



Alexandria University  
Faculty of Engineering

# **Designing Multi-Agent Unit Tests Using Systematic Test Design Patterns**

A thesis submitted to the Computer and Systems Engineering  
Department in partial fulfillment of the requirement for the degree of  
Master of Science  
(Computer Science)

**By:**

**Eng. Mohamed Abd El Aziz Khamis**

**Supervised by:**

**Prof. Dr. Mohamed Nazih El-Derini**

**Prof. Dr. Khaled Magdy Nagi**

Registered: 2005  
Submitted: 2010

## **CURRICULUM VITAE**

**Name:** Mohamed Abd El Aziz Khamis

**Place of Birth:** Alexandria, Egypt

**Nationality:** Egyptian

**Home Address:**

36 Rawdat El Rahman Street,  
WEBCO division, Betash, Agamy,  
Alexandria, Egypt.

**Email Address:**

[mohamed.abdelaziz@globalimpactsw.com](mailto:mohamed.abdelaziz@globalimpactsw.com)

**Current Position:**

Project Manager at:  
Global Impact Software,  
23 Fawzy Moaz Street, Semouha,  
Alexandria, Egypt.

**Educational Record:**

- General Secondary Certificate: 2000, Zahran Language School, Alexandria, Egypt.
- B.Sc. (Hon.): 2005, Department of Computer and Systems Engineering, Faculty of Engineering, Alexandria University, Alexandria, Egypt.

## **SUPERVISORS**

### **Prof. Dr. Mohamed Nazih El-Derini**

Professor of Computer Science  
Faculty of Engineering,  
Alexandria University,  
Alexandria, Egypt.

### **Prof. Dr. Khaled Magdy Nagi**

Professor of Computer Science  
Faculty of Engineering,  
Alexandria University,  
Alexandria, Egypt.

## **ABSTRACT**

Multi-Agent Systems (MASs) is the branch of Artificial Intelligence (AI) that views intelligence as the result of a complex structural arrangement of multiple independent autonomous interacting agents; in other words it is a social theory of intelligence. Multi-agent systems gained much interest in recent years because of the increasing complexity, openness, and distributivity of current and future systems.

As intelligent agent-based systems take over operations in the financial community, transportation, manufacturing, utilities, aerospace, and the military, assurances will need to be given to the owners and operators of these systems assuring that these non-deterministic learning systems operate correctly.

One important class of problems in multi-agent systems is testing such autonomous entities, where requirements have to be specified primarily as goals or context for agent behavior. The main challenge for intelligent agent testing arises from the agent flexibility.

This thesis addresses a MAS unit testing approach which aims at removing faults along the application development phase. We present a tool for designing test cases to help MASs developers in testing each agent individually. It relies on the use of Mock Agents to guide the design and implementation of agent unit test cases. The implementation of this approach is built on top of JADE platform. This approach extended JUnit test framework in order to execute JADE test cases.

We design test patterns for the ten most famous MAS social design patterns found in the literature of Agent Design Patterns. For the sake of generality and extensibility, we implement an Eclipse plug-in that gives the ability to the MAS unit test developer to generate the Mock Agent that interacts with his/her Agent Under Test (AUT) through a set of design patterns. By this way, the MAS developer focuses on the Business Logic of his/her MAS without the burden of building Unit Testing Framework to create Test Cases.

We provide the first implementation that triggers the refactoring process. The developer then uses the reflection of the Eclipse SDK to reflect the changes made in the AUT (to complete its actual behaviour) in the generated mock agent. Our repository consists of a set of XML files that represent the behaviour 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 (the ten agent *social design* patterns).

We evaluate the code coverage by using EMMA a code coverage tool to prove that the generated test files (mock agent, associated resource file, and AUT test case) from our Eclipse Plug-in totally covers the AUT code.

## **ACKNOWLEDGMENTS**

I begin by thanking Allah for everything has been done for me.

I am very grateful and thankful to Prof. Dr. Khaled Nagi for his constructive supervision, his experienced guidance and unfailing advice.

I would like to thank Prof. Dr. Khalil Ahmed and much appreciate Prof. Dr. Mohamed Nazih for his valuable review for this thesis.

Finally, I wish to thank my great parents for the continuous support for me.

## TABLE OF CONTENTS

<b>CURRICULUM VITAE.....</b>	<b>ii</b>
<b>SUPERVISORS .....</b>	<b>iii</b>
<b>ABSTRACT.....</b>	<b>iv</b>
<b>ACKNOWLEDGMENTS .....</b>	<b>vi</b>
<b>TABLE OF CONTENTS.....</b>	<b>vii</b>
<b>ABBREVIATIONS.....</b>	<b>x</b>
<b>LIST OF FIGURES.....</b>	<b>xi</b>
<b>LIST OF ALGORITHMS .....</b>	<b>xii</b>
<b>LIST OF DIFINITIONS.....</b>	<b>xii</b>
<b>LIST OF TABLES.....</b>	<b>xii</b>
<b>CHAPTER 1: INTRODUCTION.....</b>	<b>1</b>
1.1 General .....	1
1.2 Motivation and Scope of the Work.....	1
1.3 Thesis Organization.....	2
<b>CHAPTER 2: BACKGROUND .....</b>	<b>4</b>
2.1 Introduction.....	4
2.2 Testing in Agent Oriented Software Engineering.....	4
2.2.1 Testing in Software Engineering.....	4
2.2.2 Testing in Knowledge Engineering .....	5
2.2.3 Challenges for Testing in Agent Oriented Software Engineering .....	6
2.2.4 Classes of Testing in Agent Oriented Software Engineering .....	7
2.2.4.1 Testing .....	7
2.2.4.2 Run-time monitoring .....	8
2.2.4.3 Static analysis .....	10
2.2.4.4 Model checking.....	11
2.2.4.5 Theorem proving.....	11
2.3 Taxonomy of MAS Testing and Debugging Approaches .....	12
2.3.1 Software Testing and Debugging Definitions.....	12
2.3.2 A Multi-level Approach for Testing and Debugging MAS .....	13
2.3.2.1 Agent Level Testing .....	14
2.3.2.2 Society Level Testing .....	15
2.3.3 A Survey of Testing and Debugging Multi-Agent Systems.....	16
2.3.3.1 Testing .....	16
2.3.3.1.1 Passi .....	16
2.3.3.1.2 Test Agent .....	19
2.3.3.1.3 Madkit .....	20
2.3.3.1.4 XMLaw .....	22

2.3.3.2 Debugging .....	24
2.3.3.2.1 Zeus .....	24
2.3.3.2.2 Prometheus .....	26
2.3.4 Multi-Agent Systems Testing and Debugging Approaches Comparison .....	28
2.4 MAS unit testing approach using Mock Agents.....	29
2.4.1 A unit test approach for MASs .....	29
2.4.1.1 Test-Case Design based on Mock Agents.....	30
2.4.1.2 Test Case Execution .....	30
2.4.2 Applying the unit test approach on top of JADE .....	31
2.4.2.1 JADE Mock Agent .....	31
2.5 Tropos Requirements-Driven Framework .....	31
2.6 Taxonomy of Agent Social Design Patterns .....	33
2.6.1 Social Patterns Categories .....	33
2.6.1.1 Pair patterns category .....	33
2.6.1.2 Mediation patterns category.....	34
2.7 Using Mocks and Code Refactoring in TDD .....	35
2.7.1 Test Driven Development.....	35
2.7.2 Code Refactoring .....	35
2.7.3 Mocks and Code Refactoring.....	36
2.8 Conclusion .....	36
<b>CHAPTER 3: PROPOSED FRAMEWORK FOR DESIGNING TEST CASES TO MAS .....</b>	<b>37</b>
3.1 Introduction.....	37
3.2 Approach .....	37
3.2.1 Agile Software Engineering Test Driven Development Process.....	37
3.2.2 Social Software Design Patterns .....	37
3.3 Framework components.....	37
3.4 Suggested Support Concept .....	39
3.5 System Architecture .....	40
3.5.1 JUnit Framework .....	40
3.5.2 JADE Framework classes used in the approach.....	41
3.5.3 MASUnitTesting Framework .....	42
3.5.4 Proposed MAS Application .....	43
3.6 MockAgentDesigner Eclipse Plug-in .....	46
3.7 Implementation through Eclipse plug-in.....	47
3.7.1 MockAgentDesigner main form .....	47
3.7.2 Working Example: Service Trading.....	49
3.8 Adding new design pattern .....	51
3.9 Conclusion .....	56
<b>CHAPTER 4: DESIGN OF TEST DESIGN PATTERNS .....</b>	<b>57</b>
4.1 Introduction.....	57
4.1.1 Design Pattern Template .....	57
4.1.2 Collaboration diagram.....	57
4.1.3 Structure Diagram.....	57
4.1.4 Behaviour Diagram.....	58
4.1.5 Test Design Patterns.....	58
4.1.6 Pattern Implementation .....	58



4.2	Booking Pattern.....	59
4.2.1	Design Pattern Template .....	59
4.2.2	Collaboration diagram.....	60
4.2.3	Structure Diagram.....	61
4.2.4	Behaviour Diagram.....	62
4.2.5	Test Design Patterns.....	62
4.2.5.1	BookingClientAgent.....	62
4.2.5.2	BookingServiceProviderAgent.....	67
4.2.6	Pattern Implementation .....	71
4.2.6.1	BookingClientAgent.....	71
4.2.6.2	BookingServiceProviderAgent.....	72
4.3	Broker Pattern .....	73
4.3.1	Design Pattern Template .....	73
4.3.2	Collaboration diagram.....	74
4.3.3	Structure Diagram.....	74
4.3.4	Behaviour Diagram.....	75
4.3.5	Test Design Patterns.....	76
4.3.5.1	BrokerClientAgent .....	76
4.3.5.2	BrokerAgent.....	78
4.3.5.3	BrokerServiceProviderAgent .....	80
4.3.6	Pattern Implementation .....	81
4.3.6.1	BrokerClientAgent .....	81
4.3.6.2	BrokerAgent.....	82
4.3.6.3	BrokerServiceProviderAgent .....	83
4.4	Conclusion .....	84
<b>CHAPTER 5: EVALUATION .....</b>		<b>85</b>
5.1	Introduction.....	85
5.2	EMMA Code Coverage Results .....	85
5.3	Conclusion .....	88
<b>CHAPTER 6: CONCLUSIONS AND SUGGESTIONS FOR FUTURE WORK .....</b>		<b>89</b>
6.1	Conclusions.....	89
6.2	Suggestions for Future Work .....	90
<b>BIBLIOGRAPHY.....</b>		<b>92</b>
<b>APPENDIX A</b>		

## **ABBREVIATIONS**

AI:	Artificial Intelligence
MAS:	Multi-Agent System
MASs:	Multi-Agent Systems (as a branch of AI)
AOSE:	Agent Oriented Software Engineering
PDB:	Problem Database
QoS:	Quality of Service
AUT:	Agent Under Test
ACL:	Agent Communication Language
AID:	JADE Agent Identifier
BDI:	Belief-Desire-Intention
DF:	Directory Facilitator
FIPA:	Foundation for Intelligent Physical Agents
UML:	Unified Modeling Language
SKwyRL:	Social arChitectures for Agent Software Systems EngineeRing
TDD:	Test Driven Development

## LIST OF FIGURES

Figure 2.1: Minimal validation in software engineering.....	5
Figure 2.2: Classes of Testing in AI .....	5
Figure 2.3: Interaction test framework .....	8
Figure 2.4: Certification Management Architecture.....	10
Figure 2.5: Kinds of Tests.....	12
Figure 2.6: PASSI - Test Framework main classes .....	17
Figure 2.7: TestSuiteAgent GUI – The Product Agent Under Test.....	18
Figure 2.8: Architecture of the Test Agent .....	19
Figure 2.9: XMLaw Integration Test Architecture.....	23
Figure 2.10: A micro tool view of an agent in Zeus Development Toolkit .....	25
Figure 2.11: Prometheus Debugging System Design .....	27
Figure 2.12: Unit testing Agent A .....	29
Figure 3.1: Framework of MAS unit test approach integrated with MockAgentDesigner Plug-in.....	38
Figure 3.2: UML class diagram for the MAS unit testing approach upon JADE Platform .....	40
Figure 3.3: Partial code of a Mock Agent .....	44
Figure 3.4: Partial code of a BookSellerTestCase .....	45
Figure 3.5: Entity Relationship Diagram of the MockAgentDesigner Framework.....	46
Figure 3.6: The main form of the MockAgentDesigner Eclipse Plug-in .....	48
Figure 3.7: The generated resource file MockBuyerAgent.properties.....	50
Figure 3.8: The test design pattern file MockBuyerAgent.XML .....	50
Figure 3.9: The “Test Design Pattern” form for the MockBuyerAgent.....	51
Figure 3.10: Add new Design Pattern Category.....	51
Figure 3.11: Add new Design Pattern.....	52
Figure 3.12: Add new Agent Role.....	52
Figure 3.13: Add new Interacting Role.....	53
Figure 3.14: Add new Mock Agent Template.....	53
Figure 3.15: Package and Import structure in the imported MockBuyerAgent template .....	54
Figure 3.16: Package and Import structure in the imported SellerAgent file.....	55
Figure 3.17: Package and Import structure in the imported SellerAgentTestCase file.....	55
Figure 4.1: Sample Code for MatchmakerClientAgent class.....	59
Figure 4.2: The Collaboration diagram of the Booking pattern .....	60
Figure 4.3: The Structure diagram of the Booking pattern .....	61

Figure 4.4: The Behaviour diagram of the Booking pattern .....	62
Figure 4.5: The Collaboration diagram of the Broker pattern.....	74
Figure 4.6: The Structure diagram of the Broker pattern.....	75
Figure 4.7: The Behaviour diagram of the Broker pattern.....	76

## LIST OF ALGORITHMS

Algorithm 2.1: Test-Case Design based on Mock Agents .....	30
--	----

## LIST OF DIFINITIONS

Definition 2.1: Test.....	12
Definition 2.2: Black-box testing .....	12
Definition 2.3: White-box testing.....	13
Definition 2.4: Progressive testing .....	13
Definition 2.5: Regressive testing .....	13
Definition 2.6: Performance testing.....	13
Definition 2.7: Unit Test.....	13
Definition 2.8: Integration Test .....	13
Definition 2.9: Refactoring .....	35

## LIST OF TABLES

Table 2.1: Multi-Agent Systems Testing and Debugging Approaches Comparison .....	28
Table 3.1: Unit Test Case Template .....	44
Table 3.2: PatternCategories table sample data.....	46
Table 3.3: Patterns table sample data.....	47
Table 3.4: AgentRoles table sample data.....	47
Table 3.5: InteractingRoles table sample data.....	47
Table 4.1: Design Pattern Template .....	57
Table 4.2: Booking Design Pattern Template .....	60
Table 4.3: Test Design Pattern of the ClientAgent.....	64
Table 4.4: Test Design Pattern of the ServiceProviderAgent .....	69
Table 4.5: Broker Design Pattern Template.....	74

Table 4.6: Test Design Pattern of the BrokerClientAgent ..... 77

Table 4.7: Test Design Pattern of the BrokerAgent..... 79

Table 4.8: Test Design Pattern of the BrokerServiceProviderAgent .....81

Table 5.1: EMMA Code Coverage Results On Ten Social Design Patterns.....87

Table 5.2: EMMA Code Coverage Results Statistics .....87

## CHAPTER 1: INTRODUCTION

### 1.1 General

A Multi-Agent System (MAS) is a computational environment in which individual software agents interact with each other, in a cooperative or competitive manner, and sometimes autonomously pursuing their individual goals, accessing resources and services of the environment, and occasionally producing results for the entities that initiated those software agents (1).

The agents interact in a concurrent, asynchronous and decentralized manner (2), hence MAS turn out to be complex systems. They are also non-deterministic, since it is not possible to determine a priori all interactions of an agent during its execution. Consequently, they are difficult to debug and test.

### 1.2 Motivation and Scope of the Work

The motivation behind this thesis is to make use from one of the very important Software Engineering Practices that is the *Software Testing*, in one of the very promising software models that is the *Agent Based Software Engineering Model* (3). The used MAS unit testing approach is based on the Agent definition detailed in (4) and presented below:

An agent is an autonomous, adaptive and interactive element that has a mental state. The mental state of an agent is comprised by: beliefs, goals, plans and actions. Every agent of the MAS plays at least one role in an organization. One of the attributes of a role is a number of protocols, which define the way that it can interact with other roles.

Testing of software systems is essential in the software development cycle. In multiagent systems, the non-determinism problem is increased by the known problems of testing concurrent or object-oriented systems. There exist five approaches for testing multiagent systems: testing, runtime monitoring, static analysis, model checking, and theorem proving (5). In our thesis, we will particularly deal with the testing approach.

Software Testing is an activity in which a system or component is executed under specified conditions, the results are observed or recorded and compared against specifications or expected results, and an evaluation is made of some aspect of the system or component. A test is a set of one or more test cases. The main aim of a test is to find faults (6).

The thesis presents a tool for designing test cases based upon unit testing approach for MASs. The main purpose of this approach is to help MASs developers in testing each agent individually. It relies on the use of Mock Agents to guide the design and implementation of agent unit test cases.

We design test patterns for the ten most famous MAS social design patterns found in literature of Agent Design Patterns. For the sake of generality and extensibility, we implement an Eclipse plug-in that gives the ability to the MAS unit test developer to generate the Mock Agent that interacts with his/her AUT through a set of design patterns.

The major objectives of this thesis are:

- Presents a survey for the MAS testing and debugging approaches.
- Make use of software engineering practices (in the form of software testing) that facilitates the software development life cycle.
- Presents a survey for the Agent Social Design Patterns found in the literature.
- Gives the ability to the MAS unit test developer to generate the Mock Agent that interacts with his/her AUT through a set of design patterns.
- Allow the MAS developer to focus on the Business Logic of his/her MAS without the burden of building Unit Testing Framework to create Test Cases.

### **1.3 Thesis Organization**

This thesis is organized in six chapters as follows:

- Chapter 1 begins with a general introduction to the MAS, followed by explaining the motivation behind the research done in this thesis. Finally, the major contributions in the thesis are outlined.

- Chapter 2 gives a complete background for the main concepts targeted in this thesis. The main concepts of Testing in Software and Knowledge Engineering are depicted. Then, a survey of the existing MAS testing and debugging approaches (1) is given. Also, a survey for the Agent Social Design Patterns (7) found in the literature is presented.
- Chapter 3 depicts our contribution by giving detailed design for our framework that is used in designing test cases based on Mock Agents. In this chapter, we present the adopted Agile software TDD process, then the framework components and their integration. This chapter presents as well the detailed system architecture using a UML class diagram, and the Entity Relationship Diagram (ERD) of our MockAgentDesigner Eclipse Plug-in. Finally, the test cases implementation through our system and adding new test design pattern templates are also provided.
- Chapter 4 presents the design of the test design patterns. A set of the most common Agent Design Patterns are presented with their test design patterns for the different roles played by every agent in each design pattern. The test cases implementation of each design pattern is also provided.
- Chapter 5 presents the evaluation of the experimental work done in this thesis. In this chapter, we analyze the quantitative results by using the EMMA code coverage tool.
- In Chapter 6, some concluding remarks and suggestions for future work are presented.



## CHAPTER 2: BACKGROUND

### 2.1 Introduction

This chapter gives a complete background for the main concepts targeted in this thesis. The main concepts of Testing in Software and Knowledge Engineering are depicted. Then, a survey of the existing MAS testing and debugging approaches (1) is given. Also, a survey for the Agent Social Design Patterns (7) found in the literature is presented.

### 2.2 Testing in Agent Oriented Software Engineering

The view of testing has evolved over the last years and testing is no longer seen as an activity which starts only after the coding phase is completed. Software testing is now seen as a whole process that permeates the development and maintenance activities (8). Agent-Oriented Software Engineering (AOSE) methodologies, as they have been proposed so far, mainly propose disciplined approaches to analyze, design and implement MASs. However, little attention has been paid to how multi-agent systems can be tested (9). In this section, we introduce testing from software and knowledge engineering perspective. After identification of challenges for agent and multi-agent testing, different classes for testing in this field are introduced. There exist five classes for testing multi-agent systems: testing, runtime monitoring, static analysis, model checking, and theorem proving (5).

#### 2.2.1 Testing in Software Engineering

The traditional approaches to software engineering, e.g., the waterfall model, consider testing after implementation (10). *Agile software engineering* approaches, e.g., extreme programming, address testing continuously within the implementation process (11). Furthermore, developers formulate test-cases before or during implementation, which may be executed automatically on demand. This procedure is also known as “*Test-driven Development*” (12). During development, usually three main models are generated: requirements model, design model, and implementation model. These models have to be validated in order to ensure high quality software (13). In Figure 2.1, the process of the minimal validation within software engineering is illustrated.

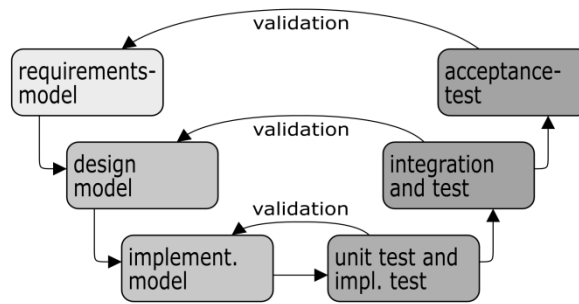


Figure 2.1: Minimal validation in software engineering

The testing of the models is performed in opposite direction of their building. In the first step, the implementation is tested for example by unit test during the coding. If the implementation model appears to be correct, modules are integrated and their composite behavior is tested. After completion of these tests the software is supposed to satisfy the specification of the design model. However, there may be inconsistencies between the design model and the requirements model. These are identified in the acceptance tests, which are the final stage of testing.

### 2.2.2 Testing in Knowledge Engineering

There are multiple approaches to test systems behavior differing in need of expertise and strength of proof: testing, run-time monitoring, static analysis, model checking, and theorem proving, Figure 2.2 (14 p. 6).

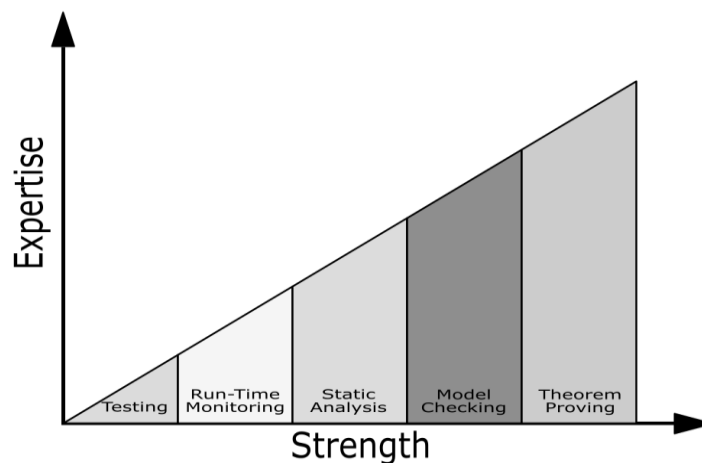


Figure 2.2: Classes of Testing in AI

**Testing** in software engineering is essential for ensuring quality of systems (13). The main purpose is to identify errors within the algorithms, modules, or software. However, there is no

possibility to recognize correctness (15). Tests are case-based, i.e., the test program creates defined sequences of input patterns and evaluates whether the output meets the pre-defined requirements (5).

**Run-time monitoring** is a procedure to analyze the behavior of a system in run-time. The benefits of run-time monitoring are the efficient usage of system resources, restoring the system states, and scalability (5).

**Static analysis** concentrates on the structures within the source code without execution of the system. Objects of the analysis are control flow and data flow. One of the benefits of the approach is the possibility to find segments of code, which a) are guaranteed free from errors, b) are sure to cause errors, c) may cause errors, or d) are unreachable (5).

**Model checking** is to verify a property of a system by exploring all of the systems reachable states. The starting point in model checking is the specification of two models: system model and property model (14). In the system model, relevant features of the system are captured as a formal abstraction of the implementation with respect to the properties to be checked while the property model is a formal specification of the requirements the system is supposed to meet (5).

**Theorem proving** is used for formal verification of software systems. Here, a mathematical model of a computer program is generated to determine whether it satisfies desired properties. Formal deduction by human-guided theorem proving (i.e., interactive proof checking) can, in principle, verify any correct design, but doing so may require unreasonable amounts of effort, time, or skill (16). The main difference to other approaches is that theorem proving uses symbolic representations of values.

### **2.2.3 Challenges for Testing in Agent Oriented Software Engineering**

In contrast to conventional software systems, intelligent agents are acting autonomously, such that requirements have to be specified primarily as goals or context for agent behavior. Furthermore, intelligent agents should act with respect to the current situation in the

environment as well as their internal state. The main challenge for intelligent agent testing arises from the agent *flexibility* (5).

However, AOSE is also a software engineering process, which is closely related to object orientation. Conventional testing requires testing in any possible state which is for complex object not feasible (17). Another dimension of complexity evolves from the key feature of multi-agent systems, flexibility, realized by emergent effects, emergent properties or emergent organization (5). Furthermore, in multi-agent systems there are multiple agents interacting with each other and the environment. Obviously, this causes concurrency problems. Testing of concurrent software is a huge challenge in conventional software engineering (18).

#### **2.2.4 Classes of Testing in Agent Oriented Software Engineering**

The context of intelligent agents and multi-agent system as discussed in the last section is challenging for testing. Current AOSE methodologies do not integrate testing in an adequate way. The formal approaches to testing suffer from complexity with respect to knowledge and computation possibly causing overextended efforts. Thus, it is proposed to apply to AOSE any of the classes of testing: testing, run-time monitoring, static analysis, model checking, and theorem proving. In the following section, possible approaches to each of these classes are discussed (5).

##### **2.2.4.1 Testing**

In AOSE, there are no agent-specific approaches for testing. However, in software engineering there are multiple testing methods available which may be applied to the agent engineering process. One of the most prominent testing approaches in Software Engineering is the *unit test*, which finds application during the implementation of functions, methods, interfaces, or classes. The key benefit of the unit test approach is that tests may be applied repeatedly and automatically, i.e., every specified test is automatically performed after changes in the implementation of the unit, even if the changes are not directly connected to the test object (5).

Testing of interaction is a challenge for AOSE as interaction in multi-agent systems is assumed to emerge within runtime. In order to test the interaction behaviour, it is necessary to establish a controlled test environment for the agent under test. In specialization of unit tests, an agent is considered as a unit. However, the test cases are more complex since they need to include the specifics of the agent environment and consider the variance of the expected results with respect to the sequence of the interaction. In consequence, testing requires the definition of allowed message sequences. Grammar can be used as syntactical specification of valid sequences. Messages in an interaction are considered as words of a language and sequences as sentences. In agent interaction the content of a message is also relevant to the behaviour of an agent.

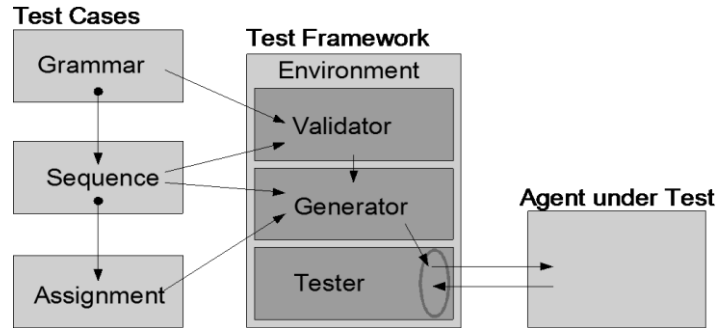


Figure 2.3: Interaction test framework

An interaction test framework is shown in Figure 2.3. The controlled environment consists mainly of the tester which utilizes a test-case generator for the generation and selection of appropriate test cases. This is done on the basis of a concrete assignment of a message which is validated by the validator-component using the specified grammar and allowed sequences. This framework enables the test of agent interaction behaviour in a controlled way without postponement of the test to the runtime of a multi-agent system (5).

#### 2.2.4.2 Run-time monitoring

Analogous to testing, in runtime monitoring the basic idea is to establish a controlled environment and a control structure for running tests under specific configurable conditions. In contrast to testing, in runtime monitoring not specific units are considered, but the system behaviour in runtime. In AOSE the challenge arises from analyzing a distributed and partially non-deterministic system. There are two main testing objectives for runtime monitoring: an agent or a multi-agent system. For runtime analysis in AOSE, it is proposed to consider grey-

and black-box approaches only, since either in heterogeneous systems, internal structures or details of coding are hidden or the non-deterministic behaviour of agent inferences is not suited for simple white-box approaches. There exist two pragmatic approaches to runtime monitoring in AOSE: simulation covering both agent and multi-agent runtime monitoring, and certification management only for agent runtime monitoring (5).

In simulation-based runtime monitoring, a controlled environment as well as a control structure is established. Additionally, a specific agent is introduced in the Multi-Agent System, which is collecting and documenting information about the dynamic system behaviour. A stochastic simulator is introduced, which triggers the control unit and establishes varying conditions in a high number of sequential runs. By this, the test procedure becomes a stochastic process and statistical analysis of the system behaviour is possible, which is considered as a grey-box test. The benefit of a simulation approach for runtime monitoring is that it is easy to extend the system for benchmarking purposes (5).

The approach discussed above finds practical application when architecture and implementation are known during test specification. However, testing agent behaviour during runtime is also necessary, especially in open multi-agent systems, where agents may join a society and offer services at any given time. Certification management addresses these issues by performing runtime monitoring automatically.

In Figure 2.4, the architecture for the agent system realizing an integrated catalogue, capability, and certification management is visualized. The agent for catalogue management is implementing the key-role and is responsible for managing service registration inquiries, invocation of the certification process, and providing an exploration interface to query on registered services. It utilizes the capability management for the match-making between consumer problems and service problem solving capabilities (5).

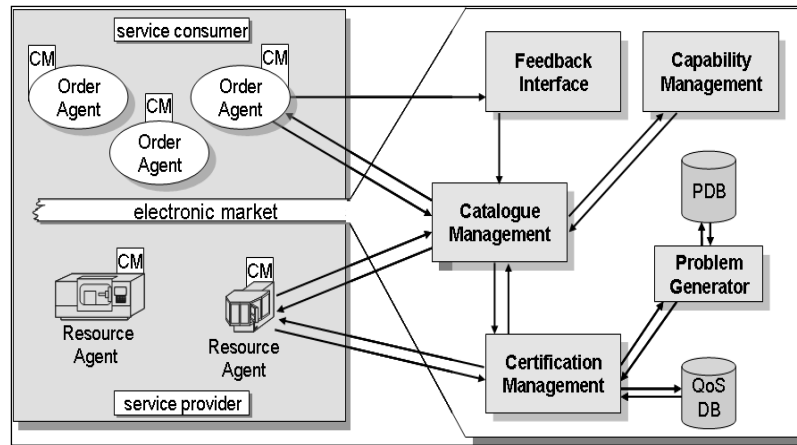


Figure 2.4: Certification Management Architecture

The agent for certification management certifies Quality of Service (QoS) of service providers with regard to provided capabilities using the capability management agent for inference on capabilities. The certification of problem solving capabilities is performed by putting the service to the test with a set of problems and a standard solution. Problems are domain specific, and are either taken from the Problem Database (PDB) or created dynamically by a problem generator agent. These agents need to be adapted for each domain in order to generate relevant problems for the certification process. The results from service are evaluated according to a standard solution. On this basis, a measure of completeness, which is the QoS for the provided capability, may be derived. The QoS are stored in a database (QoS DB), which is updated in regular intervals with re-certification of the services (5).

#### 2.2.4.3 Static analysis

The static analysis aims to identify structural errors in source code without running the system. In AOSE, the challenge is to cope with distributed and concurrent processes. *“Especially if it comes to concurrent and distributed processes, e.g., the multi-agent systems, the advantages of Petri nets are obvious”* (19 pp. 47-48). The advantage of Petri nets is that it provides not only a formal semantics but also an operational semantics. Consequently, Petri nets have been established as a key methodology for analyzing parallel or concurrent systems (18). For the analysis of concurrent systems, the identification of control structures is required, leading to an abstract Petri net model of a system. Here, the autonomy of individual agents is preserved. Petri nets are especially suited for the analysis of dynamic aspects of interaction in multi-agent systems in order to avoid dead-locks (5).

#### **2.2.4.4 Model checking**

Model checking is a model-based approach that bases on a model of a system and verifies that a temporal logic formula holds for the model. Whilst axiomatic approaches are based on syntactic proof, model checking approaches lean onto the specification language semantics. Current work concerning the application of model checking approaches in multi-agent systems can be divided into two major streams: in the first category, standard predicates are used to interpret the various intentional notions of multi-agent systems. They are used together with standard model checking techniques based on temporal logics. In the other stream methods are developed that extend common model checking techniques by adding other operators (5).

#### **2.2.4.5 Theorem proving**

Theorem proving is a logic-based proof for checking that a set of formulas satisfies a goal via inferences. Therefore both the system and the description of the system properties have to be available in a formal way (5).

Agent-based systems in reference to conventional programs exhibit intelligent behaviour because of their autonomous abilities enabled by their underlying knowledge-bases. As knowledge-bases consist of logic expressions, deriving faults from the verification and validation via theorem proving is a common task in agent-oriented development. The single agent knowledge-bases can be analyzed by deducing Petri-nets from them. Thus they present a theoretical framework for the verification that can be applied to verify any agent-architecture that can be derived to their suggested one, in polynomial time (5).

There exists one focal problem in the development of agent-based systems concerning the usage of theorem proving that has not been solved yet; the problem lies in the connection between the knowledge-bases and the implementation language of the later system. They have to be integrated in a non-logic language which impedes the application of theorem proving methods. Another problem can be found in the formalization of the system properties due to the fact that the question of which properties an agent-based system provides in detail is not answered either.



## 2.3 Taxonomy of MAS Testing and Debugging Approaches

This section aims to survey techniques and tools for testing and debugging multi-agent systems in the Software Engineering approach. There are some works that propose strategies and techniques for testing or evaluating multi-agent systems through theorem proving and model checking. However, we will see why we are not focusing on those strategies. Thus, this section surveys all tools found in literature for the Software Engineering approach (1).

### 2.3.1 Software Testing and Debugging Definitions

The aim of this section is to give some basic definitions and clarify some techniques and strategies that will appear in the testing and debugging approaches of the Survey (1).

#### Definition 2.1: Test

*A test is a set of one or more test cases. The main aim of a test is to find faults. (1)*

There are two kinds of tests: static verification and dynamic validation. The former is based on code inspection or “walk through”, symbolic execution, and symbolic verification. The later generates test data and execute the program. Figure 2.5 shows where static verification and dynamic validation tests occur during the software life cycle (20).

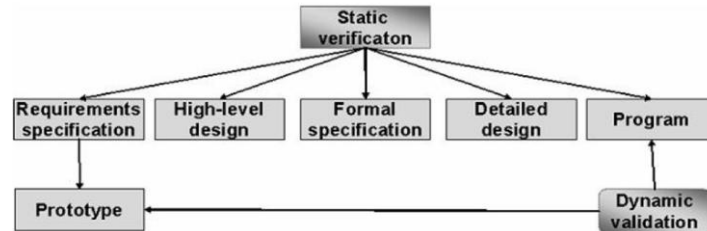


Figure 2.5: Kinds of Tests

There are several strategies for testing software and the goal of this survey is not to explain all of them. However, here we will mention the main strategies found in literature ( (20), (21)) for testing software which are related to some of the works presented in the survey.

#### Definition 2.2: Black-box testing

*Also known as functional testing or specification-based testing, that is testing without reference to the internal structure of the component or system (1).*

**Definition 2.3: White-box testing**

*Testing based on an analysis of the internal structure of the component or system. Test cases are derived from the code e.g. testing paths. Note: A test case is a set of test inputs, execution conditions, and expected results developed for a particular objective (1).*

**Definition 2.4: Progressive testing**

*Progressive testing is based on testing new code to determine whether it contains faults (1).*

**Definition 2.5: Regressive testing**

*Regressive testing is the process of testing a program to determine whether a change has introduced faults (regressions) in the unchanged code. It is based on re-execution of some/all of the tests developed for a specific testing activity (1).*

**Definition 2.6: Performance testing**

*Performance testing is to verify that all worst case performance targets have been met, and that any best case performance targets have been met (1).*

There are several types of tests. The most frequently performed are the unit test and integration test.

**Definition 2.7: Unit Test**

*A unit test performs the tests required to provide the desired coverage for a given unit, typically a method, function or class. A unit test is white-box testing oriented and may be performed in parallel with regard to other units. (1)*

**Definition 2.8: Integration Test**

*An integration test provides testing across units or subsystems. The test cases are used to provide the desired coverage for the system as a whole. It tests subsystem connectivity. (1)*

**2.3.2 A Multi-level Approach for Testing and Debugging MAS**

It is quite hard to verify that agents or multi-agent systems satisfy user requirements, behave correctly and are not malicious. One possible way to help solving this problem is to develop

and use *design patterns*, as well as algorithms and tools for evaluating multi-agent systems (2).

Testing a single agent is different from testing a community of agents. When testing a single agent a developer is more interested in the functionality of the one agent and whether the agent operates for a set of messages, environmental inputs and error conditions. On the other hand, when testing a community of agents, the tester is interested in whether the agents operate together, are coordinated, and if message passing between the agents is correct (1).

The agent society test is a kind of integration test and the integration strategy depends on the agent system architecture where agents dependencies are usually in terms of communications. In the case of a multi-layer (open or closed) architecture, it is possible to implement common strategies like the top-down or bottom-up composition. With other MAS configurations the integration problem becomes much more difficult to solve because of the proactive nature of agents, in some cases it is not possible to identify the subset of agents that could be tested in a stand-alone basis. Such situations should be faced with a study of the agent responsibilities (in terms of functionalities provided) and ontology (an agent is not expected to interact with something that is out of its knowledge) (1).

#### **2.3.2.1 Agent Level Testing**

Testing is usually done incrementally during software development. Developers also usually want to incrementally test their agents as they progress, testing functionality as it is added. Much of this testing would require another agent to trigger an event inside the agent to be tested, such as message from another agent, or an event from the environment (1). When developing a single agent (for inclusion into a community) developers want to make sure that it responds correctly to given inputs from other agents. Moreover, a sizeable set of agents, none of which have been tested, may be needed to test a single agent (1).

In (22) some of the errors types are presented, which developers run into while developing an agent, which include:

- Incorrectly addressing a message to another agent;

- Putting an incorrect request in a message so the receiving agent does not recognize the message;
- Incorrectly parsing incoming messages;
- Checking for the wrong performative, which is the type of the communicative act, in an incoming message;
- Not developing code to accept all the messages it is suppose to accept, etc.

An agent can be tested as a black-box which focuses on the behavior of the agents. Testing its behavior corresponds to begin an interaction and evaluating its result. And an agent can be tested as a white-box, which corresponds to test the agent internal behaviors, which is the same of how behaviors are related and their flow of control, where each single behavior is seen as a black-box. The result is therefore a kind of black-box testing of a subsystem (the agent composed of the agent base class and several behavior classes) (23).

#### **2.3.2.2 Society Level Testing**

Testing communities of agents involves two issues. The first one is how we can ensure that the agents in the community work together as designed previously. And the second one is how we can ensure that the resultant work is the one expected (1). During a society test, the validation of the overall results of the different agents is carried on and the successful integration of the different agents is verified. This involves checking that each agent in the community/society receives the correct messages from the correct agent, provides the correct responses, and interacts with environment correctly as a whole (22), which is related to the first issue. Moreover, it can also involve checking that the goal of this community or organization where the agents are interacting is being achieved, which is related to the second issue.

Some types of errors that can be observed during the developing of a community are miscommunicating the performative (the type of the communicative act), content on agent messages, and designing deadlocks into the messages exchanges. Another issue that has to be considered is the scalability. The larger the agent communities become, the harder it is to test them for proper functionality (22).

### **2.3.3 A Survey of Testing and Debugging Multi-Agent Systems**

In this section, we provide an overview of several works that, in one way or another, include some notion of testing and/or debugging multi-agent systems. First, we describe the approaches for testing techniques, which do not address the debugging techniques. Then, we describe the debugging techniques and its tools. It is important to notice that some debugging tools found in literature for multi-agent systems also contain testing techniques (Zeus (24) (25), Prometheus (26) (27) (28)). They are separated in two subsections because the testing techniques may be related only to test cases, and can not be considered a debugging tool. We also mention an evaluation for them considering its approaches, its relative strengths and weakness, and the open issues of each one (1).

#### **2.3.3.1 Testing**

##### **2.3.3.1.1 Passi**

The proposal of this work (23) is not to provide an exhaustive testing tool but to propose a new approach based on a simple testing framework which lets developers build a test suite effortlessly in a cheap and incremental way. As a framework, it provides a unifying application model and a partial implementation of it, trying to support the developer in creating and executing tests in a uniform and automatic way. They aim to reduce time and cost when developing MAS, guarantee quality assurance, and provide automatic activities which should assure that the program performs satisfactorily. Their framework is built on top of JADE. There is a “test-agent” which performs the set of tests related to all the capabilities of a given agent. Tests on specific tasks, on the other hand, will be referred to as “task-test” (1).

In order to reflect the two-level model, the following classes are provided (1):

- Test class, representing the test of a specific task of an agent.
- TestGroup class, representing the group of all the agent tests. It is basically a collection of Test objects. The list of task-tests to be included in a TestGroup is described in an XML file.

In general, all test methods in a TestGroup share the same fixture, which consists of objects and anything else needed to perform the test. A Test or a TestGroup is executed by a tester agent i.e. an agent that extends the TesterAgent class. Each tester agent has a behavior, an extension of the TestGroupExecutor class, which is in charge of getting the group of tests to be executed and for each test adds the corresponding behavior to the tester agent scheduler. The list of all the agent-tests that can be tested and the list of task-tests to be performed for each of them are described by means of XML files (1).

There is a single main XML file that contains the list of all the agent-tests of the application and one XML file for each agent-test that contains the list of task-tests to be executed. Developing an agent-test means therefore developing a new tester agent in charge of the group of task-tests described in the associated XML file (1).

Finally, the utility class Logger provides methods to create logs. By extending this class it is possible to create sophisticated loggers in order to provide reported information in more suitable formats. To date, reported information can be displayed in a graphical user interface (where very essential information is shown), written to a text file, printed to the standard error or organized into web pages. In Figure 2.6, a class diagram of the framework main classes is represented (1).

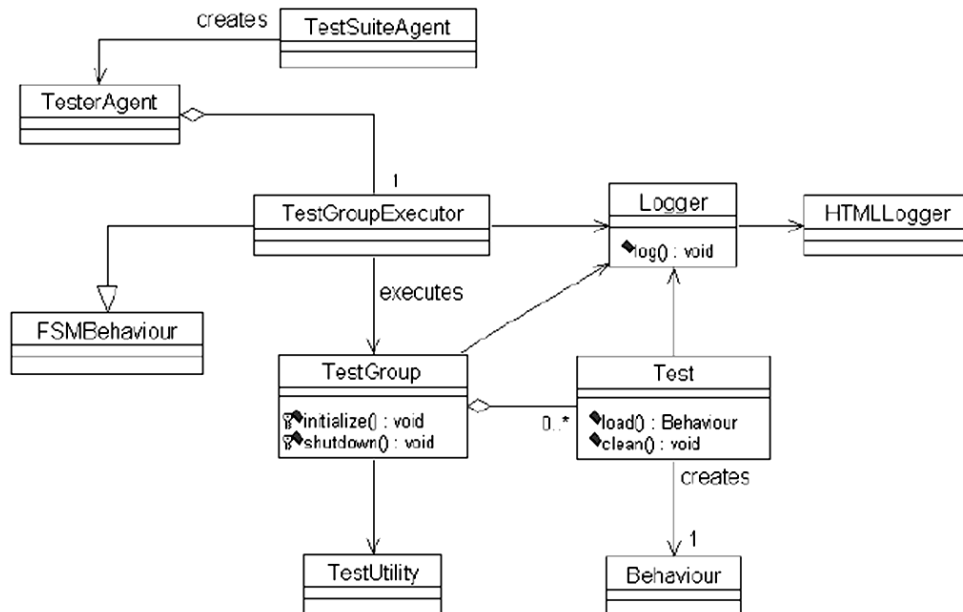


Figure 2.6: PASSI - Test Framework main classes

A single test and group of tests can be executed by simply launching the corresponding tester agent. A more convenient way of performing them is by means of the TestSuiteAgent, an agent that provides a valuable graphical interface to run tests. When a test or a group of tests are launched the TestSuiteAgent creates the proper tester agent and delegates to it the execution of the tests. During the testing activity the tester agent will send FIPA ACL messages to the TestSuiteAgent, informing it about the test outcomes and giving eventually detailed information concerning the causes of failure (1).

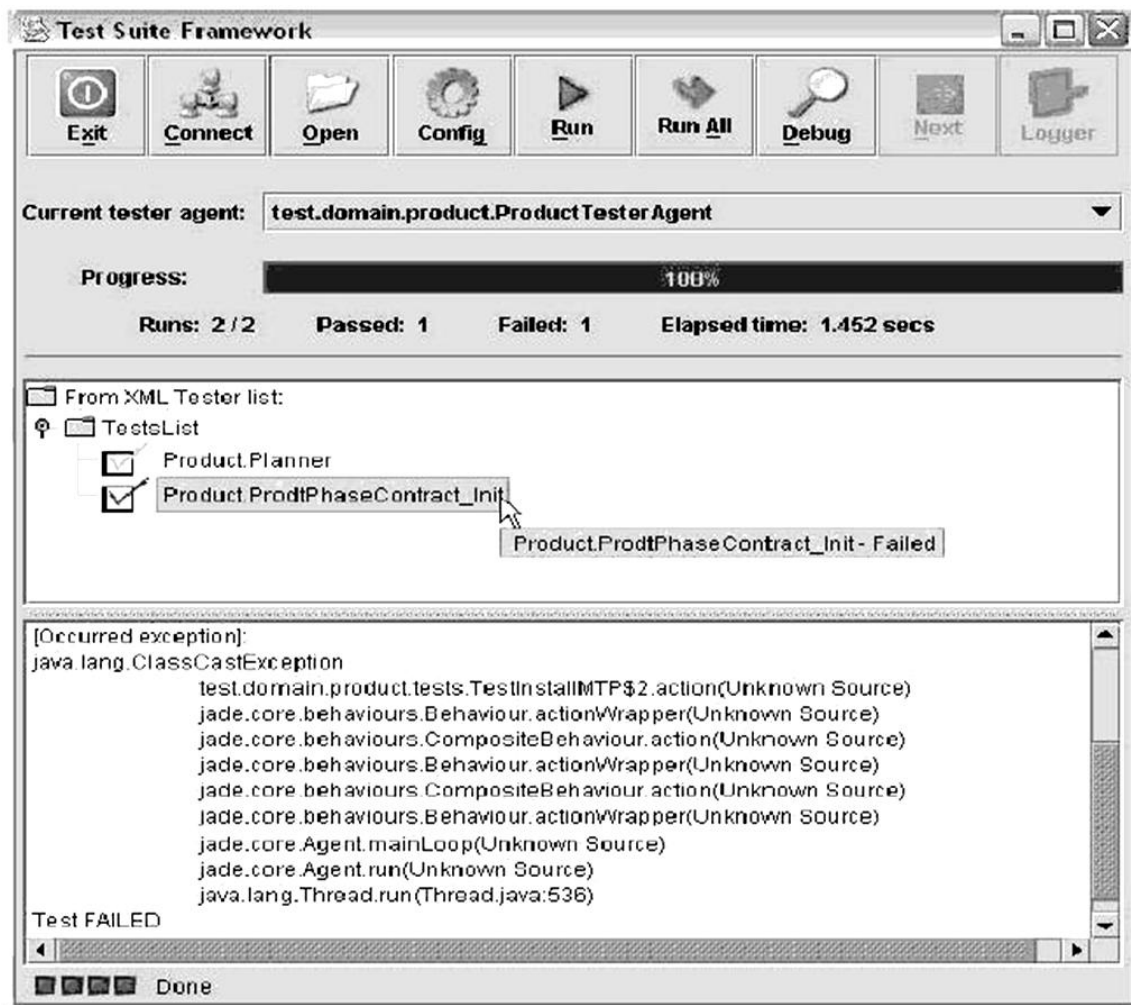


Figure 2.7: TestSuiteAgent GUI – The Product Agent Under Test

The TestSuiteAgent, as stated before, provides a graphical interface to run tests, by means of which it is possible to (1):

- View information related to the agent-tests and all the task-tests they include.
- Select and load the tests to be executed.

- Execute all agent-tests of the list in sequence and produce a final report indicating, for each agent-test, the number of task-tests which have passed and failed and the corresponding causes of failure.

#### 2.3.3.1.2 Test Agent

This work (22) provides a Test Agent which is able to be inserted or plugged into a community of agents to exercise each of the agents in the community as well as the community as a whole, and to also monitor the agents after they are deployed to insure proper functionality. Figure 2.8 shows the architecture and data flow for the test agent in its completed form. The test agent receives message specification and stores it in the message specification file (potentially as XML, which would allow it to be easily reused). From this information, test scripts can be generated to test each of the defined autonomous systems (1).

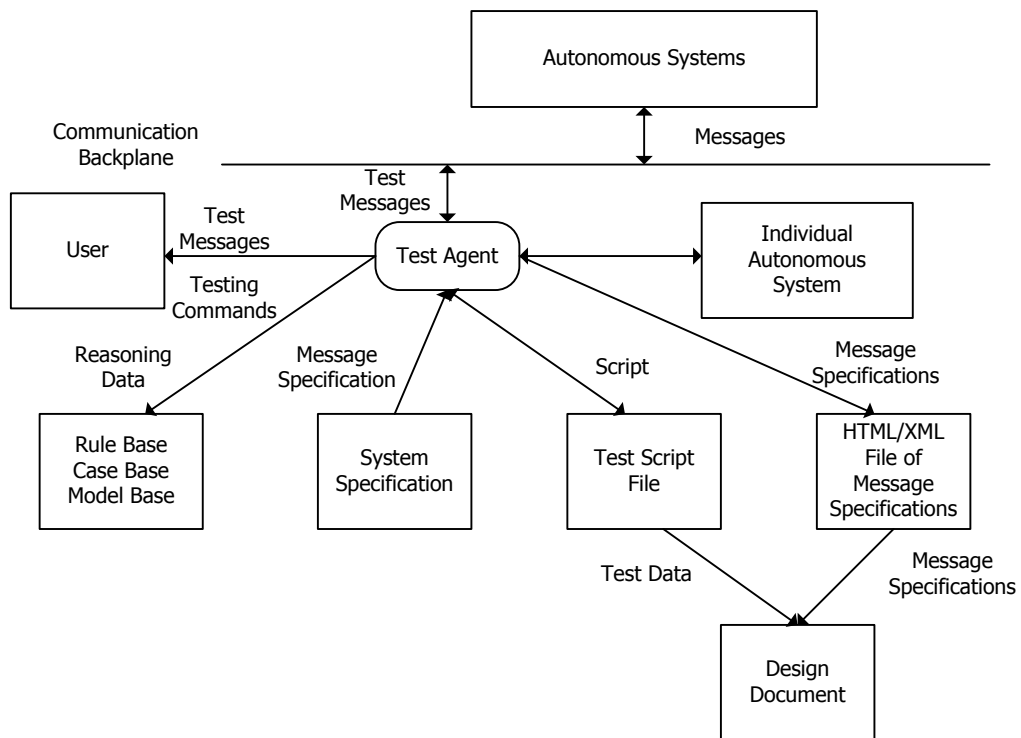


Figure 2.8: Architecture of the Test Agent

The test agent tests other agents and communities by (1):

- Testing a single agent ability to send or receive a specific message;
- Testing a single agent ability to handle valid as well as invalid messages;
- Testing a community ability to handle all defined messages and representative number of invalid messages;



- Maintaining the official message specifications for the agent community;
- Maintaining the message specifications in the design document;
- Collecting metrics on network usage, inter-agent communications and other needed metrics for scalability issues;
- Monitoring an agent system for potential errors and performance problems.

The test agent supports both regression and progression testing of agent message handling capabilities. The regression testing is used to insure that agents perform to their stated specifications and that modifications to agents do not affect existing message handling capabilities. The progression testing supports agent developers as they add new messaging capabilities to their agents. Since a complete specification of the agent communications is stored in the specification file, it is possible to output the contents of the file into an HTML, XML or other format that could then be used in a design document. The design document is automatically updated when the specification file changes (1).

Testing individual agents is done by sending specific messages from the agent to the target agents and examining the messages returned (if any). The set of valid messages for each agent will be generated from the message specification file. From this definition, the test agent will be able to generate a set of test messages and interpret the responses as valid or invalid. Rules to apply to the test messages will be stored in a rule base and will be generated from user inputs and the agent specification (1).

