https://doi.org/10.31891/2307-5732-2026-363-3
УДК 004.428:004.415.53

BELITSKYI OLEKSANDR
National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute"
https://orcid.org/0009-0009-8884-2862
e-mail: belitskyi.oleksandr@lll.kpi.ua
YUSYN YAKIV
National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute"
https://orcid.org/0000-0001-6971-3808
e-mail: yusyn@pzks.fpm.kpi.ua

METHOD FOR ADAPTING THE METAMORPHIC TESTING-AS-A-SERVICE ARCHITECTURAL PATTERN FOR THE JAVA PROGRAMMING LANGUAGE

This paper is devoted to adapting the Metamorphic Testing-as-a-Service (MTaaS) architectural pattern for the Java programming language, based on its language-specific characteristics. Metamorphic testing is a promising approach to verifying software systems in cases where constructing a classical test oracle is difficult or impossible. However, the practical adoption of metamorphic testing in software development processes is often complicated due to the need to implement metamorphic scenarios manually, the requirement for a significant amount of auxiliary code, and weak integration with existing software architectures.

The paper analyzes existing approaches to the implementation of metamorphic testing and the MTaaS architectural pattern and substantiates the choice of an annotation-based approach as the most suitable for the Java ecosystem. The proposed method is based on the use of annotations and annotation processing mechanisms for the declarative specification of metamorphic relations, automatic collection of their components, and generation of a formal specification for subsequent execution of metamorphic checks.

A prototype implementation of MTaaS for Java was developed and experimentally compared with a manual implementation of metamorphic testing to evaluate the effectiveness of the proposed approach. The experiments were conducted using two problems of different complexity: Text Normalization Pipeline and Convex Hull. The comparison was performed using object-oriented code quality metrics, including LOC, WMC, CBO, RFC, LCOM, and NOM.

The experimental results showed that for problems with a small number of metamorphic relations, the annotation-based MTaaS approach provides moderate improvements in code quality characteristics. At the same time, for complex problems involving a large number of independent metamorphic scenarios, the use of MTaaS makes it possible to significantly reduce code size, decrease complexity and coupling, and eliminate a substantial portion of auxiliary glue code.

Keywords: *metamorphic testing; architectural pattern; MTaaS; Java annotations; software quality.*

БЕЛІЦЬКИЙ ОЛЕКСНДР, ЮСИН ЯКІВ
Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського»

МЕТОД АДАПТАЦІЇ АРХІТЕКТУРНОГО ШАБЛОНУ METAMORPHIC TESTING-AS-A-SERVICE ДЛЯ МОВИ ПРОГРАМУВАННЯ JAVA

Ця робота присвячена задачі адаптації архітектурного шаблону Metamorphic Testing-as-a-Service (MTaaS) для мови програмування Java з урахуванням її мовних особливостей. Метаморфічне тестування є перспективним підходом до верифікації програмних систем у випадках, коли побудова класичного тестового оракула є складною або неможливою. Проте практичне впровадження метаморфічного тестування у процеси розроблення програмного забезпечення часто є ускладненим через необхідність реалізації метаморфічних сценаріїв власноруч, необхідність значного обсягу допоміжного коду та слабку інтеграцію з існуючими архітектурами програмних систем.

У роботі проаналізовано існуючі підходи до реалізації метаморфічного тестування та архітектурного шаблону MTaaS, а також обґрунтовано вибір анотаційного підходу як найбільш придатного для екосистеми Java. Запропонований метод базується на використанні анотацій та механізмів їх обробки для декларативного опису метаморфічних відношень, автоматичного збирання їх складових та формування формальної специфікації для подальшого виконання метаморфічних перевірок.

Для оцінки ефективності запропонованого підходу було розроблено прототип реалізації MTaaS для Java та проведено експериментальне порівняння з ручною реалізацією метаморфічного тестування. Експерименти виконано на прикладі двох задач різної складності: Text Normalization Pipeline та Convex Hull. Порівняння здійснювалося з використанням об'єктно-орієнтованих метрик якості коду, зокрема LOC, WMC, CBO, RFC, LCOM та NOM.

