DOI: 10.20998/2079-0023.2026.01.18
UDC 004.43; 004.051; 811.93

R. O. GAMZAYEV, Candidate of Technical Sciences (PhD), Docent, Associate Professor at the Department of Intelligent Software Systems and Technologies, V. N. Karazin Kharkiv National University, Svobody Sq. 6, Kharkiv-22, Ukraine, 61022; e-mail: rustam.gamzayev@ karazin.ua; ORCID: https://orcid.org/0000-0002-2713-5664
M. V. TKACHUK, Doctor of Technical Sciences, Professor, Professor at the Department of Intelligent Software Systems and Technologies, V. N. Karazin Kharkiv National University, Svobody Sq. 6, Kharkiv-22, Ukraine, 61022; e-mail: mykola.tkachuk@karazin.ua; ORCID: https://orcid.org/0000-0003-0852-1081
G. S. LEGENKYI, Student, V. N. Karazin Kharkiv National University, Svobody Sq. 6, Kharkiv-22, Ukraine, 61022; e-mail: lehenkyi2021ki11@student.karazin.ua;ORCID: https://orcid.org/0009-0006-4444-5759

DOMAIN-SPECIFIC LANGUAGE FOR INTELLIGENT TESTING OF MICROSERVICE SOFTWARE SYSTEMS BASED ON MOCK-OBJECT TECHNOLOGY

Testing service-oriented and microservice-based systems is challenging due to strong dependencies on external services and distributed environments. Mock-based testing is widely used to address this issue; however, existing solutions rely on low-level configuration mechanisms that increase complexity and reduce maintainability. This paper proposes a domain-specific language (DSL) for intelligent specification and generation of mock services. The approach enables high-level, domain-oriented description of mock behavior and supports transformation into executable configurations for existing mocking platforms. The proposed solution aims to improve readability, reduce configuration effort, and enhance testing efficiency in distributed software systems. The paper reviews existing tools for mock-based testing of service-oriented applications and provides a comparative analysis based on ease of use, configuration flexibility, and the complexity of supported test scenarios. The analysis shows that existing solutions consistently trade off between expressiveness and accessibility, a limitation that the proposed DSL aims to address. To evaluate the proposed approach, a comparative experiment was conducted across four test scenarios of varying complexity. DSL-based configurations were compared against equivalent configurations defined without the DSL, using three metrics: specification size (SLOC), maximum nested block depth, and maintainability index. The results show that the DSL reduces cyclomatic complexity. The composite quality score of DSL-based configurations exceeds the baseline by 38 % on average. These findings confirm that the proposed DSL simplifies the creation of mock services and makes distributed system testing more accessible to a wider range of project participants.

Keywords: domain-specific language, intelligent approach, software, service-oriented architecture, microservice, testing, mock-object, distributed system, metric, quality.

Introduction. Service-oriented architecture (SOA) and microservice-based systems have become widely adopted due to their scalability, flexibility, and support for modular development. However, these architectural styles also increase testing complexity, since individual services often depend on external APIs, third-party platforms, and independently evolving infrastructure. Such dependencies may be unavailable during development, unstable in test environments, or costly to invoke repeatedly, which reduces the controllability and reproducibility of testing.

To address this issue, software teams widely employ mock services [1] that simulate the behavior of external systems and enable isolated, repeatable, and controlled validation of service interactions. Mocking approaches have been widely known since the early development of unit testing practices in the 1990s, particularly within the xUnit family of frameworks introduced by Kent Beck [2]. In this context, mock objects were initially used to isolate units of code by replacing their dependencies with controllable test doubles. In modern distributed environments, this idea has evolved from unit-level isolation to service-level simulation. Mocking is especially valuable in distributed environments because it supports parallel development, facilitates the simulation of both normal and exceptional scenarios, and reduces the impact of unavailable dependencies on the testing process.

Despite the availability of mature mocking platforms, the practical configuration of mock behavior remains difficult. In many cases, mock scenarios are specified through low-level mappings, programmatic APIs, or tool-specific configuration formats, which increases the entry barrier and limits the accessibility of such solutions for non-programmer stakeholders. As a result, there is a need for higher-level mechanisms that preserve the expressive power of existing platforms while simplifying the specification of service behavior.

A promising direction is the use of domain-specific languages (DSLs), which enable users to describe solutions within a particular application domain at a higher level of abstraction than general-purpose programming languages [2,3]. In software engineering, DSLs are valued for their expressiveness, readability, and reduced technical overhead. These properties make them a suitable candidate for describing mock-service behavior in a concise and domain-oriented form.

This paper proposes a domain-specific language for an intelligent testing of service-oriented software systems based on mock-object technology. The proposed approach allows users to define mock scenarios through high-level constructs, edit them in a dedicated web interface, parse them on the server side, and transform them into executable configurations for a mock server. Such an approach is expected to improve readability, reduce configuration effort, and support automation in modern CI/CD workflows.