Testing a community of agents is also based on the message specification file. From this file, the test agent can generate a test script for the entire community of agents. This test script again, can also be run interactively or in a batch mode. The test agent is then successively substituted for each agent in the community (again using automatic testing) with the results being stored in a file or emailed to a developer (1).

#### **2.3.3.1.3 Madkit**

The Madkit approach (2) focuses on a specific kind of testing called the Record/Replay mechanism (29) used in regression testing. The Record/Replay mechanism is a test performed

during the execution of the system either in simulation or in production. It is realized through system inspection. The record phase records actions in the system (memory, environment, data update, messages, etc.). When an error occurs in the system, designers have a system that they can play and replay until they found the error and fixed it. They also compare model checking and testing, and say that contrarily to the former, testing checks that agents behave properly rather than agents are proved correct. In their approach, the Record/Replay mechanism is coupled to testing via *post-mortem* analysis. It uses the events and data stored during the record phase and checks properties without re-executing the system (1).

The information stored depends on what designers want to replay. Actually, there exist two kinds of execution replay: content-based and ordering-based execution replay. The content-based execution replay considers storing all the instructions from systems (instructions from programs as well as memory). The ordering-based execution replay only stores the non-deterministic events in order to make sure they will be replayed in the same order. As stated, the ordering-based execution replay has high level of abstraction. In case of ordering-based execution replay, the record phase uses vector clocks to store the event ordering. Hybrid approaches mixing content-based and ordering-based execution replays offer the better of the two approaches. This is the one chosen for this approach (1).

There are three monitor agents responsible for the record phase. Two of them were extended from the MadKit platform (30) and the other was created. The agents are (1):

- The group message tracer agent: is responsible to trace messages sent within the group. Each time this agent receives a message, it records it in a file. During the replay phase, the group message tracer agent reorders the messages based on the trace file.
- The organization tracer agent: is responsible to record group and role actions: when agents enter or leave a group, or when they add or remove a role. It records group and role actions in a trace file. This agent is not used in the replay phase.
- The environment tracer agent: is responsible to record the environment initial configuration and modification. This agent checks the environment at the end of each time interval and records what the modifications (new agent position, resource production or resource consumption) are. The modifications are stored in a trace file.

In the context of ordering-based execution replay, it is necessary that messages arrived in the same order to agents. Agents are constrained in the way they have to send the message to the group message tracer and not to the recipient in order to ensure message ordering. The group message tracer is responsible to reorder messages based on the trace file and deliver them to agents in this order. They also raised the idea of using the environment trace as checkpoints: the user specifies which environment situation to consider as initial situation, and then the replay is performed from this environment situation. The agents are necessarily moved to a situation compatible with this environment situation, that is, agents have the state they have just at the moment of this environment situation. Agents, in the replay phase, send messages to the group message tracer and not directly to recipients in order to repeat the exact order (1).

The post-mortem analysis uses the data recorded during the record phase. The data from the trace files are extracted and its properties verified. For instance, if the designer wants to check that after receiving a message, the recipient believes the content of the message, the designer extracts the message from the group message tracer file and checks if there is a belief about the content of the message dated after the message receipt. If yes, the recipient believes the content (1).

#### **2.3.3.1.4 XMLaw**

In this work (31), an approach is proposed for integration tests in open multi-agent systems. This approach supports the creation of test cases based on the information provided by the definition of system rules. They propose to use XMLaw, which is a language for the specification of agents interactions regulation in open multi-agent systems.

In open multi-agent systems, agents must obey social conventions in order to maintain predictable integration. Usually, these social conventions are hard coded, leading to unsuitable systems. A solution to hard coded conventions is separate the system social convention into a separate module insuring agents compliance.

This technique is called law enforcement. A law enforcement approach separates the system rules from the raw source code, making them explicit so that the development and maintainability may become more efficient (1).

XMLaw is a law enforcement language and environment supplying a structural and dynamic model. The former describes how all the law elements are related while the last shows how an event-driven architecture is implemented. XMLaw is based on a mediator agent that has an enforcement mechanism for applying the laws. The system laws are described in an XML file and their hypothesis is to reuse this language to establish system behavior (1).

Regarding the source code, some system assertions can be established, assertions are expressed as observation points. Observation points are an extensible feature that permits us to configure the test infrastructure to specific needs (1).

As stated before, the XMLaw environment provides a mediator agent, which enforces all messages exchanged by agents. All agents must use the same mediator, where non-compliant messages are always blocked. Also the environment is built with an event-driven architecture, meaning that messages can fire events and observers can subscribe to them (1).

Concerning testing and the XMLaw environment, they propose an integration test architecture providing means of executing test cases, block-unblock operation mode, allowing the execution of entire test cases, coverage evaluation of test cases and the generation of testing reports. The system takes advantage of the event-driven model used in XMLaw; all events generated in the system can be observed and be reported to a log database, Figure 2.9 illustrates the architecture (1).

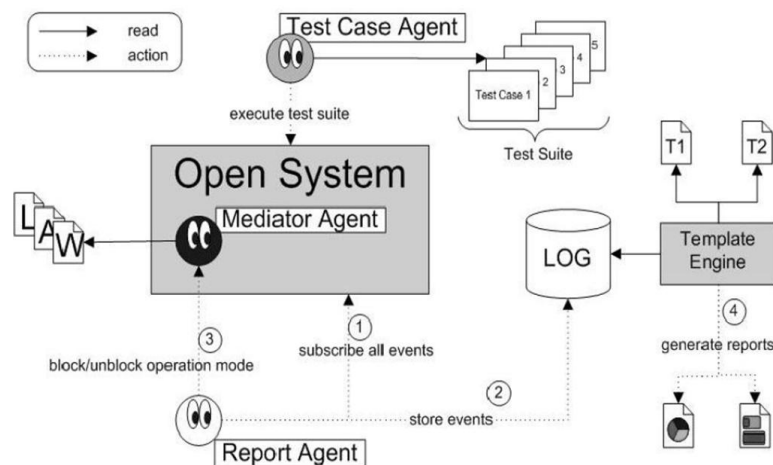


Figure 2.9: XMLaw Integration Test Architecture

Since the environment uses a mediator agent which subscribes itself to all events in order to observe all messages, they provided an agent, called Report Agent, which also subscribes itself to all events (step 1), doing so, all generated events will be stored into a log database (step 2).

They also proposed to implement a mechanism allowing noncompliant messages cross the system (step 3), which allows the execution of test suits. However, none of those mechanisms have been implemented yet.

At the end of the execution of several test cases, the database contains a log of all events fired during the whole execution and also a trace of all messages exchanged. This integration testing approach provides an engine template that, from the log data, generates reports (step 4) conforming defined in templates (T1 and T2) (1).

### **2.3.3.2 Debugging**

#### **2.3.3.2.1 Zeus**

Zeus (24), (25) is a development toolkit for constructing collaborative agent applications. Zeus defines a multi-agent system design methodology, supports the methodology with an environment for capturing user specification of agents, and automatically generates the executable source code of the user-defined agents. It also provides a suite of visualization and debugging tools for the Zeus toolkit (1).

Zeus debugging tools shift the burden of inference from the user to the visualizer. The visualizer is another agent which just requests local information from agents and tries to build a global view. Some agents may not reply because they are suspended or dead, and in such scenario, the global view presented by the visualizer will be incomplete. The tool-set presented includes (1):

- A society tool: that shows the message interchange between agents in society;
- A report tool: that graphs the society-wide decomposition of tasks and the execution states of the various subtasks;
- A micro tool: for monitoring the internal states of agents;

- A control tool: for remotely modifying the internal states of individual agents. This tool is used for the administrative management of the agents in the toolkit. e.g. killing of suspending agents, adding, modifying or deleting goals, resources, tasks, coordination strategies, etc;
- A statistic tool: for collating individual agent and society-wide statistics;
- A video tool: (within the society, report and statistic tools) which can be used, video-style, to record, replay, rewind, forward, fast-rewind and fast-forward sessions from a database.

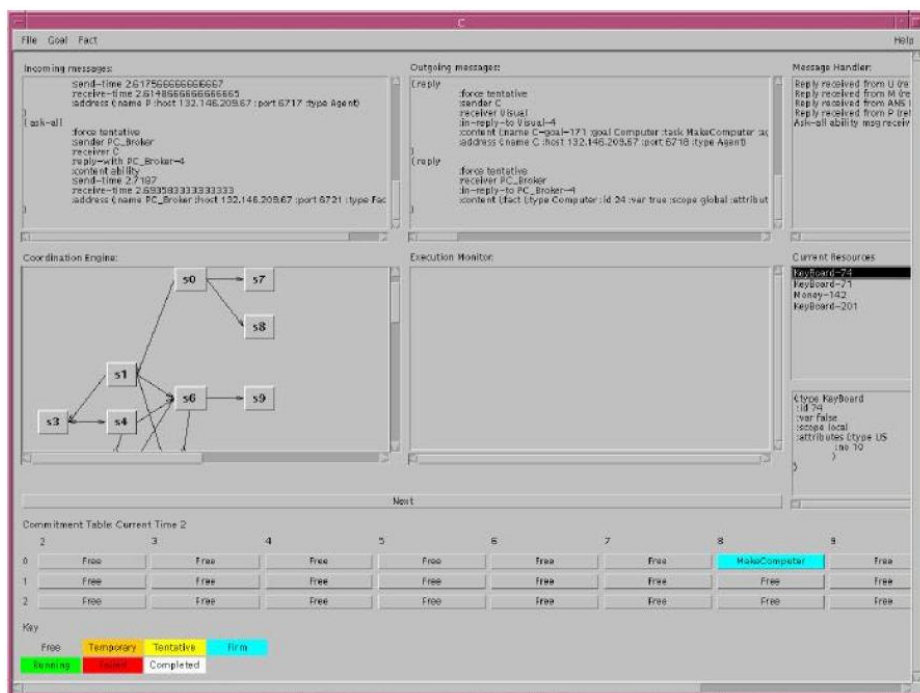


Figure 2.10: A micro tool view of an agent in Zeus Development Toolkit

The micro tool (Figure 2.10) provides views of the incoming and outgoing mail boxes, the message handler summary, the coordination graph, the execution monitor summary and a list of the items in the agent resource database. (“Adapted from (25)”). This micro tool provides users with a finer look at the internal processing of an agent. It allows them to select an agent and request to see one or more of (1):

- The messages being received by the agent.
- The messages being sent out by the agent.
- A summary of the actions taken in response to incoming messages, for example, to what module of the agent the message was dispatched to for detailed processing.

- A graphical depiction of the co-ordination process of different goals by the agent. Each node of the co-ordination graph indicates a particular state of the process, and the trace of a goal on the graph summarizes how it is being coordinated and indicates the current state of the co-ordination process.
- A diary detailing the tasks the agent has committed itself to performing, and the current status of those tasks (i.e. waiting, running, completed or failed).
- A list of the resources available to the agent including those allocated to the different tasks it has committed itself to.
- A summary of the results of monitoring executing tasks or tasks scheduled to start execution; e.g. this might indicate a task which failed to start because of inadequate resources or it might flag an executing task which has been stopped because it over-ran its scheduled allocation of processor time.

#### **2.3.3.2.2 Prometheus**

Prometheus (26), (27), (28) is a methodology which has been developed to support the building of intelligent agent systems. This methodology focuses on covering all phases of development, which includes testing and debugging that are linked to analysis and design. Their central thesis is that the design documents and systems models developed when following an agent-based software engineering methodology can be incorporated in an agent and used at run-time to provide for run-time error detection and debugging (27).

The idea is that the protocols during design specify allowable message exchanges between agents. Therefore, it is possible to check that a given execution of a system does not violate existing protocols and a debugging tool which monitors execution can hence detect violations of the protocols as specified and can notify the developer.

Monitoring can be done via eavesdropping on the communication medium (e.g. (32)), or more directly by requiring that a carbon copies of any messages sent also be sent to the debugging agent as done in the Zeus toolkit.

The approach developed in relation to Prometheus involves automatically adding code that sends copies of messages to a monitoring agent. For the debugger to reason about the correctness of a message within a particular conversation it needs to compare it against the protocol specification (1).

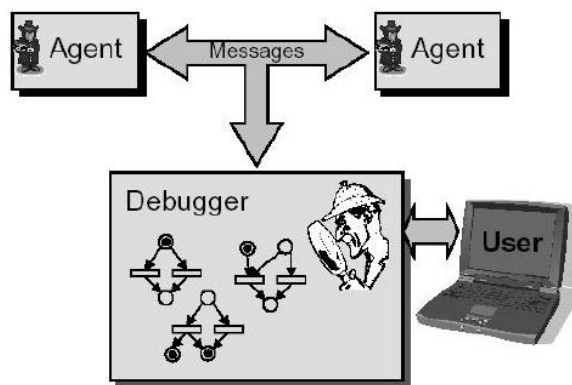


Figure 2.11: Prometheus Debugging System Design

The debugger has a library of the specified interaction protocols that it uses to detect errors. The debugger keeps a list of active conversations so that it can monitor multiple interleaved conversations simultaneously. When a message is received from an agent it is added to the appropriate conversation (or a new conversation is instantiated), and is then processed by firing any enabled transitions (Figure 2.11).

For any given conversation, the debugger does not know what protocol the agents are following. Therefore, the debugger keeps a list of possible protocols, instantiated from the interaction protocol library, which currently match the sequence of messages within the conversation.

As the conversation progresses the possible protocols list is reduced whenever a message is received that causes an error in the individual protocol. The conversation is still considered valid as long as there is at least one entry in the possible protocols list. Each time the debugger receives a message for a specific conversation an analysis is done on each protocol within the possible protocols list of that conversation to identify error situations.



As long as there is more than one protocol in the possible protocols list, errors simply lead to the conclusion that this protocol was not in fact the one that was being followed, and it is discarded from the list. However if an error is detected in the only remaining protocol, then this is reported (1).

#### 2.3.4 Multi-Agent Systems Testing and Debugging Approaches Comparison

To finish the evaluation, considering all the strengths and weakness of each approach, here associated three values to each approach according to its strategies, where: **0** means that the approach doesn't apply at all the strategy, **1** means that the approach apply weakly the strategy, and **2** means that the approach apply totally the strategy.

For the debugging approaches, a criterion for given the values was based on the fact that the tool had or not a visualization tool for that strategy (1).

	TESTING				DEBUGGING	
	Agent Level		Society Level		Agent Level	Society Level
	Black-box	White-box	Black-box	Integration		
PASSI	2	2	0	0	1	0
Test Agent	2	2	2	2	0	0
Madkit	2	2	2	2	0	0
XMLaw	0	0	0	1	0	1
ZEUS	2	2	2	2	2	2
Prometheus	0	0	0	1	0	1

Table 2.1: Multi-Agent Systems Testing and Debugging Approaches Comparison

The most suitable and complete strategy would be the integration of the one implemented in MadKit (2) which uses the record/replay mechanism, with the one implemented in Zeus (24) which have all views necessary for the debugging phase, and with the one implemented in Prometheus (26), which has a thoroughly verification between the design phase with the multi-agent execution (1).

## 2.4 MAS unit testing approach using Mock Agents

We adopt a unit testing approach for MASs (8). The main purpose of this approach is to help MASs developers in testing each agent individually. It relies on the use of *Mock Agents* to guide the design and implementation of agent unit test cases. The implementation of this approach is built on top of JADE platform (33). In order to allow the execution of the JADE unit test cases, the JUnit test framework (34) is extended.

### 2.4.1 A unit test approach for MASs

This testing approach tests the smallest building blocks of the MAS: *the agent*. Its basic idea is to verify whether each agent in isolation respects its specifications under normal and abnormal conditions. Figure 2.12 depicts a test of agent A which needs a service provided by an agent that plays role B. In this figure, A is the Agent Under Test (AUT) (8).

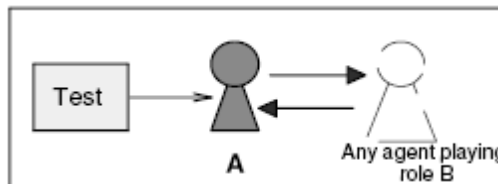


Figure 2.12: Unit testing Agent A

In order to test A in isolation, a valuable strategy is to define a “dummy” version of B, usually called stub. Stubs are fake implementations of production code that return canned results. A Mock Object is a regular object that acts as a stub, but also includes *assertions* to instrument the interactions of the target object with its neighbors.

A Mock Agent is a regular agent that communicates with just one agent: the AUT. It has just one 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 the messages that should be received from it. By testing an agent in isolation using Mock Agents, the programmer is forced to consider the agent interactions with its collaborators (or competitors), possibly before those collaborators (or competitors) exist (8).

This agent unit testing approach has two main concerns: (i) the design of a Test Case based on the use of Mock Agents; (ii) and the Test Case execution which relies on the Agent Monitor

to notify when the test script (codified in Mock Agent plan) was concluded. Next sections mention briefly these two concerns (8).

#### 2.4.1.1 Test-Case Design based on Mock Agents

Test-case design is so important because complete testing is almost impossible. The obvious strategy, then, is to try to make tests as complete as possible. Given constraints on time and cost, the key issue of testing becomes: What subset of all possible test cases has the highest probability of detecting the most errors?. The least effective test-case design methodology is to arbitrarily choose a set of test cases. In terms of the likelihood of detecting the most errors, an arbitrarily selected collection of test cases has little chance of being an optimal, or even close to optimal, subset (8).

Unit test approaches for MASs proposed so far does not define a methodology for test-case selection. Below, an error-guessing (35) test-case design technique is presented. 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 (8).

The process is as follows (8):

```
1. For each 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 a 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.
```

Algorithm 2.1:Test-Case Design based on Mock Agents

This technique should be applied for each agent of the MAS, or a subset of the agents responsible for the “core” functionalities of the MAS. Such a choice will be guided by the time and cost constraints previously mentioned. At the end of this process, a Mock Agent comprises a set of expected behaviors (under successful and exceptional scenarios) of an agent interacting with AUT (8).

#### 2.4.1.2 Test Case Execution

According to the unit test approach, the plan of a Mock Agent comprises the logic of the test. Each Test Case just starts the AUT and the corresponding Mock Agent(s) and waits for a notification from the Agent Monitor – informing that the interaction between the agents have finished – in order to ask the Mock Agent(s) whether or not the AUT acted as expected (8).

#### **2.4.2 Applying the unit test approach on top of JADE**

In order to validate this unit testing approach, it has been applied on top of JADE (36). JADE is an object-oriented framework to develop agent applications in compliance with FIPA specifications for interoperable MASs.

##### **2.4.2.1 JADE Mock Agent**

An agent in the JADE platform is defined as a Java class that extends the base Agent class from JADE framework. Each Agent contains its own thread of execution and defines a set of behaviors. A behavior is a concept in JADE platform that represents a logical activity unit of a JADE agent (36).

##### **2.4.2.2 JADE Test Suite and Test Cases**

Instead of creating a unit testing tool from scratch to exercise the Test Cases defined in this unit test approach, the JUnit framework (34) is extended to support JADE agents tests. The reason for that is to lower the developers' learning curve providing a simple, and widely used testing framework architecture. JUnit was already ported to many other languages in different paradigms (e.g. CppUnit, NUnit, PyUnit, and XMLUnit). All these frameworks, known as xUnit family of tools, have the same basic architecture (8).

### **2.5 Tropos Requirements-Driven Framework**

The increasing interest in software agents and multi-agent systems has recently led to the development of new methodologies based on *agent concepts*. One of conceptual frameworks is SKwyRL (37), which is a social pattern framework based on five complementary modeling dimensions, to introspect *social patterns*. The first two dimensions **Social** and **Intentional** of the architecture are specific to MAS. The last three dimensions **Structural**, **Communicational**, and **Dynamic** of the architecture are also relevant for traditional (non-agent) systems, but they have been adapted and extended with agent-oriented concepts (38).

The agent architecture focused by this framework is a deliberative architecture called BDI (Belief-Desire-Intention) (39). The chosen target environment for agent-oriented development is JACK (40), since it supports implementation for BDI architectures. The work in (41) introduces the generation of code from a given agent specifications to JACK platform. However, this approach does not present an explicit method for applying the patterns during the development of multi-agent systems. The patterns are applied in an ad hoc way and their choice is not clearly justified (7).

We are working based on Tropos framework (42) - a requirements-driven framework. Tropos aimed at building software that operates within a dynamic environment. It adopts the concepts and models offered by *i\** (43) framework which includes concepts such as *actor* (actors can be *agents*, *positions* or *roles*)<sup>1</sup>, as well as social dependencies among actors including *goal*, *softgoal*, *task* and *resource* dependencies (7).

Tropos proposes a set of design patterns focusing on social and intentional aspects (41). The so-called *social patterns* offer a microscopic view of a multi-agent system (MAS) at the detailed design phase to express in richer details each component presents in the architectural design of the system (44). However, Tropos does not define a detailed description of the *social patterns* as well as a systematic way to choose and apply them in order to refine the architectural components of the software under development (7).

Hence, (7) use a process for addressing this issue and a *template* for describing the social patterns consisting of the following elements: **Name**, **Intent**, **Applicability**, **Motivation Example** and **Participants**. This template is complemented by three extended UML diagrams (45) aimed at JADE (33) oriented implementation: **Collaboration**, **Structure**, and **Behaviour**. These diagrams are platform independent. The purpose is to enable easy implementation of the *social patterns* using other FIPA compliant platforms, besides JADE. These diagrams along with the template are sufficient to capture the information required to describe an *agent-oriented design pattern*. These diagrams focus on JADE implementation aiming at an *automatic code generation*, which our eclipse plug-in actually does.

## 2.6 Taxonomy of Agent Social Design Patterns

Agent-oriented software engineering is concerned with the use of agents in the development of complex distributed systems, especially in open and dynamic environments. In order to reduce development costs and promote reuse, efforts are underway to investigate the concept of *patterns* (46). A software pattern describes a *recurring problem* and *solution*; it may address conceptual, architectural or design problems (6). The efficient utilization of design patterns has the potential to a successful implementation of multi-agent systems (47).

### 2.6.1 Social Patterns Categories

Designers can be guided by a catalogue of multi-agent patterns which offer a set of standard solutions. Considerable work has been done in software engineering for defining software patterns (see e.g., (48)). *Tropos* has proposed a set of design patterns, named *Social Patterns* (41), focusing on social and intentional aspects that are recurrent in multi-agent and cooperative systems. The framework presented in (44) has classified them into two categories: Pair patterns, Mediation patterns (7):

#### 2.6.1.1 Pair patterns category

Such as booking, call-for-proposal, subscription, or bidding which describe direct interactions between negotiating agents:

**Booking pattern:** involves a client and a number of service providers (service providers are those agents which can perform some services). The client issues a request to book some resource from a service provider. The provider can accept the request, deny it, or propose to place the client on a waiting list, until the requested resource becomes available when some other client cancels a reservation.

**Call-For-Proposal pattern:** involves an initiator and a number of participants. The initiator issues a call for proposals for a service to all participants and then accepts proposals that offer the service for a specified cost. The initiator selects one participant to supply the service.

---

<sup>1</sup> A role is an abstract actor. Concrete, physical agents such as human beings or software agents play roles. A position is a collection of roles that are typically played by a single agent.

**Subscription pattern:** involves a yellow-page agent and a number of service providers. The providers advertise their services by subscribing to the yellow pages. A provider that no longer wishes to be advertised can request to be unsubscribed.

**Bidding pattern:** involves an initiator and a number of participants. The initiator organizes and leads the bidding process, and receives proposals. At every iteration, the initiator publishes the current bid; it can accept an order, raise the bid, or cancel the process.

#### 2.6.1.2 Mediation patterns category

Such as monitor, broker, matchmaker, mediator, embassy, or wrapper which feature intermediary agents that help other agents to reach an agreement on an exchange of services:

**Monitor pattern:** subscribers register for receiving, from a monitor agent, notifications of changes of state in some subjects of their interest. The monitor accepts subscriptions, request notifications from subjects of interest, receives notifications of events and alerts subscribers to relevant events. The subject provides notifications of state changes as requested.

**Broker pattern:** broker agent is an arbiter and intermediates the access to services of an agent (provider) to satisfy the request of a client. Clients access the service of providers by sending requests via the broker. A broker tasks include locating the appropriate provider, forwarding the request to the provider and transmitting results and exceptions back to the client.

**Matchmaker pattern:** a matchmaker agent locates a provider corresponding to a consumer request for service, and then hands the consumer a direct handle to the chosen provider. Contrary to the broker who directly handles all interactions between the consumer and the provider, the negotiation for service and actual service provision are two distinct phases.

**Mediator pattern:** a mediator agent mediates interactions among agents. An initiator addresses the mediator in place of asking directly another colleague, the performer. The mediator has acquaintance models of colleagues and coordinates the cooperation between them. Conversely, each performer has an acquaintance model of the mediator. While a broker

only intermediates providers with consumers, a mediator encapsulates interactions and maintains models of initiators and performers behaviours over time.

**Embassy pattern:** an embassy routes a service requested by a foreign agent to a local one and handle the response back. If the access to the local agent is granted, the foreign agent can submit messages to the embassy for translation. The content is translated in accordance to a standard ontology. Translated messages are forwarded to target local agents. The results of the query are passed back out to the foreign agent, translated in reverse.

**Wrapper pattern:** incorporates a legacy system into a multi-agent system. The wrapper agent interfaces the clients to the legacy by acting as a translator between them. This ensures that communication protocols are respected and the legacy system remains decoupled from the rest of the agent system.

## 2.7 Using Mocks and Code Refactoring in TDD

### 2.7.1 Test Driven Development

Test Driven Development (TDD) is more than just unit testing. Probably the most significant part of TDD, also known as Test First Development, is that the tests are written *first*. Before a single line of a method has been written, the test for that method should exist. Apart from completely trivial code, tests are written for nearly all production code; code does not exist unless a test has been written for it (49).

### 2.7.2 Code Refactoring

There are actually two distinct phases of coding using Test Driven Development (50). The first stage involves writing just enough code for the test to pass and not worrying too much about the elegance of the code itself. The second stage is just as important as the first – refactoring (49).

#### **Definition 2.9: Refactoring**

*Refactoring is the process of rewriting existing code without changing its external behaviour. That is, we change not what the code does but how it does it (49).*



We refactor code to: remove duplication, make maintenance easier and improve the clarity of our code (51). The TDD method is to write a little test, write a little code, write another little test, write some more code and so on. Each of the tiny code writing iterations consists of the coding and refactoring stages (49).

### **2.7.3 Mocks and Code Refactoring**

Mock objects do not only help us to write self-contained, focused unit tests. They can also be a powerful tool in “Test Driven Development”. In the same way that writing the unit test for a piece of code before the actual production code itself is written can help in creating lean functional code, writing a mock for an object that has not yet been implemented can help in developing effective domain objects. In both cases the programmer is forced to think about the “interface” his/her code presents. By creating a mock object before the production code it is a simple matter to extract the interface from the mock when implementing the domain object (49).

## **2.8 Conclusion**

Testing is essential for software engineering. Approaches for testing from conventional software engineering fail to meet the specific requirements. Unfortunately, there is no gold standard to testing in Multi-Agent System. In consequence, tools support is insufficient. As there is a broad spectrum of domains for agent technology, different testing approaches are required taking into account the actual application and domain. Consequently, developers of Multi-Agent Systems have to design, execute, and evaluate complex testing procedures by themselves. Different approaches to testing were discussed with respect to the different classes. These approaches have been successfully applied to development of multi-agent systems in logistics. However, these approaches are in need of methodology and tool support in order to fit into real-world software development.

## CHAPTER 3: PROPOSED FRAMEWORK FOR DESIGNING TEST CASES TO MAS

### 3.1 Introduction

In this chapter, we present our contribution by giving detailed design for our framework used in designing test cases based on Mock Agents. We adopt the Agile software TDD process, and depict our framework components and their integration. Here, we present a detailed system architecture using a UML class diagram, and an Entity Relationship Diagram (ERD) of our MockAgentDesigner Eclipse Plug-in. Finally, the test cases implementation through our system and adding new test design pattern templates are provided.

### 3.2 Approach

#### 3.2.1 Agile Software Engineering Test Driven Development Process

In this thesis, we adopt the *agile software engineering* approach that addresses testing continuously within the implementation process (11). Developers formulate *test-cases* before or during implementation, which may be executed automatically on demand. This procedure is also known as “*Test-driven Development*” (12). We focus on the *unit test level* where the implementation is tested by *unit test* during the coding.

#### 3.2.2 Social Software Design Patterns

We design test patterns for the ten most famous MAS social design patterns found in the literature of Agent Design Patterns, named *social patterns* (41) that focus on social and intentional aspects which are *recurrent* in multi-agent and cooperative systems.

### 3.3 Framework components

Our framework consists of five standard components as in the unit test approach (8) which uses Mock Agents to guide the design of each test case:

- **Test Suite** which consists in a set of *test cases* and a set of operations performed to prepare the test environment before a *test case* starts.

- **Test Case** defines a scenario – a set of conditions – to which an *agent under test* is exposed, and verifies whether this agent obeys its specification under such conditions.
- **Agent Under Test (AUT)** is the agent whose behavior is verified by a *test case*.
- **Mock Agent** consists of a fake implementation of a real agent that interacts with the AUT. Its purpose is to simulate a real agent strictly for testing the AUT.
- **Agent Monitor** is responsible for monitoring agents life cycle in order to notify the *test case* about agents states.

Figure 3.1 presents a collaboration diagram that illustrates the interaction between the components of the agent unit test approach integrated with our Eclipse plug-in:

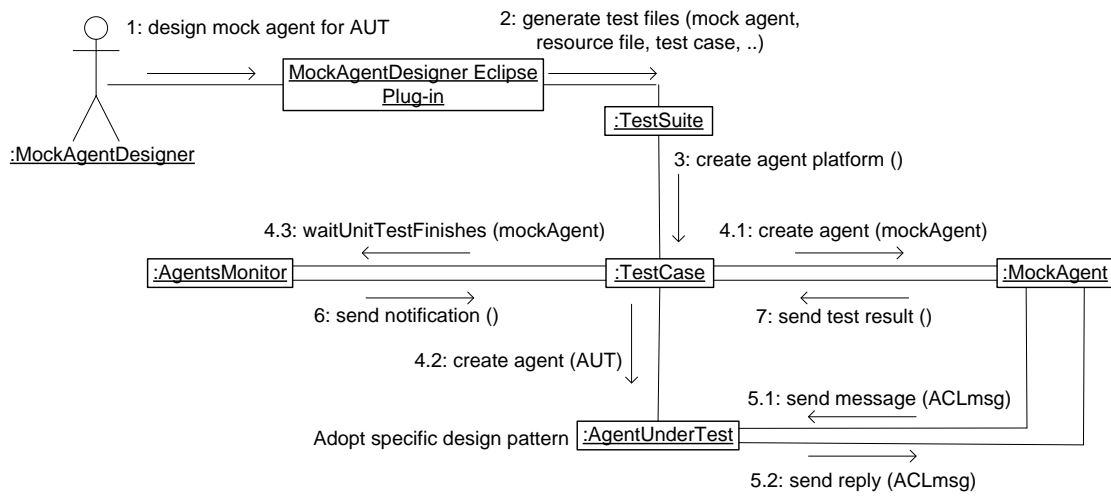


Figure 3.1: Framework of MAS unit test approach integrated with MockAgentDesigner Plug-in  
The MockAgentDesigner designs the mock agent for his/her AUT (step 1) using our Eclipse plug-in that generates the test files (mock agent, associated resource file, and AUT test case) (step 2) which tests his/her AUT following a specific design pattern.

Each agent unit test follows the common structure depicted in Figure 3.1. In step 3, the TestSuite creates the agent platform and whatever element needed to set up the test environment. After that, a TestCase is started. The TestCase creates MockAgent to every role that interacts with the AUT 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 finishes (step 4.3).

**Note:** In scenarios where the `MockAgent` initiates the interaction process (test driver), it should be created after the AUT. Also, in some scenarios, where an AUT needs to interact with more than one `MockAgent`, we can have simple *mock agents* that only send simple messages and more sophisticated *mock agents* that validate the order the messages are exchanged and its content.

At this point, the AUT and the `MockAgent` start to interact. The `MockAgent` sends a message to the AUT, and it replies (steps 5.1 and 5.2) or vice-versa. They can 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 `TestCase` that the interaction between the `MockAgent` and the AUT has concluded (step 6). Last, the `TestCase` asks the `MockAgent` whether or not the AUT acts as expected (step 7).

### 3.4 Suggested Support Concept

In this section, we discuss the *support process* that is presented to the mock agent unit test case developer. The mock agent designer has the facility to swap between the mock agent design (through our proposed interface) and the implementation (through the generated mock agent code) using Eclipse SDK refactoring.

After the mock agent designer uses our Eclipse plug-in to generate the mock agent code that codifies the test script testing the AUT (following a specific agent design pattern), the designer can use the reflection of the Eclipse SDK to reflect the changes made in the AUT (to complete its actual behaviour) in the resulted mock agent. By this way, the generated mock agent code can walk step by step with the AUT in order to codify the complete test scenario required to test the final behaviour of the AUT.

Our Eclipse Plug-in helps in providing the first implementation that triggers the refactoring process. When performing a *refactoring operation*, the user can optionally preview all of the changes resulting from a *refactoring action* before choosing to carry them out. When previewing a *refactoring operation*, the user is notified of *potential problems* and is presented with a list of the changes the *refactoring action* will perform. If the user does not preview a

*refactoring operation*, the change is made and any resultant problems are shown. If a problem is detected that does not allow the refactoring to continue, the operation is halted and a list of problems is displayed.

### 3.5 System Architecture

The architecture of our proposed framework is illustrated in Figure 3.2:

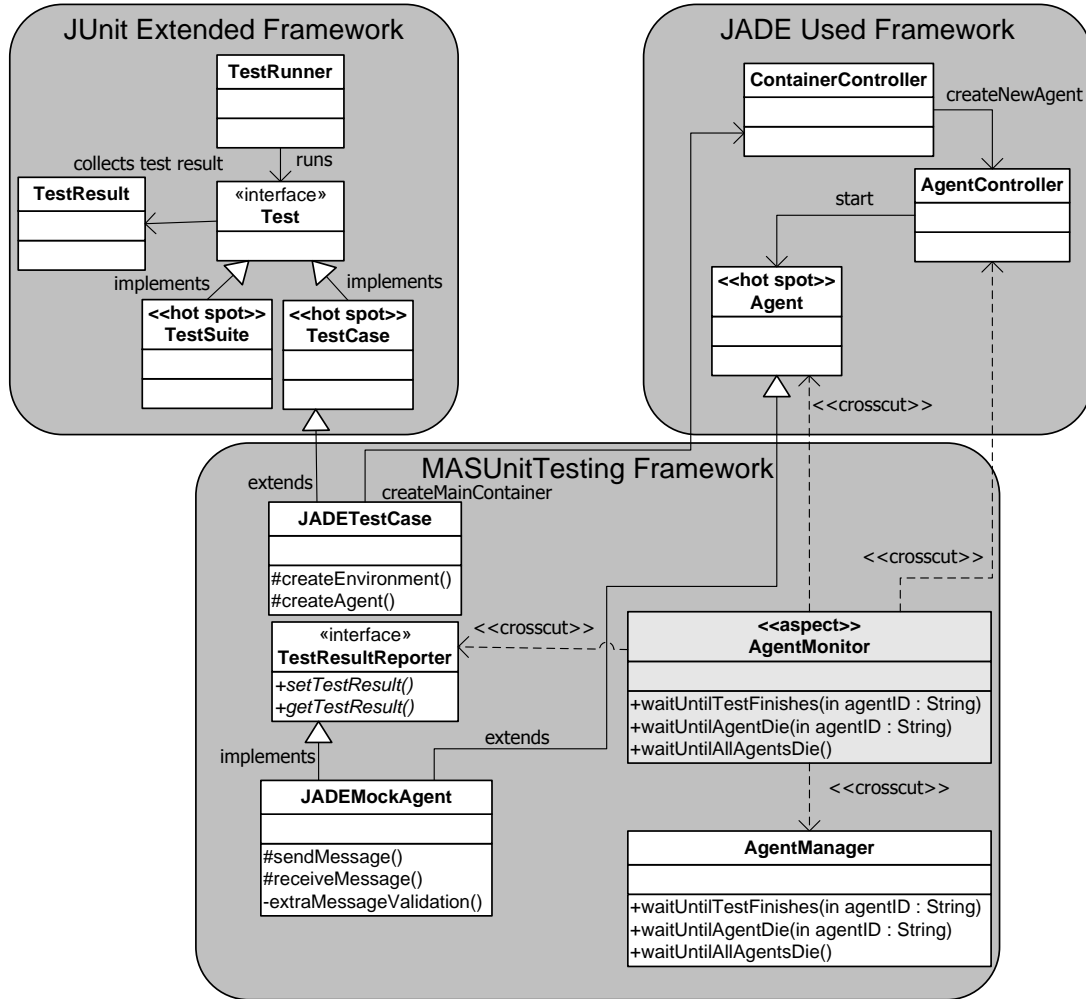


Figure 3.2: UML class diagram for the MAS unit testing approach upon JADE Platform

#### 3.5.1 JUnit Framework

**Test:** A `Test` can be run and collect its results; `TestResult`. The `Test` interface contains two abstract methods `countTestCases()` that counts the number of test cases that will be run by this test, and `run(TestResult result)` that runs a test and collects its result in a `TestResult` instance.

**TestRunner:** This class is used to run tests; the user enters the name of a class which either provides a static suite method or is a subclass of `TestCase`. `TestRunner` takes as an optional argument the name of the `testCase` class to be run.

**TestResult:** A `TestResult` collects the results of executing a test case. It is an instance of the Collecting Parameter pattern (52). The test framework distinguishes between *failures* and *errors*. A failure is anticipated and checked for with assertions. Errors are unanticipated problems like an `ArrayIndexOutOfBoundsException`.

**TestSuite:** A `TestSuite` is a Composite of Tests. It runs a collection of test cases. A `TestSuite` can extract the tests to be run automatically. To do so, the user passes the class of his/her `TestCase` class to the `TestSuite` constructor. The constructor creates a suite with all the methods starting with "test" that take no arguments.

**TestCase:** A `TestCase` defines the fixture to run multiple tests. To define a test case: Implement a subclass of `TestCase`. Define instance variables that store the state of the fixture. Initialize the fixture state by overriding the `setUp` method. Call the clean-up method after a test by overriding the `tearDown()` method that removes all agents from the environment after the execution of each test method.

### 3.5.2 JADE Framework classes used in the approach

**ContainerController:** The class `AgentContainer` is a Proxy class, allowing access to a JADE agent container. Invoking methods on instances of this class, it is possible to request services from in-process agent containers. This class must not be instantiated by applications. Instead, we use the `createContainer ()` method in class `Runtime`.

**AgentController:** The class `AgentController` is a Proxy class, allowing access to a JADE agent. Invoking methods on instances of this class, it is possible to trigger state transition of the agent life cycle. This class must not be instantiated by applications. Instead, we use the `createAgent ()` method in class `AgentContainer`.

**Agent:** 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 using `ACLMessage` objects, both unicast and multicast with optional pattern matching. Complete Agent Platform life cycle support, including starting, suspending and killing an agent. Scheduling and executing multiple concurrent activities. Simplified interaction with FIPA system agents for automating common agent tasks (Directory Facilitator registration, etc.). Application programmers must write their own agents as `Agent` subclasses, adding specific behaviours as needed and exploiting `Agent` class capabilities.

### 3.5.3 MASUnitTesting Framework

**JADETestCase:** The *test suite* (`JADETestCase`) consists of a set of *test cases* and a set of operations performed to prepare the test environment before a *test case* (`AUTTestCase`) starts. The `JADETestCase` class (which implements the *test suite* concept) extends the `TestCase` class from JUnit. This class defines a set of concrete and abstract methods which support the implementation of the test methods (equivalent to the *test case* concept). The `createEnvironment()` 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. Each test method will be able to include agents in such environment by calling `createAgent()`. The `tearDown()` method removes all agents from the environment after the execution of each test method (8).

**AgentMonitor:** The `AgentMonitor` class is responsible for monitoring agents life cycle in order to notify the Test-Case about agents states. To fulfill this requirement `AgentMonitor` implements the following methods: `waitUntilAllAgentsDie()`, `waitUntilTestFinishes()`, `waitUntilAgentDies()`. Each method periodically analyzes the agents being monitored and make the *test case* (JUnit test method), that called them, wait until the condition specified in methods signature is reached (8).

**AgentManager:** The `AgentManager` class is an auxiliary class that contains empty-body implementations of all `wait*` methods defined in the `AgentMonitor` class. The `AgentMonitor` class intercepts all methods of the `AgentManager` class, and replaces each

empty body method by the implementation provided by the `AgentMonitor` (using around advices) (8).

**JADEMockAgent:** The `JADEMockAgent` class, as any other agent in this platform, extends the `Agent` class, as illustrated in Figure 3.2. 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 (8). The `JADEMockAgent` implements two methods: `sendMessage()` 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). Its implementation is according to the Template Method design pattern (48) in order to enable the developer to perform additional assertions in the received messages (8).

**TestResultReporter:** After executing its plan (equivalent to test script), the `JADEMockAgent` needs to report the test result (success or failure) to the Test Case, which in counterpart, will be in charge of examining the test result (8). There are many ways of reporting the result of a test. Some of them are: (i) to include the test result in an ACL specific message and send it to another agent that would generate a textual/graphical report; and (ii) to define an interface that contains a set of methods that should be implemented by an agent that wants to report the result of a test script (8). In this particular implementation, the second alternative has been chosen. Thus, the `JADEMockAgent` class implements the `TestResultReporter` interface illustrated in Figure 3.2.

### 3.5.4 Proposed MAS Application

**Agent Under Test:** The Agent Under Test (AUT) is the agent whose behavior is verified by the *test case* (8). This AUT follows one of the *social agent design patterns*. The *mock agent designer* needs to reflect the changes made in the AUT to complete its actual behaviour in the resulted *mock agent* to represent the complete test scenario required to test the final behaviour of the AUT.



**Mock Agents:** For example, in order to implement the unit test of `BookSellerAgent` in a simple book trading MAS, our first test-case should be the simple success case briefly described in the table 3.1. To implement this test-case all we need is to write a *mock agent* that simulates this scenario. In order to implement a second *test case* which verifies an *exceptional scenario* we just need to implement an extra plan (JADE Behavior) in our *mock agent* class (8).

Agent	BookSellerAgent
Roles	BookSeller
Interacting Roles	BookBuyer
Successful Scenario	BookSellerAgent sells a book to an agent playing BookBuyer role.
Exceptional Scenario	A BookBuyer agent can send a “cfp” message requesting a specific book, and afterwards sends a purchase message trying to buy a different book.

Table 3.1: Unit Test Case Template

Figure 3.3 (8) illustrates the partial code of the *mock agent* which contains only the behavior that verifies the *exceptional scenario* described in Table 3.1.

```

1. public class MockBookBuyerAgent extends JADEMockAgent {
2.     ...
3.     protected void setup() {
4.         ...
5.         addBehaviour(new TestScenario());
6.     }
7. }
8. private class TestScenario extends OneShotBehaviour {
9.     public void action(){
10.        try {
11.            ...
12.            sendMessage(msgType.CFP, sellerID, bookTitle);
13.            reply = receiveReply(6000, msgType.PROPOSE);
14.            sendMessage(msgType.Accept, sellerID, otherTitle);
15.            reply2 = receiveReply(6000, msgType.FAIL);
16.        } catch (ReplyReceptionFailed e) {
17.            setTestResult( prepareMessageResult(e));
18.        }
19.    }
20. }

```

Figure 3.3: Partial code of a Mock Agent

As we can see the *mock agent* has just one plan (represented by a JADE Behavior called `TestScenario`) to test `BookSellerAgent`. The `TestScenario` class codifies the “logic” of the *test case*. This Behavior allows the agent to send, receive and check the content of the messages received from `BookSellerAgent` (the AUT in this example).

The following methods: `sendMessage()`, `receiveMessage()`, `setTestResult()`, and `prepareMessageResult()` are methods from `JADEMockAgent` class - described in section 3.6.3 – that eases the implementation of the *mock agent*. The `sellerID` variable of

type `AID` (used in line 11) contains the identification (agent local name) of the `BookSellerAgent` instance under test. In case, that the *mock agent* creates a reply to the AUT, the `createReply ()` method creates a new `ACLMessage` that is a reply to the received message. It sets the following parameters of the new message: receiver (the identification “agent local name” of the Client agent instance under test), language, ontology, protocol, conversation-id, in-reply-to, reply-with. The mock agent designer needs to set the communicative-act and the content. Of course, if he wishes to do that, he can reset any of the fields.

**AUTTestCase:** The *test case* defines a scenario – a set of conditions – to which the AUT is exposed, and verifies whether this agent obeys its specification under such conditions. The concept of *test case* is represented in the JUnit Framework by what they call a *test method*. Each test method (`AUTTestCase`) will be able to include agents in the created JADE environment by calling the `createAgent()` method that exists in the *test suite* (`JADETestCase`).

The `AUTTestCase` is a subclass of `JADETestCase`. Inside this *test case*, we implement a *test method* that creates an instance of AUT (e.g., `BookSellerAgent`), and an instance of the *mock agent* (e.g., `MockBookBuyerAgent`) that interacts with this AUT. After that, the test method asks the `AgentMonitor` to be notified when the interaction between the AUT and the *mock agent* finishes in order to ask the *mock agent* whether or not the AUT acted as expected (test result). Figure 3.4 presents a partial code of this Test Case (local variables definitions are left out) (8).

```

1. public class BookSellerTestCase extends JADETestCase {
2.     ...
3.     public void testBookSelling_Success() {
4.         ...
5.         createAgent("seller", "BookSellerAgent", argS);
6.         createAgent("buyer", "MockBookBuyerAgent", argB);
7.         AgentsManager.waitUntilTestFinishes("buyer");
8.         mockAg=environment.getLocalAgent("buyer");
9.         res=((TestReporter) mockAg).getTestResult();
10.        if(!res.equals("OK")){
11.            fail(res);
12.        }
13.    }
14. }

```

Figure 3.4: Partial code of a `BookSellerTestCase`

### 3.6 MockAgentDesigner Eclipse Plug-in

The role of this section is to depict the architecture of the MockAgentDesigner Eclipse Plug-in. Figure 3.5 represents the Entity Relationship Diagram (ERD) of the proposed mock agent designer framework:

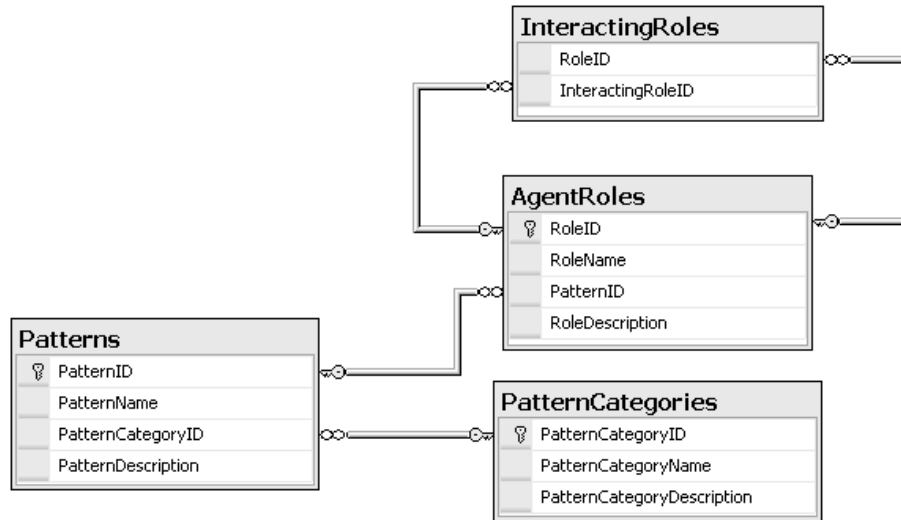


Figure 3.5: Entity Relationship Diagram of the MockAgentDesigner Framework

Every pattern category contains set of design patterns, and every design pattern follows only one pattern category, that is represented in Figure 3.5 by the one-to-many relationship between the two tables `PatternCategories` and `Patterns`. Every design pattern contains a set of agent roles, and every agent role follows only one design pattern, that is represented by the one-to-many relationship between the two tables `Patterns` and `AgentRoles`. Every agent role has a set of interacting roles, and every interacting role can interact with more than one agent role, that is represented by the two one-to-many relationships between the two tables `AgentRoles` and `InteractingRoles`.

Table 3.2 represents sample data from the `PatternCategories` table that is composed of `PatternCategory` records (`PatternCategoryID`, `PatternCategoryName`, and `PatternCategoryDescription`):

PatternCategoryID	PatternCategoryName	PatternCategoryDescription
1	PairPatterns	Describe direct interactions between negotiating agents
2	MediationPatterns	Feature intermediary agents that help other agents t...

Table 3.2: `PatternCategories` table sample data

Table 3.3 represents sample data from the `Patterns` table that is composed of `Pattern` records (`PatternID`, `PatternName`, `PatternCategoryID`, and `PatternDescription`):

PatternID	PatternName	PatternCategoryID	PatternDescription
1	BookingPattern	1	The client issues a request to book some resource from a service provider.
5	MonitorPattern	2	Subscribers register for receiving, from a monitor agent, notifications of ...

Table 3.3: Patterns table sample data

Table 3.4 represents sample data from the `AgentRoles` table that is composed of `AgentRole` records (`RoleID`, `RoleName`, `PatternID`, and `RoleDescription`):

RoleID	RoleName	PatternID	RoleDescription
1	BookingClientAgent	1	The client issues a request to book some resource from a service provider.
2	BookingServiceProviderAgent	1	The provider can accept the request, deny it, or propose to place the client on a ...
9	MonitorSubscriberAgent	5	Subscribers register for receiving, from a monitor agent, notifications of changes o...
10	MonitorAgent	5	The monitor accepts subscriptions, request notifications from subjects of interest, ...

Table 3.4: AgentRoles table sample data

Table 3.5 represents sample data of the `InteractingRoles` table that is composed of `InteractingRole` records (`RoleID`, `InteractingRoleID`):

RoleID	InteractingRoleID
1	2
2	1
9	10
10	9

Table 3.5: InteractingRoles table sample data

### 3.7 Implementation through Eclipse plug-in

This section illustrates the framework implementation through the `MockAgentDesigner` Eclipse Plug-in that is used by the mock agent designer to generate the target mock agent code.

#### 3.7.1 MockAgentDesigner main form

Figure 3.6 presents a screen shot for the main form of the `MockAgentDesigner` Eclipse Plug-in:

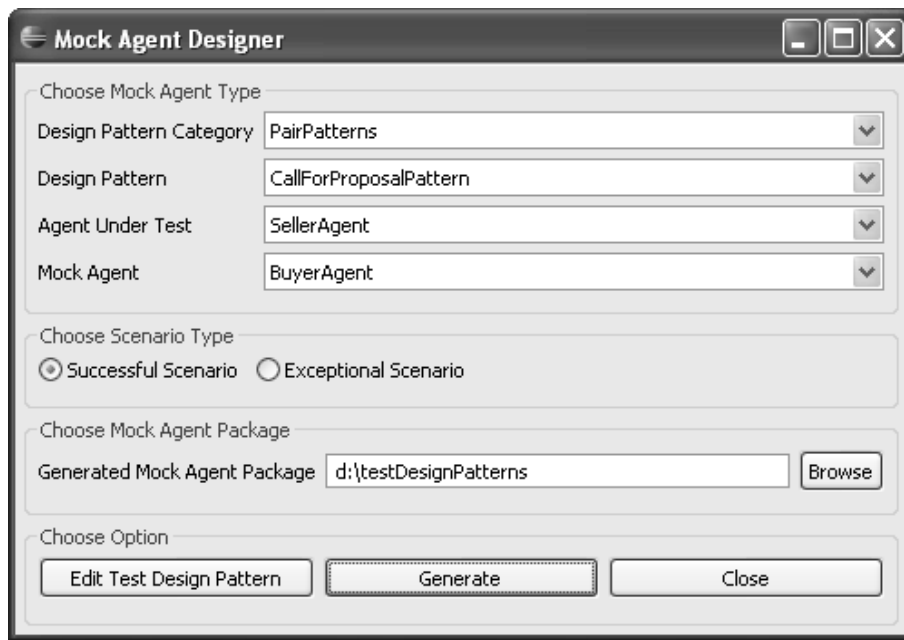


Figure 3.6: The main form of the MockAgentDesigner Eclipse Plug-in

The mock agent designer follows the following steps in order to generate the target *mock agent* file and the associated resource file:

- Choose the required *design pattern category* that his/her AUT follows. Choose the required *design pattern* from the next combo that will be loaded with all design patterns belonging to the chosen category.
- Choose the required *agent role* acting as the AUT from the next combo that will be loaded with all the exiting roles in the chosen *design pattern*. Choose the required *interacting role* acting as the *mock agent* from the next combo that will be loaded with all the roles interacting with the chosen AUT role in the chosen *design pattern*.
- Choose the *test scenario type* whether being a *successful* or an *exceptional* scenario. Browse for the package where the generated *mock agent* files will reside into.
- Click “Edit Test Design Pattern” if he/she wishes to edit the *design pattern parameters* for either the *successful* scenario or the *exceptional* one. Click the “Generate” button in order to get the required *mock agent* files generated in the chosen package.