Результати експериментального дослідження показали, що для задач із невеликою кількістю метаморфічних відношень анотаційний підхід на основі MTaaS забезпечує помірне покращення характеристик коду. Водночас для складних задач із великою кількістю незалежних метаморфічних сценаріїв використання MTaaS дозволяє суттєво зменшити обсяг коду, знизити його складність і зв'язність, а також усунути значну частину допоміжного glue-коду.

Ключові слова: *метаморфічне тестування; архітектурний шаблон; MTaaS; анотації Java; якість програмного забезпечення.*

Стаття надійшла до редакції / Received 13.01.2026
Прийнята до друку / Accepted 20.02.2026
Опубліковано / Published 26.03.2026

Introduction

In the modern software development landscape, the role of effective testing methods is steadily increasing, especially for complex information systems in which exhaustive testing that covers all possible system behaviors is either infeasible or economically impractical. Such systems are typically characterized by high computational complexity, a large volume of input and output data, and the inability to determine system behavior in advance, which, in turn, affects the complexity of test construction and the reliability of test scenarios.

Traditionally, software testing relies on a test oracle – a procedure that distinguishes between the correct and incorrect behaviors of the System Under Test [1]. However, for many classes of software systems, constructing such an oracle is difficult or practically impossible, which significantly reduces the effectiveness of oracle-based testing methods.

One approach to addressing this problem is metamorphic testing, which is based on the use of metamorphic relations. They are relations or properties that should be preserved between the outputs of multiple executions of the program with different inputs [2]. This approach makes it possible to detect faults without requiring knowledge of the exact expected result for each individual test case, which makes it promising for testing complex software systems.

Despite the existence of both theoretical and practical research in the field of metamorphic testing, its direct adoption in software development processes remains limited. One of the reasons for this is the difficulty of integrating metamorphic checks into existing software architectures and automated development and testing workflows. Industry research also highlights the growing emphasis on quality assurance and automated testing within IT strategies. According to the World Quality Report 2021-22, a survey of 1,750 CIOs and senior tech leaders shows that testing and QA are increasingly seen as key factors in achieving quality outcomes, and there is a notable focus on enhancing QA through automation and tools across software development life cycles [3].

In this context, particular attention should be paid to approaches for implementing metamorphic testing for widely used programming languages, in particular Java, which is one of the main platforms for the development of enterprise and scientific information systems. This determines the relevance of research aimed at developing and analyzing architectural and software solutions that enable the practical application of metamorphic testing in modern software systems.

Analysis of recent research

The formal foundations of metamorphic testing approach, as well as examples of its application to various classes of software systems, are presented in works [4–6]. Subsequent studies have demonstrated the effectiveness of metamorphic testing in detecting defects in complex software systems, particularly under conditions of uncertain behavior or high computational complexity. Works [7–9] show that metamorphic testing can complement traditional testing methods and provide a higher level of fault coverage without the need to construct a full-fledged test oracle. At the same time, in most of these studies, the primary focus is placed on the formulation of metamorphic relations and the evaluation of their effectiveness, while issues related to the integration of metamorphic testing into software development processes are addressed only superficially.

One of the key challenges of the practical application of metamorphic testing is the need for manual identification of metamorphic relations and their direct implementation in test code. This complicates test scalability, reduces the reusability of metamorphic checks, and increases maintenance effort as the software system evolves. To partially address these issues, a number of approaches to the automation of metamorphic testing have been proposed, including automatic test data generation, semi-automatic derivation of metamorphic relations, and integration of metamorphic checks into existing testing frameworks [10–12].

At the same time, most existing solutions focus on individual aspects of metamorphic testing and do not offer a comprehensive architectural model for its use within software systems. In particular, such approaches often tightly couple metamorphic testing to a specific testing framework or execution environment, which limits reusability.

To overcome these limitations, the Metamorphic Testing-as-a-Service (MTaaS) architectural pattern has been proposed, which treats metamorphic testing as a separate service decoupled from the core functional code of the software system [13]. Within this approach, the definition of metamorphic relations, the management of their execution, and the analysis of testing results are moved to a dedicated service layer. This makes it possible to reduce coupling between testing logic and business logic.

