ENHANCING LEGACY SYSTEMS WITH AGENT TECHNOLOGY

MAgIL FRAMEWORK

THESIS

presented to the Faculty of Economics and Social Sciences
at the University of Fribourg (Switzerland)
in fulfillment of the requirements for the degree of
Doctor of Economics and Social Sciences

by

MINH TUAN NGUYEN

from DongNai, Vietnam

Accepted by the Faculty of Economics and Social Sciences on May 30th, 2012
at the proposal of
Prof. Dr. Jacques Pasquier–Rocha (First Advisor) and
Prof. Dr. Marino Widmer (Second Advisor)

Fribourg, Switzerland
2012

To my parents, sisters and brothers NGUYEN

Acknowledgements

First of all, I would like to express my sincere gratitude to Prof. Dr. Jacques Pasquier-Rocha, my first advisor, for giving me the opportunity to perform a PhD research in his Software Engineering Group at the Department of Informatics, University of Fribourg. His continuous support of my research project, his patient guidance and encouragement allow me to accomplish this thesis.

I would like to thank Prof. Dr. Marino Widmer for his evaluation of my thesis as a second advisor.

Many thanks go to the members of the research team, in particular Dr. Patrik Fuhrer for our discussions and his advices along this research to help me clarify many issues in developing the MAgIL framework. I wish to thank in addition Andreas Ruppen, Benoît Pointet, Joseph Schaeppi, and Joël Vogt for their collaboration and useful feedbacks which contribute to the successful completion of this PhD thesis.

The MediMAS case study cannot be achieved without the contribution of the laboratory's team of the Hospital of Fribourg (HFR), Switzerland. Therefore, I would like to thank Dr. Jean-Luc Magnin, PhD, head and doyen of the HFR Laboratory department, for his precious time to explain to our research team the organization of the laboratory and its workflows of data and processes; Dr. Benoît Fellay, PhD, for his valuable advice, and the laboratory personnel for the live demonstration of the workflows in the real world.

My sincere thanks go to the family of Roland and Denise Devaud whose hospitality and continuous support allow my integration during my stay in Fribourg and the achievement of my study at the University of Fribourg. Thanks also go to my friends for creating around me a favorable environment which motivates me and allows me to optimize my study and research.

I am deeply indebted to Reverend Father Joseph Duc Khoan Nguyen who has supported and encouraged me in both academic and spiritual life, without which I would not have the chance to pursue my study and achieve my dream.

I am grateful to my parents, my sisters and brothers for their sacrifices. In our loving rural family with many children, you are working hard in Vietnam to make my academic dream come true at the University of Fribourg.

Abstract

Today's complex organizations such as hospitals, banks, airline companies, rely on sophisticated information systems to process large volume of data necessary to conduct their daily business. These information systems often inherit many weaknesses from the past during their life cycle. This is due to the fact that the organizations must preserve massive past investment in what we call legacy information systems.

This thesis analyzes the weaknesses and problems of legacy information systems, and proposes a solution using the agent technology. The proposed agent-based approach focuses on a multi-agent subsystem whose objective is to enhance and integrate different legacy information systems within and between organizations. A multi-layered architecture of an enhanced legacy information system is designed, in which the multi-agent subsystem is called Multi-Agent Integration Layer (MAgIL).

In order to unify the implementation of multi-agent subsystems, a MAgIL framework and a development process are defined. Finally, a case study in the healthcare domain is undertaken to validate the MAgIL approach and its framework.

Table of Contents

List of Figures

List of Tables

List of Program Excepts

Abbreviations and Acronyms

AID	Agent IDentifier
AMS	Agent Management System
BDI	Beliefs-Desires-Intentions
DF	Directory Facilitator
DOM	Distributed Object Management
FIPA	Foundation for Intelligent Physical Agents
FIPA ACL	FIPA Agent Communication Language
FIPA SL	FIPA Semantic Language
HFR	Hospital of Fribourg
HTML	HyperText Markup Language
JADE	Java Agent DEvelopemt Framework
JESS	the Java Expert System Shell, the Rule Engine for the Java Platform
J2JToolKit	Jess to JADE Tool Kit
KIF	Knowledge Interchange Format
KQML	Knowledge Query and Manipulation Language
LAN	Local Area Network
LIS	Legacy Information System
MAgIL	Multi-Agent Integration Layer
MAS	Multi-Agent System
MediMAS	Medical Multi-Agent System
MTS	Message Transport Service
OMG	Object Management Group
PDA	Personal Digital Assistant
PID	Personal IDentifier
SimToolKit	Simulation Tool Kit
UML	Unified Modeling Language
uniMAS	University Community's Multi-Agent System
WWW	World Wide Web
XML	eXtensible Markup Language
XSD	XML Schema Definition

1 Introduction

This introductory chapter presents the motivation and goals of the thesis, and its structure.

1.1 Motivation and Goals

The business of today's complex organizations relies on sophisticated information systems which often inherit many weaknesses from the past during their life cycle. For instance, due to their lack of flexibility, many information systems cannot integrate efficiently the ever-increasing requirements in order to assist the users or to free them from routine tasks. This is only one of many problems related to the so-called automation gap.

Another severe weakness relates to the increasing mobility of users. Many legacy information systems are designed for users working at fixed client workstations in fixed offices. They do not take into account recent advances in mobile technology such as smart phones, mobile phones.

In many information systems, the information flow from person to person may present another weakness since it still requires inefficient human interactions for example to write and send an email manually, to inform a colleague by phone, to write and send faxes, etc.

This research aims at proposing a solution to overcome these weaknesses, to enhance and integrate different legacy information systems within and between organizations. The solution is based on the software agent technology. It is a multi agent-based system, in which the key concept "assistant" plays an important role, for example:

- end-user assistants, also called personal assistants: to help the end-users search, retrieve, and filter efficiently the information from legacy information systems;

- legacy information system assistants: to help legacy information systems respond to database queries;
- manager of information system assistants: to help manager of information systems control the information flow between systems.

This thesis makes several contributions to promote the multi-agent system approach and its applications:

- The concept of software assistant agent will be treated in depth by examining its characteristics and capabilities in supporting human actors and legacy information systems.
- A layered, tiered multi-agent subsystem architecture will be proposed, in which a special layer called *Multi-Agent Integration Layer* (MAgIL) consists of software assistant agents to enhance legacy information systems. Enhancing legacy information systems means:
    - Standardizing the data communication between them,
    - Integrating the ever-increasing requirements of users into systems,
    - Allowing users work more efficiently with those systems.
- A *MAgIL framework* for developing the *integration layer* will be defined. The framework facilitates the implementation of concrete agent-based subsystems and reduces the time and costs required by the implementation process.
- A development process will be proposed to software designers in order to identify and analyze the new requirements at both user and organization levels, and to extend or upgrade a legacy information system which must fulfill those requirements. Thus, massive past investment will be preserved.
- A case study in the healthcare domain will be undertaken to validate the MAgIL approach and its framework. A concrete agent-based subsystem, called MediMAS subsystem, will be designed and implemented.

1.2 Structure of the Thesis

The dissertation is divided into eight chapters. After this introductory chapter, Chapter 2 discusses the limitations and problems of *Legacy Information Systems* in organizations.

Chapter 3 presents the promising *Agent Technology*, used to develop modern information systems, called agent-based information systems.

Chapter 4, *Agent-Based Approach for Legacy Information Systems*, applies agent technology:

- firstly, to design a seamless way to enhance legacy information systems;
- secondly, to design and implement the personal assistants that helps end-users in exploiting legacy information systems.

To reach the above two objectives, the concept of Multi-Agent Integration Layer (MAgIL) will be introduced.

Chapter 5, *The MAgIL Framework: Bring It to Life with A Concrete Subsystem*, aims at defining a framework for implementing agent-based subsystems. This chapter exposes the architecture of the MAgIL framework with its building blocks. A simple case study of agent-based subsystems will be discussed in details.

Chapter 6 presents the *Design and Implementation* of the MAgIL framework.

Chapter 7, *The MediMAS Subsystem: A Case Study*, presents a field project in which the MAgIL framework is used to build a concrete application in healthcare domain at the laboratory of the Hospital of Fribourg, Switzerland.

Finally, Chapter 8 concludes this thesis by highlighting the main achievements and proposing possible research directions in the future.

1.3 Notation and Conventions

The following conventions are adopted throughout this dissertation:

- Formatting conventions:
  Italic is used for keywords and definitions.
  Bold and *Italic* (***Bold***) is used to highlight important keywords.
  `Courier New` is used for `URL`, web addresses, and program listing.
- The citations and references comply with the *APA* style.

2

Legacy Information Systems

This chapter aims at presenting the general structure and identifying the weaknesses and issues of legacy information systems in organizations.

2.1 Definition

Today, big and complex organizations such as hospitals, banks, airline companies, still maintain legacy information systems in which they have consented massive investment in the past. Those legacy information systems play an important role in processing information necessary to conduct their daily business. In order to understand the term “legacy information system”, let’s define first what an information system is.

According to Morley and Parker (2010), an information system is a collection of elements (people, hardware, software, and data) and procedures that interact to generate information needed by the users in an organization. Information systems manage and process data from the time it is generated (such as data resulting from orders, documents, and other business transactions) through its conversion into information.

We can now define that *a legacy information system is an aged information system requiring massive investment during its long life cycle. Its hardware and software technologies are now obsolete; some are even no longer supported by the vendors*. Therefore, it usually resists modification and evolution to meet new and constantly changing business requirements as stated by (Stonebraker & Brodie, 1995).

2.2 Structure of Legacy Information Systems

From the above definition the general structure of a legacy information system is a three-layered architecture (Figure 2.1): *Hardware layer*, *Software Layer* and *Human Layer*.

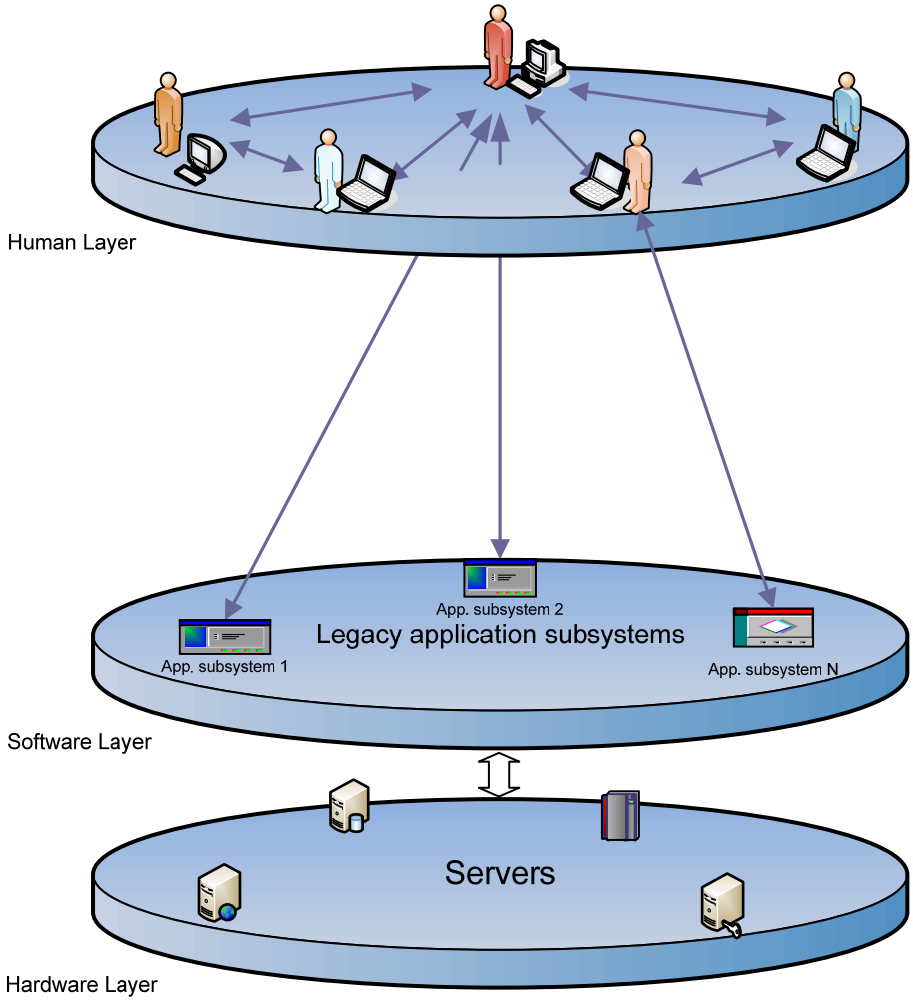

Figure 2.1: *Three-Layered Architecture of a Legacy Information System*

The hardware and software layers consist of hardware (e.g., mainframes, desktops, fixed telephones, etc.) and legacy application subsystems for different departments of the organization (e.g., human resource management, patient management, operating room management, etc.) that process, store, and distribute information. The legacy application subsystems were generally implemented as isolated silos.

The human layer consists of people using legacy application subsystems in different ways depending on their roles in the organization. Some people work as mediators to combine information flows coming from different independent legacy application subsystems. For example, a secretary in the surgery department of a hospital must combine information from the human resource management, patient management, and operating room management

subsystems to schedule a surgery for a patient. Other people are simply end-users who use the information delivered by either the mediators or a legacy application subsystem directly. For example, after scheduling a surgery, the secretary is the mediator who transmits the surgery schedule to the physician and his patient.

2.3 Weaknesses and Problems

Legacy information systems in organizations suffer from severe weaknesses and problems which are discussed in (Bisbal, Lawless, Wu, & Grimson, 1999b; Fasli, 2007; Howe, 1996; Huhns & Singh, 1997; Sommerville, 2007), and summarized below from three point of views:

Organization's point of view

- Legacy information systems are the result of massive investments in the past, therefore they cannot be easily phased out and replaced by modern information systems;
- Legacy information systems usually require many human interventions along the information flows. Thus, the transmission of information is not smooth and efficient;
- Furthermore, human intervention makes the information flows error-prone. It is difficult to trace an error and identify its root causes because there is no centralized and automated logging procedure;

End-User's point of view

- Legacy information systems are not capable to take initiatives autonomously when necessary, for example, to communicate with an end-user in case of emergency;
- Legacy information systems usually don't have the capability to benefit from the technological advantages of end-users' modern devices (smartphones, mobile phones, PDA). Therefore, end-users frequently perceive that legacy information systems perform inefficiently to process and dispatch data;
- Legacy information systems lack flexibility, meaning that new functionalities can only be partially implemented, or cannot be implemented at all, in response to new and constantly changing user requirements. Consequently, many non-automated processes or tasks must be performed by end-users themselves to satisfy their needs. Some of them are routine and tedious tasks, for example, phone calls, manual search of information in independent legacy applications, etc.;

Developer's point of view

- The maintenance and modernization of legacy information systems are generally difficult and costly because of incomplete legacy documentation, obsolete hardware

and software technologies, absence of programmers, analysts, designers involved in the development of those legacy systems in the past, etc.;

- When legacy information systems were implemented at departmental instead of enterprise-wide level, or at intra-enterprise instead of inter-enterprise level, the sharing and exchanging of information become a critical issue between such silos. For example, when two companies merge together, the weaknesses and problems discussed above become a much bigger challenge to integrate successfully two independent legacy information systems through a common communication protocol.

2.4 Literature Review of Approaches and Solutions

In the software engineering literature, a great number of approaches and solutions to change or replace legacy information systems have been proposed. The works by (Bisbal, Lawless, Wu, & Grimson, 1999a; Brodie, 1992; Oracle Corporation, 2006; Papazoglou, 1999; Stonebraker & Brodie, 1995; Thiran, Hainaut, Houben, & Bendlimane, 2006; Wu, Lawless, Bisbal, Grimson et al., 1997; Wu, Lawless, Bisbal, Richardson et al., 1997) are just a few examples. They are classified into the following three categories as discussed by Bisbal et al. (1999a) and Paradauskas and Laurikaitis (2006): *redevelopment*, *migration*, and *wrapping*.

Given a legacy information system, the research in the redevelopment category focuses on recreating that system from scratch using a modern architecture, tools, and databases. The legacy databases are redesigned; the existing applications are rewritten; etc. The resulting information system will replace the legacy one and operate on a new hardware and software platform. The redevelopment approach requires a huge undertaking and represents a high risk of failure during the redevelopment process. It would be both expensive and disruptive for business activities (Bakehouse & Wakefield, 2005).

In the migration category of works, the approaches and solutions concentrate on moving a legacy information system to a new software and hardware environment that allows the information system to be easily maintained and adapted to new business requirements while retaining the existing applications and databases of the original system without having to completely redevelop them (Bisbal et al., 1999a). The migration task represents a much more complex undertaking. It requires the involved organizations to consider each legacy data structure and every component of existing applications. This challenge is at the origin of many strategies, methodologies in the reverse-engineering technology. The Chicken Little strategy (Stonebraker & Brodie, 1995), the Butterfly methodology (Stonebraker & Brodie, 1995), and the Iterative Reengineering method (Bianchi, Caivano, & Visaggio, 2003) are just a few examples.

In the wrapping category of works, the approaches and solutions emphasize on (1) maintaining the status quo of the legacy information system, and (2) developing software components called wrappers to integrate it into the modern information system architecture of an organization. The wrapper acts as a server that provides requested services considered as black boxes. Many wrapping approaches have been proposed, for example, Distributed Object Management (DOM) by (Brodie, 1992), Agent-Based by (Azevedo, Toscano, & Bastos, 2002; Fiedrich & Burghardt, 2007; Papazoglou, 1999; Sycara & Zeng, 1996), R/W Wrapper Architecture methodology by (Thiran et al., 2006), etc.

In the DOM approach, a global object space is created, in which data are organized into data objects. To fulfill a service request, a wrapper is able to convert data from legacy database systems into data objects in the global object space. Conversely, it can also translate data objects into data structures readable by legacy systems. In his research, Brodie (1992) introduces the concept of agent, and considers wrappers as one specific category of agents.

In the agent-based approach of (Azevedo et al., 2002; Papazoglou, 1999; Sycara & Zeng, 1996), the authors define categories of agents such as brokers, translators, wrappers to help other agents to access the resources of legacy information systems. In an agent-based environment, all agents can communicate and understand each other.

This thesis will focus on the wrapping category using the agent-based approach to enhance the legacy information systems in a modern software and hardware environment.

3

Agent Technology

This chapter aims at presenting the important aspects of the agent technology such as the real world's agent concept applied to software engineering, the multi-agent systems and the agent approach of development methodologies. The chapter concludes with a discussion about the application domains of agent technology.

3.1 Introduction

Since the first introduction of the term "agent" in 1950, especially during the last two decades, the agent concept has attracted much attention from the research community in the software engineering field. This concept has been studied and are being continuously studied by several public and private research groups in universities, research laboratories, scientific associations, etc., such as the Intelligent Software Agents Lab at Carnegie Mellon University, the Software Agent Group of the MIT Media Laboratory, the MAPLE Research Group of the

University of Maryland, the AgentLink – European Co-ordination action for agent-based computing, the Foundation for Intelligent Physical Agents (FIPA), the Object Management Group (OMG), the Whitestein Technologies AG, the IBM Research Laboratory, the Telecom Italia Lab, just to name a few.

Through these researches, the agent concept has been developed as a technology to build application systems for organizations.

3.1.1 Aspects of Agent Technology

When talking about the agent technology in the software engineering field, there are many aspects to discuss. This thesis focuses on the following:

- definition of software agent,
- architecture of a software agent,
- definition of multi-agent systems,
- software environment supporting multi-agent systems,
- communication mechanism between cooperating agents,
- framework for multi-agent systems,
- methodologies to develop multi-agent systems,
- application domains.

3.1.2 From Human Agents To Software Agents

The Collins Dictionary of the English Language defines an agent as follows:

> ***Definition 3.1****: An agent is a person or thing that acts or has the power to act.*

This definition gives a general answer to "what is an agent?" It is large and refers to any entity existing in the real world such as a human, an animal, a plant, an object, etc.

Also according to the same dictionary, a human agent is stated as follows:

> ***Definition 3.2****: A human agent is a person who acts on behalf of another person, group, business, government; representative.*

A person is a human being who lives in the natural environment, who has intelligence, autonomy, and capability to observe, to reason, to infer, to act, and to react to events in its environment. These properties are manifested through his behaviors. To become an agent, this person must work for another person, a group of people, an organization, etc.

Inspired from this concept of human agent, several researchers in the software engineering field have introduced a programming concept, called *software agent*, in order to open a new

spectrum in developing computer-based information systems. The new computer-based information systems are more intelligent, more flexible, easier to develop and maintain than those based on the concept of software object.

Before going into definitions of what a software agent is, let us review its history as follows.

3.1.3 History of Agents

Software agents have a long history in computer science. The following are their milestones in the field, based on the research of (Cummins, Davy, Finnegan, & Corroll, 2003; Middleton, 2002; Nwana, 1996):

1950s: The term *agent* is borrowed from philosophy and used by researchers in the artificial intelligence field. Agents are considered as intelligent physical objects, created to satisfy goals. For example, the thermostat in the cooling system of a car can activate and deactivate the cooling system according to the temperature of the engine in order to keep it at a constant predetermined operating temperature.

1960s: Agents are defined as objects having both mental and social characteristics. This new concept is used to build intelligent machines, computer systems involving subparts (Nevell, 1962). Each subpart has a goal and can solve a part of a problem.

1970s: The agent concept evolves into two directions, artificial intelligence systems and distributed artificial intelligence systems.

In the first direction, researchers continue their research on capabilities of an intelligent agent. The objective is to create objects with capabilities of performing a given set of tasks individually based on its beliefs, desires, and intentions learned from users.

In the second direction, researchers concentrate on systems of agents with the following characteristics:

- a system has many distributed agents (multi-agent system);
- each agent in the system has a task to do, a contact address, a set of behaviors, and capabilities of coordinating and cooperating with other agents;
- the communication between agents is based on message passing (Hewitt, 1977; Hewitt, Bishop, & Steiger, 1973), a communication is made by sending of messages to recipients.

These characteristics raise several issues and challenges, such as the interaction, communication, coordination and cooperation between agents. One of the first researches that addresses these issues and challenges is the CONTRACTNet, a communication protocols for agents in a multi-agent system (Smith, 1977).

1980s: The agent concept in distributed artificial intelligence systems is ported to database systems, operating systems, and network systems fields to develop distributed database systems, multitasking operating systems for personal computers, and network management systems.

In this period, many research groups concentrate on domain knowledge for multi-agent systems (McCarthy, 1987). They formalize knowledge specific to a domain by building a corresponding ontology which will then be integrated into a multi-agent system. The agents in that multi-agent system use the same integrated domain-specific ontology to communicate between them, to understand each other, and to cooperate in solving a problem.

1990s: The agent concept becomes a new paradigm in software engineering beside other existing concepts such as object approach, functional approach. The software view of agents is extensively studied through the works of (Maes, 1994; Wooldridge & Jennings, 1994). In their research, the agent is considered as a software object, a piece of program which has the same characteristics of an agent in the artificial intelligence field.

2000s: The research on software agent continues to diversify its focus:

- multi-agent systems,
- personal assistant agents,
- mobile agents.

The research on multi-agent systems concentrates on the platforms, frameworks, communication languages and protocols (Bellifemine, Caire, & Greenwood, 2007; Gutknecht & Ferber, 2000; Lin, 2007).

The research on personal assistant agents focuses on agent-based systems which assist human actors in their daily activities (Menczer, Street, & Vishwakarma, 2002; Zhang, Ghenniwa, & Shen, 2006).

The research on mobile agents studies a kind of personal assistant agents that, apart from security considerations, can programmatically move from machine to machine to collect available data and information on available services for human actors (Braun & Rossak, 2005; Mitchell, 2004).

3.2 Software Agents

3.2.1 Definition

Software agents have many similar characteristics of human agents in the real world. Each software agent:

- has a state;
- has one or more behaviors;
- has the capability to operate autonomously and independently in a software environment;
- can use one or many communication languages and a set of specific vocabularies, called domain ontology, to communicate with other agents;
- can receive, interpret, and answer or ignore any request from other agents.

More formally, the following definitions explain in chronological order what a software agent is.

> ***Definition 3.3****: An agent (a software agent) is a computer program that simulates a human relationship by doing something that another person could do for you* (Selker, 1994).

> ***Definition 3.4****: An agent is a computer system that is situated in some environment, and that is capable of autonomous action in this environment in order to meet its design objectives* (Wooldridge & Jennings, 1994).

> ***Definition 3.5****: An agent is a program that performs a specific task on behalf of a user, independently or with little guidance. An agent has its own internal state that can be described by a set of characteristics and execution rules* (Bui, 2000).

> ***Definition 3.6****: An agent is a small, autonomous or semi-autonomous software program that performs a set of specialized functions to meet a specific set of goals, and then provides its results to a customer (e.g., person, another program) in a format readily acceptable by that customer* (Daniel H. Wagner Associates, 2005).

So far, there is still not yet a common definition which is accepted by the whole community of researchers. However, a synthesis of the above definitions allows us to define a software agent as a piece of software that can act autonomously in an agent-based environment and has capability to accomplish a given set of tasks to attain given goals on behalf of human actors and other agents.

In this thesis, we also consider the concept of software agent under the object-oriented view:

> ***Definition 3.7****: A software agent is an instantiated object at running time, which is capable to initiate, participate or refuse a communication, and to execute autonomously an operation to attain its goals during its life cycle.*

These capabilities are essential, since they differentiate objects as software agents from traditional objects as discussed in (Odell, 2002; Wooldridge, 2002).

3.2.2 Types of Agents

To classify agents, there are many taxonomies, such as multi-dimensional space (Nwana, 1996), biological and mathematical models (Franklin & Graesser, 1996), weak and strong notions of agency (Wooldridge, 2002; Wooldridge & Jennings, 1995), complexity (Russell & Norvig, 2002). This subsection aims at presenting the first three taxonomies.

3.2.2.1 Nwana's Classification

According to Nwana (1996), agents exist in a multi-dimensional space. The dimensions are *mobility*, *deliberativeness or reactivity*, *autonomy*, *cooperation*, *learning*, and *roles*. Through those dimensions and their intersections, the author has classified agents in seven categories:

- ***Mobile agents***: are agents with the capability to move between different nodes (machines) in a computer network.
- ***Collaborative agents***: are agents that can cooperate and coordinate their actions with other agents in their community.
- ***Interface agents***: are personal assistant agents with ability to communicate with users (human actors), learn from users and then assist them in learned situations.
- ***Internet agents***: help internet users in searching, collecting, filtering, and selecting information. These agents are also called *Information agents*.
- ***Reactive agents***: are agents that act simply in response to a stimulus in their environment, for example, a request of another agent or a scheduled event.
- ***Hybrid agents***: are agents that are designed by integrating different features taken from two or more agent categories. For example, an agent has capability to move between nodes (mobile agents) and to learn from users actions (interface agents).
- ***Smart agents***: are hybrid agents that have capability to learn and cooperate with other agents. They are considered as the truly "intelligent" agents.

3.2.2.2 Franklin and Graesser's Classification

In (Franklin & Graesser, 1996) software agents are considered as a branch of computational agents that has its root in autonomous agents (Figure 3.1). The software agents are further divided into three subcategories: *task-specific agents*, *entertainment agents*, and *viruses*.

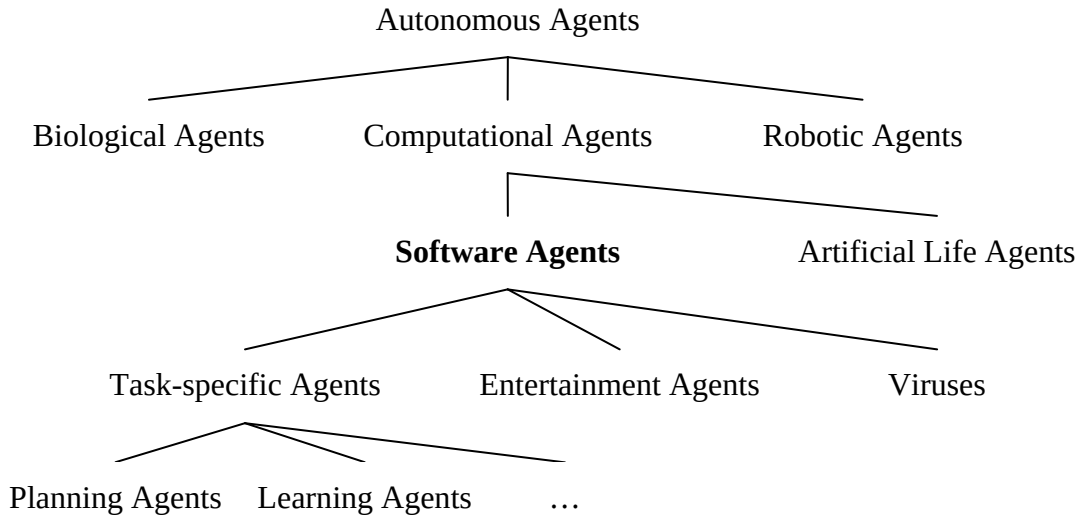

Figure 3.1: *Agent Classification by Franklin and Graesser (1996)*

The agents in each subcategory might be further categorized in subclasses based on their role such as planning, learning, communicative, reasoning, etc., giving for example: *planning agents*, *learning agents*, *communicative agents*, *reasoning agents*, *negotiating agents*, *information agents, etc*.

3.2.2.3 Wooldridge's Classification

Wooldridge and Jennings (2002; 1995) have identified and divided agents' properties into two subsets:

- the first basic subset consists of four properties: *autonomy*, *social ability*, *reactivity*, and *pro-activeness*;
- the second subset consists of three properties: *adaptiveness*, *intelligence*, *mobility*.

These two subsets of properties lead authors to classify agents into two categories: the *strong* agents and the *weak* agents.

- The strong agents are those who have four basic properties in the first subset and one or more additional properties in the second subset. The mentalistic agents and emotional agents are examples of the strong agents.
- The weak agents are agents who have only the first subset of properties. An example of weak agents is software robots (softbots) which only have the capability to

cooperate with users (social), to initiate an action based on a change of their environment (autonomy, reactivity, and pro-activeness). However, software robots cannot learn from experiences (no adaptiveness, no intelligence).

In this thesis, our agents belong to the Task-specific Software Agents category of Franklin and Graesser (see Figure 3.1). In particular, they have the following properties:

- autonomy, cooperation, reactivity, and interface as defined by Nwana;
- autonomy, social ability, reactivity, and pro-activeness as defined by Wooldridge.

3.2.3 Agent Architecture

According to Wooldridge (2002), the structure of an agent consists generally of three parts: input, process, and output, as shown in Figure 3.2.

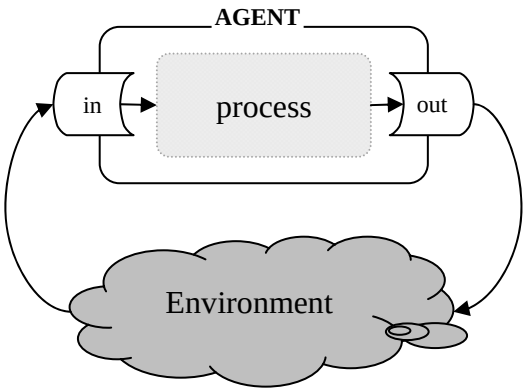

Figure 3.2: *Components of an Agent*

- The input part is a sensor which captures the changes and stimuli of the environment.
- The process part, also called kernel, processes the input data, performs the reasoning, and makes decisions based on knowledge and rules.
- The output part takes appropriate actions to reach agent's goals and to change its environment.

The kernel may be divided in many subcomponents based on their function, for example, a subcomponent for processing the input data, a subcomponent for reasoning, etc. The kernel's division and arrangement of subcomponents are determined by the type of agent architecture.

There are four main types of agent architecture (Bellifemine et al., 2007): logic based, reactive, BDI (Beliefs-Desires-Intentions), and layered architectures. The advantages and disadvantages of each one have been discussed in (Bellifemine et al., 2007; Wooldridge, 2002).

In this thesis, an hybrid architecture is used to build the internal structure of agents. That means we will essentially combine the logic based and reactive architectural types. The agent behaviors resulting from this architecture will be programmed using the procedural language Java for some of them, and the declarative language Jess for others.

3.3 Multi-Agent Systems

This section focuses on a system consisting of several agents, called a multi-agent system. More concretely, we discuss three major aspects: definition, platforms, and communications.

3.3.1 Definition

The term *multi-agent system* was initially used to refer to a set of programs called problem solvers distributed over a network and capable of coordinating their tasks to solve a given problem. The term has been then defined in different ways as discussed in (Fasli, 2007; Protogeros, 2007; Weiss, 2000; Wooldridge, 2002).

The following definition summarizes the properties of a multi-agent system.

> ***Definition 3.8:*** *A multi-agent system is a system that consists of a number of alive agents working together in an environment to accomplish designed goals in order to solve a common problem*.

This definition stresses four key elements: alive agents, cooperation, environment, and goals.

- ***Alive agents***: This expression refers to active entities such as persons, robots, intelligent objects, software agents, etc., which have resources, possess knowledge and skills, and can autonomously perform actions.
- ***Cooperation***: Agents are required to work together through their "social" networks using their defined rules and communication languages.
- ***Environment***: It represents the settings in which agents reside and act. It motivates and stimulates agents.
- ***Goals***: A multi-agent system must fulfill the goals for which it is designed.

3.3.2 Agent Platforms

The term *environment* used in Definition 3.8 refers to an agent platform. Its role is to provide the basic services to agents and to assist them to perform efficiently the assigned tasks. From the development point of view, the agent platform reduces the complexity of implementing a multi-agent system.

According to the Foundation for Intelligent Physical Agents (FIPA), any agent platform must have at least three services (Odell, 2007):

- The agent management services, also called Agent Management System (AMS), allow the management of each agent residing in the platform, e.g. creating, registering, suspending, removing, etc. Once an agent is initiated, AMS gives it a name, an address and registers that agent in the AMS' control list. This pair, name and address, is used to identify each agent in the platform.
- The yellow pages services, known as Directory Facilitator (DF), allow an agent to publish its services. That agent becomes a service provider. Any agent can use the yellow pages to search a service provider according to its needs.
- The communication service, also called Message Transport Service (MTS), allows an agent to send messages to other agents that are located within the same or different platforms.

On the basis of the above FIPA specifications on minimum services, several agent platforms are proposed, for example at the time of this writing, JADE (Bellifemine et al., 2007), JACK (Agent Oriented Software Pty, 2006), AgentBuilder (Acronymics, 2004), and MadKit (Gutknecht & Ferber, 2000, 2001; Gutknecht, Ferber, Michel, & Mansour, 2008), just to name a few. The comparison of the advantages and disadvantages of these agent platforms is not within the scope of this thesis. Vrba (2003) has published a comparative study of older versions of the above platforms.

