# Principles of autonomous testing of high-performance .NET application

*Rimma* Zaripova[1], *Maxim* Kuznetsov[2,3], *Valery* Kosulin[1], *Marat* Perukhin[2], and *Marat* Nuriev[4*]

[1]Kazan State Power Engineering University, Kazan, Russia
[2]Kazan National Research Technological University, Kazan, Russia
[3]Kazan State Agrarian University, Kazan, Russia
[4]Kazan National Research Technical University named after A. N. Tupolev – KAI, Kazan, Russia

**Abstract.** In the landscape of software development for high-performance .NET applications, autonomous testing emerges as a critical strategy to ensure reliability, scalability, and performance. This article delves into the practice of autonomous, or unattended, testing—where automated test cases are executed independently without human intervention. Our exploration is grounded in the application of autonomous testing in environments handling large data volumes and supporting high concurrency, which are typical scenarios for mission-critical .NET applications. We discuss the benefits of autonomous testing, including its ability to significantly increase test coverage, enhance defect detection at early stages, and ensure consistent and reliable testing outcomes across various scenarios. The implementation of robust testing frameworks such as NUnit, xUnit, or MSTest, which support features like parallel test execution and test parameterization, plays a foundational role in the effective deployment of autonomous testing systems. Moreover, the article highlights the necessity of integrating autonomous testing into continuous integration and deployment pipelines to facilitate continuous testing. This integration ensures that every code change is thoroughly validated before deployment, thereby enhancing software quality and accelerating delivery cycles. We also examine the challenges and best practices in fostering a culture that supports autonomous testing within organizations. By emphasizing the strategic importance of training, cross-functional collaboration, and continuous improvement, we propose methods to overcome resistance to change and enhance the adoption of autonomous testing practices.

## 1 Introduction

Autonomous testing, also referred to as unattended or automated testing, is an indispensable practice for ensuring the reliability, scalability, and performance of mission-critical .NET applications. It involves the creation and execution of automated test cases that can run independently, without human intervention, to validate the application's behavior under various conditions and workloads [1,2]. Autonomous testing is particularly crucial for high-

* Corresponding author: marat_nu1@mail.ru

© The Authors, published by EDP Sciences. This is an open access article distributed under the terms of the Creative Commons Attribution License 4.0 (https://creativecommons.org/licenses/by/4.0/).

performance .NET applications, which often handle substantial amounts of data, support numerous concurrent users, and require robust performance under heavy loads [3].

Autonomous testing offers several benefits for high-performance .NET applications. It enables increased test coverage and thoroughness, allowing for exhaustive testing of the application's functionality, including complex scenarios, high concurrency, and extreme load conditions. This approach facilitates early defect detection and prevention [4,5], identifying and addressing issues promptly before they propagate to later stages or reach production environments, significantly reducing the cost and effort required to fix issues. Additionally, autonomous testing ensures improved consistency and repeatability, executing the same steps consistently and providing reliable results [6], regardless of the test environment or the person executing the tests. It accelerates feedback cycles and supports continuous integration, providing rapid feedback to developers and enabling them to address issues promptly, which is essential for high-performance applications where even minor changes can have significant performance implications. Furthermore, autonomous testing enables scalability and performance testing by simulating high concurrency and load conditions, thoroughly testing the application's ability to handle substantial workloads and large volumes of data [7,8].

Effective autonomous testing for high-performance .NET applications requires several key components. A robust testing framework, such as NUnit, xUnit, or MSTest, provides the infrastructure for writing, organizing, and running automated tests, offering features like parallel test execution, data-driven testing, and test parameterization. Test cases and test suites cover a wide range of scenarios, including high concurrency, large data volumes, extreme load conditions, and edge cases [9]. Test data management involves the creation, maintenance, and cleanup of large and complex test datasets, employing techniques like data generation, data masking, and test data virtualization. The test execution environment should mimic the production environment closely, including hardware specifications, network configurations, and dependencies like databases and external services [10,11]. Comprehensive reporting and monitoring mechanisms track test results, identify failures, and analyze performance metrics like response times, throughput, and resource utilization.