### 3.7.2 Working Example: Service Trading

The generated mock agent file `MockBuyerAgent.java` depends on two major portions:

First, the type of the *mock agent* to which the mock agent designer wants to generate code, in this example the generated code is for the `MockBuyerAgent` that tests the behaviour of the `SellerAgent` that follows the `CallForProposalPattern` that consequently follows the `PairPatterns` category. There are some fixed code snippets for any domain that represent the skeleton of the `CallForProposalPattern`.

Second, the settings that could differ from one domain to another domain like the tick period of the ticker behaviour after which the `MockBuyerAgent` schedules a request to the `SellerAgent`. Also, the performative, conversationID and the content of the messages exchanged between the `MockBuyerAgent` and the `SellerAgent` in both of the successful scenario and the exceptional one.

There is a set of *placeholders* resident in the *mock agent template* `MockBuyerAgent.java` as resource values that are read from an associated resource file `MockBuyerAgent.properties`. The mock agent designer can change the values of those placeholders before mock agent generation for the successful and/or exceptional scenarios from the “Test Design Pattern” form that is prompted from the main form of the `MockAgentDesigner` plug-in (see Figure 3.6). Those values are saved in the test design pattern file `MockBuyerAgent.XML`.

A package structure is created inside the *mock agent package* entered by the mock agent designer (`x:\testDesignPatterns`). A folder is created with the design pattern category name (`PairPatterns`), inside which a new folder is created with the design pattern name (`CallForProposalPattern`), inside which a new folder is created with the AUT name (`SellerAgent`) in which the AUT file `SellerAgent.java` exists and finally a folder is created with the mock agent name (`MockBuyerAgent`) containing the generated mock agent file `MockBuyerAgent.java` and its associated properties file

MockBuyerAgent.properties, also a test case file SellerAgentTestCase.java is generated in the same folder.

Notice, that the supporting package MASUnitTesting is also generated inside the same *mock agent*. This package contains all the supporting classes existing in the MASUnitTesting framework that are mentioned in section 3.6.3.

Figure 3.7 represents the generated resource file MockBuyerAgent.properties that is associated with the generated file MockBuyerAgent.java that contains the values of placeholders existing in the mock agent file.

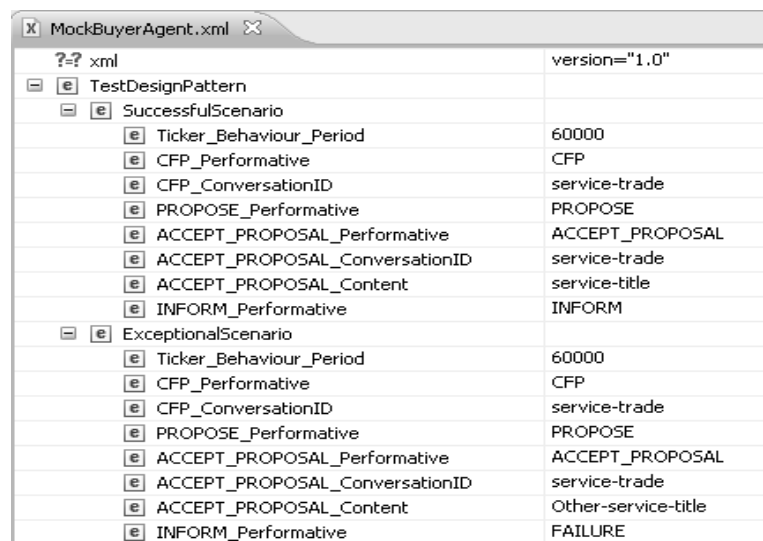
```

1 Ticker_Behaviour_Period = 60000
2 CFP_Performative = CFP
3 CFP_ConversationID = service-trade
4 PROPOSE_Performative = PROPOSE
5 ACCEPT_PROPOSAL_Performative = ACCEPT_PROPOSAL
6 ACCEPT_PROPOSAL_ConversationID = service-trade
7 ACCEPT_PROPOSAL_Content = service-title
8 INFORM_Performative = INFORM

```

Figure 3.7: The generated resource file MockBuyerAgent.properties

Figure 3.8 represents the test design pattern file MockBuyerAgent.XML from which the resource file MockBuyerAgent.properties reads the values of placeholders in the successful scenario or a sample exceptional scenario according to the “Scenario Type” option that the mock agent designer chooses in the main form of the MockAgentDesigner Eclipse Plug-in (see Figure 3.6).



MockBuyerAgent.xml	version="1.0"
TestDesignPattern	
SuccessfulScenario	
Ticker_Behaviour_Period	60000
CFP_Performative	CFP
CFP_ConversationID	service-trade
PROPOSE_Performative	PROPOSE
ACCEPT_PROPOSAL_Performative	ACCEPT_PROPOSAL
ACCEPT_PROPOSAL_ConversationID	service-trade
ACCEPT_PROPOSAL_Content	service-title
INFORM_Performative	INFORM
ExceptionalScenario	
Ticker_Behaviour_Period	60000
CFP_Performative	CFP
CFP_ConversationID	service-trade
PROPOSE_Performative	PROPOSE
ACCEPT_PROPOSAL_Performative	ACCEPT_PROPOSAL
ACCEPT_PROPOSAL_ConversationID	service-trade
ACCEPT_PROPOSAL_Content	Other-service-title
INFORM_Performative	FAILURE

Figure 3.8: The test design pattern file MockBuyerAgent.XML

Figure 3.9 represents the form where the mock agent designer can change the values of placeholders in the successful and/or exceptional scenarios by clicking on the “Edit Test Design Pattern” button in the main form of the MockAgentDesigner Eclipse plug-in (see Figure 3.6):

Figure 3.9: The “Test Design Pattern” form for the MockBuyerAgent

### 3.8 Adding new design pattern

In order to add a new design pattern in addition to the already existing ones, the first step is to decide whether it belongs to the already existing pattern categories or a new pattern category has to be defined. We will use the same example of the `CallForProposalPattern` as if it was a newly added pattern. Figure 3.10 represents the form where the user can define a new *design pattern category*:

Figure 3.10: Add new Design Pattern Category



By clicking the “Save” button, the newly added *design pattern category* is saved and a package with the *design pattern category* name is created in the following package:

**ECLIPSE\_HOME\plugins\MockAgentDesigner\_1.0.0\testDesignPatterns**

Figure 3.11 represents the form where the user can define some new *design patterns* that belong to the previously defined *design pattern category*:

The dialog box is titled "Design Pattern". It contains a "Choose Design Pattern" section with three fields: "Design Pattern Name" (text input with "CallForProposalPattern"), "Pattern Category Name" (dropdown menu with "PairPatterns"), and "Design Pattern Description" (text area with "The initiator issues a call for proposals for a service to all participants and then accepts proposals that offer the service for a specified cost."). At the bottom, there is a "Choose Option" section with five buttons: "Add", "Edit", "Save" (highlighted with a dashed border), "Delete", and "Cancel".

Figure 3.11: Add new Design Pattern

By clicking the “Save” button, the newly added *design pattern* is saved and a package with the *design pattern* name is created in the following package: **ECLIPSE\_HOME\plugins\**

**MockAgentDesigner\_1.0.0\testDesignPatterns\PatternCategoryName**

Figure 3.12 represents the form where the user can define some new *agent roles* that belong to the previously defined *design pattern*:

The dialog box is titled "Agent Role". It contains a "Choose Agent Role" section with three fields: "Agent Role Name" (text input with "SellerAgent"), "Design Pattern Name" (dropdown menu with "CallForProposalPattern"), and "Agent Role Description" (text area with "The buyer selects one seller to supply the service"). At the bottom, there is a "Choose Option" section with five buttons: "Add", "Edit", "Save" (highlighted with a dashed border), "Delete", and "Cancel".

Figure 3.12: Add new Agent Role

By clicking the “Save” button, the newly added *agent role* is saved and a package with the *agent role* name is created in the following package:

**ECLIPSE\_HOME\plugins\MockAgentDesigner\_1.0.0\testDesignPatterns\PatternCategoryName\PatternName**

Figure 3.13 represents the form where the user can define the *interacting roles* for every *agent role* existing in the newly added *design pattern*:



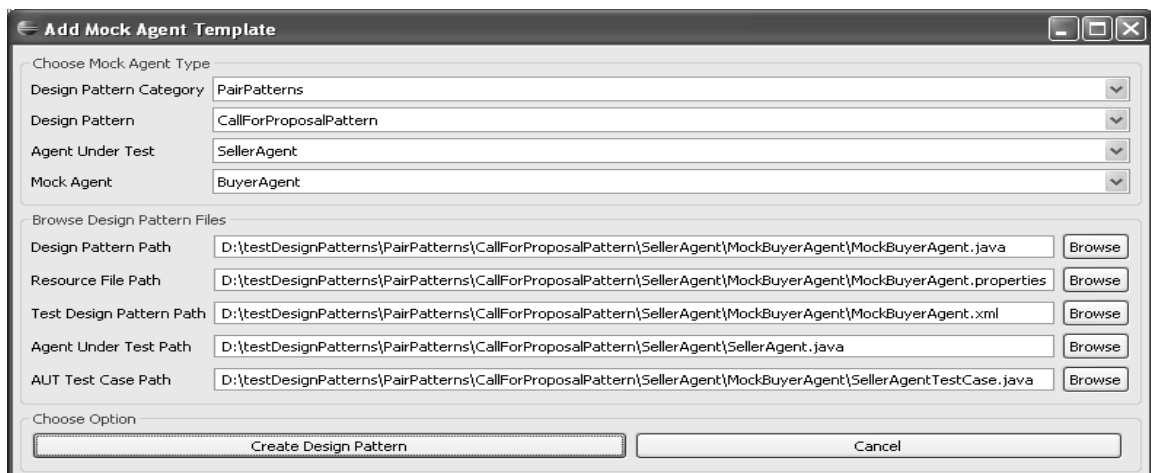
The dialog box titled "Mock Interacting Role" contains a section "Choose Mock Interacting Role" with three dropdown menus: "Design Pattern Name" (CallForProposalPattern), "Role Name" (SellerAgent), and "Interacting Role Name" (BuyerAgent). Below this is a "Choose Option" section with five buttons: "Add", "Edit", "Save", "Delete", and "Cancel".

Figure 3.13: Add new Interacting Role

By clicking the “Save” button, the newly added *interacting role* is saved and a package with the *interacting role* name preceded with the word “Mock” is created in the following package:

**ECLIPSE\_HOME\plugins\MockAgentDesigner\_1.0.0\testDesignPatterns\PatternCategoryName\PatternName\AgentRoleName**

Figure 3.14 represents the form where the user can upload a new *mock agent template* or overwrite an already existing one MockBuyerAgent.java that is used to test a specific agent role SellerAgent in the newly added *design pattern* CallForProposalPattern:



The dialog box titled "Add Mock Agent Template" contains a section "Choose Mock Agent Type" with four dropdown menus: "Design Pattern Category" (PairPatterns), "Design Pattern" (CallForProposalPattern), "Agent Under Test" (SellerAgent), and "Mock Agent" (BuyerAgent). Below this is a "Browse Design Pattern Files" section with five text input fields and "Browse" buttons: "Design Pattern Path", "Resource File Path", "Test Design Pattern Path", "Agent Under Test Path", and "AUT Test Case Path". At the bottom is a "Choose Option" section with two buttons: "Create Design Pattern" and "Cancel".

Figure 3.14: Add new Mock Agent Template

The template file `MockBuyerAgent.java`, the resource file `MockBuyerAgent.properties`, the test design pattern file `MockBuyerAgent.XML` and the AUT test case file `SellerAgentTestCase.java` are saved in the following path:  
`ECLIPSE_HOME\plugins\MockAgentDesigner_1.0.0\testDesignPatterns\PairPatterns\CallForProposalPattern\SellerAgent\MockBuyerAgent\`

The AUT file `SellerAgent.java` is saved in the following path:  
`ECLIPSE_HOME\plugins\MockAgentDesigner_1.0.0\testDesignPatterns\PairPatterns\CallForProposalPattern\SellerAgent\`

For avoiding compilation errors, the imported template file `MockBuyerAgent.java` must follow the *package* and *import* structure illustrated in Figure 3.15:

```

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 3.15: Package and Import structure in the imported `MockBuyerAgent` template

The mock agent template creator must take care of the following points:

- The mock agent resides in the following package:  
**`DesignPatternCategory.DesignPattern.AgentUnderTest.MockAgent`**
- Import all the required Java, Jade and JUnit classes as illustrated in Figure 3.15
- The mock agent (e.g., `MockBuyerAgent`) extends the `JADEMockAgent` class that exists in the `MASUnitTesting` package.

For avoiding compilation errors, the imported AUT file `SellerAgent.java` must follow the *package* and *import* structure illustrated in Figure 3.16:

```

1 /**
2  * Agent Under Test (AUT): is the agent whose behavior is verified by a Test Case.
3  */
4
5 package PairPatterns.CallForProposalPattern.SellerAgent;
6
7 import java.util.*;
8
9 import jade.core.Agent;
10 import jade.core.behaviours.CyclicBehaviour;
11 import jade.core.behaviours.OneShotBehaviour;
12 import jade.lang.acl.ACLMessage;
13 import jade.lang.acl.MessageTemplate;
14 import jade.domain.DFService;
15 import jade.domain.FIPAAException;
16 import jade.domain.FIPAAgentManagement.DFAgentDescription;
17 import jade.domain.FIPAAgentManagement.ServiceDescription;
18
19 public class SellerAgent extends Agent {
20 }

```

Figure 3.16: Package and Import structure in the imported SellerAgent file

The importer must take care of the following points:

- The AUT file resides in the following package:  
**DesignPatternCategory.DesignPattern.AgentUnderTest**
- Import all the required Java and Jade classes as illustrated in Figure 3.16
- The AUT (e.g., SellerAgent) extends the class Agent that exists in the following jade.core package

For avoiding compilation errors, the imported AUT test case file SellerAgentTestCase.java must follow the *package* and *import* structure illustrated in Figure 3.17:

```

1 /** Test Case: defines a scenario - a set of conditions - to which an Agent Under Test is exposed,
2  * and verifies whether this agent obeys its specification under such conditions.
3  */
4 package PairPatterns.CallForProposalPattern.SellerAgent.MockBuyerAgent;
5
6 import jade.core.AID;
7 import jade.wrapper.AgentController;
8 import jade.wrapper.AgentContainer;
9
10 import junit.framework.Test;
11 import junit.framework.TestResult;
12 import junit.framework.TestSuite;
13
14 import MASUnitTesting.AgentManager;
15 import MASUnitTesting.JADEMockAgent;
16 import MASUnitTesting.JADETestCase;
17 import MASUnitTesting.TestResultReporter;
18
19 public class SellerAgentTestCase extends JADETestCase {
20 }

```

Figure 3.17: Package and Import structure in the imported SellerAgentTestCase file

The importer must take care of the following points:

- The AUT test case file resides in the following package:  
**DesignPatternCategory.DesignPattern.AgentUnderTest.MockAgent**
- Import all the required Jade and JUnit classes as illustrated in Figure 3.17

- The AUT test case (e.g., `SellerAgentTestCase`) extends the class `JADETestCase` that exists in the `MASUnitTesting` package.

### 3.9 Conclusion

In this chapter, we present our approach and our framework components. We introduce a tool for designing test cases based on a unit testing approach for MASs (8). The unit testing framework is consisting of five standard components: *Test Suite*, *Test Case*, *AUT*, *Mock Agents*, and *Agent Monitor*.

We discuss the *architecture* of our system that depicts the low level design of the adopted *unit test framework* integrated with our *Eclipse plug-in*. A UML class diagram is given for the components of the unit test framework. Also, an Entity Relationship Diagram is given for the Eclipse Plug-in.

The *test cases implementation* through our system is depicted. The mock agent designer can generate the target mock agent files, which depend on the type of the mock agent and the settings that could differ from one domain to another that are set through some placeholders. Those placeholders are resident in the mock agent file as resource values that are read from an associated resource file.

The mock agent designer has the ability to add a new design pattern in addition to the already existing ones and configuring the values of the placeholders of the added design pattern in successful and/or exceptional scenarios that are stored in corresponding XML files. In our work, we provide implementations for a vast majority of agent design patterns.

## CHAPTER 4: DESIGN OF TEST DESIGN PATTERNS

### 4.1 Introduction

In this chapter, we present each of the agent social patterns complemented with its test design patterns. In order to present those social patterns in a methodological way, we follow the design process mentioned in (7) that consists of a *template* describing each pattern in some main points as: **Name**, **Intent**, **Applicability**, **Motivation Example** and **Participants**. This template is complemented by three extended UML diagrams (45) aimed at JADE (33) oriented implementation: **Collaboration**, **Structure**, and **Behaviour** diagrams.

#### 4.1.1 Design Pattern Template

Template Element	Description
<b>Name</b>	Represents the name of the pattern.
<b>Intent</b>	Represents the intent behind representing the pattern.
<b>Applicability</b>	Represents the case where this pattern is applicable.
<b>Motivation Example</b>	Represents the main example that motivates this design pattern.
<b>Participants</b>	Represents the pattern participants and the role of every participant.

Table 4.1: Design Pattern Template

#### 4.1.2 Collaboration diagram

The collaboration among agents involved in every pattern is described in terms of UML sequence diagram (45). The participant agents are exchanging messages, characterized by FIPA performatives. These messages define every pattern protocol.

#### 4.1.3 Structure Diagram

The structure diagram (45) is an extension of the UML class diagram that provides a sufficient description of the agents involved in the pattern aiming at a JADE implementation. The agent is featured as a class stereotyped with <<Agent>>. Every agent performs one or more behaviours which are also featured as a class stereotyped with <<Behaviour>>. To represent that an agent can perform a behaviour, an association between that agent and its behaviour is used. The navigation is placed from the agent to the behaviour. The agents ability to establish a communication with other agents is represented by a dependency

between the communicating agents. Both the <<Agent>> and the <<Behaviour>> classes extend the JADE Agent and Behaviour classes, respectively. The <<Message>> stereotype in the <<Behaviour>> class indicates the FIPA performatives that an agent performing that behaviour may handle, i.e. the agent communication interface.

#### 4.1.4 Behaviour Diagram

The behavioural diagram (45) is an extension of the UML activity diagram. It shows activities which trigger message sending, activities performed after the reception of certain messages and the evaluation of conditions deciding which activity is going to be performed. This diagram describes the whole behaviour of the pattern, capturing all the possible activities to be performed by agents involved in every pattern.

#### 4.1.5 Test Design Patterns

The first *test-case* should be the simple success case (pattern normal behaviour). To implement this *test case*, all we need is to write a *mock agent* that simulates this *successful scenario*. In order to implement a second *test case* which verifies an *exceptional scenario*, the mock agent test case designer can change the placeholders values in the *mock agent* class.

#### 4.1.6 Pattern Implementation

The JADE framework is used to define the code skeleton of every social pattern. The association between <<Agent>> class and <<Behaviour>> class in the structure diagram indicates that the <<Behaviour>> class is an inner class of the <<Agent>> class in the JADE code, Figure 4.1. In this case, the behaviour can only be performed by the agent which possesses its declaration. Notice that when a behaviour can be performed by an agent, this behaviour is added to the agent `setup()` method. In the Structure diagram, the dependency between the <<Agent>> classes indicates that an agent can establish a communication with another interacting agent. The information stereotyped with <<Message>> in the <<Behaviour>> class in the Structure diagram indicates the FIPA performatives that can be handled by the agent performing that behaviour. Thus, each performative present in the Behaviour class will generate an IF inside its `action()` method in the corresponding agent code. Methods, attributes, generalization, GUI classes and both the Agent and Behaviour classes are implemented similarly to object-oriented in Java (53).