Brief overview of related work. Research on testing of service-oriented and distributed systems includes a variety of approaches aimed at improving test isolation, automation, and maintainability. One of the most widely adopted techniques is the use of mock objects, which

simulate the behavior of external dependencies and enable controlled validation of system interactions. An empirical study of open-source projects demonstrates that mocking is extensively used in practice, primarily to isolate components, emulate external services, and verify interactions between system elements [4]. At the same time, the study highlights that the configuration of mock behavior is often manual and tool-specific, which increases complexity and reduces accessibility.

In service-oriented settings, mock-based testing is supported by specialized platforms such as WireMock, MockServer, Hoverfly, Beeceptor, and Mocky. These tools enable request matching, response generation, proxying, and interaction verification [5]. However, they differ significantly in usability and capability, as illustrated in Table 1. Tools with high configuration flexibility (WireMock, MockServer) expose complex, low-level interfaces – JSON mappings, programmatic APIs, CLI-driven setup – that require substantial developer expertise. Conversely, simpler tools (Mocky, Beeceptor) offer accessible web-based interfaces but support only basic scenarios and lack the expressive power needed for complex service interactions. This trade-off between usability and expressiveness remains a consistent limitation across existing solutions.

In addition to traditional mocking approaches, several studies explore automated and specification-based techniques for testing service-oriented systems. In particular, specification-driven approaches enable automatic generation of test cases for REST APIs based on formal service descriptions, improving coverage and reducing manual effort [6]. These approaches demonstrate the benefits of higher-level abstractions in testing but are primarily focused on test generation rather than behavior simulation.

Another important research direction is the use of domain-specific abstractions and formal models for defining expected system behavior. For example, advanced approaches to test oracle generation allow modeling uncertainty and multi-level validation in complex systems such as cyber-physical environments [7]. These techniques improve the reliability of test validation but often introduce additional complexity in the specification.

In parallel, artificial intelligence techniques are increasingly applied to software testing. AI-based methods enable automation of test analysis, pattern recognition, and optimization of test suites [8]. However, these approaches are typically focused on test generation and analysis rather than on modeling external service behavior through mocks.

More recently, industrial solutions have started to explore AI-driven mocking and service virtualization. These approaches automatically generate mock services from recorded traffic and dynamically adapt system behavior for testing purposes [9]. While they significantly reduce manual configuration effort and increase realism, they often rely on complex infrastructure and provide limited transparency and control over mock behavior specification.

Overall, existing research highlights the importance of abstraction, automation, and simulation in testing distributed systems. However, there remains a gap between expressive power and usability, particularly in defining mock behavior in a simple and domain-oriented way. This motivates the need for higher-level approaches, such as the domain-specific language proposed in this work.

Problem statement. The analysis of existing tools and research presented above reveals several challenges in mock-based testing of service-oriented systems. Although prior studies emphasize the importance of abstraction, automation, and simulation, current solutions still show a clear gap between usability and expressive power.

In particular, tools that provide sufficient flexibility for modeling complex interaction scenarios (such as WireMock or MockServer) rely on low-level and technically complex configuration mechanisms. At the same time, tools that are accessible to a wider audience (such as Mocky or Beeceptor) support only simple scenarios and lack the expressive power required to model realistic service interactions.

This trade-off is not limited to issues of user interface or ease of use. It reflects the absence of an appropriate level of abstraction between the tester's intent and the configuration mechanisms underlying existing solutions – a gap that remains insufficiently addressed in both research and industrial practice.

To better illustrate this limitation, consider the typical interaction model described earlier (see fig. 1). In production environments, a service-oriented system communicates directly with external services. In testing environments, these dependencies are replaced by a mock server whose behavior is controlled through a configuration mechanism. However, regardless of the complexity of the test scenario, such configuration is currently defined using low-level technical artifacts, such as configuration files, code written in a general-purpose programming language that interacts with the mock server API, or CLI commands. As a result, the specification of test scenarios becomes tightly coupled with implementation details, which reduces clarity and increases effort.

Table 1 – Overview of existing approaches

Criterion	Mocky	WireMock	MockServer	Beeceptor	Hoverfly
Ease of use	High	Medium	Medium	High	Low
User interface	Web UI	CLI, Java, Web UI	CLI, Java, UI	Web UI	CLI, Web UI
Configuration flexibility	Low	High	High	Low	Medium
Complexity of testing scenarios	Simple	Complex	Complex	Medium	Medium

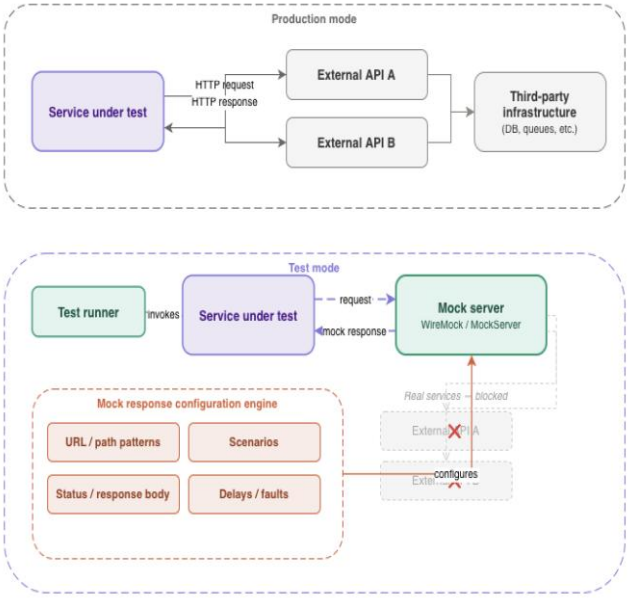

Fig. 1. Mock server-based testing architecture

This situation leads to several specific problems:

1. Low-level specification of mock behavior. Mock scenarios are described in implementation-oriented terms rather than domain concepts. Testers are forced to work with request matching rules and response structures instead of directly expressing test intent.
2. High configuration and maintenance effort. As the number of mock services and endpoints grows, the amount of configuration artifacts increases significantly. The lack of concise and reusable constructs leads to duplication, reduced readability, and higher maintenance effort.
3. Limited participation of stakeholders. The need for programming skills or deep knowledge of specific tools limits the involvement of QA engineers, test analysts, and domain experts. This creates bottlenecks and increases dependency on developers.
4. Limited support for automation and integration. Existing configuration approaches are not well suited for structured, version-controlled, and CI/CD-integrated workflows, which makes it difficult to incorporate mock-based testing into modern DevOps practices.

To address these problems, this work proposes the design and implementation of a domain-specific language (DSL) for specifying mock service behavior. The proposed approach introduces a higher level of abstraction for describing service behavior, enables intent-oriented scenario definition, reduces configuration complexity through concise and expressive syntax, lowers the entry barrier for non-developer stakeholders, and supports automatic transformation of DSL specifications into executable configurations (e.g., for WireMock).

Main materials and methods. We defined the lexical rules that break the text stream into individual tokens. These rules serve as the foundation for subsequent syntactic analysis and the construction of the logical structure of the code. Thanks to them, each element a string, a number, or a comment will have an unambiguous definition, which is key to the stability and accuracy of the system's operation. The core lexical components of the DSL include tokens for strings, numbers, boolean values, and others that form the basic elements of the syntax. Base lexical components of our DSL presented at Table 2.

Table 2 – Lexical components our DSL

Token	Value	Description
STRING	"" ('\\' ["\\] | ~["])* ""	A quoted string with escape sequence support
TRIPLE_STRING	""" ('\\"""' | .)*? """	A triple-quoted string that can contain any content
NUMBER	'-'? [0-9]+('.' [0-9]+)?	Integer or decimal number
BOOLEAN	'true' | 'false' | 'TRUE' | 'FALSE'	Boolean values
LINE_COMMENT	'//' ~[\r\n]* -> skip	Inline comment
BLOCK_COMMENT	'/*' .*? '*/' -> skip	Block comment
WS	[\t\r\n]+ -> skip	Whitespace, tabs, and newlines
Token	Value	Description

Development of this specialized language, each test endpoint is described as a single block beginning with the “DEFINE ENDPOINT” keyword construct, followed by a block enclosed in curly braces that contains the endpoint description.

Formally, an endpoint definition can be represented as a tuple:

$$E = \langle R, P, D \rangle, \quad (1)$$

where R is the request definition (method, path, matching conditions);

P is the response definition (direct, fault, or proxy response type, along with optional delays and webhooks);

D is an optional textual description.

Each component given in (1) is specified through a dedicated DSL block: DEFINE REQUEST, DEFINE RESPONSE, and SET DESCRIPTION TO, respectively.The request is described inside a block beginning with DEFINE REQUEST. It contains a number of mandatory commands, such as setHttpMethodCommand, setUrlMatchTypeCommand and setPathCommand. Additional settings can also be specified, namely: priority definition, query parameters, headers, cookies, form parameters, and various rules for the incoming request body.

The response is defined inside the DEFINE RESPONSE block. Within this block, one of three response types can be specified: a direct response, a fault response, or a proxy response. For a direct response, the ADD DIRECT RESPONSE block is used, containing commands for setting the status code, the response body type, and the corresponding content. To simulate an error condition, ADD FAULT RESPONSE is used, along with the fault

type set via setFaultTypeCommand. In the case of a proxy response, the definition is formed through ADD PROXY RESPONSE, which contains a command for specifying the remote service URL, along with additional parameters for host rewriting and template usage.

Optionally, a description can be defined via the setEndpointDescriptionCommand. Individual blocks in the grammar extend the capabilities for describing service response behavior. If necessary, negation is possible through the keyword "NOT".

The grammar provides a strictly formalized description of conditions for request validation. Commands for them are written using special constructs. There are simple and composite conditions. All of them follow the same format, where the condition type and its value are specified. They allow the description to be broken down into structural blocks, each responsible for a specific part of the test endpoint definition.

The simplest validation method only checks for the presence of a request body. It does not pay attention to its content. This is convenient when you need to ensure that the request is not empty, and specific data is not expected.

