https://doi.org/10.31891/csit-2024-4-12
UDC 004.8:004.41:004.67

Oleksandr KARATAIEV
Kharkiv National University of Radioelectronics

TOWARDS MULTI-AGENT PLATFORM DEVELOPMENT

This paper focuses on the design and evaluation of a FIPA standard compliant multi-agent platform. The relevance of the topic is due to the growing need for flexible, reliable, and efficient software solutions capable of solving complex intelligent problems in distributed environments. The study is dedicated to the problem of developing and evaluating an agent platform using the Kotlin programming language. The main goal of this work is to design and implement a modular, scalable, and adaptive agent platform. The existing frameworks for the development of multi-agent systems are reviewed, the key components of such systems are highlighted, and the advantages of using Kotlin in the context of a multi-agent architecture are discussed. The scientific contribution of the paper is the creation of a modern FIPA-compliant multi-agent platform that exploits the advantages of the Kotlin language. The performance and resource intensity of the developed system are analyzed, and the platform's compliance with FIPA standards and its interoperability are evaluated. Two different metrics are used to ensure the quality of the system. One of the metrics is the percentage of covered code. This metric is measured using the kover library. We achieved 71.4% coverage of classes and 57.1% coverage of commands. Further coverage is complicated by the use of multi-threaded technologies. The second metric is the system's score for comments from the sonarlint evaluation tool. During development, 16 comments were identified and fixed. This allows us to achieve a high level of code quality and ensure quality for the future. The study demonstrates the potential of integrating modern language capabilities with the multi-agent paradigm, opening new perspectives for the development of efficient and scalable solutions in the area of distributed intelligent systems.

Keywords: multi-agent systems, FIPA, Kotlin, software development, performance evaluation.

Олександр КАРАТАЄВ
Харківський національний університет радіоелектроніки

НА ШЛЯХУ ДО РОЗРОБКИ МУЛЬТИАГЕНТНОЇ ПЛАТФОРМИ

Ця стаття присвячена розробці та оцінці мультиагентної платформи, сумісної зі стандартом FIPA. Актуальність теми зумовлена зростаючою потребою в гнучких, надійних та ефективних програмних рішеннях, здатних вирішувати складні інтелектуальні задачі в розподілених середовищах. Дослідження присвячено проблемі розробки та оцінки агентської платформи з використанням мови програмування Kotlin. Основною метою цієї роботи є розробка та впровадження модульної, масштабованої та адаптивної агентської платформи. Розглянуто існуючі фреймворки для розробки мультиагентних систем, виділено ключові компоненти таких систем та обговорено переваги використання Kotlin у контексті мультиагентної архітектури. Науковий внесок статті полягає у створенні сучасної мультиагентної платформи, сумісної з FIPA, яка використовує переваги мови Kotlin. Проаналізовано продуктивність і ресурсомісткість розробленої системи, а також оцінено відповідність платформи стандартам FIPA та її взаємодію. Для забезпечення якості системи використовуються дві різні метрики. Одним із показників є відсоток охопленого коду. Цей показник вимірюється за допомогою бібліотеки kover. Ми досягли 71,4% охоплення занять і 57,1% охоплення команд. Подальше покриття ускладнюється використанням багатопотокових технологій. Другий показник – це оцінка системи для коментарів інструменту оцінки sonarlint. Під час розробки було виявлено та виправлено 16 зауважень. Це дозволяє нам досягти високого рівня якості коду та забезпечити якість на майбутнє. Дослідження демонструє потенціал інтеграції сучасних мовних можливостей із мультиагентною парадигмою, відкриваючи нові перспективи для розробки ефективних та масштабованих рішень у сфері розподілених інтелектуальних систем.

Ключові слова: мультиагентні системи, FIPA, Kotlin, розробка ПЗ, оцінка продуктивності.

Introduction

Contemporary software necessitates attributes such as flexibility, reliability, and efficiency. Addressing these requirements has led to the emergence of an agent-based paradigm within the field of software engineering. This paradigm allows for the modeling of intelligent functions, particularly those associated with knowledge processing, through a network of agents. These agents interact both with the external environment and amongst themselves to address intricate intellectual tasks. The proliferation of information and associated technologies aligns seamlessly with the capabilities of multi-agent systems. Concurrently, advancements in distributed object technologies, providing the necessary infrastructure, have further bolstered the agent-based approach.

Agent-based systems represent one of the most active and substantial areas of research and development within information technology in recent years [1-3]. Over the past five to ten years, the conceptual underpinnings of agent-based systems have become almost ubiquitous. Contemporary definitions of agents span a spectrum from general autonomous agents, software agents, and intelligent agents to more specific classifications such as interface agents, virtual agents, information agents, and mobile agents. Typically, agents are characterized by specific attributes. For instance, Wooldridge and Jennings [4], in their seminal work, proposed a notion of agents that implies autonomy—the ability to function without interference, social ability for interaction with other agents, reactivity enabling agents to perceive and respond to a dynamic environment, and proactivity for purposeful behavior. These characteristics are widely regarded as key qualities defining "agency." From a software engineering perspective, an agent is an autonomous software entity with a prolonged lifespan, capable of adapting its behavior according to environmental changes and interacting with other agents [5].

This study aims to implement an agent-based platform and conduct a comprehensive evaluation to assess its performance, adherence to FIPA (Foundation for Intelligent Physical Agents) standards [6], and overall suitability for modern software systems. This work is devoted to the development and improvement of the agent platform. The multi-agent software system is utilizing the Kotlin language. Following to FIPA standards, the multi-agent system (MAS) leverages Kotlin's features to enhance performance and integrates support for modern technologies. However, it faces challenges, including documentation, developer-friendly approach, and testing.

Thus, the following research questions have arisen.

RQ1. To what extent does the given agent platform adhere to FIPA standards, specifically in terms of communication protocols and agent interactions?

RQ2. How does the agent platform perform in terms of responsiveness, scalability, and resource utilization under varying workloads?