```

public class Client extends Agent {
    protected void setup() { // Put agent initializations here
        this.addBehaviour( new RequestProvider(); // Add the behaviour }
    protected void takeDown() { // Put agent clean-up operations here }
    private class RequestProvider extends Behaviour { /** Inner class */
        public void action() {
            ACLMessage reply = myAgent.receive(mt);
            if (reply != null) {
                if (reply.getPerformative() == ACLMessage.INFORM ) { }
                else if (reply.getPerformative() == ACLMessage.REFUSE) { } }
            else { block(); } //end of action method
        public boolean done() { return true; }
    } // End of inner class RequestProvider
} // End of agent class

```

Figure 4.1: Sample Code for MatchmakerClientAgent class

It is important to note that the `YellowPage`, `ServicesDirectory`, or `Matchmaker` agent is already implemented by `jade.domain.DFService` library (the `YellowPage` is called `Directory Facilitator` in JADE). Thus, any agent implemented in JADE may use the yellow pages service to locate service providers. In our test design patterns, we don't mock the `DFService` agent as it is already part of the JADE library that is well tested.

We present two patterns as samples, one related to the pair patterns category (booking pattern), and one related to the mediation patterns category (broker pattern), the rest of patterns are depicted in Appendix A.

## 4.2 Booking Pattern

### 4.2.1 Design Pattern Template

Table 4.2 represents the Booking design pattern template (38).

Template Element	Description
Name	Booking Pattern
Intent	To let a client issues a request to book some resource from a service provider.
Applicability	Use when an agent (client) needs to book some resource from another agent (service provider).
Motivation Example	Consider an application of hotel room reservation, in which, each agent can play a customer role or a service provider role. The client <code>BookingClient</code> sends a reservation request <code>ReservationRequestSent</code> containing the characteristics (place, room, etc.) of the resource it wishes to obtain from service provider. The service provider may alternatively answer with a denial <code>SPRefusedExternal</code> , a waiting list proposal <code>SPWLProposed</code> or an approval <code>SPResourceProposed</code> , i.e., a resource proposal when there exists such a resource that satisfies the characteristics the client sent.





### 4.2.3 Structure Diagram

Figure 4.3 represents the structure diagram of the Booking pattern.

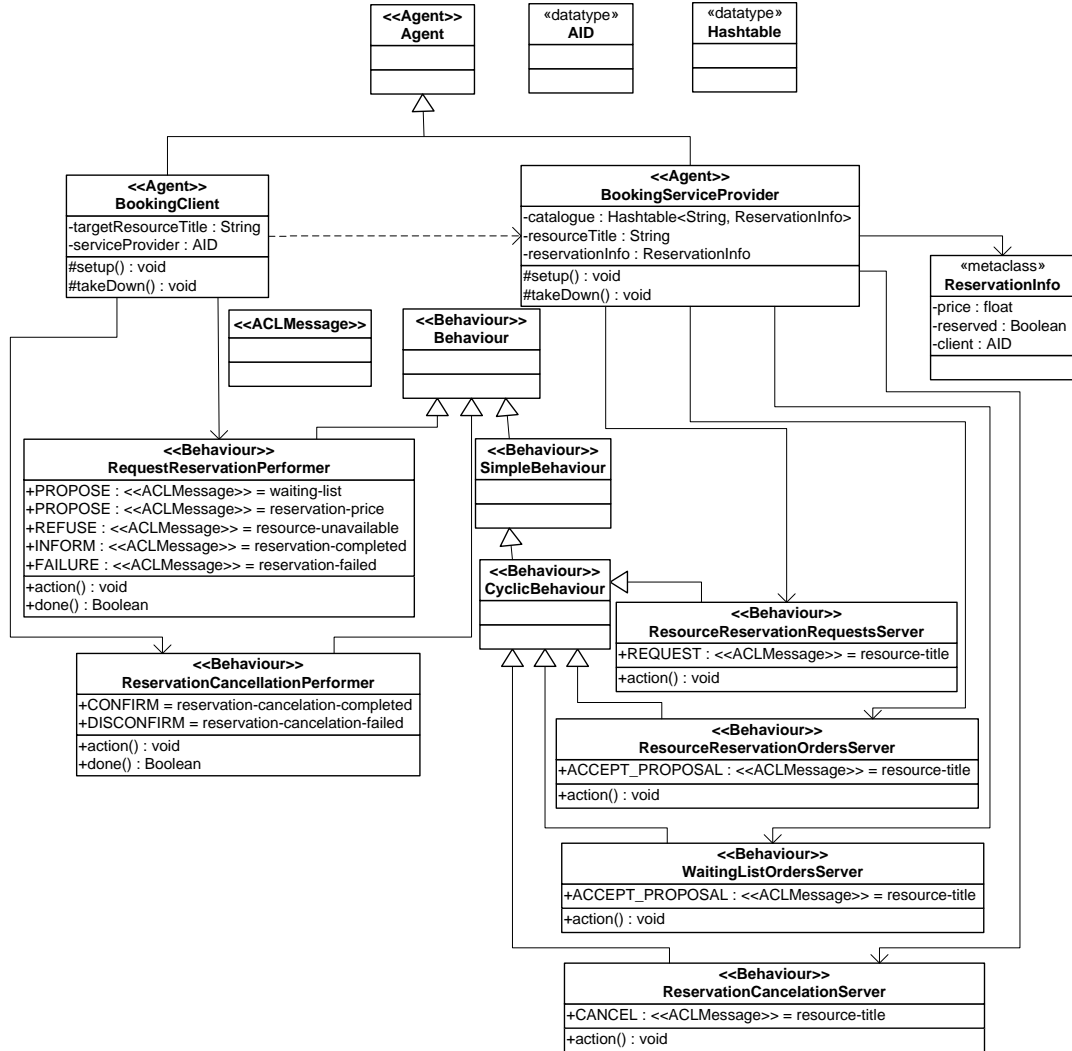


Figure 4.3: The Structure diagram of the Booking pattern

#### 4.2.4 Behaviour Diagram

Figure 4.4 represents the behaviour diagram of the Booking pattern.

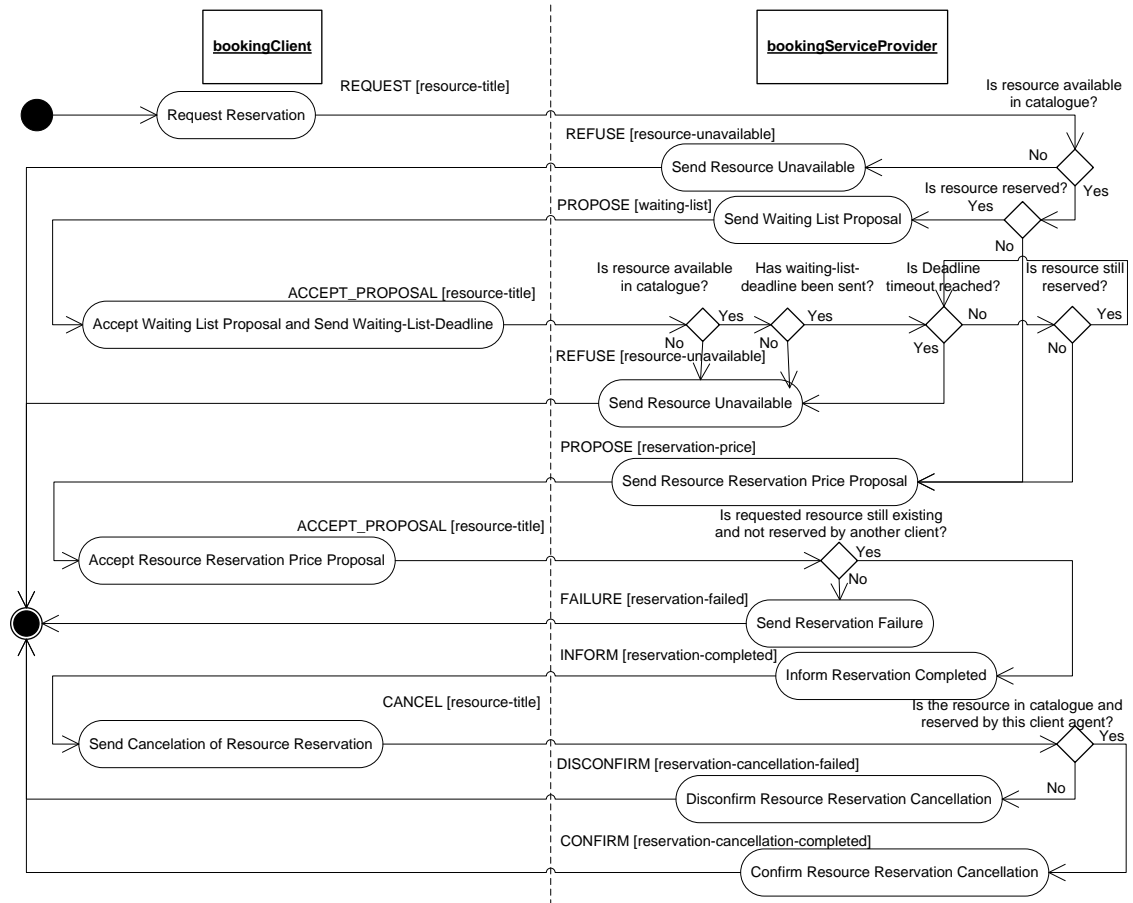


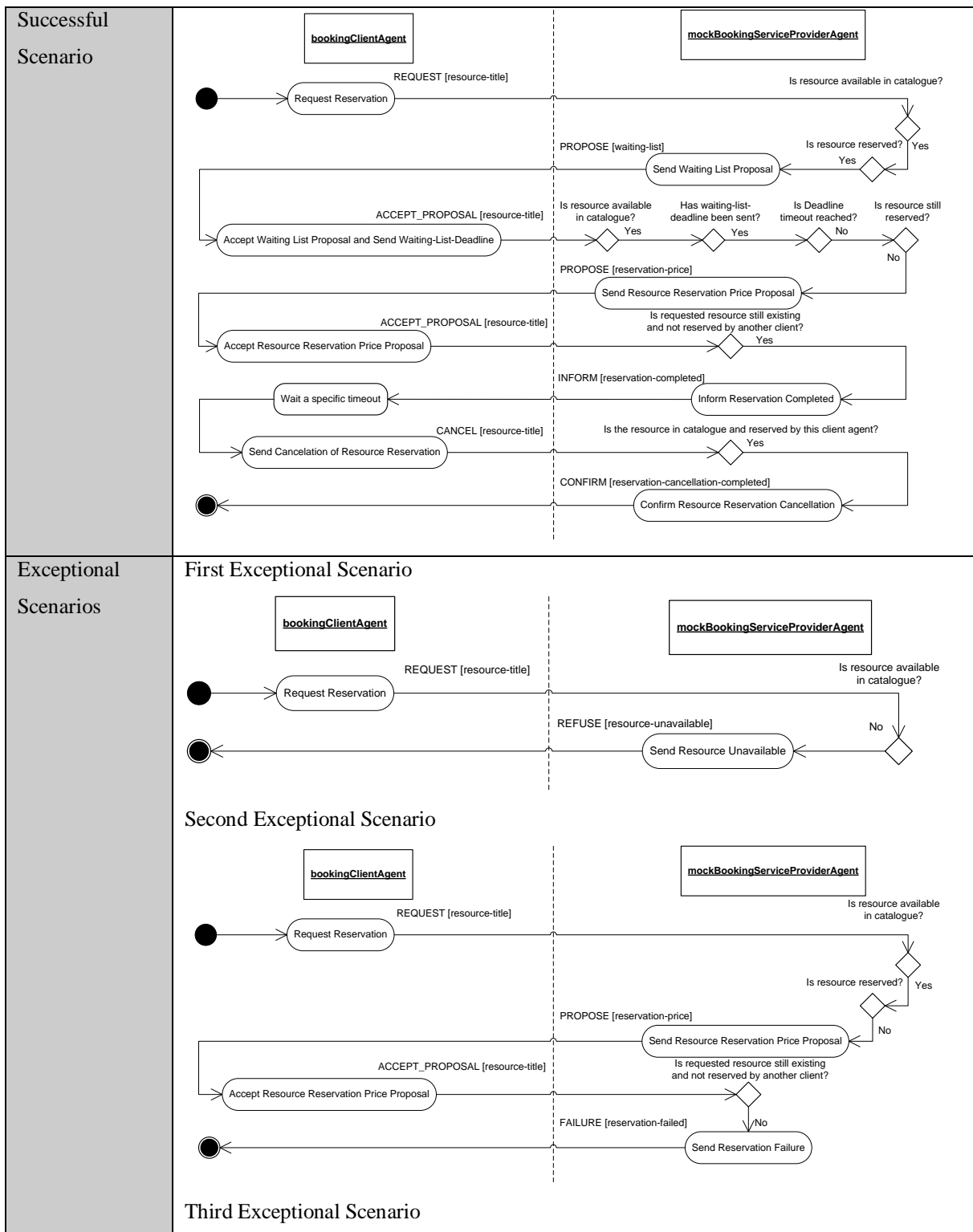
Figure 4.4: The Behaviour diagram of the Booking pattern

#### 4.2.5 Test Design Patterns

##### 4.2.5.1 BookingClientAgent

Table 4.3, represents the design of test design pattern of the Agent Under Test.

Agent	BookingClientAgent
Roles	BookingClient
Interacting Roles	ServiceProvider



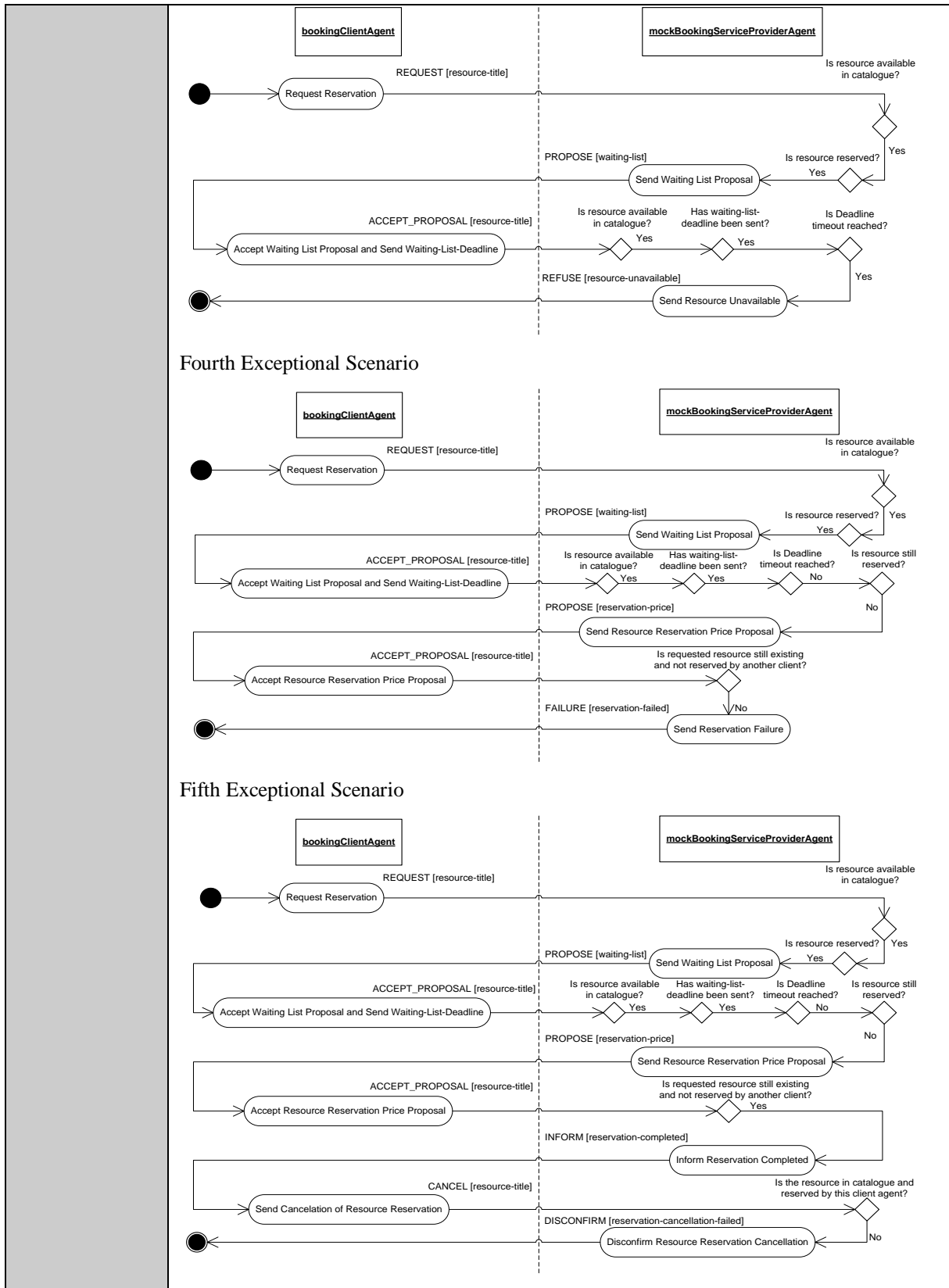


Table 4.3: Test Design Pattern of the ClientAgent

When the `mockBookingServiceProviderAgent` receives a resource reservation request message from the `bookingClientAgent`, it replies with a message characterized by either:

- i. A PROPOSE performative, indicating not only that the resource is available but also the resource reservation price, in this case the `bookingClientAgent` will send offer acceptance (i.e., reservation order) to the `mockBookingServiceProviderAgent` (normal scenario).
- ii. A PROPOSE performative, indicating that the resource is currently reserved by another client agent, in this case the `bookingClientAgent` will send waiting-list acceptance to the `mockBookingServiceProviderAgent` (normal scenario).
- iii. A REFUSE performative, indicating that the resource is unavailable, in this case the `bookingClientAgent` will terminate (First exp scenario).
- iv. No RESPONSE, indicating that the `mockBookingServiceProviderAgent` does not respond to the request message, in this case the `RequestReservationPerformer` behaviour will be blocked and is put into the blocked behaviours queue by the agent scheduler. When a new message (PROPOSE/REFUSE) is inserted in the agent message queue, the blocked behaviour becomes available for execution again, so that it has a chance to process the received message.

When the `mockBookingServiceProviderAgent` receives the offer acceptance message from the `bookingClientAgent`, it replies with a message characterized by either:

- i. An INFORM performative, indicating that the resource reservation is done successfully, in this case the `bookingClientAgent` reserves the resource for the requested time period (normal scenario).
- ii. A FAILURE performative, indicating that the requested resource, for any reason, is no more available in the catalogue or has been reserved to another client in the meanwhile,

in this case the `bookingClientAgent` will terminate (second and fourth exceptional scenarios).

- iii. No RESPONSE, indicating that the `mockBookingServiceProviderAgent` does not respond to the offer acceptance message, in this case the `RequestReservationPerformer` behaviour will be blocked and is put into the blocked behaviours queue by the agent scheduler. When a new message (INFORM/FAILURE) is inserted in the agent message queue, the blocked behaviour becomes available for execution again, so that it has a chance to process the received message.

When the `mockBookingServiceProviderAgent` receives the waiting-list acceptance message from the `bookingClientAgent`, it replies with a message characterized by either:

- i. A PROPOSE performative, indicating that the resource becomes unreserved and also the resource reservation price, in this case the `bookingClientAgent` will send offer acceptance (i.e., reservation order) to the `mockBookingServiceProviderAgent` (normal scenario).
- ii. A REFUSE performative, indicating that the requested resource, for any reason, is no more available in the catalogue or still being reserved to another client reaching the waiting-list timeout, in this case the `bookingClientAgent` will terminate (third exceptional scenario).
- iii. No RESPONSE, indicating that the `mockBookingServiceProviderAgent` does not respond to the waiting-list acceptance message, in this case the `RequestReservationPerformer` behaviour will be blocked and is put into the blocked behaviours queue by the agent scheduler. When a new message (PROPOSE/REFUSE) is inserted in the agent message queue, the blocked behaviour becomes available for execution again, so that it has a chance to process the received message.

When the `mockBookingServiceProviderAgent` receives the resource reservation cancelation message from the `bookingClientAgent`, it replies with a message characterized by either:

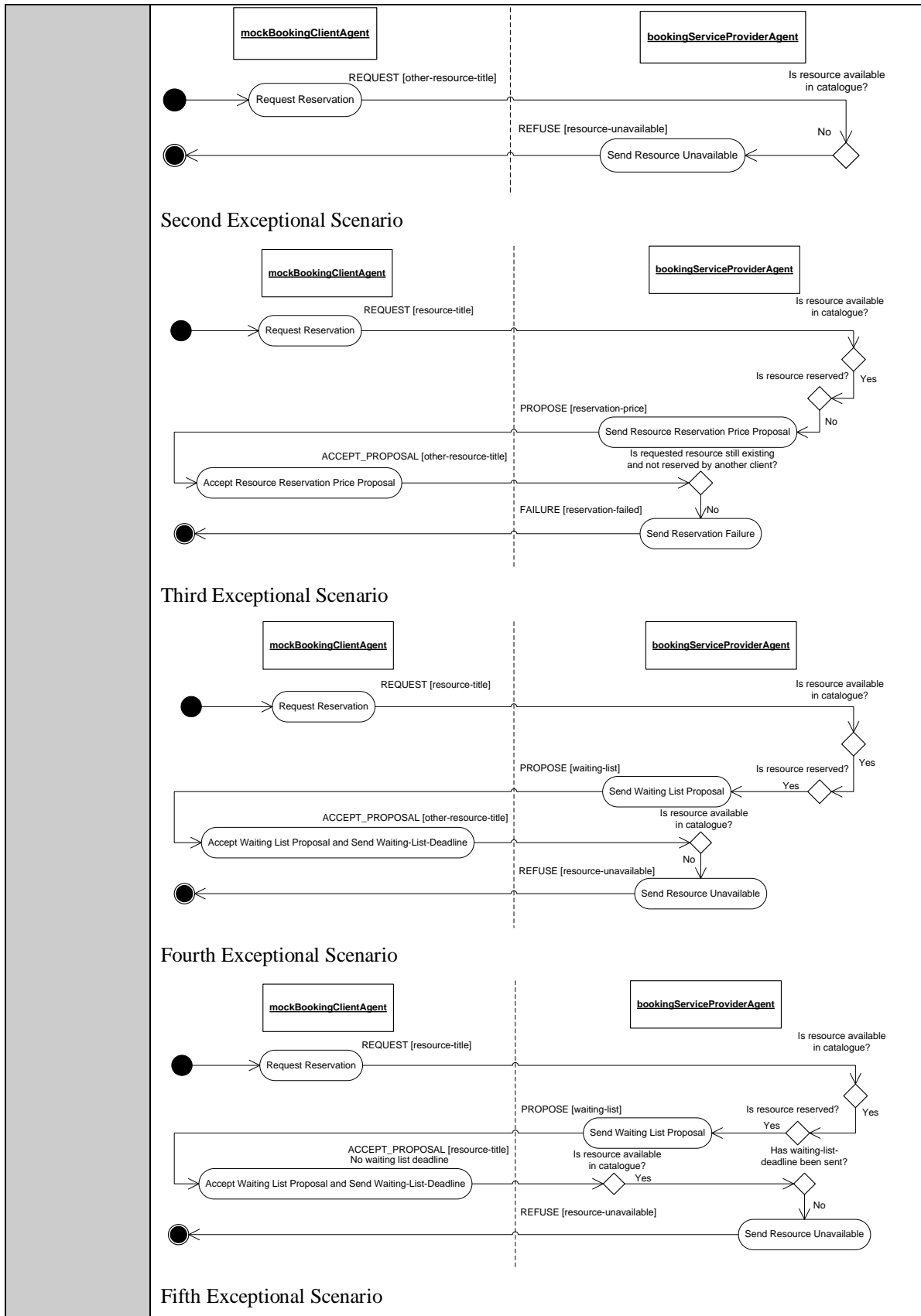
- i. A **CONFIRM** performative, indicating that the resource is in the catalogue and reserved by the `bookingClientAgent`, in this case the reservation is successfully canceled and the `bookingClientAgent` can terminate (normal scenario).
- ii. A **DISCONFIRM** performative, indicating that the resource is unreserved by `bookingClientAgent`, in this case the `bookingClientAgent` will terminate (fifth exceptional scenario).
- iii. No **RESPONSE**, indicating that the `mockBookingServiceProviderAgent` does not respond to the resource reservation cancelation message, in this case the `RequestReservationPerformer` behaviour will be blocked and is put into the blocked behaviours queue by the agent scheduler. When a new message (**CONFIRM/DISCONFIRM**) is inserted in the agent message queue, the blocked behaviour becomes available for execution again, so that it has a chance to process the received message.

#### 4.2.5.2 BookingServiceProviderAgent

Table 4.4, represents the design of test design pattern of the Agent Under Test.

Agent	<code>BookingServiceProviderAgent</code>
Roles	<code>BookingServiceProvider</code>
Interacting Roles	<code>BookingClient</code>
Successful Scenario	Is the same as the "successful scenario" depicted in the previous section with swapping the AUT and mock agent roles.
Exceptional Scenario	First Exceptional Scenario





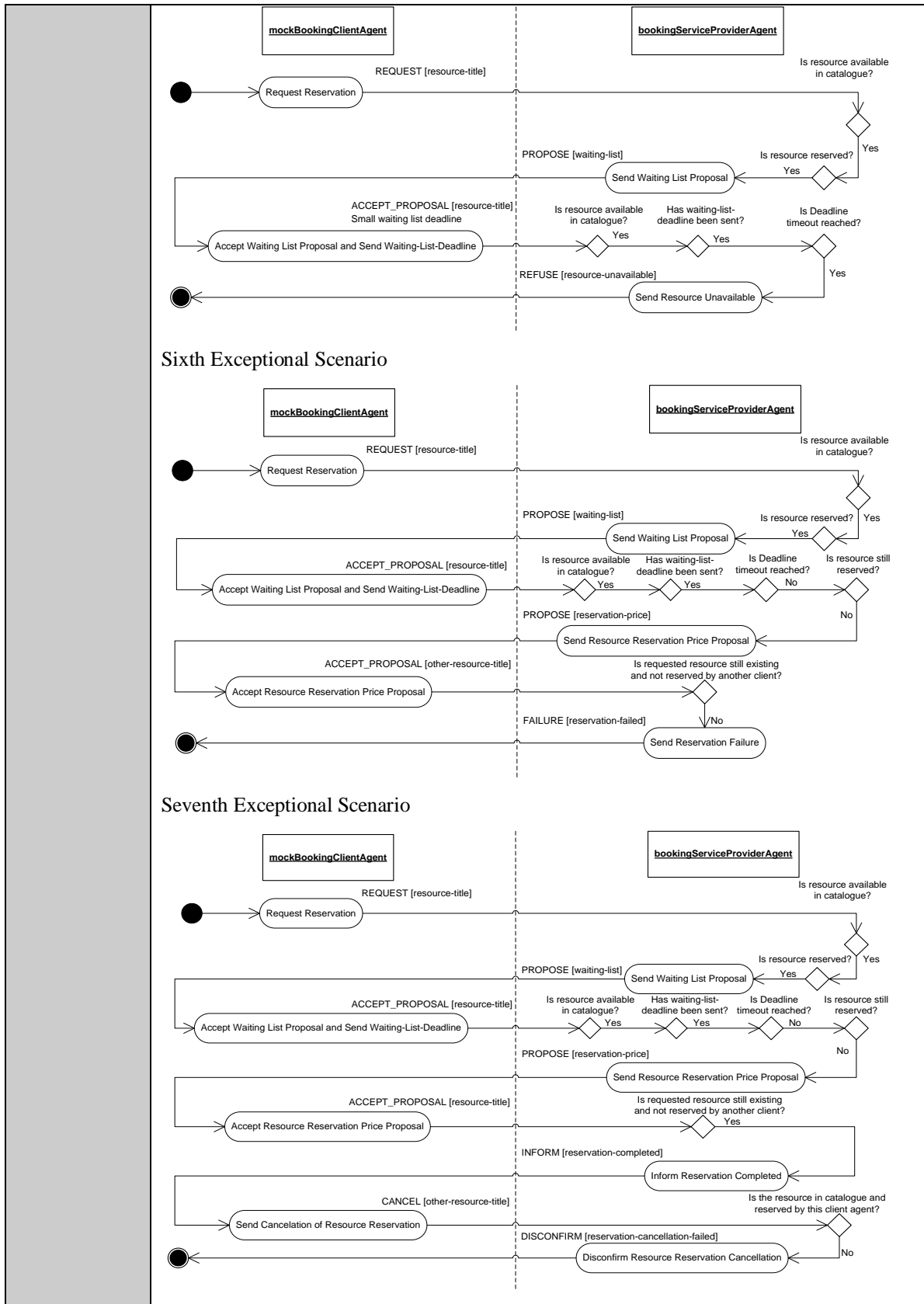


Table 4.4: Test Design Pattern of the ServiceProviderAgent

When the `mockBookingClientAgent` joins the environment, it sends a message characterized by a `REQUEST` performative indicating the willingness to book a specific resource from the `bookingServiceProviderAgent`. In this case, if the resource is unavailable, the `bookingServiceProviderAgent` replies with a `REFUSE` message (first exceptional scenario).

If the `mockBookingClientAgent` does not send a request message, the `ResourceReservationRequestsServer` behaviour will be blocked and is put into the blocked behaviours queue by the agent scheduler. When a new message (`REQUEST`) is inserted in the agent message queue, the blocked behaviour becomes available for execution again, so that it has a chance to process the received message.

When the `mockBookingClientAgent` receives a reservation proposal from the `bookingServiceProviderAgent`, it sends a message characterized by an `ACCEPT_PROPOSAL` performative indicating offer acceptance (i.e. reservation order). In this case, if the resource in the offer acceptance differs than the one in the original reservation request or the resource had been reserved by another client in the meanwhile, the `bookingServiceProviderAgent` replies with a `FAILURE` message (second and sixth exceptional scenarios).

If the `mockBookingClientAgent` does not send an accept proposal message, the `ResourceReservationOrdersServer` behaviour will be blocked and is put into the blocked behaviours queue by the agent scheduler. When a new message (`ACCEPT_PROPOSAL`) is inserted in the agent message queue, the blocked behaviour becomes available for execution again, so that it has a chance to process the received message.

When the `mockBookingClientAgent` receives a waiting-list proposal from the `bookingServiceProviderAgent`, it sends a message characterized by an `ACCEPT_PROPOSAL` performative indicating waiting-list acceptance. In this case, if the resource in the waiting-list acceptance is unavailable in catalogue or the waiting-list deadline

has not been sent or the deadline time-out is reached and the resource is still reserved, the `bookingServiceProviderAgent` replies with a REFUSE message (third, fourth and fifth exceptional scenarios).

If the `mockBookingClientAgent` does not send a waiting-list accept proposal message, the `WaitingListOrdersServer` behaviour will be blocked and is put into the blocked behaviours queue by the agent scheduler. When a new message (waiting-list ACCEPT\_PROPOSAL) is inserted in the agent message queue, the blocked behaviour becomes available for execution again, so that it has a chance to process the received message.

When the `mockBookingClientAgent` finishes the reservation time period, it sends a message characterized by a CANCEL performative indicating the willingness to cancel resource reservation from the `bookingServiceProviderAgent`. In this case, if the resource is unreserved by the `mockBookingClientAgent`, the `bookingServiceProviderAgent` replies with a DISCONFIRM message (seventh exceptional scenario).

If the `mockBookingClientAgent` does not send a reservation cancelation message, the `ReservationCancelationServer` behaviour will be blocked and is put into the blocked behaviours queue by the agent scheduler. When a new message (CANCEL) is inserted in the agent message queue, the blocked behaviour becomes available for execution again, so that it has a chance to process the received message.

#### **4.2.6 Pattern Implementation**

##### **4.2.6.1 BookingClientAgent**

The mock agent designer uses our Eclipse Plug-in in order to generate the code of the mock agent `MockBookingServiceProviderAgent` that tests the agent role `BookingClientAgent` which follows the design pattern `BookingPattern` that belongs to the pattern category `PairPatterns`.

The following files are generated:

- The mock agent file `MockBookingServiceProviderAgent.java` and its associated resource file `MockBookingServiceProviderAgent.properties`
- The client test case file `BookingClientAgentTestCase.java`
- The agent under test file `BookingClientAgent.java`

The mock agent `MockBookingServiceProviderAgent` has a plan (test scenario) represented by four JADE Behaviors called `ResourceReservationRequestsServer`, `ResourceReservationOrdersServer`, `WaitingListOrdersServer` and `ReservationCancellationServer` to test the AUT agent `BookingClientAgent`. To execute the AUT agent test case `BookingClientAgentTestCase`, all we need is to create a subclass of `JADETestCase` and to implement a test method `testBookingClient()` that creates an instance of the AUT `BookingClientAgent`, and an instance of the mock agent `MockBookingServiceProviderAgent`. After that, this test method waits until their interaction finishes and asks the mock agent about the test result.

#### 4.2.6.2 BookingServiceProviderAgent

The mock agent designer uses our Eclipse Plug-in in order to generate the code of the mock agent `MockBookingClientAgent` that tests the agent role `BookingServiceProviderAgent` which follows the design pattern `BookingPattern` that belongs to the pattern category `PairPatterns`.

In this case, the following files are generated:

- The mock agent file `MockBookingClientAgent.java` and its associated resource file `MockBookingClientAgent.properties`
- The service provider test case file `BookingServiceProviderAgentTestCase.java`
- The agent under test file `BookingServiceProviderAgent.java`

The mock agent `MockBookingClientAgent` has a plan (test scenario) represented by two JADE Behavior called `RequestReservationPerformer` and `ReservationCancellationPerformer` to test the AUT agent

BookingServiceProviderAgent. To execute the AUT agent test case BookingServiceProviderAgentTestCase, all we need is to create a subclass of JADETestCase and to implement a test method testBookingServiceProvider() that creates an instance of the AUT BookingServiceProviderAgent, and an instance of the mock agent MockBookingClientAgent. After that, this test method waits until their interaction finishes and asks the mock agent about the test result.

## 4.3 Broker Pattern

### 4.3.1 Design Pattern Template

Table 4.5 represents the Broker design pattern template.

Template Element	Description
<b>Name</b>	Broker Pattern
<b>Intent</b>	The broker allows decoupling of the client and service-provider by accepting requests from a client, farming out the work to a willing and available service-provider, and returning results to the client. This allows for communication and location transparency for interoperating applications.
<b>Applicability</b>	If many clients and many service-providers exist in a particular application, this pattern is applicable. If distributed heterogeneous components are to be integrated, but their independence is to be maintained, this pattern is applicable. Distributed, heterogeneous components may not have been designed to interoperate: they may be dispersed over multiple platforms and may be implemented in different languages.
<b>Motivation Example</b>	<p>In multi-agent systems involving numerous agents with a range of capabilities, it is not feasible for each client to hold capability models for each service-provider. This would result in a complete graph if agents held both roles of client and service-provider, and complex graphs for lesser cases, with consequent messaging overhead in maintaining the capability models. It is more efficient for a broker or a number of brokers to serve as go-betweens or matchmakers, maintaining capability models of service-providers and connecting clients to service-providers providing their needs.</p> <p>Consider an application of E-Business Broker, in which, each agent can play a customer role or a service provider role. The client <code>customer1</code> sends a service request <code>ServiceRequestSent</code> containing the characteristics of the service it wishes to obtain from the broker. The broker may alternatively answer with a denial <code>BRRefusalSent</code> or an acceptance <code>BRAcceptanceSent</code>. In the case of an acceptance, the broker sends a</p>



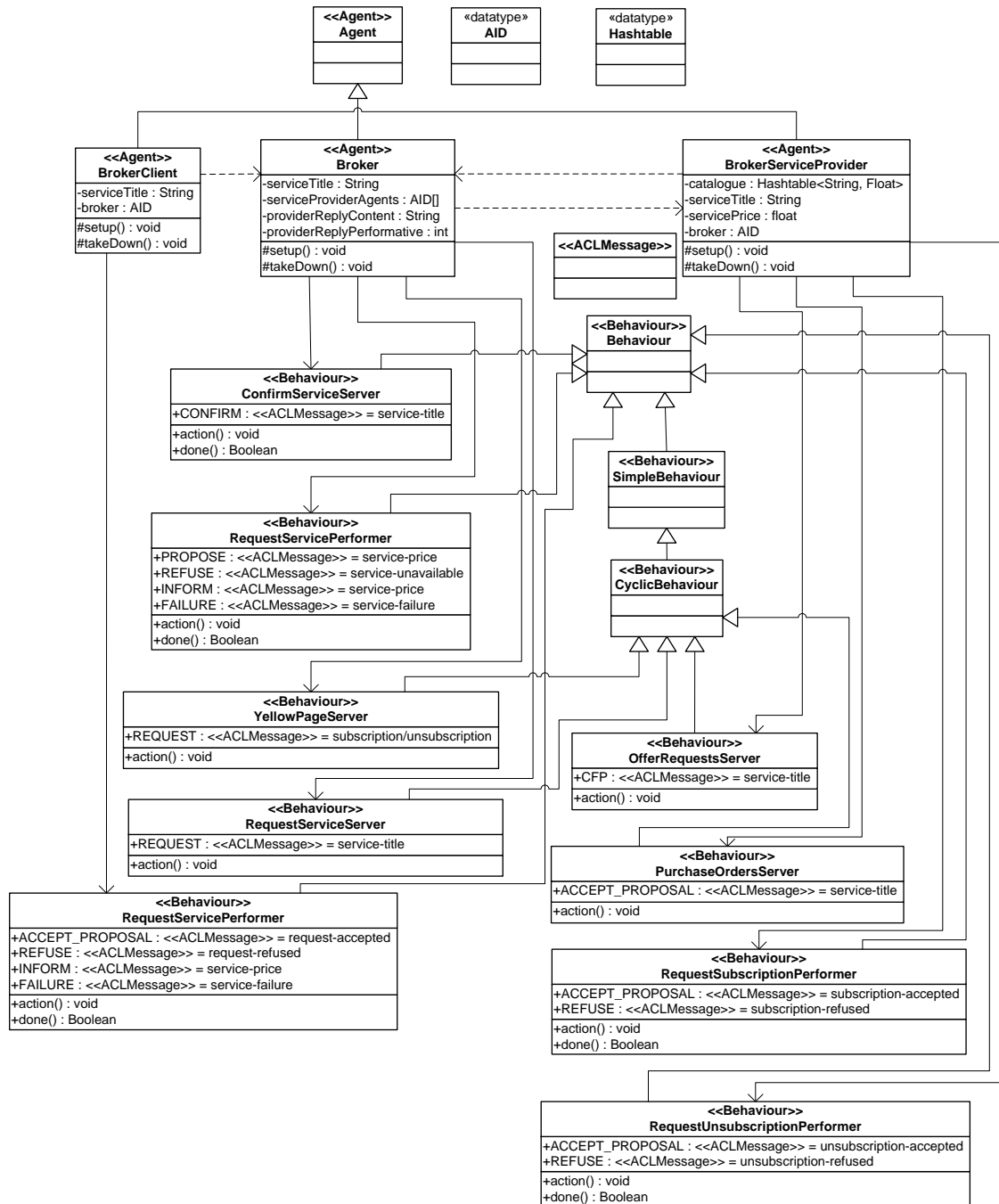


Figure 4.6: The Structure diagram of the Broker pattern

#### 4.3.4 Behaviour Diagram

Figure 4.7 represents the behaviour diagram of the Broker pattern.





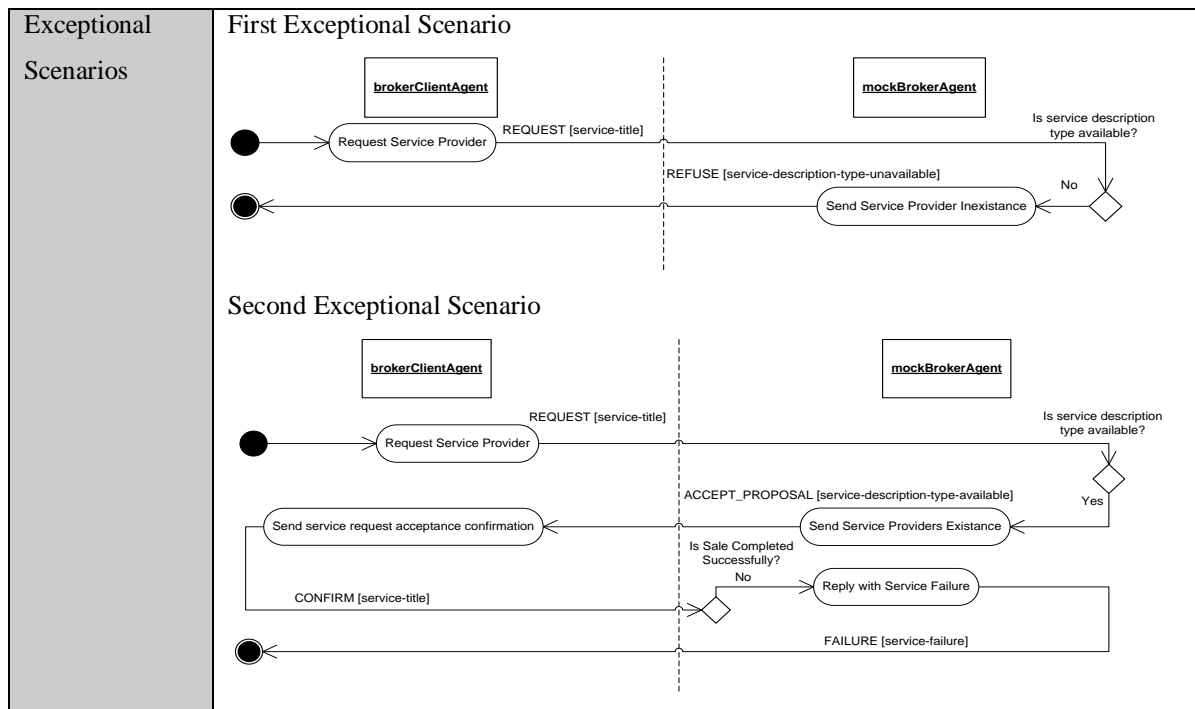


Table 4.6: Test Design Pattern of the BrokerClientAgent

When the mockBrokerAgent receives the "service-providing" request message from the brokerClientAgent, it replies with a message characterized by either:

- An ACCEPT\_PROPOSAL performative, indicating that the service description type is available (normal scenario).
- A REFUSE message, if the TickerBehaviour time period is not reasonable enough, the brokerClientAgent will either found no serviceProviderAgents registered with the mockBrokerAgent yet, or it will only deal with a small sample of serviceProviderAgents that may not represent the service real market price or even may not have the service available (first exceptional scenario).
- No RESPONSE, indicating that the mockBrokerAgent does not respond to the "service-providing" request message, in this case the brokerClientAgent will wait the given period specified in the constructor of the TickerBehaviour in order to re-request from the mockBrokerAgent in case any agents had registered as "service-providing" in the meanwhile.

For the service reply in normal and exceptional scenarios of the `mockBrokerAgent` testing the `brokerClientAgent`, please review the call for proposal pattern Appendix A.2.5.1 BuyerAgent (normal and third exceptional scenarios).

#### 4.3.5.2 BrokerAgent

Table 4.7, represents the design of test design pattern of the Agent Under Test.

Agent	BrokerAgent
Roles	Broker
Interacting Roles	BrokerClient, BrokerServiceProvider
Successful Scenario	<p><b>MockBrokerClientAgent</b></p> <p><b>MockBrokerServiceProviderAgent</b></p>
Exceptional Scenarios	MockBrokerClientAgent

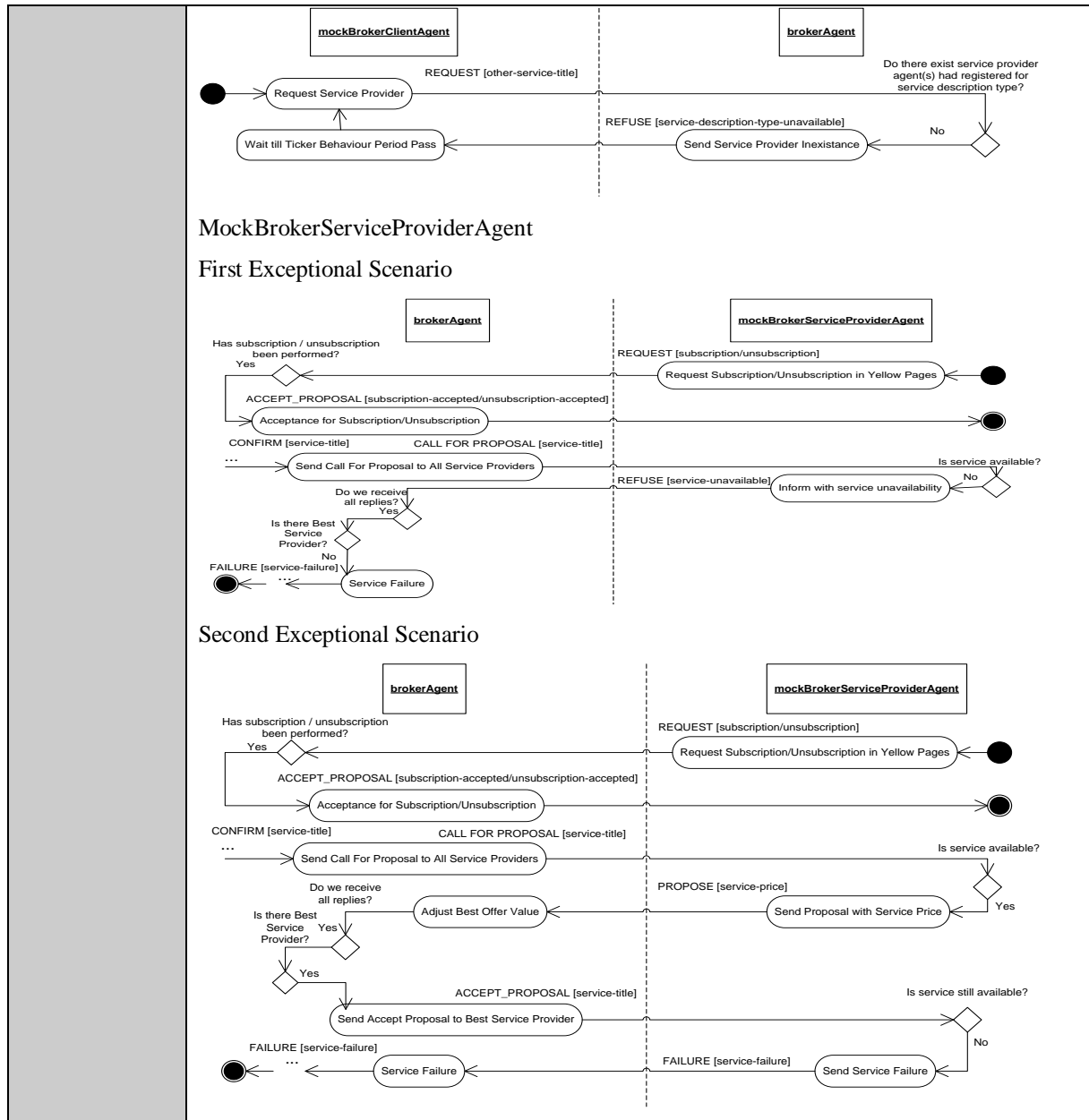


Table 4.7: Test Design Pattern of the BrokerAgent

When the `mockBrokerClientAgent` joins the environment, it sends a message characterized by a `REQUEST` performative indicating the willingness to be informed by a specific service price from the `brokerAgent`. In this case, if the service is unavailable, the `brokerAgent` replies with a `REFUSE` message (`MockBrokerClientAgent` exceptional scenario).

If the `mockBrokerClientAgent` does not send a request message, the `RequestServiceServer` behaviour will be blocked and is put into the blocked behaviours queue by the agent scheduler. When a new message (REQUEST) is inserted in the agent message queue, the blocked behaviour becomes available for execution again, so that it has a chance to process the received message.

For the subscription/unsubscription normal and exceptional scenarios of the `mockBrokerServiceProviderAgent` testing `brokerAgent`, please review the subscription pattern Appendix A.1.5.1 `SubscriptionYellowPageAgent`.

For the call for proposal normal and exceptional scenarios, please review the call for proposal pattern Appendix A.2.5.1 `BuyerAgent` (normal, second and third exceptional scenarios).

#### 4.3.5.3 BrokerServiceProviderAgent

Table 4.8, represents the design of test design pattern of the Agent Under Test.

Agent	BrokerServiceProviderAgent
Roles	BrokerServiceProvider
Interacting Roles	Broker
Successful Scenario	<pre> sequenceDiagram     participant mockBrokerAgent     participant brokerServiceProvderAgent     mockBrokerAgent-&gt;&gt;brokerServiceProvderAgent: REQUEST [subscription/unsubscription]     brokerServiceProvderAgent-&gt;&gt;BlackCircle1(( )): Request Subscription/Unsubscription in Yellow Pages     mockBrokerAgent-&gt;&gt;brokerServiceProvderAgent: ACCEPT_PROPOSAL [subscription-accepted/unsubscription-accepted]     brokerServiceProvderAgent-&gt;&gt;BlackCircle2(( )): Acceptance for Subscription/Unsubscription     mockBrokerAgent-&gt;&gt;brokerServiceProvderAgent: CALL FOR PROPOSAL [service-title]     brokerServiceProvderAgent-&gt;&gt;BlackCircle3(( )): Send Call For Proposal to All Service Providers     mockBrokerAgent-&gt;&gt;brokerServiceProvderAgent: Do we receive all replies?     brokerServiceProvderAgent-&gt;&gt;BlackCircle4(( )): Adjust Best Offer Value     mockBrokerAgent-&gt;&gt;brokerServiceProvderAgent: Is there Best Service Provider?     brokerServiceProvderAgent-&gt;&gt;BlackCircle5(( )): Send Accept Proposal to Best Service Provider     mockBrokerAgent-&gt;&gt;brokerServiceProvderAgent: ACCEPT_PROPOSAL [service-title]     brokerServiceProvderAgent-&gt;&gt;BlackCircle6(( )): INFORM [service-price]     brokerServiceProvderAgent-&gt;&gt;BlackCircle6(( )): Remove service from catalogue and Inform Sale Completed     </pre>
Exceptional Scenarios	First Exceptional Scenario

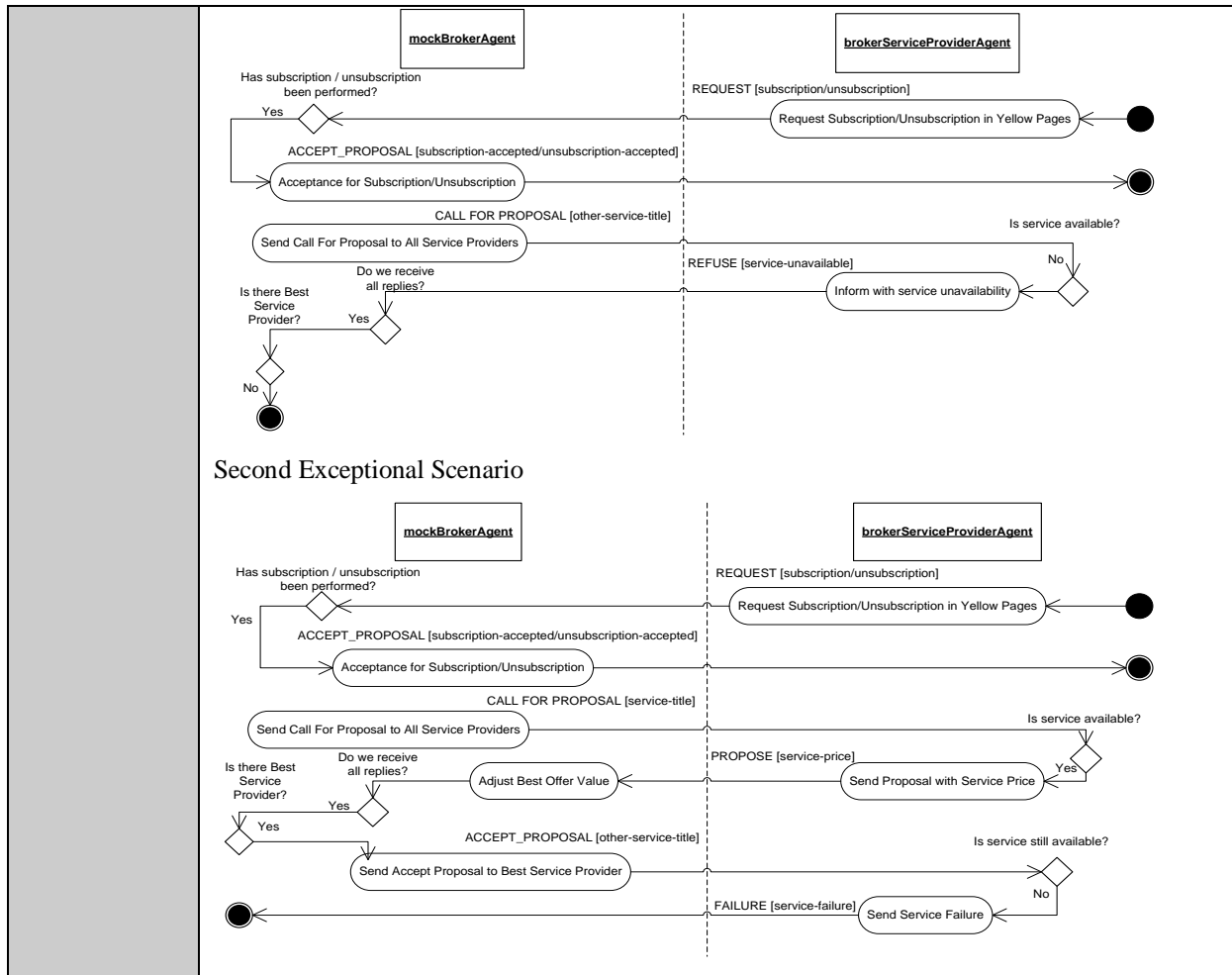


Table 4.8: Test Design Pattern of the BrokerServiceProvidingAgent

For the subscription/unsubscription normal and exceptional scenarios of the `mockBrokerAgent` testing the `brokerServiceProvidingAgent`, please review the subscription pattern Appendix A.1.5.2 SubscriptionServiceProvidingAgent.

For the call for proposal normal and exceptional scenarios, please review the call for proposal pattern Appendix A.2.5.2 SellerAgent (normal, First and Second exceptional scenarios).

### 4.3.6 Pattern Implementation

#### 4.3.6.1 BrokerClientAgent

The mock agent designer uses our Eclipse Plug-in in order to generate the code of the mock agent `MockBrokerAgent` that tests the agent role `BrokerClientAgent` which follows the design pattern `BrokerPattern` that belongs to the pattern category `MediationPatterns`.

In this case, the following files are generated:

- The mock agent file `MockBrokerAgent.java` and its associated resource file `MockBrokerAgent.properties`
- The client test case file `BrokerClientAgentTestCase.java`
- The agent under test file `BrokerClientAgent.java`

The mock agent `MockBrokerAgent` has a plan (test scenario) represented by two JADE Behaviors called `RequestServiceServer` and `ConfirmServiceServer` to test the AUT agent `BrokerClientAgent`. To execute the AUT agent test case `BrokerClientAgentTestCase`, all we need is to create a subclass of `JADETestCase` and to implement a test method `testBrokerClient()` that creates an instance of AUT `BrokerClientAgent`, and an instance of the mock agent `MockBrokerAgent`. After that, this test method waits until their interaction finishes and asks the mock agent about the test result.

#### **4.3.6.2 BrokerAgent**

The mock agent designer uses our Eclipse Plug-in in order to generate the code of the mock agent `MockBrokerClientAgent` that tests the agent role `BrokerAgent` which follows the design pattern `BrokerPattern` that belongs to the pattern category `MediationPatterns`.

In this case, the following files are generated:

- The mock agent file `MockBrokerClientAgent.java` and its associated resource file `MockBrokerClientAgent.properties`
- The broker test case file `BrokerAgentTestCase.java`
- The agent under test file `BrokerAgent.java`

The mock agent `MockBrokerClientAgent` has a plan (test scenario) represented by one JADE Behavior called `RequestServicePerformer` to test the AUT agent `BrokerAgent`. To execute the AUT agent test case `BrokerAgentTestCase`, all we need is to create a subclass of `JADETestCase` and to implement a test method `testBrokering()` that creates

an instance of the AUT `BrokerAgent`, and an instance of the mock agent `MockBrokerClientAgent`. After that, this test method waits until their interaction finishes and asks the mock agent about the test result.

The mock agent designer uses our Eclipse Plug-in in order to generate the code of the mock agent `MockBrokerServiceProviderAgent` that tests the agent role `BrokerAgent` which follows the design pattern `BrokerPattern` that belongs to the pattern category `MediationPatterns`.

In this case, the following files are generated:

- The mock agent file `MockBrokerServiceProviderAgent.java` and its associated resource file `MockBrokerServiceProviderAgent.properties`
- The broker test case file `BrokerAgentTestCase.java`
- The agent under test file `BrokerAgent.java`

The mock agent `MockBrokerServiceProviderAgent` has a plan (test scenario) represented by four JADE Behaviors called `RequestSubscriptionPerformer`, `OfferRequestsServer`, `PurchaseOrdersServer` and `RequestUnsubscriptionPerformer` to test the AUT agent `BrokerAgent`. To execute the AUT agent test case `BrokerAgentTestCase`, all we need is to create a subclass of `JADETestCase` and to implement a test method `testBrokering()` that creates an instance of the AUT `BrokerAgent`, and an instance of the mock agent `MockBrokerServiceProviderAgent`. After that, this test method waits until their interaction finishes and asks the mock agent about the test result.

#### **4.3.6.3 BrokerServiceProviderAgent**

The mock agent designer uses our Eclipse Plug-in in order to generate the code of the mock agent `MockBrokerAgent` that tests the agent role `BrokerServiceProviderAgent` which follows the design pattern `BrokerPattern` that belongs to the pattern category `MediationPatterns`.



In this case, the following files are generated:

- The mock agent file `MockBrokerAgent.java` and its associated resource file `MockBrokerAgent.properties`
- The service provider test case file `BrokerServiceProviderAgentTestCase.java`
- The agent under test file `BrokerServiceProviderAgent.java`

The mock agent `MockBrokerAgent` has a plan (test scenario) represented by two JADE Behaviors called `YellowPageServer` and `RequestServicePerformer` to test the AUT agent `BrokerServiceProviderAgent`. To execute the AUT agent test case `BrokerServiceProviderAgentTestCase`, all we need is to create a subclass of `JADETestCase` and to implement a test method `testBrokerServiceProvider()` that creates an instance of the AUT `BrokerServiceProviderAgent`, and an instance of the mock agent `MockBrokerAgent`. After that, this test method waits until their interaction finishes and asks the mock agent about the test result.

#### **4.4 Conclusion**

This chapter presents the design of the test design patterns. A set of the most common Agent Design Patterns are presented with their test design patterns for the different roles played by every agent in each design pattern. The test cases implementation of each design pattern is also provided.

## CHAPTER 5: EVALUATION

### 5.1 Introduction

In this section, we analyze the quantitative results by using a code coverage tool to proof that the generated test files (mock agent, associated resource file, and AUT test case) from our Eclipse Plug-in totally covers the AUT code. EMMA (55) is a tool for measuring *coverage* of Java software. Such a tool is essential for detecting dead code and verifying which parts of the application are actually exercised by the test suite and interactive use. *Code coverage* refers to a software engineering technique whereby the mock agent test case designer tracks quality and comprehensiveness of his/her test suite by determining simple metrics like the percentage of {classes, methods, lines, etc} that got executed when the test suite ran. Emma makes statistics about the code and registers how much time each method, line of code, will be covered during a specific scenario.

EMMA's fundamental units of coverage are *basic blocks* (all other types of coverage are derived from basic block coverage in one way or another). A basic block is a sequence of bytecode instructions without any jumps or jump targets. In other words, a basic block always (in the absense of exceptions) executes as one atomic unit (if it is entered at all). Because several Java source lines can be in the same basic block, for efficiency reasons at execution time it makes more sense to keep track of basic blocks rather than individual lines. Furthermore, a given Java source file contains a large number of lines that are not executable (they do not map to any "real" bytecode), such as comments, import statements, field declarations, method signatures, etc.

### 5.2 EMMA Code Coverage Results

Table 5.1 represents the code coverage results obtained from running the EMMA code coverage tool by changing the resource files placeholder values in the ten *social design patterns* we initially support.

Pair Patterns	class, %	method, %	block, %	line, %	class, %	method, %	block, %	line, %	class, %	method, %	block, %	line, %
Pattern	AUT				Mock Agent				Test Case			

Name												
Booking Pattern	Booking Client				Mock Booking Service Provider				Booking Client Test Case			
	100% (4/4)	100% (15/15)	99% (518/523)	98% (107/109)	100% (6/6)	100% (25/25)	93% (628/676)	91% (118/130)	100% (1/1)	83% (5/6)	81% (92/114)	77% (23/30)
	Booking Service Provider				Mock Booking Client				Booking Service Provider Test Case			
	100% (6/6)	100% (18/18)	99% (538/543)	98% (114/116)	100% (4/4)	100% (22/22)	97% (608/627)	95% (119/125)	100% (1/1)	83% (5/6)	81% (92/114)	76% (22/29)
Subscription Pattern	Subscription Yellow Page				Mock Subscription Service Provider				Subscription Yellow Page Test Case			
	100% (2/2)	100% (6/6)	100% (216/216)	100% (49/49)	100% (4/4)	100% (21/21)	96% (381/395)	95% (76/80)	100% (1/1)	75% (3/4)	89% (63/71)	83% (15/18)
	Subscription Service Provider				Mock Subscription Yellow Page				Subscription Service Provider Test Case			
	100% (4/4)	100% (14/14)	100% (318/318)	100% (67/67)	100% (2/2)	100% (13/13)	92% (262/286)	88% (53/60)	100% (1/1)	75% (3/4)	89% (63/71)	84% (16/19)
Call-For-Proposal Pattern	Buyer				Mock Seller				Buyer Test Case			
	100% (3/3)	100% (12/12)	97% (391/404)	95% (76.6/81)	100% (3/3)	100% (17/17)	90% (322/358)	83% (65/78)	100% (1/1)	83% (5/6)	80% (87/109)	76% (22/29)
	Seller				Mock Buyer				Seller Test Case			
	100% (3/3)	100% (10/10)	95% (262/275)	89% (59/66)	100% (3/3)	100% (19/19)	95% (464/491)	91% (83.6/92)	100% (1/1)	83% (5/6)	80% (87/109)	74% (20/27)
Bidding Pattern	Bidder				Mock Auctioneer				Bidder Test Case			
	100% (8/8)	100% (30/30)	98% (652/665)	95% (145/152)	100% (4/4)	100% (28/28)	96% (1361/1420)	96% (249.8/261)	100% (1/1)	83% (5/6)	83% (107/129)	79% (27/34)
	Auctioneer				Mock Bidder				Auctioneer Test Case			
	100% (4/4)	100% (21/21)	99% (1233/1246)	98% (237.8/242)	100% (8/8)	100% (37/37)	94% (778/832)	90% (153/170)	100% (1/1)	83% (5/6)	86% (130/152)	82% (32/39)

Mediation Patterns	class, %	method, %	block, %	line, %	class, %	method, %	block, %	line, %	class, %	method, %	block, %	line, %
Pattern Name	AUT				Mock Agent				Test Case			
Monitor Pattern	Notification Subscriber				Mock Monitor				Notification Subscriber Test Case			
	100% (4/4)	100% (15/15)	99% (344/349)	97% (70/72)	100% (3/3)	100% (23/23)	93% (294/317)	90% (54/60)	100% (1/1)	83% (5/6)	78% (77/99)	74% (20/27)
	Monitor				Mock Notification Subscriber				Monitor Test Case			