3.3.3 Agent Communication

Agents must communicate between them to cooperate:

- querying and identifying an agent;
- expressing desires and intentions to another agent;
- transmitting data to agents;
- task coordination with other agents.

Each communication is characterized by a communication mode, a communication language, a communication protocol, and a set of vocabularies called ontology.

3.3.3.1 Communication Modes

There exist three communication modes:

- ***One-to-One***. It is also called point-to-point, or unicast. This communication mode allows an agent to send a message to another agent. For example, a lab technologist agent communicates with a physician agent to transmit a medical result.

- ***One-to-Many***. It is also called multicast. This communication mode allows an agent to send a message to a group of agents. For example, a personal assistant agent who helps its owner buy a book which is selling in several bookstores communicates with bookstore agents to ask the book's price.
- ***One-to-All***. It is also called broadcast. This communication mode allows an agent to send a message to all agents in the system, such as a notification of an important event, an alert. For example, let us imagine a course where a professor and his students are supported by a multi-agent system consisting of personal assistant agents; Suppose that the professor must cancel a course session, he instructs his personal assistant agent to inform his students; the professor's personal assistant agent will transmit a cancellation notification to all students' personal assistant agents which in turn alert students.

3.3.3.2 Communication Languages

As the communication among agents is based on the exchange of messages, each message must be encoded in a language understandable for them, called the communication language. The communication languages are divided into two categories: the languages used to define the structure of a message and those to describe its content, the body. The structure of a message contains generally the following elements:

- the sender's address;
- the receiver's address;
- the subject of the message;
- the information content to be sent;
- the language which defines the syntax and the encoding scheme of the information content;
- the name of ontology where the semantics of vocabulary is defined.

Two most discussed agent communication languages for the first category are: KQML (Finin, Fritzson, McKay, & McEntire, 1994) and FIPA ACL (FIPA, 2002). Examples of the second category of agent communication languages are KIF (Genesereth & Fikes, 1992) and FIPA SL (FIPA, 2002).

In a group of cooperating agents, they must share a common communication language to understand each other. An agent that participates in several groups may use several languages.

3.3.3.3 Communication Protocols

FIPA has defined several communication protocols, called also FIPA interaction protocols, for agents in a multi-agent system (FIPA, 2002). The list of these interaction protocols

consists of: *FIPA Request Interaction Protocol*, *FIPA ContractNet Interaction Protocol*, *FIPA Cancel Meta Interaction Protocol*, *FIPA Query Interaction Protocol*, just to name a few. The following discussion focuses on the first three protocols which will be used later by agents in the next chapters.

FIPA Request Interaction Protocol

The *FIPA Request Interaction Protocol* allows an agent, called an initiator, to request another agent, called a participant, to perform a task. The UML sequence diagram of the request interaction is presented in Figure 3.3 and described below:

- *Ini 1:* The initiator sends a *request* message to the participant. The *request* message indicates the task which the initiator would like the participant to perform.

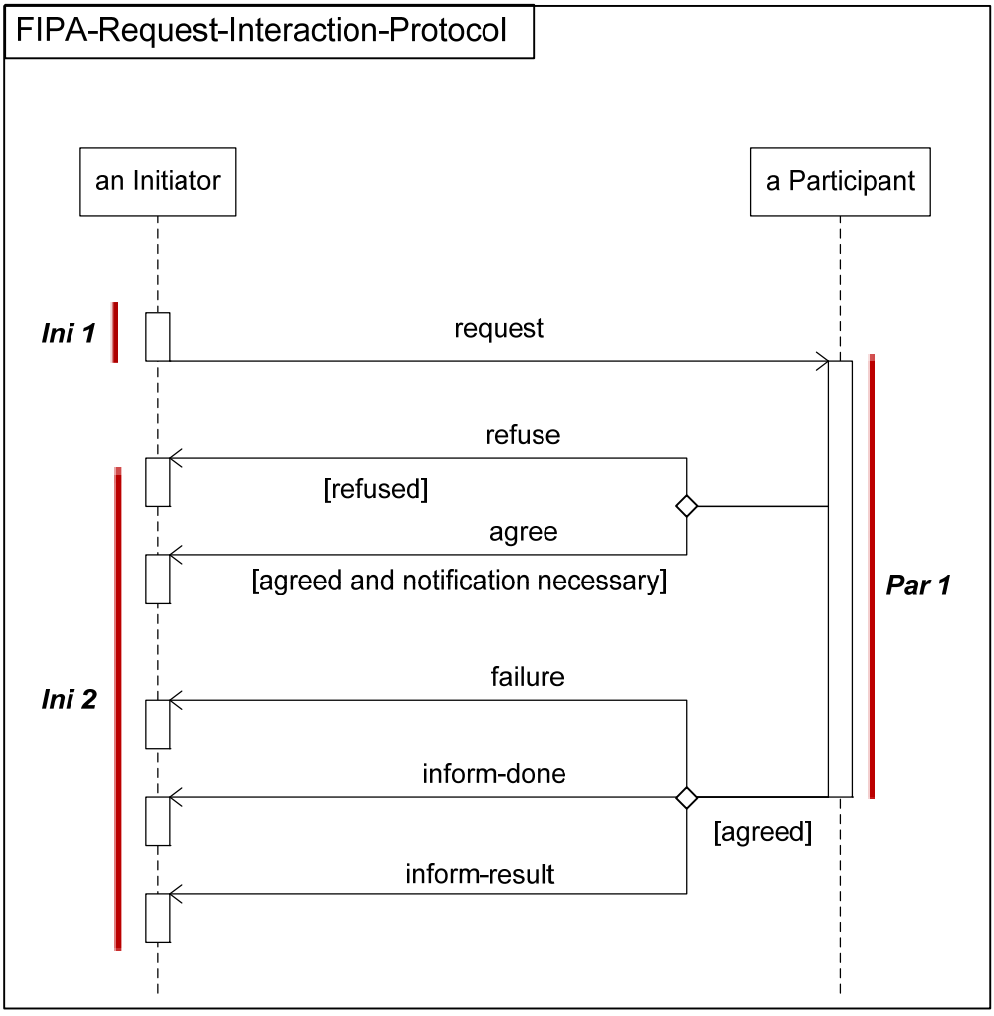

Figure 3.3: *FIPA Request Interaction Protocol*

- *Par 1:* The participant receives the *request* message, then interprets it, and makes a decision to either refuse or agree to perform the task.

If the task is refused, the participant replies to the initiator by a *refuse* message, presented by the *[refused]* arrow. The *refuse* message lets the initiator know that its request is denied.

If the participant agrees to perform the task, it communicates with the initiator as follows:

- The participant replies to the initiator by an *agree* message to let it know that the request were accepted and the task is being performed, as presented by the *[agreed and notification necessary]* arrow. The *agree* message is optional.
- Upon the task completion, the participant sends a message to the initiator to report execution result, as presented by the *[agreed]* arrow. The message might be a *failure, inform-done*, or *inform-result* message. The *failure* message is sent in case the participant fails in its attempt to fill the request; the *inform-done* message is sent if the participant successfully completes the request and only wishes to inform the initiator that its request is done; the *inform-result* message is sent if the participant wishes to inform the initiator that its request is fulfilled successfully and to deliver the result simultaneously.

- *Ini 2:* If the initiator receives the *refuse* message, the interaction will be stopped. In contrast, the initiator will continue to wait until it receives one of the completion messages (*failure*, *inform-done*, or *inform-result*), after that, the interaction will be finished.

FIPA ContractNet Interaction Protocol

The *FIPA ContractNet Interaction Protocol* is an interaction protocol, first proposed by (Smith, 1977), used in negotiation between an agent called an initiator and other agents called participants to select one agent with whom a contract will be concluded for executing some tasks. For example, an agent negotiates with several bookstore agents using FIPA ContractNet to buy a book at the cheapest bookstore. FIPA (2002) has standardized this protocol for contract negotiation between cooperating agents. The complex interaction between the initiator and the participants is depicted by the sequence diagram (Figure 3.4), and described as follows:

- *Step 1:* The initiator sends a message to *m* participants to call for their proposals (*cfp* messages). A *cfp* message contains a subject, requirements, and a reply deadline.
- *Step 2:* Each participant interprets the *cfp* message, examines the requirements in the message, sends back to the initiator:
  - either a *refuse* message,
  - or a *propose* message containing a proposal.

The participant can also ignore the *cfp* message without replying to the initiator.

Let:

n: the number of participants that reply to the initiator within the *reply deadline,* $n \leq m$.

r: the number of participants that refuse the call for proposal.

p: the number of participants that offer a proposal. $n = r + p$.

- *Step 3:* The initiator receives *(r+p)* messages. Once the deadline for proposal is reached, the initiator evaluates the *p* proposals based on its selection criteria giving two list:
  - list of unsuitable proposals (LUP),
  - list of suitable proposals (LSP) ordered by preferences or by the degree of fulfillment of the initiator's objective (best suitable proposal, second best proposal, etc.).

  The second list will be used in the next step to find out a contractor.

  Let:

  u: the number of unsuitable proposals.

  s: the number of suitable proposals, $0 \leq s \leq p$.

  $u + s = p$.

- *Step 4:*

  Case 4.1: The initiator sends a *reject-proposal* message to each participant in the LUP list of proposals.

  Case 4.2: The initiator selects the best suitable proposal from the LSP list and sends an *accept-proposal* message to the best selected participant.

- *Step 5:* The rejected participants take no further action because their interactions with the initiator terminate immediately. The selected participant executes the tasks defined in its best proposal. Upon completion, the selected participant sends to the initiator a final message which is either a *failure,* an *inform-done,* or *inform-result* message.

  The *failure* message is sent in case the participant fails in its attempt to execute the tasks; the *inform-done* message is sent if the participant successfully completes the tasks and only wishes to inform the initiator that its tasks are done; the *inform-result* message is sent if the participant wishes to inform the initiator that its tasks are done and to deliver the result simultaneously.

- *Step 6:*
  - *Case 6.2 & 6.3:* Upon receipt of an *inform-done* or *inform-result* message, the unused proposals in LSP become, in certain sense, unsuitable (LUP list). Therefore, the initiator repeats *Case 4.1* by sending a *reject-proposal* message to the unsolicited participants. This completes the interaction between the initiator and the participants.
  - *Case 6.1*: Upon receipt of a *failure* message, the initiator selects another suitable proposal from the LSP list then goes to *Case 4.2* of Step 4, and so forth.

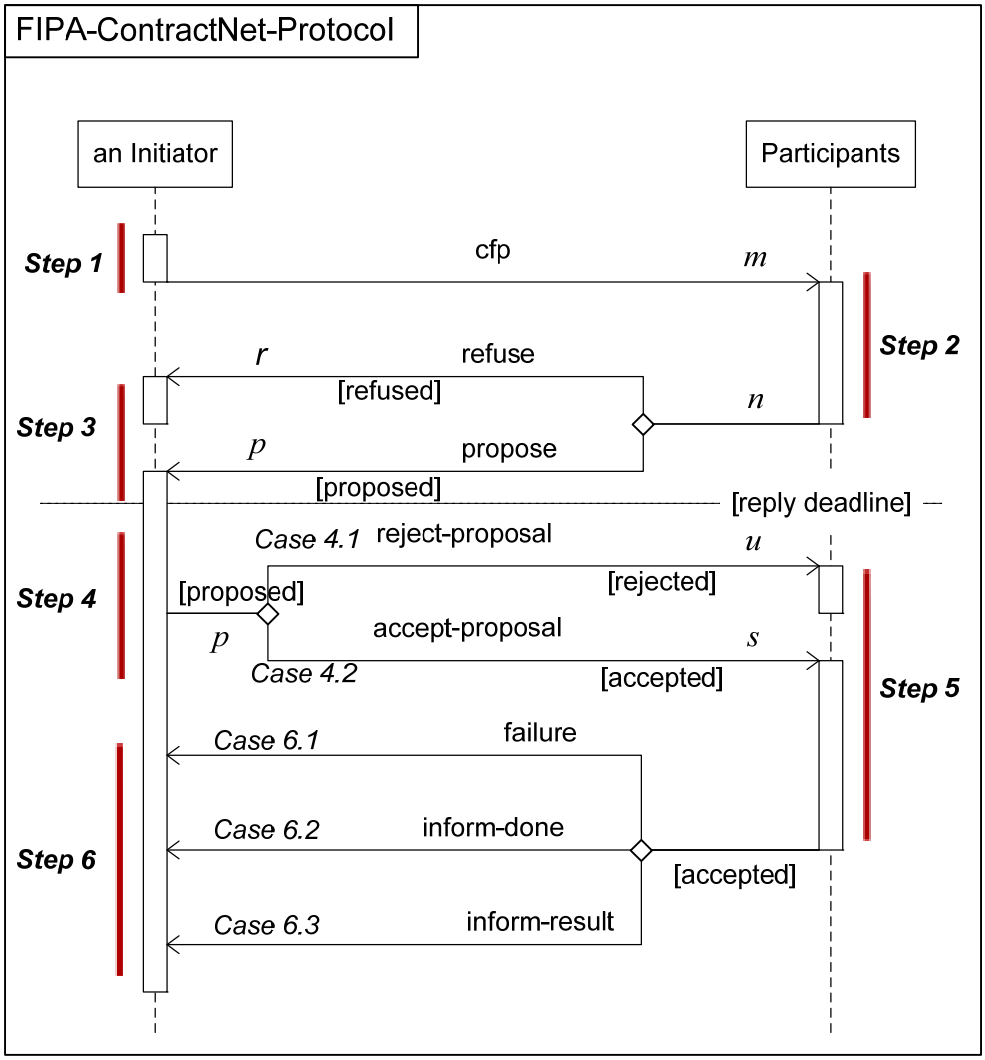

Figure 3.4: *FIPA ContractNet Interaction Protocol*

FIPA Cancel Meta Interaction Protocol

An initiator agent might need to cancel a message previously sent to a participant. FIPA provides a *Cancel Meta Interaction Protocol* allowing the initiator and the participant to interact between them (Figure 3.5).

- *Step 1:* The initiator sends to the participant a *cancel* message containing the parameters required to identify the message to be cancelled.
- *Step 2:* The participant interprets the *cancel* message, executes the cancellation request, and reports the result to the initiator. The result is either an *inform-done* or a *failure* message. The *inform-done* message lets the initiator know that its request has been executed successfully, while the *failure* message shows that the cancellation could not be done.

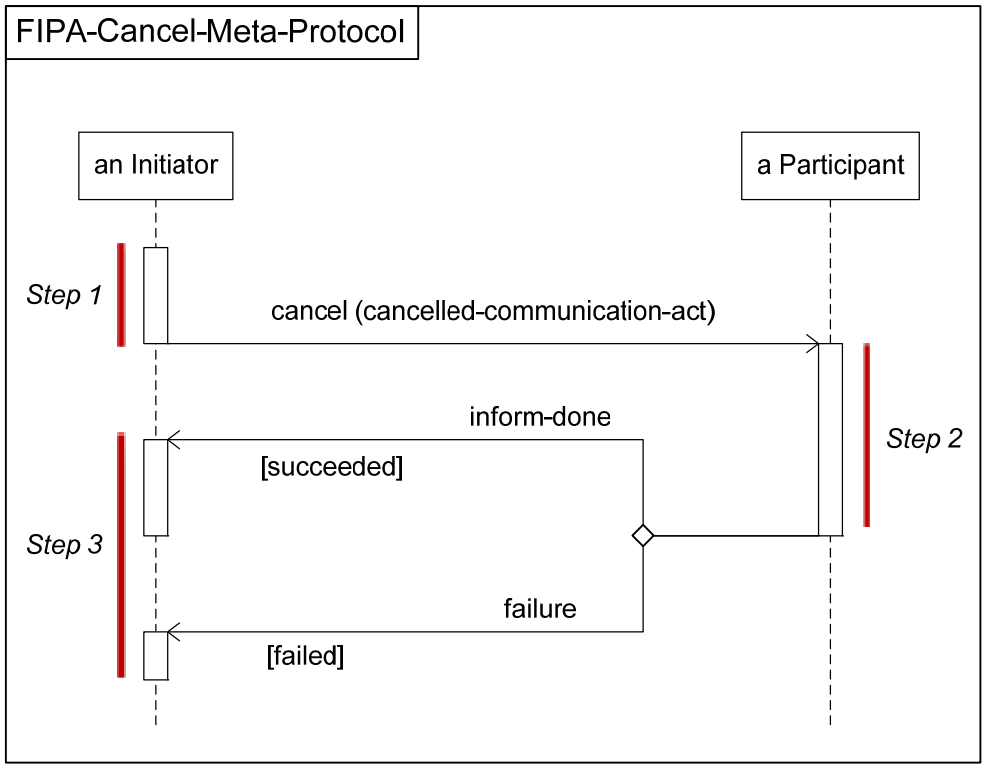

Figure 3.5: *FIPA Cancel Meta Interaction Protocol*

- *Step 3:* The initiator receives the success or failure result message and terminates its interaction.

3.3.3.4 Ontology in Communications

In a communication between two or more agents within a particular domain (e.g., medicine, finance, etc.), the agents need to share not only a common agent communication language and a common protocol, but also a common ontology.

For example, like in human world, a conversation without sharing the common terminology in a domain leads easily to misunderstanding between the participants for at least two reasons:

- ***homonym***: a word can have different meanings and express different things (e.g., *doctor (n)*: 1. a person licensed to practice medicine, 2. a person who has been awarded a higher academic degree in any field of knowledge, etc.);

- ***synonym***: the same thing can be presented and expressed through different words (e.g., the words *physician*, *doctor*, *medical practitioner* can be used to talk about a person licensed to practice medicine.).

An ontology is a formal representation of knowledge as a set of concepts within a domain, and the relationships between those concepts. It is used to describe the domain and to share a common understanding of the entities within that domain (Davies, Studer, & Warren, 2006).

The usage of an ontology in a multi-agent system allows agents to avoid misunderstandings in their communications.

There are a number of tools to develop ontologies. The tools that are attracting much attention of the research community are: Protégé (Stanford Center for Biomedical Informatics Research, 2007), TopBraidComposer (TopQuadrant, 2007), OpenCyc (Cycorp, 2001), Ontolingua (Stanford University, 2005).

3.4 Agent-Oriented Methodologies for Multi-Agent Systems Development

Today, several agent-oriented methodologies to build multi-agent systems have been developed: Gaia (Wooldridge, Jennings, & Kinny, 2000; Zambonelli, Jennings, & Wooldridge, 2005), MaSE (DeLoach, 2004), MAS-CommonKADS (Iglesias & Garijo, 2005), and JADE_Methodology (Nikraz, Caire, & Bahri, 2006), just to name a few. They are based on different theoretical foundations (Henderson-Sellers & Giorgini, 2005): Artificial Intelligence, Object-Oriented Programming, and **i*** organization modeling framework *Tropos* (Giorgini, Kolp, Mylopoulos, & Castro, 2005). These methodologies contribute significantly to the rigorous and systematic development of agent-based systems. Most of them do not emphasize the ontological approach which we consider as a critical way to share the knowledge between human actors and software agents through the successive phases of the development process.

JADE_Methodology is a agent-oriented methodology that supports the ontological approach. It encompasses the analysis and design phases to develop software agents on the JADE platform. This methodology proposes to build the ontology at the end of the design phase in order to share the knowledge between software agents.

In this thesis, we inspire from the theoretical foundations presented in Tropos and adapt the JADE_Methodology approach to establish a new development process which will be discussed in Chapter 7.

3.5 Application Domains using Agent Technology

Agent technology is recognized as a suitable technology for developing complex and extensible systems (Fasli, 2007; Ferber, 1999; Wooldridge, 2002), which might involve independent subsystems built by distinct development teams, in order to solve intelligently a given problem. These systems require a high flexibility, for example, adding or removing a subsystem could not impact the operation of other subsystems or the whole system.

Such complex and extensible systems might be applied to many domains such as information retrieval, e-commerce, e-healthcare, personal assistant, social simulation, distributed sensing, virtual environment. The following are three domains which have more and more applications developed by using agent technology.

In the information retrieval domain, software agents can be used to assist users in retrieving and managing information from several information sources which are accessible over the internet. These software agents, also called *Information Agents*, can be designed as:

- personal information agents, for example, the Newt system in (Maes, 1994), to search and filter out information, then suggest to users the result;
- a multi-agent system, for example, Grouper (Zamir & Etzioni, 1999), MetaCrawler (InfoSpace, 2009), CiteSeer (Steve, Lee, & Kurt, 1997), in which each agent searches and filters out information in a field, then the result of agents are collected to cluster and index retrieved information for users.

In the e-commerce domain, software agents can be used to assist merchants and clients in selling, buying, and exchanging information of goods and services on the internet. These software agents can be designed as a multi-agent system, such as ShopBots and PriceBots (Clark, 2000; Greenwald & Kephart, 1999; Kephart & Greenwald, 2002; Waldeck, 2005), IntelliShopper (Menczer et al., 2002), to do business.

In the e-healthcare domain, software agents can be used to assist healthcare professionals and patients, and to enhance healthcare services. These software agents can be designed as a multi-agent system, such as HealthAgents (González-Vélez et al., 2009), SHARE-it (Cortés et al., 2008), ASPIC (Tolchinsky, Cortés, & Grecu, 2008), to share information among healthcare professionals, between the healthcare professionals and the patients.

Today, in most organizations, the information needs are satisfied by information systems in which modern and legacy subsystems must co-exist despite the high cost of maintaining the latter. The next chapters of the thesis will study “how” to apply the agent technology to enhance legacy information systems, especially those in the e-healthcare domain.

4

Agent-Based Approach for Legacy Information Systems

The present chapter aims at defining the concept of multi-agent subsystem and an organizational model to exploit the existing legacy information system in a company despite its weaknesses discussed in Chapter 2.

This chapter is structured as follows. The first section introduces a new integration layer and its objectives to enhance the legacy information systems. The second section presents a multi-agent subsystem in the integration layer. The third section discusses a model of the multi-agent subsystem. The fourth section presents the operational characteristics of agents in the multi-agent subsystem. The fifth section proposes a framework for the multi-agent subsystem. Finally, the conclusion section highlights the potential advantages of the agent-based approach to enhance legacy information systems.

4.1 Integration Layer and Its Objectives

As presented in Figure 2.1, the traditional architecture of legacy information systems consists of three layers: *Human Layer*, *Software Layer*, and *Hardware Layer*. The innovatory idea is now to add a new software *Integration Layer* into the traditional architecture between the human and software layers as shown in Figure 4.1 (Nguyen, Fuhrer, & Pasquier, 2008).

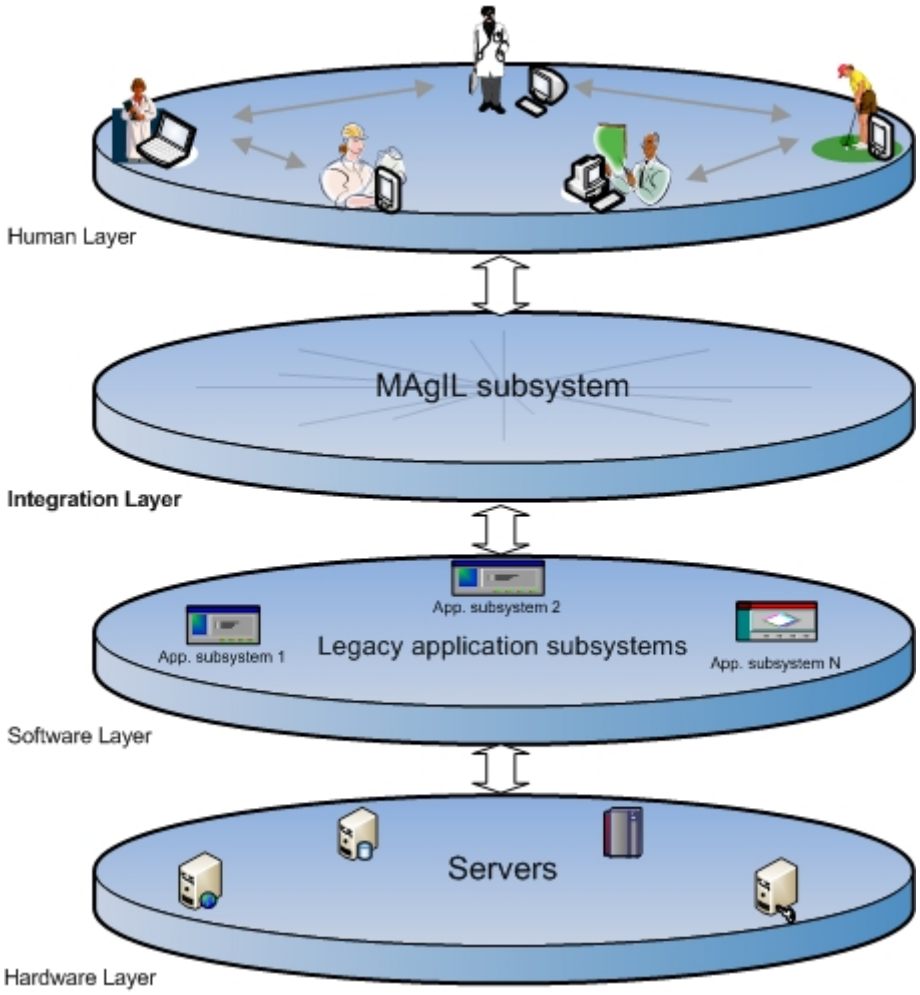

Figure 4.1: *Architecture of an Enhanced Legacy Information System*

The Integration Layer is targeted to provide a software environment for fulfilling three sets of objectives:

- assisting human actors in their inter-personal relationships and in their interaction with the legacy subsystems;
- facilitating data exchange between the legacy subsystems;
- allowing new functionalities to be added, as far as practicable, directly into the Integration Layer, thus minimizing the impact of new user requirements on the maintenance of the legacy subsystems during their life cycle.

This software environment is termed **MAgIL subsystem**. The acronym **MAgIL** stands for Multi-Agent Integration Layer.

4.2 Multi-Agent Subsystem in the Integration Layer

The MAgIL subsystem has three categories of software agents:

- the personal agents, acting as the personal assistants of human actors;
- the legacy information system (lis) agents, playing the role of the representatives of legacy subsystems;
- the service agents, providing the system-wide surveillance services such as monitoring, alert notification, time-out control.

These agents work in cooperation between them, and with the human and software layers. This cooperation is represented by Figure 4.2, using the graphical symbols explained on the next page.

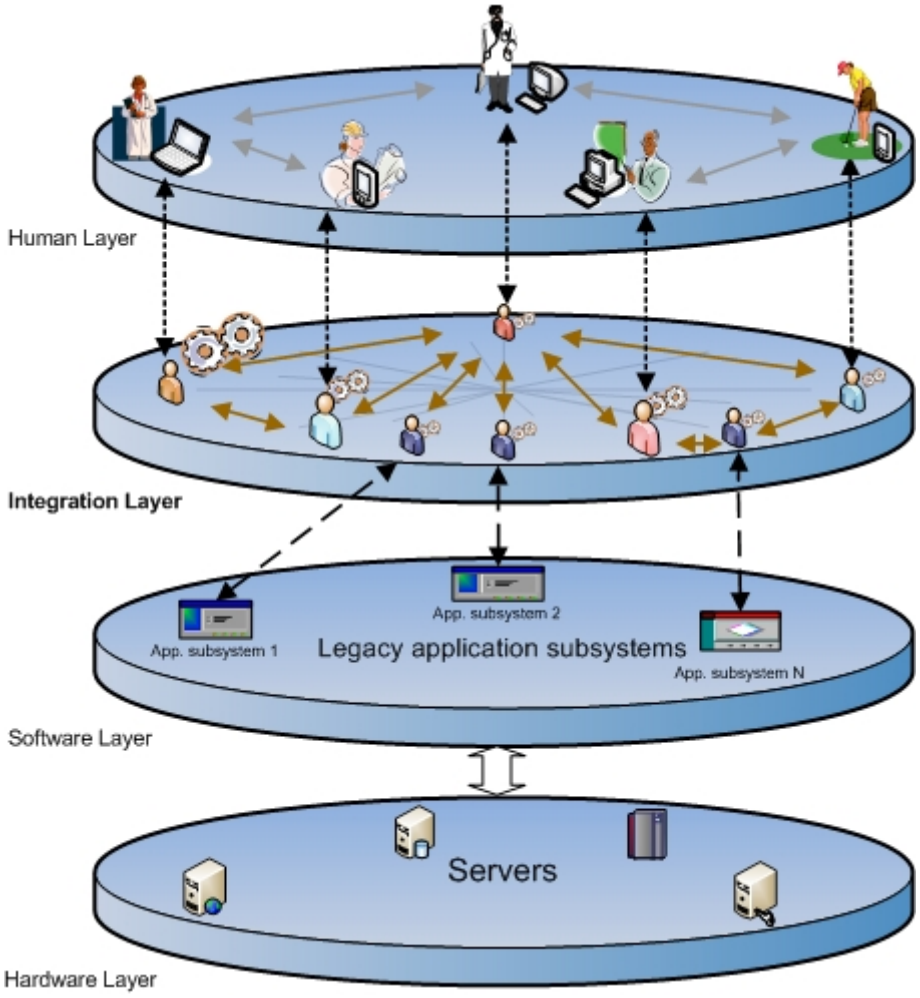

Figure 4.2: *Cooperation of Agents within the Integration Layer and among layers in an Enhanced Legacy Information System*

For the Human, Integration, and Software Layers:

- the people icons in the Integration Layer ( ) represent agents;
- the cogwheel icons ( ) beside each agent mean that agents can work autonomously;
- the solid double arrows ( ) in the Integration Layer describe the cooperation between agents within, and between categories;
- the vertical dotted double arrows ( ) depict inter-layer interactions between human actors and their personal agents;
- the vertical dashed double arrows ( ) depict inter-layer interactions between lis and service agents in the Integration Layer and their associated subsystems in the Software Layer.

4.3 Organizational Model of the Multi-Agent Subsystem

In this section we focus on the MAgIL subsystem as an organization involving human actors and agents. The three categories of software agents defined in the previous section are the foundation of an organizational model (Figure 4.3) that will be described in detail by considering the simple case of a human actor *A* who needs to retrieve some information from several legacy application subsystems in his company.

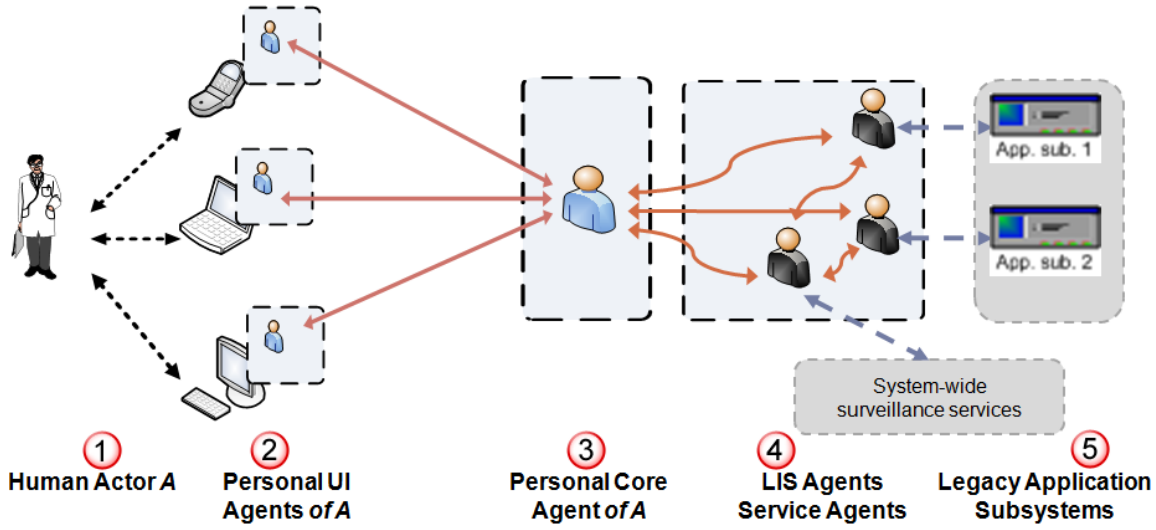

Figure 4.3: *Organizational Model of the Multi-Agent Subsystem*

4.3.1 Personal Agents

A human actor *A* can have simultaneously several personal agents.

One of these personal agents is called the **personal core agent** which is an instance of the **personal core agent class** (see Figure 4.3 ③). There must exist one and only one personal core agent in the model for each human actor.

The other personal agents of human actor *A* are called **personal user interface (ui) agents** which are instances of the **personal ui agent class** (see Figure 4.3 ②). *A* can be assisted by as many personal ui agents as needed, for example, one working on his laptop, one on his smart phone, and another one on his mobile phone.

All personal ui agents of human actor *A* work with a single associated personal core agent.

4.3.2 LIS Agents

Each legacy subsystem has its own representative, called **lis agent** that is an instance of the **lis agent class** (see Figure 4.3 ④).

4.3.3 Service Agents

Each system-wide surveillance activity is carried out by an assigned **service agent** that is an instance of the **service agent class** (see Figure 4.3 ④).

The operational characteristics of agents in this organizational model are presented in the next section.

4.4 Operational Characteristics of Agents in the Multi-Agent Subsystem

4.4.1 Personal UI Agents

A personal ui agent:

- resides on a fixed or mobile personal computer device owned by a human actor;
- has a user interface which can be audio or graphical to communicate with human actors;
- has short life, because the agent is released, for example, every time a user disconnects from a legacy information system or turns off his personal computer device;
- does not need to memorize its state of knowledge (e.g., tasks to do, completed tasks, etc.) due to the limited resource of the user's mobile device.

When alive, a personal ui agent:

- receives the requests from the human actor,

- forwards them to the personal core agent for further processing,
- delivers the results to the human actor.

4.4.2 Personal Core Agents

A personal core agent:

- resides on a computer that offers the possibility to communicate with the personal ui agents and with the lis agents;
- has a relatively long operational life for several reasons:
  - it should be ready anytime round the clock to serve its associated personal ui agents;
  - the computer where it resides is purposefully installed to run with minimum downtime (e.g., maintenance, crash, etc.);
- needs to memorize its state of knowledge (e.g., tasks to do, completed tasks, etc.) to operate efficiently.

A personal core agent processes the requests received from its personal ui agents. To this end, a personal core agent:

- searches a directory of available services registered by agents to discover those lis agents or personal core agents who can provide the information or service to answer the requests;
- sends a query to the discovered agents to get the results;
- finally returns the results to its personal ui agents.

