DESIGNING MULTI-AGENT UNIT TESTS USING SYSTEMATIC TEST DESIGN PATTERNS

Khaled M. Nagi[1] and Mohamed A. Khamis[2]
[1]*Department of Computer and Systems Engineering, Faculty of Engineering, University of Alexandria.*
[2]*Computer Science and Engineering, Egypt-Japan University of Science and Technology (E-JUST)*

ABSTRACT

Following the Test-Driven Development (TDD) paradigm, we present a framework that allows Multi-Agent System (MAS) developers to write test scenarios that test each agent individually. The framework relies on the concepts of building *mock agents* and testing common *agent interaction design patterns*. We analyze the most common agent interaction patterns including *pair* and *mediation* patterns in order to provide stereotype implementation for their corresponding test cases. These implementations serve as test building blocks and are provided as a set of ready-for-reuse components in our repository. This way, the developer can concentrate on testing the business logic itself and spare him/her the burden of implementing tests for the underlying agent interaction patterns. Our framework is based on standard components such as the JADE agent platform, the junit framework, and the eclipse plug-in architecture. In this paper, we present *the design and function* of the framework. We demonstrate how we can use the proposed framework to define more *stereotypes* in the code repository and provide an *analysis of the code coverage* for our designed stereotype test code implementations.

KEYWORDS

Test-Driven Development, Mock Agent, Agent Social Design Patterns, Code Generation, Code Coverage.

1. INTRODUCTION

Agents interact in a concurrent, asynchronous and decentralized manner (Huget and Demazeau 2004), making MAS complex systems (Jennings and Nicholas 2001). They are also non-deterministic since it is very difficult to determine a priori all interactions of an agent during its execution. Consequently, they are difficult to debug and test. Much effort has been made in the past to identify common interactions between agents and define their design patterns. However, defining standard tests for these interactions have not yet been defined. There are five approaches to testing multi-agent systems: testing, runtime monitoring, static analysis, model checking, and theorem proving (Timm et al. 2006). In this paper, we specifically deal with the *testing* approach. Our aim is to let the developer concentrate on testing the business logic rather than the underlying MAS framework and the different interactions between the participating agents.

During our project, we design test patterns for the *ten* most famous MAS social design patterns found in literature, such as the *call-for-proposal agent* interaction design pattern (Coelho et al. 2006) and the *matchmaker agent* interaction design pattern (Silva et al. 2005). We design and implement an *eclipse plug-in* to enable the MAS unit test developer to generate a *mock agent* interacting with the Agent Under Test (AUT) following one of the identified social design patterns. The agents and the mock agents run within the JADE Platform (Bellifemine et al. 2005). The MAS unit test developer has the ability to add further test design patterns for an existing agent interaction pattern or for a newly identified one. This way, the MAS developer focuses on testing the business logic of the MAS without the burden of implementing unit tests for the implemented design pattern itself.

The framework provides the first implementation that usually triggers the continuous refactoring process typical to TDD. The developer uses the reflection capabilities of the eclipse SDK to reflect the changes made in the AUT in the generated mock agent. The *repository* consists of a set of XML and java files that represent the behavior of the different mock agents existing in the implemented design patterns. In our work, we provide implementations for a vast majority of agent design patterns. We evaluate the code coverage by

using EMMA (EMMA 2006), a code coverage tool, to demonstrate that the generated test files (mock agent, associated resource files, and AUT test cases) completely cover the AUT code for the agent interaction pattern.

The remainder of this paper is organized as follows. Section 2 gives background on the MAS unit testing approach using mock agents. Section 3 illustrates our contribution by giving detailed design of our framework used in designing test cases based on mock agents. Section 4 presents the design of the test design patterns. Section 5 analyzes the quantitative results using EMMA. Finally, Section 6 presents our conclusions and future work.