The next validation method focuses on the structure of the request body, such as XML. It compares the content with an expected document of this format. The comparison is performed character by character, and XMLUnit can be activated. This allows you to ignore minor differences, for example, in formatting.

JSON structure validation compares the request body with an expected object of this format. It is possible to customize the validation to disregard the order in arrays. You can also ignore extra elements that are not present in the expected value.

Endpoint definition is a key element described using the DEFINE ENDPOINT construct. It brings together both the request and the response, serving as the wrapper that formalizes how a client is expected to interact with the system under development. In addition to its core components, an endpoint definition can be supplemented with an optional description using the SET DESCRIPTION TO command, which allows an explanatory note to be added that clarifies the purpose or specific characteristics of the endpoint.

The constituent parts of a request definition include the QUERY_PARAMS, FORM_PARAMS, HEADERS, and COOKIES constructs. All of them are optional, and each is responsible for a distinct type of input data. They contain dedicated blocks that describe conditions for specific fields, with each block consisting of one or more rules. Each rule specifies a field name and defines the criteria it must satisfy. This provides a highly flexible way of describing under which conditions a request should trigger an endpoint's response (see fig. 2).

This example declares an endpoint for handling user login via a POST request at the path "/api/login". In response, the server returns a status code of 200 with a JSON body containing the message "Login successful". The "DEFINE REQUEST" construct allows specifying which HTTP request should trigger the endpoint response. First, the request method is defined – for example, GET, POST, PUT, DELETE, PATCH, OPTIONS, TRACE, or ANY. Then the path is set according to the specified type.

```
DEFINE ENDPOINT {
    SET DESCRIPTION TO "Login API endpoint"
    DEFINE REQUEST {
        SET METHOD TO POST
        SET URL_MATCH_TYPE TO PATH
        SET REQUEST_PATH TO "/api/login"
    QUERY_PARAMS {
        ADD QUERY_PARAM_RULE {.. }
    }
    HEADERS {...}
    COOKIES { ADD COOKIE_RULE {... }
    }
    FORM_PARAMS {ADD FORM_PARAM_RULE {...}
    }
    DEFINE RESPONSE {
        ADD DIRECT RESPONSE {
            SET STATUS_CODE TO 200
            SET BODY_TYPE TO JSON
            SET VALUE TO "{\"message\": \"Login successful\"}"    }  }
}
```

Fig. 2. Definition of ENDPOINT

It can be simple, include parameters, or even use regular expressions. The path is defined using the following types: PATH, PATH_AND_QUERY, ANY_URL, PATH_AND_QUERY_REGEX, PATH_TEMPLATE, and PATH_REGEX.

Conditions can be simple or composite. A simple condition checks a single value. The most basic of them is EQUALS, which verifies whether the value matches exactly. Next is CONTAINS, which checks whether the value includes a substring. MATCHES_REGEX is applied for regular expressions, while types such as MATCHES_JSON_PATH or MATCHES_XPATH are used to validate against a specific path within a JSON or XML structure respectively. There are also conditions for validating more complex formats. For example, MATCHES_JSON_SCHEMA checks whether the incoming JSON conforms to a given schema. EQUALS_JSON or EQUALS_XML compare the corresponding request bodies for identity. For dates, EQUALS_DATE_TIME, BEFORE, or AFTER can be used to verify time boundaries or exact matches. Composite conditions combine multiple simple ones. Logical operations can be used depending on the type. There is also VALUES_INCLUDE, which checks whether values from a set of conditions are included in the input data, and VALUES_ARE_EXACTLY, which requires an exact match of the set of values.

Furthermore, any simple or composite condition can be negated to indicate that it must not be satisfied (see fig.3).

Response to a request is defined through the responseDefinition block, which can be direct, fault-based, or a proxy. It is possible to add delays and webhooks to create realistic testing scenarios.

A direct response is defined within the "ADD DIRECT RESPONSE" block and allows specifying a custom response. It can include a status code, body type, and content.

Headers can also be added optionally. Dynamic templating can be enabled for greater flexibility. The status

code is set using the straightforward command "SET STATUS_CODE TO".

```
QUERY_PARAMS {
  ADD QUERY_PARAM_RULE {
    SET NAME TO "filter"
    ADD CONDITION {
      SET REQUEST_CONDITION_TYPE TO OR
      ADD CONDITION {
        SET REQUEST_CONDITION_TYPE TO EQUALS
        SET VALUE TO "active"
      }
      ADD CONDITION {
        NOT (
          SET REQUEST_CONDITION_TYPE TO EQUALS
          SET VALUE TO "inactive"
        )
      }
    }
  }
}
```

Fig. 3. Usage of condition

Body type is defined via "SET BODY_TYPE TO", with supported formats including JSON, XML, HTML, TEXT, and BASE64. The response content is specified using the "SET VALUE TO" command. Headers are added optionally through "HEADERS".

