Design patterns multiagents driven for software development

Arsene O. [*] **Dumitrache I.** [*]

[*] *Laboratory of Intelligent Systems, "Politehnica" University of Bucharest, SPl. Independentei 313, 060042 Bucharest, Romania (e-mail: octavianarsene@gmail.com)*

Abstract: The complexity of nowadays systems imposes development of a dynamic architecture based on modularization, dependency, processing and data access principles. The field of software design and development is rich in patterns. Software patterns can be seen as a library of code modules that is reused during development of programs. The paper propose a software pattern multiagent systems driven. There will be taken four main patterns: singleton, transfer object, abstract factory and data access object and will be implemented as software agents. Patterns have always participants who are related through the pattern structure. This leads to the observation that different patterns can be created by varying type and features of the participants. An entire family of patterns can be built from one pattern; thus , there will be many smaller building blocks, cooperating and computing toward one goal. This is the trigger for using software agents as blocks. Each project functionality will be decomposed into many tasks, which will be completed by software agents. Each task will be solved by software agents acting as software pattern blocks. The agents (software pattern implementation) will be automatically generated based on the specificity of the task. The implementation is done using JADE, a mature software agent technology platform. The paper is focused on the automation of the task - software pattern(s) mapping and implementation.

Keywords: Software architecture, MultiAgent Systems, Software patterns

1. INTRODUCTION

The challenges of developing high-performance, high-reliability and high-quality software systems are too much for ad hoc and informal engineering techniques that might have worked in the past on less demanding systems. Designing systems is a hard problem. Reusing existing designs (in whole or in a part) helps the architect of the system when a new solution should be created. Design patterns are a recent software engineering problem-solving discipline that emerged from the object-oriented community. A design pattern is an abstract solution to a problem [Kaisler, 2005]. The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.

Object Management Group (OMG) launched in 2001 a new initiative called Model Driven Architecture (MDA). The essence of MDA is that the creation of an executable software architecture should be driven by the formulation of models rather than by manually writing source code. Source code is generated from the models by a compilation step much as machine code is generated from source code. The MDA initiative aims to move software development to a higher level of abstraction [Arlow and Neustadt, 2003]. MDA proposes an alternative model driven approach to software development in which source code is built from Unified Modeling Language (UML) models. The value of an UML model increases with its level of abstraction. The MDA separates concerns, thus the business functionality (without technical details, is represented in UML model diagrams) is represented in the Platform Independent Model (PIM) and the platform aspects are represented in the Platform Specific Model (PSM), customized for a particular technology, for example, Java Platform. This makes the PIM portable on many different platforms, provided that there is a suitable PIM-to-PSM mapping and a PSM-to-code compiler for the target platform.

The practical way to deal with MDA is to start from design specifications and going forward to software implementation. The business analysts will create UML diagrams first (e.g. use cases, domain problem concepts encoding into classes, state and sequence diagrams), then the MDA-enabled modeling tool will generate the code according to the platform technology.

The present paper propose another approach for software creation, there are already databases, web-services providing different levels of business knowledge. The idea is to leverage these systems in order to extract the knowledge from them, exposing it as ontologies and based on these to be able to create new business specifications and particular software agents. Different software design pattern are implemented by agents (e.g. singleton, transfer object, abstract factory and data access object) as working units. Each task is mapped to one or more agents. The agents assignment to a specific task is done automatically due to the usage of the same concepts strings (taken from the ontology) as trigger attribute of the software pattern and in the description of that task. If the *database* string is part of the task enunciation then the software patterns having the same trigger attribute value will be instantiated and linked to the task. The agents cooperate among them in order to fulfill the task. The task is created based on the ontology of the domain, the concepts and the relations between them will be discovered by the ontology agents. There will be considered two main data sources for domain knowledge: relational databases and web-services. The model for the ontology extraction from the relational databases is based on its metadata information, the ontology agents discover the tables, views, relations between entities according to all three normalized forms. The Java driver vendors let users know the capabilities of a Database Management System (DBMS) in combination with the driver based on Java Database Connectivity (JDBC) technology that is used with it. The Web Service Definition Language (WSDL) file is used for ontology creation for a specific web-service source [Hui et al., 2007]. The WSDL file contains the data types definitions and the exposed methods [Newcomer, 2002], which will be mapped to the ontology concepts and relationships.