2. BACKGROUND

2.1 MAS Unit Testing Approach using Mock Agents

The adopted MAS testing approach tests the smallest building blocks of MAS, namely, *the agent*. Its basic idea is to verify whether each agent in isolation respects its specification under *successful* and *exceptional* scenarios. A *Mock Agent* is a regular agent that communicates with just one other agent: the *Agent Under Test* (AUT). It has a plan to test the AUT. The Mock Agent plan is equivalent to a test script, since it defines the messages that should be sent to the AUT and asserts the messages that should be received from it (Coelho et al. 2006).

The agent unit testing approach has two main concerns: the design of a test case based on the use of Mock Agents; *and* the test case execution, which relies on the Agent Monitor to notify when the test script is concluded. These two concerns are analyzed in (Coelho et al. 2006) and are summarized below.

Unit test approaches for MAS proposed so far do not define a methodology for test-case selection. However, a test-case design technique is presented in (Myers 2004), which is based on error-guessing. The basic idea of an error-guessing technique is to enumerate a list of possible error-prone situations and then write test cases based on the list (Coelho et al. 2006). The following pseudo-code sketches the approach.

```
1. For the agent to be tested
1.1. List the set of roles that it plays
2. For each role played by the AUT
2.1. List the set of other roles that interacts with it
3. For each interacting role
3.1. Implement in the Mock Agent a plan that codifies a successful scenario.
3.2. List possible exceptional scenarios that the Mock Agent can take part.
3.3. Implement in the Mock Agent an extra plan that codifies each exceptional scenario.
```

The plan of a Mock Agent comprises the logic of the test. Each test case starts the AUT and the corresponding Mock Agents and waits for a notification from the Agent Monitor – informing when the interaction between the agents has finished – in order to ask the Mock Agents whether or not the AUT acted as expected (Coelho et al. 2006).

2.2 Agent Social Design Patterns

MAS Designers are usually guided by a catalogue of multi-agent patterns that offer a set of standard interactions. *Tropos* (Castro et al. 2002) is a requirements-driven framework that proposes a set of design patterns, named *Social Patterns* (Kolp et al. 2002), focusing on social and intentional aspects that are recurrent in multi-agent and cooperative systems. The framework presented in (Kolp et al. 2005) has classified them into two categories: *pair* patterns and *mediation* patterns (Silva et al. 2005). Pair patterns (such as booking, call-for-proposal, subscription, or bidding) describe direct interactions between negotiating agents. Mediation patterns **(**such as monitor, broker, matchmaker, mediator, embassy, or wrapper) feature intermediary agents that help other agents reach an agreement on an exchange of services. In our work, we consider each of these design pattern and design suitable testing scenarios following the methodology mentioned in Section 2.1.

3. MOCK AGENT TEST CASES DESIGN FRAMEWORK

We adopt the *agile software engineering* approach that addresses testing continuously within the implementation process (Beck and Andres 2004). Developers formulate *test-cases* during or even before implementation, which are automatically executed on demand. This procedure is also known as *"*Test-driven Development*"* (Beck 2002). We focus on the *unit test level* where the implementation is tested by *unit test* during the coding.

3.1 Component Interaction

Figure 1 presents a collaboration diagram that illustrates the interaction between the components of our framework:

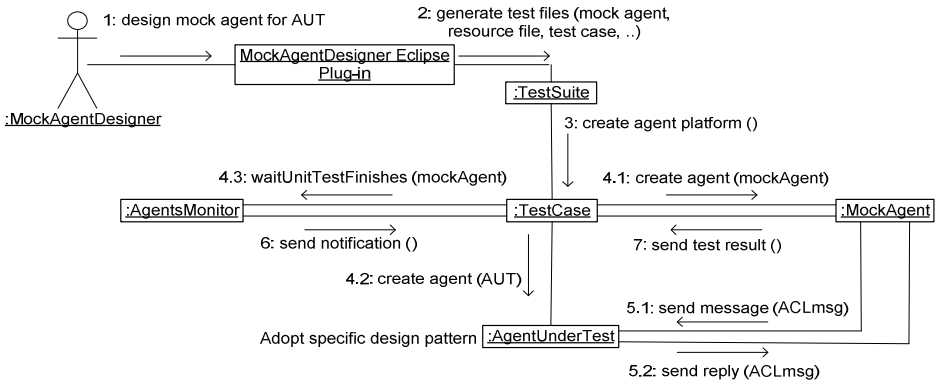

Figure 1 Collaboration diagram of the components of the proposed framework

The *MockAgentDesigner* designs the mock agent for the AUT (step 1) using our eclipse plug-in. The plug-in generates the necessary files employed in the *TestSuite* (step 2). The files include the Mock Agent java source code file, the associated resource files and AUT test cases following the design pattern selected from the plug-in repository. In step 3, the TestSuite creates the agent platform and whatever element needed to set up the test environment. Afterwards, the framework instantiates the *TestCase*. During the TestCase execution, the framework creates *MockAgent* to every role that interacts with the *AgentUnderTest* in the scenario defined by the TestCase (step 4.1). Next, it creates the AUT (step 4.2) and asks the *AgentsMonitor* to be notified when the interaction between the AUT and the MockAgent is complete (step 4.3). At this point, the AUT and the MockAgent begin to interact. The MockAgent sends a message to the AUT, and it replies (steps 5.1 and 5.2) or vice-versa. They repeat steps 5.1 and 5.2 as many times as necessary to perform the test codified in the mock agent plan. When a MockAgent concludes its plan, the AgentsMonitor notifies the framework that the interaction between the MockAgent and the AUT has concluded (step 6). Last, the TestCase records whether or not the AUT acted as expected (step 7).

3.2 System Design

The design of our proposed framework is illustrated in Figure 2. *TestRunner* is used to execute the *Test*. The test results are collected in *TestResult*. Tests can be grouped in *TestSuite*. A *TestCase* defines the fixture to run multiple tests. The *Agent* class is the common super class for user-defined software agents. It provides methods to perform basic agent tasks, such as message passing, both unicast and multicast with optional pattern matching. The class *ContainerController* is a proxy class, allowing access to a JADE agent container and the *AgentController* is a Proxy class, allowing access to a JADE agent.

The *JADETestCase* constitutes the heart of our framework. It consists of a set of *Test*s and a set of operations to be performed to prepare the test environment before a test case is executed. The *createEnvironment()* and *createAgent()* method is called inside the *JADETestCase* constructor. It is responsible for creating the JADE environment that will be active during the execution of all test methods.

The *AgentMonitor* class is responsible for monitoring agent life cycle in order to notify the *JADETestCase* about agent states. To fulfill this requirement, *AgentMonitor* implements the following methods: *waitUntilAllAgentsDie()*, *waitUntilTestFinishes()*, and *waitUntilAgentDies()* (*similar* to (Coelho et al. 2006)). The *AgentManager* class is an auxiliary class that contains empty-body implementations of all *wait** methods defined in the *AgentMonitor* class. The *JADEMockAgent* plan (equivalent to a JADE Behavior) is analogous to a test script, since it defines the messages that should be sent to the AUT and which messages that should be received from it. The *JADEMockAgent* implements two methods: *sendMessage()* which triggers the test method and *receiveMessage()*. The *receiveMessage()* method performs assertions concerning the received message (e.g., whether the message was received within a specific timeout, or if it obeys a pre-defined format). During execution, the *JADEMockAgent* needs to report the test result (success or failure) to the *TestCase*, which in counterpart, will be in charge of examining the test result. For reporting the result of the test, the *JADEMockAgent* class implements the *TestResultReporter* interface.