4.4.3 LIS Agents

A lis agent:

- resides on a server hosting the associated legacy information system;
- has a relatively long operational life for several reasons:
  - it should be ready anytime round the clock to serve the personal core agents and service agents;
  - the server where it operates is purposefully installed to run with minimum downtime (maintenance, crash, etc.);
- needs to memorize its state of knowledge (e.g., tasks to do, completed tasks, etc.) to operate efficiently.

A lis agent:

- receives queries from the personal core agents and service agents,
- processes the queries by addressing the associated legacy subsystem,
- returns the information to the requesters.

4.4.4 Service Agents

A service agent:

- resides on a server hosting system-wide surveillance tasks;
- has a relatively long operational life for several reasons:
  - it should be active round the clock to fulfill system-wide surveillance tasks, and to contact any agent at anytime if necessary;
  - the server where it operates is purposefully installed to run with minimum downtime (maintenance, crash, etc.);
- needs to memorize its state of knowledge to operate efficiently.

Each service agent executes its assigned task, for example:

- monitoring the system,
- logging the system activities,
- sending alert notification,
- making reports to personal core agents.

A service agent may send queries to lis agents to obtain information required to produce notification and reports.

4.4.5 Communication between Agents

The communication between agents is based on the asynchronous exchange of messages. For example, a personal core agent *P1* sends a message to another personal core agent *P2* to request help to perform a task *T*; *P1* suspends the task *T* and can execute other tasks while waiting a reply from *P2*; *P2* can respond to or ignore *P1's* request; if *P2* responds to that request, then *P1* can resume the task *T* with the help of *P2*.

Each message is encoded in a communication language (e.g., KQML, FIPA ACL, KIF, FIPA SL, etc.) using a communication protocol (e.g., FIPA Request Interaction Protocol, FIPA Contract Net Interaction Protocol, etc.) and an ontology known by involved agents.

4.5 MAgIL Framework for Developing the Multi-Agent Subsystems

In Section 3.5 we saw that the agent-based approach can be interestingly applied to a variety of domains. In a given domain, each application has its own MAgIL subsystem as part of its architecture. For example:

- In the healthcare domain, the caregivers (e.g., physicians, lab technologists, lab directors, etc.) cooperate between themselves in all kinds of healthcare processes. To this end, each healthcare application will have its own MAgIL subsystem which might consist of:
    - personal agents to assist physicians, lab technologists and lab director;
    - lis agents to communicate with the legacy healthcare application subsystems;
    - and service agents to notify and alert the availability of lab results, to capture events that happen inside the system.

  Chapter 7 of this thesis will present in depth a concrete MAgIL subsystem called MediMAS, as a case study in this domain.
- In the academic domain, members of a university community (e.g., professors, assistants, students, etc.) use information retrieval applications to find and retrieve information, for instance courses, books, research papers, etc., and to exchange information between themselves. In this environment, the MAgIL subsystem might consist of:
    - personal agents to assist professors and students;
    - lis agents to communicate with the legacy retrieval information application subsystems;
    - and service agents to monitor system activities.

  A concrete MAgIL subsystem called uniMAS will be discussed in Chapter 8, as a potential research project of our software engineering group.
- In the e-commerce domain, the actors such as customers, merchants, and product suppliers rely on business application subsystems to exchange the information about goods and services. In this environment, the MAgIL subsystem of an application might consist of:
    - personal agents to assist customers, merchants, and product suppliers;
    - lis agents to communicate with the legacy business application subsystems;
    - and service agents to monitor system activities.

This thesis does not cover the study of concrete MAgIL subsystems for ecommerce applications.

The previous examples by domains show that different MAgIL subsystems have in common the personal agents, the lis agents, and the service agents which are instances of their corresponding classes discussed in section 4.3. These common characteristics suggest the usefulness of having a framework to build efficiently any MAgIL subsystem.

A **MAgIL framework** is defined as a framework with the following features:

- it consists of the personal ui agent class, the personal core agent class, the lis agent class, and the service agent class;
- these agent classes form an organizational model (see section 4.3);
- a concrete multi-agent subsystem can be created from this framework by extension of the agent classes to add specific functionalities of an application (see Figure 4.4).

This extensibility feature is the reason why we consider the MAgIL framework as a semi-finished multi-agent subsystem.

The MAgIL framework unifies the implementation process of concrete multi-agent subsystems, and helps developers save their time and efforts throughout this process.

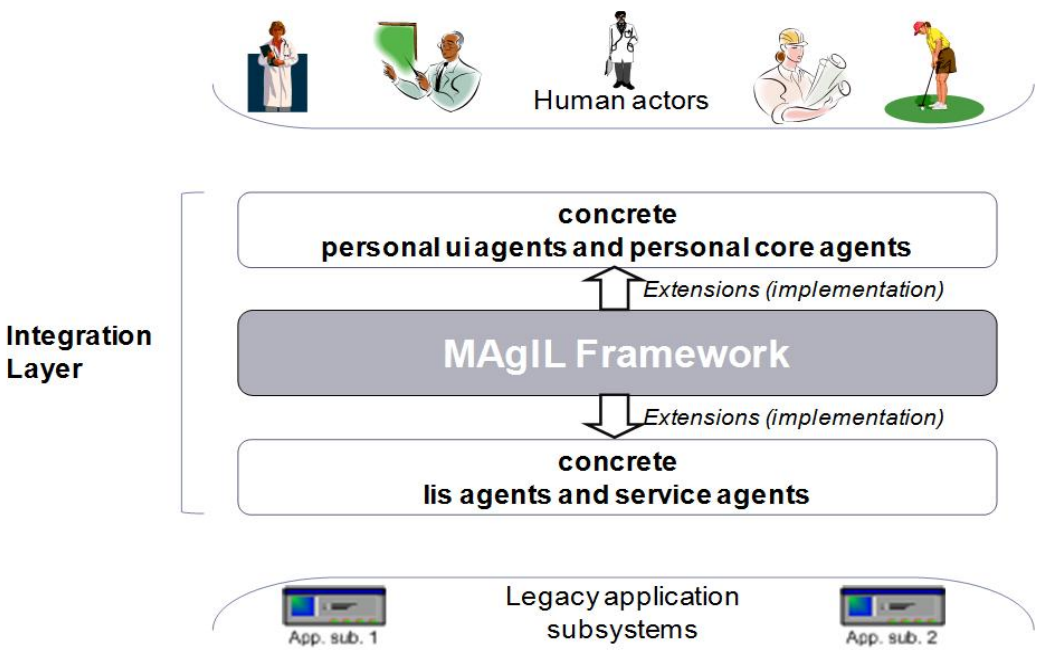

Figure 4.4: *MAgIL Framework and its Extensions into a Concrete Multi-Agent Subsystem at the Integration Layer*

4.6 Conclusion

The present chapter has proposed a software agent-based approach to enhance legacy information systems in a multi-domain multi-application environment. The MAgIL subsystem discussed in this approach plays the facilitator's role for human actors and legacy subsystems.

Using the MAgIL subsystem approach:

- organizations can satisfy users requirements while minimizing changes on legacy information systems;
- human actors can delegate their tedious and routine tasks to personal agents and have more time for focusing on higher level tasks, improving productivity, creating opportunities for innovation and improving their job satisfaction.

The suggested organizational model of agents in the MAgIL subsystem allows human actors to interchange information between themselves and with legacy subsystems anytime and anywhere.

The idea of building a MAgIL framework for MAgIL subsystems is also introduced. The MAgIL framework unifies the implementation process, and helps developers to save time and effort in the implementation phase of concrete subsystems.

The next chapter focuses on the development of this framework: its architectural design, its implementation, and utilization in a concrete application domain.

5

The MAgIL Framework: Bring It to Life with A Concrete Subsystem

In Chapter 4, the MAgIL framework was defined, and considered as a semi-finished extensible agent-based subsystem. The present chapter aims at providing its architecture and bringing that framework to life through a concrete subsystem.

This chapter is structured as follows. The first section focuses on the technological choices. The second section discuses the naming convention for agent classes. The third section presents the architecture of the MAgIL framework, and the framework extensions to create concrete agent-based subsystems. The fourth section presents a simple concrete agent-based subsystem developed from the MAgIL framework.

5.1 Technological Choices

The combination of the following three technologies is chosen to implement the MAgIL framework:

- the agent-based technology with the JADE platform,
- the rule-based technology with the Jess rule engine,
- and the object-oriented technology with the Java programming language.

5.1.1 Agent-based technology with the JADE platform

JADE is acronym of Java Agent DEvelopment framework. It is a set of software programs forming an agent platform written entirely in Java language (Bellifemine et al., 2007). The JADE product is open source. It allows developers to create agents in an agent-based subsystem. The agents use basic services and utilities provided by the platform to work together, for example, agent management services, messaging services, directory and notification services, communication languages, communication protocols, etc.

Thus, using the JADE platform, developers do not need to reinvent the basic services and utilities. Consequently, they can save time and efforts in building agent-based subsystems. This platform presents other advantages:

- JADE is FIPA compliant;
- It can run on a variety of operating systems (e.g., Windows, Mac OS X, Linux, etc.);
- It is extensively used in both research and industrial fields, the reason why developers may benefit from the know-how and experiences of a large community of experts.

5.1.2 Rule-based technology with the Jess rule engine

Jess is acronym of Java Expert System Shell. It is a rule engine written entirely in Java language by Ernest Friedman-Hill at Sandia National Laboratories (Friedman-Hill, 2003).

A Jess rule consists of two parts like the *if-then* statements in plain English. For example, the following rule:

```
(defrule turnon_heater_system
    (< ?room_temperature_celsius 15)
    =>
    (printout t "Please turn on the heater system!" crlf)
)
```

is equivalent to:

if *the room temperature is less than 15 degrees Celsius*
then *display "Please turn on the heater system!"*

The rule's name is *turnon_heater_system*. Syntactically, the left-hand side (LHS) part of the rule expresses the *conditions* to be evaluated against facts (e.g., "`(< ?room_temperature_celsius 15)`"). The right-hand side (RHS) part specifies the *actions* to be taken (e.g., "`(printout t "Please turn on the heater system!" crlf)`").

A Jess rule engine is composed of three components:

- a rule base to store rules;
- a working memory to hold facts;
- an inference engine to decide and execute the appropriate *actions* (RHS) resulting from the evaluation of the conditions (LHS).

The Jess rule engine supports both forward and backward chaining, and uses the Rete algorithm to increase the inference speed.

In agent-based subsystems, an agent linked to a Jess rule engine behaves according to the rules specified by a human actor. We say that the agent is dotted with intelligence in assisting the actor.

5.1.3 Object-oriented technology with the Java programming language

Java is an object-oriented programming language (Sun Microsystems, 1994). A program containing classes of objects coded in Java may be reused, compiled, and ported to different computer systems (e.g., PC, mobile devices, etc.). Therefore, with the Java programming language, an agent will be coded as an object which is a piece of reusable and portable software.

Java programmers may benefit from a uniform development environment in which the same programming language, Java, is used by both JADE platform and Jess rule engine.

5.2 Naming Conventions for Agent Classes

In Section 4.4, we have defined the classes of agents, their roles, and their organizational model. This section provides naming conventions for these classes which will be used later to design and implement the architecture of the MAgIL framework.

- An **abstract agent class**, named **AbstractAgent**, is a superclass containing the common characteristics of all subclasses called role-specific agent classes.
- The names of the **role-specific agent classes** are given in Table 5.1.

Table 5.1: *Names of Role-Specific Agent Classes*

Agent classes	Names
personal ui agent class	**PersonalUIAgent**
personal core agent class	**PersonalCoreAgent**
lis agent class	**LISAgent**
service agent class	**ServiceAgent**

5.3 Layered Architecture of MAgIL Subsystems

In Section 4.5, we have defined the MAgIL framework and discussed its use in developing MAgIL subsystems. In the present section, we will study the design of MAgIL subsystems as a three-layer architecture shown in Figure 5.1.

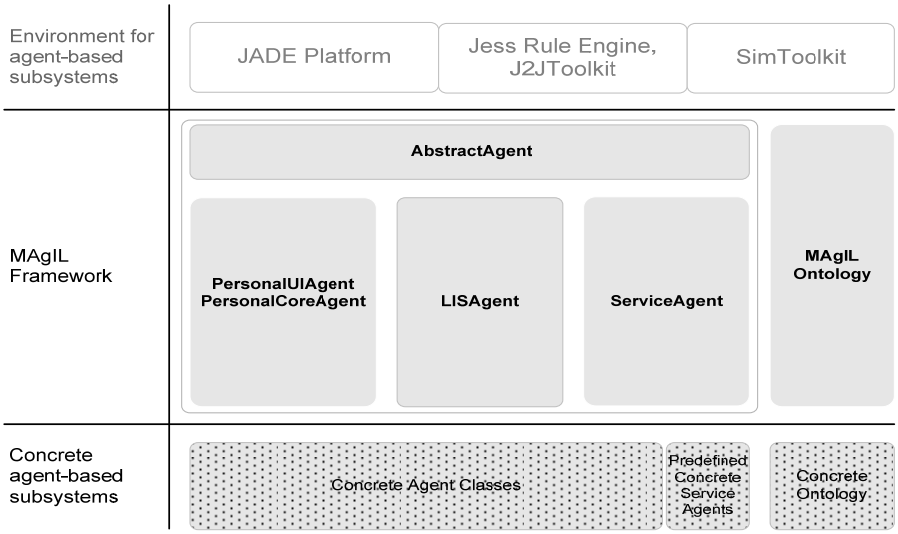

Figure 5.1: *The three layers of MAgIL Subsystems*

5.3.1 Layer 1: Environment for agent-based subsystems

Layer 1 defines the development and runtime environment for MAgIL subsystems. According to our technological choices, this environment comprises the JADE platform and the Jess rule engine.

J2JToolkit (Vogt, 2008) contains libraries allowing the developers to plug the Jess rule engine into an agent class. As a result, the agent class acquires the inference capability to behave in compliance with its own set of rules. Jess declarative language is used to edit these rules in an external file which will be loaded by agents at runtime. Rules can be updated by human actors at anytime and take effect instantaneously either by reloading the external file or by sending a request-to-update message to agents. This declarative approach with Jess is an alternative to the coding of rules by Java procedural language.

A simulation software module was developed by Pointet (2008) to simulate an agent-based

application project, called MediMASim, in the healthcare domain. In this thesis, the original simulation module was separated from MediMASim and retooled as a software product called SimToolkit which can then be used to simulate any MAgIL subsystem. The developers use SimToolkit to assess the working of MAgIL subsystems in compliance with their objectives. For example, in a MAgIL subsystem whose purpose is to manage the medical test orders in a laboratory of a hospital, SimToolkit allows the developers to study the distribution of agents on human actor's PCs, laptops, PDAs, etc. and the interactions between human actors and agents. SimToolkit can also create agents to simulate human actors themselves. The simulation model and scenarii are specified in an input XML file which must conform to the XML Schema provided by SimToolkit. The XML Schema determines the constraints and data types of elements in the input XML document and its structure. Appendix B contains the SimToolkit's XML Schema file and an input XML file pertaining to the case study HelloMAS discussed in the next section.

In Chapter 6, we will study how to build the MAgIL framework from JADE and Jess.

5.3.2 Layer 2: MAgIL Framework

Layer 2 defines the MAgIL framework itself which consists of two building blocks. The first building block contains the abstract agent class and its three following subclasses known as role-specific agent classes:

- personal core agent class and personal ui agent class,
- lis agent class,
- service agent class.

In Figure 5.1, the first building block is presented using the naming conventions for agent classes from Section 5.2.

The second building block contains MAgILOntology. This ontology defines a set of vocabularies to represent the knowledge about the agent classes in the MAgIL framework and their relationships with human actors, legacy information systems (LISs), and services. MAgILOntology as a common language allows the efficient communication between developers and across applications during the software development process. MAgILOntology as a component of agent-based subsystems, allows agents to understand each other during runtime.

Firstly, MAgILOntology consists of the following hierarchy of basic concepts. Each concept is characterized by its semantics and attributes:

- the concept of human actor;
- the concept of legacy information system (LIS);

- the concept of service;
- the concept of agent, further specialized into three subconcepts: the personal agent, the lis agent, and the service agent;
- the personal agent subconcept is specialized into two concepts at lower level: the personal core agent and the personal ui agent;
- the concept of agent's action.

Secondly, MAgILOntology contains three relationships between concepts:

- the relationship between the personal agent and the human actor which states that one or more personal agents may work for a human actor;
- the relationship between the lis agent and the legacy information system (LIS) which states that one or more lis agents may exist to represent a LIS;
- the relationship between the service agent and the service concept which states that a service is provided by one service agent.

5.3.3 Layer 3: Concrete agent-based subsystems

Layer 3 defines the structure of a concrete agent-based subsystem implemented as an extension of the MAgIL framework.

By extension we mean that each agent class in the MAgIL framework can be concretized into one or more subclasses with extended attributes and behaviors required by their specific roles or functions in a concrete agent-based subsystem. For example, as shown in Figure 5.2, the concrete personal core agent classes represent distinct extensions of the personal core agent class in the framework, the concrete service agent classes are distinct extensions of the service agent class, etc.

In Figure 5.2, along with the MAgIL framework, some concrete agent classes can be fully predefined in Layer 3, because they are practically required in most of real-world applications. Those predefined concrete agent classes are ready to be instantiated without any further extensions by developers. For example, the concrete audit agent class may be a predefined concrete service agent ready to be used in any application.

Finally, a concrete ontology can be created by extending MAgILOntology through the specialization of existing concepts as required by the given application domain.

Chapter 7 will discuss a concrete ontology for an application in the healthcare domain.

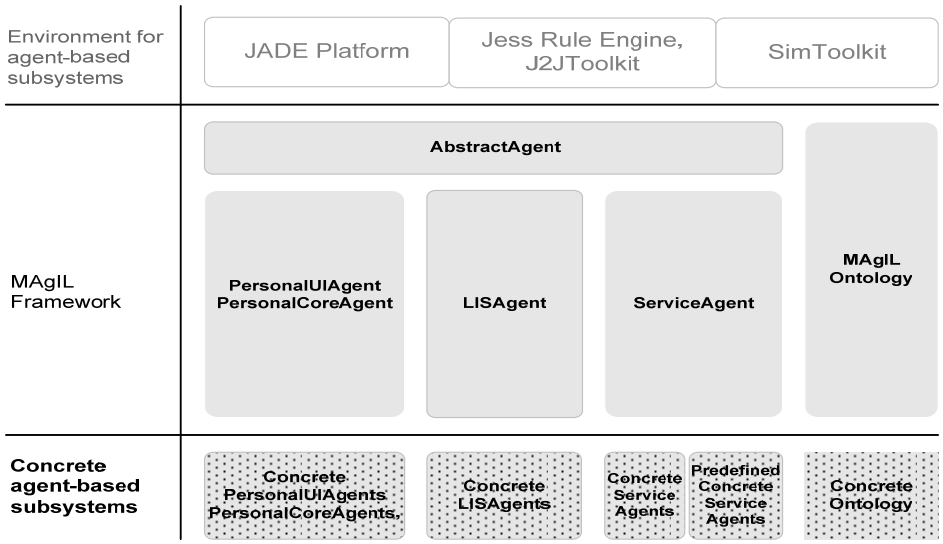

Figure 5.2: *A Concrete Agent-Based Subsystem as Extension of the MAgIL Framework*

5.4 A Concrete Agent-Based Subsystem: HelloMAS

HelloMAS is a small case study involving three human actors exchanging greeting messages. The present section explains:

- how the HelloMAS subsystem is built from the MAgIL framework, and
- how it works.

5.4.1 HelloMAS Subsystem

Despite its simplicity, HelloMAS contains the basic ingredients of a multi-agent subsystem (see Figure 5.3):

- a concrete personal ui agent class, named HelloPersonalUIAgent, obtained by extending the PersonalUIAgent class from the MAgIL framework;
- a concrete personal core agent class, named HelloPersonalCoreAgent, obtained by extending the PersonalCoreAgent class from the MAgIL framework;
- a predefined audit agent class, AuditAgent;
- a HelloMASOntology concretized by extending the MAgILOntology from the MAgIL framework.

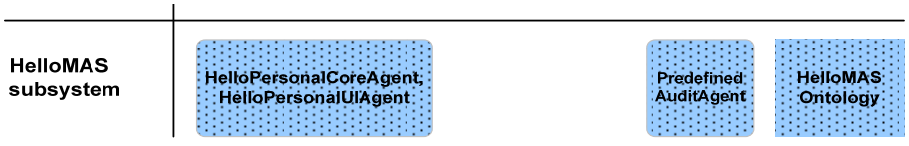

Figure 5.3: *HelloMAS Subsystem as Extension of the MAgIL Framework*

The instances of HelloPersonalCoreAgent and HelloPersonalUIAgent will cooperate to help human actors exchange their greeting messages, using a set of vocabularies in HelloMASOntology. An instance of AuditAgent will log all activities of the personal core and ui agents.

5.4.2 The Working of HelloMAS Subsystem

5.4.2.1 HelloMAS Environment Setup

The HelloMAS environment consists of two parts: the server side and the client side. The three human actors on the client side are Vesper Lynd, James Bond, and Felix Leiter.

Server side setup

An agent-based environment for HelloMAS must be setup on a dedicated server by starting:

- the JADE platform,
- and the predefined audit agent.

Also at environment setup, three personal core agents are instantiated from HelloPersonalCoreAgent on the same dedicated server, one for each human actor:

- Vesper Lynd's personal core agent, named *LyndCoreAgent*;
- James Bond's personal core agent, named *BondCoreAgent*;
- Felix Leiter's personal core agent, named *LeiterCoreAgent*.

One attribute of each personal core agent instance contains the pre-entered personal profile of the corresponding human actor (pid, first name, last name, gender, age).

One behavior of each personal core agent instance follows the rules upon receipt of a greeting request, which can be stated in plain English as follows:

```
If the personal core agent detects a greeting request from its corresponding human actor called the initiator, then it creates a greeting message with the following content: "Hello, my name is [Initiator's last name], [Initiator's full name].", and dispatches it to other personal core agents.
```

Another behavior follows the rules upon receipt of a greeting message:

```
If a greeting message is detected as coming from a personal core agent associated with a female initiator, then acknowledge receipt by replying to that personal core agent as follows, using "Mrs." title: "Hello Mrs. [Initiator's last name], nice to receive your greeting which will be forwarded to [Responder's last name], [Responder's full name]."
```

```
If a greeting is detected as coming from a personal core agent
associated with a male initiator, then acknowledge receipt by replying
to that personal core agent as follows, using "Mr." title: "Hello Mr.
[Initiator’s last name], nice to receive your greeting which will be
forwarded to [Responder's last name], [Responder's full name]."
```

The above two behaviors illustrate how the personal core agents are set up to work autonomously and cooperate between them on behalf of human actors without their intervention.

To complete the personal core agents’ setup process, a request for registration in the Services Directory will be sent by each started personal core agent to the Directory Facilitator (also known as *yellow pages agent*) of the JADE platform. The registration data include at minimum: the personal core agent’s identifier, its host’s IP address, and the personal profile of the corresponding human actor.

After registration, the three personal core agents are ready to cooperate between themselves round the clock to process the greeting messages according to the behavior rules described above.

Client side setup

Vesper Lynd, James Bond, and Felix Leiter set up their own computer devices before sending and receiving greetings:

- a mobile phone for Vesper Lynd,
- a desktop PC for James Bond,
- a smart phone for Felix Leiter.

At some moment of the day, Vesper Lynd launches the HelloMAS client software on her mobile. A personal ui agent, called *LyndUIAgent*, immediately starts on her device and attempts to establish a first contact with *LyndCoreAgent*.

To this end, *LyndUIAgent*:

- collects Vesper Lynd’s profile (pid, first name, last name, gender, age) stored in the mobile as shown in Figure 5.4 (1),
- and requests the Directory Facilitator to scan the Services Directory searching for a registered personal core agent having a matching profile (Figure 5.4 (2)).

Evidently, the Directory Facilitator will:

- discover *LyndCoreAgent* (registered during server side setup) (Figure 5.4 (3)),
- return *LyndCoreAgent*’s identifier and IP address to *LyndUIAgent* (Figure 5.4 (4)).

Using these two pieces of information, *LyndUIAgent* sends to *LyndCoreAgent* a request to enter the list of known personal ui agents maintained by *LyndCoreAgent* (Figure 5.4 (5)).

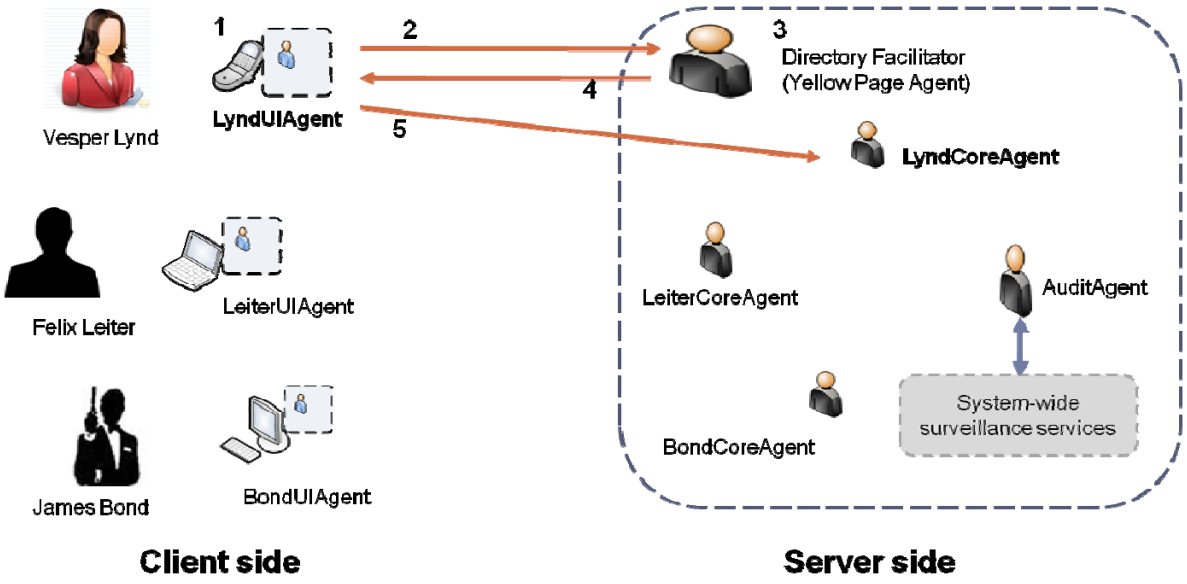

Figure 5.4: *LyndUIAgent's Setup*

In a similar manner, anywhere at anytime James Bond and Felix Leiter can launch their respective HelloMAS client software on their personal computer devices. *BondUIAgent* and *LeiterUIAgent* will start and attempt to register with *BondCoreAgent* and *LeiterCoreAgent*, respectively.

In Figure 5.5, after the successful client setup the name of each personal ui agent is displayed in the title bar of the user interface (Ⓐ Ⓑ Ⓒ) on the corresponding human actor's computer device.

5.4.2.2 HelloMAS Agents in Action and Rules Processing

Human actors' view

To send a greeting message to everybody, Vesper Lynd selects and activates the `Greet Everybody` command from the pop-up menu in the bottom right of her mobile's screen, as shown in Figure 5.5①.

Vesper Lynd receives instantaneously two "acknowledgement of receipt" messages from Bond's and Leiter's core agents on behalf of James Bond and Felix Leiter (Figure 5.5②).

If the computer devices of James Bond and Felix Leiter are online, the greeting message coming from Vesper Lynd will appear on their display (Figure 5.5③).

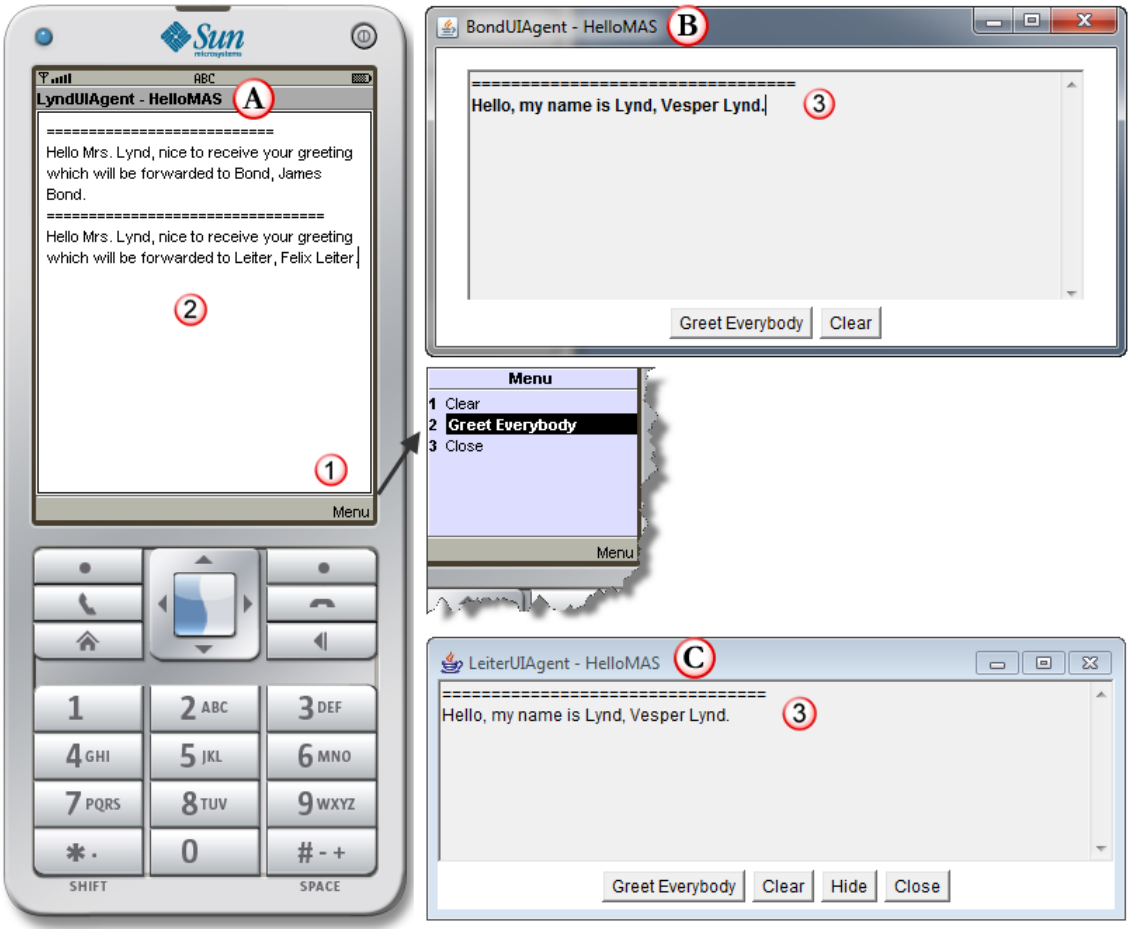

Figure 5.5: *HelloMAS Agents in Action*

View behind the scenes

Now, let's focus on the agents' actions through a sequence of steps numbered from 1 to 8 in Figure 5.6.

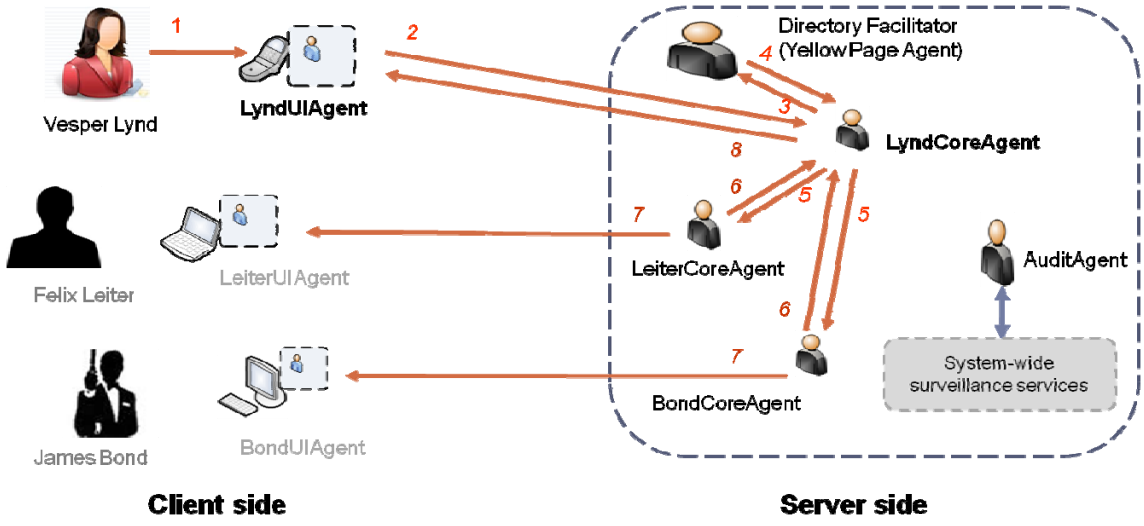

Figure 5.6: *Vesper Lynd Greets Everybody*

When Vesper Lynd activates the `Greet Everybody` command, as shown in Figure 5.6 (1):

- *LyndUIAgent* transmits to *LyndCoreAgent* a request containing the "GreetAction" concept (Figure 5.6 (2)).
- *LyndCoreAgent* interprets the request based on the HelloMASOntology, in particular the "GreetAction" which means that a greeting message must be created and dispatched to other personal core agents.

To perform "GreetAction":

- *LyndCoreAgent* asks the Directory Facilitator to provide a list of all personal core agents retrieved from the Services Directory (Figure 5.6 (3)).
- The Directory Facilitator returns to *LyndCoreAgent* the requested list containing *BondCoreAgent* and *LeiterCoreAgent* (Figure 5.6 (4)).
- *LyndCoreAgent* creates a greeting message using the following format:

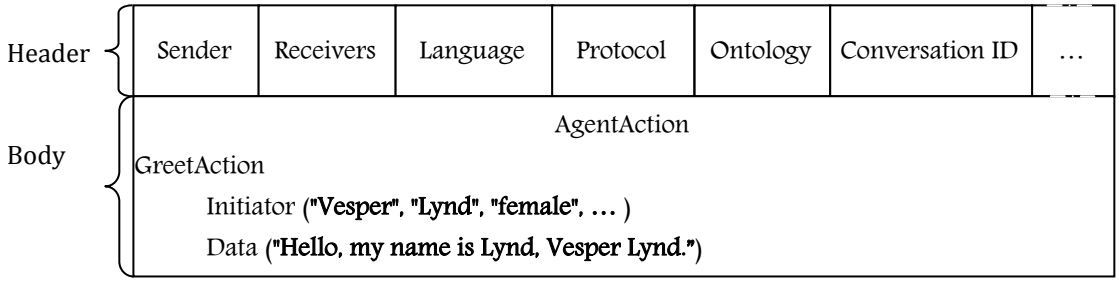

Figure 5.7: *The structure of Vesper Lynd's greeting message*

The data part is determined by the initiator's behavior rule for greeting.