Integrating autonomous testing into a continuous integration and deployment pipeline enables continuous testing, ensuring that every code change is thoroughly validated before being deployed to production. Specialized load testing tools and frameworks, such as Apache JMeter, Visual Studio Load Test, and NUnit Load Test Runner, simulate real-world user loads and measure the application's performance under various load conditions [12]. Parallelization and distributed test execution techniques reduce overall testing time, enabling efficient execution of large test suites by running tests in parallel across multiple machines or utilizing cloud-based test execution services [13,14].

By embracing autonomous testing principles and incorporating them into the development lifecycle, organizations can enhance the quality, reliability, and performance of their high-performance .NET applications, while reducing the risk of critical defects and regressions [15]. Additionally, autonomous testing facilitates continuous delivery and deployment, enabling organizations to rapidly respond to changing market demands and deliver value to their customers more frequently.

## 2 Refactoring for testability

Testability is a crucial aspect of software development, especially for high-performance .NET applications that demand robust and reliable autonomous testing. However, over time, codebases can become entangled with tight coupling, global state dependencies, and a lack of clear separation of concerns, making it challenging to write and maintain automated tests effectively. Refactoring, the disciplined technique of restructuring existing code without

altering its external behavior, plays a pivotal role in enhancing the testability of the codebase and enabling more efficient and reliable autonomous testing [16,17].

The principle of separating concerns promotes modular design, where each component or module has a well-defined responsibility. By adhering to this principle, components become more focused and independent, facilitating the creation of isolated and targeted tests. This separation reduces the complexity of test setup and teardown, as dependencies and external interactions can be more easily managed. The principles of dependency inversion and dependency injection are fundamental to improving testability [18]. By decoupling components from their dependencies and injecting them from the outside, developers can substitute real dependencies with mock objects or test doubles during autonomous testing. This practice eliminates the need for complex test setups involving external systems or services, enabling more controlled and isolated testing environments [19,20].

Composition promotes higher modularity and flexibility compared to inheritance-based designs. By favoring composition, individual components can be tested independently, without the need to create and maintain complex inheritance hierarchies in the test suite. This approach simplifies the testing process and reduces the risk of unintended side effects or coupling across the inheritance hierarchy. Global state and singletons introduce tight coupling and make it difficult to isolate components during testing. These anti-patterns can lead to unpredictable behavior, as the state can be modified from multiple locations, making it challenging to set up and tear down tests. Refactoring code to eliminate global state and singletons can significantly improve testability and reduce the complexity of test setup and teardown [21,22]. Functional programming principles, such as pure functions, immutable data structures, and side-effect-free operations, can greatly enhance testability. Pure functions are inherently testable, as they produce the same output for a given input, without relying on external state or causing side effects [23]. Immutable data structures eliminate the need for complex setup and teardown procedures, as the data cannot be modified unexpectedly during the test execution.

Identifying and extracting interfaces or abstract classes from concrete implementations is a powerful technique for improving testability. By introducing these abstractions, developers can leverage mock objects and test doubles during autonomous testing, facilitating the isolation of components for focused testing. This technique promotes loose coupling and enables the substitution of dependencies with test-friendly implementations. Implementing dependency injection patterns, such as constructor injection or property injection, allows for the substitution of dependencies with mock objects or test doubles during testing [24,25]. This practice eliminates the need for complex test setups and improves the maintainability of the test suite, as dependencies can be easily swapped in and out.

Refactoring code to adhere to the Single Responsibility Principle can significantly improve modularity and testability. Each component or module should have a single, well-defined responsibility, making it easier to isolate and test its behavior without unintended side effects or dependencies on other components [26,27]. Refactoring code to remove global state and singletons is a crucial step towards enhancing testability. This can involve techniques such as introducing dependency injection, extracting state into separate classes, or encapsulating state management within defined scopes. By eliminating global state, developers can create more predictable and isolated testing environments, reducing the risk of unexpected side effects or race conditions [28]. Refactoring code to embrace functional programming principles, such as pure functions and immutable data structures, can significantly improve testability. Pure functions, which do not have side effects and always produce the same output for a given input, are inherently testable and can simplify the testing process. Immutable data structures eliminate the need for complex setup and teardown procedures, as the data cannot be modified unexpectedly during the test execution [29,30].