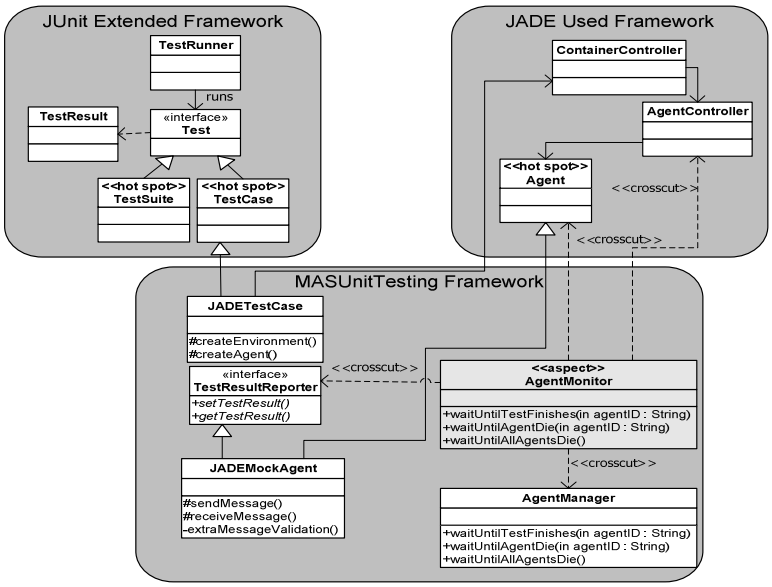

Figure 2. UML class diagram for the proposed framework

3.3 Definition of Test Case Stereotypes and their Usage

In our proposed framework, we offer an extendible set of pattern categories. Every pattern category contains a set of agent interaction design patterns. In each of these design patterns, a set of agent roles can be defined. Every agent role has a set of interacting roles, and every interacting role can interact with more than one agent role.

4. DEFINING TEST CASE STEREOTYPES WITH THE ECLIPSE PLUG-IN

4.1 Example: the Booking Pattern

Due to space restriction, we provide only a simplified sample for designing test patterns for the booking design pattern. The intent of the booking pattern is to let a client issue a request to book a resource from a service provider. Two roles are identified: the *booking client* and the *service provider*. In this section, we

elaborate only on the booking client role, which would be the AUT. Thus, the mock agent would be the service provider agent.

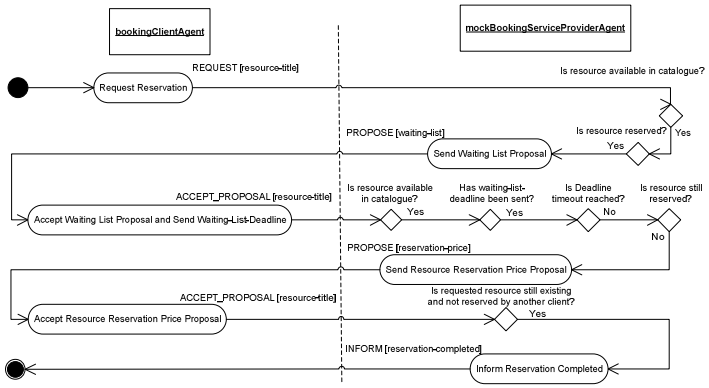

Figure 3. Sequence diagram for a successful scenario

Figure 3 illustrates the sequence diagram of the successful scenario in which the booking interaction successfully completes. The interaction begins with sending the request reservation. The mock agent simulates a check for availability of the resource in the catalogue and assumes its reservation, thus issuing a waiting-list proposal. The AUT normal behavior is to accept the proposal and to send a waiting list deadline. The mock agent simulates a re-check for availability, assumes the readiness of the resource, checks that the deadline has not been reached and sends a resource reservation price proposal with an unbeatable price (e.g., 0,-€). The AUT should accept the price proposal and send a notification to the mock agent. Immediately, the mock agent sends an inform reservation completed message to the AUT in order to avoid any timeouts from the side of the AUT.

In the exceptional scenarios, all successful assumptions previously made by the mock agent will be reverted, one at a time. So, the first exceptional behavior is illustrated in Figure 4a. in which the mock agent sends a resource unavailable and responds immediately after receiving the reservation request. The agent monitor asserts that the MAS terminates correctly under this scenario. The second exceptional scenario is illustrated in Figure 4b in which the mock agent sends a reservation failure message upon receiving the accept reservation price proposal message from the AUT. This analysis is carried on untill we define five exceptional scenarios based on the successful one.