In papers devoted to MTaaS, authors mainly focus on the conceptual description of this architectural pattern; however, an implementation for the C# programming language also exists. At present, there is a lack of studies that analyze how the characteristics of specific programming languages and execution platforms influence the ways in which MTaaS can be implemented.

Thus, the analysis of existing research indicates the presence of a solid scientific foundation, but also highlights the practical immaturity of the Metamorphic Testing-as-a-Service architectural concept. Therefore, this work aims to take a first step toward the development and analysis of approaches to implementing the MTaaS architectural pattern while considering the specific characteristics of the Java programming language.

Formulation of the goals of the article

The aim of this paper is to develop and analyze an approach to implementing the Metamorphic Testing-as-a-Service architectural pattern for the Java programming language, based on language-specific mechanisms. To achieve this aim, the following tasks were formulated and addressed in this study:

- Analysis of existing approaches to metamorphic testing and the Metamorphic Testing-as-a-Service architectural pattern in order to identify limitations related to their implementation aspects.
- Development of an implementation model of the MTaaS pattern for the Java programming language based on the use of annotations and annotation processing mechanisms.
- Creation of a prototype software implementation that demonstrates the practical applicability of the proposed approach to implementing the MTaaS pattern in Java applications.

- Experimental evaluation of the proposed approach by comparing a manual implementation of metamorphic testing with an implementation based on the MTaaS pattern.

Presentation of the main material

Justification of the choice of implementation

In the original work [13], the MTaaS architectural pattern is implemented using two main approaches: contracts and attributes. The contract-based implementation is built around the idea that the user specifies certain relationships in a separate YAML file, while MTaaS generates files that the developer must implement manually. The alternative approach is attribute-based, in which the developer explicitly implements the required classes and marks them with appropriate attributes to enable metamorphic testing.

In Java, annotations are an almost direct analogue of attributes in C# [14]. A significant advantage of this form of MTaaS implementation in Java is that annotations represent a long-established mechanism for declaratively attaching metadata to classes, fields, methods, and other program elements. Another important factor is that the Java community has long accepted annotations as a mechanism that performs certain infrastructure tasks, for example, widely used frameworks and tools such as Spring, JPA, Lombok, MapStruct, etc.

It is also worth noting that annotations allow metadata to be added without modifying the business logic itself. As a result, this approach does not force developers to rewrite their code in a contract-oriented style and is more suitable for gradual adoption.

The main difficulty of implementing the contract-based approach in Java lies in the absence of built-in language mechanisms to support it. In contrast to Java, in C# the contract-based approach can be realized at the language level through the use of partial classes and methods, which makes it possible to separate functional implementation from contract logic without modifying the core code. This approach is a language feature of C# and has no direct counterpart in Java.

Java lacks mechanisms for partial classes and methods, as well as standardized means for declaratively specifying contracts at the level of the language or the execution platform. As a result, implementing a contract-based approach in Java is only possible through the use of external tools, which leads to dependence on specific libraries, complicates the build and execution process, and fails to provide a single established standard. This makes the contract-based approach less suitable for implementing MTaaS in Java.

General idea of MTaaS operation for Java

In the proposed approach to implementing MTaaS, the core mechanism is the compile-time analysis of annotations used for the declarative specification of metamorphic relations in the code. Specifically, during the compilation stage, a specialized annotation processor performs the detection of all components marked with MTaaS annotations.

All identified elements are grouped according to the corresponding metamorphic relation names specified in the annotations. This makes it possible to combine individual parts of a relation (the artifact under test, input and output data transformations, the data generator, and the comparator) into a single coherent model. Based on this model, a representation of the metamorphic relation is formed that separates its structural components from the verification logic.

Implementation

There are 5 annotations in total: ArtifactEntry, DataGenerator, InputMetamorphosis, OutputMetamorphosis, and OutputModelComparer.