Templating can be activated via setDynamicResponseTemplatingCommand, which allows generating dynamic content in the response. An example of this construct is shown in the Fig. 4.

```
DEFINE RESPONSE {
  ADD DIRECT RESPONSE {

    SET STATUS_CODE TO 200
    SET DYNAMIC_RESPONSE_TEMPLATING TO TRUE
      HEADERS {
        ADD HEADER {
          SET NAME TO "headerName"
          SET VALUE TO "headerValue"
        }
        ADD HEADER {
          SET NAME TO "headerName"
          SET VALUE TO "headerValue"
        }
      }

      SET BODY_TYPE TO TEXT
      SET VALUE TO "response body query"
  }
}
```

Fig. 4. Dynamic Response

A fault response is defined through "ADD FAULT RESPONSE" and is used to simulate various types of errors. It allows modeling situations where a connection is closed without a response, or when malformed data is sent. The fault type is set via setFaultTypeCommand. The list of possible values is provided in Table 3.

A proxy response is defined in the "ADD PROXY RESPONSE" block and allows redirecting a request to another URL. In this type of response, the target address can be set using the setUrlCommand.

Delays for a response are defined in the "DELAY" block and allow simulating processing time or network latency. If no delay is needed, the type corresponding to the value NO_DELAY is used. For a fixed delay, FIXED is used. To simulate a random delay, RANDOM_UNIFORM can be applied. All types support configuring time units via timeUnits, which can be milliseconds, seconds, minutes, or hours.

Table 3 – Falt response values

Fault Type	Description
NO_FAULT	No fault.
CLOSE_SOCKET_ WITH_NO_RESPONSE	Closes the connection without a response.
SEND_BAD_HTTP_DATA_ THEN_CLOSE_SOCKET	Sends malformed data and then closes the socket.
PEER_CONNECTION_RESET	Simulates a situation where the connection is reset at the network level.
SEND_200_RESPONSE_THEN_ BAD_HTTP_DATA	First sends a valid response, then sends malformed data.

Webhooks are placed in the "WEB_HOOKS" block and allow additional HTTP requests to be sentautomatically when an endpoint is called. Each one is defined via webHookDefinition. Within it, the request method is defined using setHttpMethodCommand, the URL is specified via setUrlCommand, and headers can also be added. The body type is defined through setResponseBodyTypeCommand. The actual content is set via setValueCommand. Additionally, a delay before sending the webhook can be configured using responseDelay. This structure provides great flexibility and allows not only generating standard, error, or proxy responses, but also extending functionality through integrations with other services.

Results and discussion. Developed for creating and managing mock services in SOA, this system uses a DSL to simplify the process. Its modular architecture allows for adding new features or adapting to changing requirements. It consists of four main components: a web interface, Java-based middleware, a MongoDB database, and WireMock. The component diagram of the system is presented in fig 5.

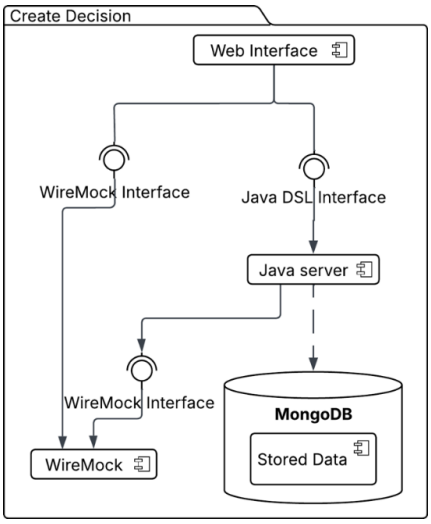

Fig. 5. Component diagram

The middleware is implemented in Java using the Spring Boot framework. It is responsible for processing the DSL code received from the web interface. For parsing, it uses ANTLR to analyze the syntax and create an AST. Based on this tree, the tool generates mock service configurations in a format compatible with WireMock. The service also interacts with MongoDB to store DSL scripts, processing statuses, and drafts.

MongoDB serves as the primary storage for DSL scripts, parsing results, and temporary saves. The system uses two main collections: definition_attempts and quick_saves. The first contains all code interpretation attempts, including success status and error logs, allowing for history tracking and diagnostics. The second stores the latest version of the code without creating a full configuration, which is useful for intermediate saves during editing. All storage logic is built using Spring Data MongoDB.

The web interface is the primary point of user interaction. Built with React, TypeScript, and Vite, it provides a fast and responsive user experience. It features the Monaco Editor for writing and editing DSL code, offering syntax highlighting, autocompletion, and real-time validation (see fig.6). The code entered by the user is sent to the backend via an HTTP/REST API for processing.

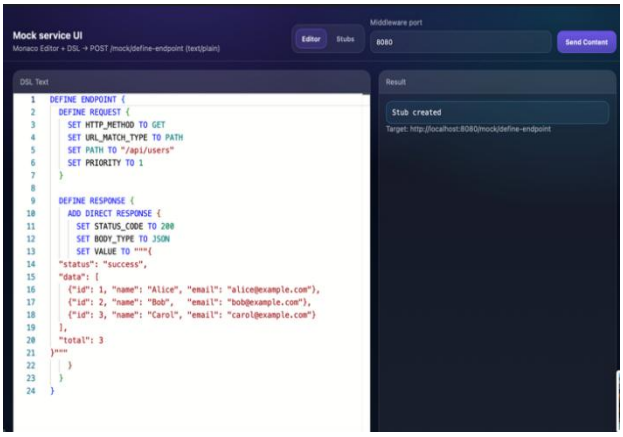

Fig. 6. Web interface view

WireMock is the core engine for setting up mock services. It is deployed in a Docker container to ensure an isolated environment and simplify deployment. WireMock functions as a standalone service that receives HTTP requests containing configurations from the Java middleware, based on which it creates simulated endpoints.