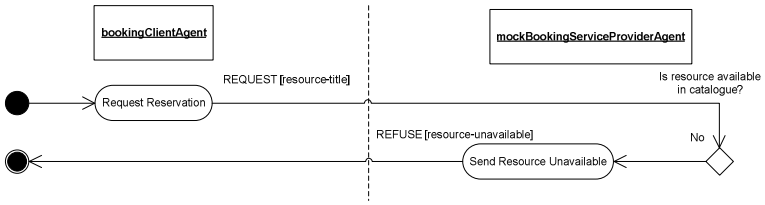

Figure 4a. Sequence diagram for the first exceptional scenario

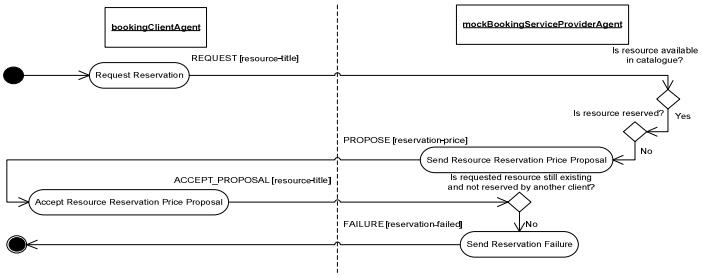

Figure 4b. Sequence diagram for the second exceptional scenario

In order to complete the test suite for this interaction pattern, reversing the roles of the AUT and the mock agent pattern are also done. It is simply omitted here due to space limitation.

4.2 Using the Eclipse Plug-In

Figure 5 presents the entry point for defining new stereotypes for testing agent interaction design patterns. The user provides the necessary information defined in Section 3.3 and is guided through a wizard-based plug-in to reach the central management screen illustrated in Figure 6.

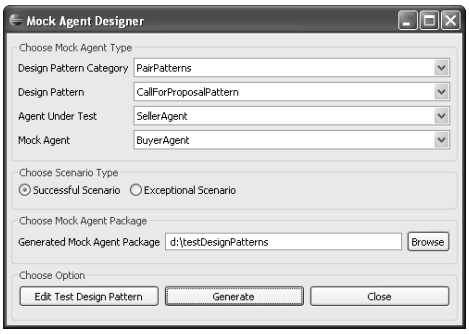

Figure 5. The entry point the Eclipse Plug-in

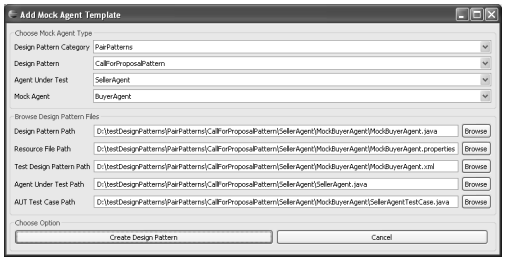

Figure 6. Add new Mock Agent Template

The generated java source code already contains the necessary **`imports`** (e.g., Figure 7) as well as the **`method`** definitions needed as part of the design highlighted in Section 3.2. The generated code is guaranteed to be free of compilation errors. The agent interaction performatives as well as resource values are defined in a properties file. The mock agent designer can change the values of these. The definition of the test scenario is defined in a separate XML file that describe the fixture and the test scenarios, etc. in a declarative manner.

```
1  /**
2   * Design Pattern Category:     PairPatterns
3   * Design Pattern:              CallForProposalPattern
4   * Agent Under Test:           SellerAgent
5   * Mock Agent:                 MockBuyerAgent
6   */
7  package PairPatterns.CallForProposalPattern.SellerAgent.MockBuyerAgent;
8
9  import java.util.MissingResourceException;
10 import java.util.ResourceBundle;
11
12 import jade.core.AID;
13 import jade.core.behaviours.Behaviour;
14 import jade.core.behaviours.TickerBehaviour;
15 import jade.lang.acl.ACLMessage;
16 import jade.lang.acl.MessageTemplate;
17
18 import junit.framework.TestResult;
19
20 import MASUnitTesting.JADEMockAgent;
21 import MASUnitTesting.ReplyReceptionFailed;
22
23 public class MockBuyerAgent extends JADEMockAgent {
24 }
```