RQ3. What improvements or optimizations can be suggested based on the evaluation results to enhance the platform's performance and compliance with FIPA standards?

Overview of the Methods and Tools for Multi-Agent Systems Development

In the evolution of multi-agent systems development, various frameworks have emerged to facilitate the creation of such systems. This section explores prominent frameworks with a focus on their compatibility with FIPA standards, integration with the Java and Kotlin language, community support, and ongoing development efforts. Three frameworks that meet or partially meet these criteria are examined: Java Agent Development framework (JADE) [7], Foundation for Intelligent Physical Agents Operating System (FIPA-OS) [8], Intelligent Agents JACK [9] and Open Agent Architecture (OAA) [10]. Let's take a closer look at each of these frameworks.

JADE [7] is a versatile software platform providing essential middleware functions that are application-independent, simplifying the implementation of distributed applications utilizing the software agent abstraction. Operating as a fully distributed system, JADE accommodates agents, each functioning as a separate thread potentially executing on a remote machine. Key features include autonomy, proactivity, a unique identifier for each agent, and the ability to initiate communication transparently with other agents. JADE is fully compliant with FIPA standards, offers debugging and monitoring mechanisms, and seamlessly integrates with modern web technologies.

FIPA-OS [8] is a component-based toolkit designed for the swift development of FIPA-compliant agents. Core functionalities include a base class for agent implementation, a task manager for task division, an agent communication manager for protocol compliance, a message transport service for communication, and an abstract factory interface for specific instances, along with a database factory (DB) for interaction with popular database implementations.

JACK Intelligent Agents [9], developed by Agent Oriented Software, is a Java-based agent development environment tailored for closed systems with a defined number of agents. Agents communicate using messages, with the system designed for agents to collaborate on achieving goals. JACK's counterpart, the FIPA JACK framework, extends its functionality to open systems [11].

Open Agent Architecture [10] provides an architecture for multi-agent systems, supporting programming languages such as Java, Perl, and Prolog. Designed for open systems, OAA mandates agent collaboration to achieve goals and offers efficient and reliable communication. OAA introduces a facilitator agent that resolves coordination and cooperation between agents. Each agent must register its data with the facilitator, which, in turn, facilitates communication and task allocation among agents. Importantly, the facilitator can assign the same task to multiple agents concurrently.

In summary, these frameworks present diverse approaches to multi-agent system development, catering to different application scenarios and requirements. Each framework brings unique features and capabilities, contributing to the rich landscape of agent-based development tools.

The Foundations for Intelligent Physical Agents (FIPA) stands as a standards organization within the IEEE Computer Society, advocating for agent-based technology and the seamless interoperability of its standards with other technologies [6]. Despite the increasing prominence of agent-based systems, the analysis reveals a scarcity of modern multi-agent platform solutions conforming to FIPA standards. The primary beneficiaries of such platforms are small projects and startups, particularly those seeking profitability and convenience in multi-agent platform development. The key user base comprises developers who prioritize clarity, ease of use, and direct applicability for writing multi-agent systems.

In the current market, the array of available multi-agent systems is limited [12, 13]. Among these, only a handful are pertinent to contemporary needs. JADE remains one of the most popular development platforms, despite being written in an older version of Java and demanding substantial resources for operation [14]. The IBM Agent Builder platform [15], while user-friendly with a graphical interface and simplified development, falls short of FIPA standards. The Mava framework [16], although an alternative, may lack the required flexibility for certain projects. Each of these frameworks carries its own set of advantages and disadvantages.

In conclusion, the pursuit of an ideal multi-agent platform is ongoing, necessitating a balance between adherence to standards, developer convenience, and modern technological integration.

Aims and Objectives

A multi-agent system is a complex information system consisting of a large number of interacting agents. Agent-based modeling is a modern approach to studying many processes. Agents allow modeling various problems for their further analysis and optimization. Multi-agent systems can be applied in various fields, for example, modeling market behavior in economics, modeling biological and social systems in science, modeling complex technical systems in engineering, and many other areas [17-19].

The MAS development highlights the persistent need for comprehensive documentation, developer-centric design, and robust testing practices. This study aims to implement an agent-based platform and conduct a comprehensive evaluation to assess its performance, adherence to FIPA standards, and overall suitability for up-today software systems.

The objectives of this paper are

1) to design and implement an agent-based platform, emphasizing modularity, scalability, and adaptability;
2) to analyze the impact of the agent platform on system resources and overall software performance in order to evaluate the performance of the implemented agent platform;
3) to evaluate the extent to which the implemented agent platform conforms to relevant FIPA standards, ensuring interoperability and compatibility with other FIPA-compliant systems;
4) to assess the practical applicability of the agent-based platform.

Multi-Agent Platform Development and Evaluation

Developing a multi-agent platform is a challenging but promising task. In the implementation of a multi-agent system, a foundational step involves defining its key characteristics. FIPA [6] provides a comprehensive description of the primary components constituting a multi-agent system, detailing their roles and features. At the core of such systems lies the Agent, a computational process endowed with autonomous communication functionality, acting as a program that collaborates with other agents through an agent communication language [4]. An agent plays a key role by consolidating one or more service capabilities specified in the service specification into a unified and integrated execution model. This integration fosters effective collaboration and seamless interaction within the multi-agent system.

The key components of a multi-agent system are agents, a directory broker (DF), an agent platform (AP), and a message transport system (MTS), which provide efficient interaction and communication between agents [6]. While optional, the directory facilitator serves as a crucial component of the agent platform, implemented as a service when present. The DF offers yellow pages services, furnishing information about available services to other agents. Agents can register their services with the DF or query it to retrieve information about services provided by fellow agents. In a node, multiple DFs can coexist, potentially forming federations. The agent platform serves as the physical infrastructure for deploying agents, encompassing agent management components like the directory service, agent management service (AMS), message service, and the agents themselves. Notably, the internal structure of the AP is left to the discretion of agent system developers and is not standardized in [6].