Section 2 presents briefly the software patterns, ontology and software agents concepts used in the proposed synergy. Section 4 describes the automatic creation process of the software patterns based on the ontology and software agents. Section 4 covers the software architecture of the prototype used to implement the concepts presented in the section 4. The last section presents the conclusions and directions for the further research.

2. MAIN CONCEPTS

2.1 Software patterns

Design patterns are a recent software engineering problem-solving discipline that emerged from the object-oriented community. The separation of the technical infrastructure and business logic requires the introduction of software patterns in order to avoid shortcomings of enterprise development platforms [Bien, 2009]. Design patterns are models of partial solutions [Kaisler, 2005]. The class is the object oriented implementation of a real concept. A design pattern is defined as a collection of abstract classes, which are specialized relative to a particular problem. Design patterns are not just designs. A pattern must be instantiated, an object is a run-time class instantiation. This means the programmer writes code and makes evaluations and trade-offs about how to perform certain operations.

Design patterns vary in their granularity and level of abstraction. Design patterns are classified by two criteria [Gamma et al., 1995]:

(1) *purpose*, reflects what a pattern does. Patterns can have one of the following purposes:
    - *creational*, concerns the process of object creation;
    - *structural*, deals with the composition of classes or objects;
    - *behavioral*, characterize the ways in which classes or objects interact and distribute responsibility.
(2) *scope*, specifies whether the pattern applies to classes or objects.

Java Platform Enterprise Edition (Java EE), which provides a unified platform for developing distributed, server-centric applications. Java EE establishes standards for areas of enterprise computing needs such as database connectivity, enterprise business components, message-oriented middleware (MOM), web-related components, communication protocols, and interoperability. The complexity of the server solutions lead to the raising of Java EE patterns as a new category. They hover somewhere between a design pattern and an architectural pattern, while the strategies document portions of each pattern at a lower level of abstraction [Alur et al., 2003]. The classification of Java EE patterns is based on the following three logical architectural tiers: (1) presentation tier, (2) business tier and (3) integration tier.

The scope of the paper covers only the following software patterns:

- *Data Access Object* (DAO), which is a Java EE pattern ([Alur et al., 2003]); it provides access to data varies depending on the source of the data. Access to persistent storage, such as to a database, varies greatly depending on the type of storage (relational databases, object-oriented databases, flat files, and so forth) and the vendor implementation.
- *Singleton* (creational pattern, [Gamma et al., 1995, 2005], ensures a class only has one instance, provides also a global point of access to it.
- *Abstract Factory* (creational pattern, [Gamma et al., 1995, 2005], provides an interface for creating categories of related objects without specifying their concrete classes; it defers the creation of real objects to its concrete factory subclasses. An application typically needs only one concrete factory per object family, thus the factories are implemented as singletons.
- *Transfer Object* (Java EE pattern), is designed to optimize data transfer across application tiers; instead of sending or receiving individual data elements, a Transfer Object contains all the data elements in a single structure required by the request or response.

The relations between chosen software pattern are the following:

- DAO pattern can be made highly flexible by adopting the Abstract Factory. The underlying storage is subject to change from one implementation to another (e.g. Oracle, MySQL);
- AbstractFactory is implemented as a Singleton, an application typically needs only one instance per product family (server provider);
- AbstractFactory uses Transfer Objects to transport data to and from its clients.

The complete catalogs for design patterns can be found in [Gamma et al., 1995, Alur et al., 2003]. The term software pattern used in the paper is referring to design pattern.

2.2 Ontology

An ontology is a formal, explicit specification of a shared conceptualisation. A *conceptualisation* refers to an ab-

stract model of some phenomenon in the world which identifies the relevant concepts of that phenomenon [Gruber, 1993], [Fensel, 2001]. Ontologies were developed in AI to facilitate knowledge sharing and reuse. Ontologies are developed to provide a machine-processable semantics of information sources that can be communicated between different agents (software and humans).
The identification of the key concepts is the first step of the ontology creation process. The class is the formal way to represent a concept. Different ontology languages provide different facilities. The most recent development in standard ontology languages is OWL, endorsed by the World Wide Web Consortium (W3C) to promote the Semantic Web vision. The PROTEGE framework is going to be used for creating, maintaining the domain ontology. PROTEGE is a free, open source ontology editor and knowledge-base framework. The PROTEGE-OWL editor is an extension of PROTEGE that supports the Web Ontology Language (OWL) [Horridge et al., 2004]. The OWL ontology may include descriptions of classes, properties and their instances. The OWL-DL language is used due to its expressivity and computational efficiency.
The real life entities from the domain are encoded as classes instantiations within an ontology, called individuals. Each individual has a clear identity that makes it different than others, even they have common attributes.
An ontology can be used in two ways: (1) an XML file OWL compliant into an ontology repository (file system or database) or (2) as object tree in memory of the application.

