Date of publication MAR-31, 2024, date of current version NOV-13, 2023.
www.computingonline.net / computing@computingonline.net

Print ISSN 1727-6209
Online ISSN 2312-5381
DOI 10.47839/ijc.23.1.3437

A Comparative Study of Data Annotations and Fluent Validation in .NET

VOLODYMYR SAMOTYY[1,2], ULYANA DZELENDZYAK[2], NAZAR MASHTALER[2]

[1] Department of Automatic Control and Information Technology, Cracow University of Technology, Warszawska 24, Cracow, 31155, Poland, vsamotyy@pk.edu.pl

[2] Department of Computerized Automatic Systems, Lviv Polytechnic National University, S. Bandery 12, Lviv, 79013, u.dzelendzyak@gmail.com, nazar.o.mashtaler@lpnu.ua

Corresponding author: Volodymyr Samotyy (e-mail: vsamotyy@pk.edu.pl).

ABSTRACT This article presents a comparative study of two validation approaches in .NET – Data Annotations and Fluent Validation – analyzing their syntax, functionality, and other factors (such as readability, maintainability, and performance). The study begins by examining the Data Annotations approach, an in-built validation mechanism in the .NET Framework that uses validation attributes to validate model properties. While Data Annotations offers a simple syntax and is well-known to .NET developers, it may not be ideal for more complex validation scenarios and could become verbose and difficult to maintain. The study then introduces the Fluent Validation approach, which utilizes a fluent syntax to define validation rules in a more expressive, readable, and concise manner. With its flexible architecture and fluent API (application programming interface), Fluent Validation provides greater control over the validation process, enabling better maintainability and performance. The study concludes by highlighting the merits and drawbacks of both approaches, noting that the choice of validation approach will depend on the specific requirements of the project at hand.

KEYWORDS validation; NET Data Annotations; Fluent Validation; syntax; readability; maintainability; performance; expressive syntax; flexible architecture.

I. INTRUDUCTION

VALIDATION is a crucial factor concerning web form design in e-commerce. The Baymard Research Institute, which specializes in usability research, conducted a study on the use of web forms. They analyzed 37 open sources, each of which conducted its research on web forms. The results showed that nearly 70% of users abandoned their shopping carts, with 60% of these abandonments occurring because they were simply exploring the product and not yet ready to make a purchase. However, 27% of users specifically cited the web form as the reason for abandoning the checkout, citing issues such as complexity, length, and validation problems [1, 2].

These findings indicate that improving the web form interface could lead to a significant increase in conversion rates, by up to 35%. Given the size of the e-commerce market in the US, which is 260 billion dollars, simply improving the web form by reducing the number of fields and ensuring proper validation could result in an additional 260 billion dollars in annual earnings for the industry.

Validation is such a serious concept that there is even a special ISO standard that defines validation. “Validation – confirmation, through the provision of objective evidence, that the requirements for a specific intended use or application have been fulfilled” (ISO 9000:2005) [3, 4].

The purposes of validation are to obtain the correct data in the correct format for subsequent processing, to protect the user from all sorts of data interception, and for application protection. We do not want hacking to be enabled through the web form, allowing others to gain access to the user's data entered earlier.

The most important field of client validation, without which nothing can be processed, is validation on the server. This is the main validation method and it is responsible for security. Client validation primarily affects the user’s experience, also UX. Effective validation contributes to and significantly enhances positive user’s interactions and experiences on a website's UI/UX [5, 6].

What makes a good validation? A study was undertaken by Luke Wroblewski, a well-known usability expert [7, 8]. He concluded that, firstly, the validation must be in the right place. This refers to the form of a message about a certain error, where the user's error message is displayed so that it is convenient for the user to work with.

Secondly, validation must occur at the right time (inline validation), no one wants to fill out the entire form, only to find that, when submitting, there are some errors which means they need to go back and start over. It would be even worse if some data was lost.

Thirdly, the field needs to be the right color. We are used to seeing error fields highlighted in red, but research shows that

one of the best options is still orange. It does not overwhelm the user as much; orange is also a color that blind people can see.

Fourthly, understandable language must be used. This is a broad concept. We must explicitly tell the users what they did wrong and how to fix it, and not just highlight the erroneously filled field and leave the user to deal with it alone.

According to Luke Wroblewski, inline validation leads to improvements in the following metrics [7, 8].

- a 22% increase in success rates;
- a 22% decrease in errors made;
- a 31% increase in satisfaction rating;
- a 42% decrease in completion times;
- a 47% decrease in the number of eye fixations.

We have reviewed why validation is important from a user's experience point of view and how it can help us to make our product more user-friendly. Also, validation is important on the back-end side. The validation of data inputs for APIs is critical in ensuring the quality and accuracy of data being processed and stored. It serves multiple purposes such as data integrity, data security, and user experience.