To evaluate the proposed approach, three metrics were selected. Configuration size is measured as the number of non-empty source lines (SLOC) [10], which reflects the effort required to author and maintain specifications. Structural complexity is captured by Maximum Nested Block Depth - the deepest level of block nesting along any path through the specification, which correlates with comprehendsion difficulty [11]. Finally, overall maintainability is assessed using the Maintainability Index (MI), a composite score on a 0–100 scale that combines Halstead Volume, Cyclomatic Complexity, and lines of code [12].

We focused on how well the solution supports realistic testing needs: how quickly and unambiguously an endpoint can be described in the DSL, how accurately request-matching rules are applied, how flexible the response configuration is, and whether the resulting behavior remains predictable across multiple scenarios.

First, we created a simple baseline endpoint to confirm that the system can register a stub and return a fixed response without any additional request constraints. This case represents the “happy path” where a client sends a request and receives a predefined JSON payload. It allowed us to validate the basic flow end-to-end: parsing the DSL, creating the stub, and serving the response.

Next, we introduced a header-dependent scenario to test conditional behavior based on request metadata. In this experiment, the response depends on the presence and value of a specific HTTP header. This verifies that the proposed approach supports matching rules beyond the URL and method and can model real-world cases where different users or roles produce different outcomes. For example, we created an endpoint whose returned payload changes according to the value of an X-Role (or similar) request header.

After that, we tested more complex behavior where the response depends on the request body. We created POST endpoints that accept a JSON payload (e.g., containing fields such as amount, currency, and status) and return different responses depending on the provided values. This step demonstrates that the DSL can express request-body conditions and that the stubbed behavior can simulate multiple branches such as approved/declined/pending, which is typical for payment-like or workflow-like integrations.

In addition, we verified dynamic response templating. Specifically, we created an endpoint that reads the value of a request header and injects it into the JSON response using WireMock templating.

Finally, we executed a series of curl requests to validate the created stubs in practice, including POST requests with JSON bodies, form-like bodies, and different HTTP methods against the same path (e.g., GET /info vs POST /info). This confirmed that the mock service correctly differentiates requests by method, header constraints, and body conditions, and that it can provide predictable responses suitable for integration testing and local development with no request body and no matching rules.

To aggregate the three metrics into a single quality indicator, each metric was normalized to the [0, 1] range using min-max scaling. For SLOC and Depth (lower is better), the normalization was inverted. For MI (higher is better), standard normalization was applied. The composite score was computed as the unweighted mean of the three normalized values:

$$\text{Score} = \frac{\text{norm\_inv}(\text{SLOC}) + \text{norm\_inv}(\text{Depth}) + \text{norm}(\text{MI})}{3}. \quad (2)$$

On average, the DSL approach achieves a composite score of 0.58 compared to 0.42 for the baseline configuration approach, representing an improvement of approximately 38 %.

Conclusions and future work. Conclusions and future work. The paper presented the domain-specific

language for intelligent specification and generation of mock-objects in service-oriented and microservice architectures. The proposed DSL provides a higher level of abstraction over existing mocking platforms, enabling concise, intent-oriented description of mock behavior without requiring some deep knowledge of tool-specific configuration mechanisms. The elaborated software solution was implemented as a modular platform comprising a React-based web editor with the Monaco Editor, a Spring Boot middleware using ANTL [13] for DSL parsing and AST construction, a MongoDB persistence layer, and a WireMock execution engine deployed in Docker container. The effectiveness of the proposed approach was evaluated through a comparative experiment across the four test scenarios with increasing complexity. The DSL-based configurations were assessed against equivalent low-level configurations using three selected metrics: a specification size (SLOC), a maximum nested block depth, and a maintainability index. The composite quality score of DSL-based configurations, presented in formula (2), exceeded the baseline by 38 % on average, demonstrating measurable improvements in code quality and specification clarity, presented in Table 4. The proposed DSL consistently reduced the nesting depth and produced more maintainable specifications, particularly, in appropriate scenarios involving conditional logic, dynamic response templating, and multi-branch request matching. Taking these points into account, the several directions could be identified for future development of the proposed approach. First, a performance optimization for large-scale software projects with numerous mock services would reduce latency in DSL parsing and configuration generation, making the platform suitable for enterprise-grade test environments. Second, an extending protocol support beyond HTTP/HTTPS to include WebSocket and gRPC would broaden applicability to real-time systems and IoT architectures. Third, a security hardening with an encryption of stored DSL scripts in DBMS MongoDB for stronger authentication between the middleware, and with usage of WireMock tool for production deployments where especially a data confidentiality is critically important. Additionally, a hybrid text-and-visual editing model could be explored to support both technically experienced users and non-developer stakeholders such as QA analysts and domain experts. Although the usage of graphical interfaces causes some additional implementation complexity, they may significantly lower the entry barrier and increase adoption in heterogeneous project teams. Besides that, an empirical validation to be done previously in CI/CD pipelines and large-scale multi-service projects, would also help to identify some performance bottlenecks, and to refine the already implemented DSL constructs based on suitable real-world usage patterns.