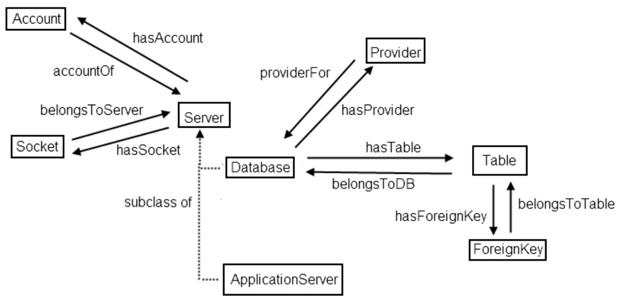

Fig. 1. IT systems ontology

The system ontology used in the proposed framework is depicted in the Fig. 1. The following classes were defined:

- *Server*, is the root class for any server. It has the two relations: *hasSocket* with Socket and *hasAccount* with Account class;
- *Database*, is the database server class. It has two new attributes: *connectionURL*, *dbInstance*. The connectionURL is a driver specific string used for connection to a database. The dbInstance is the instance name we want to connect to. There are two relations: *hasProvider* with Provider and *hasTable* Table and other two inherited from Server class;
- *ApplicationServer*, is the application server class. The web-services applications are deployed here;
- *Socket*, identify the server in the application in the network based on two attributes: *ip* address and *port*. There is one relation defined: *belongsToServer* with Server;
- *Account*, *username* and *password* used for connection to a server;
- *Provider*, identify the driver name offered by each specific database vendor, it gives out the connection to the database and implements the protocol for transferring the query and result between client and database;
- *Table*, database table having three attributes: *name*, *column*, *primaryKey* and two relations: *hasForeignKey* with ForeignKey and *belongsToDB* with Database class.
- *ForeignKey*, is the foreign key of a table; it has three attributes: *name*, *referencedTable*, *column* and one relation ForeignKey *belongsToTable* with Table.

The real IT servers are encoded as individuals (instances) of the classes: Socket, Account, Provider, Database, they are provided in order to initialize the software prototype. The following values are individuals samples:

- SOCKET_DB_001 (name of the individual): *ip*=10.0.0.1, *port*= 1500;
- ACCOUNT_001: *username*= user1, *password*= test1, *accountOf* DB_001 individual;
- ORACLE (Provider individual): *name*= Oracle, *driver*= oracle.jdbc.Driver, *providerFor* DB_001;
- DB_001 (Database individual): *connectionURL*= $jdbc : oracle : thin$: @, *dbInstance*= instance1, *hasAccount* ACCOUNT_001, *hasProvider* ORACLE, *hasSocket* SOCKET_DB_001.

During the initialization phase of the software prototype the *Table* individuals and their attributes and relations are not discovered yet. This is done later by the ontology agent based on already provisioned *Database* individuals. This allows a dynamically discovery and an automated provisioning with new *Table* individuals.

The design patterns ontology has the following structure:

- DesignPattern, as root class for any pattern;
- AbstractFactory, has four attributes: *Database*, *Socket*, *Account* and *Table*;
- DAO, has an attribute: *Provider*;
- TransferObject, used for returning of the response values. It does not have attributes defined into ontology, it is a dynamic data structure based on a column name - values map.

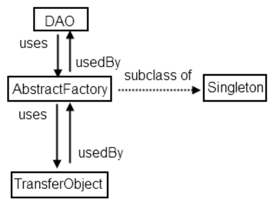

Fig. 2. Design patterns ontology

There is one relation defined between design pattern's classes: DesignPattern *uses* DesignPattern. The agents acting as wrappers over the patterns are linked based on the relations defined within the design patterns ontology.

The first relation: DesignPattern1 *uses* DesignPattern2, triggers the creation of the second pattern. The first agent aggregates the reference to the second one, thus the agent will always know who should it call to in order to accomplish the task. The relations between four software patterns is depicted in the Fig. 2.