In FIPA systems, agents engage in communication through the exchange of messages. This interaction involves three key aspects: message structure, message representation, and message transport [6]. Together, these aspects constitute the foundation for efficient information exchange and communication between agents within FIPA systems. This systematic approach ensures effective coordination and communication, pivotal for the successful operation of multi-agent systems.

The software considered in this study is based on a multi-agent system platform developed in accordance with the FIPA standard. The platform's programming language is Kotlin. The platform has a console interface and provides only functions for working with agents. The platform provides the physical infrastructure in which agents can be deployed. According to the FIPA standard, any multi-agent system must support different operating systems and be portable. The platform under development will work in any environment where it is possible to install the Java Virtual Machine (JVM): Windows, Mac OC X, Linux, Solaris.

Despite the advantages of using multi-agent systems, such a system usually has some disadvantages that should be considered when implementing a platform. First, due to the autonomy of the agents, the results of the platform have limited predictability and control. Second, multi-agent platforms are often prone to crashes and errors. Third, computational results may be inaccurate.

In addition, the following issues need to be addressed during the design and implementation of the platform:

- the problem of distributed problem solving;
- modeling of agent behavior;
- issues of coordination and cooperation of agents.

Kotlin is a compact programming language that allows you to write less code to achieve the desired results [20]. Using Kotlin together with multi-agent architecture allows you to create efficient and high-performance systems with minimal overhead. Coroutines technology in Kotlin provides a convenient and easy way to work with asynchronous code [21]. Multi-agent systems require interaction between different agents in an asynchronous way, and Coroutines allow you to conveniently manage threads of execution and create non-linear execution threads.

Multi-agent systems are based on the exchange of messages and event processing between different agents. Kotlin has a rich set of tools for working with events and messages, which simplifies communication between agents and the implementation of the desired functionality.

Kotlin, together with Coroutines technology, provides built-in support for concurrency, which allows you to efficiently multitask and manage parallel execution. This is especially important in multi-agent systems where many agents can work simultaneously. Kotlin provides built-in support for KDoc, which is a documentation language for Kotlin code. KDoc allows developers to create clear, understandable, and detailed documentation for classes, functions, variables, and other code elements. This greatly facilitates understanding of the system's functionality, promotes rapid implementation and collaboration between developers, and helps ensure the quality and reliability of the code base. Thanks to KDoc, developers can quickly find the necessary information about classes and their methods, parameters, returned values, and other details, which contributes to effective project work and facilitates interaction with other members of the development team. Thus, the choice of programming language is determined by the capabilities of the Kotlin language and its advantages.

To effectively solve a problem, it is important to have indicators by which we can determine whether the problem has been solved. For each part of the development task, a different evaluation method was chosen that will accurately show whether the task was completed. For the task of implementing a standardized approach for the system functionality, namely for messaging, creating new agents, checking the ability of an agent to perform tasks, and connecting agents to data, the method of checking for availability was chosen. The verification of the task will be to check and demonstrate the components responsible for implementing these mechanisms. The implementation of these components should be compared with the patterns and standards that will be used to solve the problem.

Two different metrics should be used for the task of improving the quality of the system. One of the metrics will be the percentage of covered code, where 100% coverage means that the unit tests will run and test all operations in the system when they are launched. This indicator can be measured using the kover library (https://github.com/Kotlin/kotlinx-kover). The successful completion of the task can be considered if the code coverage is more than 50%, which will prove that more than half of the code used will meet the quality indicator. The second metric will be the system's score for comments from the sonarlint evaluation tool [22]. The number of comments in the system should be 0. If any comments should not be corrected, this should be noted along with the reason. It was decided to measure the task of adding documentation by the number of classes with available documentation and their ratio to the total number of classes. The task will be completed successfully if the number of classes with documentation is more than half of all classes in the system. The task of adding a configuration should be measured by its existence in principle and the number of agent characteristics that can be defined in an alternative way (i.e. not directly through the code), for example, using a configuration file.

To solve a task a multi-agent platform must go through a number of stages. The main task is solved by an agent, it contains a function that will solve the task and a function that will evaluate the ability of this agent to perform the task. A set of agents can be managed by a DF, which in turn is represented by an agent implementation. Agents of the same type can usually be combined into APs, and an MTS implementation exists to exchange messages between different APs. The entire process of a multi-agent system is represented by an AMS.

Figure 1 shows a sequence diagram that demonstrates the successful resolution of a received task. To solve the task, AMS passes the task to the corresponding DF agent. In turn, MS begins to search among the agents it knows who can perform the task. From the set of agents that answered that they can solve the task, DF selects one agent to solve it. This choice can be made either based on the first agent that responded affirmatively to the query, or by using the success rates that agents can return. This stage can be described as a handshake, i.e., the DF and the agent agree on the task. After that, the agent starts performing the task, each agent can be uniquely implemented for this purpose. After successfully completing the task, it returns the result, which in turn the DF passes to the client.

In case of failure, the difference is that the agent may return an error, or the system will not be able to find an agent that can perform this task. Then, according to the settings, the system switches to the waiting mode for new agents that can perform this task. If, upon reaching the maximum waiting time, no agent is found that can successfully complete this task, the task is considered impossible to complete and a corresponding message is returned to the user. The corresponding scheme is shown in Fig. 2.

The components show the various parts of a system and their dependencies. The main components of the system:

- The operational system within which the agent platform runs;
- Java Runtime Environment (JRE), within which agents work;
- Kotlin Runtime Environment (KRE), which manages threads.

The platform functions cover:

- support for computational activities of agents (analysis of any set of elements and grouping);
- providing the facilitator with catalogs where agents can register their services according to the FIPA standard;
- providing an agent management system in accordance with the FIPA standard;
- providing a messaging service between agents.