Summarized, the proposed solution represents a meaningful contribution to the field of SOA and microservice - based applications testing. By reducing of manual configuration effort, lowering the risk of specification errors, and improving team-wide accessibility of mock-based testing, it addresses a gap that remains insufficiently resolved in both research and industrial practice. The elaborated approach can also serve as a foundation for further research in some related areas, e.g., for an automated test generation from available DSL specifications, and for an AI-assisted mock scenarios building based on recorded system traffic.

Declaration on the use of generative AI. The authors used generative AI tools for English language checking and paraphrasing. All content was reviewed and approved by the authors.

References

1. MockServer. *Official Website.* Available at: https://www.mock-server.com/ (accessed: 15.02.2026).
2. Beck K. *Kent Beck's Guide to Better Smalltalk.* Cambridge, Cambridge University Press, 1998. 408 p. ISBN 978-0-521-64437-2.
3. Fowler M. *Domain-Specific Languages.* Boston, Addison-Wesley Professional, 2010. 640 p. ISBN 978-0-321-71294-3.
4. Mernik M., Heerin J., Sloane A. *When and How to Develop Domain-Specific Languages.* Available at: https://dl.acm.org/doi/10.1145/1118890.1118892 (accessed: 15.02.2026). DOI: https://doi.org/10.1145/1118890.1118892.
5. Ed-douibi H., Izquierdo J. L. C., Cabot J. Automatic Generation of Test Cases for REST APIs: A Specification-Based Approach. *IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC).* 2018. Available at: https://www.researchgate.net/publication/328991978_Automatic_Generation_of_Test_Cases_for_REST_APIs_A_Specification-Based_Approach (accessed: 15.02.2026). DOI: 10.1109/EDOC.2018.00031.
6. Valle P., Arrieta A., Han L., Ali S., Yue T. Defining and generating multi-level and uncertainty-wise test oracles for cyber-physical systems. *Softw Syst Model.* 2025, vol. 24, pp. 679–704. Available at: https://link.springer.com/article/10.1007/s10270-025-01271-8 (accessed: 15.02.2026). DOI: 10.1007/s10270-025-01271-8.
7. De Almeida R., Da Silva R. M., Serrano L. S., Campos Junior H. S., Neves V. O. Mock Objects in Software Testing: An Analysis of Usage in Open-Source Projects. *SBQS '23: Proceedings of the XXII Brazilian Symposium on Software Quality*. 2023, pp. 72–79 Available at: https://doi.org/10.1145/3629479.3629510 (accessed: 15.02.2026).
8. Felderer, M., Enoiu, E. P., Tahvili, S. (2023). Artificial Intelligence Techniques in System Testing. *Romero J. R., Medina-Bulo I., Chicano F. (eds). Optimising the Software Development Process with Artificial Intelligence. Natural Computing Series.* Springer, Singapore. Available at: https://doi.org/10.1007/978-981-19-9948-2_8 (accessed: 15.02.2026).

Table 4 – Result's Tests

	SLOC		Max Nested Block Depth		Maintainability index	
	DSL	Configuration	DSL	Configuration	DSL	Configuration
Test case 1	23	24	3	2	64.55	59.22
Test case 2	74	55	3	3	52.49	51.82
Test case 3	69	71	3	3	52.82	49.03
Test case 4	85	101	3	3	50.41	45.67

9. *From External Chaos to Business Value: The Power of AI Mocking.* Available at: https://speedscale.com/blog/from-external-chaos-to-business-value-the-power-of-ai-mocking/ (accessed: 15.02.2026).
10. Gomez N., Batham S., Volonte M., Tiffany D. Do. Virtual Interviewers, Real Results: Exploring AI-Driven Mock Technical Interviews on Student Readiness and Confidence. *CSCW Companion '25: Companion Publication of the 2025 Conference on Computer-Supported Cooperative Work and Social Computing*. 2025, pp. 209–213. Available at: https://dl.acm.org/doi/10.1145/3715070.3749227 (accessed: 15.02.2026). DOI: https://doi.org/10.1145/3715070.374922.
11. Bhatt K., Tarey V., Patel P. Analysis Of Source Lines Of Code (SLOC) Metric. *International Journal of Emerging Technology and Advanced Engineering.* 2012, vol. 2, iss. 5, pp. 150–154. Available at: https://www.researchgate.net/publication/281840565_Analysis_Of_Source_Lines_Of_CodeSLOC_Metric (accessed: 15.02.2026).
12. Harrison W. A., Magel K. I. A complexity measure based on nesting level. *ACM SIGPLAN Notices.* 1981, vol. 16, iss. 3, pp. 63–74. Available at: https://dl.acm.org/doi/10.1145/947825.947829 (accessed: 15.02.2026). DOI: https://doi.org/10.1145/947-825.947829
13. Oman P., Hagemeister J. Metrics for assessing a software system's maintainability. *Proceedings of the International Conference on Software Maintenance (ICSM)*, 1992, pp. 337–344. Available at: https://www.researchgate.net/publication/2954310_Using_Metrics_to_Evaluate_Software_System_Maintainability (accessed: 15.02.2026). DOI: 10.1109/ICSM.1992.242525.
14. *What is ANTLR (ANother Tool for Language Recognition)?\, Terensi Parr.* Available: https://www.antlr.org/ (accessed: 15.02.2026)