AbstractFactory *is a subclass* of Singleton and *uses* TransferObject for response values. The subclass relation implies composition between Singleton and AbstractFactory, they will have the same lifetime. DAO *uses* AbstractFactory, there is more than one implementation at factory level: *OracleFactoryGen* and *MySQLFactoryGen*. Oracle and MySQL are two different database providers. Both instances provide same behaviour to the client pattern by implementing the same interface. In case of database product this interface defines basic methods mapped to SQL manipulation functions: select, insert, update, delete.

2.3 Software agents

Agent-Oriented Programming (AOP) is a relatively new software paradigm that brings concepts from the theories of Artificial Intelligence (AI) into distributed systems research field. AOP models an application as a collection of components called agents characterized by autonomy, reactivity, proactivity, social ability [Bellifemine et al., 2007], [Wooldridge, 2009].

In the proposed framework is used Java Agent DEvelopment (JADE) API as agent-oriented middleware [Bellifemine et al., 2007]. Agent communication is implemented according to the Foundation for intelligent, physical agents (FIPA). There are two agent categories implemented: ontology and software pattern. All agents are registered into the Director Facilitator (DF) *Yellow page* JADE service, allowing publishing of the provided services. The agents can easily discover others by sending a search request to the DF. Once an agent finds the partner agent the services provided by the latter, in terms of methods, can be accessed by the first one.

The ontology agent is responsible for gathering data sources structural information. In case of a database source, the agent extracts the database entities names, column's tables, primary keys, foreign keys. The agent uses JDBC technology in order to connect to the database and to extract the necessary data.

The software pattern instances are implemented as software agents; the agent societies are defined for each pattern type: *DAO*, *Abstract Factory*, *TransferObject*. A software agent instance can be reused. The AbstractFactory software pattern is implemented as a Singleton, the society will be in charge of the factory single instance creation. The instance is registered into the Yellow Page JADE service and can be found by the DAO agents in order to start server session for solving database tasks. In case the Singleton agent stops unexpectedly due to an error, another instance will be created and an error message will be sent to the agent administration board (the people group in charge of the agent framework errors analysis, it is implemented as a group email address).

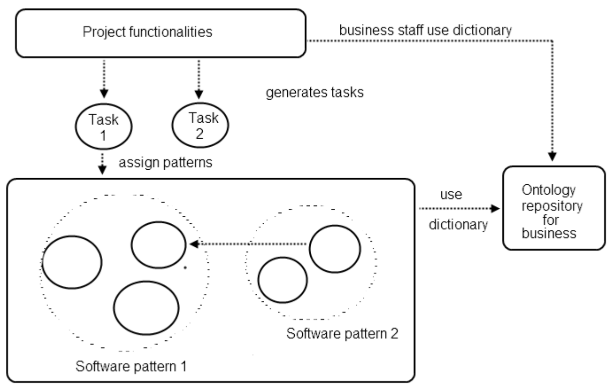

Fig. 3. Software pattern generation flow

3. PROCESS MODEL

The idea of the automatic creation of the software patterns fits into the following process (see Fig. 3):

(1) Business team build the project(s) following the current vision of the company. Based on the identified business needs raised from the company vision, the upper management team will be able to launch few information technology (IT) projects;
(2) The IT project is decomposed by business analysts into several functionalities. The analysts team will use concepts from an ontology based dictionary during the translation phase of the project into functionalities. The ontology encodes the hardware and software components of the company IT configurations (e.g. specific databases, application servers, protocols, data model). The same dictionary will be used later on by the agent framework to build software patterns corresponding to the task;
(3) Each functionality is decomposed into at least one task by the business analysts;
(4) Every task is automatically mapped to the corresponding software pattern(s). For example, the task TASK-001: "Get *customer information* from the *database* DB-001." is assigned to the following software patterns:
   - *Data Access Object*
   - *Singleton*
   - *Abstract Factory*
   - *Transfer Object*
(5) The software patterns are instantiated as software agents. Each software pattern has a special attribute: *trigger*, which represents the list of concepts (e.g. database, web-service) from the ontology. The task description will contain at least one trigger concept, thus the connection between task and software pattern is based on these strings;
(6) The semantic query expressed by the task is translated automatically by the instantiated agent into specific product query. If the product is database then the query is Structured Query Language (SQL) based, in case of a web-service product the query is Simple Object Access Protocol (SOAP) based.
(7) The query is run against selected data source. The *database* DB-001 is an ontology individual having the following attributes: Vendor, Socket, Account.