In terms of data integrity, input validation helps to ensure that the data being processed and stored is accurate, complete, and consistent. Input validation ensures consistency by checking that the data entered in a system meets specified rules, such as format, range, and type. This helps prevent inaccurate or incomplete information from being stored, maintaining the integrity and reliability of the stored data. It is achieved by checking the data for correctness and completeness before it is processed and stored, thus reducing the likelihood of errors in the data.

In terms of data security, input validation plays a crucial role in protecting against malicious attacks such as cross-site scripting (XSS) and SQL (structured query language) injection. By validating inputs, APIs can prevent these types of attacks by blocking malicious data inputs that contain malicious code. This helps to protect sensitive data and prevent unauthorized access to data and systems.

Finally, input validation also has an impact on user's experience. By providing clear and concise feedback to users regarding their data inputs, APIs can improve the user's experience. This can result in increased user's satisfaction, reduced errors, and faster completion times. Furthermore, the research by Luke Wroblewski shows that proper validation can lead to significant improvements in completion times, error rates, and satisfaction ratings.

Real-life examples underscore the importance of server-side validation.

Preventing hacking attacks involves validating user's input, such as login credentials, to ensure it adheres to the correct format and meets security requirements like minimum password length. Failure to validate inputs can expose vulnerabilities, allowing malicious users to inject harmful code or steal sensitive information.

Enforcing business rules is exemplified by an e-commerce API validating user's input for a product order, confirming it falls within specified parameters like the available stock of the product. This validation minimizes errors and ensures the API operates in alignment with established business rules.

Improving user's experience through API validation of user's input enables the provision of clear error messages and guidance to help user correct mistakes. For example, a weather API might verify the validity of a city name, offering a list of suggestions if the input is invalid.

Maintaining data integrity is crucial for an API storing data in a database. Validation of inputs ensures data meets constraints such as data types or length requirements, preserving data integrity and preventing errors that could cause the API to crash or produce incorrect results.

In conclusion, input validation is a crucial aspect of development that should not be overlooked. Proper validation strategies implemented in APIs ensure data quality, accuracy, enhanced data security, and an improved user's experience [9, 10].

II. REVIEW OF FLUENT INTERFACE APPROACH

It has been established that proper validation procedures are crucial for the successful creation and implementation of any product. This paper aims to examine the traditional validation methods utilized in .Net and compare them with a proposed alternative approach [11, 12].

Before comparing classic .Net validation to alternative approaches, it is important to examine the Fluent Interface approach. The Fluent Interface design pattern was first introduced and widely recognized by Martin Fowler [13, 14], a renowned software developer and author in the field of software engineering. The pattern is characterized using method chaining, where the result of each method call is passed as an input to the subsequent method in the chain, creating a readable, concise, and expressive syntax for complex operations. This approach was inspired by natural language and was intended to make code more readable and easier to understand.

The standard Object-Oriented Programming (OOP) approach involves creating objects, defining classes and methods, and then using those objects to perform specific operations. In this approach, each method typically returns a value or updates the state of the object and is called in a separate statement. An example is given in Code Snippet 1.

```
class BankAccount
{
    public int AccountNumber { get; set; }
    public string AccountHolder { get; set; }
    public int Balance { get; set; }
    public string AccountType { get; set; }
    public string BankName { get; set; }

    public void Deposit(int amount)
    {
        this.Balance += amount;
    }
    public void Withdraw(int amount)
    {
        this.Balance -= amount;
    }
}

// Create a new bank account with the specified properties
BankAccount account = new BankAccount
{
    AccountNumber = 123456,
    AccountHolder = "John Doe",
    Balance = 0,
    AccountType = "Savings",
    BankName = "MyBank"
};

// Deposit an amount into the account
account.Deposit(100);

// Withdraw an amount from the account
account.Withdraw(50);
```

Code Snippet 1 – Standard Object-Oriented Programming (OOP) approach

The code defines a C# class called "BankAccount" with properties representing account details such as number, holder, balance, account type, and bank name. It includes methods for depositing and withdrawing funds. An instance of this class is created, representing a bank account for "John Doe" with an initial balance of 0 in a savings account at "MyBank". The code then simulates a deposit of 100 units and a withdrawal of 50 units from the account. In summary, the class encapsulates basic banking functionality with properties for account details and methods for financial transactions.