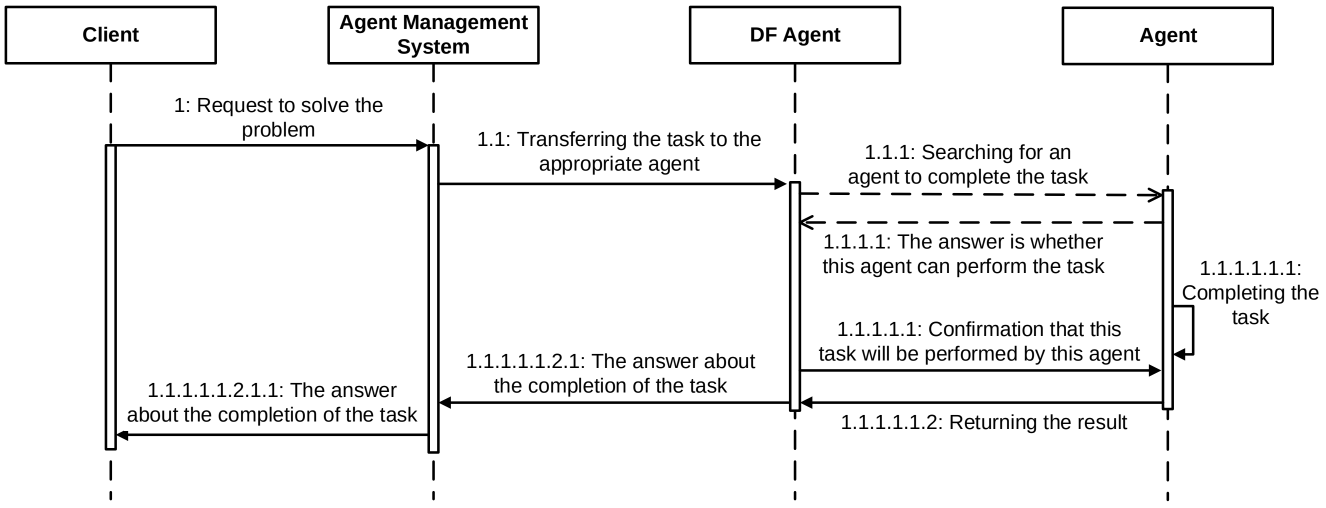

Client
Agent Management System
DF Agent
Agent
1: Request to solve the problem
1.1: Transferring the task to the appropriate agent
1.1.1: Searching for an agent to complete the task
1.1.1.1: The answer is whether this agent can perform the task
1.1.1.1.1.1: Completing the task
1.1.1.1.1: Confirmation that this task will be performed by this agent
1.1.1.1.1.2.1: The answer about the completion of the task
1.1.1.1.1.2.1.1: The answer about the completion of the task
1.1.1.1.1.2: Returning the result

Fig. 1. Sequence diagram for the case of successful completion of the task

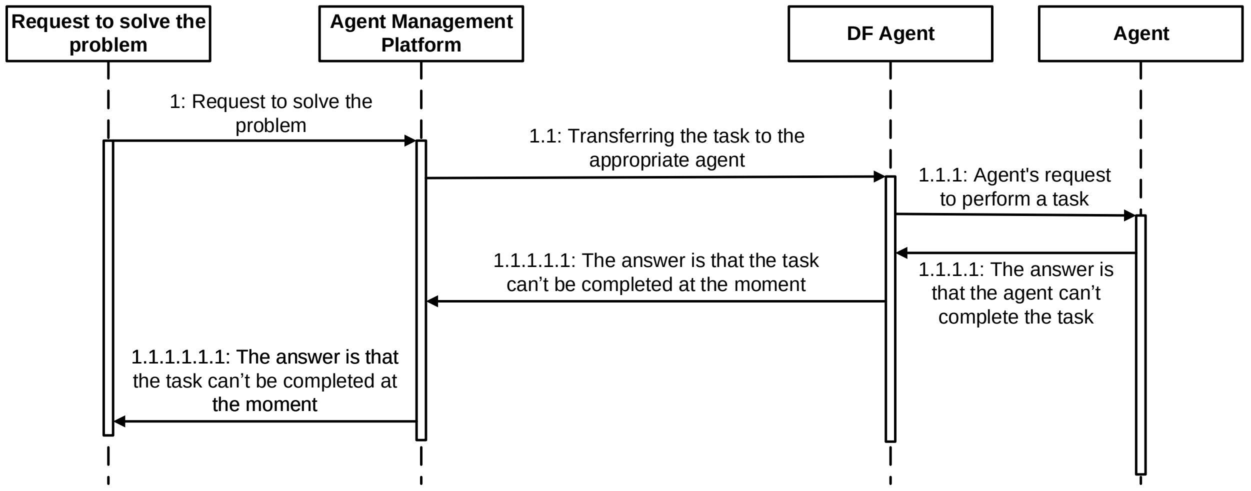

Request to solve the problem
Agent Management Platform
DF Agent
Agent
1: Request to solve the problem
1.1: Transferring the task to the appropriate agent
1.1.1: Agent's request to perform a task
1.1.1.1: The answer is that the agent can't complete the task
1.1.1.1.1: The answer is that the task can't be completed at the moment
1.1.1.1.1.1: The answer is that the task can't be completed at the moment

Fig. 2. Sequence diagram for a failed task

The users of the system which is developed are:

– programmers who are interested in working on the project through its further development;
– engineers or researchers who want to use the agent platform in their research work related to multi-agent modeling.

To solve the problem of writing unit tests for software, we decided to use two main tools, junit [23] and mock [24]. Junit is a software that allows you to create unit tests for programming languages running on the jvm. Main advantages of junit are [23]:

1) JUnit provides a wide range of annotations to define different types of tests, such as @Test, @Before, @After, @BeforeEach, @AfterEach, and others. This allows you to conveniently organize test scenarios and define preceding and following actions before the test;

2) support for parameterized tests. JUnit provides the ability to run the same test with different input parameters. This allows you to effectively test functionality with different input options.

