'acqua può essere "camminabile" per un insetto ma non per altre specie animali.

In quest'ottica è possibile ripensare al concetto stesso di oggetto in OO, evidenziando come le proprietà di un oggetto non siano necessariamente oggettive ma possano dipendere dall'interazione con altre entità. Ad es., nella classe stampante non esistono operazioni che possano essere direttamente invocate sulla stampante senza la mediazione di un ruolo. I ruoli diventano in questo modo un mezzo per specificare insiemi di *affordances*, cioè di possibilità interazione con gli oggetti che dipendono dalla soddisfazione di certi requisiti.

Bibliografia

Albano A.; Bergamini R.; Ghelli G.; Orsini R. (1993). An object data model with roles. In *Procs. of Very Large DataBases (VLDB'93)*, pp. 39–51.

Baldoni M.; Boella G.; van der Torre L. W. N. (2006a). Modelling the interaction between objects: Roles as affordances. In *Procs. of Knowledge Science, Engineering and Management, KSEM'06*. A cura di Lang J., Lin F., Wang J., volume 4092 di *Lecture Notes in Computer Science*, pp. 42–54. Springer.

Baldoni M.; Boella G.; van der Torre L. (2006b). Powerjava: ontologically founded roles in object oriented programming language. In *Procs. of OOPS Track of ACM SAC'06*, pp. 1414–1418. ACM.

Baldoni M.; Boella G.; van der Torre L. (2006c). Roles as a coordination construct: Introducing powerJava. *Electronic Notes in Theoretical Computer Science*, **150**(1), 9–29.

Boella G.; van der Torre L. (2004). An agent oriented ontology of social reality. In *Procs. of FOIS'04*, pp. 199–209, Amsterdam. IOS Press.

Boella G.; van der Torre L. (2005). The ontological properties of social roles: Definitional dependence, powers and roles playing roles. In *Procs. of LOAIT workshop at ICAIL'05*.

Booch G. (1988). *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, Reading (MA).

Gamma E.; Helm R.; Johnson R.; Vlissides J. (1995). *Design Patterns: Elements of Reusable Software*. Addison-Wesley.

Gibson J. (1979). *The Ecological Approach to Visual Perception*. Lawrence Erlabum Associates, New Jersey.

Gottlob G.; Schrefl M.; Rock B. (1996). Extending object-oriented systems with roles. *ACM Transactions on Information Systems*, **14(3)**, 268 – 296.

Gruber T. (1993). A translation approach to portable ontology specifications. *Knowledge Acquisition*, **5**(2), 199–220.

Guarino N.; Welty C. (2002). Evaluating ontological decisions with ontoclean. *Communications of ACM*, **45(2)**, 61–65.

Herrmann S. (2002). Object teams: Improving modularity for crosscutting collaborations. In *Procs. of Net.ObjectDays*.

Kristensen B.; Osterbye K. (1996). Roles: conceptual abstraction theory and practical language issues. *Theory and Practice of Object Systems*, **2**(3), 143–160.

Loebe F. (2005). Abstract vs. social roles - a refined top-level ontological analysis. In *Procs. of AAAI Fall Symposium Roles'05*, pp. 93–100. AAAI Press.

Masolo C.; Vieu L.; Bottazzi E.; Catenacci C.; Ferrario R.; Gangemi A.; Guarino N. (2004). Social roles and their descriptions. In *Procs. of Conference on the Principles of Knowledge Representation and Reasoning (KR'04)*, pp. 267–277. AAAI Press.

Noble J. (1997). Basic relationship patterns. In *Procs. of EuroPLOP*.

Papazoglou M.; Kramer B. (1997). A database model for object dynamics. *The Very Large DataBases Journal*, **6(2)**, 73–96.

Rumbaugh J. (1987). Relations as semantic constructs in an object-oriented language. In *Procs. of the OOPSLA-87: Conference on Object-Oriented Programming Systems*, pp. 466–481, Languages and Applications, Orlando, FL.

Rumbaugh J.; Jacobson I.; Booch G. (2004). *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education.

Steimann F. (2000). On the representation of roles in object-oriented and conceptual modelling. *Data and Knowledge Engineering*, **35**, 83–848.

Viganò F.; Colombetti M. (2006). Specification and verification of institutions through status functions. In *COIN@AAMAS'06 Workshop*.