- ArtifactEntry – Entry point to the metamorphic relation (relation entrypoint). This is the main class or method that describes the functionality under test.
- DataGenerator – Input data generator for the metamorphic relation. It forms the initial model (input model), which is then transformed by metamorphoses.
- InputMetamorphosis – Metamorphic transformation of input data. Determines how the input changes to obtain a new test scenario. It takes the initial input and transforms it into a modified one.
- OutputMetamorphosis – It is a transformation of the results after executing the artifact. That is, how the output model (output) is expected to change if the input changes. Specifies the expected behavior of the result when the input data changes.
- OutputModelComparer – Compares two results (original and transformed). This is a class for checking an invariant – whether the expected relationship is met. It specifies the criteria for equality. The class marked with this annotation must implement the Comparator functional interface.

The structural UML diagram of the annotation processing component of the MTaaS architectural pattern for Java is shown in Figure 1.

In general, the structure is kept the same as in the original work. However, in order to improve code maintainability, the logic of several modules was decomposed to ensure compliance with the Single Responsibility Principle.

Participants.

- MtaasProcessor is the main input class. Implements javax.annotation.processing.Processor. This is the class that Java automatically starts when compiling. It gets RoundEnvironment from the compiler, finds all elements with the required annotations, passes them to RelationCollector, and after processing, generates YAML via YamlEmitter.
- AnnotationValueReader – it is a utility for reading parameters from annotations. It searches for relation, value or relationName keys and is also used in RelationCollector to correctly group classes by relation name.
- RelationCollector collects all parts belonging to one metamorphic relation (relationName). It classifies elements by type (artifact, data generator, metamorphosis, comparer) and creates a RelationParts object for each relationName. Also checks if all roles are filled (otherwise it outputs an error like [MTaaS][order-total] Can't find all required parts for relation).

- RelationParts – a model structure for storing “parts” of one relation.
- RelationSemantics is a logical interpretation of a relation. If RelationParts is “what it is”, then RelationSemantics is “how it behaves”. It checks the correctness of the combinations (whether all roles are present, whether methods are present, etc.). It also creates a description for YAML (relation name, method names) and prepares information for SourceGenerator.
- Role – an Enum of roles in a relation.
- YamlEmitter – generates a file in which all relationships are displayed.
- SourceGenerator – is responsible for generating Java classes.