They are individuals as well and provide information related to the connection and identification of the data sources discovered in the organization.

(8) The response is sent to the task by the triggered software pattern. In case of TASK-001, the response will be wrapped into a Customer transfer object having all details (e.g. first and last name, location, company).

4. SOFTWARE ARCHITECTURE

The application architecture is based on the following components (4):

- Java Platform Standard Edition (Java SE)
- ontology services module
- translation module
- agent container

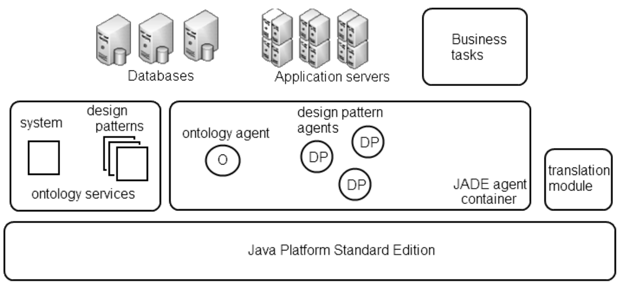

Fig. 4. Application architecture

Java SE portability allows to deploy application modules on different operating systems. The ontology services and agent container are built in top of the Java SE software development kit (SDK). PROTEGE API is used for integration of the RDF-OWL ontology files with agent framework and with business tasks module. The JADE API is used for agent container implementation.
The ontology agent is started first and checks all databases provisioned in the ontology file. The ontology database individuals related to existing entities: tables, columns, constraints, will be created as well by the ontology agent after the checking stage is done. If a new table is discovered then a new individual will be automatically created into the system ontology and linked to the parent server. The ontology agent is implemented as *CyclicBehaviour* according to JADE behaviour API. That means the agent will check after new entities every 30 minutes. The updated ontology is saved automatically in order to persist the new discovered entities.
Once the task is launched the software pattern agents are created based on the trigger mechanism explained in subsection 2.2. Each type of pattern agent is created by its specialized agent society at task module or other agents request. Each task is transform by translation module in data source language, for databases in SQL and in SOAP messages for web-services. For tasks involving Oracle database (e.g. TASK-001 from section) the following pattern agents are created:

- DAOGen, is the DAO agent;
- OracleFactoryGen, is the AbstractFactory agent for Oracle database. For MySQL database will be created MySQLFactoryGen agent based on the driver information stored in the ontology individual Provider;
- TransferObjectGen, is the TransferObject agent used by OracleFactoryGen for sending results to the client task module.

After the task request is completed the agents are not destroyed, they are reused by future requests.

5. STUDY CASE

The proposed framework has been tested using an Oracle database and a schema having seven tables. The ontology agent creates the following individuals within the ontology:

- Socket individual: SOCKET_DB_001, with the attributes: *ip*=10.0.0.1, *port*= 1521;
- Account individual: ACCOUNT_001, having the following attributes: *username*= hr, *password*= welcome1. ACCOUNT_001 has the *accountOf* relation with DB_001 individual;
- Provider individual: ORACLE, with the following attributes: *name*= Oracle, *driver*= oracle.jdbc.Driver. ORACLE has the *providerFor* relation with DB_001 individual;
- Database individual: DB_001, having the attributes: *connectionURL*= *jdbc* : *oracle* : *thin* : @, *dbInstance*= XEit has the following relations: *hasAccount* with ACCOUNT_001, *hasProvider* with ORACLE and *hasSocket* with SOCKET_DB_001;
- Table individuals: REGION, COUNTRIES, LOCATIONS, DEPARTMENTS, EMPLOYEES, JOBS, JOB_HISTORY, all these have the *belongsToDB* relation with DB_001 individual;

There were created ForeignKey individuals and corresponding relations as well:

- COUNTR_REG_FK *belongsToTable* COUNTRIES, it is the foreign key (FK) of the COUNTRIES table referencing REGION table primary key (PK);
- LOC_C_ID_FK *belongsToTable* LOCATIONS, it is the FK of the LOCATIONS table referencing COUNTRIES PK;
- DEPT_LOC_FK *belongsToTable* DEPARTMENTS, it is the FK of the DEPARTMENTS table referencing LOCATIONS PK;
- DEPT_MGR_FK *belongsToTable* DEPARTMENTS, it is the FK of the DEPARTMENTS table referencing EMPLOYEES PK;
- EMP_DEPT_FK *belongsToTable* EMPLOYEES, it is the FK of the EMPLOYEES table referencing DEPARTMENTS PK;
- EMP_JOB_FK *belongsToTable* EMPLOYEES, it is the FK of the EMPLOYEES table referencing JOBS PK;
- EMP_MANAGER_FK *belongsToTable* EMPLOYEES, it is the FK of the EMPLOYEES table referencing EMPLOYEES PK;
- JHIST_DEPT_FK *belongsToTable* JOB_HISTORY, it is the FK of the JOB_HISTORY table referencing DEPARTMENTS PK;