- *LyndCoreAgent* sends the greeting message to *BondCoreAgent* and *LeiterCoreAgent*, as shown in Figure 5.6 (5).
- *BondCoreAgent* and *LeiterCoreAgent* interpret the greeting message based on HelloMASOntology. The concept "GreetAction" is analyzed and understood as a greeting from Vesper Lynd.
- *BondCoreAgent* and *LeiterCoreAgent* send their "acknowledgement of receipt" messages to *LyndCoreAgent* (Figure 5.6 (6)) using the following format.

- James Bond's "acknowledgement of receipt" message:

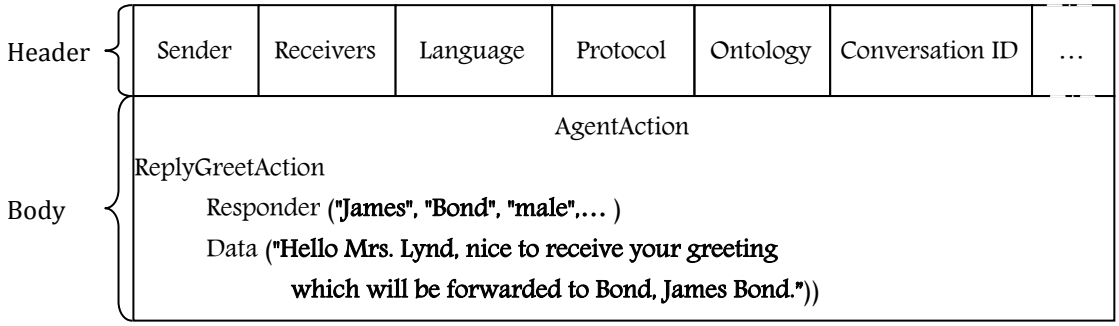

Figure 5.8: *The structure of James Bond's acknowledgement of receipt message*

- Felix Leiter's "acknowledgement of receipt" message:

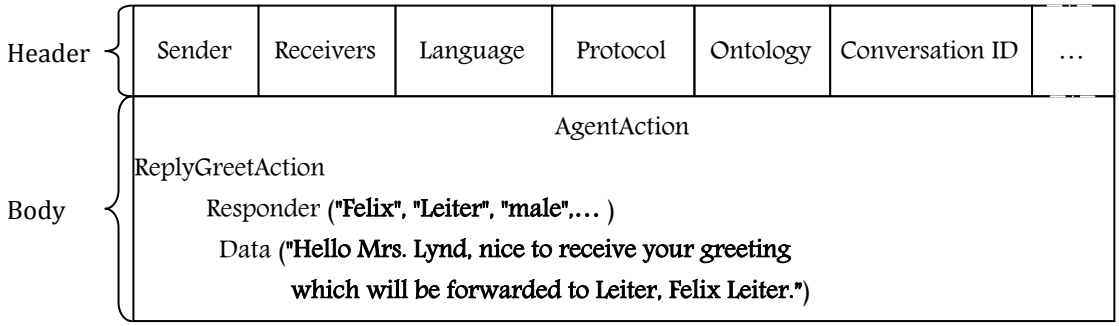

Figure 5.9: *The structure of Felix Leiter's acknowledgement of receipt message*

The data part is determined by the responder's behavior rule for acknowledging receipt.

- *BondCoreAgent* and *LeiterCoreAgent* forward the greeting message to *BondUIAgent* and *LeiterUIAgent*, respectively (Figure 5.6 (7)).
- *BondUIAgent* and *LeiterUIAgent* interpret the greeting message based on the HelloMASOntology to display its data part (see Figure 5.7) on James Bond's and Felix Leiter's computer devices, respectively.
- *LyndCoreAgent* interprets the ReplyGreetAction in the acknowledgement of receipt messages from *BondCoreAgent* and *LeiterCoreAgent* and forwards both of them to *LyndUIAgen*t (Figure 5.6 (8)).
- *LyndUIAgent* displays the data part of the ReplyGreetAction concept (see Figure 5.8 and Figure 5.9) on Vesper Lynd's mobile phone.

The following section presents the possibility of customizing rules and how the agents adjust their behavior to satisfy the changing requirements of human actors.

Rules Customization

One day, James Bond would like to replace his initial acknowledgement of receipt:

> `"Hello Mrs. [Initiator's last name], nice to receive your greeting which will be forwarded to [Responder's last name], [Responder's full name]."`

with the following statement if the greeting message comes from a lady:

> `"Hello Mrs. [Initiator's last name], Mr. Bond will try to contact you later."`

At setup on the server side, *BondCoreAgent* was created on the server along with its behavior's rules. Therefore, in this HelloMAS version, Bond must explain his request for change to the server's system administrator.

The following steps will be performed by the system administrator to customize the "acknowledgement of receipt" rule of *BondCoreAgent*:

- Launch the *JessAdminUIAgent* software. A user interface will appear on the administrator's console (Figure 5.10 ①).
- Enter exactly the lines of code, written in the Jess language, in the left panel of the user interface (Figure 5.10 ②).

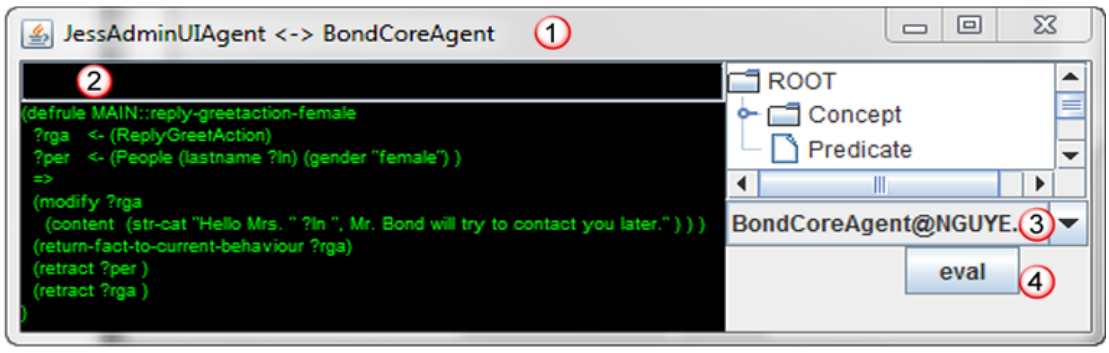

Figure 5.10: *Editing Rule and Assigning It to BondCoreAgent*

Let's examine in depth the above code:

- Line 1: define the rule whose name is `reply-greetaction-female`.
- Line 2: save the pointer to the fact containing `ReplyGreetAction` concept into the variable `rga`.
- Line 3: save the pointer to the fact containing People concept into the variable `per`.

- Lines 5 and 6: change the data part of the `ReplyGreetAction` concept to the new acknowledgement of receipt.
- Line 7: return the changed concept to the current behavior of *BondCoreAgent*.
- Lines 8 and 9: remove the pointers from the working memory.

- Select *BondCoreAgent* from the drop-down list in the lower right panel (Figure 5.10③).
- Click on *eval* button to fire the rule (Figure 5.10④).

Effects of Rules Customization

Let's see what happens from the human actor's view when Vesper Lynd greets everybody after Bond's rule customization.

Vesper Lynd's greeting message is displayed on the computer devices of Bond and Leiter (Figure 5.11ⒷⒸ).

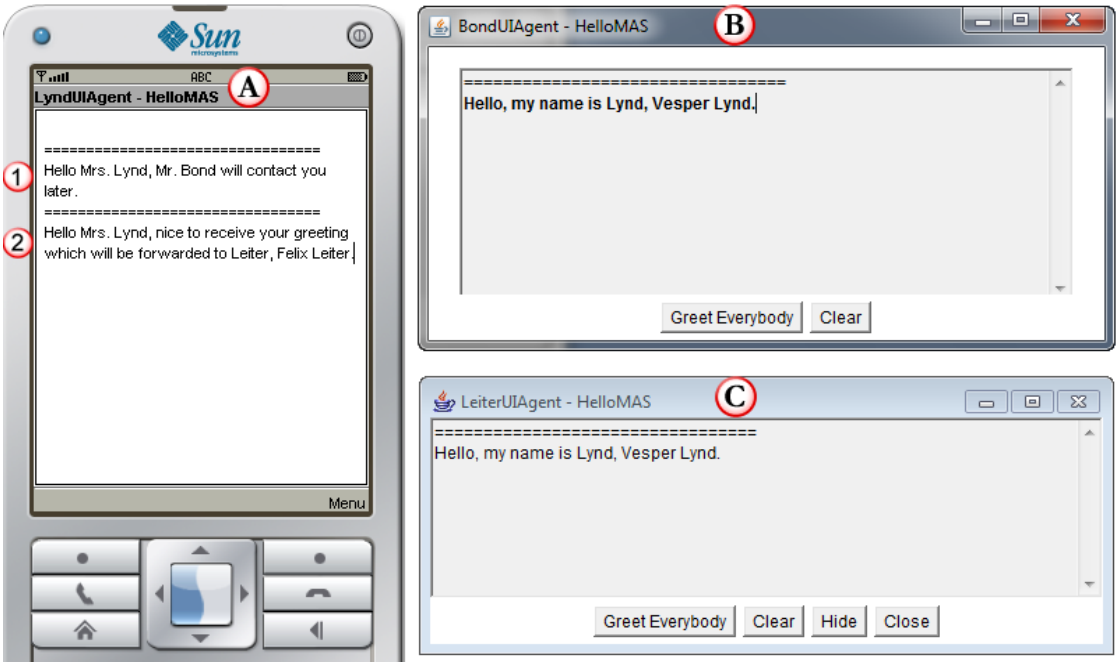

Figure 5.11: *HelloMAS Agents in Action – After Rule Customization*

The effect of Bond's rule customization may be seen in Figure 5.11Ⓐ. Bond's acknowledge of receipt (Figure 5.11①) now reads:

```
Hello Mrs. Lynd, Mr. Bond will try contact you later.
```

which is different from that of Leiter (Figure 5.11②):

```
Hello Mrs. Lynd, nice to receive your greeting message which will be
forwarded to Leiter, Felix Leiter.
```

At the end of this chapter, we now have acquired the basic knowledge about the architecture of MAgIL subsystems, the MAgIL framework, and the concrete HelloMAS agent-based subsystem built from that framework. We also learn how agents in a concrete agent-based subsystem communicate between them and interact with human actors. We are ready to go a step further to explore, in the next chapters, the technical aspects of the MAgIL framework and its use to develop a more complex application in the e-healthcare domain.

6

The MAgIL Framework: Design and Implementation

Chapter 5 has provided the three-layered architecture of MAgIL subsystems: agent-based subsystem's environment layer, MAgIL framework layer, and concrete agent-based subsystem layer. The present chapter aims at discussing the design and implementation of the MAgIL framework in the second layer of the architecture.

This chapter is structured as follows. The first section presents the design of the framework. The second section dicusses its implementation. Finally, the third section focuses on how the framework components are assembled into Java packages.

6.1 The MAgIL Framework Design

The MAgIL framework is software. Its design is based on the layered and tiered architecture and the object-oriented approach. This section elaborates on these two designed aspects.

6.1.1 Layered and Tiered Architecture of the MAgIL Framework

In Chapter 5, we defined MAgIL subsystems as a three-layered architecture (see Figure 5.1). Inside Layer 2 of this architecture, the MAgIL framework will be further divided into two sublayers (see Figure 6.1):

- *MAgIL Framework sublayer 2.1*: the abstract agent class, AbstractAgent;
- *MAgIL Framework sublayer 2.2*: the role-specific agent classes which are extensions of AbstractAgent.

Figure 6.1: *The two sublayers of the MAgIL Framework*

From the sublayers we now define three tiers. Each tier is made up of the abstract agent class from sublayer 2.1 and an appropriate number of role-specific agent classes selected from sublayer 2.2 as shown in Figure 6.2.

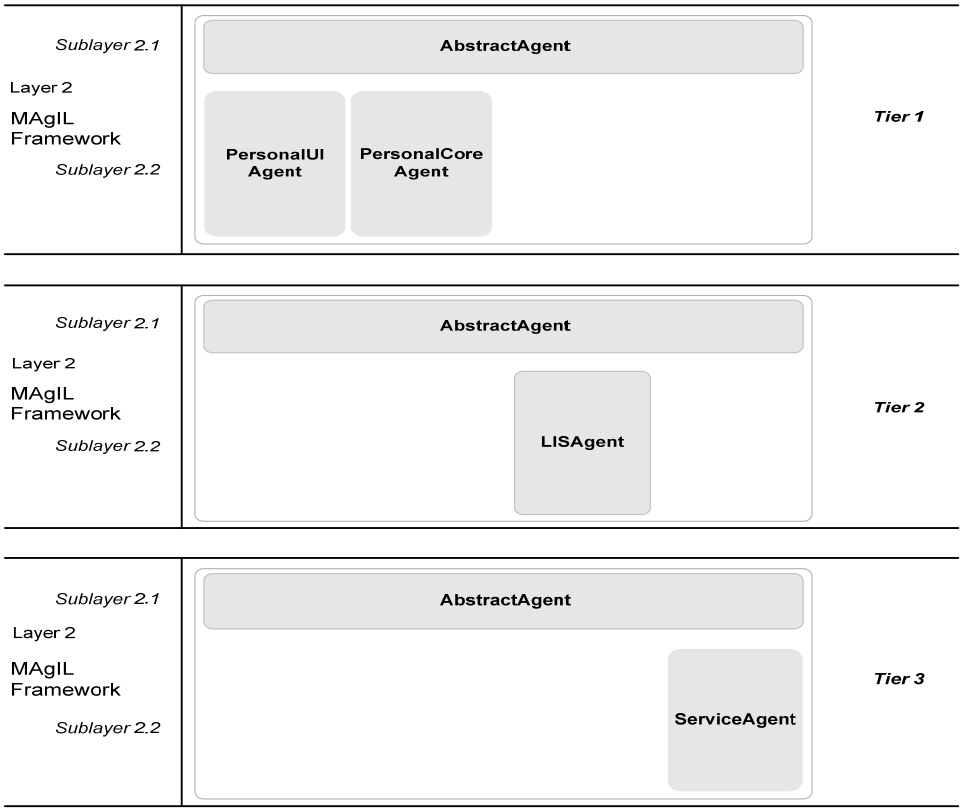

Figure 6.2: *The three tiers of the MAgIL Framework*

- *Tier 1* consists of AbstractAgent and two role-specific agent classes, PersonalUIAgent and PersonalCoreAgent;
- *Tier 2* consists of AbstractAgent and one role-specific agent class, LISAgent;
- *Tier 3* consists of AbstractAgent and one role-specific agent class, ServiceAgent.

All the agent classes share a common MAgILOntology which is therefore an important component of the MAgIL framework architecture (Figure 6.3).

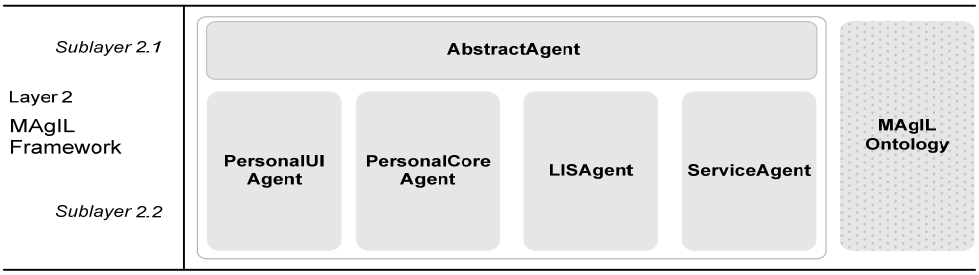

Figure 6.3: *MAgILOntology as part of the MAgIL Framework*

The three tiers and MAgILOntology are generally distributed on different computers according to the specific role of agents. For example, Tier 1's personal agents are implemented on computers called clients, Tier 2's and Tier 3's agents on computers called servers. Clients and Servers form a network of a MAgIL agent-based subsystem.

The layered and tiered architecture of the MAgIL framework and its distributed implementation comply with the layered application guidelines defined by Microsoft Patterns & Practices Team (2009):

> "*Layers* describe the logical groupings of the functionality and components in an application, whereas *tiers* describe the physical distribution of the functionality and components on separate servers, computers, networks, or remote locations."

6.1.2 Object-Oriented Design of Agent Classes

In the MAgIL framework, each agent class, either abstract or role-specific, is until now being considered as a black-box. Now, this section discusses its internal design.

6.1.2.1 Abstract Agent Class

The abstract agent class, **AbstractAgent**, is designed as an object class in the object oriented system. Its design is based on a pattern called *template method* in (Gamma, Helm, Johnson, & Vlissides, 1994) which defines a set of attributes and initialization methods for the abstract agent class, in particular, the following ones:

- an attribute with a type of **AbstractAgentState** which is an object class describing the states (e.g., initiated, active, idle, suspended, waiting, etc.) of the abstract agent class

and its relationship with human actors, legacy subsystems, system-wide surveillance services, etc., that will be specified later in role-specific agent classes;

- a setup method which contains calls to a number of operations:
  - the *hook operations* on the one hand, i.e., methods implemented by default and customizable in subclasses to initialize the agent's state, ontologies, and communication languages;
  - the *primitive operations* on the other hand, i.e., abstract methods which must be implemented in subclasses to attach appropriate behaviors to the agent according to its specific role.

A behavior in the agent concept is a set of actions defining how concrete agents (i.e., instances of the subclasses of the abstract agent class) operate to respond to a given event. There are five types of behaviors:

- *One shot behavior* presents the actions performed by agents only once in a session or a period of time. For example, agents prepare and send a notification to another agent. This behavior is loaded into the agents' memory when needed and unloaded automatically as soon as the actions are completed.
- *Cyclic behavior* presents the actions repeatedly executed by agents in their lifecycle. For example, the cyclic "(1) wait until being notified → (2) receive a notification" behavior is loaded once into the agents' memory, and cycles (loops) until it is terminated or unloaded by agents.
- *Ticker behavior* presents the actions whose execution by agents must follow a specific timing. For example, agents check and update their members list every 5 minutes. Once loaded into the agents' memory, this kind of behavior is activated at scheduled time until it is unloaded by agents.
- *Request initiator behavior* presents the actions executed by agents to request something (e.g., a service, an information, etc.) from another agent. This request is performed according to a communication protocol called FIPA-Request-Interaction-Protocol (cf. Section 3.3.3.3). The behavior is loaded when needed and unloaded automatically as soon as agents receive the result of the request.
- *Request responder behavior* presents the actions executed by agents in order to respond to a request issued by the request initiator behavior. This behavior is loaded once into the agents' memory and resides in it until it is unloaded by agents.

These five types of behaviors are designed as five separate template object classes in a library called “behaviors”. Each behavior type follows the *template method* pattern discussed above.

Diagrammatically, the abstract agent class and its state class are linked as shown in Figure 6.4. The behavioral classes (types) stand alone in the diagram. Their connection with the abstract agent class will be studied in our discussion about role-specific agents in the next sections.

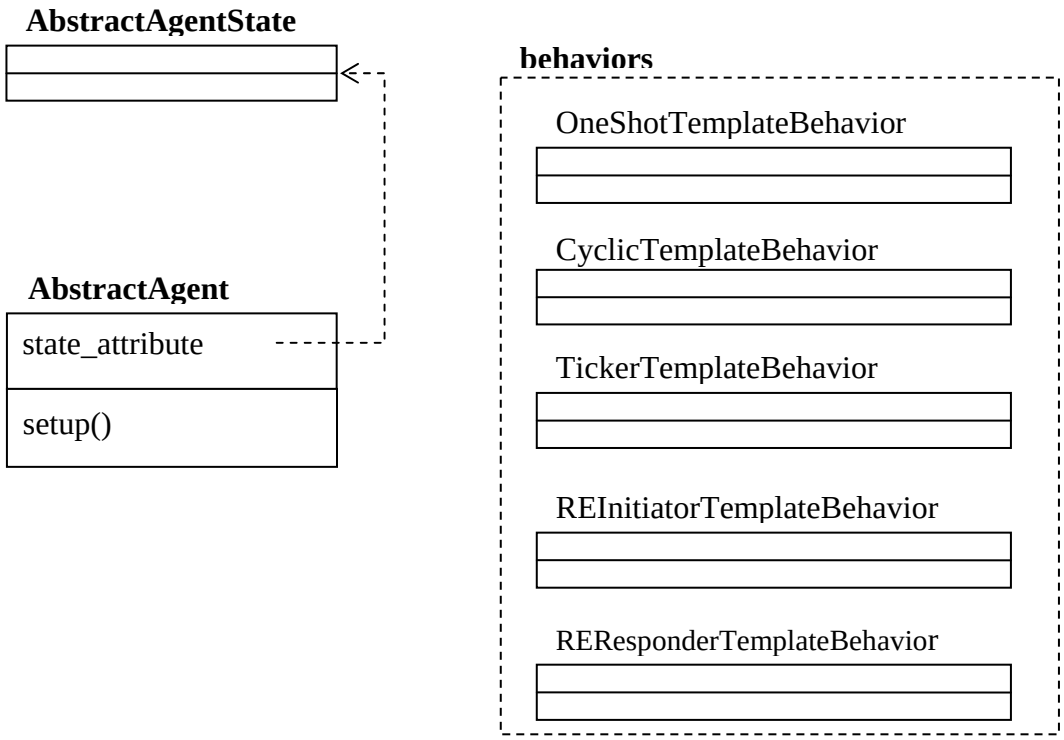

Figure 6.4: *Structure of the Abstract Agent Class*

The role-specific agent classes, such as personal ui agent class, personal core agent class, lis agent class, and service agent class, are designed as subclasses of the abstract agent class. Therefore, they will have the similar structure as the abstract agent class discussed above.

6.1.2.2 Personal UI Agent Class

The personal ui agent class, **PersonalUIAgent**, is made up of attributes and methods based on the structure of AbstractAgent. We focus on two special attributes and the body shown in Figure 6.5:

- *“state_attribute” (inherited from AbstractAgent)* specifies the properties of the personal ui agent class: its owner (human actor) and its personal core agent identifier defined in **PersonalUIAgentState** which is a subclass of AbstractAgentState.
- *“ui_attribute”* is of type **PersonalUI** which is an object class defining the user interface of the personal ui agent class. The purpose of the user interface is to observe

and visualize the changing states of the personal ui agent and to accept command input from the human actor (user). To this end, the PersonalUI class is designed following the *observer* pattern (Gamma et al., 1994). The user interface may be graphical, audio, vibration.

- The body implements the abstract methods (primitive operations) and eventually re-implements the hooks of the abstract agent class in order to add two one-shot behaviors and one request-responder behavior which are explained below:
  - The two *one shot behaviors* are SubscriberBehavior and CancellerBehavior which are subclasses of the OneShotTemplateBehavior class. They specify how to request a personal core agent each time a personal ui agent would like to register into or unregister from the list of members maintained by the personal core agent class discussed in the next section. These two behaviors allow a personal ui agent to request its personal core agent for a registration at the beginning of its life cycle, and a un-registration at the end.
  - The *request responder behavior* is PUIAREResponderBehavior which is a subclass of the REResponderTemplateBehavior class. It specifies how to respond to a request using FIPA-Request-Interaction-Protocol. This behavior allows a personal ui agent to answer any request from other agents during its life cycle.

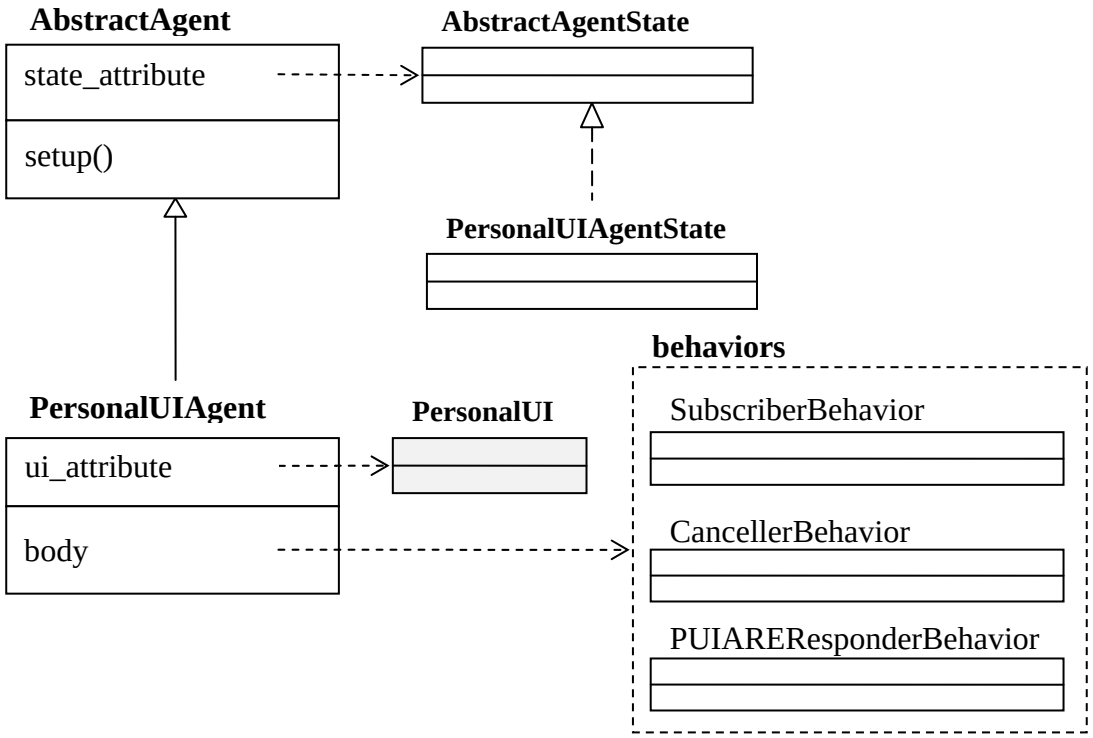

Figure 6.5: *Structure of the Personal UI Agent Class*

6.1.2.3 Personal Core Agent Class

The attributes and methods of the personal core agent class, **PersonalCoreAgent, are also based on the structure of AbstractAgent. More specifically,** as shown in Figure 6.6, it has:

- a *state_attribute* specifying the properties of the personal core agent class: its owner (human actor) and its personal ui agent identifiers defined in **PersonalCoreAgentState** which is a subclass of AbstractAgentState.
- a body implementing the abstract methods (primitive operations) from the abstract agent class to add two following behaviors:
  - a *cyclic behavior*, RegAndUnregBehavior, which is a subclass of CyclicTemplateBehavior. It specifies how a personal core agent fulfills a request for registration into and un-registration from its list of members during its lifetime. The requests come from the personal ui agents.
  - a *request responder behavior*, PCoreAREResponderBehavior, which is a subclass of REResponderTemplateBehavior. It describes how a personal core agent responds to a request from other agents using FIPA-Request-Interaction-Protocol between them.

  In addition to the methods from the AbstractAgent class, the body may have other methods which are specific to personal core agents such as a method to lookup another personal core agent, a legacy information system or a service agent, to inform personal ui agents. These specific methods are not presented in Figure 6.6.

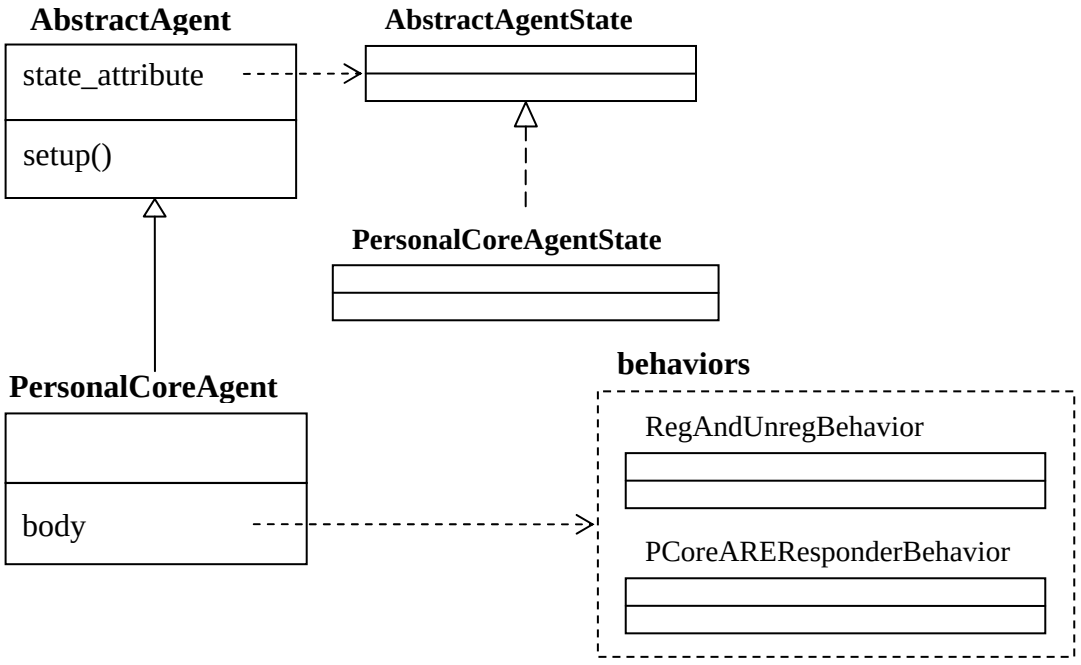

Figure 6.6: *Structure of the Personal Core Agent Class*

The way by which the personal core agent class and the personal ui agent class cooperate is to some extent similar to the cooperation between the subject and observer classes in the *observer pattern* presented by Gamma et al. (1994). In fact, there are similarities and differences in comparing the design of the above two agent classes and that of the observer pattern:

- Similarities: when the personal core agent's state changes, its personal ui agents are informed and update their states correspondingly, as well as their user interfaces. This process is similar to the one which allows a subject to notify the observers which will then update their states.
- Differences: the inform message sent by the personal core agent contains its state in plain text which will be used directly by the personal ui agents. In contrast, the subject's notification does not include its state. Therefore, the observers must call back the subject to get its state.

6.1.2.4 LIS Agent and Service Agent Classes

In a similar manner, the lis agent and service agent classes are designed as shown in Figure 6.7 and Figure 6.8, respectively. Their request responder behaviors are LISAREResponderBehavior and SvcARERespondeBehavior, respectively.

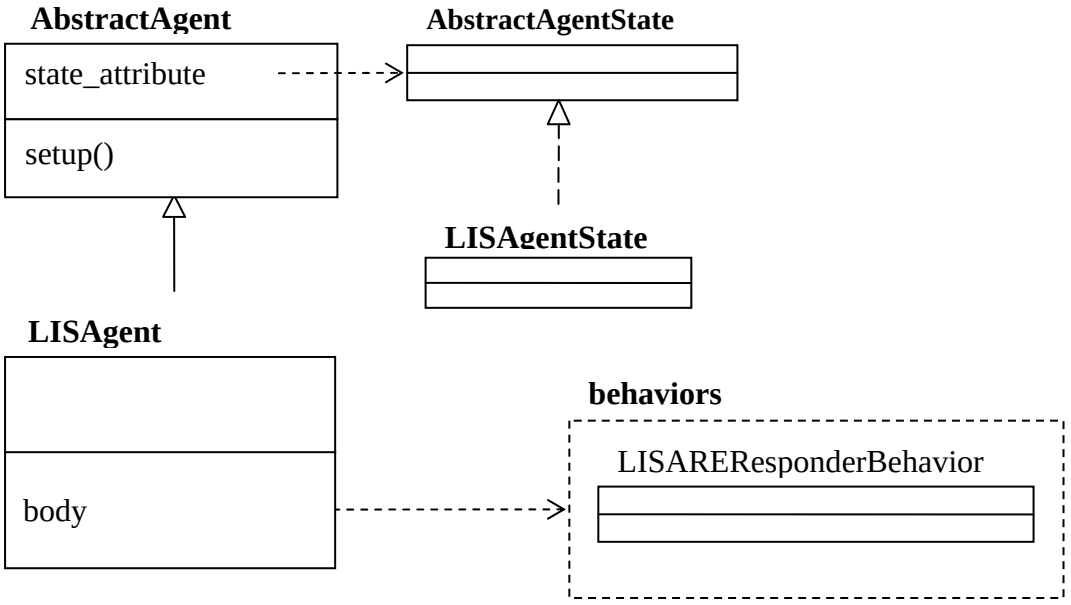

Figure 6.7: *Structure of the LIS Agent Class*

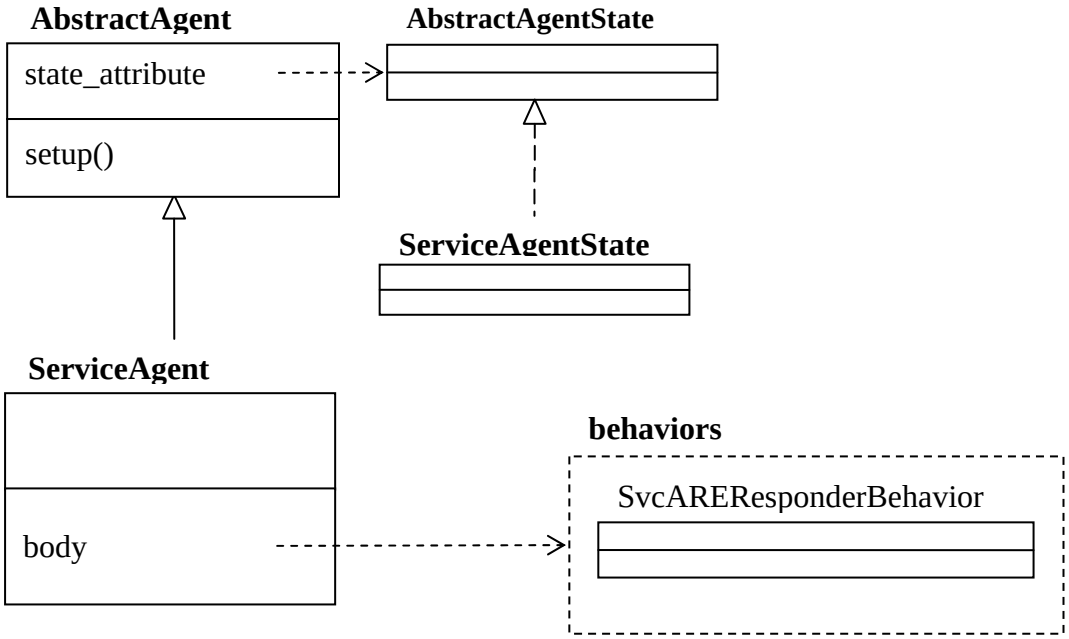

Figure 6.8: *Structure of the Service Agent Class*

6.1.3 The Design of MAgILOntology's Structure

The vocabularies, hierarchy of concepts, and relationships between concepts are defined in Section 5.3.2 and form a building block of MAgIL framework layer 2. The design of MAgILOntology is based on that definition, as shown in Figure 6.9.