	100% (4/4)	100% (21/21)	100% (371/371)	100% (74/74)	100% (4/4)	100% (22/22)	90% (380/422)	89% (73.8/83)	100% (1/1)	83% (5/6)	81% (95/117)	78% (25/32)
					Mock Notification Source				Monitor Test Case			
					100% (2/2)	100% (14/14)	92% (217/236)	88% (42/48)	100% (1/1)	83% (5/6)	81% (96/118)	78% (25/32)
	Notification Source				Mock Monitor				Notification Source Test Case			
	100% (2/2)	100% (7/7)	97% (170/175)	95% (35/37)	100% (2/2)	100% (20/20)	96% (246/257)	96% (48/50)	100% (1/1)	83% (5/6)	79% (82/104)	75% (21/28)
Broker Pattern	Broker Client				Mock Broker				Broker Client Test Case			
	100% (3/3)	100% (11/11)	99% (355/360)	97% (66/68)	100% (3/3)	100% (22/22)	95% (282/296)	93% (53/57)	100% (1/1)	83% (5/6)	78% (77/99)	74% (20/27)
	Broker				Mock Broker Client				Broker Test Case			
	100% (5/5)	100% (25/25)	98% (752/764)	97% (160.6/165)	100% (3/3)	100% (18/18)	96% (418/437)	93% (75/81)	100% (1/1)	83% (5/6)	81% (95/117)	76% (22/29)
					Mock Broker Service Provider				Broker Test Case			
					100% (6/6)	100% (28/28)	97% (648/667)	95% (127/133)	100% (1/1)	83% (5/6)	81% (96/118)	77% (24/31)
	Broker Service Provider				Mock Broker				Broker Service Provider Test Case			
	100% (6/6)	100% (21/21)	99% (549/554)	98% (118/120)	100% (3/3)	100% (21/21)	94% (549/586)	94% (114.6/122)	100% (1/1)	83% (5/6)	79% (82/104)	75% (21/28)
Matchmaker Pattern	Matchmaker Client				Mock Matchmaker				Matchmaker Client Test Case			
	100% (2/2)	100% (9/9)	98% (366/375)	95% (70/74)	100% (2/2)	100% (17/17)	93% (204/219)	91% (39/43)	100% (1/1)	83% (5/6)	81% (96/118)	78% (25/32)
					Mock Matchmaker Provider				Matchmaker Client Test Case			
					100% (2/2)	100% (14/14)	91% (202/221)	87% (40/46)	100% (1/1)	83% (5/6)	82% (98/120)	78% (25/32)
	Matchmaker				Mock Matchmaker Client				Matchmaker Test Case			
	100% (3/3)	100% (13/13)	98% (377/385)	95% (82/86)	100% (2/2)	100% (16/16)	92% (266/289)	87% (53/61)	100% (1/1)	83% (5/6)	81% (95/117)	77% (24/31)

					Mock Matchmaker Provider				Matchmaker Test Case			
					100% (4/4)	100% (21/21)	97% (460/476)	96% (91/95)	100% (1/1)	83% (5/6)	81% (96/118)	77% (24/31)
	Matchmaker Provider				Mock Matchmaker				Matchmaker Provider Test Case			
	100% (5/5)	100% (18/18)	99% (464/469)	98% (96/98)	100% (2/2)	100% (13/13)	95% (272/286)	93% (56/60)	100% (1/1)	83% (5/6)	82% (97/119)	79% (26/33)
Mediator Pattern	Mediator Client				Mock Mediator				Mediator Client Test Case			
	100% (2/2)	100% (9/9)	96% (227/236)	91% (43/47)	100% (2/2)	100% (24/24)	92% (324/354)	87% (58/67)	100% (1/1)	83% (5/6)	82% (97/119)	77% (23/30)
	Mediator				Mock Mediator Client				Mediator Test Case			
	100% (6/6)	100% (34/34)	97% (874/905)	94% (166/176)	100% (2/2)	100% (16/16)	92% (269/292)	86% (51/59)	100% (1/1)	83% (5/6)	86% (140/162)	82% (33/40)
	Mock Mediator Provider				Mediator Test Case				Mediator Test Case			
	100% (5/5)	100% (26/26)	95% (539/567)	93% (107/115)	100% (1/1)	83% (5/6)	84% (117/139)	79% (27/34)				
	Mediator Provider				Mock Mediator				Mediator Provider Test Case			
	100% (5/5)	100% (19/19)	99% (466/471)	98% (97/99)	100% (4/4)	100% (21/21)	97% (525/544)	94% (98/104)	100% (1/1)	83% (5/6)	82% (102/124)	77% (24/31)
Embassy Pattern	Foreign				Mock Embassy				Foreign Test Case			
	100% (2/2)	100% (12/12)	99% (402/407)	97% (75/77)	100% (3/3)	100% (36/36)	97% (585/604)	95% (109/115)	100% (1/1)	83% (5/6)	85% (127/149)	82% (31/38)
	Embassy				Mock Foreign				Embassy Test Case			
	100% (4/4)	100% (35/35)	99% (736/744)	97% (140/144)	100% (2/2)	100% (19/19)	96% (459/478)	93% (84/90)	100% (1/1)	83% (5/6)	87% (146/168)	83% (34/41)
	Mock Local				Embassy Test Case				Embassy Test Case			
	100% (2/2)	100% (14/14)	91% (199/218)	87% (40/46)	100% (1/1)	83% (5/6)	87% (144/166)	84% (36/43)				
	Local				Mock Embassy				Local Test Case			
	100% (2/2)	100% (7/7)	97% (152/157)	94% (33/35)	100% (2/2)	100% (19/19)	95% (398/417)	92% (69/75)	100% (1/1)	83% (5/6)	84% (117/139)	80% (28/35)
Wrapper Pattern	Wrapper Client				Mock Wrapper				Wrapper Client Test Case			
	100% (2/2)	100% (10/10)	98% (242/247)	96% (43/45)	100% (2/2)	100% (28/28)	96% (415/434)	93% (76/82)	100% (1/1)	83% (5/6)	83% (107/129)	79% (26/33)
	Wrapper				Mock Wrapper Client				Wrapper Test Case			
	100% (4/4)	100% (29/29)	99% (599/604)	98% (113/115)	100% (2/2)	100% (17/17)	94% (284/303)	89% (51/57)	100% (1/1)	83% (5/6)	85% (126/148)	82% (31/38)
	Mock Wrapper Source				Wrapper Test Case				Wrapper Test Case			
	100% (2/2)	100% (14/14)	91% (199/218)	87% (40/46)	100% (1/1)	83% (5/6)	85% (126/148)	82% (31/38)				
	Wrapper Source				Mock Wrapper				Wrapper Source Test Case			
	100% (2/2)	100% (7/7)	97% (152/157)	94% (33/35)	100% (2/2)	100% (19/19)	95% (355/374)	91% (62/68)	100% (1/1)	83% (5/6)	83% (107/129)	79% (26/33)

Table 5.1: EMMA Code Coverage Results On Ten Social Design Patterns

Table 5.2 represents statistics about the EMMA code coverage results obtained from table 5.1

Covered Item	class, %	method, %	block, %	line, %	class, %	method, %	block, %	line, %	class, %	method, %	block, %	line, %
Metrics	AUT				Mock Agent				Test Case			
Minimum Coverage Percentage	100%	100%	95%	89%	100%	100%	90%	83%	100%	75%	78%	74%
Average Coverage Percentage	100%	100%	98%	96%	100%	100%	94%	91%	100%	83%	83%	79%
100% Coverage Percentage	100%	100%	12%	12%	100%	100%	0%	0%	100%	0%	0%	0%

Table 5.2: EMMA Code Coverage Results Statistics

Here are the main points that could be extracted from Table 5.2:

- All AUT classes have a minimum code coverage of 100%
- All AUT classes have an average code coverage of 100%
- All 100% of the AUT classes have 100% code coverage.
- All AUT code lines have a minimum code coverage of 89%
- All AUT code lines have an average code coverage of 96%
- Not covered lines are out of the scope of either normal or exceptional scenarios mentioned in chapter 4 (as null argument passed to the AUT class, some java `FIPAEException` catch exception blocks,...)

### 5.3 Conclusion

This chapter presents the evaluation of the experimental work done in this thesis. In this chapter, we analyze the quantitative results by using the EMMA (55) code coverage tool to proof that the generated test files (mock agent, associated resource file, and AUT test case) from our Eclipse Plug-in totally covers the AUT code in the ten *social design patterns* we initially support.

## CHAPTER 6: CONCLUSIONS AND SUGGESTIONS FOR FUTURE WORK

### 6.1 Conclusions

In this thesis, we introduce a tool for designing test cases based on a unit testing approach for MASs (8). The main purpose of this approach is to help MASs developers in testing each agent individually. It relies on the use of *mock agents* to guide the design and implementation of agent unit test cases. Each mock agent performs a test script, in which it sends and receives messages from the agent being tested. Each mock agent is responsible for testing a single role of an agent (AUT), under successful and exceptional scenarios. The implementation of this approach is built on top of JADE platform. This approach extends JUnit test framework in order to execute JADE test cases.

Our framework is consisting of five standard components: *Test Suite*, *Test Case*, *AUT*, *Mock Agents*, and *Agent Monitor*. We discuss the *architecture* of our system through the low level design of the adopted *unit test framework* integrated with our *Eclipse plug-in*. A UML class diagram is given for the components of the unit test framework. Also, an Entity Relationship Diagram is given for the Eclipse Plug-in.

The *test cases implementation* through our system is depicted. The mock agent designer can generate the target mock agent files, which depend on the type of the mock agent and the settings that could differ from one domain to another that are set through some placeholders. Those placeholders are resident in the mock agent file as resource values that are read from an associated resource file.

The mock agent designer has the ability to add a new design pattern in addition to the already existing ones and configuring the values of the placeholders of the added design pattern in successful and/or exceptional scenarios that are stored in corresponding XML files. In our work, we provide implementations for a vast majority of agent design patterns.

We also present the design of the agent test design patterns. A set of the most common Agent Design Patterns are presented with their test design patterns for the different roles played by

every agent in each design pattern. The test cases implementation of each design pattern is also provided. Finally, we present the code coverage results obtained from running the EMMA code coverage tool by changing the resource files placeholder values in the ten *social design patterns* we initially support.

## 6.2 Suggestions for Future Work

Our work represents an initial step in the definition of a complete MAS testing process which will provide strategies to the integration and system testing levels. As a future work, is addressing the integration of the adopted testing process with existing development methodologies. Also, how the Mock Agents can be used to progressively test, agents integration scenarios.

For code generation, agent interaction diagrams and specification of agent communication protocols, can work as a source of information to generate the Mock Agents source code. The definition of this generative approach can improve the productivity of the agent unit testing proposal and motivates even more developers to adopt it in the development of large scale multi-agent systems.

For system evaluation, the number of code lines of the generated mock agent used in testing the AUT in the different design patterns can be compared to the other mock objects generators used in unit testing objects like, MockMaker, MockCreator, EasyMock,... etc.

We would like to work with concurrent threads with its known problems as racing, deadlocks. Traditional code coverage metrics, such as statement, block, and branch, fail to address test adequacy concerns introduced by concurrency. Let's take statement coverage as an example. Although an imperfect metric, statement coverage is popular, and it is helpful to a degree in finding out how thoroughly we have tested our code.

Unfortunately, statement coverage does not give us any visibility as to how much of the concurrency in our code is being tested. Statement coverage tells us that we have 100 percent coverage when we use one thread on a type meant to be accessed simultaneously. Clearly,

statement coverage is inadequate to tell us how much of the concurrency aspects of the code we have tested (56).

We need measure the amount of contention that occurred during our tests. Researchers at IBM came up with a code coverage model called Synchronization Coverage that can do just this. Like any code coverage metric, synchronization coverage attempts to measure how well we have tested our code. Any sort of locking in our application means that our code was meant to be accessed simultaneously, and we seriously should be curious whether our test suite actually exercises those locks. We should aim for 100 percent synchronization coverage, especially if our application expects to run in parallel a lot (56).

Like any code coverage metric, synchronization coverage is not perfect, it has its limitations. A chief limitation that synchronization coverage shares with other coverage metrics is that it cannot measure what is not there. Synchronization coverage cannot tell us that we need to lock over some resource, only that the resources we have locked have been contended over. Thus, even 100 percent synchronization coverage does not mean that our testing effort is done, only that we have achieved some level of thoroughness in our testing (56).

We also looking forward in practically using some tools (as mentioned in section 2.3) to assess the quality of our test by asserting the validity of the messages contents exchanged between the AUT and the mock agents.

Finally, we would like to support a wider range of agent design patterns.



## BIBLIOGRAPHY

1. **Gatti, M. and Staa, A.** *Testing & Debugging Multi-Agent Systems: A State of the Art Report*. Rio, Brazil : Computer Science Department, Pontifical Catholic University of Rio de Janeiro – PUC, February 2006.
2. *Evaluating multiagent systems: a record/replay approach*. **Huget, M.-P. and Demazeau, Y.** 2004. Proceedings. IEEE/WIC/ACM International Conference on 2004. pp. 536 - 539. Digital Object Identifier 10.1109/IAT.2004.1343013.
3. **Wallis, M.** *An introduction to Agent Based Software Engineering with a focus on software reusability*. Newcastle : Computer Science, Faculty of Engineering and Built Environment, University of Newcastle, 4 20, 2004.
4. *Agents in Object-Oriented Software Engineering*. **Garcia, A., Lucena, C. and Cowan, D.** s.l. : Elsevier, April 2004, Software: Practice & Experience, Vol. 34, pp. 489-521.
5. **Timm, I. J. and Scholz, T.** Testing in Agent Oriented Software Engineering. s.l. : Universität Bremen, Center for Computing Technologies (TZI), Hendrik Fürstenau: Universität Hohenheim, Wirtschaftsinformatik 2, IV.8 Testing, p. 9.
6. *Patterns of Intelligent and Mobile Agents*. **Kendall, E. A., et al.** USA : St. Paul, 1998. In: Proceedings of the 2nd International Conference on Autonomous Agents, Agents 98. pp. 92 – 99.
7. **Silva, C., et al.** *Describing Agent-Oriented Design Patterns in Tropos*. Centro de Informática - Universidade Federal de Pernambuco (UFPE) Av., Prof. Luiz Freire. Recife PE, Brazil : s.n. S/N, 50732-970.
8. **Coelho, R., et al.** *Unit Testing in Multi-agent Systems using Mock Agents and Aspects*. PUC-Rio, Brazil : Computer Science Department, Pontifical Catholic University of Rio de Janeiro.
9. *Process Models for Agent-based Development*. **Cernuzzi, L., Cossentino, M. and Zambonelli, F.** 2, March 2005, Journal of Engineering Applications of Artificial Intelligence, Vol. 18, pp. 205-222.
10. *Managing the Development of Large Software systems*. **Royce, WW.** s.l. : TRW, August 1970. In: Proceedings of the IEEE Wescon. pp. 1-9.

11. **Beck, K and Andres, C.** *Extreme Programming Explained*. s.l. : Addison-Wesley Professional, 2004.
12. **Beck, K.** *Test Driven Development by Example*. s.l. : Addison-Wesley Professional, 2002. 0-321-14653-0.
13. **Thaller, GE.** *Software-Test - Verifikation und Validation 2*. Hannover : Verlag Heinz Heise GmbH & Co KG, 2002. 3-88229-198-2.
14. *Verification and Validation and Artificial Intelligence*. **Menzies, T and Pecheur, C.** s.l. : Academic Press, expected Publication in 2005, In: Advances of Computers 65.
15. **Dijkstra, EW.** The Programming Task Considered as an Intellectual Challenge. [Online] Transcribed by McCarthy DC, Eindhoven, 1969. <http://www.cs.utexas.edu/users/EWD/transcriptions/EWD02xx/EWD273.html>.
16. **Rushby, J.** Theorem proving for verification. In *Franck Cassez, editor, Modelling and Verification of Parallel Processes*. Nantes : Springer LNAI, June 2000, Vol. 2067.
17. *State-Based Testing of object-oriented Programmes*. **Turner, C and Robson, D.** Montreal : IEEE Computer Society Press, 1993. In: Proceedings of International Conference on Software Maint.
18. **Riedemann, EH.** *Testmethoden für sequentielle und nebenläufige Software-Systeme*. Stuttgart : B.G. Teubner, 1997.
19. *Formal Semantics for AUML Agent Interaction Protocols Diagrams*. **Cabac, L. and Moldt, D.** Berlin Heidelberg New York : Springer, 2005. In: Odell J, Giorgini P, Müller JP (eds) Agent-Oriented Software Engineering V. 5th International Workshop (AOSE 2004). pp. 47-61.
20. **Sommerville, I.** *Software Engineering*. 6th Edition. s.l. : Addison Wesley, 2000.
21. **Pressman, R. S.** *Engenharia de Software*. 5th Edition. Rio de Janeiro : McGraw-Hill, 2002.
22. *A Test Agent for Testing Agents and Their Communities*. **Rouff, C.** s.l. : IEEE, 2002. Aerospace Conference Proceedings. Vol. Volume 5, pp. 5 - 2638. Digital Object Identifier 10.1109/AERO.2002.1035446.

23. *Multi-agent systems implementation and testing*. **Caire, G., et al.** Vienna, Austria : s.n., April 2004. In Proc. of From Agent Theory to Agent Implementation - Fourth International Symposium (AT2AI-4).
24. *Visualising and debugging distributed multi-agent systems*. **Ndumu, D. T., et al.** [ed.] O Etzioni, J. P. Müller and J. M. Bradshaw. Seattle, Washington, United States : ACM Press, New York, NY, 326-333, 1999. In Proceedings of the Third Annual Conference on Autonomous Agents. DOI=<http://doi.acm.org/10.1145/301136.301220>.
25. *ZEUS: an Advanced Tool-kit for Engineering Distributed Multi-Agent Systems*. **Nwana, H, et al.** London : s.n., 1998. In Proceedings of PAAM98.
26. *An exploration of bugs and debugging in multi-agent systems*. **Poutakidis, D., Padgham, L. and Winikoff, M.** 2003. In the proceedings of the 14th International Symposium on Methodologies for Intelligent Systems (ISMIS 2003).
27. *Debugging Multi-Agent Systems using Design Artifacts: The case of Interaction Protocols*. **Poutakidis, D., Padgham, L. and Winikoff, M.** Bologna, Italy : s.n., July 15-19, 2002. In the proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2002).
28. **Padgham, L., Winikoff, M. and Poutakidis, D.** *Adding Debugging Support to the Prometheus Methodology. Engineering Applications of Artificial Intelligence, special issue on Agent-oriented Software Development.* 2nd Issue. March 2005. pp. 173-190. Vol. 18. DOI:10.1016/j.engappai.2004.11.018.
29. **Ronsse, M., et al.** *Record/replay for nondeterministic program executions.* s.l. : Communications of the ACM, September 2003. 46(9).
30. *The MADKIT agent platform architecture*. **Gutknecht, O. and Ferber, J.** 2000. In Agents Workshop on Infrastructure for Multi-Agent Systems. pp. 48-55.
31. **Rodrigues, L.F, et al.** *Towards an Integration Test Architecture for Open MAS (SEAS).* s.l. : SBES, 2005.

32. **Heselius, J.** *Debugging parallel systems: A state of the art Report. MRTC Report.* s.l. : Mlardalen Real-Time Research Centre, Mlardalen University, 2002. ISSN 1404-3041 ISRN MDH-MRTC-63/2002-1-SE.
33. **Bellifemine, F., et al.** Jade Programmer's Guide, Jade Programming tutorial - JADE 3.3. [Online] 2005. <http://sharon.cselt.it/projects/jade/>.
34. **Gamma, E. and Beck, K.** JUnit: A regression testing framework. [Online] 2000. [Cited: February 22, 2006.] <http://www.junit.org>.
35. **Myers, G. J.** *The Art of Software Testing.* [ed.] Wiley. 2nd Edition. New Jersey : John Wiley & Sons, 2004. p. 227.
36. *JADE - A FIPA2000 Compliant Agent Development Environment.* **Bellifemine, F., Poggi, A. and Rimassa, G.** Montreal : ACM Press, 2001. In: International conference on autonomous agents. Vol. 5, pp. 216-217.
37. *SKWYRL, "Social arChitectures for Agent Software Systems EngineeRing".* [Online] <http://www.isys.ucl.ac.be/skwyr/>.
38. **Do, T. T. and Kolp, M.** *Social Patterns for Designing Multiagent Systems.* Louvain-La-Neuve, Belgium : Information Systems Research Unit, Catholic University of Louvain, Place de Doyens, 1, 1348.
39. **Rao, A.S. and Georgeff, M.P.** *BDI agents: from theory to practice.* s.l. : Australian Artificial Intelligence Institute, 1995. Technical Note 56.
40. *JACK Intelligent Agents.* [Online] 2005. <http://www.agent-software.com/>.
41. *Information Systems Development through Social Structures.* **Kolp, M., Giorgini, P. and Mylopoulos, J.** Ishia, Italy : s.n., 2002. In: Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering.
42. *Towards Requirements-Driven Information Systems Engineering: The Tropos Project.* **Castro, J., Kolp, M. and Mylopoulos, J.** s.l. : Information Systems News, Elsevier, 2002, Vol. 27, pp. 365-89.
43. **Yu, E.** *Modelling Strategic Relationships for Process Reengineering.* Canada : Department of Computer Science. University of Toronto, 1995. Ph.D. thesis.

44. *Introspecting Agent Oriented Design Patterns*. **Kolp, M., et al.** s.l. : World Publishing, 2005. In: S. K. Chang (Eds), *Advances in Software Engineering and Knowledge Engineering*. Vol. III.
45. **Rumbaugh, J., Jacobson, I. and Booch, G.** *The Unified Modeling Language*. s.l. : Addison Wesley, 1999. Reference Manual.
46. *Agent system development method based on agent patterns*. **Tahara, Y., Ohsuga, A. and Honiden, S.** Los Angeles, California, USA : s.n., 1999. In: *Proceedings of the 21st International Conference on Software Engineering*. pp. 356 – 367.
47. *Architectural design patterns for multiagent coordination*. **Hayden, S., Carrick, C. and Yang, Q.** Seattle, USA : s.n., 1999. In: *Proceedings of the 3rd International Conference on Autonomous Agents, Agents 99*.
48. **Gamma, E., et al.** *Design Patterns: Elements of Reusable Object-Oriented Software*. s.l. : Addison-Wesley, 1995.
49. **Keen, D.** *Object Oriented Programming, Mock Objects and test driven design (TDD)*.
50. **Fowler, M.** *Refactoring: Improving the Design of Existing Code*. s.l. : Addison Wesley Longman, 1999.
51. **Larman, C.** *Applying UML and patterns: an introduction to object-oriented analysis and design and iterative development*. 3rd Edition. s.l. : Prentice Hall PTR, 2005.
52. **Beck, K.** *Smalltalk Best Paractice Patterns*. s.l. : Prentice Hall, 1996. ISBN-10: 013476904x.
53. *Java Technology*. [Online] 2005. <http://www.java.sun.com/>.
54. **Do, T. T., et al.** *A Framework for Design Patterns in Tropos*. Louvain-La-Neuve, Belgium : ISYS - Information Systems Research Unit, Catholic University of Louvain, Place des Doyens, 1, 1348.
55. *EMMA Code Coverage Tool*. [Online] <http://emma.sourceforge.net/>.
56. **Dern, C. and Tan, R. P.** *Synchronization Coverage, Code Coverage for Concurrency*. [Online] <http://msdn.microsoft.com/en-us/magazine/ee310108.aspx>.
57. FIPA protocol documentation. [Online] <http://www.fipa.org/>.

58. **Kolp, M., et al.** *Introspecting Agent-Oriented Design Patterns*. Louvain-La-Neuve, Belgium : ISYS - Information Systems Research Unit, IAG - School of Management, Catholic University of Louvain, Place des Doyens, 1, 1348.
59. **Hayden, S. C., Carrick, C. and Yang, Q.** *Architectural Design Patterns for Multiagent Coordination*. Canada : Information Service Agents Lab, Simon Fraser University, Burnaby, BC V5A 1S6, October 2, 1998.
60. **Woods, S. G. and Barbacci, M.** *Architectural Evaluation of Collaborative Agent-Based Systems*. USA : Carnegie Mellon University, 1999. Technical Report. CMU/SEI-99-TR-025.

## APPENDIX A

### A.1 Subscription Pattern

#### A.1.1 Design Pattern Template

Table A.1 represents the Subscription design pattern template.

Template Element	Description
Name	Subscription Pattern
Intent	To let a number of service providers to advertise their services by subscribing to the yellow pages.
Applicability	Use when an agent (service provider) needs to advertise its services by subscribing to another agent (yellow page).
Motivation Example	Consider an application of book-trading, in which, each agent can play a BookSeller role, a BookBuyer role or both. As soon as a BookSeller subscriptionServiceProvider agent joins the environment, it registers itself in a ServiceDirectory subscriptionYellowPage as a “book-seller” and starts to wait for “book-buying” requests. When a BookBuyer agent joins the environment, it initially looks for the agents already registered in the ServiceDirectory as “book-sellers”. After that, it sends a “cfp” message to all the agents registered as “book-sellers”.
Participants	The ServiceProvider advertises its services by subscribing to the YellowPage and if it wishes no longer to be advertised, it can request to be unsubscribed. The YellowPage agent allows agents to publish one or more services they provide so that other agents can find and successively exploit them.

Table A.1: Subscription Design Pattern Template

#### A.1.2 Collaboration diagram

Figure A.1 represents the collaboration diagram of the Subscription pattern (54).

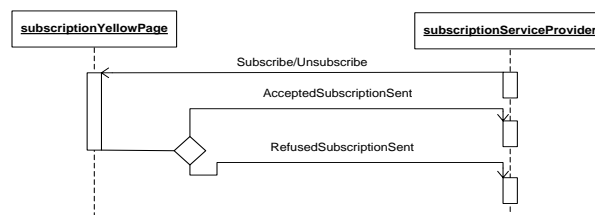


Figure A.1: The Collaboration diagram of the Subscription pattern

Figure A.2 represents the structure diagram of the Subscription pattern.



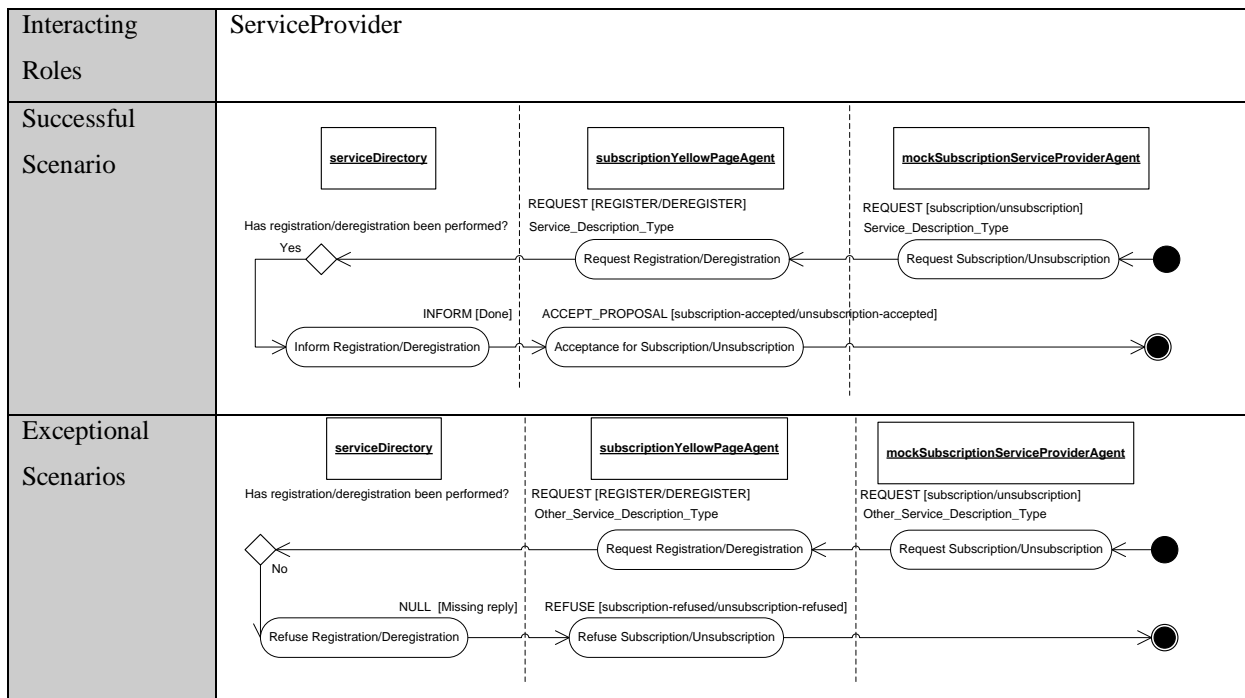


Table A.2: Test Design Pattern of the YellowPageAgent

When the `mockSubscriptionServiceProviderAgent` joins the environment, it sends a message characterized by a `REQUEST` performative indicating the willingness to subscribe/unsubscribe its service with the `subscriptionYellowPageAgent`. In this case, if a `NULL` message has been received from the JADE `DFService` agent, the `subscriptionYellowPageAgent` replies with a `REFUSE` message (exceptional scenario).

If the `mockSubscriptionServiceProviderAgent` does not send a request message, the `YellowPageServer` behaviour will be blocked and is put into the blocked behaviours queue by the agent scheduler. When a new message (`REQUEST`) is inserted in the agent message queue, the blocked behaviour becomes available for execution again, so that it has a chance to process the received message.

#### A.1.5.2 SubscriptionServiceProviderAgent

Table A.3, represents the design of test design pattern of the Agent Under Test.

Agent	ServiceProviderAgent
Roles	ServiceProvider
Interacting	YellowPage

Roles	
Successful Scenario	Is the same as the "successful scenario" depicted in the previous section with swapping the AUT and mock agent roles.
Exceptional Scenarios	

Table A.3: Test Design Pattern of the ServiceProviderAgent

When the `mockSubscriptionYellowPageAgent` receives a subscription/ unsubscription request message from the `subscriptionServiceProviderAgent`, it tries to subscribe/unsubscribe the `DFAgentDescription` of the `subscriptionServiceProviderAgent` with the JADE `DFService` agent and replies with a message characterized by either:

- i. An `ACCEPT_PROPOSAL` performative, indicating that the subscription/ unsubscription has been done successfully with the `DFService`, in this case the `subscriptionServiceProviderAgent` can start receiving requests from the client agents requesting the subscribed service or terminate in case of unsubscription (normal scenario).
- ii. A `REFUSE` performative, indicating that a `NULL` message has been received from the `DFService` agent, in this case the `subscriptionServiceProviderAgent` will terminate (exceptional scenario).
- iii. No `RESPONSE`, indicating that the `mockSubscriptionYellowPageAgent` does not respond to the subscription/unsubscription request message, in this case the `RequestSubscriptionPerformer/` `RequestUnsubscriptionPerformer` behaviour will be blocked and is put into the blocked behaviours queue by the agent scheduler. When a new message (subscription/unsubscription `ACCEPT_PROPOSAL/REFUSE`) is inserted in the agent message queue, the blocked

behaviour becomes available for execution again, so that it has a chance to process the received message.

### **A.1.6 Pattern Implementation**

#### **A.1.6.1 SubscriptionYellowPageAgent**

The mock agent `MockSubscriptionServiceProviderAgent` has a plan (test scenario) represented by two JADE Behaviors called `RequestSubscriptionPerformer` and `RequestUnsubscriptionPerformer` to test the AUT agent `SubscriptionYellowPageAgent`. To execute the AUT agent test case `SubscriptionYellowPageAgentTestCase`, all we need is to create a subclass of `JADETestCase` and to implement a test method `testSubscription()` that creates an instance of AUT `SubscriptionYellowPageAgent`, and an instance of the mock agent `MockSubscriptionServiceProviderAgent`. After that, this test method waits until their interaction finishes and asks the mock agent about the test result.

#### **A.1.6.2 SubscriptionServiceProviderAgent**

The mock agent `MockSubscriptionYellowPageAgent` has a plan (test scenario) represented by one JADE Behavior called `YellowPageServer` to test the AUT agent `SubscriptionServiceProviderAgent`. To execute the AUT agent test case `SubscriptionServiceProviderAgentTestCase`, all we need is to create a subclass of `JADETestCase` and to implement a test method `testSubscriptionServiceProvider()` that creates an instance of the AUT `SubscriptionServiceProviderAgent`, and an instance of the mock agent `MockSubscriptionYellowPageAgent`. After that, this test method waits until their interaction finishes and asks the mock agent about the test result.

## **A.2 Call-For-Proposals Pattern**

### **A.2.1 Design Pattern Template**

Table A.4 represents the Call-For-Proposals design pattern template.

Template Element	Description

<b>Name</b>	Call-For-Proposals Pattern
<b>Intent</b>	To let a buyer to issue a call for proposals for a service to all participants (sellers) and then accepts proposals that offer the service for a specified cost.
<b>Applicability</b>	Use when an agent (buyer) needs to call for proposals for a service to all participants agents (sellers) and then accepts proposals that offer the service for a specified cost.
<b>Motivation Example</b>	<p>Consider an application of book-trading, in which, each agent can play a <code>BookSeller</code> role, a <code>BookBuyer</code> role or both. As soon as a <code>BookSeller</code> agent joins the environment, it registers itself in a <code>ServiceDirectory</code> as a “book-seller” and starts to wait for “book-buying” requests.</p> <p>When a <code>BookBuyer</code> agent joins the environment, it initially looks for the agents already registered in the <code>ServiceDirectory</code> as “book-sellers”. After that, it sends a “cfp” message to all the agents registered as “book-sellers”. When the <code>BookSeller</code> agent receives a “cfp” message from a <code>BookBuyer</code>, it searches in its catalogue for the requested book. If it is available, the <code>BookSeller</code> agent sends a “propose” message in reply to “cfp” message, whose content is the book price. If on the other hand, the <code>BookSeller</code> agent does not have the book on catalogue; it will send a “refuse” message informing the <code>BookBuyer</code> agent that the book is not available.</p> <p>The <code>BookBuyer</code> agent receives all proposals/refusals from <code>BookSeller</code> agents and chooses the agent with the best offer. Then, it sends the chosen seller a “purchase” message. When the <code>BookSeller</code> agent receives a “purchase” message it removes the book from the catalogue and sends an “inform” message to notify to the <code>BookBuyer</code> agent that the book sale was complete. However, if for any reason the book is no more available in the catalogue the <code>BookSeller</code> agent sends a “failure” message informing <code>BookBuyer</code> agent that the requested book is no more available.</p> <p>If the <code>BookBuyer</code> agent receives a message indicating that the sale was complete, the agent can terminate. Otherwise, it will re-execute its plan and try to buy the book again from some other agent.</p>
<b>Participants</b>	The initiator (buyer) issues a call for proposals for a service to all participants (sellers) and then accepts proposals that offer the service for a specified cost. The seller is the service provider that sends a service price proposal to the buyer that found it through the yellow-page service. The services directory (yellow-page) agent allows seller agents to publish one or more services they provide so that buyer agents can find and successively exploit them.

Table A.4: Call-For-Proposals Design Pattern Template

## A.2.2 Collaboration diagram

Figure A.4 represents the collaboration diagram of the Call-For-Proposals pattern (8).