Figure 7. Package and import structure in the generated MockBuyerAgent template

4.3 Current Repository Contents

Currently, we designed and implemented the mock agents for *ten* interaction design patterns:

- Pair patterns: such as booking (Do and Kolp 2003), call-for-proposal (Coelho et al. 2006), subscription (Do et al. 2003), and bidding (FIPA 2005)
- Mediation patterns: such as monitor (Kolp et al. 2005), broker (Do et al. 2003), matchmaker (Silva et al. 2005), mediator (Kolp et al. 2005), embassy (Hayden et al. 1999), and wrapper (Kolp et al. 2005)

5. EVALUATION

In this section, we use EMMA (EMMA 2006), a code coverage tool, to prove that the execution of the generated test files within our proposed framework actually provides good code coverage. *Code coverage* refers to a software engineering technique whereby the mock agent test case designer tracks quality and comprehensiveness of the test suite by determining simple metrics like the percentage of classes, methods, lines executed when the test suite ran. In Table 1, we illustrate the line coverage for the ten interaction design patterns we implemented with our framework.

Table 1. EMMA Code Coverage Results Statistics

Pattern Type	Pattern Name	AUT – (Line %)	Mock Agent – (Line %)	Test Case – (Line %)
Pair	Booking pattern	client **(98%)**	service provider (91%)	client (77%)
		service provider **(98%)**	client (95%)	service provider (76%)
Pair	Subscription pattern	yellow page **(100%)**	subscription provider (95%)	yellow page (83%)
		subscription provider **(100%)**	yellow page (88%)	subscription provider (84%)
Pair	Call-For-Proposal	buyer **(95%)**	seller (83%)	buyer (76%)
		seller **(89%)**	buyer (91%)	seller (74%)
Pair	Bidding pattern	bidder **(95%)**	auctioneer (96%)	bidder (79%)
		auctioneer **(98%)**	bidder (90%)	auctioneer (82%)
Mediation	Monitor pattern	notification subscriber **(97%)**	monitor (90%)	notification subscriber (74%)
		monitor **(100%)**	notification subscriber (89%)	monitor (78%)
			notification source (88%)	monitor (78%)
		notification source **(95%)**	monitor (96%)	notification source (75%)
Mediation	Broker pattern	client **(97%)**	broker (93%)	client (74%)
		broker **(97%)**	client (93%)	broker (76%)
			service provider (95%)	broker (77%)
		service provider **(98%)**	broker (94%)	service provider (75%)
Mediation	Matchmaker pattern	client **(95%)**	matchmaker (91%)	client (78%)
			matchmaker provider (87%)	client (78%)
		matchmaker **(95%)**	client (87%)	matchmaker (77%)
			matchmaker provider (96%)	matchmaker (77%)
		matchmaker provider **(98%)**	matchmaker (93%)	matchmaker provider (79%)
			client (89%)	matchmaker provider (79%)
Mediation	Mediator pattern	client **(91%)**	mediator (87%)	client (77%)
		mediator **(94%)**	client (86%)	mediator (82%)
			mediator provider (93%)	mediator (79%)
		mediator provider **(98%)**	mediator (94%)	mediator provider (77%)
Mediation	Embassy pattern	foreign **(97%)**	embassy (95%)	foreign (82%)
		embassy **(97%)**	foreign (93%)	embassy (83%)
			local (87%)	embassy (84%)
		local **(94%)**	embassy (92%)	local (80%)
Mediation	Wrapper pattern	client **(96%)**	wrapper (93%)	client (79%)
		wrapper **(98%)**	client (89%)	wrapper (82%)
			source (87%)	wrapper (82%)
		source **(94%)**	wrapper (91%)	source (79%)

In summary, the above table indicates that all AUT code lines have a minimum code coverage of 89% and an average of 96%. EMMA also offers the possibility of highlighting lines of code that are not covered during the execution of the test suites. By investigating these lines, it appears that they occurred outside both normal and exceptional scenarios, such as null arguments passed to the AUT classes, some java FIPA exceptions and catch exception blocks.