The mockk tool allows you to use mock objects for testing, which allows you to achieve full unit testing of only the object we are testing, without regard to the implementation of other objects [24]. Its main advantages are:

1) ease of mocking, MockK provides a convenient and easy way to create mocks of objects for testing. It has a clear syntactic structure that makes it easy to create fake objects and set their behavior;

2) customization flexibility, MockK provides a wide range of features and options for customizing the behavior of mocks. You can set return values, monitor method calls, define exceptions, and much more.

Checks are an implementation of the Check class, which in turn is an implementation of the Predicate interface [25], which allows you to work with any data and only need to implement one check method, and therefore can be considered a functional interface. This provides such an advantage as implementing the interface in lambda. The class diagram for some standard components is shown in Fig. 3.

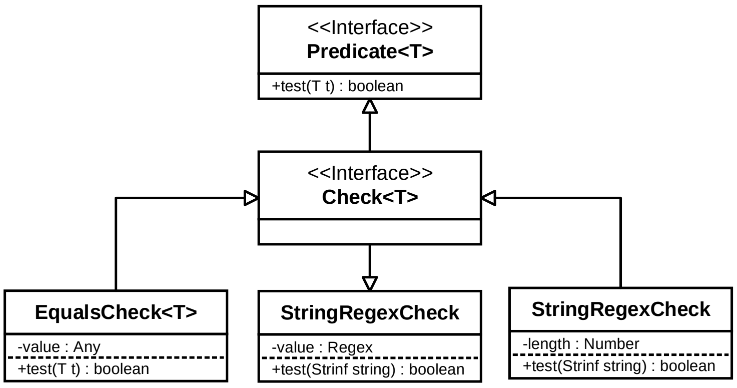


Fig. 3. Class diagram for the Check

To provide a clear mechanism for checking the agent's ability to perform tasks, we decided to use the chain of responsibilities pattern. A chain of responsibilities is a behavioral design pattern that allows requests to be passed along a chain of handlers. Each subsequent handler decides whether it can process the request itself and whether it is worth passing the request further down the chain. It makes it easy to implement a set of checks for each agent. The mechanism of this pattern is demonstrated in Fig. 4.

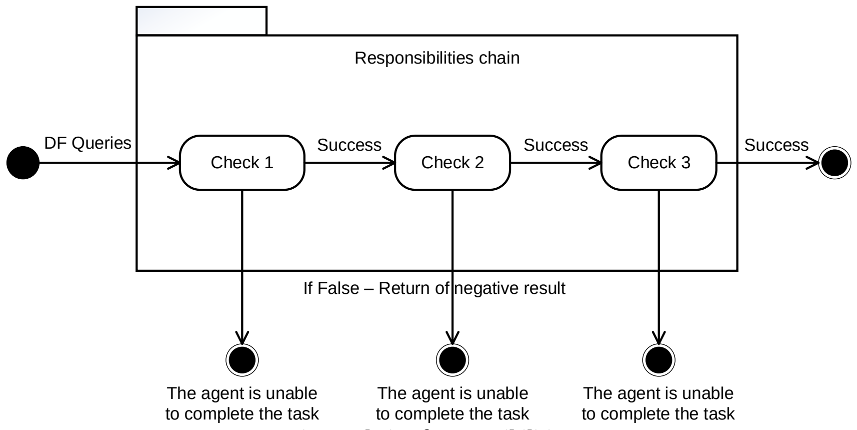

Responsibilities chain
DF Queries
Check 1
Success
Check 2
Success
Check 3
Success
If False – Return of negative result
The agent is unable to complete the task
The agent is unable to complete the task
The agent is unable to complete the task

Fig. 4. Chain of responsibilities

The messaging technology can be implemented using the kotlin SharedFlow tool. SharedFlow is a thread that shares messages between all collectors in broadcast mode. Its basic pattern of broadcast mode of use allows you to quickly deliver messages to a large number of agents in the system. SharedFlow is useful for broadcasting events that occur within an application to agents that can be created and deleted. In addition, this technology has such advantages over BroadcastChannel as simplicity, because it does not need to implement all the channel APIs, which provides faster and simpler implementation, buffer implementation, for more productive system operation, and it is error-resistant, because it cannot be closed except by a direct termination signal.

The repository pattern was chosen to save the data. A repository is an interface that specifies methods for saving, updating, and deleting information. Its implementation specifies where exactly the data should be stored and implements the corresponding methods. Its method of saving data is automatically connected to the start of the agent, which allows you to save all the data that the agent receives for further work with them. The factory method pattern was chosen for the standardized creation of agents. The factory method is a generative design pattern that defines a common interface for creating objects in a superclass, allowing subclasses to change the type of objects they create. It solves the problem of the lack of information about future types of agents in the system, meaning that using this pattern, it will not be difficult to add any type of new agent in the future. To create a new type of agent in the system, it is enough to implement the agent's behavior for performing tasks, use standard ones, or write your own conditions for checking whether the agent can perform the task, and specify additional characteristics (such as the name pattern and the method of saving). The factory method will assemble all this, as well as the standard parts of the agent implementation, and give us a ready-made agent to add to the system.

Finally, after adding all the architectural improvements, the structure of the multiagent system should look like in Fig.5. In addition to the standard components for a multiagent system, such as Agent, DirectorFacilitator, Bahavior, AgentManagementSystem, MessageTransportSystem, new components have been implemented to improve the architecture, such as Repository, BehaviorWithRepository, Check.