Received 06.03.2026
Accepted 30.03.2026
Published 20.05.2026

УДК 004.43; 004.051; 811.93

Р. О. ГАМЗАЄВ, кандидат технічних наук (PhD), доцент, доцент кафедри інтелектуальних програмних систем і технологій, Харківський національний університет імені В. Н. Каразіна, майдан Свободи, 4, м. Харків, Україна, 61022; e-mail: rustam.gamzayev@ karazin.ua; ORCID: https://orcid.org/0000-0002-2713-5664
М. В. ТКАЧУК, доктор технічних наук, професор, професор кафедри інтелектуальних програмних систем і технологій, Харківський національний університет імені В. Н. Каразіна, майдан Свободи, 4, м. Харків, Україна, 61022; e-mail: mykola.tkachuk@karazin.ua; ORCID: https://orcid.org/0000-0003-0852-1081
Г. С. ЛЕГЕНЬКИЙ, студент, Харківський національний університет імені В. Н. Каразіна, майдан Свободи, 4, м. Харків, Україна, 61022; e-mail: lehenkyi2021ki11@student.karazin.ua;ORCID: https://orcid.org/0009-0006-4444-5759

ПРОБЛЕМНО-ОРІЄНТОВАНА МОВА ДЛЯ ІНТЕЛЕКТУАЛЬНОГО ТЕСТУВАННЯ МІКРОСЕРВІСНИХ ПРОГРАМНИХ СИСТЕМ НА ОСНОВІ ТЕХНОЛОГІЇ MOCK-ОБ'ЄКТІВ

Тестування сервісно-орієнтованих та мікросервісних систем є складним завданням через їх сильну залежність від зовнішніх сервісів та розподілених середовищ. Тестування на основі макетів об'єктів (mock-об'єктів) широко використовується для вирішення цієї проблеми; однак існуючі рішення використовують низькорівневі програмні механізми конфігурування таких систем, що збільшує складність процесів їх супроводу. У цій статті пропонується предметно-орієнтована мова (domain-specific language - DSL) для інтелектуальної підтримки процесів специфікації та генерації mock-об'єктів. Цей підхід дозволяє забезпечує можливість високорівневого опису поведінки таких об'єктів та підтримує автоматизацію побудови їх конфігурацій існуючих технологічних платформ. Запропоноване рішення спрямоване на покращення читабельності, зменшення зусиль на конфігурацію та підвищення ефективності тестування в розподілених програмних системах. У статті розглядаються існуючі інструменти для тестування сервісно-орієнтованих програм на основі mock-об'єктів та проводиться їх порівняльний аналіз, заснований на критеріях простоти використання, гнучкості конфігурації та складності підтримуваних тестових сценаріїв. Аналіз показує, що існуючі рішення мають певні недоліки та обмеження щодо їх реалізації, усунення яких уможливлює застосування мови DSL. Для оцінки запропонованого підходу було проведено порівняльний експеримент за чотирма тестовими сценаріями різної складності. Конфігурації на основі DSL порівнювалися з еквівалентними конфігураціями, визначеними без використання DSL, за допомогою трьох кількісних метрик: розмір специфікації (SLOC), максимальна глибина вкладених блоків та індекс супроводжуваності. Сукупний показник якості тестових конфігурацій, побудованих на основі використання DSL, перевищує їх показних для базового сценарію у середньому на 38 %. Ці висновки підтверджують, що запропонована мова DSL спрощує створення макетних сервісів і робить тестування розподілених систем доступнішим для ширшого кола учасників проекту.

Ключові слова: предметно-орієнтована мова, інтелектуальний підхід, програмне забезпечення, сервісно-орієнтована архітектура, мікросервіс, тестування, mock-об'єкт, розподілена система, метрика, якість.

Повні імена авторів / Author's full names

Автор 1 / Author 1: Гамзаєв Рустам Олександрович / Gamzayev Rustam Oleksandrovych
Автор 2 / Author 2: Ткачук Микола Вячеславович / Tkachuk Mykola Vaycheslavovych
Автор 3 / Author 3: Легенький Гліб Сергійович / Legenkyi Glib Sergiiovyc

ЗМІСТ