The Extract Method Object and Command Pattern refactoring techniques can help encapsulate complex operations or algorithms into separate objects or classes, making them more testable and reusable [31,32]. These techniques promote modularity and facilitate the creation of focused tests that exercise specific behaviors without the need for complex setup or dependencies. Design patterns such as the Observer, Mediator, and Strategy patterns can improve testability by promoting loose coupling, separating concerns, and enabling the substitution of dependencies with test-friendly implementations [33]. These patterns can be applied through refactoring to enhance the modularity and testability of the codebase.

By refactoring the codebase for testability, organizations can unlock the full potential of autonomous testing for their high-performance .NET applications. Improved modularity, loose coupling, and adherence to best practices like dependency inversion, the Single Responsibility Principle, and functional programming principles enable more efficient and reliable automated testing. This, in turn, leads to higher software quality, better performance under load, and increased confidence in the reliability and scalability of the application [34,35].

## 3 Testing interactions with test doubles

In the context of high-performance .NET applications, components often interact with external dependencies such as databases, web services, message queues, or other third-party systems [36]. These interactions can introduce complexities and dependencies that make it challenging to write and maintain autonomous tests effectively. Attempting to test against real dependencies can lead to brittle, slow, and non-deterministic tests, hindering the efficiency and reliability of the autonomous testing process. To address this issue, the concept of test doubles plays a crucial role in enabling isolated and reliable testing of component interactions. Test doubles, also known as mock objects or fake implementations, are objects that mimic the behavior of real dependencies, allowing developers to substitute them during testing [37,38]. By using test doubles, developers can isolate the component under test from its dependencies, enabling focused and deterministic testing without the need for complex setup or external system interactions [39].

Test doubles come in various forms. Dummy objects are the simplest, acting as simple placeholder objects used to satisfy method parameter requirements when the actual value of the parameter is not relevant for the test [40]. Stubs are test doubles that provide canned responses or pre-programmed behavior to simulate the expected behavior of a real dependency. They are useful for testing interactions with dependencies that have predictable outcomes, such as returning hardcoded data or simulating a successful operation [41]. Mocks are more sophisticated test doubles that can record and verify interactions between the component under test and its dependencies. They allow developers to assert that specific methods were called with the correct parameters and in the expected order, enabling detailed interaction testing [42,43]. Fakes are lightweight implementations of real dependencies, typically used when the real implementation is too complex or resource-intensive for testing purposes. They provide a simplified version of the functionality required for testing, often without the overhead of the real implementation. Spies are test doubles that record information about the interactions between the component under test and its dependencies, without modifying the behavior of the dependencies themselves [44,45]. They are useful for verifying that certain methods were called or specific state changes occurred during the test execution, without altering the behavior of the dependency itself.

Test doubles offer several benefits. They enable developers to isolate the component under test from its dependencies [46], allowing for focused and deterministic testing without the need for complex setup or external system interactions. This isolation eliminates potential sources of non-determinism and makes it easier to identify and diagnose issues within the

component under test. By removing dependencies on external systems or services, tests that utilize test doubles can execute faster, improving the overall speed and efficiency of the autonomous testing process [47]. This is particularly important for high-performance .NET applications, where testing against real dependencies can be time-consuming and resource-intensive. Test doubles provide controlled and predictable behavior, eliminating the potential for non-deterministic or flaky tests that can arise when interacting with external systems or services. This predictability ensures consistent and reliable test results, regardless of the test environment or external factors [48]. Additionally, by using test doubles, developers can test edge cases, error scenarios, and boundary conditions that may be difficult or impossible to simulate with real dependencies. This enables more thorough testing of error handling, fault tolerance, and exceptional behavior, which is crucial for ensuring the reliability and resilience of high-performance .NET applications. The use of test doubles also encourages better software design practices, such as loose coupling and the separation of concerns. By decoupling components from their dependencies, the codebase becomes more modular and testable, ultimately improving the overall quality and maintainability of the software [49,50].

In the .NET ecosystem, various frameworks and libraries are available for creating and managing test doubles. Popular testing frameworks like NUnit, MSTest, and xUnit provide built-in mocking capabilities or extensions for creating and managing test doubles, offering basic mocking functionality suitable for simple scenarios [51]. Dedicated mocking libraries like Moq, NSubstitute, and FakeItEasy offer advanced features for creating and configuring test doubles, as well as fluent APIs for specifying expected behavior and interaction verification. These libraries often provide more powerful and expressive APIs for working with test doubles, making them suitable for more complex scenarios. Isolation frameworks like Microsoft.Extensions.DependencyInjection and AutoFixture can simplify the process of creating and managing test doubles, particularly when working with dependency injection and object composition scenarios. In some cases, developers may choose to implement custom test doubles tailored to their specific needs, providing greater flexibility and control but requiring more effort and maintenance.