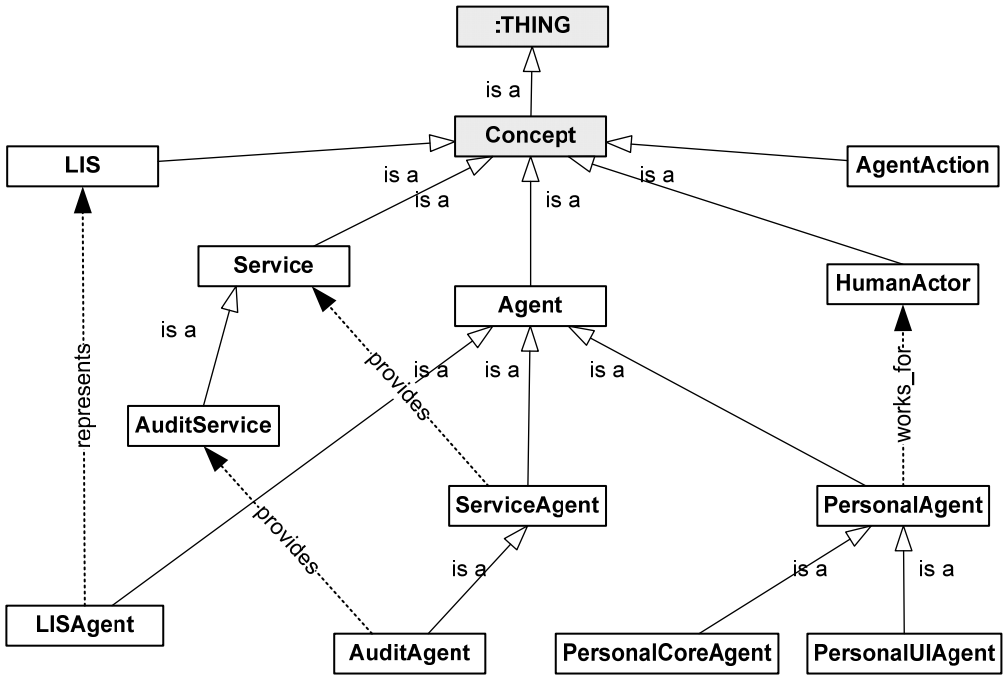

Figure 6.9: *MAgILOntology's Structure*

Each rectangular box presents a concept defined in the set of vocabularies and is considered as a class of objects: Concept class, LIS class, Service class, Agent class, HumanActor class, AgentAction class, etc.

The "is a" structure depicts the hierarchy of concepts. For example, an instance of the ServiceAgent class "is an" agent of the Agent class which, in turn, "is a" Concept.

The dotted arrow with label presents the relationship between two concepts. It is a many-to-one relationship. For example, many instances of the PersonalAgent class work for a human actor in the HumanActor class; many instances of the LISAgent class can represent a legacy information system in the LIS class.

6.2 The MAgIL Framework Implementation

The agent classes designed in Section 6.1 are implemented on the JADE platform and the Jess rule engine by using the Java object-oriented programming language. These technologies are chosen in Section 5.1. The source codes of this implementation are provided in (Nguyen, 2009). This section focuses on: (1) how to implement the abstract agent class and its behavioral classes on the top of JADE; (2) how to add an existing behavior into the abstract agent class as well as any agent class of the framework; and finally (3) how to implement MAgILOntology for the agent classes of the MAgIL framework.

6.2.1 Implementing the Abstract Agent Class and Its Subclasses

The JADE platform provides an agent class, jade.core.Agent, from which other agent classes can be developed. In the MAgIL framework the abstract agent class, AbstractAgent, is implemented as a subclass of jade.core.Agent using the structure designed in Section 6.1. The implementation is presented in the following excerpt of the source code (Listing 6.1). Some comments below are useful to understand Listing 6.1.

In the first line, the keyword "`abstract class`" means that the object class `AbstractAgent` will be extended to implement the role-specific agent classes of the MAgIL framework. The AbstractAgent class has a protected variable, `aState`, which corresponds to the AbstractAgent attribute with a type `AbstractAgentState`. The `setup()` method is the *template method*. It calls, in particular:

- `this.init()` to specify and initiate an agent's state and its user interface (if required).
- `this.registerOntologiesAndLanguages()` to register additional required ontologies and languages which agents in a role-specific agent class would use to communicate between themselves. The called method "`protected void registerOntologiesAndLanguages()`" which is empty in the AbstractAgent class will be customized in the subclasses of AbstractAgent.

- `this.registerWithAnAgent()` to register the services of an agent (if required) with another agent.
- `this.doAddInitialBehaviours()`to add the initial behaviors into an agent.

```
public abstract class AbstractAgent extends jade.core.Agent {

      /**  the attributes of the abstract agent class */
      protected       AbstractAgentState aState ;

      // other attributes intentionally hidden
      . . .

      /** the template method */
      protected void setup() {

          // initiate the agent's state and its user interface
          this.init();

          . . .

          // register ontologies and languages for the agent
          this.registerOntologiesAndLanguages();

          // register with another agent(e.g., DF Agent, CoreAgent)
          this.registerWithAnAgent();

          // add initial behaviors for the agent
          this.doAddInitialBehaviours();
      }
      /** the primitive operations */
      protected void init(){
          . . .
      };
      protected void registerOntologiesAndLanguages() { ... };
      protected void registerWithAnAgent() { ... };
      protected abstract void doAddInitialBehaviours();
      . . .

}
```

Listing 6.1: *Implementation of the Abstract Agent Class (code excerpt)*

Based on the abstract agent class explained in Listing 6.1, the role-specific agent classes (subclasses of AbstractAgent) are implemented by:

- extending AbstractAgent;
- then, customizing the empty methods from AbstractAgent;
- finally, concretizing the abstract methods from AbstractAgent.

Listing 6.2 shows an excerpt of the code to implement the personal core agent class in the MAgIL framework.

```
public class PersonalCoreAgent extends AbstractAgent {

      . . .

      /*
       * (non-Javadoc)
       * @see magil.AbstractAgent#doAddInitialBehaviours()
       */
      protected void doAddInitialBehaviours() {
          this.addBehaviour( new RegAndUnregBehaviour( this ) );
          this.addMoreSpecificBehaviours();
      }

      /**
       * The customization hook to add more behaviors
       * for the subclasses of the personal core agent class
       */
      protected void addMoreSpecificBehaviours(){ ... }

      . . .
}
```

Listing 6.2: *Implementation of the Personal Core Agent Class (code excerpt)*

In Listing 6.2, the method doAddInitialBehaviours() contains two calls. The first one, `this.addBehaviour( new RegAndUnregBehaviour( this ) )`, can be decomposed as follows:

- instantiate the RegAndUnregBehaviour behavior;
- attach the instantiated behavior to the personal core agent.

The second call, `this.addMoreSpecificBehaviours()`, leads to the bottom line of Listing 6.2, `protected void addMoreSpecificBehaviours()`. This method which is initially empty (i.e., contains no behaviors) will be customized later by developers to add specific behaviors into the subclasses of the personal core agent class.

6.2.2 Implementing the Classes in the Library of Behaviors

The JADE platform provides numerous classes of behaviors:

- OneShotBehaviour,
- CyclicBehaviour,
- TickerBehaviour,
- SimpleAchieveREInitiatior,
- SimpleAchieveREResponder, etc.

In the MAgIL framework the template classes of behaviors (cf. Figure 6.4) are implemented by extending the original classes provided by JADE. For example, the behavioral template

class OneShotTemplateBehavior is an extension of the JADE-provided OneShotBehaviour class as shown in the following excerpt from the source code (Listing 6.3).

```
import jade.core.Agent;
import jade.lang.acl.ACLMessage;

public class OneShotTemplateBehaviour extends
      jade.core.behaviours.OneShotBehaviour {

      protected ACLMessage message;
      /**@param agent */
      public OneShotTemplateBehaviour(Agent agent ) {
          super( agent );
          this.message = initMessage();
      }
      /**The method defines an ACLMessage.
       * It can be overridden in subclasses.
       * @return an instance of the ACLMessage
       */
      protected ACLMessage initMessage() {
          return new ACLMessage( ACLMessage.REQUEST );
      }

      /**The method defines the agent's action
       * "send a message".*/
      public void action() {
          this.myAgent.send( this.message );
      }
}
```

Listing 6.3: *Implementation of the OneShotTemplateBehavior Class*

Once extended from the JADE-provided behavioral classes, the behavioral template classes are further extended as needed to implement the specific behaviors for the role-specific agent classes. For example, the SubscriberBehaviour class is an extension of the template OneShotTemplateBehaviour class as shown in the following excerpt from the source code (Listing 6.4).

```
public class SubcriberBehaviour extends OneShotTemplateBehaviour {

      protected AID receiver;

      public SubcriberBehaviour( Agent agent, AID aid ) {
        super(agent);
        this.receiver = aid;
      }

      /* (non-Javadoc)
       * @see magil.behaviours.OneShotTemplateBehaviour#initMessage()
       */
      protected ACLMessage initMessage() {

        ACLMessage CASub = new ACLMessage(ACLMessage.SUBSCRIBE);

        CASub.addReceiver( this.receiver );
        CASub.setLanguage( FIPANames.ContentLanguage.FIPA_SL0 );
        CASub.setOntology( BasicOntology.ACLMSG );
        CASub.setReplyWith(
                      MAgILConstants.SUBSCRIPTION_MSG_MATCHREPLYWITH);
        CASub.setConversationId( new DUID().toString() );
        CASub.setContent( MAgILConstants.SUBSCRIPTION_MSG_CONTENT );

        return CASub;
      }
}
```

Listing 6.4: *Implementation of the SubcriberBehavior Class*

In order to initiate an ACLMessage which is specific to SubscribeBehaviour, in Listing 6.4, the group of lines between "`protected ACLMessage initMessage()`" and "`return CASub`" overrides the initMessage() method defined in the OneShotTemplateBehaviour.

6.2.3 Adding a Class of Behaviors into an Agent Class

The jade.core.Agent class provides a method called **addBehaviour**(*[name of behavior]*).

Every agent class is a subclass of jade.core.Agent. Therefore, addBehaviour will be used by an agent class to add into it a behavior class from the behavior library.

For example, to add the behavior class SubscriberBehavior into the agent class PersonalUIAgent, the following code:

this.addBehaviour (*magil.puiagent.behaviors.SubscriberBehavior()*);

is inserted in the body of the personal ui agent class.

In this example, the method addBehavior is called from the body of the agent class. More generally, the addBehaviour method can also be called from within any behavior.

6.2.4 Implementing MAgILOntology

MAgILOntology is implemented in two steps using the Protégé ontology tool suite (Stanford Center for Biomedical Informatics Research, 2007) with the Ontology Bean Generator plug-in (van Aart, 2007).

6.2.4.1 Editing Step

The graphical user interface of the Protégé ontology tool suite (Figure 6.10) is used to input all object classes, their hierarchical structure and relationships as previously designed (Figure 6.9).

In the right panel CLASS EDITOR, the HumanActor concept is currently edited. In the Template Slots section, the attribute "**pid**" is defined with type **String**. The left panel CLASS BROWSER displays the tree of object classes added by developers. The object classes ":THING", ":SYSTEM-CLASS", "AgentAction", "AID" (Agent IDentifier), "Concept" and "Predicate" are initially created by default by Protégé; the class AID is exactly equivalent to the Agent class which is defined in MAgILOntology.

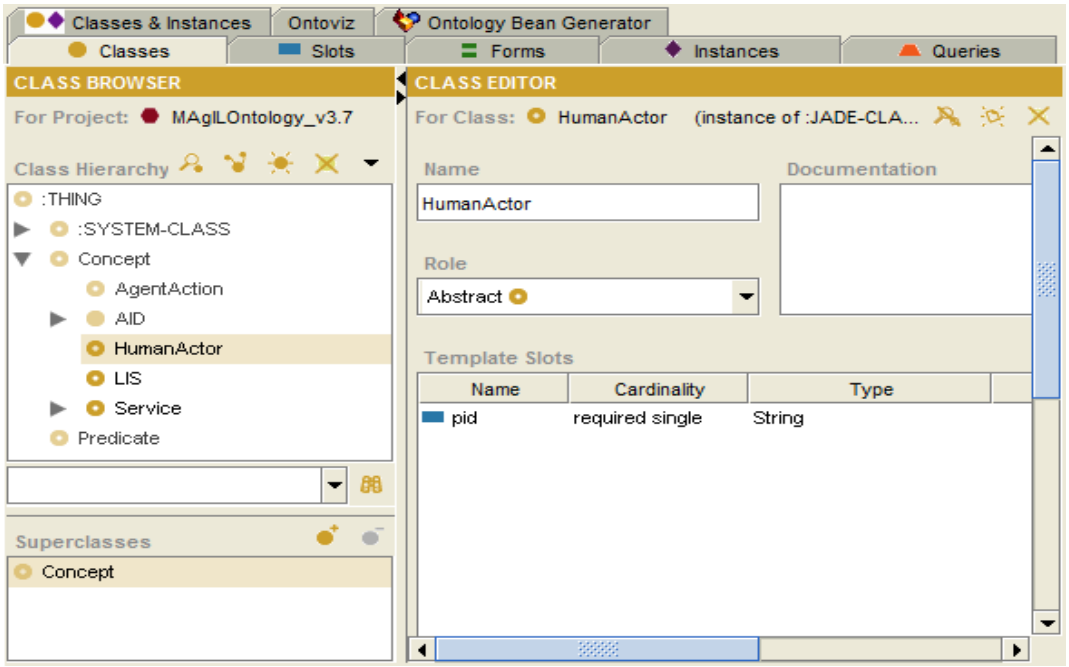

Figure 6.10: *Editing Object Classes of MAgILOntology in the Protégé Ontology Tool Suite*

6.2.4.2 Translation Step

Once all object classes of the ontology are inputted in Protégé, the Ontology Beans Generator can be used to translate them into Java classes.

In Figure 6.11 the graphical user interface specifies the package's name and domain name of the ontology as well as the location to save it. Clicking on the button **Generate Beans** will

create the Java classes according to those specifications. The Java class of the HumanActor class which is shown in Listing 6.5 is an example, in which:

- **HumanActor** declares the class name of the concept to be implemented;
- **pid** is an attribute of the HumanActor class;
- **setPid** and **getPid** are two methods of the HumanActor class, used to initialize the pid attribute and to get its value, respectively.

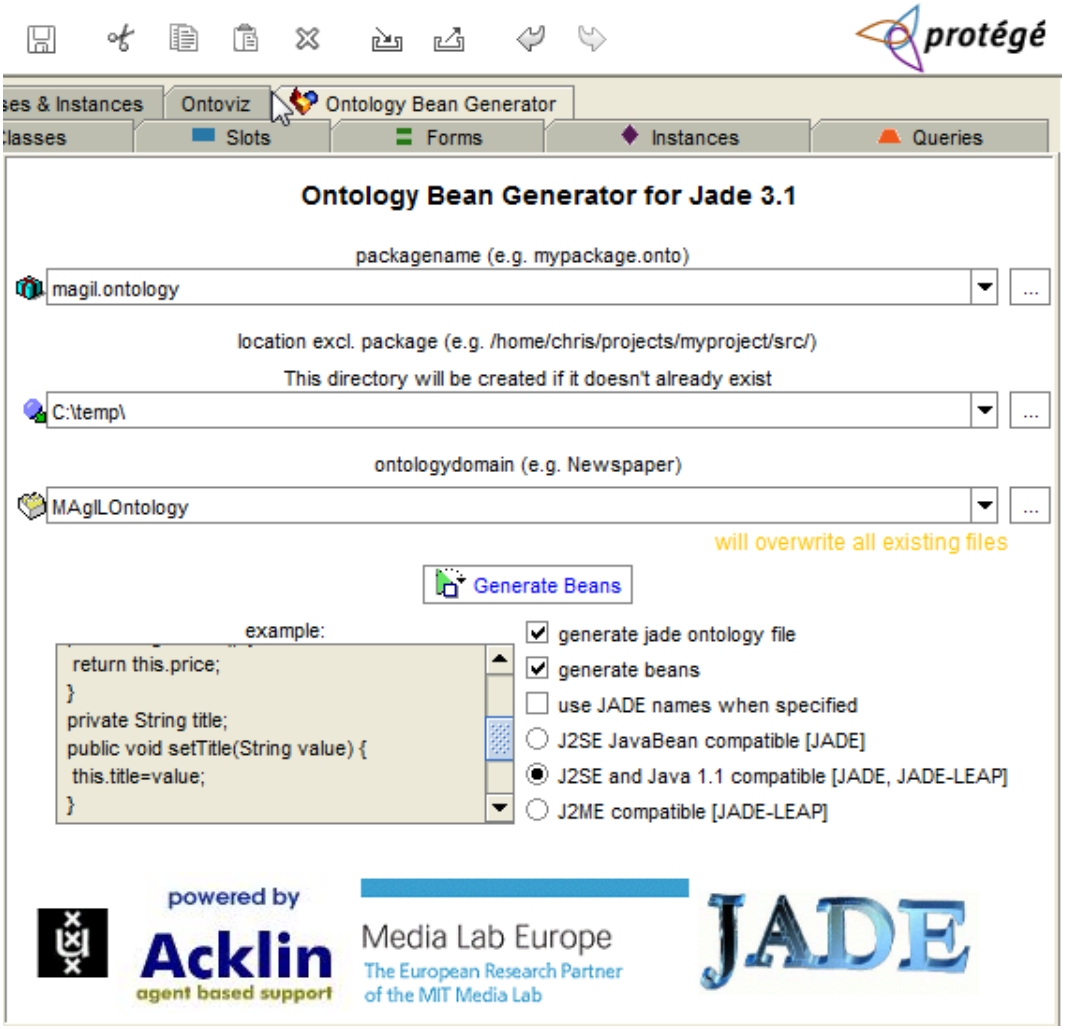

Figure 6.11: *Translating Object Classes in MAgILOntology into Java Classes*

```
package magil.ontology;

import jade.content.*;
/**
* Protege name: HumanActor
* @author ontology bean generator
* @version 2010/07/30, 14:28:09
*/
public class HumanActor implements Concept {
   /**
    * Protege name: pid
   */
   private String pid;
   public void setPid(String value) {
     this.pid=value;
   }
   public String getPid() {
     return this.pid;
   }
}
```

Listing 6.5: *The Human Actor Concept in a Java Class*

6.3 The Java Packages of the MAgIL Framework

The source codes of the MAgIL framework are organized into a hierarchy of Java packages and subpackages, where each package contains a set of classes and interfaces that implement a specific functionality. The main packages are enumerated below:

- `magil` is the main package of the MAgIL framework. It includes the object classes, which constitute the abstract agent class.
- `magil.behaviors` is a subpackage of `magil` that contains behaviors.
- `magil.puiagent` contains the implementation of the personal ui agent. The behaviors of this agent class are placed in a subpackage, named `magil.puiagent.behaviors`.
- `magil.pagent` contains the implementation of the personal core agent. The behaviors of this agent class are placed in a subpackage, named `magil.pagent.behaviors`.
- `magil.lisagent` contains the implementation of the lis agent. The behaviors of this agent class are placed in a subpackage, named `magil.lisagent.behaviors`.
- `magil.svcagent` contains the implementation of the service agent. The behaviors of this agent class are placed in a subpackage, named `magil.svcagent.behaviors`.
- `magil.ontology` contains the implementation of the MAgILOntology.

These packages and their hierarchy are illustrated in Figure 6.12. The double slashes (“//”) and comments are to describe the content of each package instead of listing the content in detail.

magil
|-// the abstract agent class and its state
|**-behaviors**
| |-// the template behaviors of the abstract agent class
|**-puiagent**
| |-// the personal ui agent class and its state
| |**-behaviors**
| | |-// the specific behaviors of the personal ui agent class
|**-pagent**
| |-// the personal core agent class and its state
| |**-behaviors**
| | |-// the specific behaviors of the personal core agent class
|**-lisagent**
| |-// the lis agent class and its state
| |**-behaviors**
| | |-// the specific behaviors of the lis agent class
|**-svcagent**
| |-// the service agent class and its state
| |**-behaviors**
| | |-// the specific behaviors of the service agent class
|**-ontology**
| |-// the elements (Java object classes) of the MAgIL's ontology

Figure 6.12: *MAgIL Framework's Java Packages*

7

The MediMAS Subsystem: A Case Study

The current chapter presents an application, called *MediMAS*, which is a subsystem in the information system of the Hospital of Fribourg. The subsystem is developed from the MAgIL framework. The first section provides an overview of the application. The second section shows the application from the user point of view. The third section presents a development process to build the MediMAS subsystem on the MAgIL framework. The last section highlights some experiences from this MediMAS project.

7.1 Introduction

MediMAS (Medical Multi-Agent System) is an application in e-healthcare domain, developed in a case study conducted at the Hospital of Fribourg (HFR) Laboratory (Nguyen et al., 2008;

Ruppen, 2007). MediMAS aims at demonstrating the use of the agent concept for improving the business process of the HFR Laboratory by enhancing its existing legacy information system and by assisting its healthcare personnel.

7.1.1 Case Study: the HFR Laboratory

The HFR Laboratory provides medical analysis ordered by hospital departments and private clinics in the canton of Fribourg. The laboratory covers different domains assigned to geographically dispersed sites: haematology, immuno-haematology, chemistry, and microbiology. It daily receives hundreds of orders along with specimens, analyzes the specimens, and delivers final results to the requesters (physicians, hospital departments, etc.). The methods of transmission of lab results depend on their urgency level (urgent or non urgent) and their nature (critical or non critical).

Besides the lab equipments for carrying out medical analysis, the personnel of the HFR Laboratory are supported in their daily tasks by the WinDMLAB Multisite system (Javet, 2007), coupled with a post mail system and a telephone communication system. They constitute the major components of the current HFR Laboratory Information System (cLIS) which is the legacy information system to be enhanced. The cLIS's architecture is characterized by three layers (cf. Figure 2.1):

- The bottom layer (Hardware Layer) consists of the lab equipments for analyzing specimens, the servers for running applications, the computer network for receiving and distributing data, and the telephone network for exchanging voice data.
- The middle layer (Software Layer) consists of the lab application programs used for medical analysis and lab results management: Database, WinDMLAB Windows application and WinDMLAB Web application provided by the WinDMLAB Multisite system. This system stores the medical orders and lab results in a centralized database and ensures the availability of information across the sites of the laboratory and the hospital departments.
- The top layer (Humans Actors) consists of human actors (e.g., lab director, lab technologists, physicians, etc.) using the lab application programs and the office equipments to execute the business processes of the HFR Laboratory. Each authorized human actor through the WinDMLAB Multisite system can access the lab results on their patients for reviewing at any level of detail.

The overall business process of the laboratory, illustrated by Figure 7.1, can be explained as follows:

- A physician orders a medical analysis by sending a request consisting of an order form and specimens to the laboratory (see Figure 7.1 ①). In the order form, the physician

must specify the priority degree by choosing one of two options: urgent or non urgent. An urgent request has a higher priority than a non urgent one. The physician also specifies one of following references:

- will be called back by the lab technologist upon the analysis completion (see Figure 7.1 ③ⓑ);
- will phone the laboratory to be informed about the results (see Figure 7.1 ③ⓒ);

- A lab technologist schedules and performs the requested analysis according to its priority degree (see Figure 7.1 ②). When the analyzing is terminated, the lab technologist must validate the lab results as either “critical” or “non critical”. A critical lab result is an unexpected or unpredictable result in a particular clinical setting and has the potential for serious adverse outcome to the patient or others if not dealt with promptly.
  - If the lab results are non critical, the lab technologist follows the reference specified in the order form (see Figure 7.1 ③ⓐ);
  - If the lab results are critical, the lab technologist must call the requester back within 30 minutes to alert him (see Figure 7.1 ③ⓑ). If the lab technologist cannot reach the requester, the lab director will be informed to take a decision.

  The lab results are recorded in the WinDMLAB Multisite system and the hard copy of the lab results will be sent to the requester via either fax or snail mail systems (see Figure 7.1 ④). Note that internal requesters (e.g., physicians in the hospital) can consult the WinDMLAB system to retrieve the results.
- The physician knows the lab results and treats his patient (see Figure 7.1 ⑤).

The business process shows that, besides parts using the WinDMLAB Multisite system, many parts still rely on the telephone, fax, and snail mail systems to get things done, for example, in the following circumstances:

- A lab technologist call a physician to inform him about the status of the medical analysis and to transmit the lab results;
- A physician calls the laboratory to be informed about a particular medical analysis;
- A lab technologist asks by phone his director to make a decision in an emergency situation, and so forth.

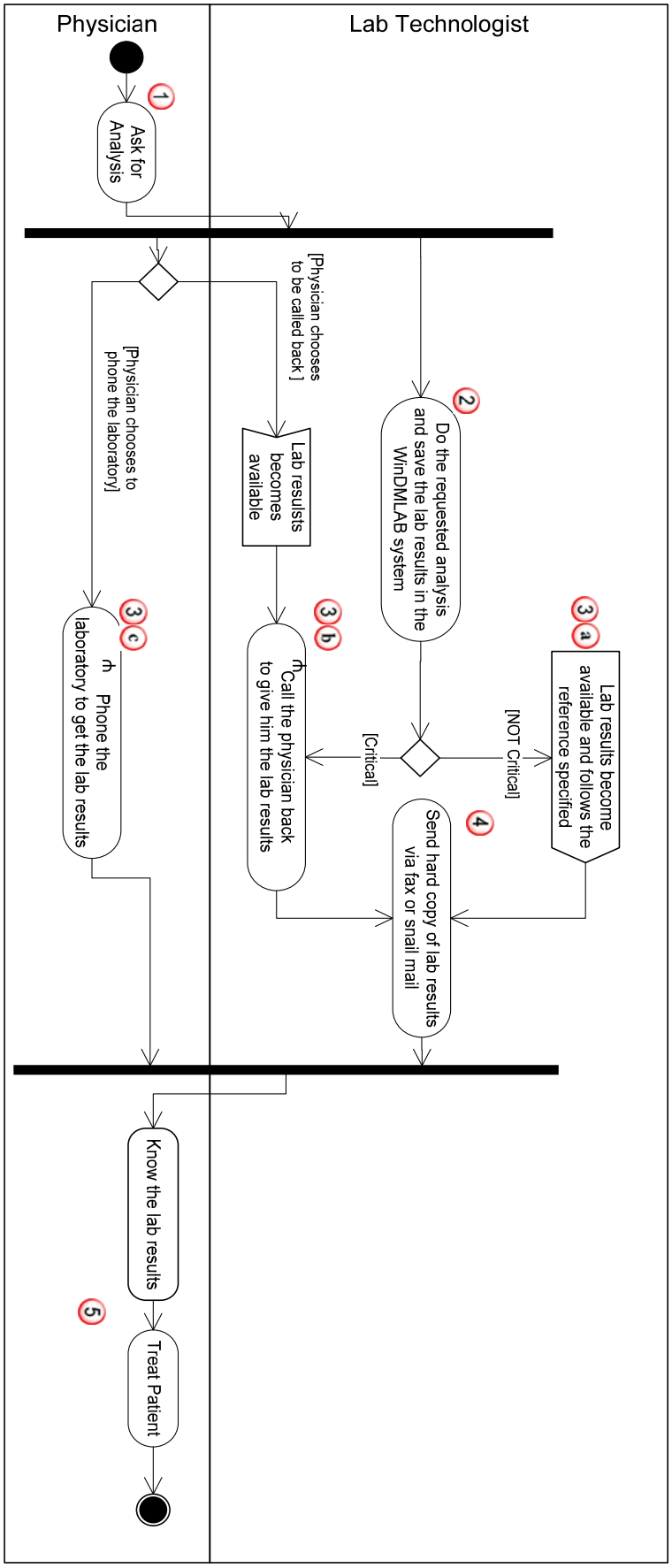

Figure 7.1: *UML Activity Diagram – The Business Processes of the HFR Laboratory*

7.1.2 Problem Analysis of the Information Flow

The business process and cLIS raise several inefficiencies and numerous potential problems in the information flow of the laboratory:

- Even though the major part of the lab results (80%) are transferred from the laboratory to the requesters through the WinDMLAB Multisite system, the service quality of the remaining 20% depends on manual operation and paper work, resulting in waste of time, inefficiencies, etc. For example, any mistake of a lab technologist in transferring medical results to a physician may cause dramatic consequences on patients.
- To establish a successful phone communication, two human actors must be present. Therefore, it will be a waste of time if they cannot reach one another when needed.
- The parts of business processes which are executed through the telephone, fax, snail mail system cannot be logged automatically in cLIS for monitoring and tracking purposes.
- The requesters cannot know at anytime the current status of their medical analysis.
- Physicians and lab technologists who use cLIS still spend a lot of time for searching, retrieving, consulting, and interchanging lab results.
- Because of the above time-consuming tasks, physicians and lab technologists cannot devote 100% of their time to the medical activities.

In summary, the information does not flow smoothly from the laboratory to the requesters. This problem illustrates the so-called “automation gap” (Gozdan, 2007; Opalis Software, 2007). What is needed is a systematic and strategic approach to avoid error-prone manual processes.

7.1.3 A Software Agent Solution

The “automation gap” may be overcome by using different software technologies, for example, JavaSpaces with SMS message technology, Web services technology, Multi-agent technology, and so forth.

This chapter does not aim at comparing the advantages and disadvantages of each technology. Our purpose is to suggest an agent-based approach (cf. Chapter 4) to improve the business process by enhancing cLIS and reducing tasks of human actors. More concretely, we develop and integrate the following software agents into cLIS:

- the personal agents for assisting physicians, lab technologists, and lab director;
- the lis agents for the WinDMLAB Multisite system;

- the service agents for providing system-wide surveillance services, for example, the alert monitoring service, event logging service.

The improved business processes are illustrated in Figure 7.2, in which the agents work on behalf of human actors. Thanks to agents that intervene between the lab technologist and physician (requester), the processes of sending and receiving information about the lab results are now automated. From Figure 7.1 to Figure 7.2, the processes are improved through simplification of many routine tasks. For example, the lab technologists and the physicians no longer need to call each other via phone to synchronize the reception of lab results. Indeed, an agent process called "Notify and Alert the availability of lab results" (Figure 7.2) replaces two former processes performed by human actors (Figure 7.1): the lab technologist (see 3b, "Call the physician back to give him the lab results"), and the physician (see 3c, "Phone the laboratory to get the lab results"). As a result of this enhancement, the human actors can really concentrate on their professional activities such as carrying out the requested analysis, saving the lab results in the WinDMLAB systems, and treating patients. Furthermore, routine tasks are executed more reliably by agents than by human actors.

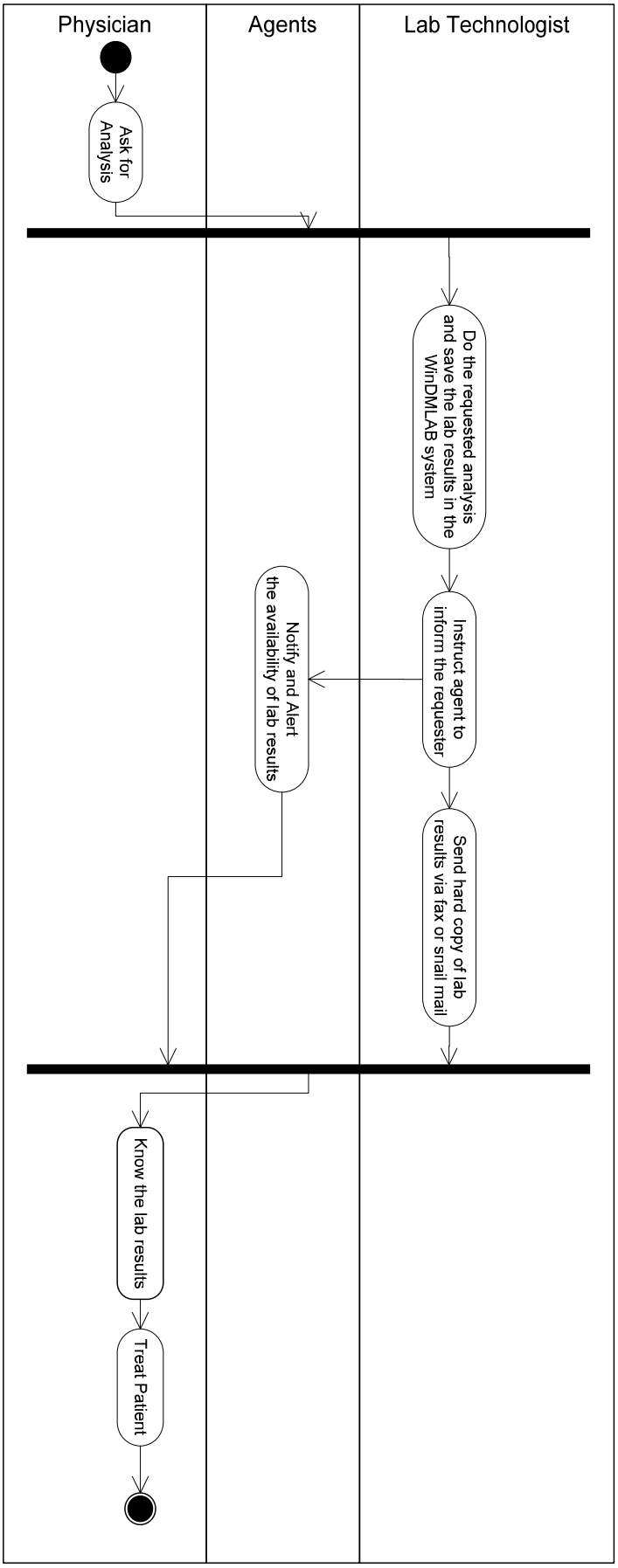

Figure 7.2: *UML Activity Diagram – The Enhanced Business Processes of the HFR Laboratory with Software Agents*

7.2 The MediMAS Prototype

The MediMAS prototype consists of a multi-agent subsystem and a lab-oriented ontology.