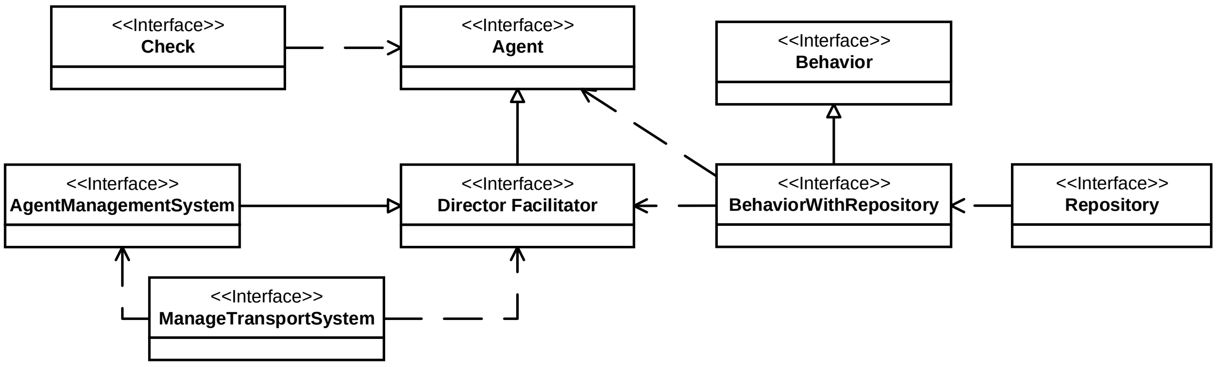


Fig. 5. Structure of a multi-agent system

For all components, standard implementations are created. These implementations realise the basic functionality of these components, such as saving data before processing it by the agent, standard implementation of some standard agents check conditions, such as greater than, regex to string, size check, etc. For the repository, a standard data storage in the program memory is created, with the possibility of adding a database.

As a result of the development, the goal of adding a standardized approach to development was achieved. Thus, a standardized approach to adding a condition for checking the ability of an agent to perform a task appeared in the agent class (fig. 6). To do this, we used the chain of possibilities pattern, added the check interface and its standard implementations, which cover the most common checks such as full match or checking the term against a regular expression. This allows the end user, i.e. the developer, to write standard checks faster, and if necessary, write their own check, which can be passed directly to agents using lambda expressions.

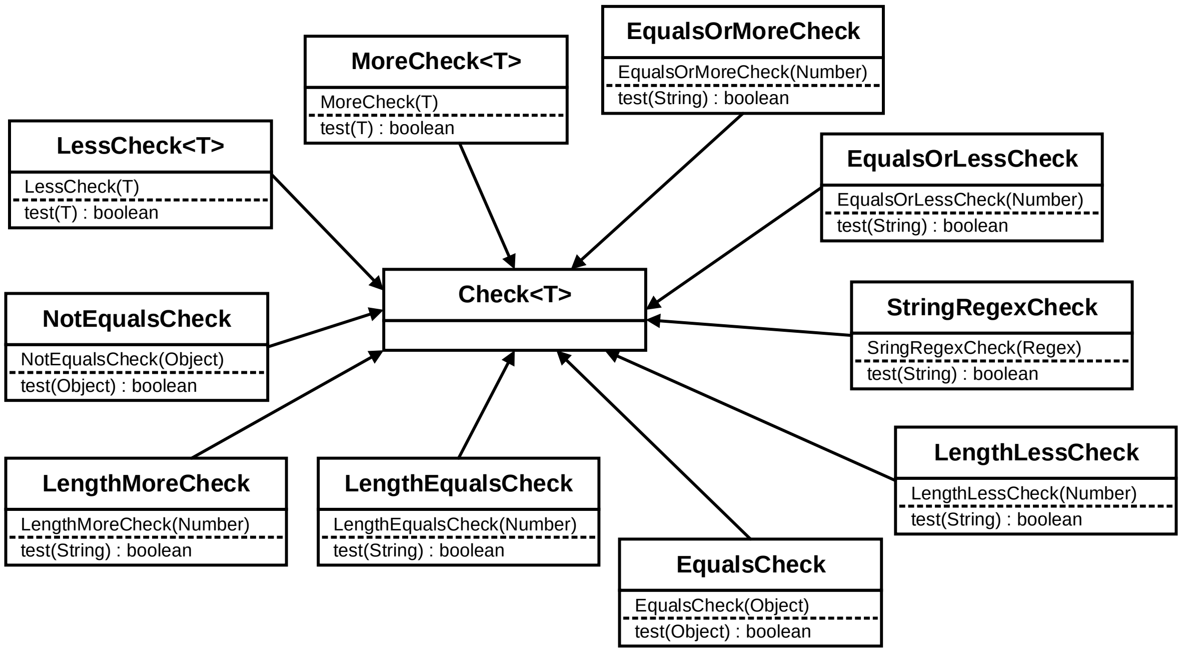


Fig. 6. Standard implementations of the check interface

To ensure a standardized approach to saving agent data, we developed the Repository interface and its standard implementation, which allows saving data to the program memory. To implement automatic saving, the program was supplemented with the BehaviorWithRepository component, which automatically accesses the repository when receiving data. All relevant classes that used the Behavior interface were updated to work with the new BehaviorWithRepository. If it is necessary to save data to a database, the developer only needs to implement the Repository interface for the required database management system. The main methods that are needed to save data are save, which allows you to save and find all the saved data. Additionally, the delete and findById methods can also be implemented if necessary. All those components and their methods are shown in Fig. 7.

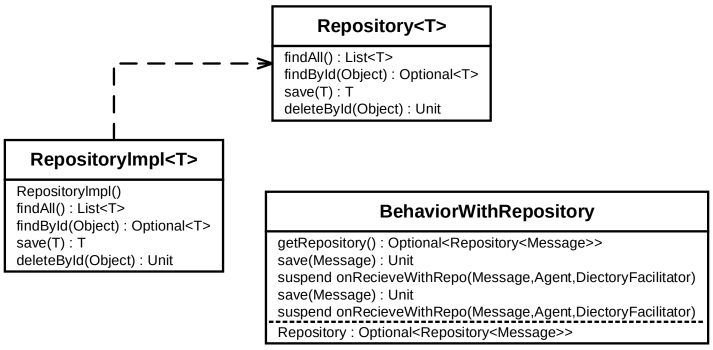

Repository<T>
findAll() : List<T>
findById(Object) : Optional<T>
save(T) : T
deleteById(Object) : Unit
RepositoryImpl<T>
RepositoryImpl()
findAll() : List<T>
findById(Object) : Optional<T>
save(T) : T
deleteById(Object) : Unit
BehaviorWithRepository
getRepository() : Optional<Repository<Message>>
save(Message) : Unit
suspend onRecieveWithRepo(Message,Agent,DiectoryFacilitator)
save(Message) : Unit
suspend onRecieveWithRepo(Message,Agent,DiectoryFacilitator)
Repository : Optional<Repository<Message>>