When implementing test doubles in high-performance .NET applications, it's essential to follow best practices and guidelines. This includes adhering to the principles of loose coupling and the separation of concerns, as well as maintaining clear and consistent naming conventions for test doubles. Additionally, developers should strive to keep test double configurations and assertions focused and concise, ensuring that tests remain readable and maintainable over time.

By effectively utilizing test doubles, organizations can improve the reliability and maintainability of their autonomous testing efforts for high-performance .NET applications. Test doubles enable developers to isolate and thoroughly test component interactions, verify edge cases and error scenarios, and maintain a deterministic and controlled testing environment, ultimately leading to higher software quality and better performance under load.

## 4 Adopting autonomous testing in organizations

Implementing autonomous testing practices within an organization can be a transformative endeavor, yielding significant improvements in software quality, reliability, and performance. However, this transition requires a strategic approach, addressing not only technical aspects but also organizational and cultural factors. Successful adoption of autonomous testing involves careful planning, training, and fostering a culture that values quality and embraces automation.

Establishing a testing culture:

1) Leadership buy-in: garnering support and commitment from organizational leadership is crucial for the successful adoption of autonomous testing. Leaders should understand the long-term benefits of investing in automated testing and provide the necessary resources and support for its implementation.
2) Quality-focused mindset: cultivating a quality-focused mindset throughout the organization is essential. This mindset should emphasize the importance of writing testable code, adhering to best practices, and prioritizing quality alongside features and performance.
3) Cross-functional collaboration: autonomous testing should be a collaborative effort involving developers, testers, and other stakeholders. Cross-functional teams can share knowledge, provide feedback, and contribute to the continuous improvement of testing practices.
4) Continuous learning and improvement: autonomous testing practices and tools are constantly evolving. Organizations should foster a culture of continuous learning and improvement, encouraging team members to stay up-to-date with the latest trends, techniques, and tools.

Implementing autonomous testing practices:
1) Gradual adoption: transitioning to autonomous testing practices should be a gradual process, starting with pilot projects or specific modules. This approach allows teams to learn, refine their practices, and build confidence before scaling to larger projects.
2) Test automation strategy: developing a comprehensive test automation strategy is crucial. This strategy should define the scope, priorities, tools, frameworks, and processes for autonomous testing, aligning with the organization's goals and quality standards.
3) Test automation framework: establishing a robust and scalable test automation framework is essential. This framework should provide a consistent and maintainable approach to writing, executing, and reporting autonomous tests, promoting reusability and collaboration across teams.
4) Continuous integration and deployment (CI/CD): integrating autonomous testing into the CI/CD pipeline ensures that every code change is automatically validated, reducing the risk of introducing regressions and enabling faster feedback cycles.
5) Performance testing infrastructure: for high-performance .NET applications, organizations should establish dedicated performance testing infrastructure and environments. This infrastructure should be capable of simulating real-world load conditions and capturing relevant performance metrics.
6) Test data management: effective test data management strategies should be implemented to ensure the availability of realistic and representative test data, while maintaining data privacy and security.
7) Monitoring and reporting: implementing comprehensive monitoring and reporting mechanisms is crucial for tracking test results, identifying failures, and analyzing test metrics over time. This data can inform decision-making and drive continuous improvement efforts.

Overcoming challenges and measuring success:
1) Training and skill development: organizations should invest in training and skill development programs to equip team members with the necessary knowledge and expertise in autonomous testing principles, tools, and frameworks.
2) Addressing resistance to change: adopting autonomous testing may face resistance from team members accustomed to manual testing practices. Addressing concerns, communicating the benefits, and providing ample support can help overcome this resistance.
3) Measuring success metrics: defining and tracking relevant success metrics is essential for evaluating the effectiveness of autonomous testing practices. Metrics such as test coverage, defect detection rates, and time-to-market can provide valuable insights.
4) Continuous improvement and adaptation: autonomous testing practices should be continuously evaluated and improved based on feedback, lessons learned, and evolving

organizational needs. Adapting to new technologies, tools, and industry best practices can help organizations stay ahead of the curve.