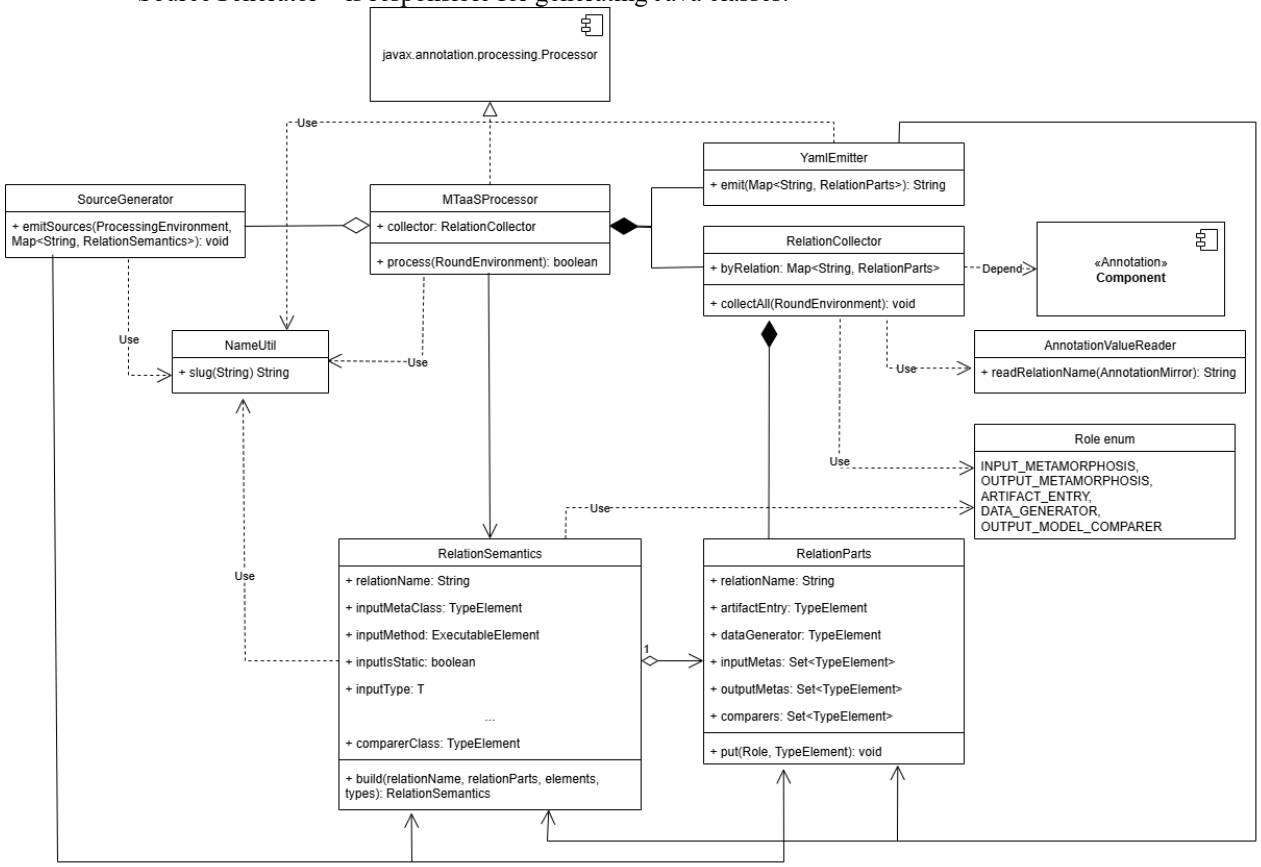

Figure 1. Structural diagram of the processing module of the MTaaS pattern

Experiment

To evaluate the annotation-based approach, a comparison was conducted using code quality metrics against a classical implementation of metamorphic testing, as if the developer manually implemented metamorphic tests. In the manual implementation, metamorphic tests are realized using the JUnit library; that is, each test contains code for data generation, function invocation, result processing, and result comparison.

Two benchmark tasks were selected for conducting the experiment: TextNormalizationPipeline and ConvexHull.

TextNormalizationPipeline is a software pipeline that performs sequential transformations of textual data in order to bring it into a normalized form. Typically, such a pipeline includes several stages of string transformation, such as case normalization, removal or unification of punctuation marks, whitespace normalization, and the application of consecutive transformations to the string. The final result depends on a combination of multiple transformations, which is an important characteristic of this task.

Task overview:

- Input: a list of strings.
- Artifact: a normalization algorithm that returns a list of normalized strings and statistics over vocabulary.
- Core functionality to test:
  - tokenization;
  - lowercase;
  - filtering stop-words.
- Input metamorphoses:
  - add whitespace;
  - change case;
  - duplicate sentences.
- Output MR:
  - vocabulary size invariant;

- token frequency scales;
- ordering invariant.

For the comparative analysis, the results of which are presented in Table 1, the CK library version 0.7.0 was used [15]. In Table 1, the columns "Mean," "Minimum," and "Maximum" present the corresponding average, minimum, and maximum values of the metrics for the classes that played a role in metamorphic testing.

Table 1

Code quality metrics of the two developed software for the Text Normalization task

Metric	Approach	Mean	Minimum	Maximum
Lines of Code (LOC)	Manual	18.25	5	42
	Annotation-based	14.14	5	30
Weighted Methods per Class (WMC)	Manual	4.93	1	9
	Annotation-based	4.75	1	9
Coupling Between Objects (CBO)	Manual	2.94	0	11
	Annotation-based	1.93	0	4
Response for Class (RFC)	Manual	5.19	0	13
	Annotation-based	5.14	0	11
Lack of Cohesion of Methods (LCOM)	Manual	2.12	0	20
	Annotation-based	1.79	0	20
Number of Methods (NOM)	Manual	2.62	1	8
	Annotation-based	2.07	1	8

First of all, the Lines of Code (LOC) metric shows a slight reduction in the average number of lines of code when using the annotation-based approach (14.14 versus 18.25). Although the difference is not significant, it indicates an overall reduction in code volume. This, in turn, has a positive effect on the readability and maintainability of the software.