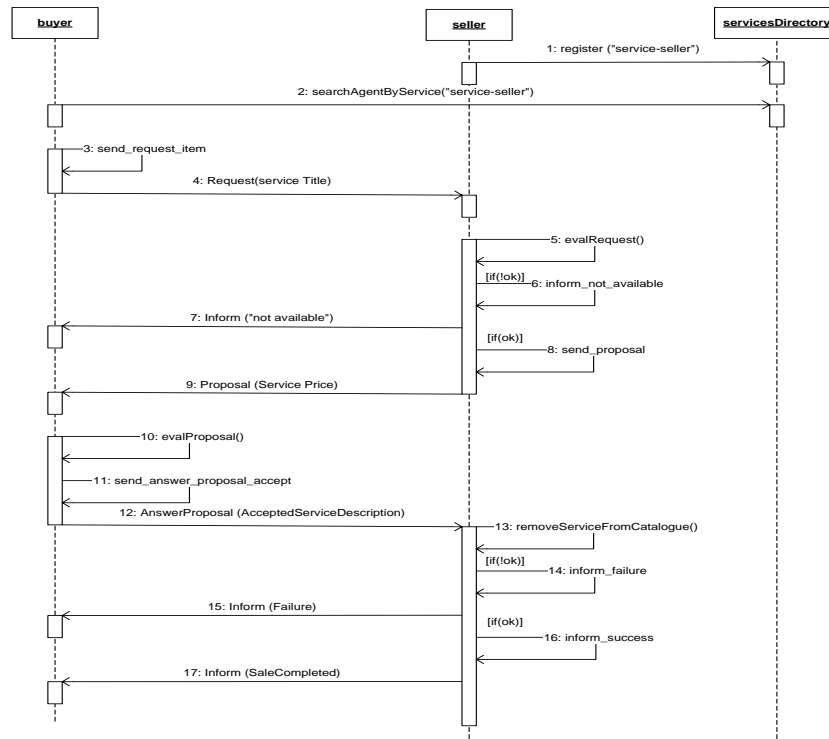


Figure A.4: The Collaboration diagram of the Call-For-Proposals pattern

## A.2.3 Structure Diagram

Figure A.5 represents the structure diagram of the Call-For-Proposals pattern.

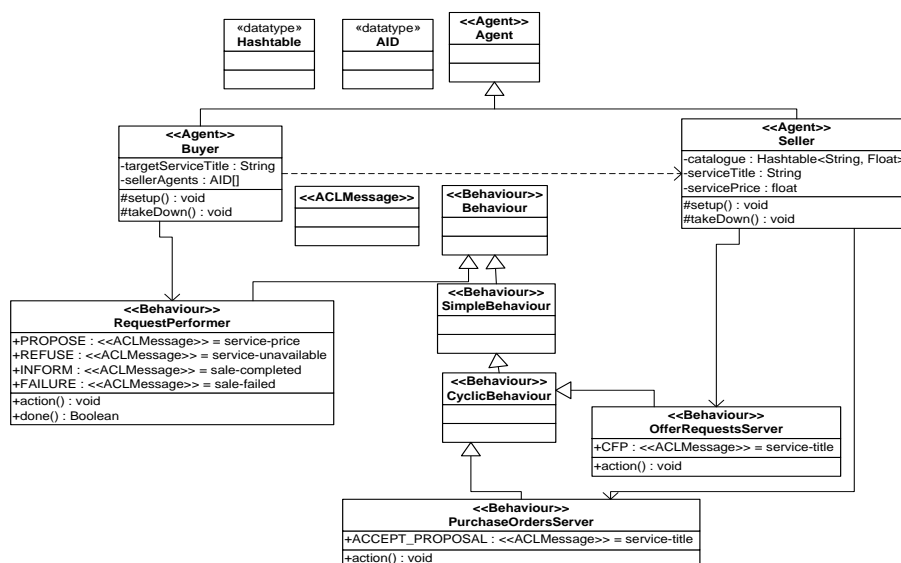


Figure A.5: The Structure diagram of the Call-For-Proposals pattern

## A.2.4 Behaviour Diagram

Figure A.6 represents the behaviour diagram of the Call-For-Proposals pattern.

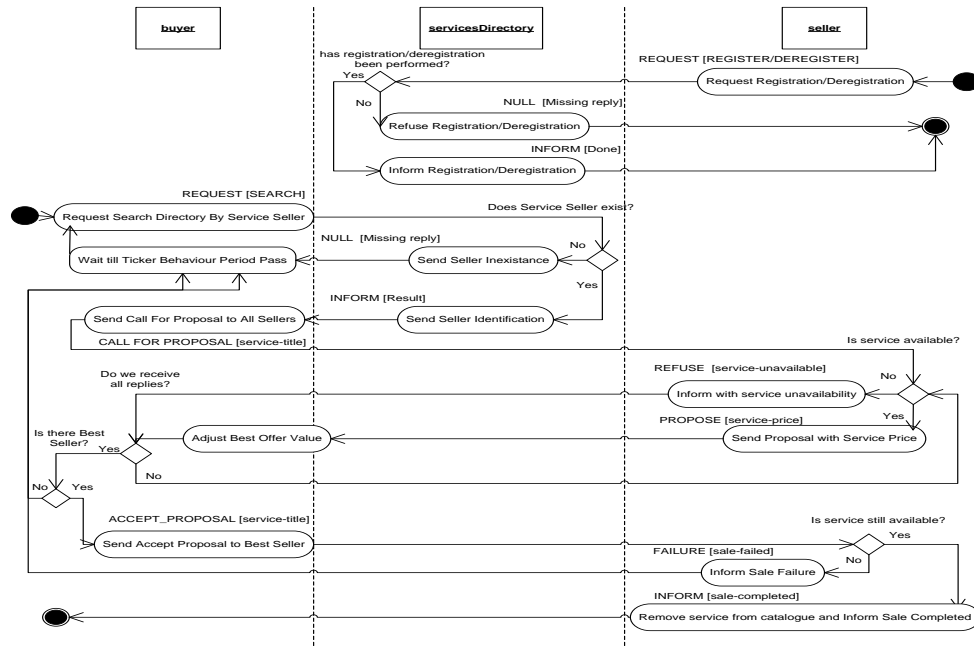


Figure A.6: The Behaviour diagram of the Call-For-Proposals pattern

## A.2.5 Test Design Patterns

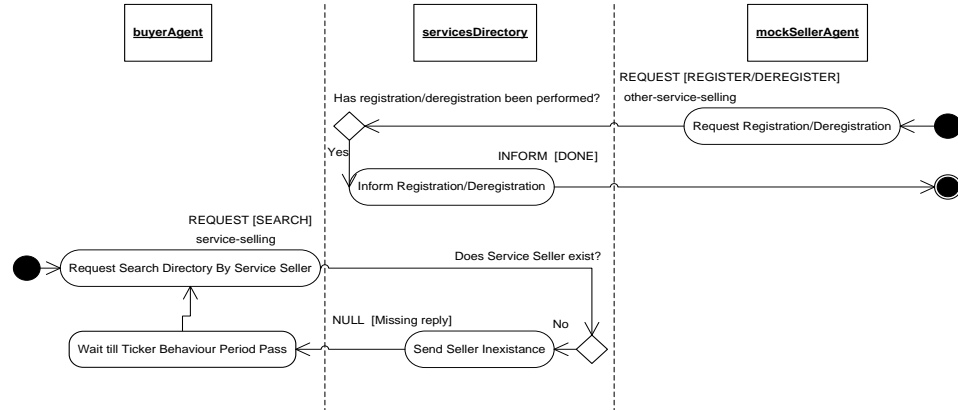
### A.2.5.1 BuyerAgent

Table A.5, represents the design of test design pattern of the Agent Under Test.

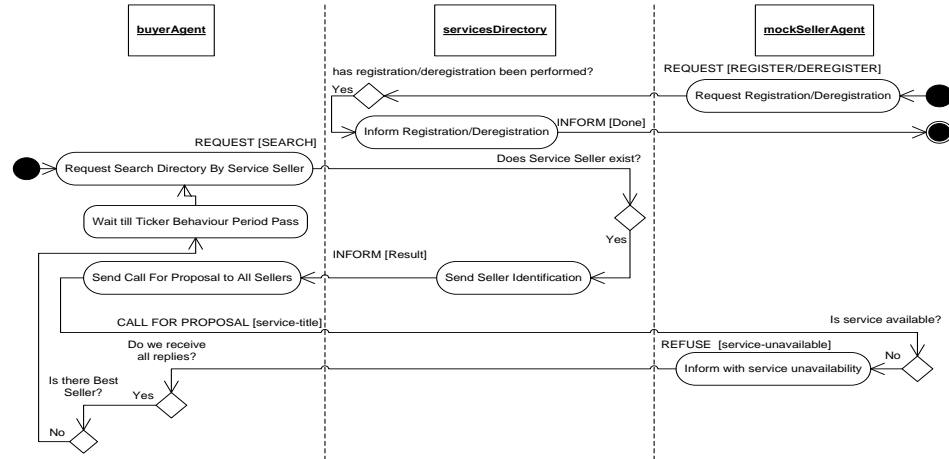
Agent	BuyerAgent
Roles	Buyer
Interacting Roles	Seller
Successful Scenario	<pre> sequenceDiagram     participant buyerAgent     participant servicesDirectory     participant mockSellerAgent      mockSellerAgent-&gt;&gt;mockSellerAgent: REQUEST [REGISTER/DEREGISTER] service-selling     mockSellerAgent-&gt;&gt;mockSellerAgent: Request Registration/Deregistration     mockSellerAgent-&gt;&gt;mockSellerAgent: Inform Registration/Deregistration     mockSellerAgent-&gt;&gt;mockSellerAgent: INFORM [Done]      buyerAgent-&gt;&gt;servicesDirectory: REQUEST [SEARCH] service-selling     buyerAgent-&gt;&gt;servicesDirectory: Request Search Directory By Service Seller     servicesDirectory-&gt;&gt;servicesDirectory: has registration/deregistration been performed?     servicesDirectory-&gt;&gt;servicesDirectory: Yes     servicesDirectory-&gt;&gt;servicesDirectory: No     servicesDirectory-&gt;&gt;servicesDirectory: INFORM [Result]     servicesDirectory-&gt;&gt;servicesDirectory: Does Service Seller exist?     servicesDirectory-&gt;&gt;servicesDirectory: No     servicesDirectory-&gt;&gt;servicesDirectory: Send Seller Identification     servicesDirectory-&gt;&gt;servicesDirectory: Yes     servicesDirectory-&gt;&gt;servicesDirectory: Is service available?     servicesDirectory-&gt;&gt;servicesDirectory: No     servicesDirectory-&gt;&gt;servicesDirectory: REFUSE [service-unavailable]     servicesDirectory-&gt;&gt;servicesDirectory: Inform with service unavailability     servicesDirectory-&gt;&gt;servicesDirectory: Yes     servicesDirectory-&gt;&gt;servicesDirectory: PROPOSE [service-price]     servicesDirectory-&gt;&gt;servicesDirectory: Send Proposal with Service Price     servicesDirectory-&gt;&gt;servicesDirectory: Is service still available?     servicesDirectory-&gt;&gt;servicesDirectory: No     servicesDirectory-&gt;&gt;servicesDirectory: FAILURE [sale-failed]     servicesDirectory-&gt;&gt;servicesDirectory: Inform Sale Failure     servicesDirectory-&gt;&gt;servicesDirectory: Yes     servicesDirectory-&gt;&gt;servicesDirectory: INFORM [sale-completed]     servicesDirectory-&gt;&gt;servicesDirectory: Remove service from catalogue and Inform Sale Completed      buyerAgent-&gt;&gt;servicesDirectory: Send Call For Proposal to All Sellers     buyerAgent-&gt;&gt;servicesDirectory: CALL FOR PROPOSAL [service-title]     buyerAgent-&gt;&gt;servicesDirectory: Do we receive all replies?     buyerAgent-&gt;&gt;servicesDirectory: Yes     buyerAgent-&gt;&gt;servicesDirectory: Adjust Best Offer Value     buyerAgent-&gt;&gt;servicesDirectory: Is there Best Seller?     buyerAgent-&gt;&gt;servicesDirectory: No     buyerAgent-&gt;&gt;servicesDirectory: ACCEPT_PROPOSAL [service-title]     buyerAgent-&gt;&gt;servicesDirectory: Send Accept Proposal to Best Seller     buyerAgent-&gt;&gt;servicesDirectory: Remove service from catalogue and Inform Sale Completed     </pre>

## Exceptional Scenarios

### First Exceptional Scenario



### Second Exceptional Scenario



### Third Exceptional Scenario

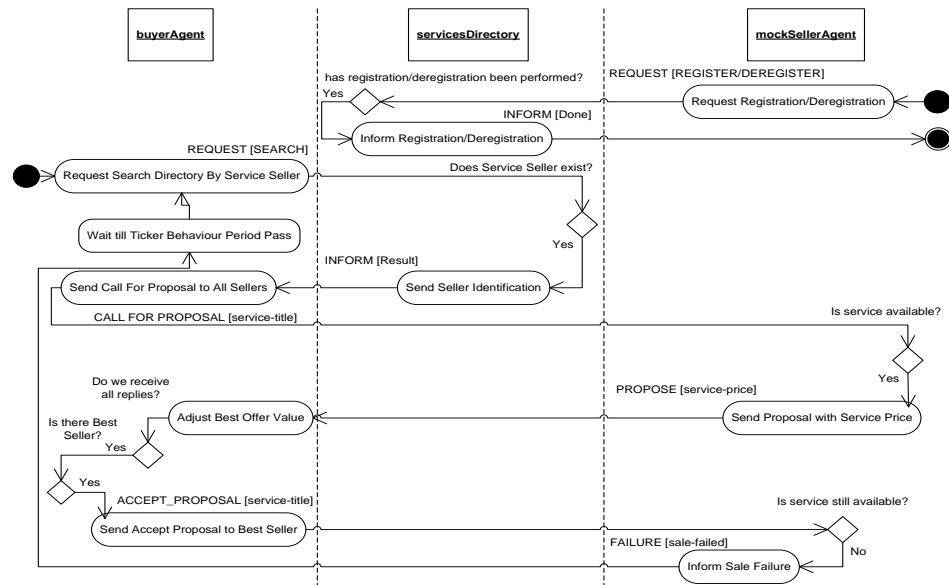


Table A.5: Test Design Pattern of the BuyerAgent

When the JADE `DFService` agent receives the "service-selling" request message from the `buyerAgent`, it replies with a message characterized by either:

- i. An `INFORM` performative, indicating not only that the sellers have been located but also the sellers identification (normal scenario).
- ii. A `NULL` message, if the `TickerBehaviour` time period is not reasonable enough, the `buyerAgent` will either found no `mockSellerAgents` registered with the `DFService` agent yet, or it will only deal with a small sample of `mockSellerAgents` that may not represent the service real market price or even may not have the service available (first exceptional scenario).
- iii. No `RESPONSE`, indicating that the `DFService` agent does not respond to the "service-selling" request message, in this case the `buyerAgent` will wait the given period specified in the constructor of the `TickerBehaviour` in order to re-request from the `DFService` agent in case any agents had registered as "service-selling" in the meanwhile.

When the `mockSellerAgent` receives a call for proposal message from the `buyerAgent`, it replies with a message characterized by either:

- i. A `PROPOSE` performative, indicating not only that the service is available but also the service price, in this case the `buyerAgent` will check whether this offer is the best one so far, if so, the `buyerAgent` will set it as the best offer and wait for the proposals/refusals of the remaining known seller agents (if any) in order to send offer acceptance to the best seller (normal scenario).
- ii. A `REFUSE` performative, indicating that the service is unavailable, in this case the `buyerAgent` will wait for the proposals/refusals of the remaining known seller agents (if any), If all known seller agents send `REFUSE` messages, then the `RequestPerformer` behaviour will be done, and the control of execution will be returned to the caller



`TickerBehaviour` in order to re-look for any agents had registered in the `DFService` agent as “service-selling” in the meanwhile (second exceptional scenario).

- iii. No RESPONSE, indicating that the `mockSellerAgent` does not respond to the call for proposal message, in this case the `RequestPerformer` behaviour will be blocked and an event is fired to notify its parent behaviour. The behaviour is put into the blocked behaviours queue by the agent scheduler. When a new message (PROPOSE/REFUSE) is inserted in the agent message queue, the blocked behaviour becomes available for execution again, so that it has a chance to process the received message.

When the `mockSellerAgent` receives the offer acceptance message from the `buyerAgent`, it replies with a message characterized by either:

- i. An INFORM performative, indicating that the service sale was completed and the service is removed from the catalogue, in this case the `buyerAgent` can terminate (normal scenario).
- ii. A FAILURE performative, indicating that the service, for any reason, is no more available in the catalogue, in this case the `buyerAgent` will re-execute its plan and try to buy the service again from some other agent (third exceptional scenario).
- iii. No RESPONSE, indicating that the `mockSellerAgent` does not respond to the offer acceptance message, in this case the `RequestPerformer` behaviour will be blocked and an event is fired to notify its parent behaviour. The behaviour is put into the blocked behaviours queue by the agent scheduler. When a new message (INFORM/FAILURE) is inserted in the agent message queue, the blocked behaviour becomes available for execution again, so that it has a chance to process the received message.

#### A.2.5.2 SellerAgent

Table A.6, represents the design of test design pattern of the Agent Under Test.

Agent	SellerAgent
-------	-------------

Roles	Seller
Interacting Roles	Buyer
Successful Scenario	Is the same as the "successful scenario" depicted in the previous section with swapping the AUT and mock agent roles.
Exceptional Scenarios	<p>First Exceptional Scenario</p> <p>Second Exceptional Scenario</p>

Table A.6: Test Design Pattern of the SellerAgent

For the subscription/unsubscription normal and exceptional scenarios of the mockBuyerAgent testing the sellerAgent, please review the subscription pattern Appendix A.1.5.2 SubscriptionServiceProviderAgent.

When the `mockBuyerAgent` receives the identification of all the agents registered as “service-selling”, it sends a message characterized by a `CALL FOR PROPOSAL` performative indicating the willingness for sellers proposals to offer a specific service. In this case, if the service is unavailable the `sellerAgent` replies with a `REFUSE` message (first exceptional scenario).

If the `mockBuyerAgent` does not send a call for proposal message, the `OfferRequestsServer` behaviour will be blocked and is put into the blocked behaviours queue by the agent scheduler. When a new message (`CALL FOR PROPOSAL`) is inserted in the agent message queue, the blocked behaviour becomes available for execution again, so that it has a chance to process the received message.

When the `mockBuyerAgent` receives the proposal from the `sellerAgent`, it sends the `sellerAgent` a message characterized by an `ACCEPT_PROPOSAL` performative indicating offer acceptance (i.e. purchase order). In this case, if the service in the offer acceptance differs from the initial one in the CFP message, the `sellerAgent` replies with a `FAILURE` message (second exceptional scenario).

If the `mockBuyerAgent` does not send an accept proposal message, the `PurchaseOrdersServer` behaviour will be blocked and is put into the blocked behaviours queue by the agent scheduler. When a new message (`ACCEPT_PROPOSAL`) is inserted in the agent message queue, the blocked behaviour becomes available for execution again, so that it has a chance to process the received message.

## **A.2.6 Pattern Implementation**

### **A.2.6.1 BuyerAgent**

The mock agent `MockSellerAgent` has a plan (test scenario) represented by two JADE Behaviors called `OfferRequestsServer` and `PurchaseOrdersServer` to test the AUT agent `BuyerAgent`. To execute the AUT agent test case `BuyerAgentTestCase`, all we need is to create a subclass of `JADETestCase` and to implement a test method

`testBuying()` that creates an instance of the AUT `BuyerAgent`, and an instance of the mock agent `MockSellerAgent`. After that, this test method waits until their interaction finishes and asks the mock agent about the test result.

#### A.2.6.2 SellerAgent

The mock agent `MockBuyerAgent` has a plan (test scenario) represented by a JADE Behavior called `RequestPerformer` to test the AUT agent. To execute the AUT agent test case `SellerAgentTestCase`, all we need is to create a subclass of `JADETestCase` and to implement a test method `testSelling()` that creates an instance of the AUT `SellerAgent`, and an instance of the mock agent `MockBuyerAgent`. After that, this test method waits until their interaction finishes and asks the mock agent about the test result.

### A.3 Bidding Pattern

#### A.3.1 Design Pattern Template

Table A.7 represents the Bidding design pattern template.

Template Element	Description
<b>Name</b>	Bidding Pattern
<b>Intent</b>	To let an auctioneer to find the market price of a good by initially proposing a price below that of the supposed market value and then gradually raising the price.
<b>Applicability</b>	Use when an agent (auctioneer) needs to organize and lead an auction, and receives proposals from all participant agents (bidders).
<b>Motivation Example</b>	<p>In the English Auction, the auctioneer seeks to find the market price of a good by initially proposing a price below that of the supposed market value and then gradually raising the price. Each time the price is announced, the auctioneer waits to see if any buyers will signal their willingness to pay the suggested price. The Auctioneer issues a new call for bids to the confirmed bidder agents with an incremented price than the best accepted proposed price. The auction continues until no buyers are prepared to pay the proposed price, at which point the auction ends. If the last price that was accepted by a buyer exceeds the auctioneer (privately known) reservation price, the good is sold to that buyer for the agreed price. If the last accepted price is less than the reservation price, the good is not sold.</p> <p>In the Dutch Auction, the auctioneer attempts to find the market price for a good by</p>

	starting bidding at a price much higher than the expected market value, then progressively reducing the price until one of the buyers accepts the price. The rate of reduction of the price is up to the auctioneer and they usually have a reserve price below which not to go. If the auction reduces the price to the reserve price with no buyers, then the auction terminates.
<b>Participants</b>	The initiator (auctioneer) organizes and leads the bidding process, and receives proposals. The participants (bidders) send their proposals to the auctioneer signaling their willingness to pay the proposed price in order to buy the proposed goods.

Table A.7: Bidding Design Pattern Template

### A.3.2 Collaboration diagram

Figure A.7 represents the collaboration diagram of the Bidding pattern (English Auction) (57).

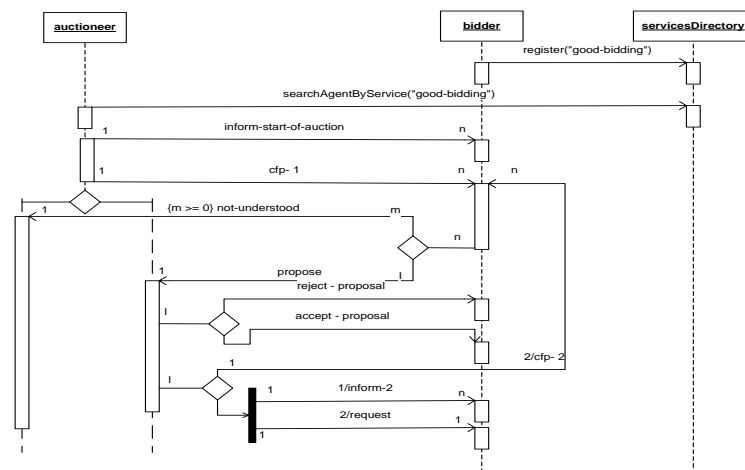


Figure A.7: The Collaboration diagram of the Bidding pattern (English Auction)

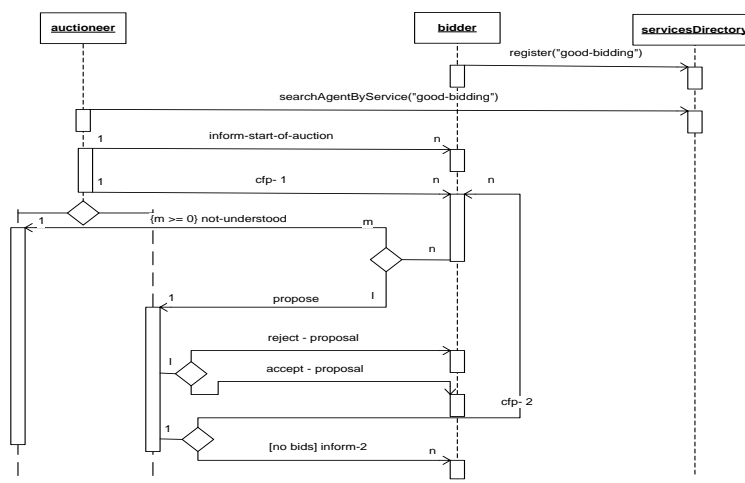


Figure A.8: The Collaboration diagram of the Bidding pattern (Dutch Auction)

For similarity of behaviour, we found it is sufficient to implement the English Auction Pattern.

### A.3.3 Structure Diagram

Figure A.9 represents the structure diagram of the Bidding pattern.

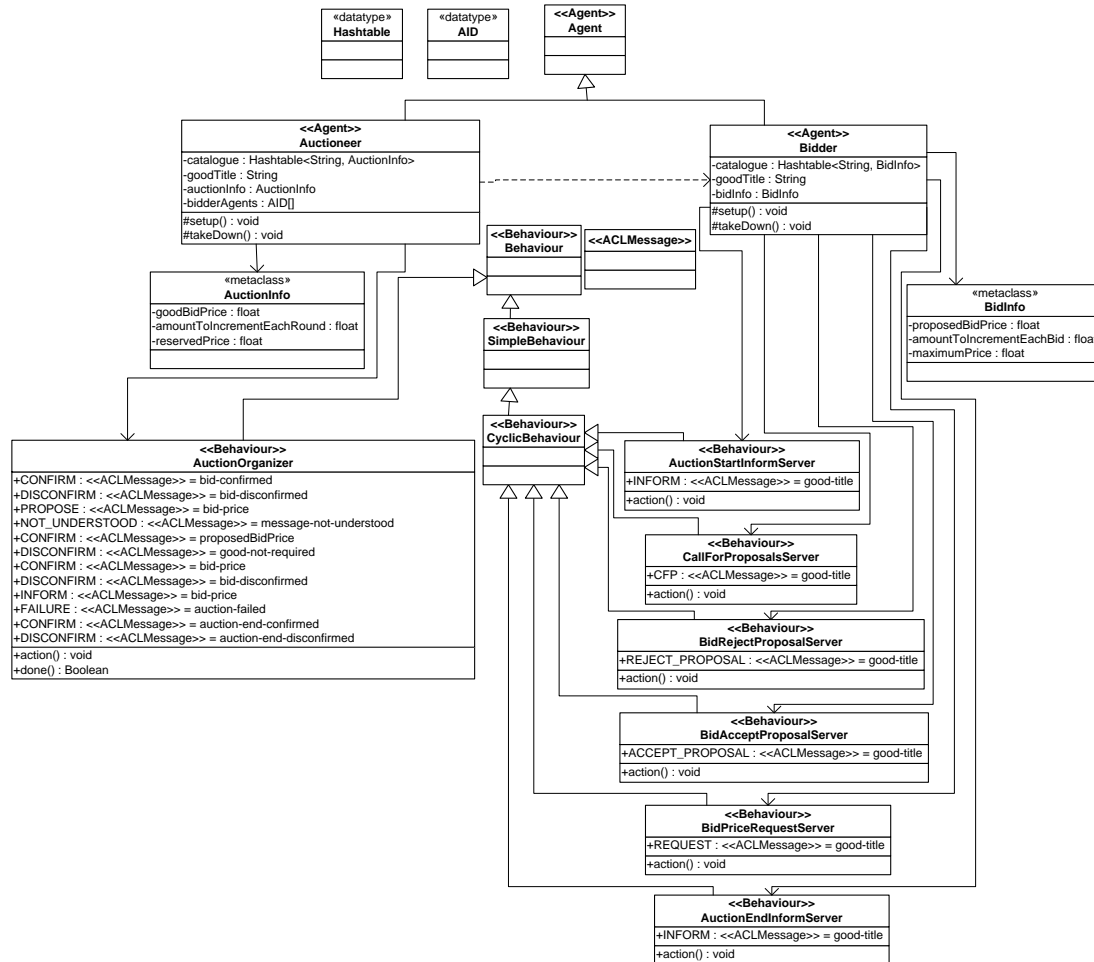


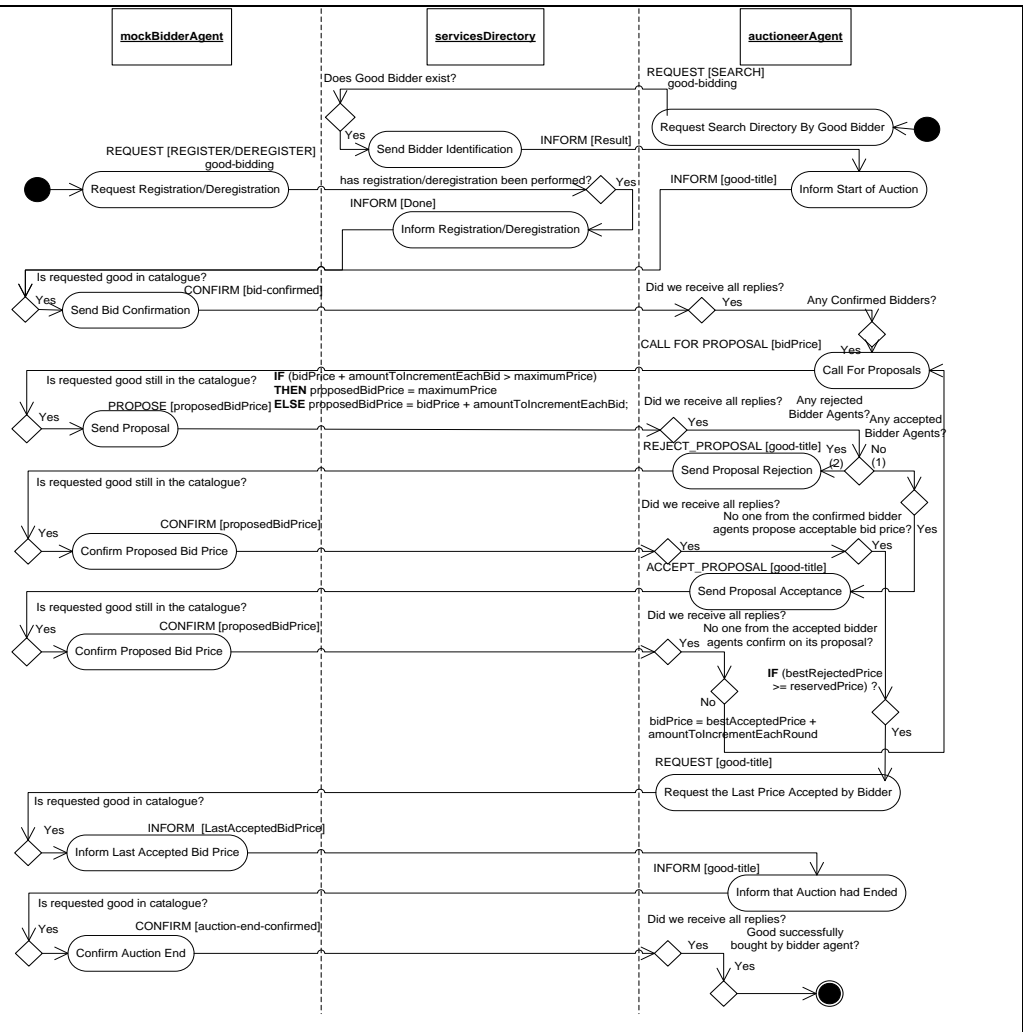
Figure A.9: The Structure diagram of the Bidding pattern

### A.3.4 Behaviour Diagram

Figure A.10 represents the behaviour diagram of the Bidding pattern.



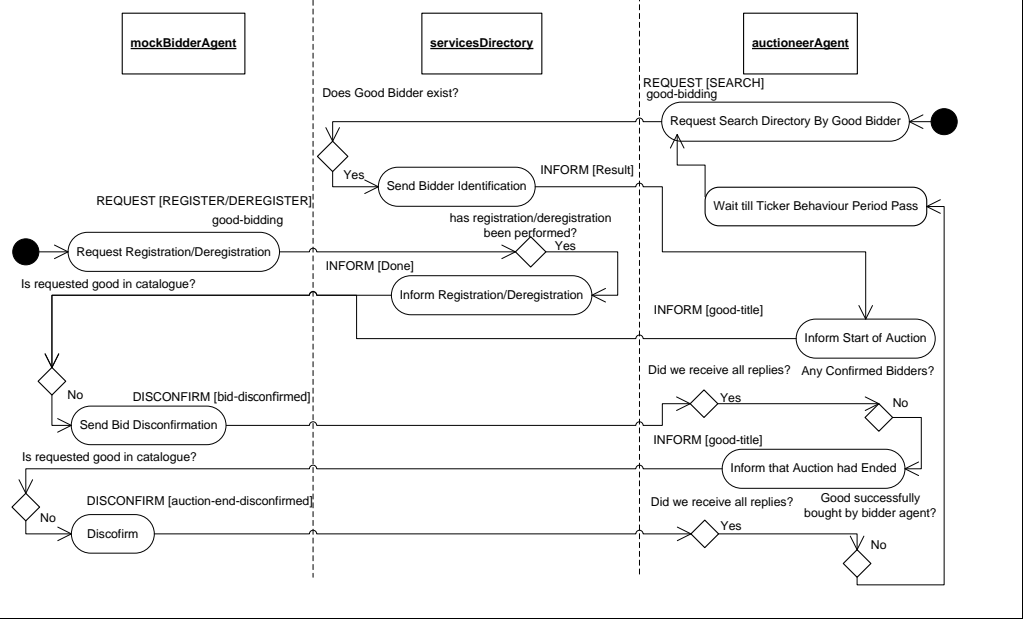
Successful Scenario



## Exceptional Scenarios

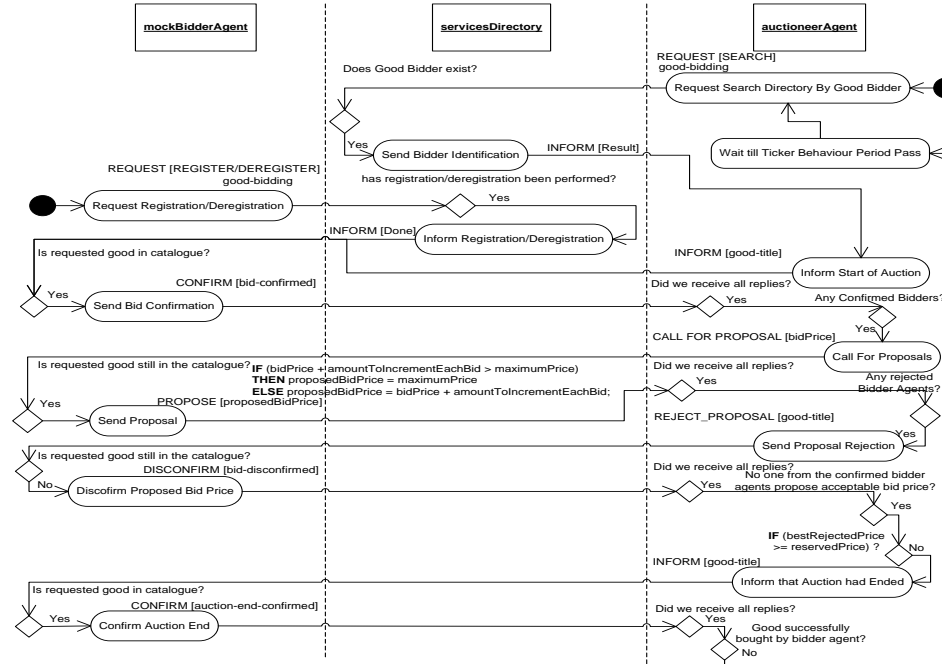
---

First Exceptional Scenario

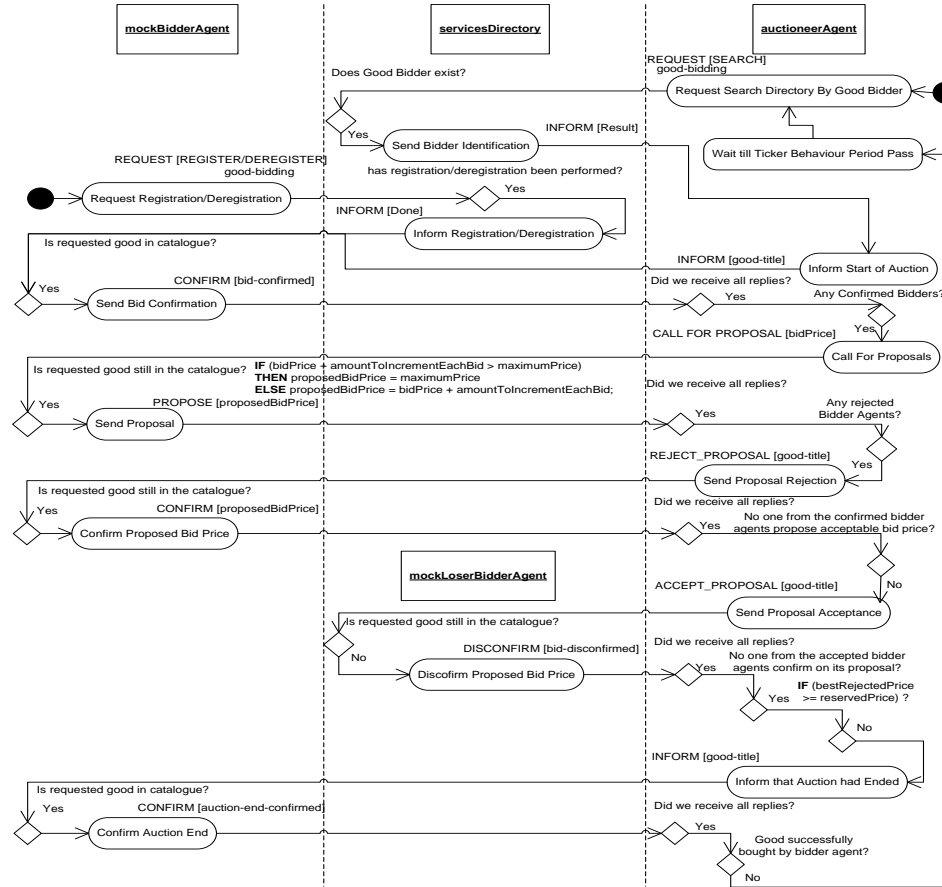




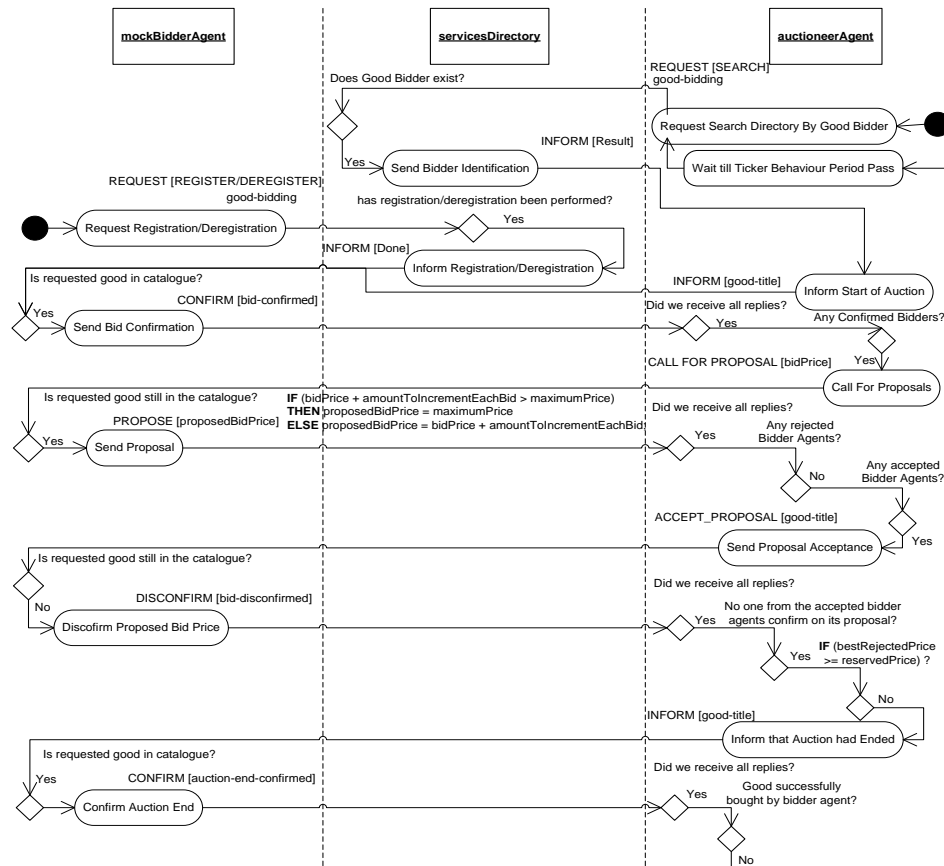
## Second Exceptional Scenario



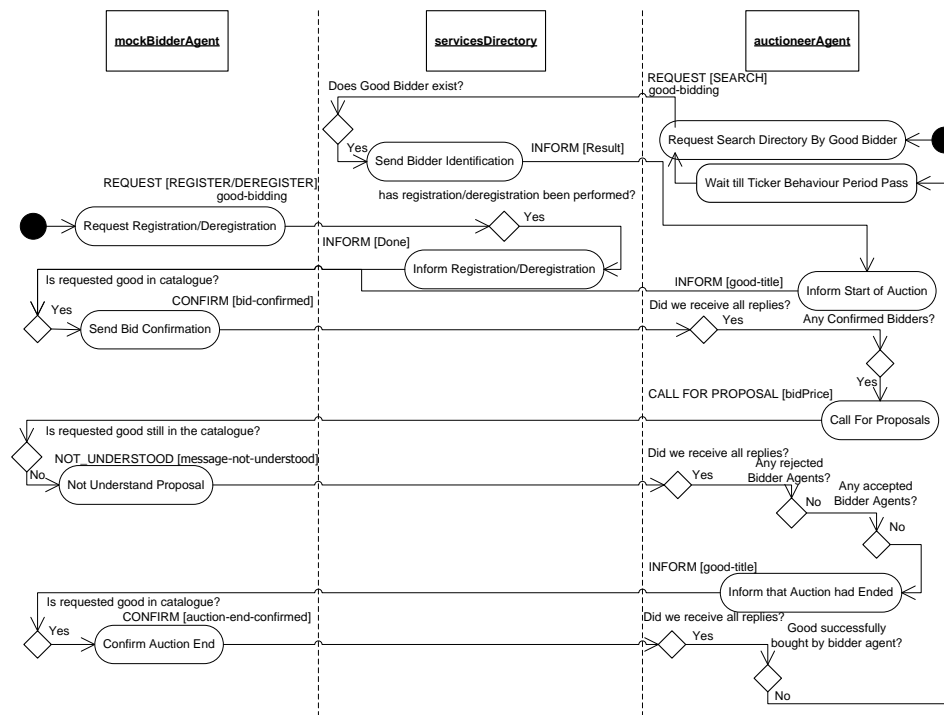
## Third Exceptional Scenario



#### Fourth Exceptional Scenario



### Fifth Exceptional Scenario



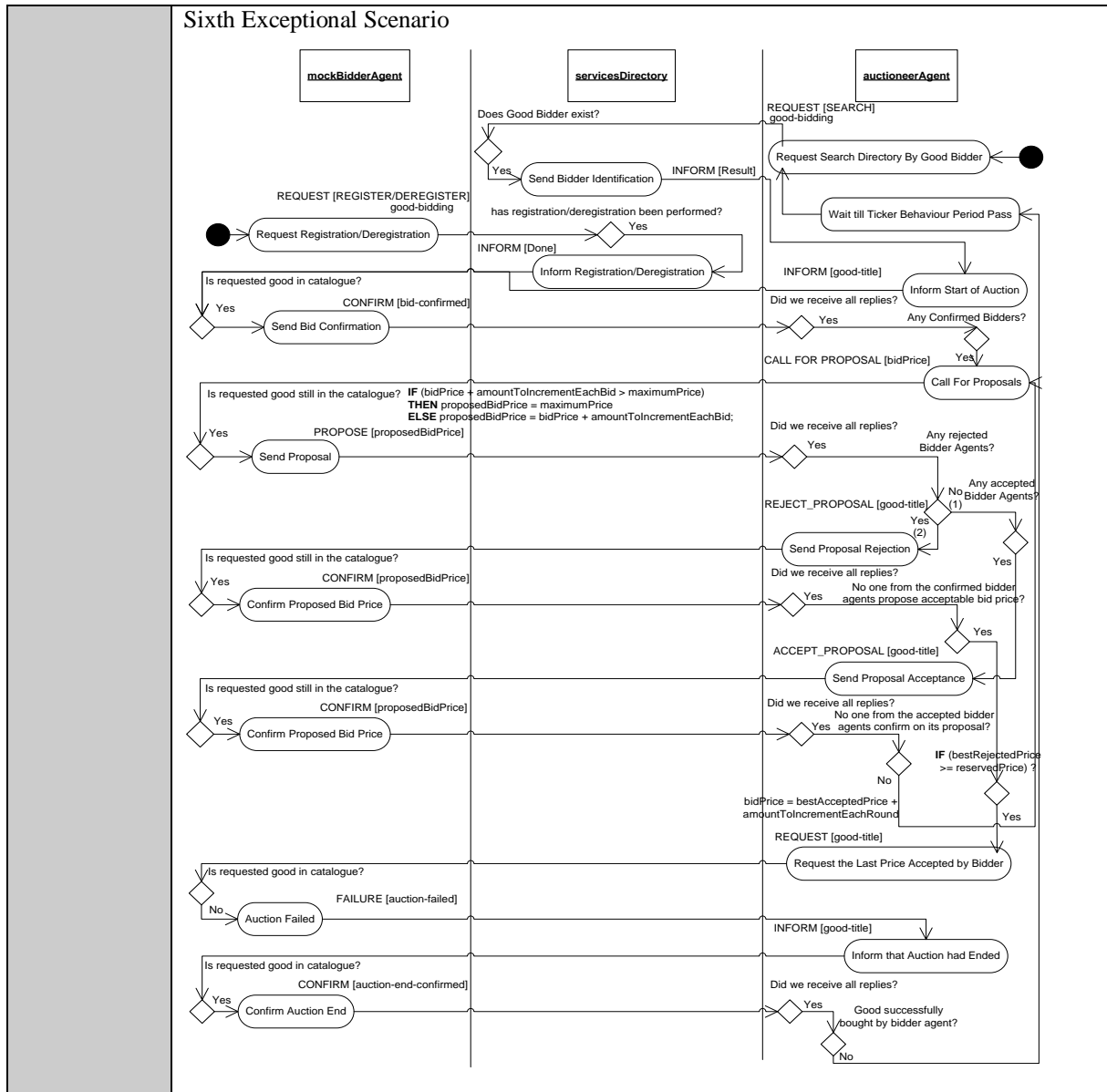


Table A.8: Test Design Pattern of the AuctioneerAgent

For the subscription/unsubscription normal and exceptional scenarios of the mockBidderAgent testing the auctioneerAgent, please review the call for proposal pattern Appendix A.2.5.1 BuyerAgent.

When the mockBidderAgent receives an "Auction-Start" inform message from the auctioneerAgent, it searches in its catalogue for the requested good and replies with a message characterized by either:

- i. A CONFIRM performative, indicating that the good is available and required to be purchased, in this case the `auctioneerAgent` will add the `mockBidderAgent` to the confirmed bidder list, and wait for the (CONFIRM/DISCONFIRM) messages of the remaining known bidder agents in order to send call for proposals to the confirmed bidder agents (normal scenario).
- ii. A DISCONFIRM performative, indicating that the good is unavailable, in this case the `auctioneerAgent` will wait for the (CONFIRM/DISCONFIRM) messages of the remaining known bidder agents. If all known bidder agents send DISCONFIRM messages, then the `auctioneerAgent` will send "Auction-End" inform message to all known bidder agents and get response back, and then the `AuctionOrganizer` behaviour will be done, and the control of execution will be returned to the caller `TickerBehaviour` in order to re-look for any agents had registered with the `DFService` agent as “good-bidding” in the meanwhile (first exceptional scenario).
- iii. No RESPONSE, indicating that the `mockBidderAgent` does not respond to the inform message, in this case the `AuctionOrganizer` behaviour will be blocked and an event is fired to notify its parent behaviour. The behaviour is put into the blocked behaviours queue by the agent scheduler. When a new message (CONFIRM/DISCONFIRM) is inserted in the agent message queue, the blocked behaviour becomes available for execution again, so that it has a chance to process the received message.

When the `mockBidderAgent` receives a call for proposal message from the `auctioneerAgent`, it searches in its catalogue for the requested good and replies with a message characterized by either:

- i. A PROPOSE performative, indicating that the good is available and still required to be purchased and the bid price is not exceeding the maximum price that could be paid by the `mockBidderAgent`, in this case the `auctioneerAgent` will check if the proposed bid price is smaller than its suggested price, then it will add `mockBidderAgent` to the rejected bidders list and sends reject proposal message, else will add the

`mockBidderAgent` to the accepted bidders list and sends accept proposal message (normal scenario).

- ii. A `NOT_UNDERSTOOD` performative, indicating that the good has been purchased from another `auctioneerAgent` in the meanwhile or the bid price is exceeding the maximum price that could be paid by the `mockBidderAgent`, in this case the `auctioneerAgent` will wait for the (`PROPOSE/NOT_UNDERSTOOD`) messages of the remaining known bidder agents. If all known bidder agents send `NOT_UNDERSTOOD` messages, then the `auctioneerAgent` will send "Auction-End" inform message to all known bidder agents and get response back, and then the `AuctionOrganizer` behaviour will be done, and the control of execution will be returned to the caller `TickerBehaviour` in order to re-look for any agents had registered with the `DFService` agent as “good-bidding” in the meanwhile (fifth exceptional scenario).
- iii. No `RESPONSE`, indicating that the `mockBidderAgent` does not respond to the call for proposal message, in this case the `AuctionOrganizer` behaviour will be blocked and an event is fired to notify its parent behaviour. The behaviour is put into the blocked behaviours queue by the agent scheduler. When a new message (`PROPOSE/NOT_UNDERSTOOD`) is inserted in the agent message queue, the blocked behaviour becomes available for execution again, so that it has a chance to process the received message.

When the `mockBidderAgent` receives a bid reject proposal message from the `auctioneerAgent`, it replies with a message characterized by either:

- i. A `CONFIRM` performative, indicating that the good is still existing in its catalogue and required to be purchased, in this case, the `auctioneerAgent` will check whether this proposal is the best one so far, if so, the `auctioneerAgent` sets it as the best rejected proposal and wait for the (`CONFIRM/DISCONFIRM`) messages of the remaining bidder agents. At the end, check if there is no bidder agent proposes an acceptable price and the best rejected proposed price exceeds the auctioneer, privately known, reservation price,

the `auctioneerAgent` ends the auction transaction with the best rejected bidder agent (normal scenario).

- ii. A `DISCONFIRM` performative, indicating that the good, for any reason, is no more available in the catalogue, in this case the `auctioneerAgent` will wait for (`CONFIRM/DISCONFIRM`) messages of the remaining bidder agents. At the end, if there is no bidder agent proposes an acceptable price, then the `auctioneerAgent` will send "Auction-End" inform message to all known bidder agents and get response back, and then the `AuctionOrganizer` behaviour will be done, and the control of execution will be returned to the caller `TickerBehaviour` in order to re-look for any agents had registered with the `DFService` agent as "good-bidding" in the meanwhile (second exceptional scenario).
- iii. No `RESPONSE`, indicating that the `mockBidderAgent` does not respond to the reject proposal message, in this case the `AuctionOrganizer` behaviour will be blocked and an event is fired to notify its parent behaviour. The behaviour is put into the blocked behaviours queue by the agent scheduler. When a new message (`CONFIRM/DISCONFIRM`) is inserted in the agent message queue, the blocked behaviour becomes available for execution again, so that it has a chance to process the received message.

When the `mockBidderAgent` receives a bid accept proposal message from the `auctioneerAgent`, it replies with a message characterized by either:

- i. A `CONFIRM` performative, indicating that the good is still existing in its catalogue and required to be purchased, in this case, the `auctioneerAgent` will check whether this proposal is the best one so far, if so, the `auctioneerAgent` sets it as the best accepted proposal and wait for the (`CONFIRM/DISCONFIRM`) messages of the remaining bidder agents. At the end, the `auctioneerAgent` issues a new call for bids to the confirmed bidder agents with an incremented price than the best accepted proposed price (normal scenario).

- ii. A DISCONFIRM performative, indicating that the good, for any reason, is no more available in the catalogue, in this case the `auctioneerAgent` will wait for (CONFIRM/DISCONFIRM) messages of the remaining bidder agents. At the end, if there is no accepted bidder agent confirmed on its proposals, then the `auctioneerAgent` will check if the best rejected proposed price exceeds the auctioneer, privately known, reservation price, then ends the auction transaction with the best rejected confirmed bidder agent, else sends an "Auction-End" inform message to all known bidder agents and get response back, and then the `AuctionOrganizer` behaviour will be done, and the control of execution will be returned to the caller `TickerBehaviour` in order to re-look for any agents had registered with the `DFService` agent as “good-bidding” in the meanwhile (third and fourth exceptional scenarios).
- iii. No RESPONSE, indicating that the `mockBidderAgent` does not respond to the accept proposal message, in this case the `AuctionOrganizer` behaviour will be blocked and an event is fired to notify its parent behaviour. The behaviour is put into the blocked behaviours queue by the agent scheduler. When a new message (CONFIRM/DISCONFIRM) is inserted in the agent message queue, the blocked behaviour becomes available for execution again, so that it has a chance to process the received message.

When the `mockBidderAgent` receives a bid price request message from the `auctioneerAgent`, it searches in its catalogue for the requested good and replies with a message characterized by either:

- i. An INFORM performative, indicating not only that the good is available, but also the good price, in this case the `auctioneerAgent` will end the auction and terminate (normal scenario).

- ii. A FAILURE performative, indicating that the good, for any reason, is no more available in the catalogue, in this case the `auctioneerAgent` will send "Auction-End" inform message to all known bidder agents and get response back, and then the `AuctionOrganizer` behaviour will be done, and the control of execution will be returned to the caller `TickerBehaviour` in order to re-look for any agents had registered with the `DFService` agent as "good-bidding" in the meanwhile (sixth exceptional scenario).
- iii. No RESPONSE, indicating that the `mockBidderAgent` does not respond to the request message, in this case the `AuctionOrganizer` behaviour will be blocked and an event is fired to notify its parent behaviour. The behaviour is put into the blocked behaviours queue by the agent scheduler. When a new message (INFORM/FAILURE) is inserted in the agent message queue, the blocked behaviour becomes available for execution again, so that it has a chance to process the received message.

When the `mockBidderAgent` receives an "Auction-End" inform message from the `auctioneerAgent`, it searches in its catalogue for the requested good and replies with a message characterized by either:

- i. A CONFIRM performative, indicating that the good is available, in this case the `auctioneerAgent` will wait for the bidding (CONFIRM/DISCONFIRM) messages of the remaining known bidder agents and check if the good had been sold to terminate (normal scenario).
- ii. A DISCONFIRM performative, indicating that the good is unavailable, in this case the `auctioneerAgent` will wait for the (CONFIRM/DISCONFIRM) messages of the remaining known bidder agents and check if the good had not been sold the `AuctionOrganizer` behaviour will be done, and the control of execution will be returned to the caller `TickerBehaviour` in order to re-look for any agents had registered with the `DFService` agent as "good-bidding" in the meanwhile (first exceptional scenario).



- iii. No RESPONSE, indicating that the `mockBidderAgent` does not respond to the inform message, in this case the `AuctionOrganizer` behaviour will be blocked and an event is fired to notify its parent behaviour. The behaviour is put into the blocked behaviours queue by the agent scheduler. When a new message (CONFIRM/DISCONFIRM) is inserted in the agent message queue, the blocked behaviour becomes available for execution again, so that it has a chance to process the received message.

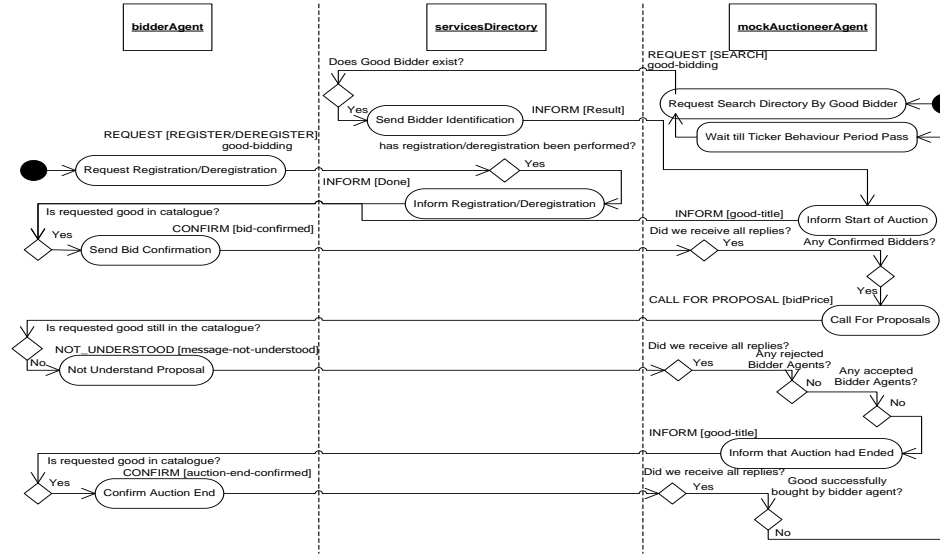
### A.3.5.2 BidderAgent

Table A.9, represents the design of test design pattern of the Agent Under Test.

Agent	BidderAgent
Roles	Bidder
Interacting Roles	Auctioneer
Successful Scenario	Is the same as the "successful scenario" depicted in the previous section with swapping the AUT and mock agent roles.
Exceptional Scenarios	<p>First Exceptional Scenario</p> <pre> sequenceDiagram     participant bidderAgent     participant servicesDirectory     participant mockAuctioneerAgent      bidderAgent-&gt;&gt;servicesDirectory: Does Good Bidder exist?     servicesDirectory-&gt;&gt;mockAuctioneerAgent: REQUEST [SEARCH] good-bidding     mockAuctioneerAgent-&gt;&gt;servicesDirectory: Request Search Directory By Good Bidder     servicesDirectory-&gt;&gt;bidderAgent: Send Bidder Identification     bidderAgent-&gt;&gt;servicesDirectory: INFORM [Result]     servicesDirectory-&gt;&gt;mockAuctioneerAgent: Wait till Ticker Behaviour Period Pass     mockAuctioneerAgent-&gt;&gt;servicesDirectory: INFORM [other-good-title]     servicesDirectory-&gt;&gt;bidderAgent: Inform Start of Auction     bidderAgent-&gt;&gt;servicesDirectory: REQUEST [REGISTER/DEREGISTER] good-bidding     servicesDirectory-&gt;&gt;mockAuctioneerAgent: Did we receive all replies? Any Confirmed Bidders?     mockAuctioneerAgent-&gt;&gt;servicesDirectory: No     servicesDirectory-&gt;&gt;bidderAgent: DISCONFIRM [bid-disconfirmed]     bidderAgent-&gt;&gt;servicesDirectory: Send Bid Disconfirmation     servicesDirectory-&gt;&gt;mockAuctioneerAgent: INFORM [other-good-title]     mockAuctioneerAgent-&gt;&gt;servicesDirectory: Did we receive all replies? Good successfully bought by bidder agent?     mockAuctioneerAgent-&gt;&gt;servicesDirectory: No     servicesDirectory-&gt;&gt;bidderAgent: DISCONFIRM [auction-end-disconfirmed]     bidderAgent-&gt;&gt;servicesDirectory: Disconfirm     </pre> <p>Second Exceptional Scenario</p>



### Fourth Exceptional Scenario



### Fifth Exceptional Scenario

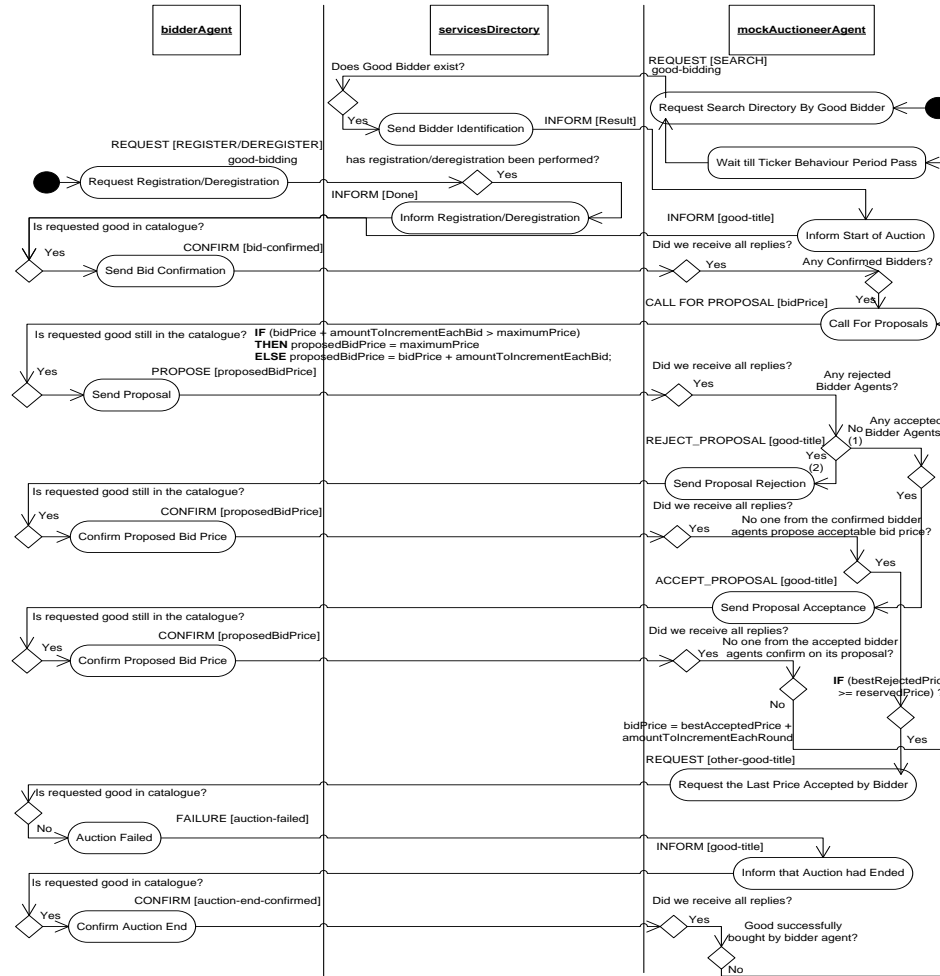


Table A.9: Test Design Pattern of the BidderAgent

For the subscription/unsubscription normal and exceptional scenarios of the `mockAuctioneerAgent` testing the `bidderAgent`, please review the subscription pattern `Appendix A.1.5.2 SubscriptionServiceProviderAgent`.

When the `mockAuctioneerAgent` receives the identification of all the agents registered as “good-bidding”, it sends a message characterized by an `INFORM` performative indicating the auctioneer announcement to bidders with auction starting on the specified good. In this case, if the good is unavailable, the `bidderAgent` replies with a `DISCONFIRM` message (first exceptional scenario).

If the `mockAuctioneerAgent` does not send an "Auction-Start" inform message, the `AuctionStartInformServer` behaviour will be blocked and is put into the blocked behaviours queue by the agent scheduler. When a new message (`INFORM`) is inserted in the agent message queue, the blocked behaviour becomes available for execution again, so that it has a chance to process the received message.

When the `mockAuctioneerAgent` receives all confirmation/disconfirmation messages from all `bidderAgents`, it sends a message characterized by a `CALL FOR PROPOSAL` performative to the confirmed `bidderAgents` indicating the auctioneer willingness for bidder proposals to bid on the specified good. In this case, if the good in the call for proposal message differs than the one in the "Auction-Start" inform message, the `bidderAgent` replies with a `NOT_UNDERSTOOD` message (fourth exceptional scenario).

If the `mockAuctioneerAgent` does not send a call for proposal message, the `CallForProposalsServer` behaviour will be blocked and is put into the blocked behaviours queue by the agent scheduler. When a new message (`CALL FOR PROPOSAL`) is inserted in the agent message queue, the blocked behaviour becomes available for execution again, so that it has a chance to process the received message.

When the `mockAuctioneerAgent` receives all proposal messages from the `bidderAgents`, it sends the `bidderAgents` a message characterized by an

ACCEPT\_PROPOSAL/REJECT\_PROPOSAL performative indicating bid proposal acceptance/rejection respectively. In both cases, if the good is unavailable the bidderAgent replies with a DISCONFIRM message (second and third exceptional scenarios).

If the mockAuctioneerAgent does not send bid accept/reject proposal message, the BidAcceptProposalServer/BidRejectProposalServer behaviour will be blocked and is put into the blocked behaviours queue by the agent scheduler. When a new message (ACCEPT\_PROPOSAL/REJECT\_PROPOSAL) is inserted in the agent message queue, the blocked behaviour becomes available for execution again, so that it has a chance to process the received message.

When the mockAuctioneerAgent receives all proposal confirmation/disconfirmation messages from the bidderAgents, it sends the best bidderAgent that confirms on its rejected proposal (in case, no other bidder agent proposal is accepted and the proposed price exceeds the auctioneer, privately known, reservation price) a message characterized by a REQUEST performative to complete the auction transaction. In this case, if the good is unavailable the bidderAgent replies with a FAILURE message (fifth exceptional scenario).

If the mockAuctioneerAgent does not send a bid price request message, the BidPriceRequestServer behaviour will be blocked and is put into the blocked behaviours queue by the agent scheduler. When a new message (REQUEST) is inserted in the agent message queue, the blocked behaviour becomes available for execution again, so that it has a chance to process the received message.

When the mockAuctioneerAgent receives the good price inform message from the winning bidderAgent, it sends to all known bidderAgents (that are initially informed by "Auction-Start") a message characterized by an INFORM performative indicating the auctioneer announcement to bidders with "Auction-End" on the specified good. In this case, if the good is unavailable, the bidderAgent replies with a DISCONFIRM message (first exceptional scenario).