7.2.1 Multi-Agent Subsystem

The MediMAS subsystem has seven agent categories working for human actors in their daily tasks and the WinDMLAB Multisite system which is regarded as a legacy application subsystem:

Software Agent Category	Role	Owner
▪ Physician Agent	Personal Assistant	Physicians
▪ Lab Technologist Agent	Personal Assistant	Lab Technologists
▪ Lab Director Agent	Personal Assistant	Lab Director
▪ Lab Notification Manager Agent	System-wide surveyor who monitors and dispatches the notifications, reminders, and alerts	
▪ WinDMLAB LIS Agent	Representative of the WinDMLAB Multisite	
▪ Audit Agent	System-wide surveyor who provides a log service to MediMAS subsystem	
▪ Yellow Pages Agent	Directory Facilitator	

Figure 7.3 depicts their organization and shows the social ability of agents to cooperate with each other.

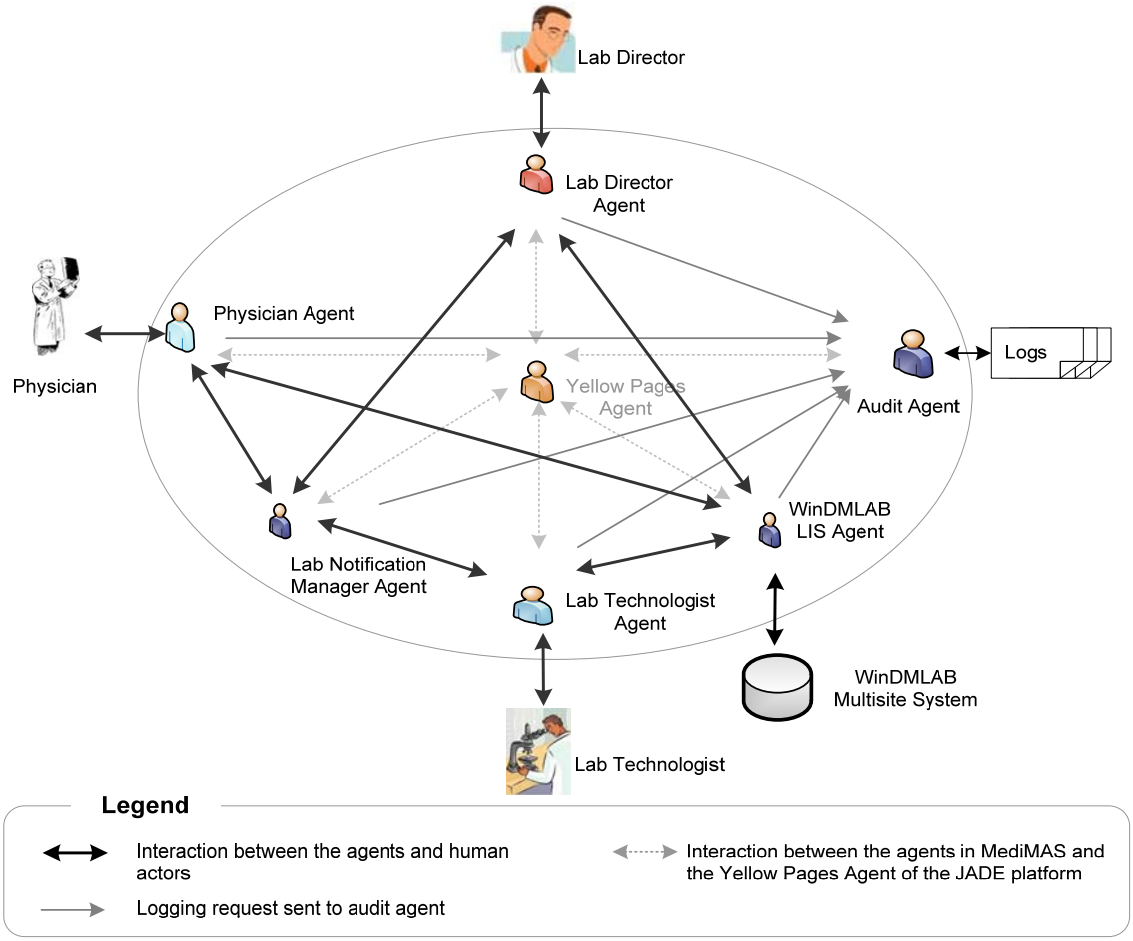

Figure 7.3: *MediMAS's Agents Organization*

7.2.2 Lab-Oriented Ontology

The human actors in this case study use a set of vocabularies specific to the healthcare domain, such as lab technologist, requester, physician, patient, analysis order, analysis result to communicate and share the knowledge between themselves, as well as between them and the WinDMLAB Multisite system. This set of vocabularies is a component of MediMASOntology. The software agents also use this ontology to communicate with each other and to work together to fulfill their goals.

MediMASOntology defines the semantic of the set of vocabularies as a structure of classes of objects illustrated by rectangular boxes in Figure 7.4.

- The top box "`:THING`" is the root of the ontology.
- The box `concept` is a thing, from which an "is a" structure is constructed starting from `HumanActor` down to `LabDirector`, `LabTechnologist`.
- The box `predicate` is another thing that defines an "is a" structure consisting of three classes of objects `IsCritical`, `IsUrgent`, and `isAcknowledged`.

- The box `AgentAction` is a concept that defines a number of actions performed by agents in MediMAS: `AskForAnalysisAction`, `NotifyAction`, `RemindAction`, `AlertAction`, etc.
- The box `AnalysisOrder` is another concept that defines an "is a" structure for `AnalysisResult`. An `AnalysisOrder` has a list of `AnalysisItemRefNbr`. An `AnalysisResult` consists of a list of `AnalysisItemRefNbrResult`.

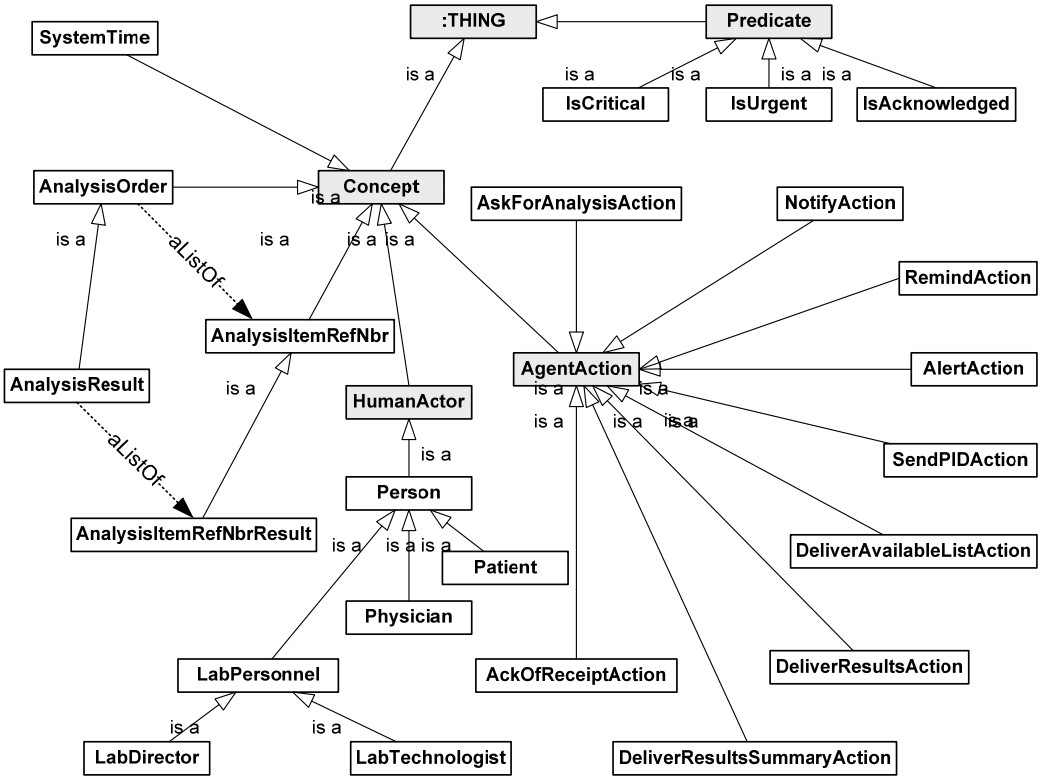

Figure 7.4: *MediMASOntology's Structure*

7.2.3 Agents in Action

7.2.3.1 Assignment of Agents to Human Actors

Table 7.1 introduces the four human actors with different roles depending on their workplace. Each human actor has a personal identifier (PID) shown in the fourth column.

Table 7.1: *The Human Actors' Roles*

Human Actor	Role	Workplace	PID
Jacques	Lab Director	HFR Laboratory	P19811133
Patrik	Lab Technologist	HFR Laboratory	P19811127
Tuan	Physician	HFR	P19811129
Andreas	Physician	Private Clinic	P19811132

In our MediMAS case study, each human actor owns exactly two personal agents, a personal Core agent, and a personal UI agent (cf. Section 4.3). They are the instances of the personal Core agent class and the personal UI agent class, respectively. Their instantiated names and class names are defined in Table 7.2, as well as their categories. Table 7.2 also shows Patrik owns a service agent called LabNotificationManagerAgent-2, who monitors and dispatches lab results' notifications and alerts as instructed by Patrik.

Table 7.2: *The Agents Assigned to Human Actors*

Human Actor	Name of Agent	Agent Class (in Java)	Agent Category
Jacques	Jacques_ LabdirectorCoreAgent	LabDirectorCoreAgent	Lab Director Agent
	Jacques_ LabdirectorUIAgent	LabDirectorUIAgent	
Patrik	Patrik_ LabTechnologistCoreAgent	LabTechnologistCoreAgent	Lab Technologist Agent
	Patrik_ LabTechnologistUIAgent	LabTechnologistUIAgent	
	LabNotificationManager Agent-2	LabNotificationManagerAgent	Lab Notification Manager Agent
Tuan	Tuan_ PhysicianCoreAgent	PhysicianCoreAgent	Physician Agent
	Tuan_ PhysicianUIAgent	PhysicianUIAgent	
Andreas	Andreas_ PhysicianCoreAgent	PhysicianCoreAgent	
	Andreas_ PhysicianUIAgent	PhysicianUIAgent	

7.2.3.2 Agents Incarnation

In the environment of the MediMAS prototype, all agents need the JADE platform to incarnate. Therefore, the JADE platform must be started first.

Next, the Audit Agent (MediMAS_AuditAgent) and the WinDMLAB LIS Agent (WinDMLAB_LISAgent) must be incarnated to be ready to log all events of the MediMAS subsystem and to provide data extracted from the database of the WinDMLAB Multisite system, respectively.

Then, the Core agents assigned to Jacques, Patrik, Tuan, and Andreas must be incarnated to be ready working around the clock for their owner. LabNotificationManagerAgent-2 is also incarnated to be ready to monitor and dispatch lab results' notifications and alerts.

Last, the UI agents of Jacques, Patrik, Tuan, and Andreas are incarnated to work with their owner on personal computer devices.

The agents, once instantiated, are attached to containers. A container is a runtime environment for agents (Bellifemine et al., 2007).

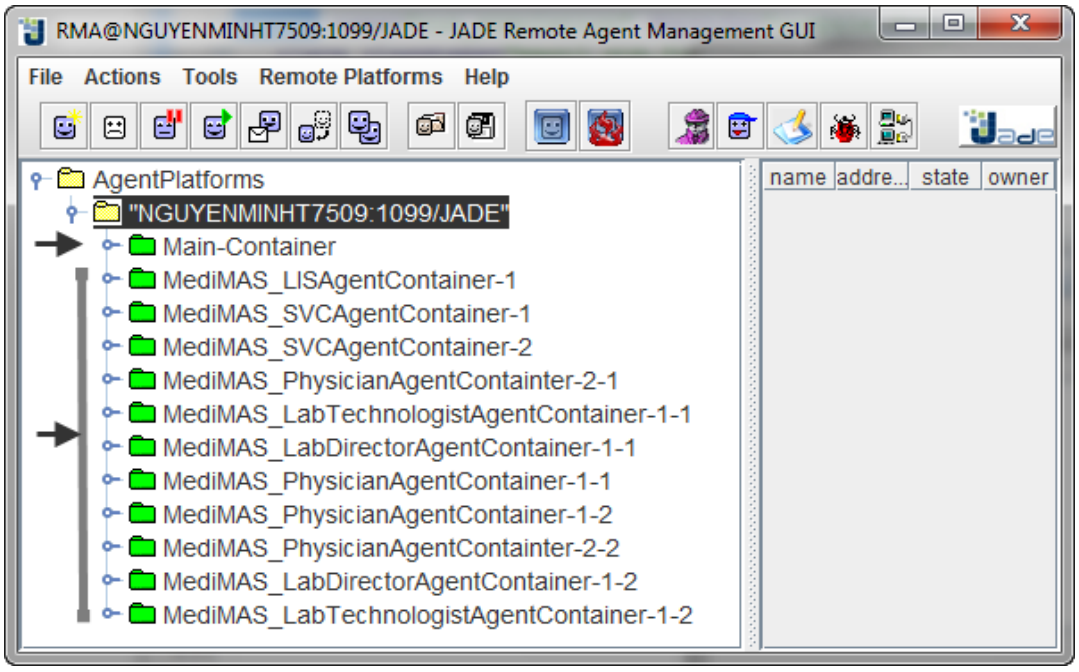

Figure 7.5: *Containers of Instantiated Agents*

Figure 7.5 shows the containers of instantiated agents:

- `Main-Container` is the container of the agents belonging to the JADE platform.
- The containers with the prefix `MediMAS_` are for agents specific to the MediMAS subsystem:
  - `MediMAS_LISAgentContainer-1`, container of `WinDMLAB_LISAgent`;
  - `MediMAS_SVCAgentContainer-1`, container of `MediMAS_AuditAgent`;
  - `MediMAS_SVCAgentContainer-2`, container of `LabNotificationManagerAgent-2`;
  - `MediMAS_LabTechnologistAgentContainer-1-1`, container of `Patrik_LabTechnologist`**`Core`**`Agent`.

    `MediMAS_LabTechnologistAgentContainer-1-2`, container of `Patrik_LabTechnologist`**`UI`**`Agent`;
  - `MediMAS_LabDirectorAgentContainer-1-1`, container of `Jacques_Labdirector`**`Core`**`Agent`;

    `MediMAS_LabDirectorAgentContainer-1-2`, container of `Jacques_Labdirector`**`UI`**`Agent`;

- `MediMAS_PhysicianAgentContainer-1-1`, container of `Tuan_Physician`**`Core`**`Agent`;

  `MediMAS_PhysicianAgentContainer-1-2`, container of `Tuan_Physician`**`UI`**`Agent`;

- `MediMAS_PhysicianAgentContainer-2-1`, container of `Andreas_Physician`**`Core`**`Agent`;

  `MediMAS_PhysicianAgentContainer-2-2`, container of `Andreas_Physician`**`UI`**`Agent`;

As an example, Figure 7.6 shows the contents of `MediMAS_LISAgentContainer-1`. `WinDMLAB_LISAgent@NGUYENMINHT7509:1099/JADE` is the full name of the agent. It concatenates four elements using appropriate delimiters:

- `WinDMLAB_LISAgent`, the agent's local name given in this prototype (cf. Table 7.2);
- `@NGUYENMINHT7509`, the server's name where the agent resides;
- `:1099`, the server's port number through which the agent communicate with other agents;
- `/JADE`, the name of the platform in which the agent is incarnated.

These pieces of information form a globally unique name for the agent in the MediMAS subsystem.

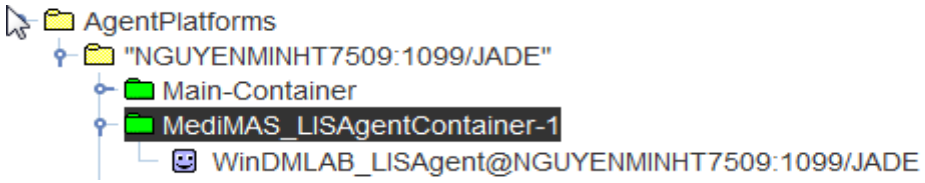

Figure 7.6: *WinDMLAB_LISAgent and Its Container*

Another example shows the contents of `MediMAS_LabTechnologistAgentContainer-1-1` (Figure 7.7). `Patrik_LabTechnologistCoreAgent@NGUYENMINHT7509:1099/JADE` is the global name of LabTechnologistCoreAgent assigned to Patrik.

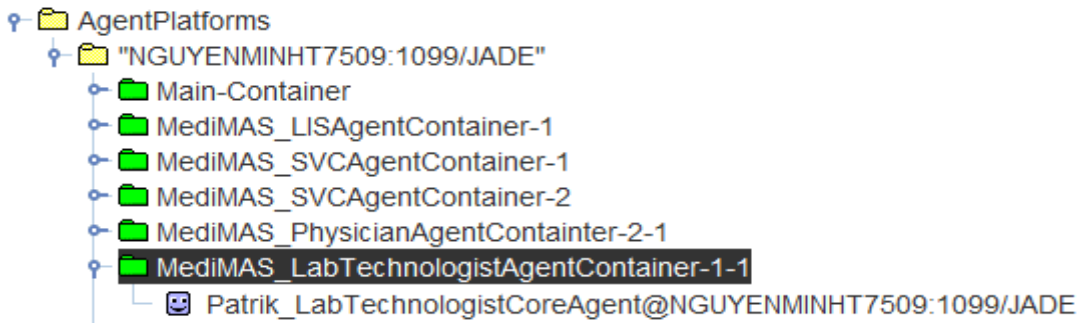

Figure 7.7: *Patrik_LabTechnologistCoreAgent and Its Container*

In the following scenario, starting by asking for analysis subprocess, we study in finer detail the activities of human actors and their assigned personal agents, and the interactions between each human actor and agents.

7.2.3.3 Scenario of subprocess 1 – Asking for Analysis

Tuan sends via snail mail three analysis orders to the HFR Laboratory. Each order consists of a filled order form and one or many specimens.

Upon receipt, the HFR Laboratory assigns the identifiers (order ID) `nlab-007`, `nlab-008`, and `nlab-009` to the three orders. `Nlab-007` and `nlab-008` were specified by Tuan as non urgent and `nlab-009` as urgent (Table 7.3).

Andreas, another physician, sends an urgent analysis order to the HFR Laboratory, identified by `nlab-999` (Table 7.3).

Table 7.3: *The Identification Number and Priority Degree of the Analysis Orders*

Requester	Identification Number of The Order Form	Priority Degree (Urgent or Non urgent)
Tuan	nlab-007	Non urgent
Tuan	nlab-008	Non urgent
Tuan	nlab-009	Urgent
Andreas	nlab-999	Urgent

The lab analysis of specimens will be scheduled by Patrik, lab technologist who takes into account the priority degree of the orders.

7.2.3.4 Scenario of subprocess 2 – Processing of Lab Results after Specimens' Analysis

The results of specimens' analysis are transferred from the lab equipments into the WinDMLAB database. The orders containing results exceeding thresholds are considered critical. In our scenario, the criticality of analysis orders is assumed as follows:

- `nlab-007` order is non critical;
- `nlab-008` order is critical due to results exceeding thresholds;
- `nlab-009` order is non critical;
- `nlab-999` order is critical due to results exceeding thresholds.

In the real world (HFR Laboratory), each lab technologist has a workstation which receives analysis results from different lab equipments and displays them as soon as an analysis order is completed. The results exceeding thresholds are highlighted and blinked, in which case the lab technologist must acknowledge their presence by checking an acknowledgement checkbox on screen. These real world operational details are not included in our prototype.

Table 7.4: *Lab technologist's rules of actions upon order completion – Decision Table DT1*

Conditions		Rule 1	Rule 2	Rule 3
C1	Requester: *priority degree (urgent or non urgent)?*	Non urgent	Non urgent	Urgent
C2	Lab technologist: *presence of results exceeding thresholds acknowledged?*	No	Yes	-
Actions				
A1	*Instruct software agent community to notify the requester of the availability of lab results*	✓	✓	✓
A2	*Send hard copy of the lab results to the requester via normal delivery*	✓		
A3	*Send hard copy of the lab results to the requester via express mail*		✓	✓

In our scenario, Patrik applies the rules defined in Table 7.4 to take appropriate actions at each analysis completion: Rule 1 is applied to `nlab-007`, Rule 2 to `nlab-008`, and Rule 3 to `nlab-009` and `nlab-999`.

In decision table DT1 (cf. Table 7.4), actions A2 and A3 are straightforward. Therefore, we now focus on how Patrik performs the action A1 and how the software agents cooperate with him to notify requesters, Tuan and Andreas, of the availability of lab results.

7.2.3.5 Scenario of subprocess 3 – Notification of Analysis Completion

Notification of the completed nlab-007

The subprocess to notify the completed nlab-007 can be explained step by step as follows:

- Patrik enters "nlab-007" in the `Order ID` field of his LabTechnologistUIAgent's GUI (Figure 7.8a).
- When Patrik clicks the `View result summary` button, `Patrik_LabTechnologistUIAgent` sends a request message M1 to `Patrick_LabTechnologistCoreAgent`. M1 contains the ontology concept `DeliverResultsSummaryAction` and the order ID "nlab-007". `DeliverResultsSummaryAction` asks Patrik's core agent to retrieve the attributes of `nlab-007`: order priority degree, degree of results criticality and the requester ID.
- `Patrick_LabTechnologistCoreAgent` forwards M1 to `WinDMLAB_LISAgent`.
- `WinDMLAB_LISAgent` queries the WinDMLAB database, then sends a response message M2 containing the extracted attributes to `Patrick_LabTechnologistCoreAgent`.
- `Patrick_LabTechnologistCoreAgent` forwards M2 to `Patrik_LabTechnologistUIAgent`.

- `Patrik_LabTechnologistUIAgent` displays the `nlab-007` attribute values received in M2. In Figure 7.8b, we note that `Urgent order` and `Critical result` are unchecked, and the requester ID attribute contains P19811129 for internal use by software agents to identify Tuan in their communication (cf. Table 7.1).

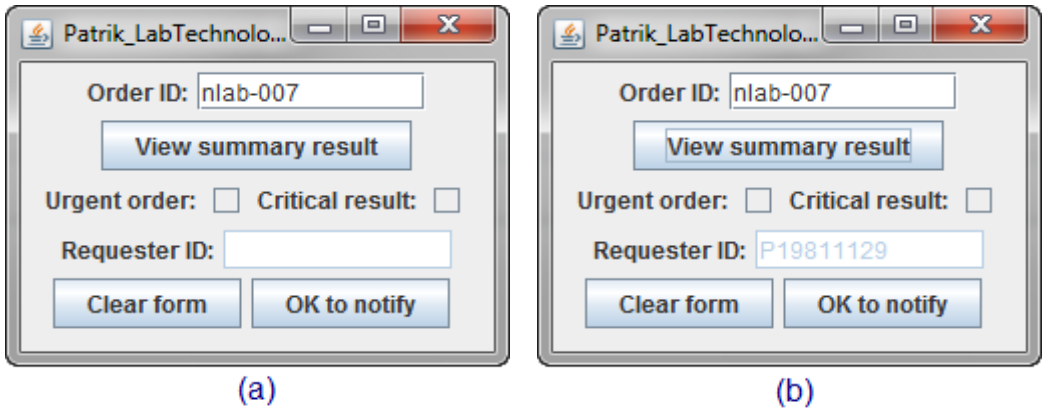

Figure 7.8: *Patrik_LabTechnologistUIAgent's GUI for nlab-007*

- When Patrick clicks the `OK to notify` button, `Patrik_LabTechnologistUIAgent` sends a request message M3 to `Patrick_LabTechnologistCoreAgent`. M3 contains the ontology concept `NotifyAction` along with the `nlab-007` attribute values. `NotifyAction` asks Patrik's core agent to send a notification to the requester whose PID is P19811129 for `nlab-007`.
- `Patrick_LabTechnologistCoreAgent` forwards M3 to `LabNotificationManagerAgent-2`.
- `LabNotificationManagerAgent-2` sends to `Tuan_PhysicianCoreAgent` an inform message M4 which contains the ontology concept `NotifyAction` along with the `nlab-007` attribute values and the initial number of reminders equal to 0. `NotifyAction` asks Tuan's core agent to send a notification to Tuan.
- `Tuan_PhysicianCoreAgent` forward M4 to `Tuan_PhysicianUIAgent`.
- `Tuan_PhysicianUIAgent` displays the content of M4, i.e., the `nlab-007` attribute values and the initial number of reminders. The notification subprocess 3 is completed at this point as the `nlab-007`'s notification has reached its recipient, i.e., the physician Tuan. The next subprocess 4 will explain how Tuan interacts with his agents after being notified.

`LabNotificationManagerAgent-2` has also a monitoring role. To this end, `LabNotificationManagerAgent-2`:

- logs the `nlab-007` attribute values (non urgent order, non critical result, P19811129) in its database of orders to monitor;

- applies the decision tables DT2 and DT3 (see Table 7.5 and Table 7.6) to send reminders to Tuan and alert Jacques, depending on how Tuan reacts to the first notification.

  The decision tables are either programmed directly into agent behaviors by using the procedural language Java, or declared in an input text file using the declarative language Jess. In the latter approach, the input file will be read by an agent behavior taken from the J2JToolkit's library.

Table 7.5: *LabNotificationManagerAgent's monitoring process – Decision Table DT2*

Conditions		**Rule 1**	**Rule 2**	**Rule 3**
C1	Input from requester: *priority degree (urgent or non urgent)?*	Urgent	Non urgent	-
C2	Input from lab technologist: *presence of results exceeding thresholds acknowledged?*	No	No	Yes
Actions				
A1	*Remind the requester every 20 minutes*		✓	
A2	*Remind the requester every 10 minutes*	✓		
A3	*Remind the requester every 5 minutes*			✓
A4	*Increase the number of reminders by 1*	✓	✓	✓
A5	*Go to DT3*	✓	✓	✓

Table 7.6: *LabNotificationManagerAgent's monitoring process – Decision Table DT3*

Conditions		**Rule 1**	**Rule 2**
C1	Software agent: *Number of reminders >3?*	No	Yes
Actions			
A1	*Alert lab director*		✓

Notification of the completed nlab-008, nlab-009, nlab-999

The notification of completed `nlab-008`, `nlab-009`, and `nlab-999` follows the same steps of subprocess 3 as previously applied to `nlab-007`.

The notification subprocess for `nlab-008` and `nlab-009` involves Patrik, the software agent community, and Tuan. For `nlab-999`, Patrik, the software agent community, and Andreas are involved.

For monitoring purpose, `LabNotificationManagerAgent-2` will log the attribute values of:

- `nlab-008` (non urgent order, critical result, P19811129),
- `nlab-009` (urgent order, non critical result, P19811129),
- `nlab-999` (urgent order, critical result, P19811132);

then apply decision tables DT2 and DT3 (see Table 7.5 and Table 7.6) to send reminders to Tuan and Andreas, and alert lab director Jacques, depending on how the requesters react on the first notification.

7.2.3.6 Scenario of subprocess 4 – Acknowledgment of Notification Receipt

Tuan's acknowledgment of receipt

After the last step of the subprocess 3, the notification window displayed by `Tuan_PhysicianUIAgent` for physician Tuan (Figure 7.9) shows three notifications waiting for acknowledgement.

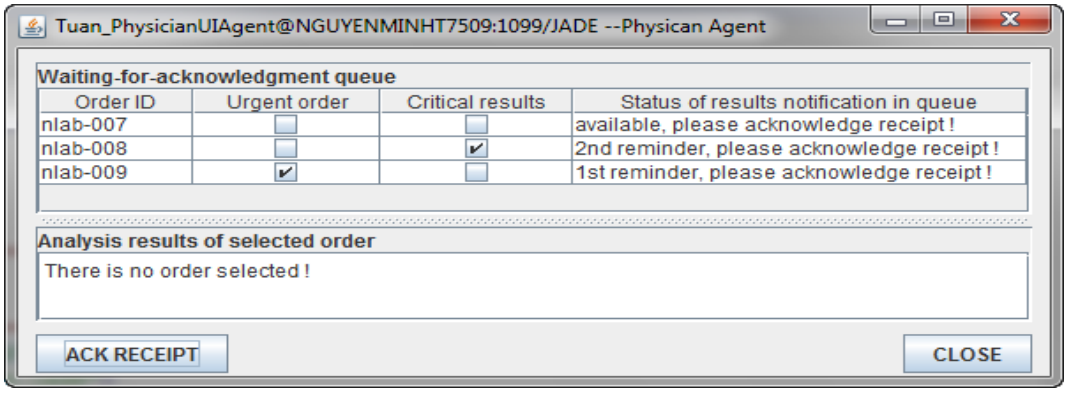

Figure 7.9: *Tuan_PhysicianUIAgent's GUI – Order notifications received*

For each notification, the attributes of the corresponding order (Order ID, Urgent order, and Critical results) are already discussed in subprocess 3; the fourth attribute gives the current *Status of results notification* and requires acknowledgement of receipt by the requester.

- When Tuan clicks anywhere on the `nlab-007` line, a request is transmitted from `Tuan_PhysicianUIAgent` to `WinDMLAB_LISAgent` via `Tuan_PhysicianCoreAgent`. The request message received by `WinDMLAB_LISAgent` contains the ontology concept `DeliverResultsAction` and the order ID "nlab-007", asking `WinDMLAB_LISAgent` to deliver the `nlab-007` results.
- `WinDMLAB_LISAgent` queries the WinDMLAB database. The extracted `nlab-007` results will be delivered by `WinDMLAB_LISAgent` to `Tuan_PhysicianUIAgent` via `Tuan_PhysicianCoreAgent`.
- `Tuan_PhysicianUIAgent` displays the `nlab-007` results in the lower pane of its GUI (Figure 7.10).

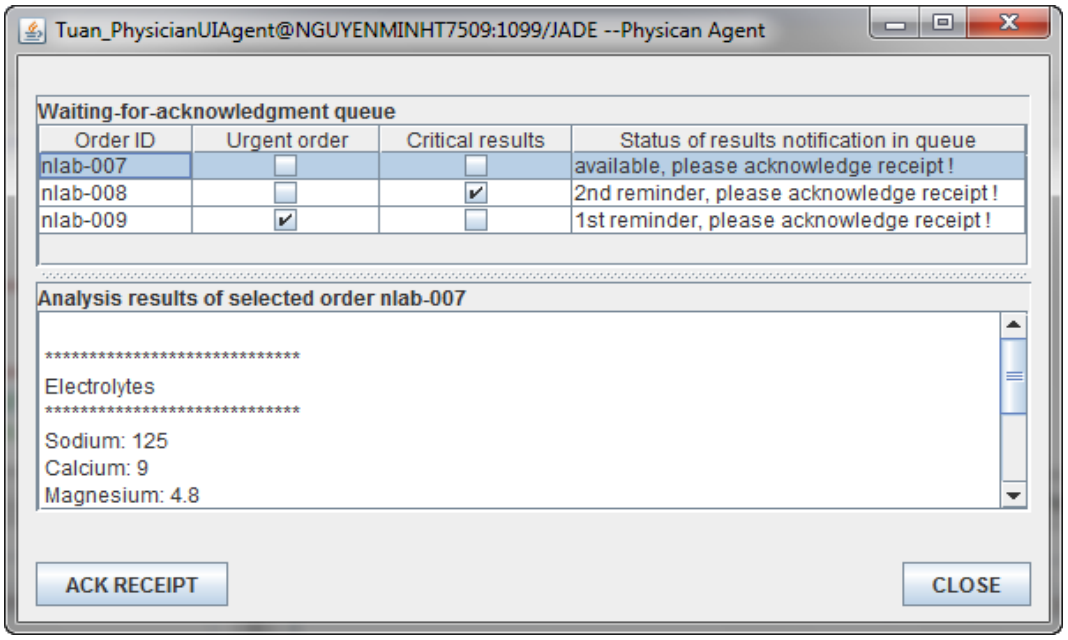

Figure 7.10: *Tuan_PhysicianUIAgent's GUI – Nlab-007 results displayed*

- When Tuan clicks the `ACK RECEIPT` button after reading the results, an inform message is transmitted from `Tuan_PhysicianUIAgent` to `LabNotificationManagerAgent-2` via `Tuan_PhysicianCoreAgent`. The inform message received by `LabNotificationManagerAgent-2` contains the ontology concept `AckOfReceiptAction` and the order ID "nlab-007" informing `LabNotificationManagerAgent-2` that the `nlab-007` results has *been read*.
- `LabNotificationManagerAgent-2` flags the `nlab-007` log record as "already read" and terminates `nlab-007` monitoring.
- `Tuan_PhysicianUIAgent` removes the `nlab-007` line from the *Waiting-for-acknowledgement Queue*. Figure 7.11 shows the remaining lines `nlab-008` and `nlab-009`.

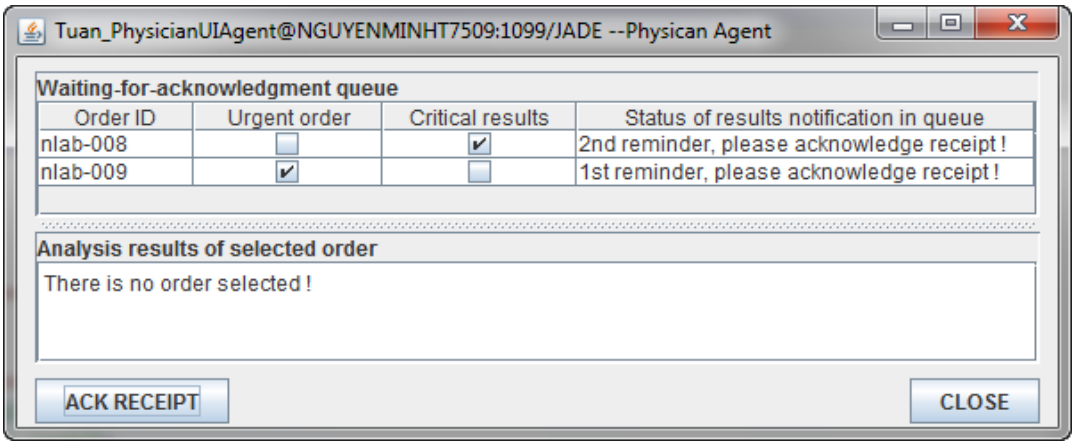

Figure 7.11: *Tuan_PhysicianUIAgent's GUI – Nlab-007 notification removed*

Andreas' acknowledgment of receipt

In a similar manner, Andreas views the `nlab-999` results and acknowledges the receipt, as shown in chronological order from Figure 7.12 through Figure 7.14.