The values of Weighted Methods per Class (WMC) are close for both approaches (4.93 for the manual approach and 4.75 for the annotation-based one). This indicates that the algorithmic complexity of individual classes in the text normalization task remains similar regardless of the testing implementation approach. Such a result is quite expected given the nature of the task, which does not require extensive branching or complex execution scenarios.

The Coupling Between Objects (CBO) metric decreases when using the annotation-based approach (1.93 versus 2.94), which indicates weaker coupling between classes and better modularity of the implementation. Similarly, the values of Response for Class (RFC) are almost identical (5.14 versus 5.19), indicating that the annotation-based approach does not negatively affect the complexity of class interactions with external components.

The Lack of Cohesion of Methods (LCOM) metric has a lower average value in the annotation-based implementation (1.79 versus 2.12), which indicates that classes have more clearly defined responsibilities. A similar trend is observed for the Number of Methods (NOM) metric, whose average value decreases from 2.62 in the manual approach to 2.07 in the annotation-based one, indicating a reduction in the amount of boilerplate code.

The Text Normalization task was selected as an example in which metamorphic relations are relatively simple, such as text duplication, case transformation, and whitespace normalization. In this task, a significant portion of the logic can be implemented as glue code, and some classes can be reused across multiple relations.

In the general case, the greater the number of independent metamorphic relations and execution scenarios, the more advantageous the MTaaS approach becomes. To demonstrate this effect, the construction of the convex hull of a set of points was chosen as the second benchmark task. Several reasons motivated the selection of this particular problem. Firstly, the convex hull construction function preserves its result under a number of transformations, which directly aligns with the fundamental principles of metamorphic testing, because these transformations can be mapped into well-defined, mathematically grounded metamorphic relations:

- Permute MR – permuting the input points does not change the result.
- Translate MR – translating the entire set by a vector does not change the shape of the hull.
- Rotate MR – rotating the coordinates by an arbitrary angle produces a result equivalent to the rotated hull.
- Scale MR – scaling the set changes the hull proportionally.

Secondly, a convex hull may consist of dozens of points, and in the general case, it is difficult to manually construct the correct result. Oracle-based testing would require an expected set of points, a specific traversal order, and repeated computation using the same algorithms. Metamorphic testing avoids these difficulties, as it verifies structural properties rather than specific coordinate values.

Task overview:

- Input: a list of two-dimensional points (x, y).
- Artifact: a convex hull construction algorithm that returns an ordered list of points forming the outer boundary of the set.
- Core functionality to test:
  - geometric processing of coordinates;
  - construction of the minimal convex polygon;
  - ignoring internal points.
- Input metamorphoses:

- permutation of point order;
- scaling of coordinates;
- translation of the point set;
- rotation around the origin;
- addition of internal points.

- Output MR:
  - invariance of the convex hull shape;
  - invariance of the number of vertices;
  - preservation of the relative order of vertices (up to cyclic shifts);
  - invariance of the hull area (except in the case of scaling).

The results of the comparative analysis for this task are shown in Table 2. Its "Mean," "Minimum," and "Maximum" columns meaning is the same as in Table 1.

Table 2

Code quality metrics of the two developed software for the Convex Hull task

Metric	Approach	Mean	Minimum	Maximum
Lines of Code (LOC)	Manual	32.5	10	66
	Annotation-based	10.9	3	22
Weighted Methods per Class (WMC)	Manual	8.75	1	20
	Annotation-based	3.00	1	8
Coupling Between Objects (CBO)	Manual	2.62	1	5
	Annotation-based	2.38	1	4
Response for Class (RFC)	Manual	7.88	3	18
	Annotation-based	2.67	1	6
Lack of Cohesion of Methods (LCOM)	Manual	4.62	0	11
	Annotation-based	1.05	0	3
Number of Methods (NOM)	Manual	4.38	2	8
	Annotation-based	1.71	1	3

In general, it can be observed that the number of lines of code (LOC) differs almost threefold between the manual and annotation approaches, which in turn improves the maintainability of the code, especially considering the declarative way of programming when using annotations.