By successfully adopting autonomous testing practices, organizations can reap numerous benefits, including improved software quality, increased confidence in application performance and scalability, accelerated delivery cycles, and reduced long-term maintenance costs. However, this transformation requires a strategic approach, addressing both technical and organizational aspects, fostering a culture of quality and continuous improvement.

## 5 Design and testability in high-performance .NET applications

Design for testability is a critical aspect of developing maintainable, scalable, and robust high-performance .NET applications. By integrating principles of design for testability early in the design phase, developers ensure that software is easier to test, debug, and maintain. Adopting a modular approach facilitates the independent testing of each component, which enhances fault isolation and reduces the complexity of debugging when issues arise. Each module's design should focus on clear, defined functionalities that allow for targeted testing without dependencies on other parts of the application.

Encapsulation and abstraction are also fundamental. Proper encapsulation hides the internal state and behavior of objects, exposing only what is necessary through interfaces. This practice minimizes the coupling between components, simplifying unit testing and reducing the risk of side effects from code changes. Similarly, implementing a clear separation of concerns helps in organizing code around distinct features or behaviors, simplifies the understanding of the application, and isolates tests to specific functionalities, thereby minimizing the impact of changes.

Interface-based programming uses interfaces and abstract classes to promote a loose coupling design where implementation details are hidden behind an interface. This approach is vital for substituting real objects with mocks or stubs during unit testing, allowing for testing components in isolation.

Refactoring is the process of altering the internal structure of the code to make it more amenable to testing, without changing its external behavior. Effective refactoring includes simplifying complex methods, breaking down large classes into smaller ones, removing redundant code, and introducing dependency injection. Dependency injection manages dependencies between objects by injecting dependencies, rather than creating them internally, making the code more testable and allowing for the replacement of actual dependencies with mock objects during testing.

Test-Driven Development is a software development approach that emphasizes writing tests before functional code. This method leads to early bug detection, design improvement, and serves as documentation for the application, helping new developers understand the intended behavior of the system.

Continuous integration and testing with systems automate the build and testing process, providing immediate feedback on the system’s health. Utilizing automated testing frameworks such as NUnit and xUnit for .NET applications allows for automated execution of tests to ensure that all parts of the application behave as expected after changes or enhancements. Including performance testing in the continuous integration pipeline ensures that the application performs within required parameters even as new features are added or existing ones are modified.

While the integration of design for testability brings numerous benefits, it also presents challenges such as the potential overhead in the initial development phases and the necessity for developers to possess specialized skills in advanced testing techniques. Best practices include conducting regular code reviews focused on testability, maintaining comprehensive

documentation on the application's architecture and testing practices, and fostering a culture of quality and rigorous testing within the team.

Embracing design and testability principles profoundly impacts the development lifecycle of high-performance .NET applications, facilitating better testing and maintenance and aligning with modern agile practices to support faster, more reliable releases and a robust final product.

To illustrate autonomous testing in the context of high-performance .NET applications, we can consider a simple example using C# and the NUnit framework. NUnit is a popular testing framework that supports test automation, running parallel tests, and integrating with continuous integration tools. In this example, we will create a sample class and demonstrate how to write an autonomous test for it using NUnit.

1) Sample class to Test: let's assume we have a simple service in a .NET application that calculates the yearly interest for given principal amount and rate. Here's how the service might look:

```
public class InterestCalculator
{
    public double CalculateYearlyInterest(double principal, double rate)
    {
        if (principal < 0 || rate < 0)
            throw new ArgumentException("Principal and rate must be non-negative.");
        return principal * rate;
    }
}
```

2) Setting up NUnit: to use NUnit, you need to install the NUnit framework and the NUnit3TestAdapter packages via NuGet in your Visual Studio project. Additionally, install the Microsoft.NET.Test.Sdk package to integrate testing into the .NET Core project.