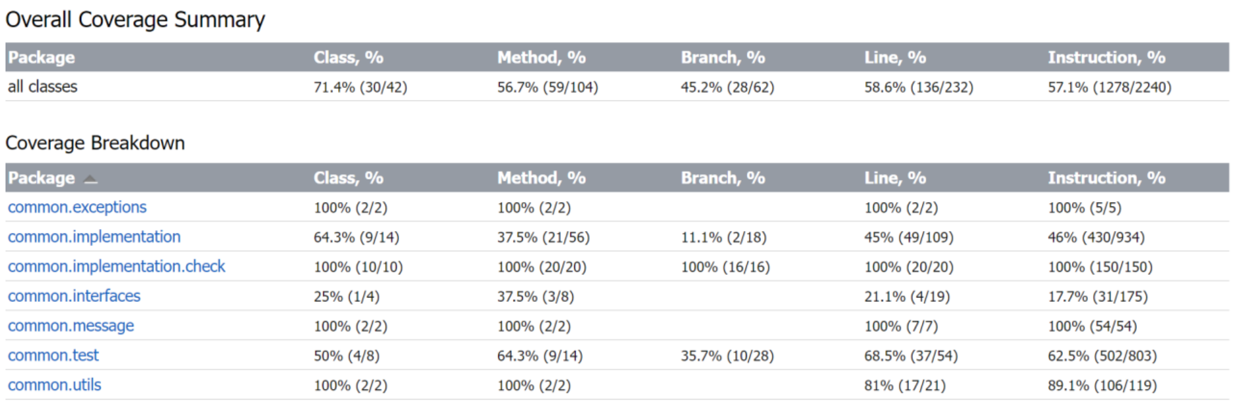
Fig. 7. New components for saving data

A new agent customization mechanism has been developed to easily conFig. agents and their fields, such as name, number, behavior, and other parameters. It provides for the automatic loading of parameters into the systems and the creation of agents using a standard template. If necessary, the number of parameters can be expanded. The main mechanism that makes this convenient is loading the required behavioral class from the configuration. This is achieved through the JVM reflection mechanism, that is, all you need to do to create the appropriate class is to write the full path to it. This method currently allows you to add 4 different agent settings, and the total number of them in the system is 6, which means that most parameters can be set using a configuration file. In addition, the next addition of agent parameters to the configuration file will take much less time due to the already implemented technology for this.

To achieve high code quality, the two main improvements were made: adding unit tests using junit technology and fixing all comments from the sonarlint plugin for automatic code analysis. This will allow us to achieve a level of code quality sometimes higher than that of competitors and ensure quality for the future, because in order for the program to work, it is necessary that all tests are executed as successful.

All the comments were analyzed and eliminated in the course of work on software quality. To eliminate some of them, we updated the libraries and used new elements that provide more stable code, while others were eliminated by removing unnecessary code or rewriting part of the code.

In addition, unit tests were added for all the main functionality of the system. We achieved 71.4% coverage of classes and 57.1% coverage of teams. Further coverage is complicated by the use of multi-threaded technologies. The coverage was measured using the kover plugin and corresponds to the task. The result is shown in Fig. 8.


Overall Coverage Summary

Package	Class, %	Method, %	Branch, %	Line, %	Instruction, %
all classes	71.4% (30/42)	56.7% (59/104)	45.2% (28/62)	58.6% (136/232)	57.1% (1278/2240)

Coverage Breakdown

Package	Class, %	Method, %	Branch, %	Line, %	Instruction, %
common.exceptions	100% (2/2)	100% (2/2)		100% (2/2)	100% (5/5)
common.implementation	64.3% (9/14)	37.5% (21/56)	11.1% (2/18)	45% (49/109)	46% (430/934)
common.implementation.check	100% (10/10)	100% (20/20)	100% (16/16)	100% (20/20)	100% (150/150)
common.interfaces	25% (1/4)	37.5% (3/8)		21.1% (4/19)	17.7% (31/175)
common.message	100% (2/2)	100% (2/2)		100% (7/7)	100% (54/54)
common.test	50% (4/8)	64.3% (9/14)	35.7% (10/28)	68.5% (37/54)	62.5% (502/803)
common.utils	100% (2/2)	100% (2/2)		81% (17/21)	89.1% (106/119)

Fig. 8. Results of modular code coverage

To check the performance of the developed agent platform, we have conducted load testing. We used the following parameters. Runtime environment: HP Pavilion Gaming 15 / AMD Ryzen 7 4800H (2.9 - 4.2 GHz) / 16 GB RAM / 512 GB SSD / nVidia GeForce GTX 1660 Ti Max-Q, 6 GB. As the results of launching agents shown, the number of agents increases, the CPU load increases. A significant increase in the number of agents can lead to an OutOfMemory exception. Thus, we can conclude that the number of agents is the main parameter that affects the reliability and efficiency of the developed agent platform.

Thus, the research conducted has shown that the use of Kotlin allows to create efficient and high-performance systems, which is especially important for multi-agent platforms. When developing an agent platform, it is necessary to consider a number of factors, and it is especially important to carefully follow the FIPA specifications. This ensures compatibility and interaction with other systems. Using the Kotlin programming language in combination with the FIPA standards allows for the creation of efficient systems, thanks to technologies such as coroutines and built-in support for parallelism.

Conclusions

This research focuses on the design and evaluation of a FIPA-compliant multi-agent platform. It reviews existing frameworks for developing multi-agent systems, highlights key components of such systems, and discusses the benefits of using Kotlin in the context of a multi-agent architecture.