If the `mockAuctioneerAgent` does not send an "Auction-End" inform message, the `AuctionEndInformServer` behaviour will be blocked and is put into the blocked behaviours queue by the agent scheduler. When a new message (INFORM) is inserted in the agent message queue, the blocked behaviour becomes available for execution again, so that it has a chance to process the received message.

**A.3.6 Pattern Implementation**

**A.3.6.1 AuctioneerAgent**

The mock agent `MockBidderAgent` has a plan (test scenario) represented by six JADE Behaviors called `AuctionStartInformServer`, `CallForProposalsServer`, `BidRejectProposalServer`, `BidAcceptProposalServer`, `BidPriceRequestServer` and `AuctionEndInformServer` to test the AUT agent `AuctioneerAgent`. To execute the AUT agent test case `AuctioneerAgentTestCase`, all we need is to create a subclass of `JADETestCase` and to implement a test method `testAuctioning()` that creates an instance of the AUT `AuctioneerAgent`, and an instance of the mock agent `MockBidderAgent`. After that, this test method waits until their interaction finishes and asks the mock agent about the test result.

**A.3.6.2 BidderAgent**

The mock agent `MockAuctioneerAgent` has a plan (test scenario) represented by a JADE Behavior called `AuctionOrganizer` to test the AUT agent `BidderAgent`. To execute the AUT agent test case `BidderAgentTestCase`, all we need is to create a subclass of `JADETestCase` and to implement a test method `testBidding()` that creates an instance of the AUT `BidderAgent`, and an instance of the mock agent `MockAuctioneerAgent`. After that, this test method waits until their interaction finishes and asks the mock agent about the test result.

**A.4 Monitor Pattern**

**A.4.1 Design Pattern Template**

Table A.10 represents the Monitor design pattern template.

Template	Description
----------	-------------

<b>Element</b>	
<b>Name</b>	Monitor Pattern
<b>Intent</b>	This pattern monitors subjects of interest for state changes and notifies agents of their occurrence.
<b>Applicability</b>	The monitor pattern is suitable if an indefinite number of agents are dependent on a subject of interest or it is desirable that the subject of interest be decoupled from the dependent agents who are to be notified of its changes.
<b>Motivation</b> <b>Example</b>	Maintaining consistency in a distributed system is a challenge. Tight coupling preserves consistency, but compromises decoupling and reusability. In this pattern, agents which must remain current with respect to a subject advise the monitor of interest in the subject by subscribing for notification of particular changes to that subject. The monitor then requests notification of state changes from the subject. When the subject changes it notifies the monitor of its new state information. The monitor then notifies subscriber agents for whom the state change is relevant. This arrangement maintains consistency whilst decoupling monitor, subscribers and subjects.
<b>Participants</b>	<p>This coordination pattern involves at least one monitor, a number of subscriber agents and at least one subject or event of interest.</p> <p>The role of the monitor is to accept subscriptions, request notifications from subjects of interest, receive such notifications of events and to alert subscribers to relevant events. The role of the subject is to provide notifications of state changes as requested.</p> <p>The role of the subscriber is to subscribe for notifications of state changes to distributed subjects (data or objects), receive notifications with current state information, and update its local state information.</p> <p>A subject may have many subscribers, each of which may express interest in different state changes. Subscribers are notified when their event or state change of interest occurs. The notification includes all state information required to update the subscriber and thereby maintain consistency with the subject state information.</p>

Table A.10: Monitor Design Pattern Template

#### A.4.2 Collaboration diagram

Figure A.11 represents the collaboration diagram of the Monitor pattern (58).

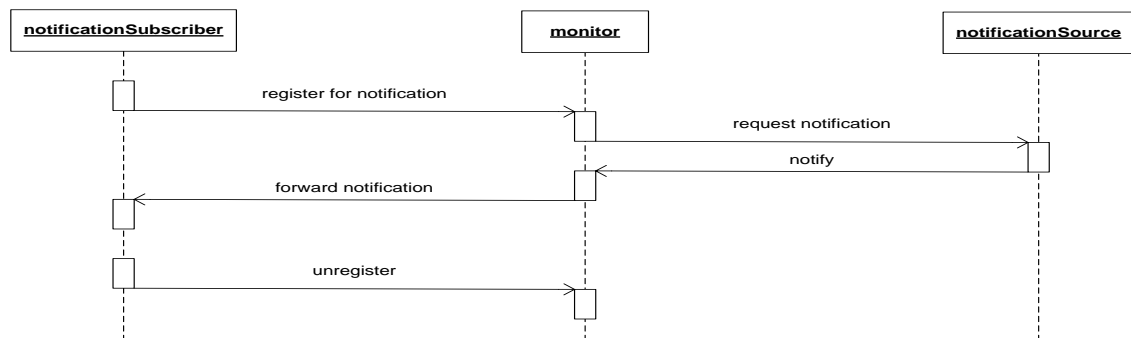


Figure A.11: The Collaboration diagram of the Monitor pattern

### A.4.3 Structure Diagram

Figure A.12 represents the structure diagram of the Monitor pattern.

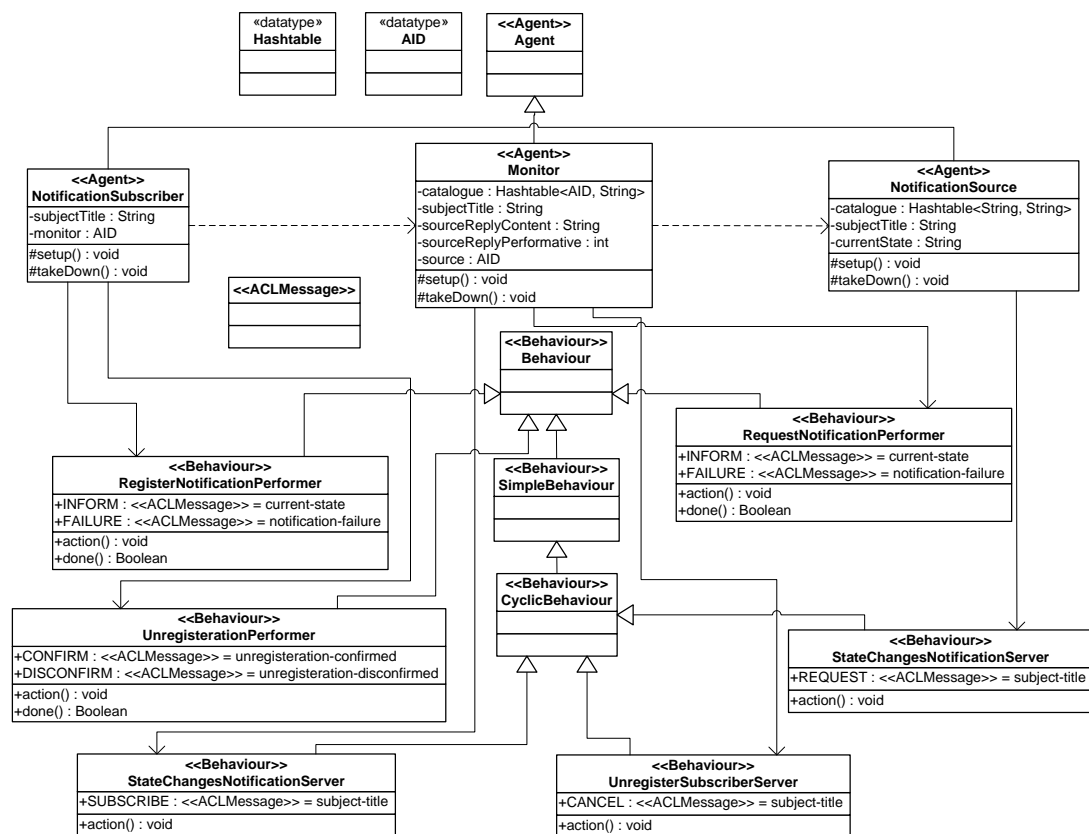


Figure A.12: The Structure diagram of the Monitor pattern

### A.4.4 Behaviour Diagram

Figure A.13 represents the behaviour diagram of the Monitor pattern.



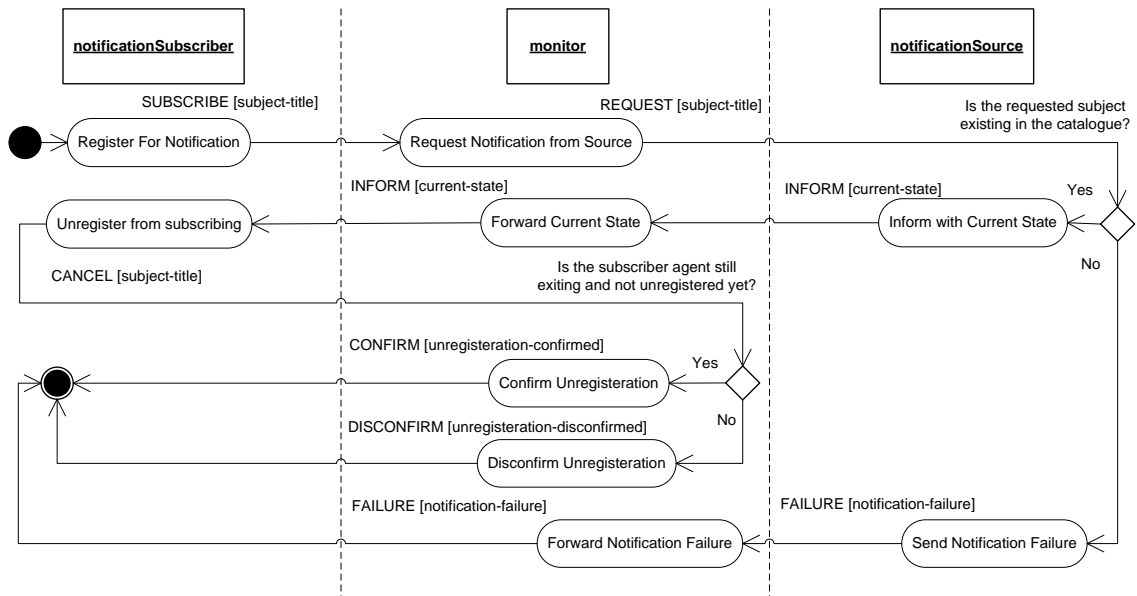


Figure A.13: The Behaviour diagram of the Monitor pattern

## A.4.5 Test Design Patterns

### A.4.5.1 NotificationSubscriberAgent

Table A.11, represents the design of test design pattern of the Agent Under Test.

Agent	NotificationSubscriberAgent
Roles	NotificationSubscriber
Interacting Roles	Monitor
Successful Scenario	<pre> sequenceDiagram     participant NS as notificationSubscriberAgent     participant MM as mockMonitorAgent      NS-&gt;&gt;MM: SUBSCRIBE [subject-title]     MM-&gt;&gt;NS: Perform the notification request from the source agent     NS-&gt;&gt;MM: INFORM [current-state]     MM-&gt;&gt;NS: Forward Current State     NS-&gt;&gt;MM: CANCEL [subject-title]     MM-&gt;&gt;NS: Is the subscriber agent still exiting and not unregistered yet?     NS-&gt;&gt;MM: Confirm Unregistration     MM-&gt;&gt;NS: Disconfirm Unregistration   </pre>
Exceptional Scenarios	<b>First Exceptional Scenario</b> <pre> sequenceDiagram     participant NS as notificationSubscriberAgent     participant MM as mockMonitorAgent      NS-&gt;&gt;MM: SUBSCRIBE [subject-title]     MM-&gt;&gt;NS: Perform the notification request from the source agent     NS-&gt;&gt;MM: FAILURE [notification-failure]     MM-&gt;&gt;NS: Forward Notification Failure   </pre>

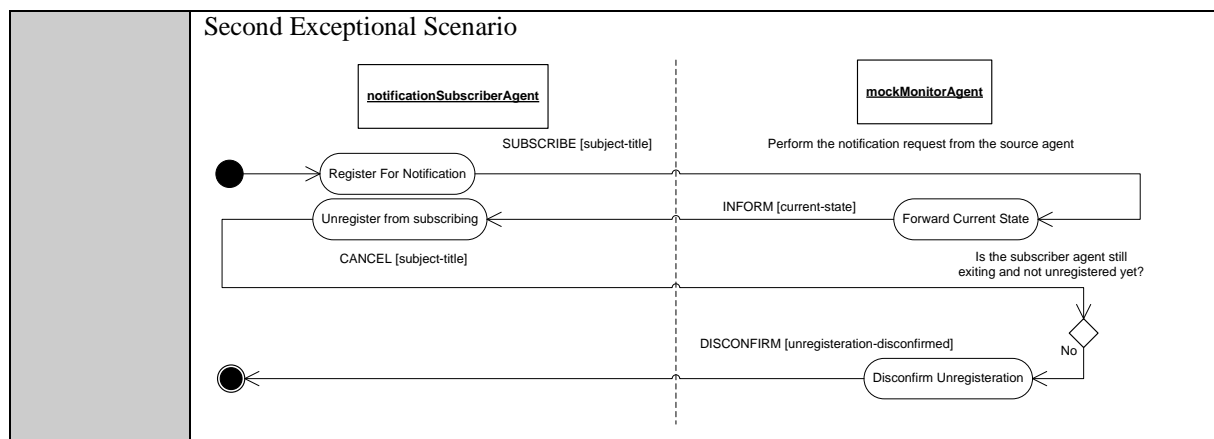


Table A.11: Test Design Pattern of the NotificationSubscriberAgent

When the `mockMonitorAgent` receives a notification registration subscribe message from the `notificationSubscriberAgent`, it replies with a message characterized by either:

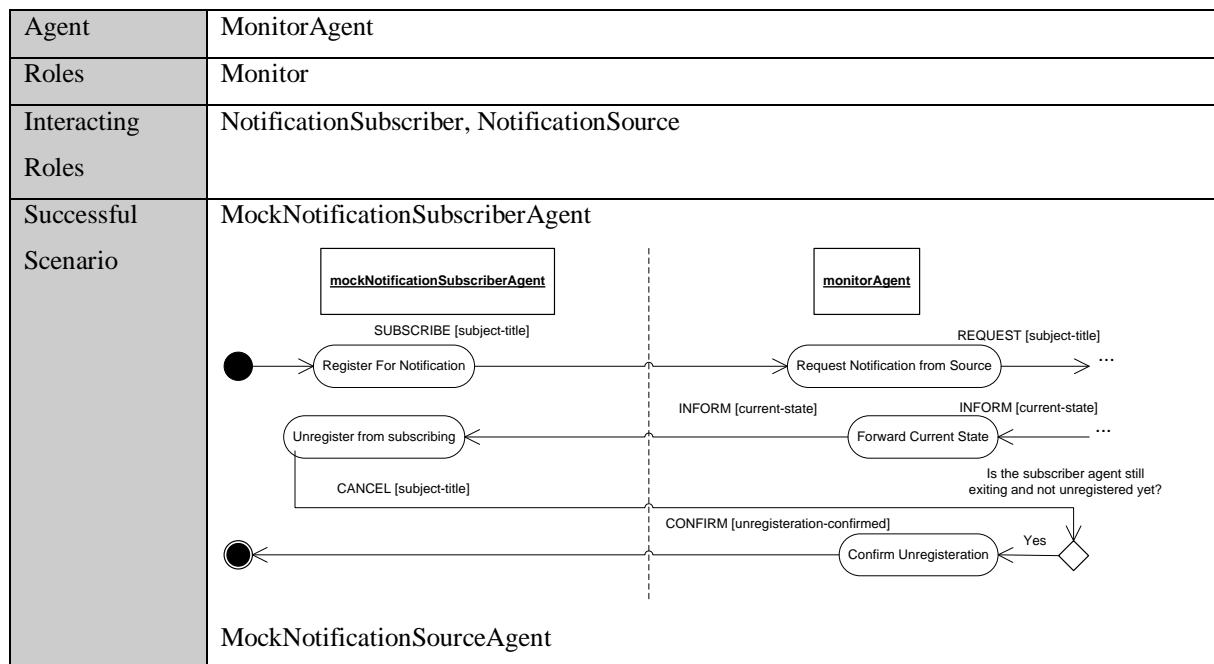
- i. An `INFORM` performative, indicating not only that the subject is available but also the subject current state, in this case the `notificationSubscriberAgent` will send notification unregistration to the `mockMonitorAgent` (normal scenario).
- ii. A `FAILURE` performative, indicating that the subject is unavailable for state notification, in this case the `notificationSubscriberAgent` will terminate (first exceptional scenario).
- iii. No `RESPONSE`, indicating that the `mockMonitorAgent` does not respond to the subscribe message, in this case the `RegisterNotificationPerformer` behaviour will be blocked and is put into the blocked behaviours queue by the agent scheduler. When a new message (`INFORM/FAILURE`) is inserted in the agent message queue, the blocked behaviour becomes available for execution again, so that it has a chance to process the received message.

When the `mockMonitorAgent` receives the notification unregistration cancel message from the `notificationSubscriberAgent`, it replies with a message characterized by either:

- i. A CONFIRM performative, indicating that notificationSubscriberAgent is still exiting and not unregistered yet, in this case the registration is successfully canceled and the notificationSubscriberAgent can terminate (normal scenario).
- ii. A DISCONFIRM performative, indicating that notificationSubscriberAgent is not currently registered to be notified by this subject, in this case the notificationSubscriberAgent will terminate (second exceptional scenario).
- iii. No RESPONSE, indicating that the mockMonitorAgent does not respond to the notification unregistration cancel message, in this case the UnregistrationPerformer behaviour will be blocked and is put into the blocked behaviours queue by the agent scheduler. When a new message (CONFIRM/DISCONFIRM) is inserted in the agent message queue, the blocked behaviour becomes available for execution again, so that it has a chance to process the received message.

#### A.4.5.2 MonitorAgent

Table A.12, represents the design of test design pattern of the Agent Under Test.



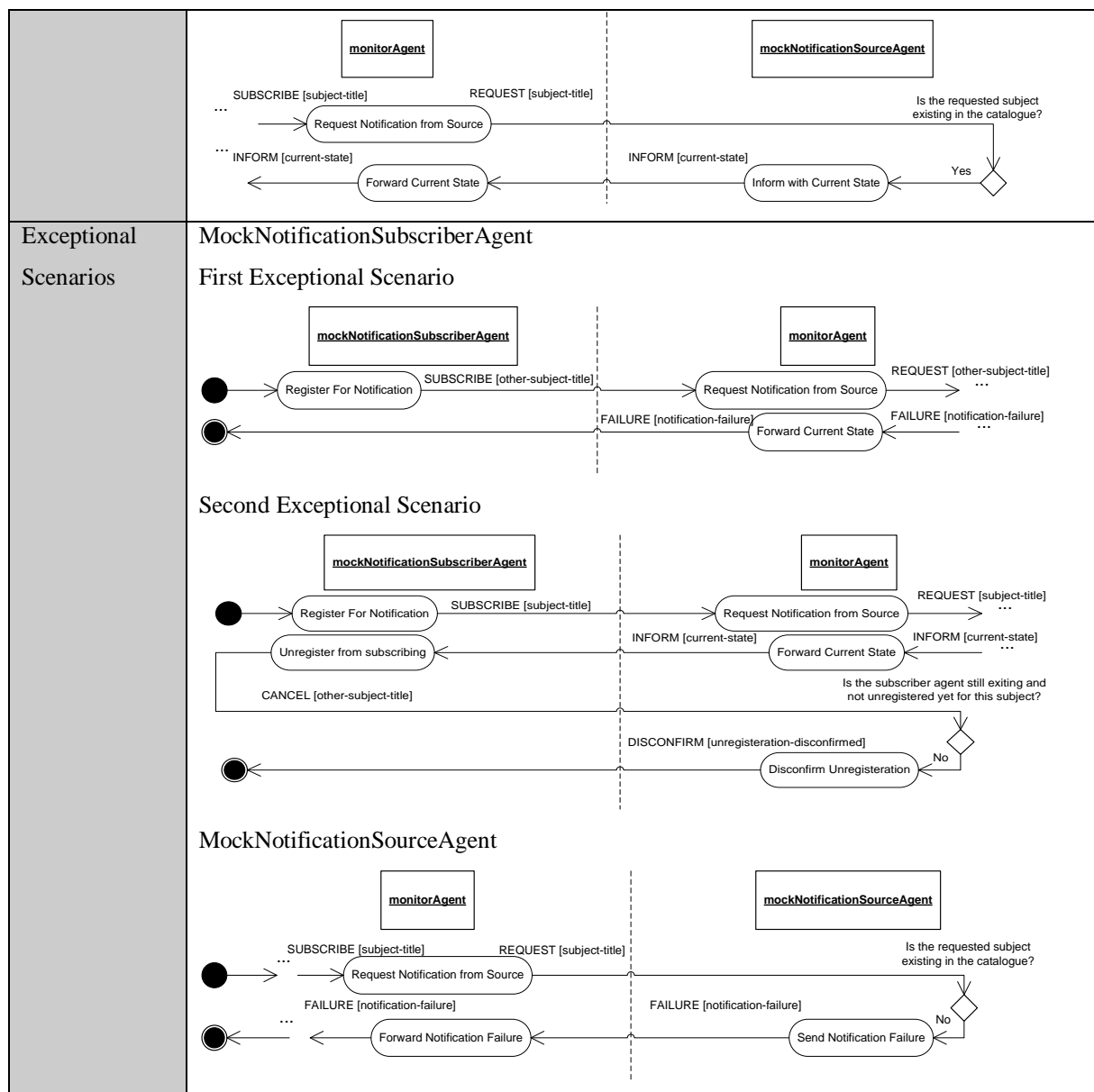


Table A.12: Test Design Pattern of the MonitorAgent

When the `mockNotificationSubscriberAgent` joins the environment, it sends a message characterized by a `SUBSCRIBE` performative indicating the willingness to be notified by a specific subject state from the `monitorAgent`. In this case, if the subject is unavailable, the `monitorAgent` replies with a `FAILURE` message (first exceptional scenario).

If the `mockNotificationSubscriberAgent` does not send a subscribe message, the `StateChangesNotificationServer` behaviour will be blocked and is put into the

blocked behaviours queue by the agent scheduler. When a new message (SUBSCRIBE) is inserted in the agent message queue, the blocked behaviour becomes available for execution again, so that it has a chance to process the received message.

When the `mockNotificationSubscriberAgent` finishes the registration time period, it sends a message characterized by a CANCEL performative indicating the willingness to cancel subject notification registration from the `monitorAgent`. In this case, if the `mockNotificationSubscriberAgent` is not currently registered to be notified by this subject, the `monitorAgent` replies with a DISCONFIRM message (second exceptional scenario).

If the `mockNotificationSubscriberAgent` does not send a notification unregistration cancel message, the `UnregisterSubscriberServer` behaviour will be blocked and is put into the blocked behaviours queue by the agent scheduler. When a new message (CANCEL) is inserted in the agent message queue, the blocked behaviour becomes available for execution again, so that it has a chance to process the received message.

When the `mockNotificationSourceAgent` receives a subject current state request message from the `monitorAgent`, it replies with a message characterized by either:

- i. An INFORM performative, indicating not only that the subject is available but also the subject current state, in this case the `monitorAgent` will forward the subject current state to the `mockNotificationSubscriberAgent` (normal scenario).
- ii. A FAILURE performative, indicating that the subject is unavailable for state notification, in this case the `monitorAgent` will forward the notification failure to the `notificationSubscriberAgent` (`MockNotificationSourceAgent` exceptional scenario).
- iii. No RESPONSE, indicating that the `mockNotificationSourceAgent` does not respond to the request message, in this case the `RequestNotificationPerformer`

behaviour will be blocked and is put into the blocked behaviours queue by the agent scheduler. When a new message (INFORM/FAILURE) is inserted in the agent message queue, the blocked behaviour becomes available for execution again, so that it has a chance to process the received message.

#### A.4.5.3 NotificationSourceAgent

Table A.13, represents the design of test design pattern of the Agent Under Test.

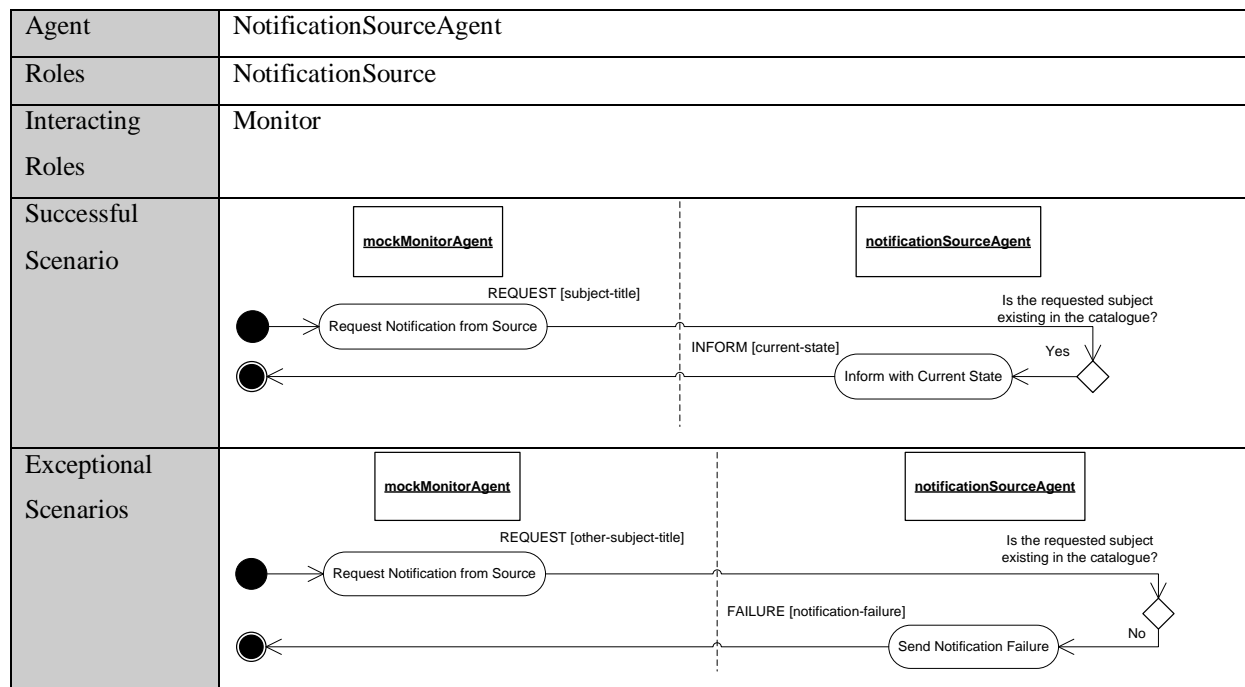


Table A.13: Test Design Pattern of the NotificationSourceAgent

When the `mockMonitorAgent` joins the environment, it sends a message characterized by a `REQUEST` performative indicating the willingness to be informed by a specific subject state from the `notificationSourceAgent`. In this case, if the subject is unavailable, the `notificationSourceAgent` replies with a `FAILURE` message (exceptional scenario).

If the `mockMonitorAgent` does not send a request message, the `StateChangesNotificationServer` behaviour will be blocked and is put into the blocked behaviours queue by the agent scheduler. When a new message (`REQUEST`) is inserted in the agent message queue, the blocked behaviour becomes available for execution again, so that it has a chance to process the received message.

## **A.4.6 Pattern Implementation**

### **A.4.6.1 NotificationSubscriberAgent**

The mock agent `MockMonitorAgent` has a plan (test scenario) represented by two JADE Behaviors called `StateChangesNotificationServer` and `UnregisterSubscriberServer` to test the AUT agent `NotificationSubscriberAgent`. To execute the AUT agent test case `NotificationSubscriberAgentTestCase`, all we need is to create a subclass of `JADETestCase` and to implement a test method `testNotificationSubscriber()` that creates an instance of the AUT `NotificationSubscriberAgent`, and an instance of the mock agent `MockMonitorAgent`. After that, this test method waits until their interaction finishes and asks the mock agent about the test result.

### **A.4.6.2 MonitorAgent**

The mock agent `MockNotificationSubscriberAgent` has a plan (test scenario) represented by two JADE Behaviors called `RegisterNotificationPerformer` and `UnregistrationPerformer` to test the AUT agent `MonitorAgent`. To execute the AUT agent test case `MonitorAgentTestCase`, all we need is to create a subclass of `JADETestCase` and to implement a test method `testMonitoring()` that creates an instance of the AUT `MonitorAgent`, and an instance of the mock agent `MockNotificationSubscriberAgent`. After that, this test method waits until their interaction finishes and asks the mock agent about the test result.

The mock agent `MockNotificationSourceAgent` has a plan (test scenario) represented by a JADE Behavior called `StateChangesNotificationServer` to test the AUT agent `MonitorAgent`. To execute the AUT agent test case `MonitorAgentTestCase`, all we need is to create a subclass of `JADETestCase` and to implement a test method `testMonitoring()` that creates an instance of the AUT `MonitorAgent`, and an instance of the mock agent `MockNotificationSourceAgent`. After that, this test method waits until their interaction finishes and asks the mock agent about the test result.

### **A.4.6.3 NotificationSourceAgent**

The mock agent `MockMonitorAgent` has a plan (test scenario) represented by a JADE Behavior called `RequestNotificationPerformer` to test the AUT agent `NotificationSourceAgent`. To execute the AUT agent test case `NotificationSourceAgentTestCase`, all we need is to create a subclass of `JADETestCase` and to implement a test method `testNotificationSource()` that creates an instance of the AUT `NotificationSourceAgent`, and an instance of the mock agent `MockMonitorAgent`. After that, this test method waits until their interaction finishes and asks the mock agent about the test result.

## A.5 Matchmaker Pattern

### A.5.1 Design Pattern Template

Table A.14 represents the Matchmaker design pattern template.

Template Element	Description
<b>Name</b>	Matchmaker Pattern
<b>Intent</b>	To locate a provider for a given service requested by a client and letting the client directly interact with the provider.
<b>Applicability</b>	Use when an agent (client) needs to directly interact with another agent (provider) to use its services but does not know what agent offers the desired service.
<b>Motivation Example</b>	<p>An agent (client) may need a specific service provided by another unknown agent (provider). An intermediary agent (matchmaker) can find the provider agent which offers the requested service and give the provider identification to the client, which can then directly interact with it.</p> <p>For example, a reporter agent needs to interact with a news agency to obtain news about a specific subject. The news agency to be contacted is going to be known at run time by a matchmaker agent which works as a yellow page informing clients (i.e., the reporter) the identification of the agent (i.e., the news agency) that provides the requested service (i.e., the news).</p>
<b>Participants</b>	The Client requests the identification of an agent that provides a specific service. The Matchmaker finds the Provider of the requested service and gives its identification to the Client. The Provider must subscribe the Matchmaker yellow page service in order to be found by the clients requesting its services.

Table A.14: Matchmaker Design Pattern Template



### A.5.2 Collaboration diagram

Figure A.14 represents the collaboration diagram of the Matchmaker pattern (7).

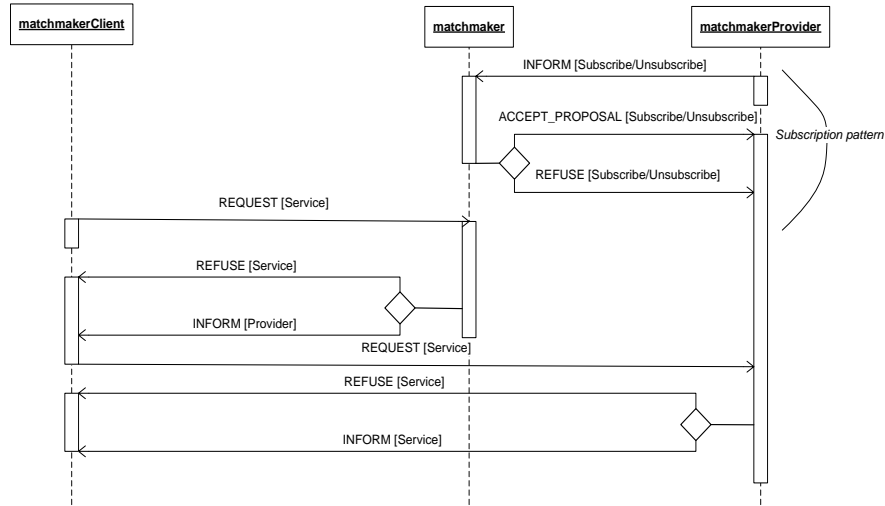


Figure A.14: The Collaboration diagram of the Matchmaker pattern

### A.5.3 Structure Diagram

Figure A.15 represents the structure diagram of the Matchmaker pattern.

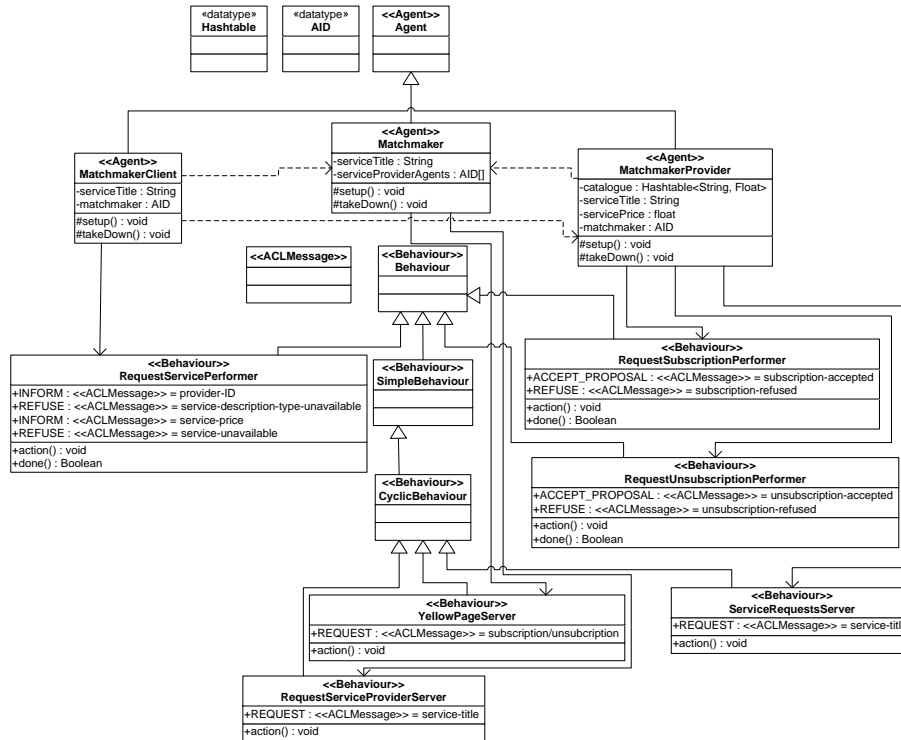


Figure A.15: The Structure diagram of the Matchmaker pattern

## A.5.4 Behaviour Diagram

Figure A.16 represents the behaviour diagram of the Matchmaker pattern (7).

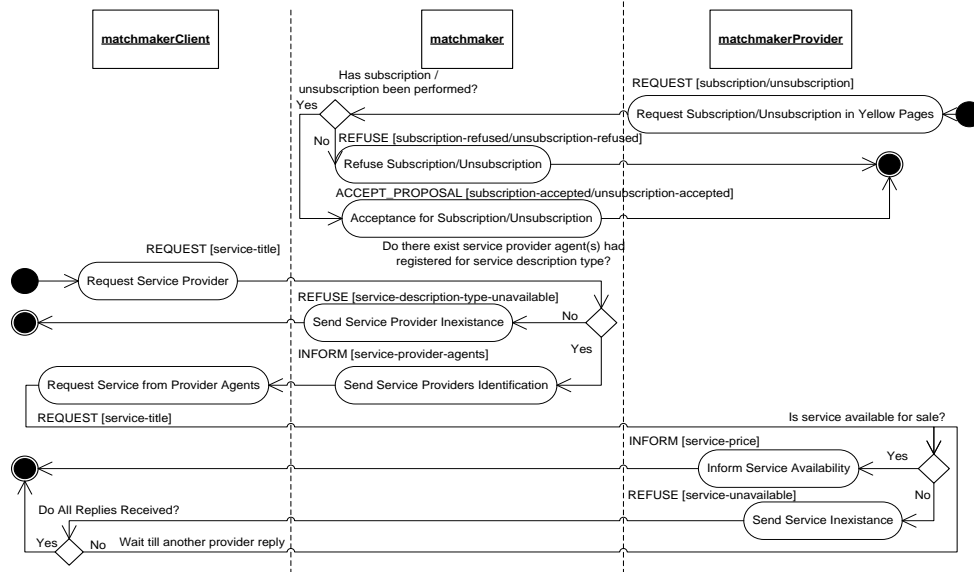


Figure A.16: The Behaviour diagram of the Matchmaker pattern

## A.5.5 Test Design Patterns

### A.5.5.1 MatchmakerClientAgent

Table A.15, represents the design of test design pattern of the Agent Under Test.

Agent	MatchmakerClientAgent
Roles	MatchmakerClient
Interacting Roles	Matchmaker, MatchmakerProvider
Successful Scenario	<p><b>MockMatchmakerAgent</b></p> <pre> sequenceDiagram     participant matchmakerClientAgent     participant mockMatchmakerAgent      matchmakerClientAgent-&gt;&gt;mockMatchmakerAgent: REQUEST [service-title]     mockMatchmakerAgent-&gt;&gt;mockMatchmakerAgent: Do there exist service provider agent(s) had registered for service description type?     mockMatchmakerAgent-&gt;&gt;mockMatchmakerAgent: Yes     mockMatchmakerAgent-&gt;&gt;matchmakerClientAgent: Send Service Providers Identification     matchmakerClientAgent-&gt;&gt;mockMatchmakerAgent: Request Service from Provider Agents     matchmakerClientAgent-&gt;&gt;mockMatchmakerAgent: REQUEST [service-title]     </pre> <p><b>MockMatchmakerProviderAgent</b></p> <pre> sequenceDiagram     participant matchmakerClientAgent     participant mockMatchmakerProviderAgent      matchmakerClientAgent-&gt;&gt;mockMatchmakerProviderAgent: REQUEST [service-title]     mockMatchmakerProviderAgent-&gt;&gt;mockMatchmakerProviderAgent: Is service available for sale?     mockMatchmakerProviderAgent-&gt;&gt;mockMatchmakerProviderAgent: Yes     mockMatchmakerProviderAgent-&gt;&gt;matchmakerClientAgent: Inform Service Availability     matchmakerClientAgent-&gt;&gt;mockMatchmakerProviderAgent: INFORM [service-price]     </pre>

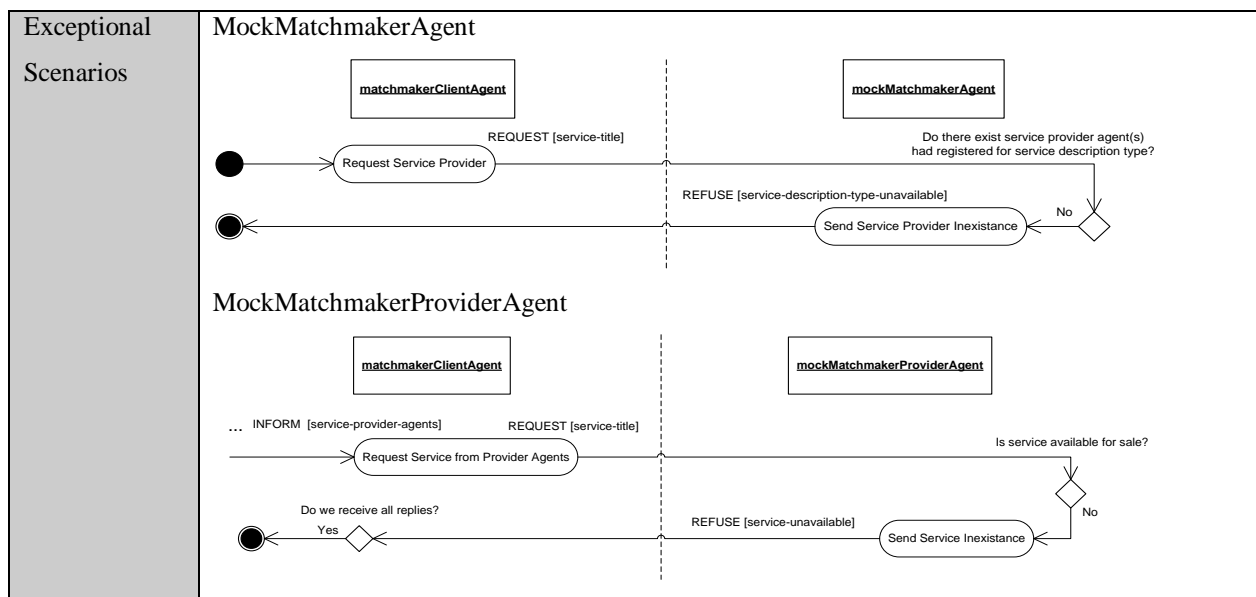


Table A.15: Test Design Pattern of the MatchmakerClientAgent

When the `mockMatchmakerAgent` receives the "service-providing" request message from the `matchmakerClientAgent`, it replies with a message characterized by either:

- An INFORM performative, indicating not only that the service providers have been located but also the service providers identification (normal scenario).
- A REFUSE message, indicating that no `serviceProviderAgents` registered with the `mockMatchmakerAgent` having the service available yet (MockMatchmakerAgent exceptional scenario).
- No RESPONSE, indicating that the `mockMatchmakerAgent` does not respond to the "service-providing" request message, in this case the `RequestServicePerformer` behaviour will be blocked and is put into the blocked behaviours queue by the agent scheduler. When a new message (INFORM/REFUSE) is inserted in the agent message queue, the blocked behaviour becomes available for execution again, so that it has a chance to process the received message.

When the `mockMatchmakerProviderAgent` receives a request message from the `matchmakerClientAgent`, it replies with a message characterized by either:

- i. An INFORM performative, indicating not only that the service is available but also the service price, in this case the `matchmakerClientAgent` will terminate (normal scenario).
- ii. A REFUSE performative, indicating that the service is unavailable, in this case the `matchmakerClientAgent` will terminate (`MockMatchmakerProviderAgent` exceptional scenario).
- iii. No RESPONSE, indicating that the `mockMatchmakerProviderAgent` does not respond to the request message, in this case the `RequestServicePerformer` behaviour will be blocked and an event is fired to notify its parent behaviour. The behaviour is put into the blocked behaviours queue by the agent scheduler. When a new message (INFORM/REFUSE) is inserted in the agent message queue, the blocked behaviour becomes available for execution again, so that it has a chance to process the received message.

#### A.5.5.2 MatchmakerAgent

Table A.16, represents the design of test design pattern of the Agent Under Test.

Agent	MatchmakerAgent
Roles	Matchmaker
Interacting Roles	MatchmakerClient, MatchmakerProvider
Successful Scenario	<p>MockMatchmakerClientAgent</p> <pre> sequenceDiagram     participant Client as mockMatchmakerClientAgent     participant Provider as matchmakerAgent     Note over Client, Provider: Message Boundary     Client-&gt;&gt;Provider: REQUEST [service-title]     activate Provider     Provider-&gt;&gt;Provider: Do there exist service provider agent(s) had registered for service description type?     Note over Provider: Yes     Provider-&gt;&gt;Client: INFORM [service-provider-agents]     Provider-&gt;&gt;Client: Send Service Providers Identification     deactivate Provider     </pre>
Exceptional Scenarios	<p>MockMatchmakerClientAgent</p> <pre> sequenceDiagram     participant Client as mockMatchmakerClientAgent     participant Provider as matchmakerAgent     Note over Client, Provider: Message Boundary     Client-&gt;&gt;Provider: REQUEST [other-service-title]     activate Provider     Provider-&gt;&gt;Provider: Do there exist service provider agent(s) had registered for service description type?     Note over Provider: No     Provider-&gt;&gt;Client: REFUSE [service-description-type-unavailable]     Provider-&gt;&gt;Client: Send Service Provider Inexistence     deactivate Provider     </pre>

Table A.16: Test Design Pattern of the MatchmakerAgent

The exceptional scenario of `mockMatchmakerClientAgent` testing `matchmakerAgent` is similar to the one in the broker pattern section 4.3.5.2 Broker Agent (`mockBrokerClientAgent` testing `brokerAgent`).

For the subscription/unsubscription normal and exceptional scenarios of `mockMatchmakerProviderAgent` testing `matchmakerAgent`, please review the subscription pattern Appendix A.1.5.1 SubscriptionYellowPageAgent.

### A.5.5.3 MatchmakerProviderAgent

Table A.17, represents the design of test design pattern of the Agent Under Test.

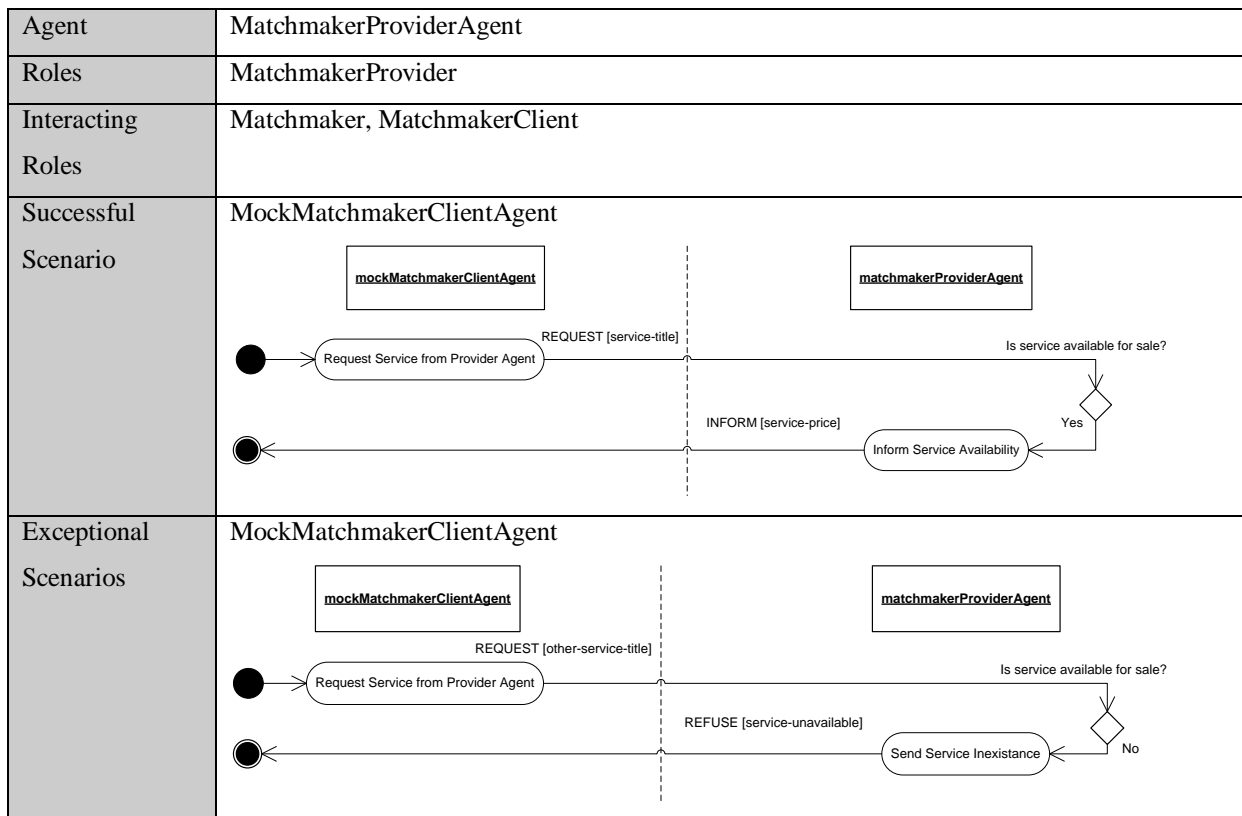


Table A.17: Test Design Pattern of the MatchmakerProviderAgent

For the subscription/unsubscription normal and exceptional scenarios of the `mockMatchmakerAgent` testing `matchmakerProviderAgent`, please review the subscription pattern Appendix A.1.5.2 SubscriptionServiceProviderAgent.

When the `mockMatchmakerClientAgent` joins the environment, it sends a message characterized by a REQUEST performative indicating the willingness to be informed by a

specific service price from the `matchmakerProviderAgent`. In this case, if the service is unavailable, the `matchmakerProviderAgent` replies with a `REFUSE` message (`MockMatchmakerClientAgent` exceptional scenario).

If the `mockMatchmakerClientAgent` does not send a request message, the `ServiceRequestsServer` behaviour will be blocked and is put into the blocked behaviours queue by the agent scheduler. When a new message (`REQUEST`) is inserted in the agent message queue, the blocked behaviour becomes available for execution again, so that it has a chance to process the received message.

### **A.5.6 Pattern Implementation**

#### **A.5.6.1 MatchmakerClientAgent**

The mock agent `MockMatchmakerAgent` has a plan (test scenario) represented by a JADE Behavior called `RequestServiceProviderServer` to test the AUT agent `MatchmakerClient`. To execute the AUT agent test case `MatchmakerClientAgentTestCase`, all we need is to create a subclass of `JADETestCase` and to implement a test method `testMatchmakerClient()` that creates an instance of the AUT `MatchmakerClientAgent`, and an instance of the mock agent `MockMatchmakerAgent`. After that, this test method waits until their interaction finishes and asks the mock agent about the test result.

The mock agent `MockMatchmakerProviderAgent` has a plan (test scenario) represented by a JADE Behavior called `ServiceRequestsServer` to test the AUT agent `MatchmakerClientAgent`. To execute the AUT agent test case `MatchmakerClientAgentTestCase`, all we need is to create a subclass of `JADETestCase` and to implement a test method `testMatchmakerClient()` that creates an instance of the AUT `MatchmakerClientAgent`, and an instance of the mock agent `MockMatchmakerProviderAgent`. After that, this test method waits until their interaction finishes and asks the mock agent about the test result.

#### **A.5.6.2 MatchmakerAgent**

The mock agent `MockMatchmakerClientAgent` has a plan (test scenario) represented by a JADE Behavior called `RequestServiceProviderPerformer` to test the AUT agent `MatchmakerAgent`. To execute the AUT agent test case `MatchmakerAgentTestCase`, all we need is to create a subclass of `JADETestCase` and to implement a test method `testMatchmaker()` that creates an instance of the AUT `MatchmakerAgent`, and an instance of the mock agent `MockMatchmakerClientAgent`. After that, this test method waits until their interaction finishes and asks the mock agent about the test result.

The mock agent `MockMatchmakerProviderAgent` has a plan (test scenario) represented by two JADE Behaviors called `RequestSubscriptionPerformer` and `RequestUnsubscriptionPerformer` to test the AUT agent `MatchmakerAgent`. To execute the AUT agent test case `MatchmakerAgentTestCase`, all we need is to create a subclass of `JADETestCase` and to implement a test method `testMatchmaker()` that creates an instance of the AUT `MatchmakerAgent`, and an instance of the mock agent `MockMatchmakerProviderAgent`. After that, this test method waits until their interaction finishes and asks the mock agent about the test result.

#### **A.5.6.3 MatchmakerProviderAgent**

The mock agent `MockMatchmakerAgent` has a plan (test scenario) represented by a JADE Behavior called `YellowPageServer` to test the AUT agent `MatchmakerProviderAgent`. To execute the AUT agent test case `MatchmakerProviderAgentTestCase`, all we need is to create a subclass of `JADETestCase` and to implement a test method `testMatchmakerProvider()` that creates an instance of the AUT `MatchmakerProviderAgent`, and an instance of the mock agent `MockMatchmakerAgent`. After that, this test method waits until their interaction finishes and asks the mock agent about the test result.

The mock agent `MockMatchmakerClientAgent` has a plan (test scenario) represented by a JADE Behavior called `RequestServicePerformer` to test the AUT agent `MatchmakerProviderAgent`. To execute the AUT agent test case `MatchmakerProviderAgentTestCase`, all we need is to create a subclass of

JADETestCase and to implement a test method `testMatchmakerProvider()` that creates an instance of the AUT `MatchmakerProviderAgent`, and an instance of the mock agent `MockMatchmakerClientAgent`. After that, this test method waits until their interaction finishes and asks the mock agent about the test result.

## A.6 Mediator Pattern

### A.6.1 Design Pattern Template

Table A.18 represents the Mediator design pattern template.

Template Element	Description
<b>Name</b>	Mediator Pattern
<b>Intent</b>	This pattern enables the encapsulation of agent interactions, i.e. the specification of how a set of agents interact. The nature of the interactions can then be maintained at only one point, the mediator.
<b>Applicability</b>	A mediator pattern is suitable if interactions between the agents are complex, but well-defined; customization of distributed behaviors is desired; or agent reuse is hampered because the agent communicates with too many other agents.
<b>Motivation Example</b>	<p>If behaviors are distributed among agents, many interactions are possible. Agents may need to hold both self models and acquaintance models of all other agents to support all possible interactions. Efficiency is reduced as interconnections proliferate, as agents become reliant on too many other agents.</p> <p>The mediator pattern encapsulates collective behavior, allowing control and coordination of the interactions of a group of agents. It introduces a layer of abstraction, preventing agents from referring to each other explicitly, and allowing their interactions to be varied independently.</p> <p>This promotes loose coupling and reduces the number of interconnections directly between the agents, since they only need to know about the mediator. Maintenance of behaviors is also easier when localized in a mediator, as opposed to behaviors which are distributed.</p>
<b>Participants</b>	This coordination pattern involves a mediator, and any number of colleague agents. The mediator coordinates the cooperative behavior of the colleagues and has acquaintance models of all colleague agents. The colleagues each have an acquaintance model of the mediator. A colleague agent addresses the mediator in place of another colleague.

Table A.18: Mediator Design Pattern Template



### A.6.2 Collaboration diagram

Figure A.17 represents the collaboration diagram of the Mediator pattern (58).

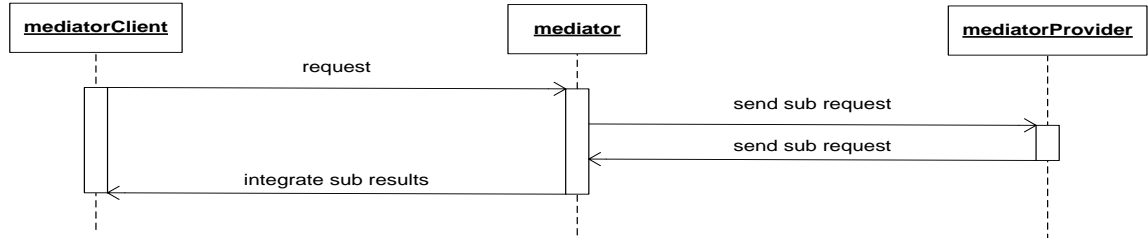


Figure A.17: The Collaboration diagram of the Mediator pattern

### A.6.3 Structure Diagram

Figure A.18 represents the structure diagram of the Mediator pattern.

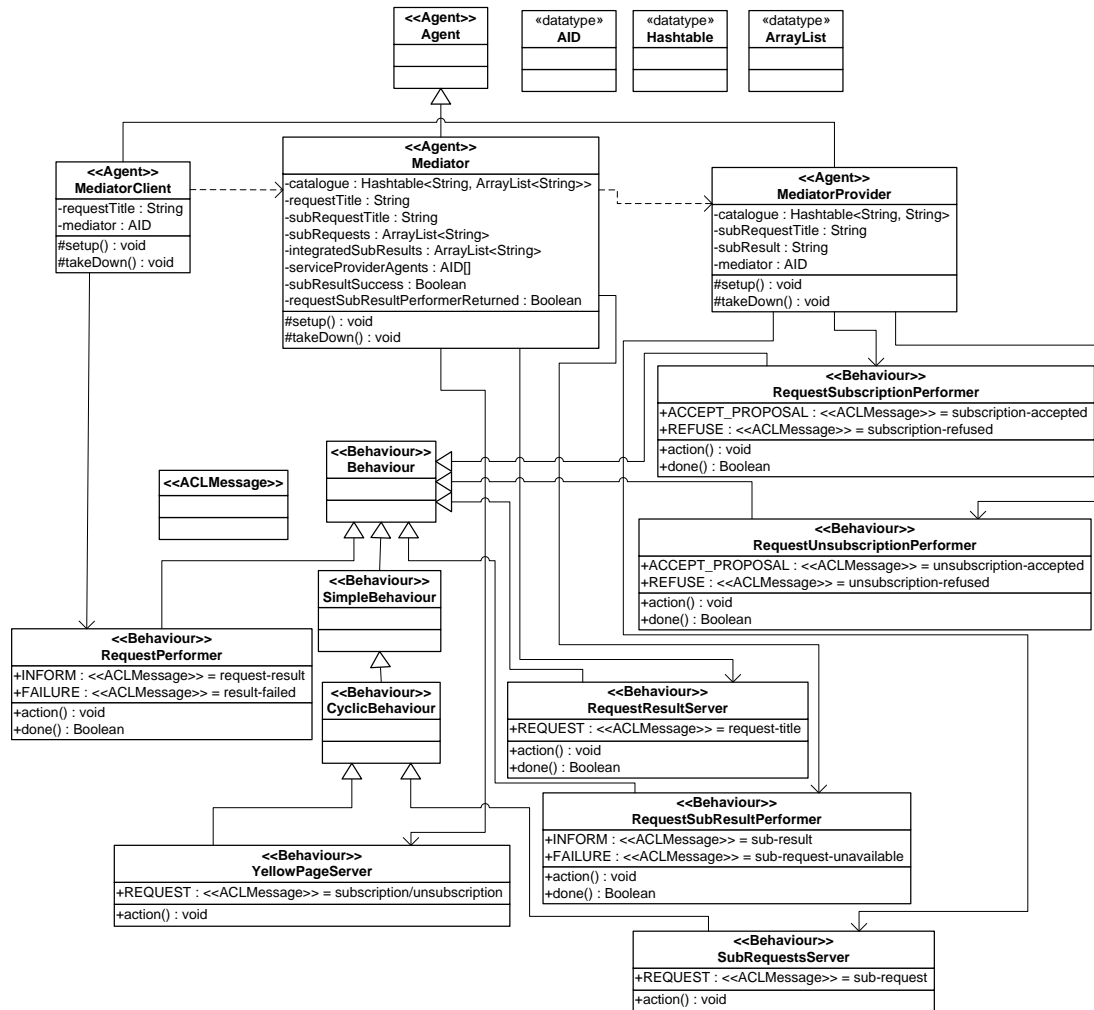


Figure A.18: The Structure diagram of the Mediator pattern

## A.6.4 Behaviour Diagram

Figure A.19 represents the behaviour diagram of the Mediator pattern.

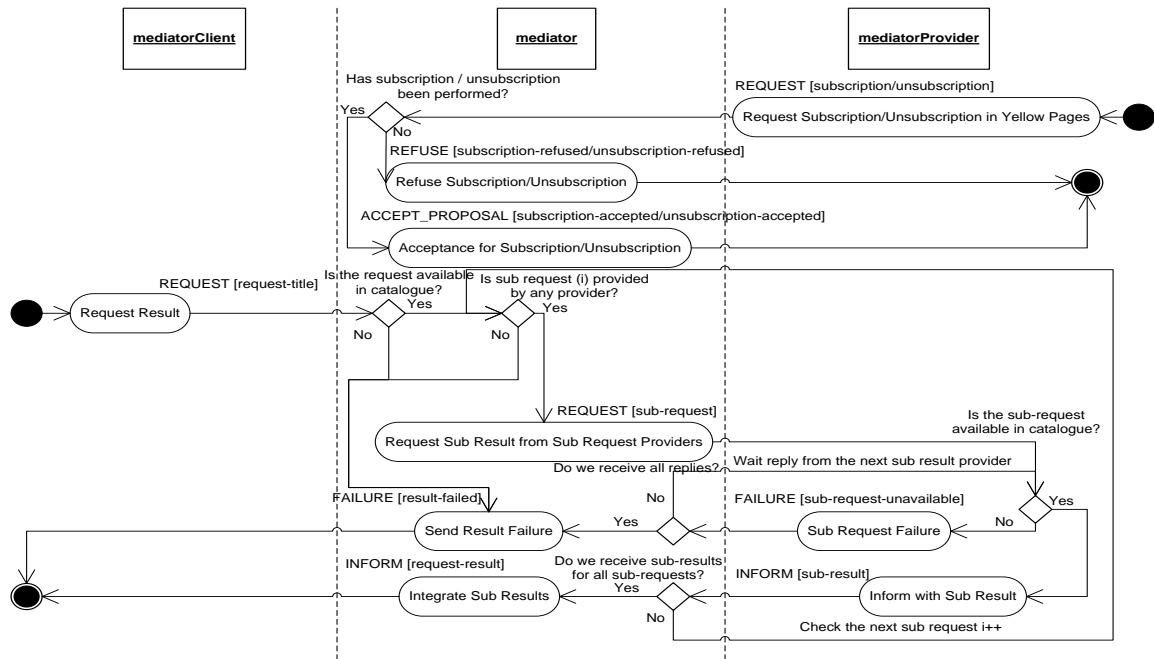


Figure A.19: The Behaviour diagram of the Mediator pattern

## A.6.5 Test Design Patterns

### A.6.5.1 MediatorClientAgent

Table A.19, represents the design of test design pattern of the Agent Under Test.

Agent	MediatorClientAgent
Roles	MediatorClient
Interacting Roles	Mediator
Successful Scenario	<pre> sequenceDiagram     participant MC as mediatorClientAgent     participant MM as mockMediatorAgent      MC-&gt;&gt;MC: Request Result     MC-&gt;&gt;MM: REQUEST [request-title]     MM-&gt;&gt;MM: Is the request available in catalogue?     MM-&gt;&gt;MM: Yes     MM-&gt;&gt;MC: INFORM [request-result]     MC-&gt;&gt;MC: Integrate Sub Results     </pre>
Exceptional Scenarios	<p>First Exceptional Scenario</p> <pre> sequenceDiagram     participant MC as mediatorClientAgent     participant MM as mockMediatorAgent      MC-&gt;&gt;MC: Request Result     MC-&gt;&gt;MM: REQUEST [request-title]     MM-&gt;&gt;MM: Is the request available in catalogue?     MM-&gt;&gt;MM: No     MM-&gt;&gt;MC: FAILURE [result-failed]     MC-&gt;&gt;MC: Send Result Failure     </pre>

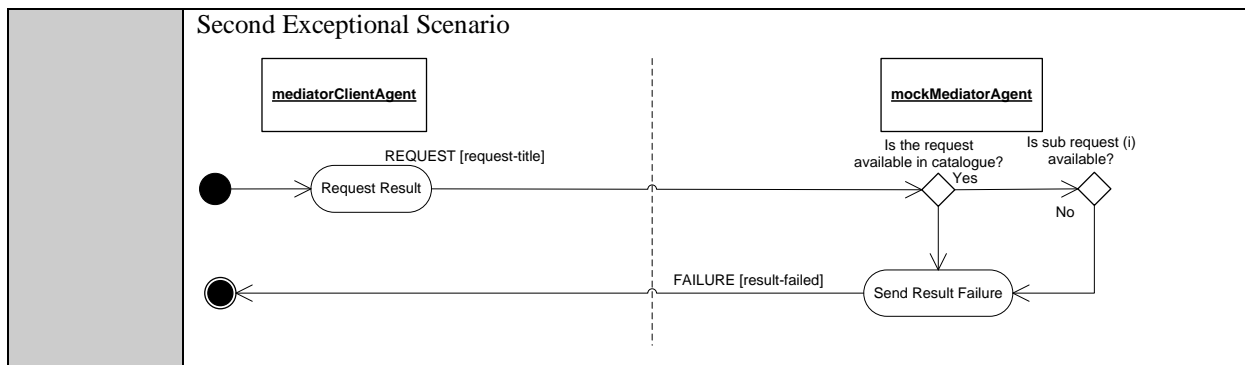


Table A.19: Test Design Pattern of the MediatorClientAgent

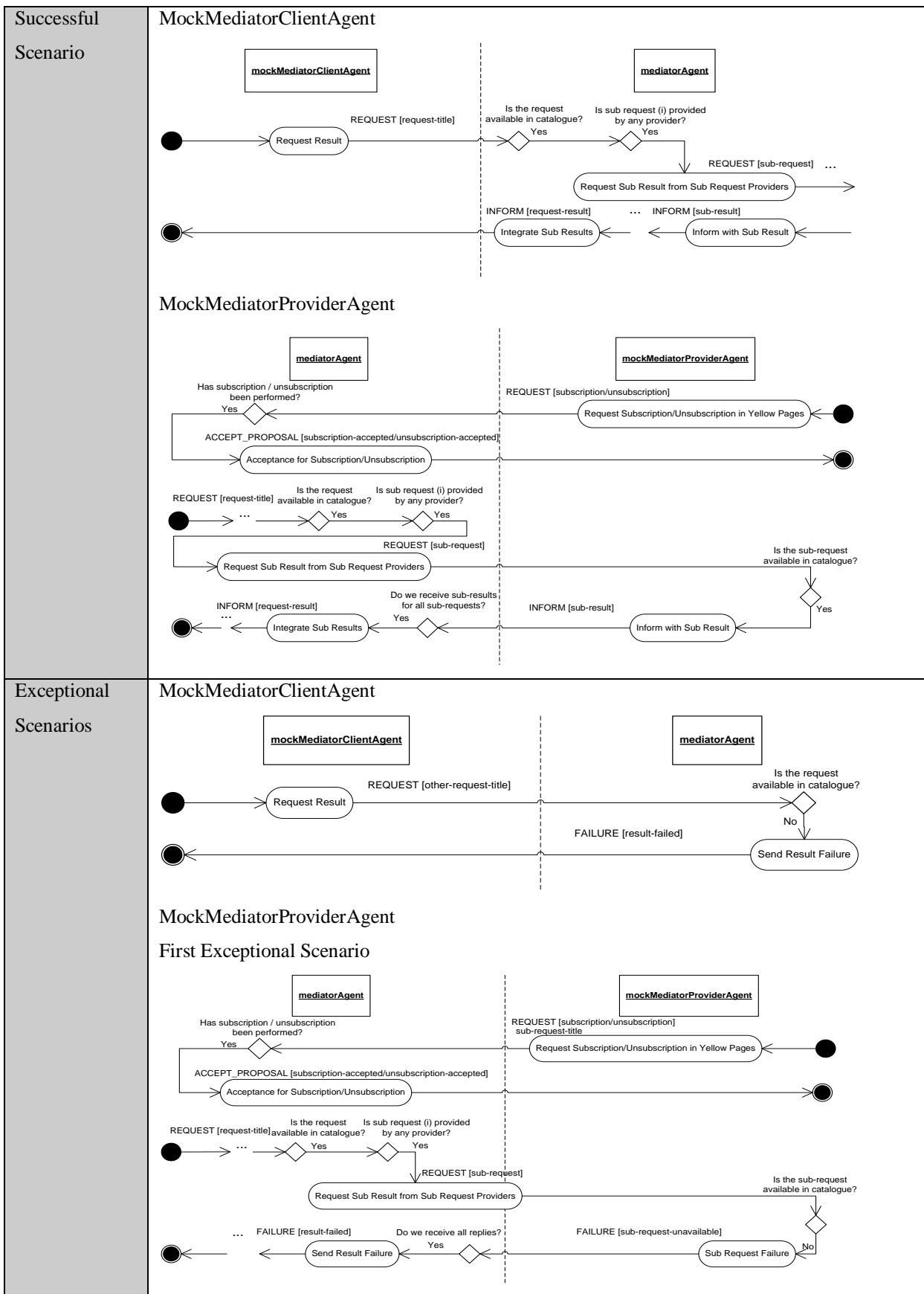
When the mockMediatorAgent receives a request message from the mediatorClientAgent, it replies with a message characterized by either:

- i. An INFORM performative, indicating not only that the request is available, but also there exist sub-results for all sub-requests, in this case the mediatorClientAgent will terminate (normal scenario).
- ii. A FAILURE performative, indicating that either the request is unavailable or there do not exist sub-results for all sub-requests, in this case the mediatorClientAgent will terminate (first and second exceptional scenarios).