3) Writing an autonomous test: here's how you can write an autonomous test for the `InterestCalculator` class:

```
using NUnit.Framework;
using System;
namespace HighPerformanceApp.Tests
{
    [TestFixture]
    public class InterestCalculatorTests
    {
        private InterestCalculator _calculator;
        [SetUp]
        public void Setup()
        {
            // Initialize the InterestCalculator before each test
            _calculator = new InterestCalculator();
        }
        [Test]
        public void CalculateYearlyInterest_WithPositiveValues_ReturnsCorrectResult()
        {
            // Arrange
            double principal = 1000; // Principal amount
            double rate = 0.05;      // 5% interest rate
            // Act
            double result = _calculator.CalculateYearlyInterest(principal, rate);
            // Assert
```

```
            Assert.AreEqual(50, result, "The calculated interest should be 50 for the given
principal and rate.");
        }
        [Test]
        public void
CalculateYearlyInterest_WithNegativeValues_ThrowsArgumentException()
        {
            // Arrange
            double principal = -1000; // Invalid principal amount
            double rate = 0.05;       // 5% interest rate
            // Act & Assert
            var ex = Assert.Throws<ArgumentException>(() =>
_calculator.CalculateYearlyInterest(principal, rate));
            Assert.That(ex.Message, Is.EqualTo("Principal and rate must be non-
negative."));
        }
    }
}
```

Explanation:

1) SetUp method: the `[SetUp]` attribute marks the method that runs before each test, setting up necessary objects or states. In this example, it initializes the `InterestCalculator` instance.
2) Test methods: each `[Test]` method represents a single test case. The first test checks for the correct calculation of interest, and the second test ensures that the method throws an exception for invalid input. Assertions are used to validate the expected outcomes.
3) Running the tests: tests can be run directly from Visual Studio's Test Explorer or through command line, and they can be integrated into a CI/CD pipeline to execute automatically whenever changes are pushed to the source code repository.

This example of autonomous testing demonstrates how tests can independently verify the functionality of parts of a high-performance .NET application without manual intervention, ensuring that changes do not break existing functionality and meet the expected outcomes. This approach is crucial for maintaining high-quality software in continuous development and deployment environments.

## 6 Conclusion

This article has extensively explored the multifaceted domain of autonomous testing, particularly within the sphere of high-performance .NET applications. We have identified that autonomous testing is not merely an option but a necessity for ensuring the reliability, performance, and scalability of applications that are increasingly complex and data-intensive. These applications, designed to handle substantial data volumes and support numerous concurrent users, demand a robust testing framework that can operate without human intervention to validate functionality under various conditions.

Through the implementation of autonomous testing, we have demonstrated the potential for significantly increased test coverage and thoroughness. This methodology allows for exhaustive testing across a wide range of scenarios, including extreme load conditions and high concurrency, facilitating early defect detection and consistent, reliable outcomes. The integration of autonomous testing into the continuous integration and deployment pipelines further ensures that each code change is meticulously validated, enhancing the software quality and accelerating the delivery process.

Moreover, our discussion emphasized the importance of testability in software design. By refactoring and adhering to principles such as separation of concerns and dependency

inversion, we improve the testability of the codebase. This not only simplifies the testing process but also enhances the maintainability and scalability of the application, thereby supporting more efficient and reliable autonomous testing.

The challenges associated with implementing autonomous testing, particularly in organizational and cultural contexts, were also addressed. Successful adoption requires not only technical adjustments but also a shift in organizational culture towards valuing quality and embracing automation. By fostering a testing culture that prioritizes continuous learning and improvement, organizations can better equip their teams with the necessary tools and knowledge to leverage autonomous testing effectively.

In conclusion, autonomous testing represents a pivotal advancement in the field of software engineering, particularly for .NET applications. By embracing these principles and integrating them into the development lifecycle, organizations can significantly enhance the quality and reliability of their applications. As we continue to navigate the complexities of modern software development, autonomous testing will undoubtedly play a critical role in shaping the future of technology development, ensuring that applications are not only functional but also robust and secure in the face of evolving demands and challenges.