It is worth noting that in the manual implementation, the WMC metric reaches 8.75 compared to 3 in the annotation-based approach. This indicates that each class of the manual metamorphic test contained a significantly larger number of independent execution paths. A similar situation is observed for the RFC metric, meaning that in the manual approach, the test class potentially invokes more external methods than in the annotation-based approach. This, in turn, indicates that the class is more dependent on other components.

The LCOM metric reflects the degree of cohesion among methods within a class: the higher the value, the weaker the relationship between methods. It should be noted that the value obtained when using the annotation-based approach is significantly lower (1.05 compared to 4.62). This indicates that the annotation-based approach leads to the formation of classes with a narrower and more focused responsibility.

The NOM metric demonstrates the number of methods contained in a class. The difference in the values of this metric indicates that the developer needs to create fewer methods when using the annotation-based approach compared to the manual one, which undoubtedly affects development speed and reduces the amount of boilerplate code.

A relatively small difference is observed only for the CBO metric. Thus, it can be concluded that the developed manual testing architecture is close to the annotation-based architecture in terms of modularity, which indicates a correct architectural design approach and is also consistent with the fact that the Single Responsibility Principle was observed when designing both testing options.

Such a significant difference in metric values can be explained by several key reasons. The first reason is that the Convex Hull problem represents almost a worst-case scenario for manual implementation of metamorphic tests, since this task contains a large number of independent metamorphic relations. For each such relation, the entire data structure is transformed, and each new scenario introduces additional glue code.

The second reason is that MTaaS allows the developer to avoid writing glue code that is required in the manual implementation. All glue code is replaced by the declarative capabilities provided by the MTaaS architectural pattern.

The third reason is the absence of an established template for writing metamorphic tests. Each developer implements metamorphic tests as it turns out, that is, any implementation will be typically practical rather than theoretically minimal.

Based on the obtained results, it can be concluded that the applicability of the architectural pattern depends on the nature of the task. If a task involves simple scenarios with a small number of metamorphic relations, MTaaS provides a moderate improvement in code characteristics. However, when MTaaS is applied to complex tasks that involve a large set of metamorphic transformations, then using MTaaS will significantly reduce the code size, reduce complexity and coherence. The observed improvements confirm the feasibility of using the MTaaS architectural pattern in real-world engineering scenarios.

Conclusions

In this study, the problem of adapting the Metamorphic Testing-as-a-Service architectural pattern for the Java programming language was considered, taking into account its language-specific features. The proposed approach is based on the use of annotations and annotation processing, which increases the declarative nature of applying metamorphic testing.

In the course of the study, a prototype of an annotation-based implementation of the MTaaS architectural pattern for Java was developed, and an experimental comparison with a manual implementation of metamorphic testing was conducted. Two tasks of different levels of complexity were used for the experimental evaluation: Text Normalization Pipeline and Convex Hull. Object-oriented metrics (LOC, WMC, CBO, RFC, LCOM, NOM) were used to evaluate the results, which made it possible to quantitatively compare the implementation approaches.

Based on the obtained results, it can be concluded that for tasks with a small number of metamorphic relations, the implementation of metamorphic testing using MTaaS provides a moderate improvement in code characteristics without significantly affecting its structure. However, when MTaaS is applied to tasks with a large number of independent metamorphic scenarios, it leads to a substantial reduction in code size, a decrease in complexity and coupling, and the elimination of glue code.

Considering the results of the experimental study, it can be stated that the Metamorphic Testing-as-a-Service architectural pattern is appropriate for use in real engineering projects. Its application is particularly relevant for complex software systems in which a manual implementation of metamorphic testing becomes difficult to maintain and scale. The proposed approach provides a foundation for further research aimed at extending the functionality of MTaaS and integrating it with modern automated testing tools and cloud-based execution environments.