- iii. No RESPONSE, indicating that the mockMediatorAgent does not respond to the request message, in this case the RequestPerformer behaviour will be blocked and is put into the blocked behaviours queue by the agent scheduler. When a new message (INFORM/FAILURE) is inserted in the agent message queue, the blocked behaviour becomes available for execution again, so that it has a chance to process the received message.

#### A.6.5.2 MediatorAgent

Table A.20, represents the design of test design pattern of the Agent Under Test.

Agent	MediatorAgent
Roles	Mediator
Interacting Roles	MediatorClient, MediatorProvider



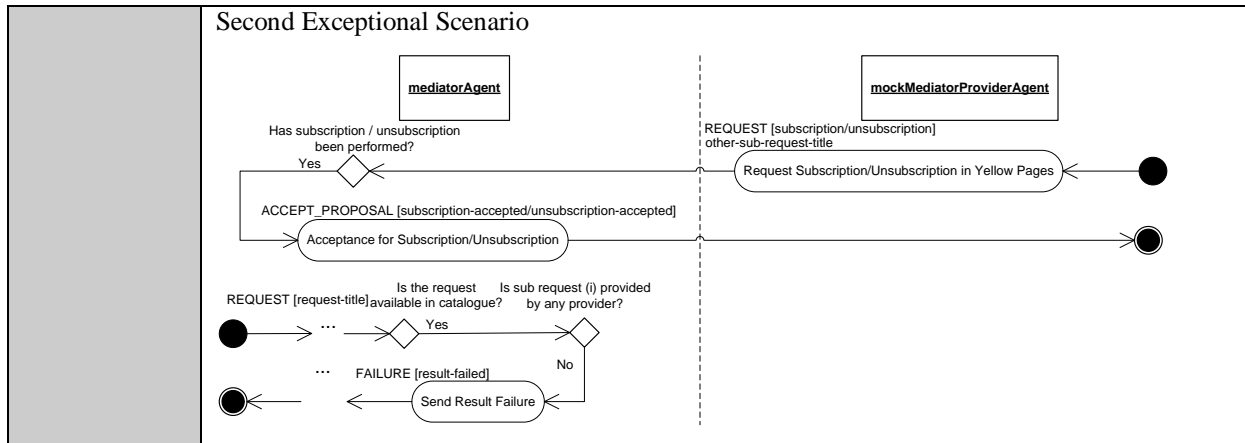


Table A.20: Test Design Pattern of the MediatorAgent

When the `mockMediatorClientAgent` joins the environment, it sends a message characterized by a `REQUEST` performative indicating the willingness to be informed by a specific request result from the `mediatorAgent`. In this case, if the request is unavailable, the `mediatorAgent` replies with a `FAILURE` message (`MockMediatorClientAgent` exceptional scenario).

If the `mockMediatorClientAgent` does not send a request message, the `RequestResultServer` behaviour will be blocked and is put into the blocked behaviours queue by the agent scheduler. When a new message (`REQUEST`) is inserted in the agent message queue, the blocked behaviour becomes available for execution again, so that it has a chance to process the received message.

For the subscription/unsubscription normal and exceptional scenarios of the `mockMediatorProviderAgent` testing `mediatorAgent`, please review the subscription pattern Appendix A.1.5.1 `SubscriptionYellowPageAgent`.

When the `mockMediatorProviderAgent` receives a request message from the `mediatorAgent`, it replies with a message characterized by either:

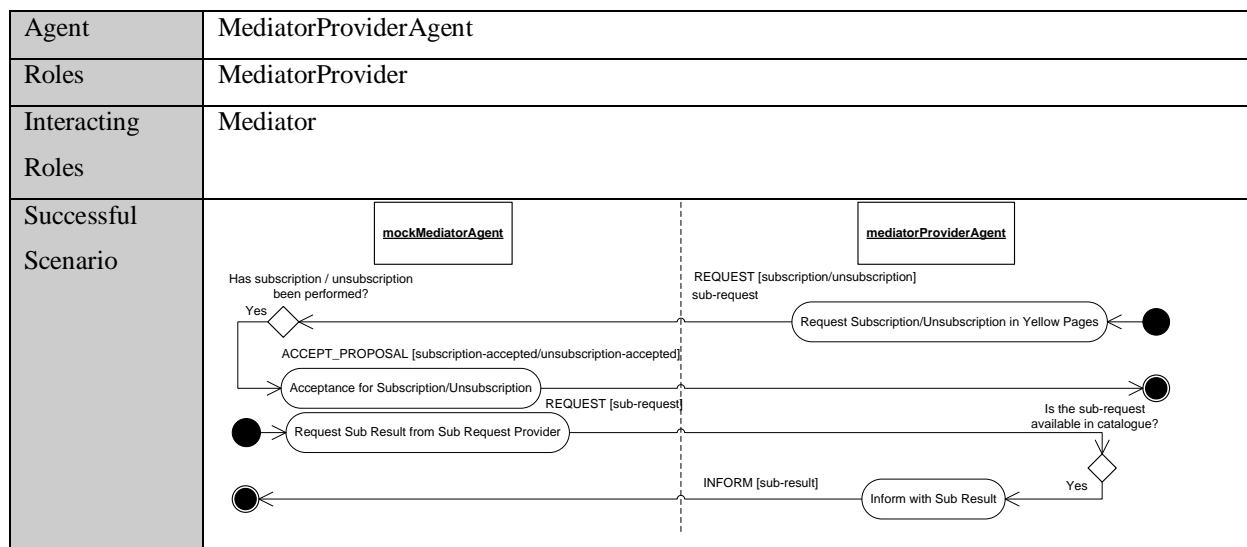
- i. An `INFORM` performative, indicating not only that the sub-request is available, but also the sub-result, in this case the `mediatorAgent` will append the sub-result to its

integratedSubResults list then starts searching for a sub-result for the next sub-request (normal scenario).

- ii. A FAILURE performative, indicating that the sub-request, for any reason, is no more available in the catalogue, in this case the mediatorAgent will wait for the inform/failure messages of the remaining known provider agents (if any), if all known provider agents send FAILURE messages, then the RequestSubResultPerformer behaviour will be done, and the control of execution will be returned to the caller RequestResultServer and the mediatorAgent will reply with a FAILURE message to the requester meditaor client agent (second exceptional scenario).
- iii. No RESPONSE, indicating that the mockMediatorProviderAgent does not respond to the request message, in this case the RequestSubResultPerformer behaviour will be blocked and is put into the blocked behaviours queue by the agent scheduler. When a new message (INFORM/FAILURE) is inserted in the agent message queue, the blocked behaviour becomes available for execution again, so that it has a chance to process the received message.

#### A.6.5.3 MediatorProviderAgent

Table A.21, represents the design of test design pattern of the Agent Under Test.



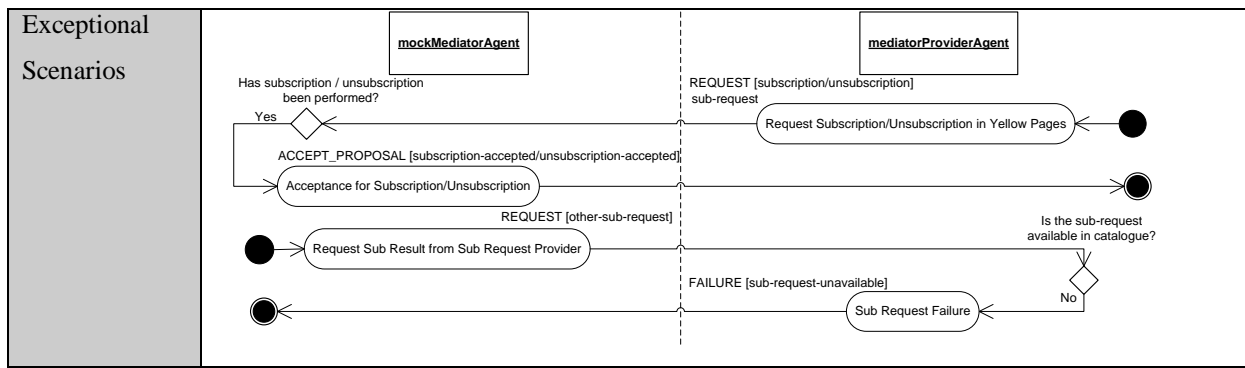


Table A.21: Test Design Pattern of the MediatorProviderAgent

For the subscription/unsubscription normal and exceptional scenarios of the `mockMediatorAgent` testing the `mediatorProviderAgent`, please review the subscription pattern Appendix A.1.5.2 `SubscriptionServiceProviderAgent`.

When the `mockMediatorAgent` receives the identification of all the agents registered as “service-providing” for the sub-request, it sends a message characterized by a `REQUEST` performative indicating the willingness to be informed by the sub-result. In this case, if the sub-request is unavailable the `mediatorProviderAgent` replies with a `FAILURE` message (exceptional scenario).

If the `mockMediatorAgent` does not send a request message, the `SubRequestsServer` behaviour will be blocked and is put into the blocked behaviours queue by the agent scheduler. When a new message (`REQUEST`) is inserted in the agent message queue, the blocked behaviour becomes available for execution again, so that it has a chance to process the received message.

## A.6.6 Pattern Implementation

### A.6.6.1 MediatorClientAgent

The mock agent `MockMediatorAgent` has a plan (test scenario) represented by a JADE Behavior called `RequestResultServer` to test the AUT agent `MediatorClientAgent`. To execute the AUT agent test case `MediatorClientAgentTestCase`, all we need is to create a subclass of `JADETestCase` and to implement a test method `testMediatorClient()` that creates an instance of the AUT `MediatorClientAgent`,

and an instance of the mock agent `MockMediatorAgent`. After that, this test method waits until their interaction finishes and asks the mock agent about the test result.

#### **A.6.6.2 MediatorAgent**

The mock agent `MockMediatorClientAgent` has a plan (test scenario) represented by a JADE Behavior called `RequestPerformer` to test the AUT agent `MediatorAgent`. To execute the AUT agent test case `MediatorAgentTestCase`, all we need is to create a subclass of `JADETestCase` and to implement a test method `testMediating()` that creates an instance of the AUT `MediatorAgent`, and an instance of the mock agent `MockMediatorClientAgent`. After that, this test method waits until their interaction finishes and asks the mock agent about the test result.

The mock agent `MockMediatorProviderAgent` has a plan (test scenario) represented by three JADE Behaviors called `RequestSubscriptionPerformer`, `SubRequestsServer` and `RequestUnsubscriptionPerformer` to test the AUT agent `MediatorAgent`. To execute the AUT agent test case `MediatorAgentTestCase`, all we need is to create a subclass of `JADETestCase` and to implement a test method `testMediating()` that creates an instance of the AUT `MediatorAgent`, and an instance of the mock agent `MockMediatorProviderAgent`. After that, this test method waits until their interaction finishes and asks the mock agent about the test result.

#### **A.6.6.3 MediatorProviderAgent**

The mock agent `MockMediatorAgent` has a plan (test scenario) represented by two JADE Behaviors called `YellowPageServer` and `RequestSubResultPerformer` to test the AUT agent `MediatorProviderAgent`. To execute the AUT agent test case `MediatorProviderAgentTestCase`, all we need is to create a subclass of `JADETestCase` and to implement a test method `testMediatorProvider()` that creates an instance of the AUT `MediatorProviderAgent`, and an instance of the mock agent `MockMediatorAgent`. After that, this test method waits until their interaction finishes and asks the mock agent about the test result.



## A.7 Embassy Pattern

### A.7.1 Design Pattern Template

Table A.22 represents the Embassy design pattern template.

<b>Template Element</b>	<b>Description</b>
<b>Name</b>	Embassy Pattern
<b>Intent</b>	Multi-agent systems may use an embassy to represent them, through which a foreign agent may communicate and gain access. The embassy provides access control, and so should feature in the design of a security mechanism for agents. The embassy may also allow translation and the establishment of communication protocols where the foreign agent is not homogeneous.
<b>Applicability</b>	If an application makes occasional reference to another application in the same environment, but it is not desirable to permanently integrate or interface these applications, an embassy is a suitable alternative. If the agents in both domains speak the same ACL, embassy facilitated interaction is possible. Otherwise more effort must be spent translating between ACLs.
<b>Motivation Example</b>	<p>An agent environment such as the Web, may comprise many different agent domains, such as an e-commerce agent, information retrieval agents and chatter bots, in the same Web space.</p> <p>These domains may be entirely independent or may wish to interact at some point e.g. an information agent wishes to query the commercial agent regarding a specific stock price. The embassy agent aims to bridge these separate agent domains, allowing interaction between heterogeneous agent applications.</p> <p>The embassy agent may need to translate different communication languages (ACLs) and content formats. Standardization and ubiquity of an ACL will make the embassy task easier.</p>
<b>Participants</b>	This coordination pattern involves at least one embassy, foreign and local/native agents.

Table A.22: Embassy Design Pattern Template

### A.7.2 Collaboration diagram

Figure A.20 represents the collaboration diagram of the Embassy pattern (59).

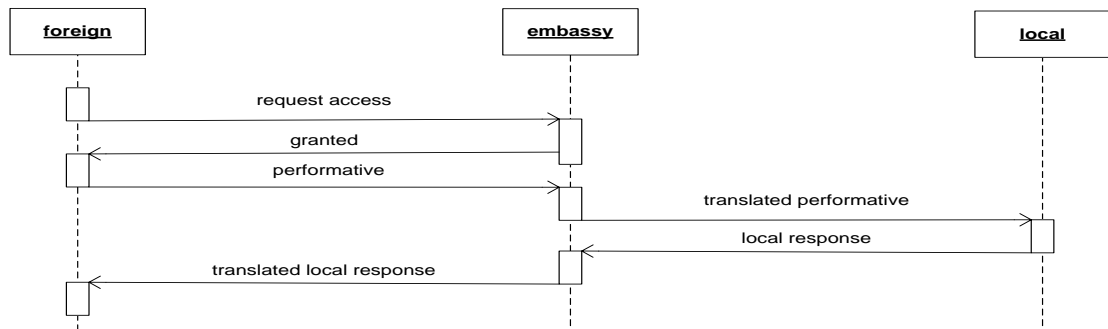


Figure A.20: The Collaboration diagram of the Embassy pattern

### A.7.3 Structure Diagram

Figure A.21 represents the structure diagram of the Embassy pattern.

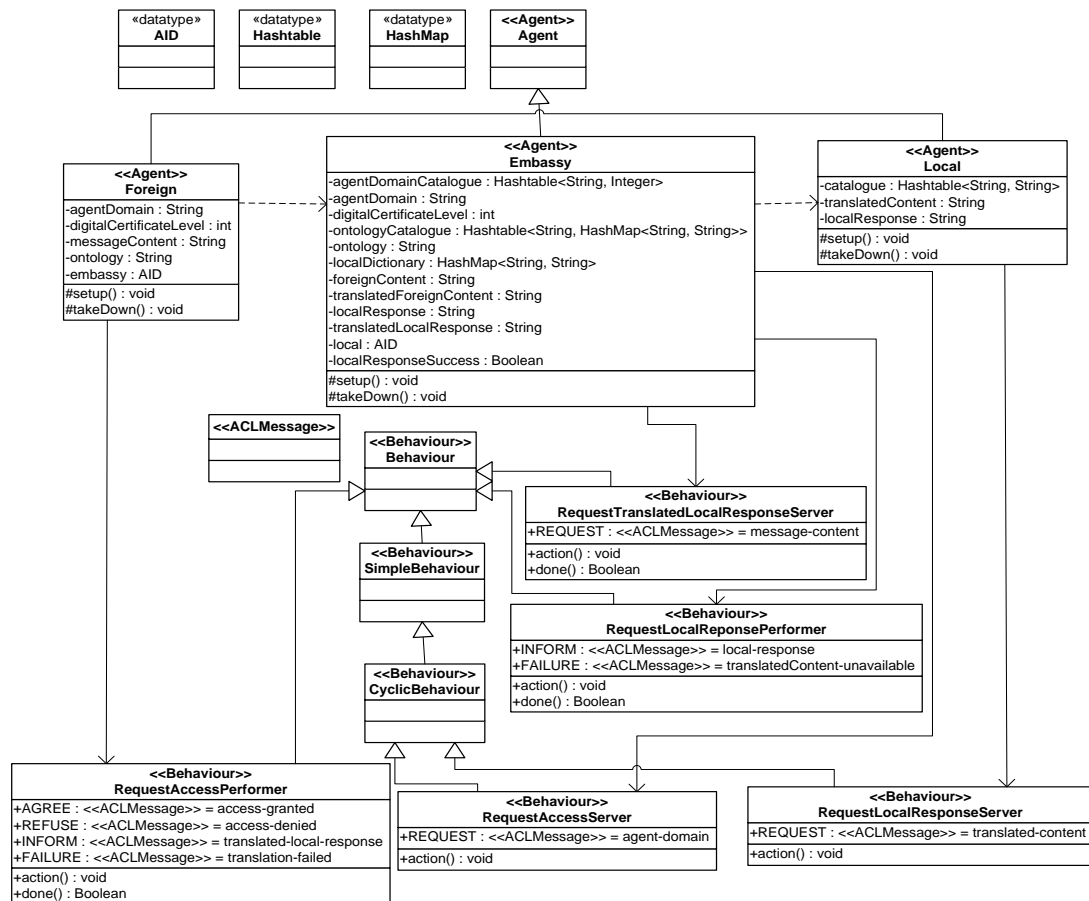


Figure A.21: The Structure diagram of the Embassy pattern

### A.7.4 Behaviour Diagram

Figure A.22 represents the behaviour diagram of the Embassy pattern.

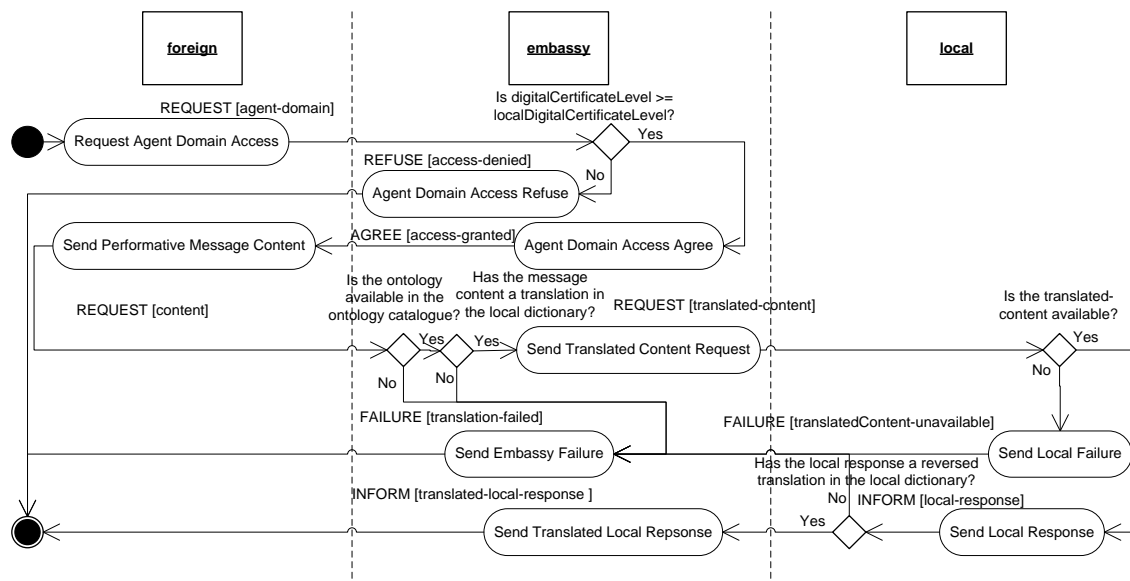


Figure A.22: The Behaviour diagram of the Embassy pattern

## A.7.5 Test Design Patterns

### A.7.5.1 ForeignAgent

Table A.23, represents the design of test design pattern of the Agent Under Test.

Agent	ForeignAgent
Roles	Foreign
Interacting Roles	Embassy
Successful Scenario	<pre> sequenceDiagram     participant foreignAgent     participant MockEmbassyAgent      foreignAgent-&gt;&gt;MockEmbassyAgent: REQUEST [agent-domain]     activate MockEmbassyAgent     MockEmbassyAgent-&gt;&gt;MockEmbassyAgent: Is digitalCertificateLevel &gt;= localDigitalCertificateLevel?     MockEmbassyAgent--&gt;&gt;foreignAgent: AGREE [access-granted]     deactivate MockEmbassyAgent     foreignAgent-&gt;&gt;MockEmbassyAgent: REQUEST [content]     activate MockEmbassyAgent     MockEmbassyAgent-&gt;&gt;MockEmbassyAgent: Is the ontology available in the ontology catalogue?     MockEmbassyAgent--&gt;&gt;foreignAgent: INFROM [translated-local-response]     deactivate MockEmbassyAgent     foreignAgent-&gt;&gt;MockEmbassyAgent: Send Translated Local Repsonse     activate MockEmbassyAgent     MockEmbassyAgent-&gt;&gt;MockEmbassyAgent: Has the local response a reversed translation in the local dictionary?     MockEmbassyAgent--&gt;&gt;foreignAgent: Send Local Response     deactivate MockEmbassyAgent     deactivate foreignAgent   </pre>
Exceptional Scenarios	<b>First Exceptional Scenario</b> <pre> sequenceDiagram     participant foreignAgent     participant mockEmbassyAgent      foreignAgent-&gt;&gt;mockEmbassyAgent: REQUEST [agent-domain]     activate mockEmbassyAgent     mockEmbassyAgent-&gt;&gt;mockEmbassyAgent: Is digitalCertificateLevel &gt;= localDigitalCertificateLevel?     mockEmbassyAgent--&gt;&gt;foreignAgent: REFUSE [access-denied]     deactivate mockEmbassyAgent     foreignAgent-&gt;&gt;mockEmbassyAgent: Agent Domain Access Refuse     activate mockEmbassyAgent     deactivate mockEmbassyAgent     deactivate foreignAgent   </pre>

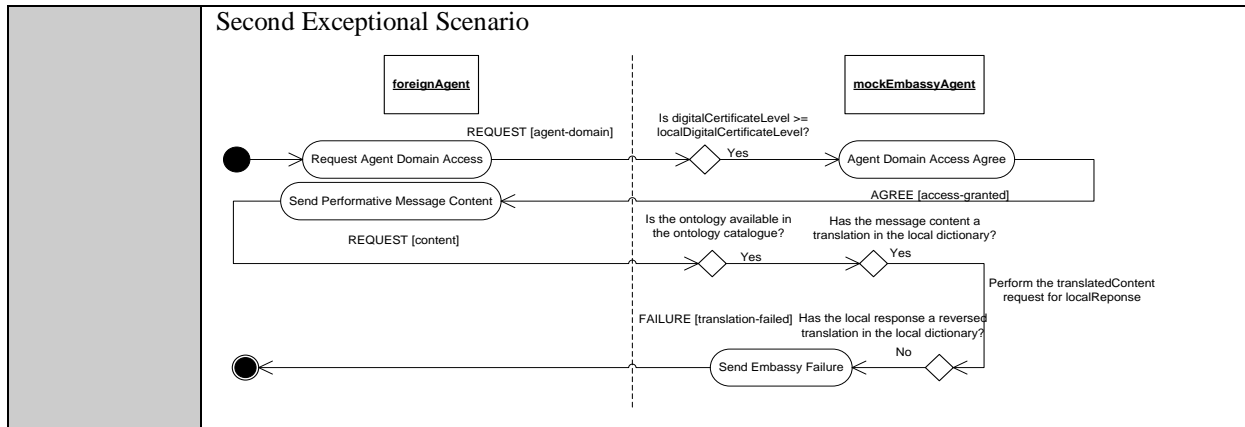


Table A.23: Test Design Pattern of the ForeignAgent

When the `mockEmbassyAgent` receives an agent-domain access request message from the `foreignAgent`, it replies with a message characterized by either:

- i. An AGREE performative, indicating not only that the agent-domain is supported, but also the `foreignAgent` digital certificate level is enough to access this agent-domain, in this case the `foreignAgent` will send the content of the performative message to the `mockEmbassyAgent` to be translated in accordance with a standard ontology (normal scenario).
- ii. A REFUSE performative, indicating that either the agent-domain is not supported or the `foreignAgent` digital certificate level is not enough to access this agent-domain, in this case the `foreignAgent` will terminate (first exceptional scenario).
- iii. No RESPONSE, indicating that the `mockEmbassyAgent` does not respond to the agent-domain access request message, in this case the `RequestAccessPerformer` behaviour will be blocked and is put into the blocked behaviours queue by the agent scheduler. When a new message (AGREE/REFUSE) is inserted in the agent message queue, the blocked behaviour becomes available for execution again, so that it has a chance to process the received message.

When the `mockEmbassyAgent` receives a translated local response request message from the `foreignAgent`, it replies with a message characterized by either:

- i. An INFORM performative, indicating that the ontology is available in the ontologyCatalogue, the message content has a translation in the localDictionary, a local response is available and having a reversed translation in the localDictionary, in this case the foreignAgent will terminate (normal scenario).
- ii. A FAILURE performative, indicating that either the ontology is unavailable in the ontologyCatalogue, or the message content has no translation in the localDictionary, or a local response is unavailable, or has no reversed translation in the localDictionary, in this case the foreignAgent will terminate (second exceptional scenario).
- iii. No RESPONSE, indicating that the mockEmbassyAgent does not respond to the translated local response request message, in this case the RequestAccessPerformer behaviour will be blocked and is put into the blocked behaviours queue by the agent scheduler. When a new message (INFORM/FAILURE) is inserted in the agent message queue, the blocked behaviour becomes available for execution again, so that it has a chance to process the received message.

#### A.7.5.2 EmbassyAgent

Table A.24, represents the design of test design pattern of the Agent Under Test.

Agent	EmbassyAgent
Roles	Embassy
Interacting Roles	Foreign, Local
Successful Scenario	MockForeignAgent



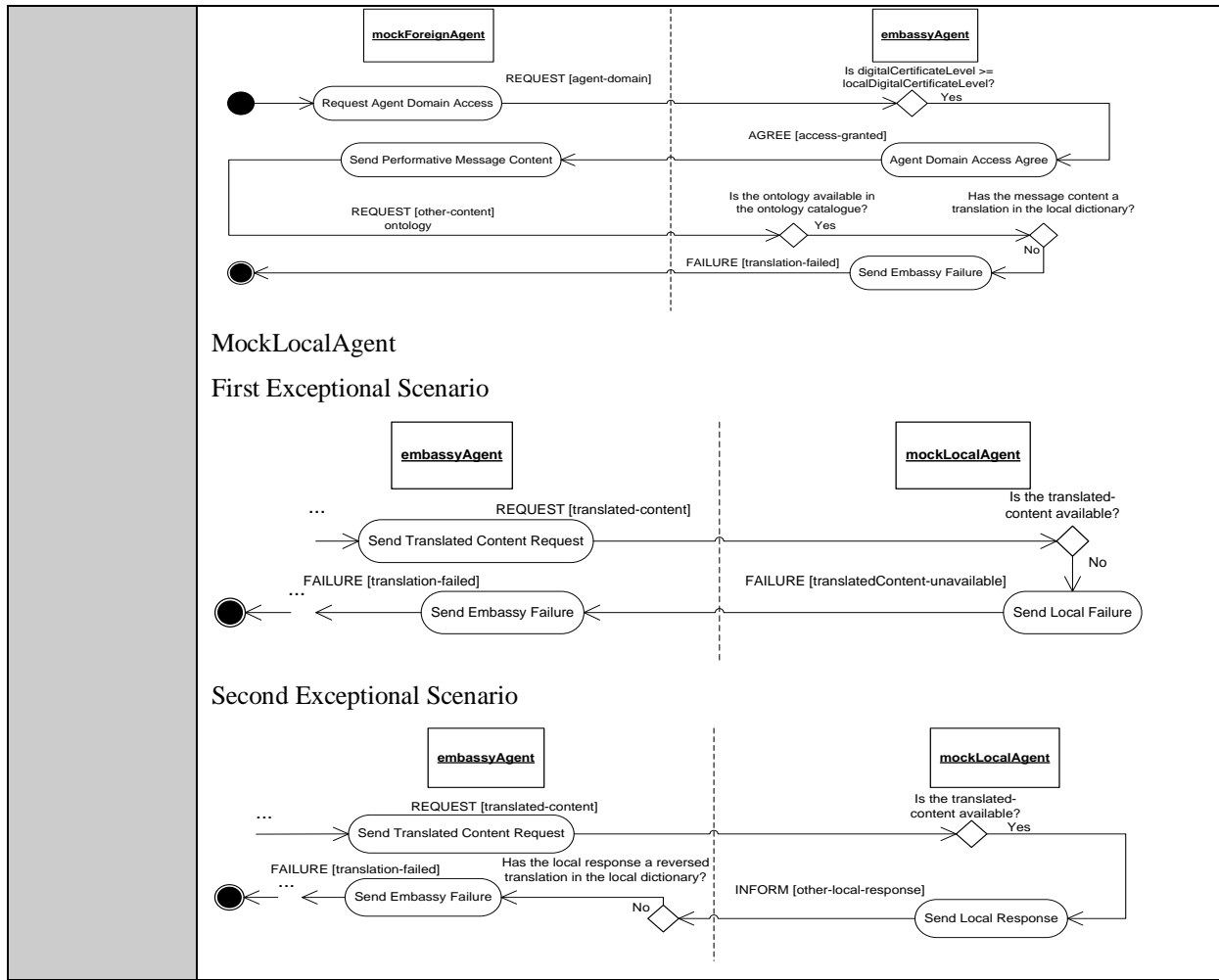


Table A.24: Test Design Pattern of the EmbassyAgent

When the `mockForeignAgent` joins the environment, it sends a message characterized by a `REQUEST` performative indicating the willingness to be granted the access to the agent-domain from the `embassyAgent`. In this case, if the agent-domain is not supported or the `mockForeignAgent` digital certificate level is not enough to access this agent-domain, the `embassyAgent` replies with a `REFUSE` message (MockForeignAgent first exceptional scenario).

If the `mockForeignAgent` does not send an agent domain request message, the `RequestAccessServer` behaviour will be blocked and is put into the blocked behaviours queue by the agent scheduler. When a new message (`REQUEST`) is inserted in the agent message queue, the blocked behaviour becomes available for execution again, so that it has a chance to process the received message.

When the `mockForeignAgent` receives an agent-domain access granted agree message from the `embassyAgent`, it sends a message characterized by a `REQUEST` performative indicating the willingness to be informed by the translated local response from the `embassyAgent`. In this case, if the ontology is unavailable in the `ontologyCatalogue` or the message content has no translation in the `localDictionary`, the `embassyAgent` replies with a `FAILURE` message (`MockForeignAgent` second and third exceptional scenarios).

If the `mockForeignAgent` does not send a translated local response request message, the `RequestTranslatedLocalResponseServer` behaviour will be blocked and is put into the blocked behaviours queue by the agent scheduler. When a new message (`REQUEST`) is inserted in the agent message queue, the blocked behaviour becomes available for execution again, so that it has a chance to process the received message.

When the `mockLocalAgent` receives a local response request message from the `embassyAgent`, it replies with a message characterized by either:

- i. An `INFORM` performative, indicating that the translated content is available in the `mockLocalAgent` catalogue, in this case, if the local response has a reversed translation in the `localDictionary` of the `embassyAgent`, then it will send the `translatedLocalResponse` back to the `foreignAgent` (normal scenario). If the local response has no reversed translation in the `localDictionary` of the `embassyAgent`, then it will send a `FAILURE` message back to the `foreignAgent` (`MockLocalAgent` second exceptional scenario).
- ii. A `FAILURE` performative, indicating that the translated content is unavailable in the `mockLocalAgent` catalogue, in this case the `embassyAgent` will send a `FAILURE` message back to the `foreignAgent` (`MockLocalAgent` first exceptional scenario).
- iii. No `RESPONSE`, indicating that the `mockLocalAgent` does not respond to the local response request message, in this case the `RequestLocalReponsePerformer`



behaviour will be blocked and is put into the blocked behaviours queue by the agent scheduler. When a new message (INFORM/FAILURE) is inserted in the agent message queue, the blocked behaviour becomes available for execution again, so that it has a chance to process the received message.

### A.7.5.3 LocalAgent

Table A.25, represents the design of test design pattern of the Agent Under Test.

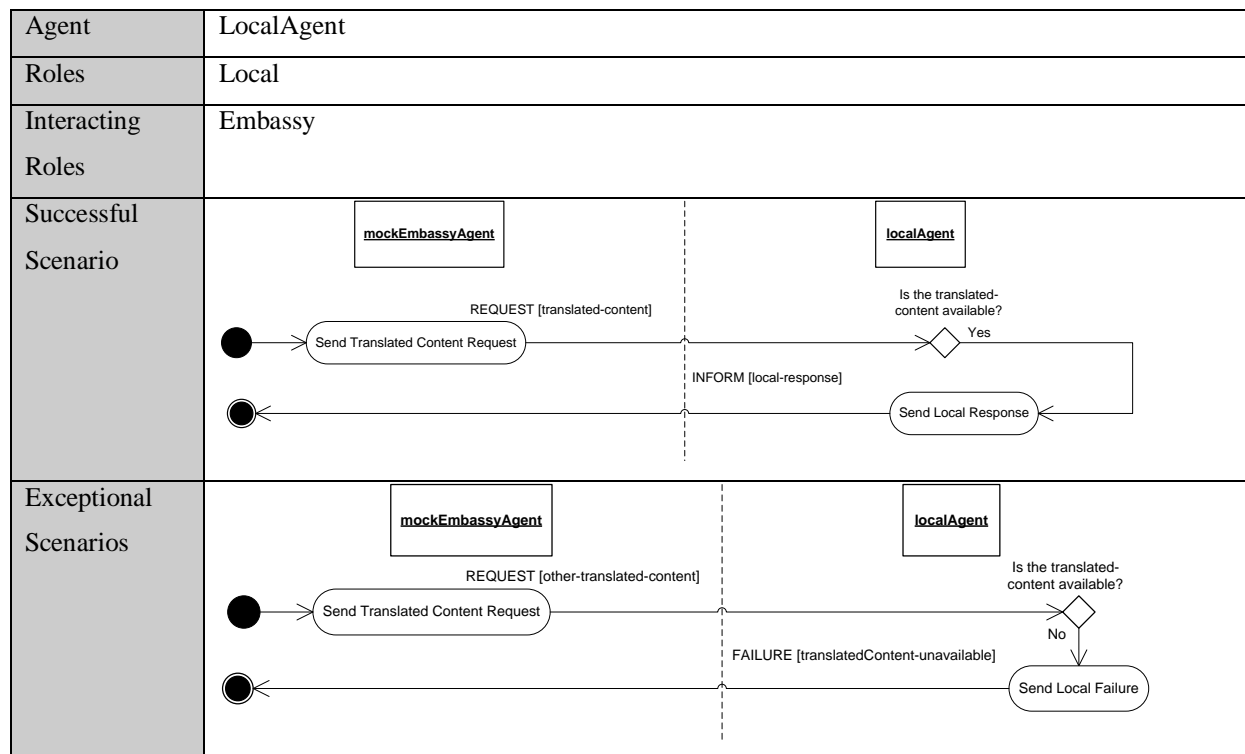


Table A.25: Test Design Pattern of the LocalAgent

When the `mockEmbassyAgent` joins the environment, it sends a message characterized by a `REQUEST` performative indicating the willingness to be informed by the local-response of the translated-content from the `localAgent`. In this case, if the translated-content is unavailable, the `localAgent` replies with a `FAILURE` message (exceptional scenario).

If the `mockEmbassyAgent` does not send a local response request message, the `RequestLocalResponseServer` behaviour will be blocked and is put into the blocked behaviours queue by the agent scheduler. When a new message (`REQUEST`) is inserted in the agent message queue, the blocked behaviour becomes available for execution again, so that it has a chance to process the received message.

## **A.7.6 Pattern Implementation**

### **A.7.6.1 ForeignAgent**

The mock agent `MockEmbassyAgent` has a plan (test scenario) represented by two JADE Behaviors called `RequestAccessServer` and `RequestTranslatedLocalResponseServer` to test the AUT agent `ForeignAgent`. To execute the AUT agent test case `ForeignAgentTestCase`, all we need is to create a subclass of `JADETestCase` and to implement a test method `testForeign()` that creates an instance of the AUT `ForeignAgent`, and an instance of the mock agent `MockEmbassyAgent`. After that, this test method waits until their interaction finishes and asks the mock agent about the test result.

### **A.7.6.2 EmbassyAgent**

The mock agent `MockForeignAgent` has a plan (test scenario) represented by a JADE Behavior called `RequestAccessPerformer` to test the AUT agent `EmbassyAgent`. To execute the AUT agent test case `EmbassyAgentTestCase`, all we need is to create a subclass of `JADETestCase` and to implement a test method `testEmbassy()` that creates an instance of the AUT `EmbassyAgent`, and an instance of the mock agent `MockForeignAgent`. After that, this test method waits until their interaction finishes and asks the mock agent about the test result.

The mock agent `MockLocalAgent` has a plan (test scenario) represented by a JADE Behavior called `RequestLocalResponseServer` to test the AUT agent `EmbassyAgent`. To execute the AUT agent test case `EmbassyAgentTestCase`, all we need is to create a subclass of `JADETestCase` and to implement a test method `testEmbassy()` that creates an instance of the AUT `EmbassyAgent`, and an instance of the mock agent `MockLocalAgent`. After that, this test method waits until their interaction finishes and asks the mock agent about the test result.

### **A.7.6.3 LocalAgent**

The mock agent `MockEmbassyAgent` has a plan (test scenario) represented by a JADE Behavior called `RequestLocalReponsePerformer` to test the AUT agent `LocalAgent`.

To execute the AUT agent test case `LocalAgentTestCase`, all we need is to create a subclass of `JADETestCase` and to implement a test method `testLocalResponse()` that creates an instance of the AUT `LocalAgent`, and an instance of the mock agent `MockEmbassyAgent`. After that, this test method waits until their interaction finishes and asks the mock agent about the test result.

## A.8 Wrapper Pattern

### A.8.1 Design Pattern Template

Table A.26 represents the Wrapper design pattern template.

Template Element	Description
<b>Name</b>	Wrapper Pattern
<b>Intent</b>	This pattern allows a legacy application to be incorporated into a multiagent system (MAS). The legacy code is extended with agent capabilities by agentifying it. Agentification may be a temporary solution until the legacy code can be upgraded or migrated to an agent implementation.
<b>Applicability</b>	If a legacy system provides functionality required by an agent system, can not easily be redeveloped as an agent and has a stable interface that can be wrapped, this pattern is appropriate. Here a legacy system is any non-agent application.
<b>Motivation Example</b>	<p>This pattern enables the encapsulation of legacy code and enhancement of the legacy interface, to agentify the legacy.</p> <p>By introducing a layer of inter-agent communication functionality, the wrapper allows agents to communicate with the legacy using an ACL, and maps responses from the legacy back to ACL for the agents.</p> <p>This ensures that agent communication protocols are respected and the legacy code remains decoupled from the agents.</p>
<b>Participants</b>	This coordination pattern involves a wrapper, a number of client agents and generally only one legacy system, since the wrapper is domain or system-specific. The role of the wrapper is to interface the agents to the legacy system by agentifying the legacy.

Table A.26: Wrapper Design Pattern Template

### A.8.2 Collaboration diagram

Figure A.23 represents the collaboration diagram of the Wrapper pattern (58).

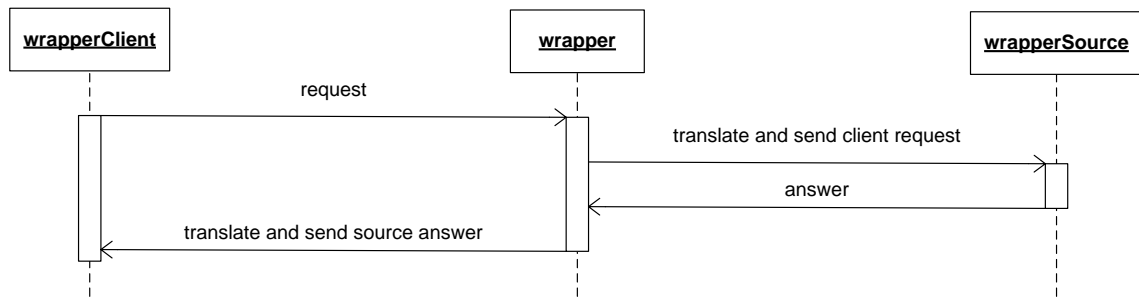


Figure A.23: The Collaboration diagram of the Wrapper pattern

### A.8.3 Structure Diagram

Figure A.24 represents the structure diagram of the Wrapper pattern.

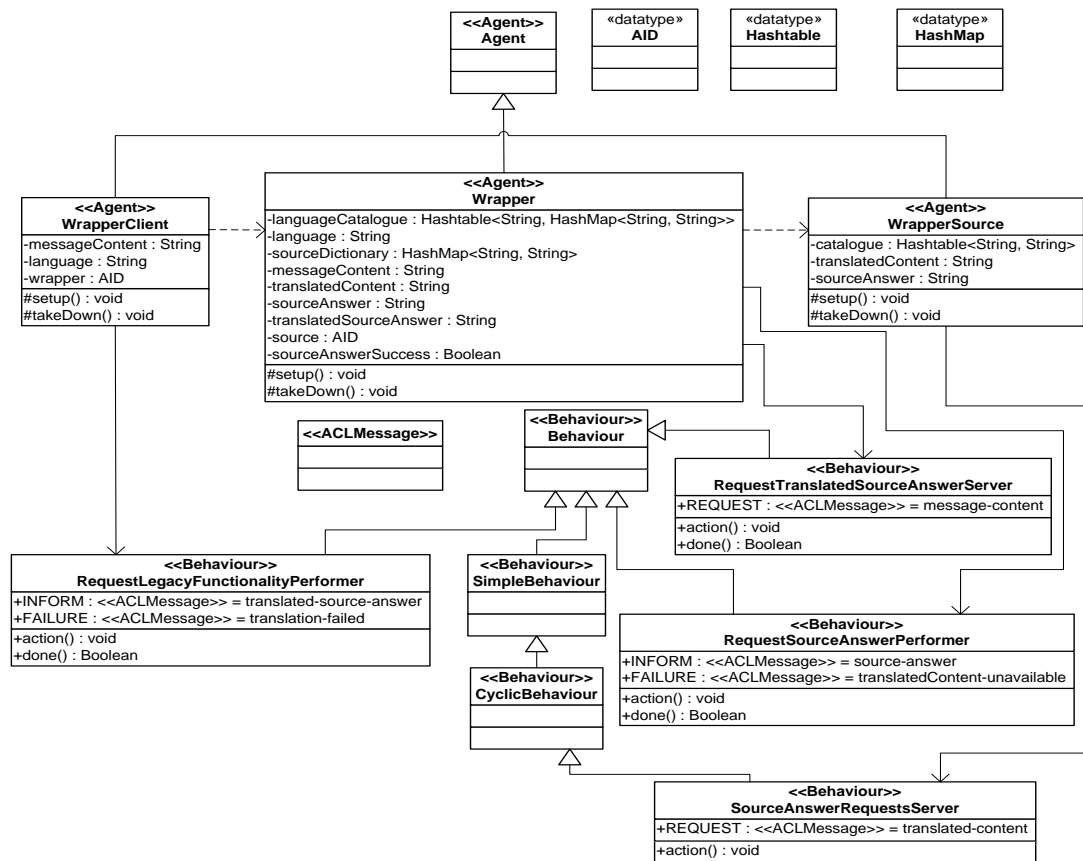


Figure A.24: The Structure diagram of the Wrapper pattern

### A.8.4 Behaviour Diagram

Figure A.25 represents the behaviour diagram of the Wrapper pattern.

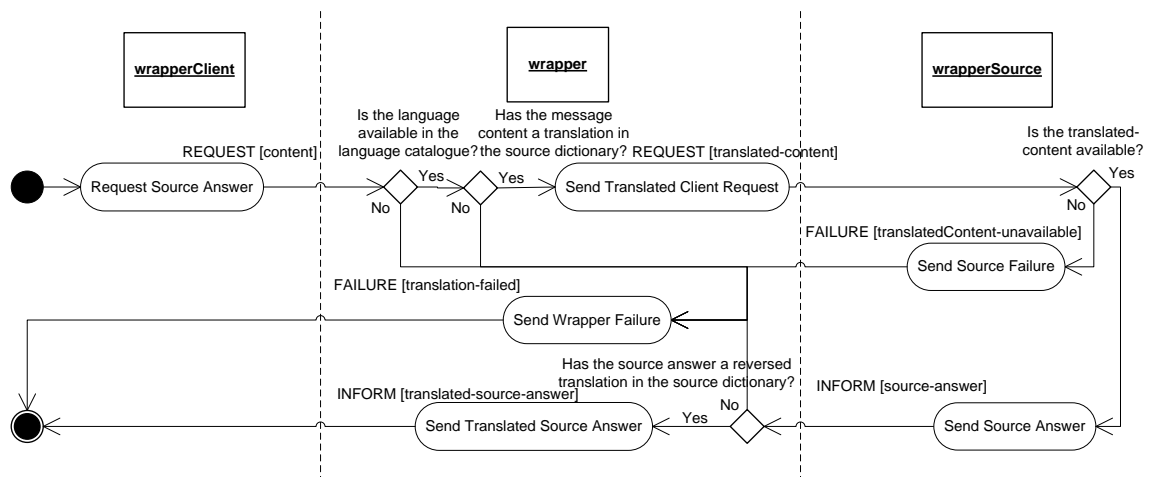


Figure A.25: The Behaviour diagram of the Wrapper pattern

## A.8.5 Test Design Patterns

### A.8.5.1 WrapperClientAgent

Table A.27, represents the design of test design pattern of the Agent Under Test.

Agent	WrapperClientAgent
Roles	WrapperClient
Interacting Roles	Wrapper
Successful Scenario	<pre> sequenceDiagram     participant WCA as wrapperClientAgent     participant MWA as mockWrapperAgent      WCA-&gt;&gt;MWA: REQUEST [content]     MWA-&gt;&gt;MWA: Is the language available in the language catalogue?     MWA-&gt;&gt;MWA: Has the message content a translation in the source dictionary?     MWA-&gt;&gt;WCA: Perform the translatedContent request for sourceAnswer     WCA-&gt;&gt;MWA: INFORM [translated-source-answer]     MWA-&gt;&gt;MWA: Has the source answer a reversed translation in the source dictionary?     MWA-&gt;&gt;WCA: Send Translated Source Answer     </pre>
Exceptional Scenarios	<pre> sequenceDiagram     participant WCA as wrapperClientAgent     participant MWA as mockWrapperAgent      WCA-&gt;&gt;MWA: REQUEST [content]     MWA-&gt;&gt;MWA: Is the language available in the language catalogue?     MWA-&gt;&gt;MWA: Has the message content a translation in the source dictionary?     MWA-&gt;&gt;WCA: Perform the translatedContent request for sourceAnswer     WCA-&gt;&gt;MWA: INFORM [translated-source-answer]     MWA-&gt;&gt;MWA: Has the source answer a reversed translation in the source dictionary?     MWA-&gt;&gt;WCA: Send Wrapper Failure     </pre>

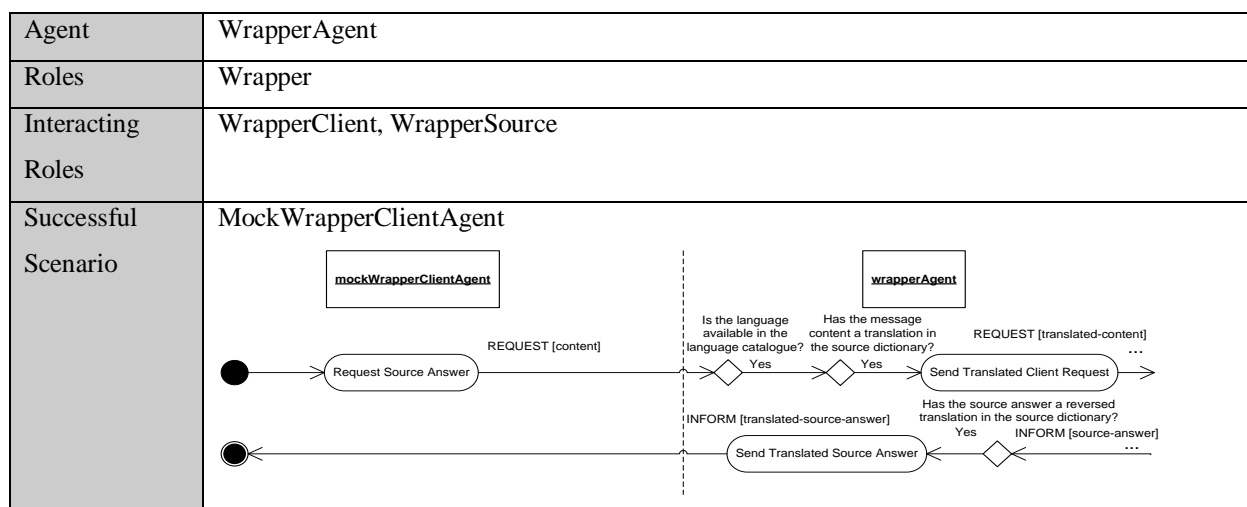
Table A.27: Test Design Pattern of the WrapperClientAgent

When the mockWrapperAgent receives a translated source answer request message from the wrapperClientAgent, it replies with a message characterized by either:

- i. An INFORM performative, indicating that the language is available in the languageCatalogue, the message content has a translation in the sourceDictionary, a source answer is available and having a reversed translation in the sourceDictionary, in this case the wrapperClientAgent will terminate (normal scenario).
- ii. A FAILURE performative, indicating that either the language is unavailable in the languageCatalogue, or the message content has no translation in the sourceDictionary, or a source answer is unavailable, or has no reversed translation in the sourceDictionary, in this case the wrapperClientAgent will terminate (exceptional scenario).
- iii. No RESPONSE, indicating that the mockWrapperAgent does not respond to the translated source answer request message, in this case the RequestLegacyFunctionalityPerformer behaviour will be blocked and is put into the blocked behaviours queue by the agent scheduler. When a new message (INFORM/FAILURE) is inserted in the agent message queue, the blocked behaviour becomes available for execution again, so that it has a chance to process the received message.

#### A.8.5.2 WrapperAgent

Table A.28, represents the design of test design pattern of the Agent Under Test.



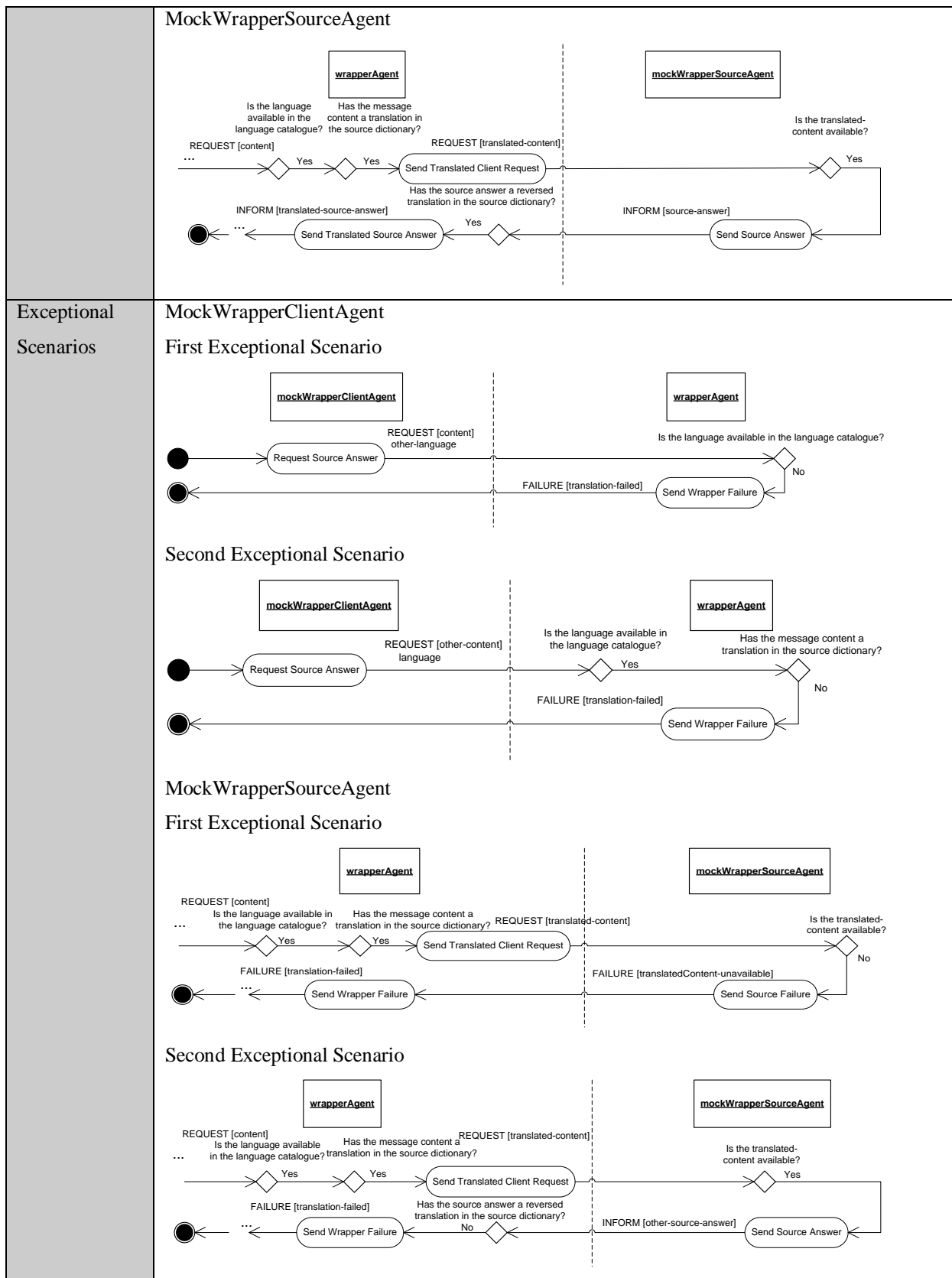


Table A.28: Test Design Pattern of the WrapperAgent

When the `mockWrapperClientAgent` joins the environment, it sends a message characterized by a `REQUEST` performative indicating the willingness to be informed by the translated source answer from the `wrapperAgent`. In this case, if the language is unavailable in the `languageCatalogue` or the message content has no translation in the `sourceDictionary`, the `wrapperAgent` replies with a `FAILURE` message (`MockWrapperClientAgent` first and second exceptional scenarios).

If the `mockWrapperClientAgent` does not send a translated source answer request message, the `RequestTranslatedSourceAnswerServer` behaviour will be blocked and is put into the blocked behaviours queue by the agent scheduler. When a new message (`REQUEST`) is inserted in the agent message queue, the blocked behaviour becomes available for execution again, so that it has a chance to process the received message.

When the `mockWrapperSourceAgent` receives a source answer request message from the `wrapperAgent`, it replies with a message characterized by either:

- i. An `INFORM` performative, indicating that the translated content is available in the `mockWrapperSourceAgent` catalogue, in this case, if the source answer has a reversed translation in the `sourceDictionary` of the `wrapperAgent`, then it will send the `translatedSourceAnswer` back to the `wrapperClientAgent` (normal scenario). If the source answer has no reversed translation in the `sourceDictionary` of the `wrapperAgent`, then it will send a `FAILURE` message back to the `wrapperClientAgent` (`MockWrapperSourceAgent` second exceptional scenario).
- ii. A `FAILURE` performative, indicating that the translated content is unavailable in the `mockWrapperSourceAgent` catalogue, in this case the `wrapperAgent` will send a `FAILURE` message back to the `wrapperClientAgent` (`MockWrapperSourceAgent` first exceptional scenario).
- iii. No `RESPONSE`, indicating that the `mockWrapperSourceAgent` does not respond to the source answer request message, in this case the



`RequestSourceAnswerPerformer` behaviour will be blocked and is put into the blocked behaviours queue by the agent scheduler. When a new message (INFORM/FAILURE) is inserted in the agent message queue, the blocked behaviour becomes available for execution again, so that it has a chance to process the received message.

### A.8.5.3 WrapperSourceAgent

Table A.29, represents the design of test design pattern of the Agent Under Test.

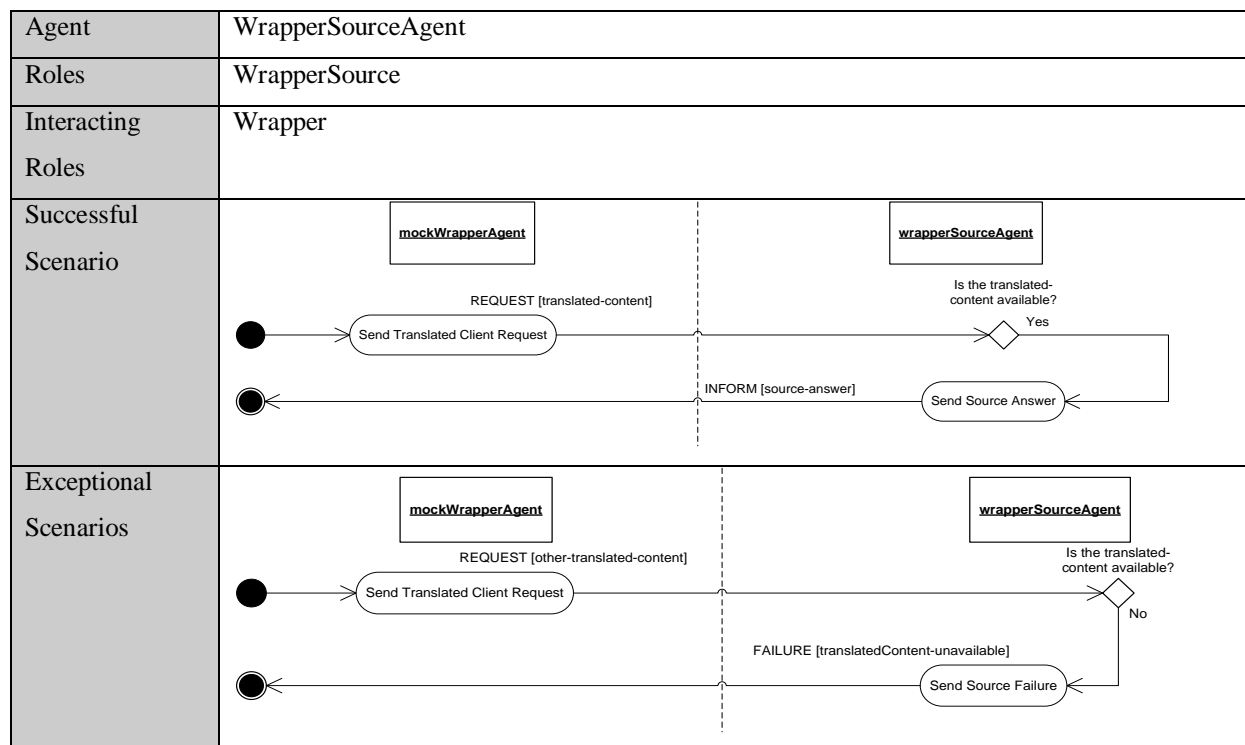


Table A.29: Test Design Pattern of the WrapperSourceAgent

When the `mockWrapperAgent` joins the environment, it sends a message characterized by a `REQUEST` performative indicating the willingness to be informed by the source-answer of the translated-content from the `wrapperSourceAgent`. In this case, if the translated-content is unavailable, the `wrapperSourceAgent` replies with a `FAILURE` message (exceptional scenario).

If the `mockWrapperAgent` does not send a source answer request message, the `SourceAnswerRequestsServer` behaviour will be blocked and is put into the blocked behaviours queue by the agent scheduler. When a new message (`REQUEST`) is inserted in the

agent message queue, the blocked behaviour becomes available for execution again, so that it has a chance to process the received message.

### **A.8.6 Pattern Implementation**

#### **A.8.6.1 WrapperClientAgent**

The mock agent `MockWrapperAgent` has a plan (test scenario) represented by a JADE Behavior called `RequestTranslatedSourceAnswerServer` to test the AUT agent `WrapperClientAgent`. To execute the AUT agent test case `WrapperClientAgentTestCase`, all we need is to create a subclass of `JADETestCase` and to implement a test method `testWrapping()` that creates an instance of the AUT `WrapperClientAgent`, and an instance of the mock agent `MockWrapperAgent`. After that, this test method waits until their interaction finishes and asks the mock agent about the test result.

#### **A.8.6.2 WrapperAgent**

The mock agent `MockWrapperClientAgent` has a plan (test scenario) represented by a JADE Behavior called `RequestLegacyFunctionalityPerformer` to test the AUT agent `WrapperAgent`. To execute the AUT agent test case `WrapperAgentTestCase`, all we need is to create a subclass of `JADETestCase` and to implement a test method `testWrapping()` that creates an instance of AUT `WrapperAgent`, and an instance of the mock agent `MockWrapperClientAgent`. After that, this test method waits until their interaction finishes and asks the mock agent about the test result.

The mock agent `MockWrapperSourceAgent` has a plan (test scenario) represented by a JADE Behavior called `SourceAnswerRequestsServer` to test the AUT agent `WrapperAgent`. To execute the AUT agent test case `WrapperAgentTestCase`, all we need is to create a subclass of `JADETestCase` and to implement a test method `testWrapping()` that creates an instance of the AUT `WrapperAgent`, and an instance of the mock agent `MockWrapperSourceAgent`. After that, this test method waits until their interaction finishes and asks the mock agent about the test result.

#### **A.8.6.3 WrapperSourceAgent**

The mock agent `MockWrapperAgent` has a plan (test scenario) represented by a JADE Behavior called `RequestSourceAnswerPerformer` to test the AUT agent `WrapperSourceAgent`. To execute the AUT agent test case `WrapperSourceAgentTestCase`, all we need is to create a subclass of `JADETestCase` and to implement a test method `testSourceAnswer()` that creates an instance of the AUT `WrapperSourceAgent`, and an instance of the mock agent `MockWrapperAgent`. After that, this test method waits until their interaction finishes and asks the mock agent about the test result.