Developing a FIPA-compliant multi-agent platform based on Kotlin is a challenging but promising task. Using Kotlin allows you to build efficient and high-performance systems, which is especially important for multi-agent platforms. However, there are a number of factors to consider when developing such a platform. It is necessary to carefully follow the FIPA specifications when implementing system components. This is a key aspect to ensure compatibility and interoperability with other systems. It is important to ensure horizontal scalability. Developing a FIPA-compliant multi-agent platform in Kotlin is a complex task that requires careful planning and implementation. The successful solution of this task will make a significant contribution to the development of multi-agent systems and their application in various fields.

The scientific contribution of this research is the creation of a modern multi-agent platform that complies with the FIPA standard. Using Kotlin in combination with a multi-agent architecture allows you to create efficient and high-performance systems with minimal overhead, thanks to technologies such as coroutines and built-in support for parallelism. The key components of a multi-agent system are agents, a directory facilitator (DF), an agent platform (AP), and a message transport system (MTS), which provide efficient interaction and communication between agents. Two metrics are used to ensure the quality of the system: the percentage of code covered, and the system score based on comments. We achieved 71.4% class coverage and 57.1% command coverage. Further coverage is complicated by the use of multi-threaded technologies. During development, 16 comments were identified and fixed. This allows us to achieve a high level of code quality and guarantee quality for the future. The research is promising in the context of integrating the capabilities of modern programming languages, such as Kotlin, with the multi-agent paradigm, which opens new horizons for the development of efficient and scalable solutions in the field of distributed intelligent systems.

References

1. Jeong, CheonSu. A Study on the Implementation Method of an Agent-Based Advanced RAG System Using Graph, 2024. arXiv preprint. URL: https://arxiv.org/abs/2407.19994.
2. de Lima, G. L., & de Aguiar, M. S. Towards a Docker-based architecture for open multi-agent systems. IAES International Journal of Artificial Intelligence, 2024, 13(1), 45–56. https://doi.org/10.11591/ijai.v13.i1.pp45-56.
3. Shahzad, R., Aslam, M., Al-Otaibi, S. T., Javed, M. S., Khan, A. R., Bahaj, S. A., & Saba, T. (2024). Multi-Agent System for Students Cognitive Assessment in E-Learning Environment. IEEE Access, 2024, 12, 15458–15467. https://doi.org/10.1109/ACCESS.2024.3356613.
4. Wooldridge M. An Introduction to Multi-Agent Systems. 2nd Ed. Wiley Publ. 2009. 488 p.
5. Rogushina, Y. V. Software agents: Definitions, taxonomies and models. Upravlyayushchie Sistemy i Mashiny, 2001 (5), 39–46.
6. The Foundation for Intelligent Physical Agents (FIPA). URL: http://www.fipa.org/
7. JAVA Agent DEvelopment Framework. URL: https://jade.tilab.com/
8. Foundation for Intelligent Physical Agents Operating System (FIPA-OS). URL: http://fipa-os.sourceforge.net/index.html.
9. Winikoff, M. Jack Intelligent Agents: An Industrial Strength Platform. In: Multi-Agent Programming. Multiagent Systems, Artificial Societies, and Simulated Organizations, 2005, vol 15. https://doi.org/10.1007/0-387-26350-0_7
10. The Open Agent Architecture. URL: https://www.ai.sri.com/~oaa/
11. German, Ernesto, and Leonid Sheremetov. An Agent Framework for Processing FIPA-ACL Messages Based on Interaction Models. Lecture Notes in Computer Science. 4951 LNCS. N.p., 2008. 88–102.
12. Wrona Z., Buchwald W., Ganzha M., Paprzycki M., Leon F., Noor N., Pal C.-V. Overview of Software Agent Platforms Available in 2023. Information. 2023. Vol. 14, iss. 348. DOI: 10.3390/info14060348.
13. Cardoso R.C., Ferrando A. A Review of Agent-Based Programming for Multi-Agent Systems. Computers. 2021, Vol. 10, iss. 16. DOI: 10.3390/computers10020016.
14. Balachennaiah, P., and J. Chinna Babu. Application of Multi Agent Systems for Advanced Energy Management in Cyber Physical Hybrid Microgrid Systems. 2024. Vol. 4951 LNCS, pp. 88–102. https://doi.org/10.1007/978-3-540-79488-2_7
15. IBM Agent Builder. URL: https://www.ibm.com/docs/en/capm?topic=builder-overview-agent.
16. Mava. URL: https://www.instadeep.com/2021/07/mava-a-new-framework-for-distributed-multi-agent-reinforcement-learning/.
17. Houhamdi Z. Multi-Agent System Testing: A Survey. International Journal of Advanced Computer Science and Applications. 2011. Vol. 2, iss. 6. DOI: 10.14569/ijacsa.2011.020620.
18. Dorri A., Kanhere S. S., Jurdak R. Multi-Agent Systems: A Survey. IEEE Access. 2018. Vol. 6. P. 28573-28593. DOI: 10.1109/ACCESS.2018.2831228.
19. Botti V., Mariani S., Omicini A., Julian V. Multi-Agent Systems. MDPI – Multidisciplinary Digital Publishing Institute. 2019. 392 p. https://doi.org/10.3390/books978-3-03897-925-8.
20. Kotlin. URL: https://kotlinlang.org
21. Coroutines. URL: https://kotlinlang.org/docs/coroutines-overview.html.
22. Linter IDE Tool & Real-Time Software for Code. URL: https://www.sonarsource.com/products/sonarlint/
23. Junit. 2023. URL: https://junit.org/junit5/.
24. MockK. 2023. URL: https://mockk.io/.
25. Predicate. 2023. URL: https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html.

Oleksandr Karataiev **Олександр Каратаєв**	Graduate student of the Department of Software Engineering, Kharkiv National University of Radioelectronics https://orcid.org/0009-0007-6654-1327 e-mail: tosanik@gmail.com	аспірант кафедри програмної інженерії, Харківський національний університет радіоелектроніки