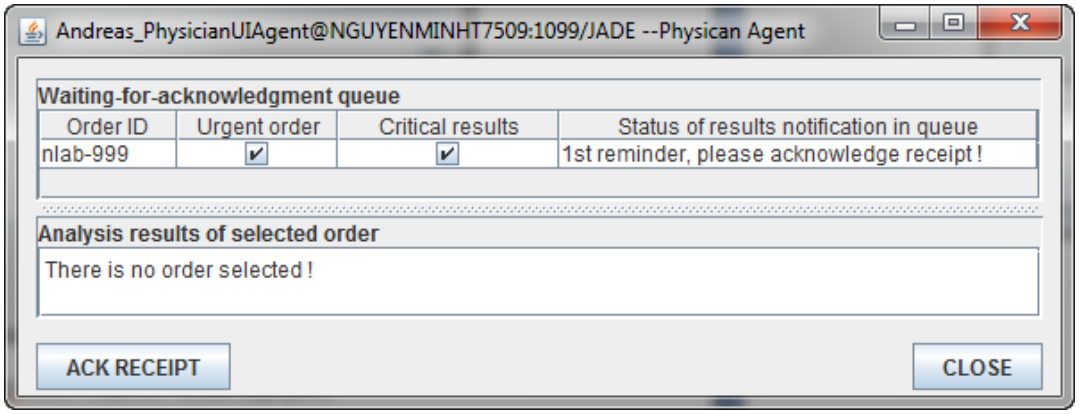

Figure 7.12: *Andreas_PhysicianUIAgent's GUI – Nlab-999 notification received*

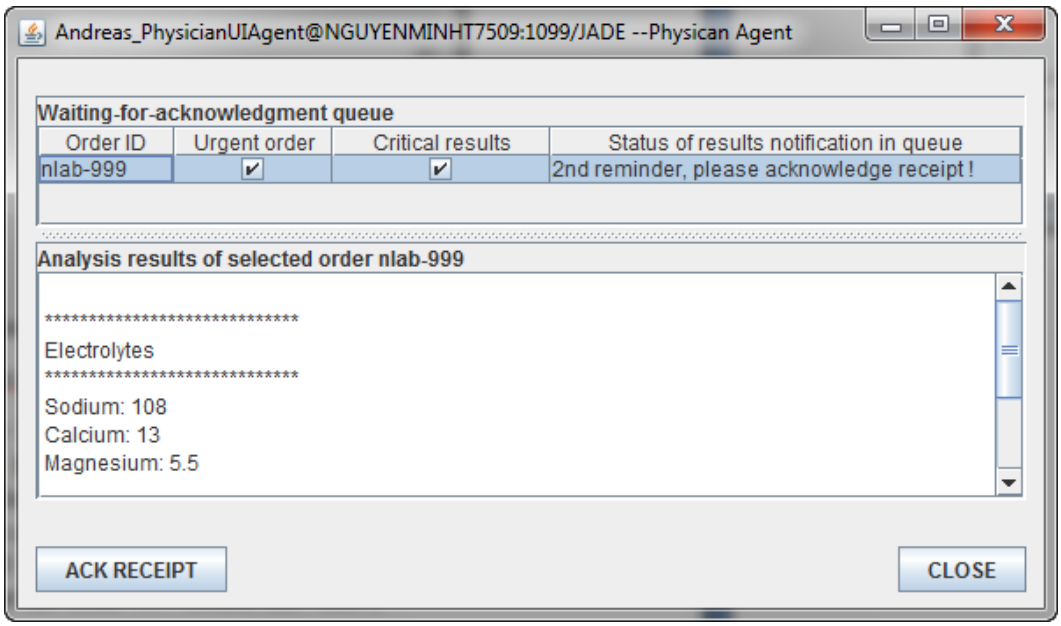

Figure 7.13: *Andreas_PhysicanUIAgent's GUI – Nlab-999 results displayed*

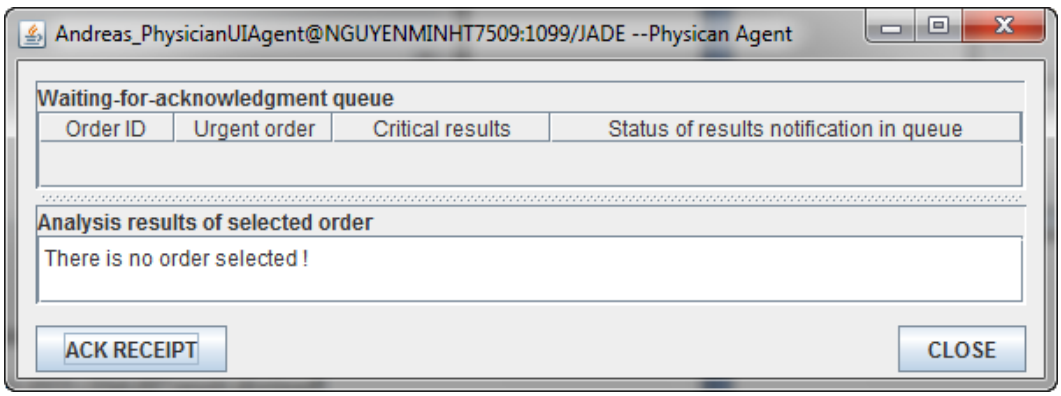

Figure 7.14: *Andreas_PhysicianUIAgent's GUI – Nlab-999 notification removed*

7.2.3.7 Scenario of subprocess 5 – Monitoring of Notification Status

Tuan has received the `nlab-008` notification, but he has not yet acknowledged the receipt. `LabNotificationManagerAgent-2` sends him a reminder every 5 minutes according to Rule 3 in decision table DT2 (cf. Table 7.5). The reminder is transmitted to Tuan via `Tuan_PhysicianCoreAgent` and `Tuan_PhysicianUIAgent`.

Starting from the 4th reminder, `LabNotificationManagerAgent-2` will also alert lab director Jacques according to Rule 2 in decision table DT3 (cf. Table 7.6).

7.2.3.8 Scenario of subprocess 6 – Acknowledgment of Alert Receipt

- The alert message received by `Jacques_LabdirectorUIAgent` contains the ontology concept `AlertAction` and the order ID "nlab-008" asking `Jacques_LabdirectorUIAgent` to alert Jacques about `nlab-008`.
- `Jacques_LabdirectorUIAgent` displays the alert in the "**Alert queue**" of its GUI (Figure 7.15).

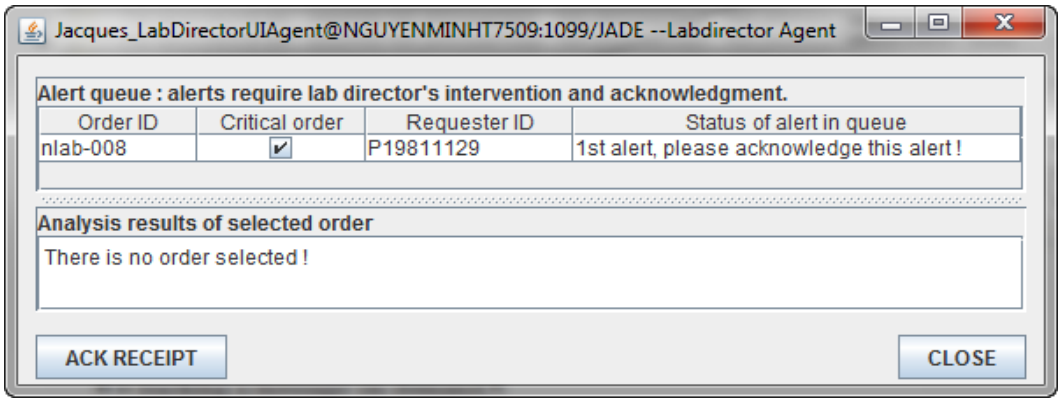

Figure 7.15: *Jacques_LabdirectorUIAgent's GUI – Nlab-008 notification received*

- When Jacques clicks anywhere on the `nlab-008` line, a request is transmitted from `Jacques_LabdirectorUIAgent` to `WinDMLAB_LISAgent` via `Jacques_LabdirectorCoreAgent`. The request message received by `WinDMLAB_LISAgent` contains the ontology concept `DeliverResultsAction` and the order ID "nlab-008" asking `WinDMLAB_LISAgent` to deliver the `nlab-008` results.
- `WinDMLAB_LISAgent` queries the WinDMLAB database. The extracted `nlab-008` results are delivered by `WinDMLAB_LISAgent` to `Jacques_LabdirectorUIAgent` via `Jacques_LabdirectorCoreAgent`.
- `Jacques_LabdirectorUIAgent` displays the `nlab-008` results in the lower pane of its GUI (Figure 7.16). Jacques immediately contacts Tuan to urges him to process the order `nlab-008` which contains critical values.
- When Jacques clicks the `ACK RECEIPT` button, an inform message is transmitted from `Jacques_LabdirectorUIAgent` to `LabNotificationManagerAgent-2 via`

`Jacques_LabdirectorCoreAgent`. The inform message contains the ontology concept `AckOfReceiptAction` and the order ID "nlab-008" informing `LabNotificationManagerAgent-2` that the alert for `nlab-008` has *been read*.

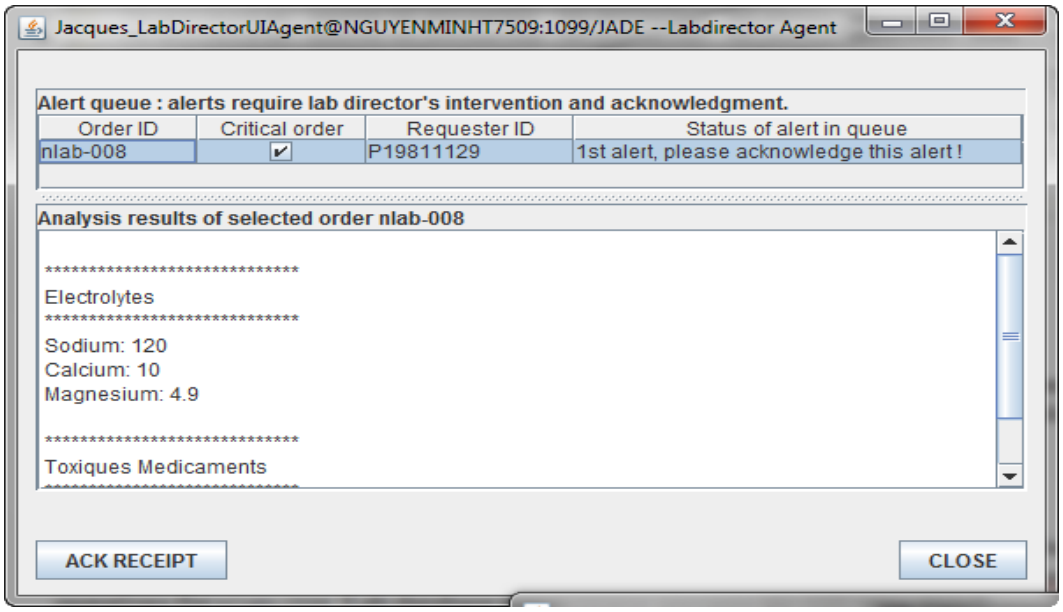

Figure 7.16: *Jacques_LabdirectorUIAgent's GUI – Nlab-008 results displayed*

- `LabNotificationManagerAgent-2` flags the `nlab-008` as "already read" and terminates `nlab-008` monitoring.
- Once the alert is acknowledged, `Jacques_LabdirectorUIAgent` removes `nlab-008` from its alert queue. Figure 7.17 illustrates the `Jacques_LabdirectorUIAgent`'s GUI after removing `nlab-008`.

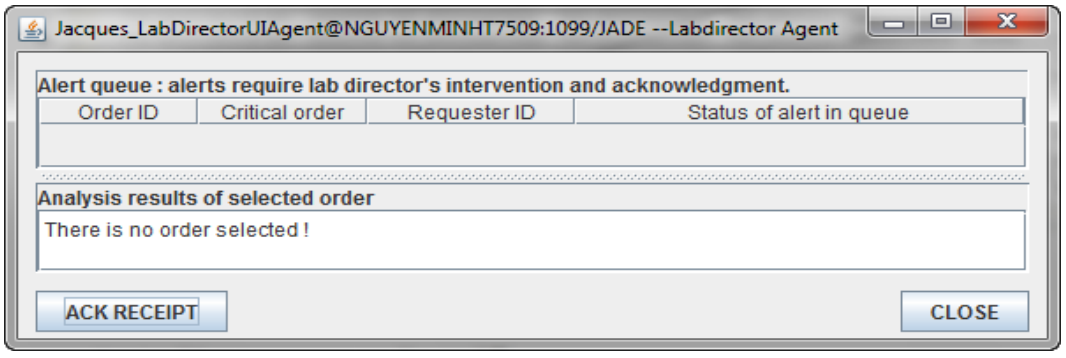

Figure 7.17: *Jacques_LabdirectorUIAgent's GUI – Nlab-008 notification removed*

In this subsection 7.3.4, we have simulated four analysis orders to demonstrate the working of agents in the MediMAS prototype and the benefits of a software agent approach to improve the business process of the HFR Laboratory and enhance the current laboratory information system (cLIS). In order to fully grasp the power of our solution, one however must consider the real HFR Laboratory, where hundreds of specimen analysis are ordered everyday by

dozen of physicians. All communication exchanges and reminder warnings are coordinated, timely delivered to the appropriate actors, and properly logged.

At this stage, we have described the functionalities of the MediMAS subsystem. The next section will discuss its development.

7.3 The Development Process Used for The MediMAS Subsystem

The development of MediMAS is based on four phases as presented in the UML activity diagrams shown in figures 7.18 and 7.19.

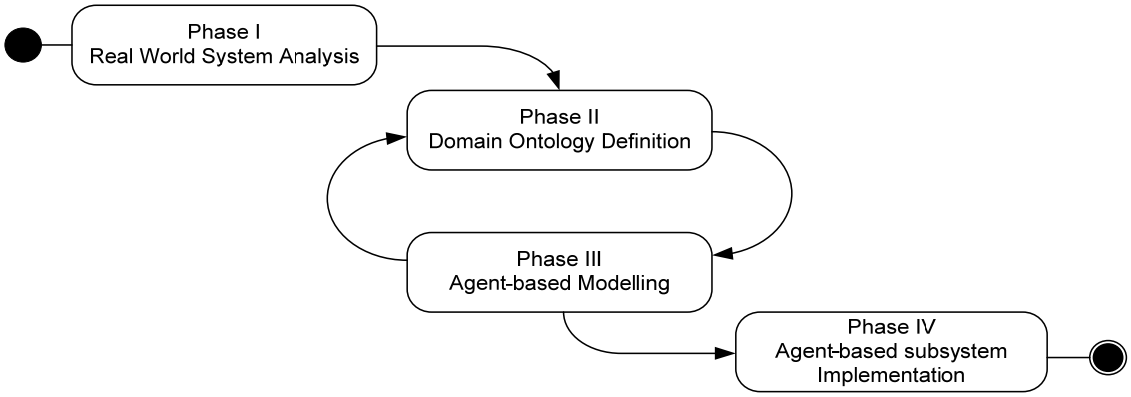

Figure 7.18: *The Four Phases of the Development Process*

7.3.1 Phase I – Real World System Analysis

The first phase aims at gathering and analyzing the users' requirements from the real world systems using requirement analysis methodologies and UML knowledge. To do this, the analyst must:

- perceive the current system in order to understand its current state, problems, new goals, and requirements;
- define the common vocabularies used between human actors and legacy application subsystems;
- determine the business process and use cases of the real world system.

The deliverables of this phase consist of a well-defined set of goals and requirements, the common vocabulary describing the entities (human actors, legacy application subsystems) with their organization as well as a set of identified use cases and business processes.

In our case study, the outputs of our real world system analysis are the three-layer information system structure of the HFR Laboratory (cf. Figure 2.1), the UML activity diagrams of the business processes (cf. Figure 7.1 and Figure 7.2), the list of problems of cLIS (cf. Section 7.1.2), and the agent-based solution to solve these problems (cf. Section 7.1.3).

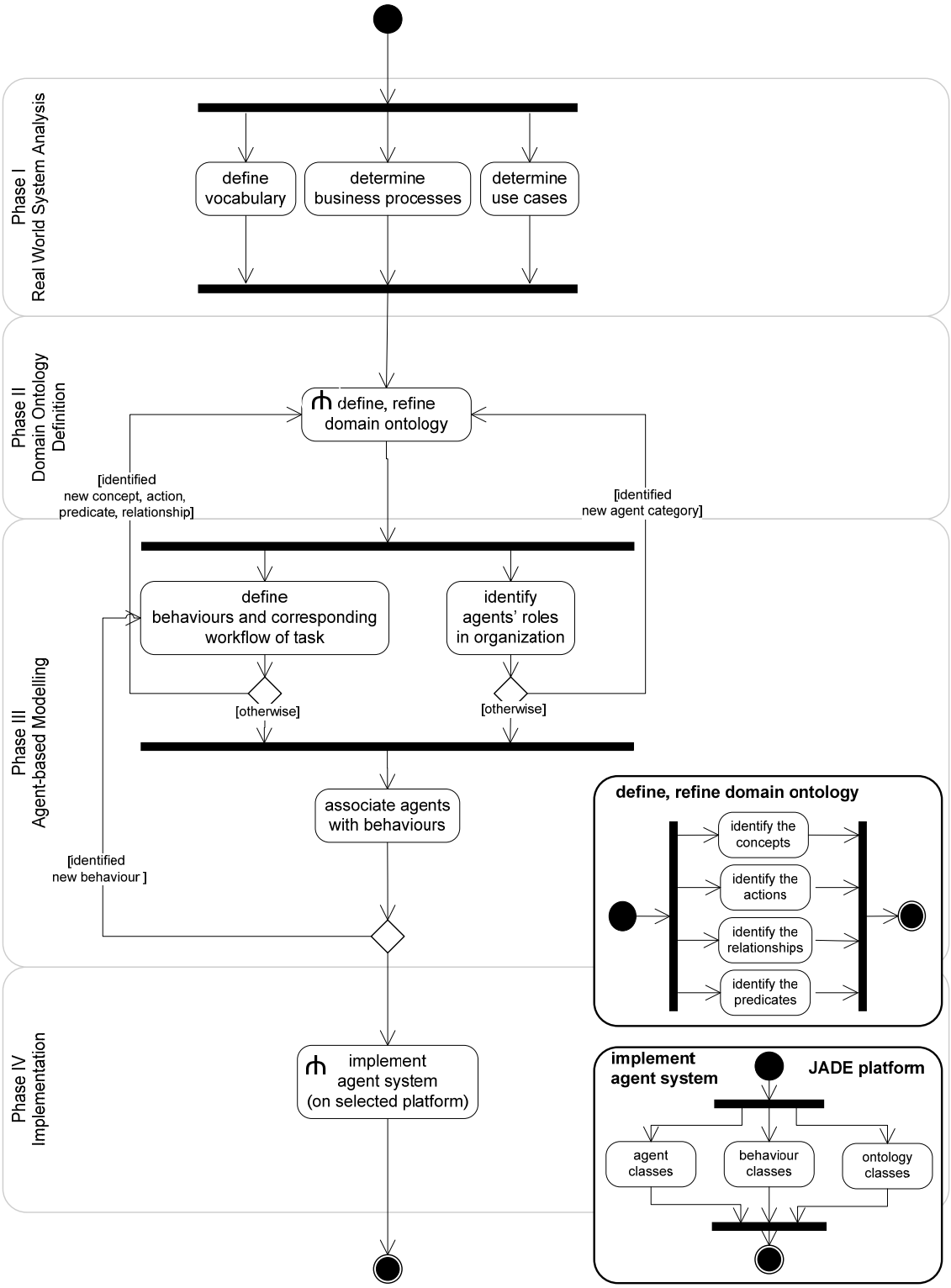

Figure 7.19: *The Guidelines and Steps of the Development Process*

7.3.2 Phase II – Domain Ontology Definition

The second phase aims at building an ontology for the future application subsystem that is used to enhance the real world system. This ontology is built either by reusing and extending an existing ontology or by creating a new one.

The ontologist uses the requirements of Phase I as input information and build the ontology using tools, such as Protégé-II suite of tools (Stanford Center for Biomedical Informatics Research, 2007) or TopBraidComposer (TopQuadrant, 2007). The output of this phase is the domain ontology that human actors and agents will use to understand each other in their communications.

In MediMAS, the Protégé-II suite of tools was used to build MediMASOntology (Figure 7.4), in which:

- Physician, LabTechnologist, LabDirector, Patient, are *concepts*;
- NotifyAction, RemindAction, AlertAction, DeliverResultsAction, are *actions*;
- isUrgent, isCritical, isAcknowledged, are *predicates*;
- aListOf is a *relationship*.

7.3.3 Phase III – Agent-based Modeling

The third phase aims at modeling the agent-based subsystem using the deliverables of Phase I and Phase II as inputs. This modeling phase consists of the following steps:

- identify the categories and roles of software agents needed by different groups of human actors and legacy application subsystems;
- define the behaviors and the corresponding workflow of tasks;
- associate behaviors to agent categories according to their roles in the organization.

The sequential order of the above steps is defined as shown in Figure 7.19 (Phase III). In addition, Phase II and III (Figure 7.18 and Figure 7.19) draw our attention to their iterative nature. For example, the arrow labeled `[identified new agent category]` means: if a new agent category is identified, it will be added into the ontology; the arrow labeled `[identified new concept, action, predicate, relationship]` means: each newly discovered concept, action, predicate, or relationship is added into the ontology.

In other words, successive refinement steps are necessary in order to enrich the domain ontology.

The deliverables of this phase are the documents:

- describing the categories of agents and their relationships (see Figure 7.3),
- specifying behaviors and corresponding workflow of tasks, and their assignment to agent categories.

In MediMAS, the agent categories and the tasks corresponding to their behaviors are listed in Table 7.7.

Table 7.7: *Tasks Performed by Agent Categories in the MediMAS Prototype*

Agent categories	Behaviors	Workflow of Tasks
Physician Agent	Receive and display notifications	• Receive notifications of the availability of lab results. • Display notifications for the physician.
	Receive and display reminders	• Receive reminders. • Display reminders for the physician.
	Send DB queries	• Send database queries to the WinDMLAB LIS agent according to a physician's information need.
	Receive and display lab results	• Receive lab results. • Display lab results for the physician.
	Send ACK of notification receipts	• Send acknowledgements of notification and reminder receipts to the lab notification manager agent.
Lab Technologist Agent	Notify the availability of lab results	• Notify the lab notification manager agent that lab results are available for a given physician.
Lab Director Agent	Receive and display alerts	• Receive alerts. • Display alerts for the lab director.
	Send ACK of alert receipts	• Send acknowledgements of alert receipts to the lab notification manager agent.
Lab Notification Manager Agent	Send notifications	• Notify the physician agent that lab results are available.
	Send reminders	• Send reminders to the physician agent.
	Send alerts	• Alert the lab director agent if a physician does not acknowledge receipt of lab results notification after a predetermined number of reminders.
	Receive notifications	• Receive notifications of the availability of lab results. • Add notifications into the list to be monitored.
	Monitor notifications, reminders, and alerts	• Monitor each notification in the list according to the decision tables of the monitoring process (see Tables 7.5 and 7.6).
	Receive ACK of notification receipts	• Receive acknowledgements of notification receipts. • Remove the acknowledged notification from the list.
	Receive ACK of alert receipts	• Receive acknowledgements of alert receipts. • Remove the acknowledged alert from the list
WinDMLAB LIS Agent	Receive DB queries and retrieve lab results	• Receive database queries. • Forward the queries to the WinDMLAB' database system for execution. • Retrieve lab results from WinDMLAB database.
	Send lab results	• Deliver lab results to the agents who sent the query.
Audit Agent	Log agent's operations	• Log MediMAS agents' operations with current date and time.

7.3.4 Phase IV – Implementation

Phase IV of the development process aims at implementing the agent-based subsystem modeled in phase III.

The previous phases I, II and III are agent platform-independent. In phase IV, the selection of a platform closely impacts the implementation process.

The programmers use the deliverables of the previous phases as inputs, and translate them into corresponding subsystem components of the chosen agent platform. For instance, if the JADE platform is chosen, the programmers must translate agent categories, behaviors, and the domain ontology into agent classes, behavior classes, and ontology class, respectively.

In the case of the MediMAS subsystem, the implementation is based on our three-layered approach as discussed in Chapter 5 and illustrated in Figure 7.20:

- Layer 1 is the development and running environment for the subsystem. It consists of the JADE platform, the Jess rule engine, and the Sim toolkit;
- Layer 2 is the MAgIL framework implemented as extensions of agent class, behavior classes, and ontology class of the JADE platform in Layer 1;
- Layer 3 is the MediMAS subsystem itself developed by extending the agent classes, behaviors classes, and the ontology class of the MAgIL framework in Layer 2.

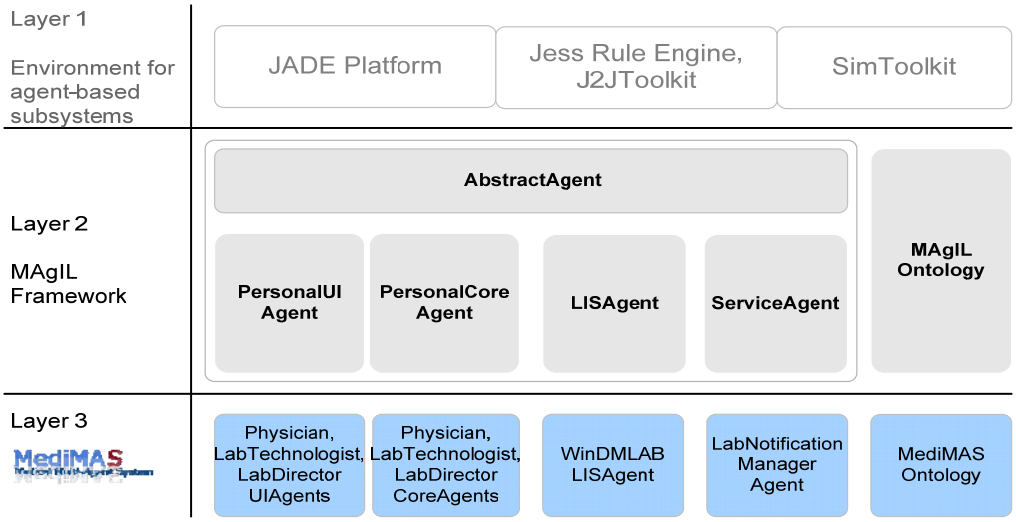

Figure 7.20: *Building Blocks of the MediMAS Subsystem*

Concretely, in Layer 3:

- PhysicianUIAgent, LabTechnologistUIAgent, LabDirectorUIAgent are extensions of PersonalUIAgent;
- PhysicianCoreAgent, LabTechnologistCoreAgent, LabDirectorCoreAgent are extensions of PersonalCoreAgent;

- WinDMLABLISAgent is an extension of LISAgent;
- LabNotificationManagerAgent is an extension of ServiceAgent;
- MediMASOntology is an extension of the MAgILOntology. It is implemented by coding the domain entities (i.e., concepts, agent actions) and relationships as object classes, either manually or through the bean generator plug-in for Protégé-II suite of tools (van Aart, 2007).

The completion of phase IV results in a multi-agent subsystem that fulfils the users' goals and requirements specified in Phase I.

7.4 Conclusion

The MediMAS subsystem has been presented as a prototype of a multi-agent system implemented from the MAgIL framework in the healthcare domain. It illustrates the major features and benefits of using agent-based approach to enhance the business process of the HFR Laboratory and its legacy information system:

- As the MAgIL framework is a semi-finished multi-agent subsystem, developers need only to extend the classes in the framework to implement the concrete MediMAS subsystem. Flexibility and reduced development time are two main benefits of this approach.
- J2JToolkit is a useful companion of the MAgIL framework by providing an interface called JessableAgent. JessableAgent allows the developer to implement agents that use the Jess rule engine to reason. This interface specifies a method, getJessCNC(), which returns an instance of type JessCNC (Jess Command aNd Control). The latter must be subclassed for each concrete agent. A JessCNC instance contains the specifications about the agent's ontology, the input text file declaring the decision tables, special behaviors for particular rules, etc. As an example, in the concrete MediMAS subsystem, we already know that the lab notification manager agent class is an extension of the MAgIL framework. This class implements the method getJessCNC() from the JessableAgent interface, which creates an appropriate JessCNC instance for the given class. The interested reader is referred to the online documentation for more details (Software Engineering Group, 2009).
- The developer can use SimToolkit to test and evaluate the concrete MediMAS subsystem. SimToolkit allows him to assess the working of the MediMAS subsystem in fulfillment of the goals established by the HFR laboratory (section 5.3.1). To this end, SimToolkit simulates human actors' daily activities, the distribution of agents on lab technologists and physicians PCs, laptops, PDAs, etc., their interactions, and the flow of information (messages, lab results, etc.) from and to human actors.

- The functionalities of the WinDMLAB Multisite system are maintained, preserving the investment in it.
- The delegation of routine tasks from human to software agents as personal assistants allows human actors to focus their attention on medical activities, such as specimen analysis, lab results interpretation, medical decision making. For example, phone calls between physicians and lab technologists (see 3c and 3b in Figure 7.1) are replaced by automatic notifications, reminders, and alerts sent by agents (see Figure 7.2).
- Human actors can view the notifications, reminders, alerts, and lab results transmitted by personal assistants anywhere, anytime, and on any device (PDAs, mobile phones, smart phones, etc.);
- All events and agents' actions are centrally logged to support auditing of the system. Traceability and exception investigation, for example, to answer the complaint of patient, are also improved.

8
Conclusion

This final chapter presents the achieved works and points out possible future research directions based on the MAgIL framework developed in this thesis.

8.1 Achieved Works

The thesis has identified and studied several limitations and problems of legacy information systems in organizations and their interconnection.

In order to overcome those pitfalls, a Multi-Agent Integration Layer (MAgIL) has been added to the traditional layered architecture of a legacy information system, hence the term “enhanced legacy information system”.

The Multi-Agent Integration Layer is itself organized in three layers:

- the agent-based environment layer (JADE, Jess, SimToolkit),
- the MAgIL framework layer,
- the concrete agent-based subsystem layer.

The MAgIL framework is a software product which aims at helping developers reduce the implementation time of concrete agent-based subsystems. In its current version, the MAgIL framework provides four agent categories: service agents, audit agents, lis agents, and

personal assistant agents. A simulation toolkit (SimToolkit) is provided at the environment layer to help developers in testing, evaluating agent-based subsystems, and making demos.

A development process has been proposed to software designers in order to identify and analyze the new requirements of users and organizations and to build a concrete agent-based subsystem that fulfills those requirements. The originality of this development process is to take into consideration the creation of an ontology from the very beginning of the process.

To validate the MAgIL framework and the development process, the MediMAS subsystem for the legacy information system of the HFR Laboratory has been developed. The work report has been published in International Journal of Telemedicine and Applications (Nguyen et al., 2008).

Throughout this thesis, the enhancement of legacy information systems by a multi agent-based solution clearly demonstrates many advantages:

- Organizations preserve their past massive investment in the legacy information systems;
- Organizations seamlessly integrate the ever-increasing business requirements of users into the existing information systems through the design of software assistant agents with minimum intervention on the existing legacy information systems;
- Users can delegate their routine tasks to the personal assistant agents so that they can really concentrate on their professional activities;
- The data could be automatically transmitted between the enhanced legacy information systems thanks to software assistant agents;
- The information extracted from the enhanced legacy information systems could be automatically transferred to the right person at the right place and the right time, instead of users having to search, retrieve, and filter that information by themselves;
- Users can efficiently share the information, news, knowledge, etc., among themselves.

8.2 Future Research Directions

The results of this thesis open many interesting directions for future research projects. Some of them will be suggested in the discussion below: enhancing the MAgIL framework, refining the development process, making a production version of the MediMAS prototype, and pursuing the research in agent-based systems by building other new applications based on the MAgIL framework.

8.2.1 Enhancing the MAgIL Framework

The MAgIL framework might be enhanced in different directions in order to increase its flexibility and usefulness in developing concrete agent-based subsystems.

In the current version of the framework, the agent classes can only interpret and execute tasks which are designed and implemented by developers; those tasks are coded in programming languages (e.g., Java, Jess, etc.) that are not easy for users to understand. Thus, the first direction is to add a new capability into the agent classes to enable them to interpret and execute any task specified at runtime by casual users who know very well the working processes, but are not necessarily programmers. The task specifications might be inputted using a markup language, such as XML, which is easy to understand, rather than programming languages.

The second direction is to add a WebServiceAgent class into the MAgIL framework whose role is to ensure the flow of information between a concrete agent-based subsystem and an application providing Web Services. For example, in a hospital application, let's imagine an existing web service indicating the current position of physicians in the building to a requester using a web browser. The responsibility of a WebServiceAgent is to use this web service to get the position of the physicians and transmit the information to the requester's personal assistant agent.

The concept of WebServiceAgent suggests that our MediMAS subsystem (c.f. Chapter 7) can be seamlessly integrated with existing web service applications such as the geolocation of physicians in a hospital area. The WebServiceAgent will play a central role linking the physician assistant agents and the geolocation web services. Without the WebServiceAgent, each physician assistant agent in MediMAS must have its own behavior implemented to exploit directly every web service application.

A third research direction relates to the security issues that are critical in a production environment. In order to ensure a desired security level, the MAgIL framework needs a security agent class. The security agents act as security officers to protect valid agents in a concrete agent-based subsystem against the malware agents whose purpose is to disrupt the working of the subsystem at different degrees of severity.

8.2.2 Automating the Development Process

In Chapter 7, the development process used to prototype the MediMAS subsystem in the healthcare domain consists of four phases to build a multi agent-based subsystem:

- Phase I – Real World System Analysis
- Phase II – Domain Ontology Definition

- Phase III – Agent-based Modeling
- Phase IV – Implementation.

Currently, this process consists essentially of guidelines. The developers are free to use existing system analysis and design methodologies and techniques such as UML. The next step could be to design an automatic generator that can translate the systems specifications into an application ontology, classes of agents and behaviors in the JADE platform.

8.2.3 The Future of the MediMAS Subsystem

In this thesis, the MediMAS subsystem has been implemented as a prototype in a healthcare environment. Its purpose is to demonstrate the feasibility and advantages of the MAgIL concept and framework.

One of possible future works is to produce a MediMAS subsystem version that can be used to enhance not only the legacy information systems of a hospital laboratory but also other existing information systems in a hospital. These enhancements allow at the same time the integration of information flows between the legacy information systems in a hospital, for example, human resource management system, patient management system, resource location management system, etc.

8.2.4 Research on Multi Agent-based Systems in the Software Engineering Group (Department of Informatics, University of Fribourg)

The research on the MAgIL framework presented in this thesis was achieved by the author as a member of the Software Engineering Group in the Department of Informatics, University of Fribourg.