- JHIST_EMP_FK *belongsToTable* JOB_HISTORY, it is the FK of the JOB_HISTORY table referencing EMPLOYEES PK;
- JHIST_JOB_FK *belongsToTable* JOB_HISTORY, it is the FK of the JOB_HISTORY table referencing JOBS PK;

Different tasks will be created based on the above ontology, like the following:

- TASK-001: GET ALL EMPLOYEES FROM DB_001;
- TASK-002: GET ALL EMPLOYEES WITHIN DEPARTMENT_ID 10 FROM DB_001;
- TASK-003: GET ALL JOB_HISTORY FOR DEPARTMENT_ID 20 FROM DB_001.

They will be displayed in a graphical interface as well in order to be accessed by the business team.

The translation module will translate semantic queries into SQL queries that will trigger the corresponding pattern agents. In our case there are instantiated the following pattern agents that will handle the database queries and results retrieval: *DAOGen* (DAO agent), *OracleFactoryGen* (abstract factory for Oracle database) and *TransferObject*. In case of TASK-001, TransferObject agent is a list of *name-value* maps like:

- MAP1: EMPLOYEE_ID=100, FIRST_NAME=Steven, LAST_NAME=King, EMAIL= SKING;
- MAP2: EMPLOYEE_ID=101, FIRST_NAME=Neena, LAST_NAME=Kochhar, EMAIL= NKOCHHAR.

6. CONCLUSION

The purpose of this paper was to explore the automation of the software development using design patterns, ontologies and agents. The main advantage of the proposed framework is the adaptation of the task's vocabulary and system ontology after new server entities are discovered without manual reconfiguration and bouncing the entire system. Another advantage is the flexibility, more patterns can be added in order to enlarge the problem domain. The scope of this prototype was the proof of concept, applied on database for getting data functionality.
The further research involves the following steps: extend to other database functionalities, include application server case in the problem domain, increase the design pattern portfolio. The possibility to add new servers dynamic and automatically by another discoverer agent would be challenging as well.

REFERENCES

D. Alur, J. Crupi, and D. Malks. *Core J2EE Patterns: Best Practices and Design Strategies.* Prentice Hall PTR, Palo Alto, California, USA, 2003.

J. Arlow and I. Neustadt. *Enterprise Patterns and MDA: Building Better Software with Archetype Patterns and UML.* Addison Wesley, 2003.

F. Bellifemine, G. Caire, and D. Greenwood. *Developing Multi-Agents Systems with JADE.* John Wiley & Sons Ltd, England, 2007.

A. Bien. *Real World Java EE Patterns Rethinking Best Practices.* press.adam-bien.com, 2009.

D. Fensel. *Ontologies: A Silver Bullet for Knowledge Management and Electronic Commerce.* Springer, Berlin, 2001.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software with Applying UML and Patterns: an Introduction to Object-Oriented Analysis and Design and Iterative Development.* Addison-Wesley, 2005.

T. R. Gruber. A translation approach to portable ontology specification. *Knowledge Acquisition*, 5:199–220, 1993.

M. Horridge, H. Knublauch, A. Rector, R. Stevens, and C. Wroe. *A Practical Guide To Building OWL Ontologies Using The Protege-OWL Plugin and CO-ODE Tools Edition 1.0.* The University Of Manchester, 2004.

Guo Hui, A. Ivan, R. Akkiraju, and R. Goodwin. Learning ontologies to improve the quality of automatic web service matching. pages 118–125, July 2007.

S.H. Kaisler. *Software Paradigms.* John Wiley and Sons, Inc., Hoboken, New Jersey, 2005.

E. Newcomer. *Understanding Web Services: XML, WSDL, SOAP, and UDDI.* Addison-Wesley, 2002.

M. Wooldridge. *An Introduction to MultiAgent Systems.* John Wiley & Sons Ltd, West Sussex, UK, 2009.