6. CONCLUSION AND FUTURE WORK

In this paper, we introduce a framework for designing test cases based on a unit testing approach for MAS. Our approach relies on the use of *mock agents* for testing common *agent design patterns*. With the introduced framework and toolset, the developer can concentrate on testing the business logic itself and spare him/her the burden of implementing tests for the underlying agent interaction patterns. In our work, we provide implementation of test stereotypes for a vast majority of agent interaction design patterns. We also demonstrate how to use our toolset to implement further stereotypes and how to add them to the available

repository. The code coverage analysis reveals good values in terms of lines of code covered in the ten *social design patterns* we initially support. Having these blocks in a reusable repository enables MAS test designers to concentrate on designing further tests with a good head start.

For the future, we would like to investigate the problems that arise while working with concurrent threads such as racing and deadlocks. Traditional code coverage metrics, such as statement, block, and branch, fail to address test adequacy concerns introduced by concurrency. Finally, we would like to implement stereotypes for other agent design patterns.

REFERENCES

Beck, K., 2002. *Test Driven Development by Example*. Addison-Wesley Professional, Boston, USA.

Beck, K. & Andres, C., 2004. *Extreme Programming Explained*. Addison-Wesley Professional, Boston, USA.

Bellifemine, F. et al, 2005. Jade Programming tutorial - JADE 3.3. Available at: http://sharon.cselt.it/projects/jade/.

Castro, J. et al, 2002. Towards Requirements-Driven Information Systems Engineering: The Tropos Project. *Information Systems - The 13th international conference on advanced information systems engineering (CAiSE*01)*. Elsevier Science Ltd, Vol. 27, No. 6, pp. 365-389.

Coelho, R. et al, 2006. Unit Testing in Multi-agent Systems using Mock Agents and Aspects. In: *Proceedings of the 2006 International Workshop on Software Engineering for Large-Scale Multi-Agent Systems*, *ACM*. New York, pp 83-90.

Do, T. T. & Kolp, M., 2003. Social Patterns for Designing Multiagent Systems. *In Proceedings of the 15th International Conference on Software Engineering & Knowledge Engineering*. San Francisco, USA, July 2003, pp. 103-110.

Do, T. T. et al, 2003. A Framework for Design Patterns in Tropos. *In Proceedings of the 17th Brazilian symposium of software engineering*. MAUNAS.

EMMA, 2006. *EMMA Code Coverage Tool*. Available at: http://emma.sourceforge.net/

FIPA, 2005. *protocol documentation*, 2005. Available at: http://www.fipa.org/.

Hayden, S. C. et al., 1999. Architectural Design Patterns for Multiagent Coordination. *In Proceedings of the 3rd International Conference on Autonomous Agents.*

Huget, M.-P. & Demazeau, Y., 2004. Evaluating multiagent systems: a record/replay approach. Intelligent Agent Technology. *In Proceedings. IEEE/WIC/ACM International Conference on 2004*, pp: 536 - 539.

Jennings & Nicholas R., 2001. An Agent-Based Approach for Building Complex Software Systems, *Communications of the ACM*, 44(4), pp: 35-41.

Myers, G. J., 2004. *The Art of Software Testing*. John Wiley & Sons, New Jersey, USA.

Kolp, M. et al, 2002. Information Systems Development through Social Structures, In: *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering*. Ishia, Italy, pp. 183 – 190.

Kolp, M. et al, 2005. *Introspecting Agent Oriented Design Patterns, In S. K. Chang (Eds), Advances in Software Engineering and Knowledge Engineering*. World Publishing, Omaha, USA.

Silva, C. et al, 2005. Describing Agent-Oriented Design Patterns in Tropos. *In Proceedings of the 19th Brazilian Symposium in Software Engineering.* Uberlandia, Minas Gerais, Brazil, pp 27-78.

Timm, I. J. et al, 2006. *Multiagent Engineering*. Springer, New York, USA.