References

1. Barr E. T., Harman M., McMinn P., Shahbaz M., Yoo S. The Oracle Problem in Software Testing: A Survey, IEEE Transactions on Software Engineering, vol. 41, no. 5, pp. 507–525, (2015). https://doi.org/10.1109/TSE.2014.2372785.
2. Sharma D. Metamorphic Testing: A New Horizon in Software Testing, Medium, (2019). Available at: https://medium.com/@mailtodevens/metamorphic-testing-a-new-horizon-in-software-testing-6fdec595dba8.
3. Capgemini, Sogeti, and Micro Focus. World Quality Report 2021-22. 13th edition (2021). Available at: https://www.capgemini.com/us-en/insights/research-library/world-quality-report-wqr-2021-22/.
4. Chen T. Y., Cheung S. C., Yiu S. M. Metamorphic testing: a new approach for generating next test cases. Technical Report HKUST-CS98-01, Hong Kong University of Science and Technology, 1998.
5. Chen T. Y., Tse T. H., Zhou Z. Q., "Fault-based testing without the need of oracles," Information and Software Technology, vol. 45, no. 1, pp. 1–9, (2003). https://doi.org/10.1016/S0950-5849(02)00129-5.
6. Segura S., Fraser G., Sanchez A. B., Ruiz-Cortés A. A survey on metamorphic testing, IEEE Transactions on Software Engineering, vol. 42, no. 9, pp. 805–824, (2016). https://doi.org/10.1109/TSE.2016.2532875.
7. Zhou Z. Q., Sun L. Metamorphic testing of driverless cars, Communications of the ACM, vol. 62, no. 3, pp. 61–67, (2019). https://doi.org/10.1145/3241979.
8. Murphy C., Kaiser G. E., Hu L., Wu L. Properties of machine learning applications for use in metamorphic testing, in Proceedings of the 20th International Conference on Software Engineering & Knowledge Engineering (SEKE), (2008).Available at: https://www.researchgate.net/publication/221271538_Properties_of_Machine_Learning_Applications_for_Use_in_Metamorphic_Testing.
9. Kanewala U., Bieman J. Using machine learning techniques to detect metamorphic relations for programs without test oracles, in Proceedings of the 24th IEEE International Symposium on Software Reliability Engineering (ISSRE), (2013). https://doi.org/10.1109/ISSRE.2013.6698899.
10. Chen T. Y., Kuo F.-C., Liu H., Poon P.-L., Towey D. Metamorphic testing: A review of challenges and opportunities, ACM Computing Surveys, vol. 51, no. 1, Article 4, (2018). https://doi.org/10.1145/3143561.
11. Gotlieb A., Botella B. Automated Metamorphic Testing, in Proceedings of the 27th Annual International Computer Software and Applications Conference (COMPSAC 2003), pp. 410–415, (2003). https://doi.org/10.1109/CMPSAC.2003.1245319.
12. Ayerdi J., Valle P., Segura S., Arrieta A., Sagardui G., Arratibel M. "Performance-Driven Metamorphic Testing of Cyber-Physical Systems," IEEE Transactions on Reliability, vol. 72, no. 2, pp. 827–845, (2023). https://doi.org/10.1109/TR.2022.3193070.
13. Yusyn Y., Zabolotnia T. "Metamorphic Testing-as-a-Service: a new design pattern of cloud serverless systems for metamorphic testing", Herald of Khmelnytskyi National University, vol. 305, no. 1, pp. 107–115, (2022). https://doi.org/10.31891/2307-5732-2022-305-1-107-115.
14. Gosling J., Joy B., Steele G., Bracha G., Bukcley A., Smith D., Bierman G. "9.7 Annotations" in The Java Language Specification. Java SE 25 Edition (2025), pp. 391–401. Available at: https://docs.oracle.com/javase/specs/jls/se25/jls25.pdf.
15. Aniche M. CK: A Tool for Calculating Chidamber and Kemerer Object-Oriented Metrics. [Online]. Available: https://github.com/mauricioaniche/ck.