## References

1. G. Falco, L. H. Gilpin, 2021 IEEE International Conference on Autonomous Systems (ICAS), Montreal, QC, Canada, 1-5 (2021)
2. P. Ding, F. Wang, D. Gu, H. Zhou, Q. Gao, X. Xiang, 2018 IEEE 8th Annual International Conference on CYBER Technology in Automation, Control, and Intelligent Systems (CYBER), Tianjin, China, 1351-1355 (2018)
3. Z. Xu, H. Li, Y. Chen, S. Liu, Z. Wan, 2023 6th International Conference on Electronics Technology (ICET), Chengdu, China, 1156-1160 (2023)
4. N. V. Andreyanov, A. S. Sytnik, M. P. Shleymovich, IOP Conference Series: Earth and Environmental Science **988(3)**, 032002 (2022)
5. G. Mingaleeva, O. Afanaseva, P. Zunino, D. T. Nguen, D. N. Pham, Energies **13(21)**, 5848 (2020)
6. G. R. Mingaleeva, M. F. Nabiullina, D. N. Pham, 2023 International Conference on Industrial Engineering, Applications and Manufacturing (ICIEAM), 233-238 (2023)
7. Y. Smirnov, A. Kalyashina, R. Zaripova, International Russian Automation Conference (RusAutoCon), 913-917 (2022)
8. Z. Gizatullin, R. Gizatullin, 2023 International Conference on Industrial Engineering, Applications and Manufacturing (ICIEAM), Sochi, Russian Federation, 261-265 (2023)
9. Z. M. Gizatullin, M. P. Shleimovich, Russ. Aeronaut **66**, 154-161 (2023)
10. S. Lyasheva, R. Safina, M. Shleymovich, 2023 International Conference on Industrial Engineering, Applications and Manufacturing, 797-802 (2023)
11. M. Shleymovich, R. Safina, 2022 International Russian Automation Conference, 289-293 (2022)
12. R. M. Shakirzyanov, A. A. Shakirzyanova, 2021 International Russian Automation Conference (RusAutoCon), 714-718 (2021)
13. Y. I. Soluyanov, A. I. Fedotov, D. Y. Soluyanov, A. R. Akhmetshin, IOP Conference Series: Materials Science and Engineering **860(1)**, 012026 (2020)
14. A. V. Chupaev, R. S. Zaripova, R. R. Galyamov, A. Y. Sharifullina, E3S Web of Conferences **124**, 03013 (2019)

15. L. V. Plotnikova, R. R. Giniyatov, S. Y. Sitnikov, M. A. Fedorov, R. S. Zaripova, IOP Conference Series: Earth and Environmental Science **288**, 012069 (2019)
16. M. Tyurina, A. Porunov, A. Nikitin, R. Zaripova, G. Khamatgaleeva, Lecture Notes in Mechanical Engineering, 391-402 (2022)
17. E. I. Gracheva, O. V. Naumov, Journal of Pharmacy and Technology **8**, 4, 26763-26770 (2016)
18. D. D. Micu, I. V. Ivshin, E. I. Gracheva, O. V. Naumov, A. N. Gorlov, E3S Web of Conferences **124**, 02013 (2019)
19. O. Soloveva, S. Solovev, R. Zaripova, F. Khamidullina, M. Tyurina, E3S Web of Conferences **258**, 11010 (2021)
20. R. F. Gibadullin, N. S. Marushkai, 2021 International Conference on Industrial Engineering, Applications and Manufacturing (ICIEAM), 404-409 (2021)
21. V. O. Kozelkova, G. A. Ovseenko, V. I. Karachin, T. Van Tung, N. C. Kien, R. S. Kashaev, 4th International Youth Conference on Radio Electronics, Electrical and Power Engineering (REEPE), 1-4 (2022)
22. R. F. Gibadullin, I. S. Vershinin, R. Sh. Minyazev, 2017 International Conference on Industrial Engineering, Applications and Manufacturing (ICIEAM), 1-6 (2017)
23. T. Petrov, A. Safin, E3S Web of Conferences **178**, 01049 (2020)
24. V. O. Kozelkova, G. A. Ovseenko, V. I. Karachin, N. C. Kien, T. Van Tung, O. V. Kozelkov, 4th International Youth Conference on Radio Electronics, Electrical and Power Engineering (REEPE), 1-5 (2022)
25. R. F. Gibadullin, G. A. Baimukhametova, M. Yu. Perukhin, 2019 International Conference on Industrial Engineering, Applications and Manufacturing (ICIEAM), 1-7 (2019)
26. G. R. Rakhmatullina, E. A. Pankova, O. V. Fukina, M. Khayytov, L. V. Chapaeva, Journal of Physics: Conference Series **2270(1)**, 012056 (2022)
27. V. A. Gerasimov, M. G. Nuriev, D. A. Gashigullin, 2022 International Russian Automation Conference (RusAutoCon), 75-79 (2022)
28. S. R. Khasanov, E. I. Gracheva, M. I. Toshkhodzhaeva, S. T. Dadabaev, D. S. Mirkhalikova, E3S Web of Conferences **178**, 01051 (2020)
29. Z. M. Gizatullin, R. M. Gizatullin, M. G. Nuriev, 2020 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus), 120-123 (2020)
30. S. Lyasheva, M. Shleymovich, R. Shakirzyanov, 2019 International Multi-Conference on Industrial Engineering and Modern Technologies (FarEastCon), 1-6 (2019)
31. E. Gracheva, M. Toshkhodzhaeva, O. Rahimov, S. Dadabaev, D. Mirkhalikova, S. Ilyashenko, V. Frolov, International Journal of Technology **11**, 8 (2020)
32. M. Shakirzyanov, R. Gibadullin, M. Nuriyev, E3S Web of Conferences **419**, 02029 (2023)
33. T. Petrov, A. Safin, E3S Web of Conferences **178**, 01016 (2020)
34. J. Yoqubjonov, R. Gibadullin, M. Nuriev, E3S Web of Conferences **431**, 07011 (2023)
35. I. Viktorov, R. Gibadullin, E3S Web of Conferences **431**, 05012 (2023)
36. R. F. Gibadullin, I. S. Vershinin, M. M. Volkova, 2020 International Multi-Conference on Industrial Engineering and Modern Technologies (FarEastCon), 1-7 (2020)