This thesis has set a major research direction, namely the enhancement of legacy systems with agent technology, for the following completed projects by the group in the field of multi agent-based systems:

- *Système multi-agent: MediMAS – étude de cas dans le domaine du E-healthcare (Ruppen, 2007)* studies a multi agent-based system operating on a computer network. The MediMAS application developed in this project has been reengineered within the new MAgIL framework proposed in this thesis.
- *MediMASim: A Test and Simulation Toolkit for the MediMAS Application (Pointet, 2008)* helps to setup and simulate interactions between human actors and software agents on a computer network. In this thesis, we use SimToolkit, which is a generalization of Pointet's work.
- *Extension de MediMAS: développement et déploiement d'agents JADE sur des supports mobiles (Schaeppi, 2008)* studies how to develop and deploy agents on

mobile devices. Some parts of the codes have been reused to program the MediMAS subsystem in the thesis.

- *Jess to JADE (J2J) Toolkit (Vogt, 2008)* studies how to create an agent with inference capabilities based on the Jess rule engine. The latter is integrated in this thesis.

A future research direction suggested by the author is "a multi-agent subsystem in the academic domain" which may be called, for example, uniMAS. Its objective is to improve the legacy academic application subsystems and assist members of the university community in finding, retrieving, and exchanging information to perform efficiently their activities in the daily life on campus.

The following are some examples to show the potential usefulness of the uniMAS subsystem. The uniMAS personal agents can help a person to find:

- colleagues of the same nationality, from the same region to organize a party;
- partners to practice together a sport (e.g., cycling, tennis, fitness, basketball, etc.);
- people willing to share a vehicle to go to a concert or to make a trip;
- a partner to exchange language skills;
- a partner to arrange mutual service offerings (e.g., one hour of Math against one hour of Accounting).

Appendix A

Class Structure of The MAgIL Framework

This appendix presents the class structure of the MAgIL framework. The main classes and interfaces are enumerated and shortly described. The interested reader is invited to consult an exhaustive and up-to-date document `javadoc` (hypertext link) on the MAgIL project website (Nguyen, 2009).

A.1 List of the MAgIL Framework Packages

`magil`	contains the java object classes and interfaces defining the generic agent class of the MAgIL framework.
`magil.svcagent`	contains classes and interfaces defining the service agent class.
`magil.svcagent.auditagent`	contains classes defining a concrete service agent class called the audit agent class.
`magil.lisagent`	contains classes and interfaces defining the lis agent class.
`magil.pagent`	contains classes and interfaces defining the personal core agent class.
`magil.puiagent`	contains classes and interfaces defining the personal ui agent class.
`magil.ontology`	contains classes defining concepts, predicates, agent actions, and relations between concepts to describe semantics of the MAgIL subsystem.

A.2 The `magil` Packages

A.2.1 Abstract Agent Classes

AbstractAgentState	This is a java interface defining the state of the generic agent class in MAgIL.
AbstractAgent	This java abstract class presents a general agent class of the MAgIL framework.

A.2.2 Agent Behavior Classes (`magil.behaviours`)

CyclicTemplateBehaviour	This class defines a template of a cyclic behavior used by certain classes of agents in a MAgIL subsystem.
OneShotTemplateBehaviour	This class defines a template of a one shot behavior used by all classes of agents in a MAgIL subsystem.
REInitiatorTemplateBehaviour	This class defines a template of a request initiator behavior. It presents the initiator part of FIPA-Request-Interaction-Protocol.
REResponderTemplateBehaviour	This class defines a template of a request responder behavior. It presents the participant part of FIPA-Request-Interaction-Protocol.
TickerTemplateBehaviour	This class defines a template of a ticker behavior.

A.3 The `magil.svcagent` Packages

A.3.1 Service Agent Classes

ServiceAgentState	This is a java interface describing a general state of the service agent class.
ServiceAgent	This class defines the service agent class of the MAgIL framework.

A.3.2 Service Agent Behavior Classes (`magil.svcagent.behaviours`)

SvcARERespondBehavior	This class defines a template of a request responder behavior for a service agent. It extends the `magil.behaviours.RESponderTemplateBehaviour` of MAgIL.

A.3.3 Audit Agent Classes (`magil.svcagent.auditagent`)

AuditAgentState	This class provides a default implementation for a ServiceAgentState. It defines the data structure of the AuditAgent's state.
AuditAgent	This class provides a default implementation for a ServiceAgent. It presents the class of audit agents of a MAgIL subsystem. It is responsible for logging the events occurring in a MAgIL subsystem, especially the communication between agents in the system.

A.3.4 Audit Agent Behavior Classes (`magil.svcagent.auditagent.behaviours`)

LogBehaviour	This class is a cyclic behavior of the audit agent. It interprets the log request from different agents and stores information into a DBMS. This class extends the `jade.core.behaviours.CyclicBehaviour` of JADE.

A.4 The `magil.lisagent` Packages

A.4.1 LIS Agent Classes

LISAgentState	This is a java interface describing a general state of the lis agent class.
LISAgent	This is a java abstract class defining the lis agent class of a MAgIL subsystem.

A.4.2 LIS Agent Behavior Classes (`magil.lisagent.behaviours`)

LISAREResponderBehavior	This class defines a template of a request responder behavior for a lis agent. It extends the `magil.behaviours.REResponderTemplateBehaviour` of MAgIL.

A.5 The `magil.pagent` Packages

A.5.1 Personal Core Agent Classes

PersonalCoreAgentState	This is a class defining the general state of a personal core agent.

PersonalCoreAgent	This personal core agent class presents the general personal core agent of a MAgIL subsystem.

A.5.2 Personal Core Agent Behavior Classes (`magil.pagent.behaviours`)

PCoreARERespondерBehaviour	This class presents a template of a responder behavior for a personal agent. It extends the `magil.behaviours.RESponderTemplateBehaviour` of MAgIL.
RegAndUnregBehaviour	This class describes the procedure to register and unregister a user interface agent.

A.6 The `magil.puiagent` Packages

A.6.1 Personal UI Agent Classes

PersonalUIAgentState	This is a class defining the general state of a personal user interface agent.
PersonalUIAgent	This personal user interface agent class presents the general personal user interface agent of a MAgIL subsystem.

A.6.2 Personal UI Agent Behavior Classes (`magil.puiagent.behaviours`)

PUIARESponderBehaviour	This class presents a template of a responder behavior for a personal ui agent. It extends the `magil.behaviours.RESponderTemplateBehaviour` of MAgIL.
CancellerBehaviour	This class presents a template of a canceller behavior for a personal ui agent. It extends the `magil.behaviours.OneShotTemplateBehaviour` of MAgIL.
SubcriberBehaviour	This class presents a subscriber behavior for a personal ui agent. It extends the `magil.behaviours.OneShotTemplateBehaviour` of MAgIL.

A.6.3 Personal UI Classes (`magil.puiagent.ui`)

PersonalUI	This interface describes a user interface for a personal user interface agent.

A.7 The `magil.ontology` Packages

MAgILOntology	This class defines the vocabularies and schemas of concepts and describes the basic structure of the ontology. Each concept represents an entity and is encoded in a class.

Appendix B

Launching A MAgIL Subsystem

This appendix presents:

- the syntax of the line command for launching an agent either on the JADE platform (e.g., for desktops or portable computers), or the JADE-LEAP platform for mobile devices such as smart phones;
- the logical steps to launch a concrete MAgIL subsystem;
- and the line command to launch a concrete MAgIL subsystem using SimToolkit.

B.1 Command Syntax to Launch an Agent

The agents in a concrete agent-based subsystem can be launched by using the syntax of a command specified in (Bellifemine et al., 2007) and described below.

B.1.1 JADE Environment

```
prompt> "java jade.Boot"  [<Option> [ ...n]] [<UserDefOption> [ ... n]]
                          [<AgentSpecifier> [ ... n]]


<Option>          ::=     Option_name [<OptionValue> [ ";" ... n] ]

<OptionValue>     ::=     Option_value


<UserDefOption>   ::=     Option_name [<OptionValue> ]


<AgentSpecifier>  ::=     Agent_name ":" <FullQualifiedClasseName>[(
                          <Argument> [ ... n] )]

<FullQualifiedClasseName>
                  ::=     Package_name "." Class_name
```

`Option_name`	a "-" symbol followed by one of the following key word: defined in JADE: container, host, port, local-host, local-port, name, container-name, gui, version, help, etc. A full list of key words is defined in (Bellifemine et al., 2007).
`Option_value`	a numeric or string value of `Option-name`.
`Agent_name`	the local name which the user attributes to agent upon instantiation.
`Package_name`	the name of package into which the agent classes are packaged.
`Class_name`	the class name of the agent which will be launched.
`Argument`	a numeric or string used to pass runtime configurable value for the agent, for example, the path to a properties file.

B.1.2 JADE-LEAP Environment

The command syntax for launching an agent in the JADE-LEAP environment is similar to that of the JADE environment. There are some minor differences which developers and users need to pay more attention to:

- The line command to launch an agent can be issued without option. In this case, the parameter "-agent" must be present followed by the desired agent name. For example, `java jade.Boot –agents myAgentName:MyAgentClass`.
- To start many agents simultaneously, the semicolon character (';') is used to separate theses agents instead of blanks (' ') in the line command.
- Arguments of an agent are separated by commas (',') instead of spaces (' '). No blank is allowed before or after the commas.

B.2 Steps To Launch a MAgIL Subsystem

The following steps present the chronological order to launch a MAgIL subsystem. For each step, the command can be either issued manually by end-users or invoked by SimToolkit.

Step 1: Start Agent Platform to prepare an agent platform for the MAgIL subsystem, for example, the JADE platform.

Step 2: Start LIS Agents and Service Agents to insure that the LIS Agents and Service Agents (e.g., AuditAgent) are in action in order to provide information and services for the personal agents.

Step 3: Start Personal Core Agents to launch the personal core agents in order to associate with end-users known by the administrator.

***Step 4: Start Personal UI* Agents** to launch the personal user interface agents to be associated with known end-users.

B.3 Launching a MAgIL Subsystem Using SimToolkit

The following line command executes the three steps described in the previous section. The input file simfile_name.xml contains the input parameters for the command "**`java jade.Boot …`**" invoked by SimToolkit during the simulation.

B.3.1 Command Syntax

```
prompt> java magil.sim.Simulator –simfile simfile_name.xml
```

`simfile_name``.xml` contains a model of the MAgIL subsystem with a simulation scenario. The simulation scenario consists of three phases: (1) launching the platform and agents; (2) simulating the agents' actions; (3) and shutting down the agents and platform. This `xml` file is constructed based on a `XML Schema`, named `simulation_schema.xsd` (see B.3.2).

The below `hellomas_simulation.xml`, in B.3.3, is a concrete example of *`simfile_name.`*`xml`. It presents a simulation scenario for the HelloMAS application discussed in Section 5.3.

B.3.2 simulation_schema.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified">
   <xsd:complexType name="propertyType">
      <xsd:simpleContent>
         <xsd:extension base="xsd:string">
            <xsd:attribute name="key" type="xsd:string"/>
         </xsd:extension>
      </xsd:simpleContent>
   </xsd:complexType>
   <xsd:complexType name="simElementType" abstract="true">
      <xsd:sequence>
         <xsd:element name="property" type="propertyType"
         maxOccurs="unbounded" minOccurs="0"></xsd:element>
      </xsd:sequence>
      <xsd:attribute name="id" type="xsd:string"/>
      <xsd:attribute name="desc" type="xsd:string"/>
   </xsd:complexType>
   <xsd:complexType name="agentType">
      <xsd:complexContent>
         <xsd:extension  base="simElementType"></xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>
   <xsd:complexType name="poolType">
      <xsd:complexContent>
         <xsd:extension base="simElementType">
            <xsd:sequence>
               <xsd:element name="agent" type="agentType"
               maxOccurs="unbounded" minOccurs="0"></xsd:element>
               <xsd:element name="pool" type="poolType"
               maxOccurs="unbounded" minOccurs="0"></xsd:element>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>
   <xsd:complexType name="phaseType">
      <xsd:complexContent>
         <xsd:extension base="simElementType">
            <xsd:choice>
               <xsd:sequence>
                  <xsd:element name="phase" type="phaseType"
                  maxOccurs="unbounded"></xsd:element>
               </xsd:sequence>
               <xsd:sequence>
                  <xsd:element name="agent" minOccurs="0"></xsd:element>
                  <xsd:element name="action" type="actionType"
                  minOccurs="0"></xsd:element>
               </xsd:sequence>
            </xsd:choice>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>
   <xsd:complexType name="simulationType">
      <xsd:complexContent>
         <xsd:extension base="simElementType">
            <xsd:sequence>
               <xsd:element name="population" type="poolType"></xsd:element>
               <xsd:element name="scenario" type="phaseType"></xsd:element>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>
   <xsd:complexType name="actionType">
      <xsd:complexContent>
         <xsd:extension base="simElementType"></xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>
   <xsd:element name="simulation" type="simulationType"></xsd:element>
</xsd:schema>
```

B.3.3 hellomas_simulation.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<simulation id="HelloMAS sim">
   <population>
      <pool id="jade_platform">
         <property key="type">core</property>
         <agent id="jade">
            <property key="gui" />
            <property key="jade_domain_df_autocleanup">true</property>
         </agent>
      </pool>
      <pool id="magil_framework">
         <property key="container">MAgIL:MainContainer</property>
         <property key="type">core</property>
         <agent id="audit_agent">
            <property key="name">AuditCoreAgent</property>
            <property key="class">magil.svcagent.auditagent.AuditAgent</property>
            <property key="profile">config/audit.xml</property>
         </agent>
      </pool>
      <pool id="hellomas_pools">
         <pool id="Bond">
            <property key="profile">config/Bond.xml</property>
            <property key="container">HelloMAS:Bond's Agents</property>
            <agent id="Bond_core">
               <property key="type">core</property>
               <property key="name">BondCoreAgent</property>
               <property key="class">hellomas.agent.jessableagent.JessableHelloPersonalCoreAgent</property>
            </agent>
            <agent id="Bond_gui">
               <property key="type">gui</property>
               <property key="name">BondUIAgent</property>
               <property key="class">hellomas.uiagent.HelloPersonalUIAgent</property>
            </agent>
         </pool>
         <pool id="Leiter">...</pool>
         <pool id="Lynd">...</pool>
         <pool id="JessUIAdmin">...</pool>
      </pool>
   </population>

   <scenario>
      <phase desc="prologue">
         <phase desc="launching JADE platform (RMA, AMS, DF)">...</phase>
         <phase desc="launching MAgIL's agent(AuditAgent)">...</phase>
         <phase desc="launching core population">...</phase>
         <phase desc="launching uiadmin population">...</phase>
         <phase desc="launching gui population">...</phase>
      </phase>

      <phase desc="logue">...</phase>

      <phase desc="epilogue">
         <phase desc="takedown gui population">...</phase>
         <phase desc="takedown uiadmin population">...</phase>
         <phase desc="takedown core population">...</phase>
         <phase desc="takedown MAgIL's agent (AuditAgent)">...</phase>
         <phase desc="takedown JADE platform (RMA, AMS, DF)">...</phase>
      </phase>
   </scenario>
</simulation>
```

References

Acronymics, Inc. (2004). Agent Builder – An Integrated Toolkit for Constructing Intelligent Software Agents. Retrieved April 4th, 2008, from http://www.agentbuilder.com.

Agent Oriented Software Pty, Ltd. (2006). JACK(TM) Intelligent Agents Agent Manual. Release 5.3. Retrieved May 2nd, 2009, from http://www.aosgrp.com/documentation/jack/Agent_Manual_WEB/index.html.

Azevedo, A. L., Toscano, C., & Bastos, J. (2002). An Intelligent Agent-Based Order Planning For Dynamic Networked Enterprises. In J. Filipe, B. Sharp & P. Miranda (Eds.), *Enterprise Information Systems III* (Vol. 3, pp. 124-131): Kluwer Academic Publishers.

Bakehouse, G., & Wakefield, T. (2005, 23th March). Legacy Information Systems. *ACCA, Student Accountant Magazine,* 5.

Beer, M., Fasli, M., & Richards, D. (2011). *Multi-agent Systems for Education and Interactive Entertainment: Design, Use and Experience*: IGI Global.

Bellifemine, F., Caire, G., & Greenwood, D. (2007). *Developing Multi-Agent Systems with JADE*. West Sussex, England: John Wiley & Sons Ltd.

Bianchi, A., Caivano, D., & Visaggio, G. (2003). Iterative Reengineering of Legacy Systems. *IEEE Transactions on Software Engineering, 29*(3), pp. 225-241.

Bisbal, J., Lawless, D., Wu, B., & Grimson, J. (1999a). *Legacy Information Systems Migration: A Brief Review of Problems, Solutions and Research Issues* (No. 38): Trinity College Dublin.

Bisbal, J., Lawless, D., Wu, B., & Grimson, J. (1999b). Legacy Information Systems: Issues and Directions. *IEEE Software, 16*(5), pp. 103-111.

Bordini, R. H., Dastani, M., & Seghrouchni, A. E. F. (2010). *Multi-agent Programming: Languages, Platforms and Applications*: Springer-Verlag New York Inc. .

Braun, P., & Rossak, W. (2005). *Mobile Agents: Basic Concepts, Mobility Models, and the Tracy Toolkit*. Heidelberg, Germany: dpunkt.verlag.

Brito, K. (2010). *Legacy Systems Maintenance*: VDM Verlag Dr. Muller Aktiengesellschaft & Co. KG.

Brodie, M. L. (1992). The Promise of Distributed Computing and the Challenges of Legacy Information Systems. In *Proceedings of the IFIP WG 2.6 Database Semantics Conference on Interoperable Database Systems (DS-5)* (Vol. 618/1992, pp. 1-31): Springer-Verlag.

Bui, X. T. (2000). Building agent-based corporate information systems: An application to telemedicine. *European Journal of Operational Research, 122*(2), pp. 242-257.

Clark, D. (2000). Shopbots Become Agents for Business Change. *IEEE Computer, 33*(2), pp. 18-21.

Cortés, U., Annicchiarico, R., Urdiales, C., Barrué, C., Martínez, A., Villar, A., et al. (2008). Supported Human Autonomy for Recovery and Enhancement of Cognitive and Motor Abilities Using Agent Technologies. In *Agent Technology and e-Health* (pp. 117-140). Basel, Switzerland: Birkhäuser Verlag.

Cummins, S., Davy, A., Finnegan, J., & Corroll, R. (2003). Abstract State of the Art: Middleware in Smart Space Management. *M-Zones State of the Art Paper, SOA paper 05/03, Ireland*.

Cycorp, Inc. (2001). OpenCyc. Retrieved May 6th, 2009, from http://www.cyc.com/cyc/opencyc/overview.

Daniel H. Wagner Associates, Inc. (2005). Software Agents. Retrieved May 6th, 2009, from http://www.wagner.com/technologies/softwareagents/softwareagents.html.

Davies, J., Studer, R., & Warren, P. (2006). *Semantic Web Technologies: Trends and Research in Ontology-based Systems*. West Sussex, England: John Wily & Sons Ltd.

DeLoach, S. (2004). The MaSE Methodology. In F. Bergenti, M.-P. Gleizes & F. Zambonelli (Eds.), *Methodologies and Software Engineering for Agent Systems* (pp. 107-125): Kluwer Academic Publishers.

Fasli, M. (2007). *Agent Technology for e-Commerce*. West Sussex, England: John Wiley & Sons Ltd.

Ferber, J. (1999). *Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence*. London, England: Addison-Wesley Professional.

Fiedrich, F., & Burghardt, P. (2007). Agent-based systems for disaster management. *Communications of the ACM, 50,* pp. 41-42.

Finin, T., Fritzson, R., McKay, D., & McEntire, R. (1994). KQML as an Agent Communication Language. In *Proceedings of the Third International Conference on Information and Knowledge Management (CIKM'94), November 29 - December 2, 1994*. Gaithersburg, Maryland: ACM Press.

FIPA (2002). FIPA Interaction Protocol Specifications. Retrieved December 18, 2009, from http://www.fipa.org/repository/ips.php3.

Franklin, S., & Graesser, A. (1996). Is it an Agent, or just a Program? A taxonomy for Autonomous Agents. In J. P. Müller, M. Wooldridge & N. R. Jennings (Eds.), *Proceeding of the Third International Workshop on Agent Theories, Architectures, and Languages* (Vol. 1193, pp. 21 - 35). London, United Kingdom: Springer-Verlag.

Friedman-Hill, E. (2003). *Jess in Action: Java Rule-Based System*: Manning Publications.

Galinium, M., & Shahbaz, N. (2012). *Migration of Legacy System to Service Oriented Architecture*: LAP Lambert Academic Publishing AG & Co KG.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. M. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. London, England: Addison-Wesley Professional.

Genesereth, M. R., & Fikes, R. E. (1992). *Knowledge Interchange Format*. Stanford, California: Logic Group, Computer Science Department, Stanford University.

Giorgini, P., Kolp, M., Mylopoulos, J., & Castro, J. (2005). Tropos: A Requirements-Driven Methodology for Agent-Oriented Software. In B. Henderson-Sellers & P. Giorgini (Eds.), *Agent-Oriented Methodologies* (pp. 20-45). London, United Kingdom: Idea Group Publishing.

González-Vélez, H., Mier, M., Julià -Sapé, M., Arvanitis, T., García-Gómez, J., Robles, M., et al. (2009). HealthAgents: distributed multi-agent brain tumor diagnosis and prognosis. *Journal of Applied Intelligence, 30*(3), pp. 191-202.

Gozdan, S. W. (2007). How big is your process automation gap? *Mortage Banking. ECT News Network.*

Greenwald, A. R., & Kephart, J. O. (1999). Shopbots and Pricebots. In *Proceeding of the Sixteenth International Joint Conference on Artificial Intelligence* (pp. 506-511). Stockholm, Sweden: Morgan Kaufmann Publishers, Inc.

Gutknecht, O., & Ferber, J. (2000). MadKit: a generic multi-agent platform. In *Proceedings of the fourth international conference on Autonomous agents* (pp. 78-79). Barcelona, Spain: ACM.

Gutknecht, O., & Ferber, J. (2001). *The MADKIT Agent Platform Architecture.* Paper presented at the International Workshop on Infrastructure for Multi-Agent Systems: Infrastructure for Agents, Multi-Agent Systems, and Scalable Multi-Agent Systems, Barcelona, Spain.

Gutknecht, O., Ferber, J., Michel, F., & Mansour, S. (2008). MADKIT Agent Platform. Retrieved May 3rd, 2009, from http://www.madkit.org.

Henderson-Sellers, B., & Giorgini, P. (2005). *Agent-Oriented Methodologies*. London, United Kingdom: Idea Group Publishing.

Hewitt, C. (1977). Viewing Control Structures as Patterns of Passing Messages. *Artificial Intelligence, 8*(3), pp. 323-364.

Hewitt, C., Bishop, P., & Steiger, R. (1973). A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence (IJCAI'73)* (pp. 235-245). Stanford, USA: Morgan Kaufmann Publishers Inc.

Howe, D. (1996). Legacy System. *Free On-line Dictionary of Computing (FOLDOC).* Retrieved May 5th, 2009, from http://foldoc.org/.

Huhns, M. N., & Singh, M. P. (1997). *Readings in Agents* (1st ed.). San Francisco, CA, USA: Morgan Kaufmann.

Iglesias, C. A., & Garijo, M. (2005). The Agent-Oriented Methodology MAS-CommonKADS. In B. Henderson-Sellers & P. Giorgini (Eds.), *Agent-Oriented Methodologies* (pp. 46-78). London, United Kingdom: Idea Group Publishing.

InfoSpace, Inc. (2009). MetaCrawlerWeb Search Home Page. Retrieved June 9th, 2009, from http://www.metacrawler.com.

Javet, D. (2007). Datamed SA - Informatique médicale et scientifique: WinDMLAB. Retrieved 09th June, 2009, from http://www.datamed.ch/.

Jedrzejowicz, P., Nguyen, N. T., Howlet, R. J., & Jain, L. C. (2010). *Agent and Multi-Agent Systems: Technologies and Applications: 4th KES International Symposium, KES-AMSTA 2010 Gdynia, Poland, June 23-25, 2010, Proceedings*: Springer-Verlag Berlin and Heidelberg GmbH & Co. K.

Jennings, N. R., & Wooldridge, M. (2010). *Agent Technology: Foundations, Applications, and Markets*: Springer-Verlag Berlin and Heidelberg GmbH & Co. K.

Kephart, J. O., & Greenwald, A. R. (2002). Shopbot Economics. *Autonomous Agents and Multi-Agent Systems, 5*(3).

Lin, H. (2007). *Architectural Design of Multi-Agent Systems: Technologies and Techniques* (1st ed.). London, United Kingdom: IGI Global.

Maes, P. (1994). Agents that Reduce Work and Information Overload. *Comm. of the ACM, 37*(7), pp. 30-40.

Martin, R. C. (2011). *Agile Software Development, Principles, Patterns, and Practices: International Edition*: Pearson.

McCarthy, J. (1987). Generality in Artificial Intelligence. *Comm. ACM, 30*(12), pp. 1030-1035.

Menczer, F., Street, W. N., & Vishwakarma, N. (2002). *IntelliShopper: A Proactive, Personal, Private Shopping Assistant.* Paper presented at the Proceedings of the first international joint conference on Autonomous agents and multiagent systems (AAMAS '02): part 3, Bologna, Italy.

Microsoft Patterns & Practices Team (2009). *Microsoft® Application Architecture Guide, 2nd Edition (Patterns & Practices)*: Microsoft Press.

Middleton, S. E. (2002). Interface agents: A review of the field. *CoRR, cs.MA/0203012*.

Mitchell, C. (2004). *Security for Mobility* (1st ed.). London, United Kingdom: The Institution of Electrical Engineers.

Morley, D., & Parker, C. S. (2010). *Understanding Computers: Today and Tomorrow, Comprehensive* (13 ed.): Course Technology.

Nevell, A. (1962). Some problems of basic organization in problem-solving programs. *The RAND Corporation, Santa Monica, California.*

Nguyen, M. T. (2009). MAgIL : Multi-Agent Integration Layer. from http://diuf.unifr.ch/drupal/softeng/projects/magil/index.htm.

Nguyen, M. T., Fuhrer, P., & Pasquier, J. (2008). Enhancing E-Health Information Systems with Agent Technology. *International Journal of Telemedicine and Applications, Electronic Health Special Issue, 2009* (Article ID 279091), 13 pages.

Nikraz, M., Caire, G., & Bahri, P. A. (2006). A Methodology for the analysis and design of multi-agent systems using JADE. *International Journal of Computer Science and Engineering, 21*(6).

Nwana, H. S. (1996). Software Agents: An Overview. *Knowledge Engineering Review, 11*(3), pp. 205-244.

Odell, J. (2002). Objects and Agents Compared. *Journal of Object Technology, 1*, pp. 41-53.

Odell, J. (2007). FIPA-The Foundation for Intelligent Physician Agents. Retrieved May 2nd, 2009, from http://www.fipa.org.

Opalis Software, Inc. (2007). Closing the IT Process Automation Gap. Retrieved May 3rd, 2009, from http://www.opalis.com.

Oracle Corporation, I. (2006). *Unlocking The Mainframe: Modernizing Legacy Systems to a Service Oriented Architecture.* (White Paper): Oracle Corporation Ltd.

Papazoglou, M. P. (1999). The Role of Agent Technology in Business to Business Electronic Commerce. In *Proceedings of the 3rd International Conference on Cooperative Information Agents III (CIA'99)* (pp. 245-264). Uppsala, Sweden: Springer-Verlag Berlin.

Paradauskas, B., & Laurikaitis, A. (2006). Business knowledge extraction from legacy information systems. *Information Technology and Control, 35*(3), pp. 214-221.

Pointet, B. (2008). *MediMASim: A Test and Simulation Toolkit for the MediMAS Application.* University of Fribourg, Fribourg, Switzerland.

Protogeros, N. (2007). *Agent and Web Service Technologies in Virtual Enterprises*. Hershey, USA: IGI Global.

Ruppen, A. (2007). *Systèmes multi agents: MediMAS - étude de cas dans le domaine du E-Health care.* Université de Fribourg, Fribourg, Suisse.

Russell, S., & Norvig, P. (2002). *Artificial Intelligence: A Modern Approach* (2nd ed.): Prentice Hall.

Sallam, A. (2011). *Integration of Web Services and Agent Technologies*: LAP Lambert Academic Publishing.

Schaeppi, J. (2008). *Extension de MediMAS: développement et déploiement d'agents JADE sur des supports mobiles.* Université de Fribourg, Fribourg, Suisse.

Selker, T. (1994). COACH: a teaching agent that learns. *Commun. ACM, 37*(7), 92-99.

Serrano, E., Botia, J. A., & Cadenas, J. M. (2011). *Advancing the state of the art in the analysis of multi-agent systems: Study and Development of Methods and Tools for Testing, Validation and Verification of Multi-Agent Systems*: LAP LAMBERT Academic Publishing.

Smith, R. G. (1977). The Contract Net: A Formalism for the Control of Distributed Problem Solving. In *Proceedings of the 5th international joint conference on Artificial intelligence - Volume 1 (IJCAI'77)* (Vol. 1, pp. 472-472). Cambridge, USA: Morgan Kaufmann Publishers Inc.

Software Engineering Group (2009). MediMAS (Medical Multi-Agent Systems) Retrieved June 22th, 2009, from http://diuf.unifr.ch/softeng/projects/medimas/.

Sommerville, I. (2007). *Software Engineering* (8th ed.). Harlow, England: Pearson Education Limited.

Stanford Center for Biomedical Informatics Research (2007). The Protégé Ontology Editor and Knowledge Acquisition System. Retrieved May 4th, 2009, from http://protege.stanford.edu.

Stanford University (2008). Ontolingua. Retrieved May 6th, 2009, from http://www.ksl.stanford.edu/software/ontolingua.

Steve, L., Lee, G., & Kurt, B. (1997). Computer and Information Science Papers CiteSeer Publications ResearchIndex. Retrieved June 9th, 2009, from http://citeseer.ist.psu.edu.

Stonebraker, M., & Brodie, M. L. (1995). *Migrating Legacy Systems: Gateways, Interfaces & the Incremental Approach* (1st ed.): Morgan Kaufmann Publishers.

Sun Microsystems, Inc. (1994). Developer Resources for Java Technology. Retrieved September 12th, 2009, from http://java.sun.com.

Sycara, K., & Zeng, D. (1996). Coordination of Multiple Intelligent Software Agents. *International Journal of Cooperative Information Systems, 5*(2-3), pp. 181-212.

Thiran, P., Hainaut, J.-L., Houben, G.-J., & Bendlimane, D. (2006). Wrapper-Based Evolution of Legacy Information Systems. *ACM Transactions on Software Engineering and Methodology 15*(4), pp. 329-359.

Tolchinsky, P., Cortés, U., & Grecu, D. (2008). Argumentation-Based Agents to Increase Human Organ Availability for Transplant. In *Agent Technology and e-Health* (pp. 65-93). Basel, Switzerland: Birkhäuser Verlag.

TopQuadrant, Inc. (2007). TopBraiComposer. Retrieved March 16th, 2009, from http://topbraidcomposer.com/index.html.

van Aart, C. J. (2007, February 27th, 2009). Ontology Bean Generator. Retrieved May 4th, 2009, from http://protege.cim3.net/cgi-bin/wiki.pl?OntologyBeanGenerator.

Vogt, J. (2008). *Jess to JADE Toolkit (J2J) - A Rule-Based Solution Supporting Intelligent and Adaptive Agents.* University of Fribourg, Fribourg, Switzerland.

Vrba, P. (2003). JAVA-Based Agent Platform Evaluation. *Holonic and Multi-Agent Systems for Manufacturing, 2744/2003*, pp. 47-58.

Waldeck, R. (2005). Prices in a Shopbot Market. *Electronic Commerce Research, 5*(3-4), pp. 367-381.

Weiss, G. (2000). *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. London, England: The MIT Press.

Weyns, D., & Gleizes, M.-P. (2011). *Agent-Oriented Software Engineering: 11th International Workshop, Aose XI, Toronto, Canada, May 10-11, 2010, Selected Papers*: Springer-Verlag Berlin and Heidelberg GmbH & Co. K.

Wooldridge, M. (2002). *An Introduction to MultiAgent Systems* (1st ed.). Chichester, England: John Wiley & Sons Ltd.

Wooldridge, M., & Jennings, N. R. (1995). Intelligent agents: Theory and practice. *Knowledge Engineering Review, 10*(2), pp. 115-152.

Wooldridge, M., & Jennings, N. R. (Eds.). (1994). *Intelligent Agents: ECAI-94 Workshop on Agent Theories, Architectures, and Languages* (1st ed. Vol. 890). Amsterdam, Netherlands: Springer-Verlag.

Wooldridge, M., Jennings, N. R., & Kinny, D. (2000). The Gaia Methodology for agent-oriented analysis and design. *Autonomous Agents and MultiAgent Systems (AAMAS), 3*(3), pp. 285-312.

Wu, B., Lawless, D., Bisbal, J., Grimson, J., Wade, V., O'Sullivan, D., et al. (1997). Legacy System Migration: A Legacy Data Migration Engine. In *Proceedings of the 17th International Database Conference (DATASEM '97)* (pp. 129-138). Brno, Czech Republic: Ed. Czechoslovak Computer Experts.

Wu, B., Lawless, D., Bisbal, J., Richardson, R., Grimson, J., Wade, V., et al. (1997). The Butterfly Methodology: A Gateway-free Approach for Migrating Legacy Information Systems. In *Proceedings of the 3rd IEEE Conference on Engineering of Complex Computer Systems (ICECCS97)* (pp. 200-205). Villa Olmo, Como, Italy: IEEE Computer Society.

Zambonelli, F., Jennings, N. R., & Wooldridge, M. (2005). Multi-Agent Systems as Computational Organizations: The Gaia Methodology. In B. Henderson-Sellers & P. Giorgini (Eds.), *Agent-Oriented Methodologies* (pp. 136-171). London, United Kingdom: Idea Group Publishing.

Zamir, O., & Etzioni, O. (1999). Grouper: a dynamic clustering interface to Web search results. *Computer Networks, 31*(11-16), pp. 1361-1374.

Zhang, Y., Ghenniwa, H., & Shen, W. (2006). Agent-based Personal Assistance in Collaborative Design Environments. In LNCS (Ed.), *Computer Supported Cooperative Work in Design II* (Vol. 3865, pp. 284-293). Berlin, Germany: Springer-Verlag.

Zimmerman, A. T. (2011). *Agent-Based Computational Architectures for Distributed Data Processing in Wireless Sensor Networks*: ProQuest, UMI Dissertation Publishing.