37. R. F. Gibadullin, M. Yu. Perukhin, B. I. Mullayanov, 2020 International Multi-Conference on Industrial Engineering and Modern Technologies (FarEastCon), 1-6 (2020)
38. S. N. Cherny, R. F. Gibadullin, 2022 International Conference on Industrial Engineering, Applications and Manufacturing (ICIEAM), 965-970 (2022)
39. V. A. Raikhlin, I. S. Vershinin, R. F. Gibadullin, Journal of Physics: Conference Series **2096**, 012160 (2021)
40. R. F. Gibadullin, I. S. Vershinin, R. Sh. Minyazev, 2018 International Conference on Industrial Engineering, Applications and Manufacturing (ICIEAM), 1-6 (2018)
41. V. A. Raikhlin, R. F. Gibadullin, I. S. Vershinin, Lobachevskii Journal of Mathematics **43**, 2, 455-462 (2022)
42. G. A. Ovseenko, R. S. Kashaev, O. V. Kozelkov, T. K. Filimonova, T. S. Evdokimova, A. M. Mardanova, 5th International Youth Conference on Radio Electronics, Electrical and Power Engineering (REEPE) **5**, 1-5 (2023)
43. R. Zaripova, A. Nikitin, Y. Hadiullina, E. Pokaninova, M. Kuznetsov, E3S Web of Conferences **288**, 01072 (2021)
44. I. N. Madyshev, V. V. Kharkov, N. Z. Dubkova, M. G. Kuznetsov, AIP Conference Proceedings **2647**, 1 (2022)
45. Z. M. Gizatullin, M. S. Shkinderov, R. R. Mubarakov, Proceedings of the 2022 Conference of Russian Young Researchers in Electrical and Electronic Engineering, 1350-1353 (2022)
46. Z. Gizatullin, M. Shkinderov, 2019 International Russian Automation Conference, 8867761 (2022)
47. A. G. Ilyin, A. S. Mahdi Khafaga, V. Yunusova, 2021 Systems of Signals Generating and Processing in the Field of on Board Communications, 1-4 (2021)
48. N. Andreyanov, M. Shleymovich, A. Sytnik, 2022 International Conference on Industrial Engineering, Applications and Manufacturing (ICIEAM), 880-885 (2022)
49. E. Kozlov, R. Gibadullin, E3S Web of Conferences **474**, 02031 (2024)
50. R. M. Petrova, E. Gracheva, 2023 5th International Conference on Control Systems, Mathematical Modeling, Automation and Energy Efficiency (SUMMA), Lipetsk, Russian Federation, 1049-1055 (2023)
51. R. M. Petrova, E. Gracheva, 2023 5th International Conference on Control Systems, Mathematical Modeling, Automation and Energy Efficiency (SUMMA), Lipetsk, Russian Federation, 1056-1061 (2023)