On the other hand, Fluent Interfaces provide a way to create a more readable, expressive, and natural-language-like syntax for using objects. Instead of using separate statements for each method call, methods are chained together using a fluid and readable syntax. An example is given in Code Snippet 2. In this example, a fluent interface pattern is introduced. In the updated "BankAccount" class, setter methods like "SetAccountNumber "and "SetAccountHolder" are modified to return the instance of the "BankAccount" class itself, allowing for method chaining. This enables a more concise and expressive way to create an account, set its properties, deposit funds, and withdraw funds in a single chain of method calls. The fluent interface pattern is a design choice that enhances readability and provides a more streamlined way to interact with the "BankAccount" class, especially when performing multiple operations in sequence.

```
class BankAccount
{
    public int AccountNumber { get; set; }
    public string AccountHolder { get; set; }
    public int Balance { get; set; }
    public string AccountType { get; set; }
    public string BankName { get; set; }

    public BankAccount SetAccountNumber(int accountNumber)
    {
        this.AccountNumber = accountNumber;
        return this;
    }

    public BankAccount SetAccountHolder(string accountHolder)
    {
        this.AccountHolder = accountHolder;
        return this;
    }

    public BankAccount Deposit(int amount)
    {
        this.Balance += amount;
        return this;
    }

    public BankAccount Withdraw(int amount)
    {
        this.Balance -= amount;
        return this;
    }
}

// Create a new bank account with the specified properties, deposit an amount
and withdraw an amount in a single chain of method calls
BankAccount account = new BankAccount()
    .SetAccountNumber(123456)
    .SetAccountHolder("John Doe")
    .Deposit(100)
    .Withdraw(50);
```

Code Snippet 2 – Fluent Interfaces approach

The Fluent Interface approach provides more expressive and readable code, as each method call is part of a chain, and the code reads like a sentence in natural language. This makes it easier to see the entire process of creating a bank account, setting its properties, and performing transactions all in one place, without having to switch between multiple statements.

Fluent interfaces are commonly used in domain-specific languages (DSLs), where they can provide a more human-readable syntax for defining complex operations. They are also used in APIs, where they can make it easier to construct complex object configurations or perform multiple operations in a single chain of method calls [13, 14]. Table 1 lists the advantages and disadvantages of the Fluent Interface approach.

Table 1. The advantages and disadvantages of the fluent interface approach

Advantages	Disadvantages
Improved code readability: Fluent Interfaces can make the code more readable and expressive, as they allow developers to write code that resembles natural language. Increased code expressiveness: Fluent Interfaces allow developers to write code that is more expressive and easier to understand, as the code can describe complex operations more intuitively and straightforwardly. Better encapsulation: Fluent Interfaces can help encapsulate complex logic behind a simple and intuitive API, making it easier for other developers to use the code. Improved code organization: Fluent Interfaces can help organize code into logical blocks and separate functionality, making it easier to maintain and reuse. Better type checking: Fluent Interfaces can help enforce type checking and prevent certain types of errors from occurring, as the compiler will catch errors early in the development process. Improved code refactoring: Fluent Interfaces can help simplify code refactoring, as it can make it easier to identify and isolate code blocks that need to be changed.	Increased complexity: Fluent Interfaces can make the code more complex and harder to understand, especially for other developers who may not be familiar with this pattern. Overuse of method chaining: Overuse of method chaining can lead to unreadable and hard-to-maintain code. Error-prone: Fluent Interfaces can be error-prone, especially if the API is not designed correctly. It can be difficult to catch errors early in the development process. Hard to debug: Debugging can be difficult, as errors in the Fluent Interface can be hard to trace back to the source. Limited compatibility: Fluent Interfaces are not always compatible with all programming languages and may not be well supported by certain tools and libraries.

This approach is commonly employed in different domains, involving libraries for database access and query building, testing frameworks like JUnit and TestNG, text processing libraries such as Apache Commons Lang and Guava, build tools like Gradle and Maven, and web page scraping and parsing libraries.

The validation approach that is considered in this article is based on the Fluent Interface method [15, 16].

III. COMPARING FLUENT VALIDATION WITH DATA ANNOTATION

Net ecosystem already has an open-source solution that allows us to use the Fluent Interface approach in validation. Fluent Validation is a .NET NuGet package for implementing model validation that is both readable and maintainable [17, 18]. It provides a fluent API for defining validation rules for .NET models in a way that is intuitive and expressive. Fluent Validation is compatible with popular .NET frameworks like ASP.NET MVC, Web API, and NancyFX.

One of the key benefits of Fluent Validation is its ability to encapsulate all validation logic within a single class for each model. This makes it easy to maintain and test the validation rules and helps to ensure the separation of concerns between the model, the view, and the controller.

Fluent Validation supports a variety of validation rules out of the box, including:

- Required fields;
- String length constraints;
- Regular expression matching;
- Numeric ranges;
- Date and time comparisons;
- Predicate-based rules;

• Custom validation rules using delegate functions.

It also supports the validation of complex objects, collections, and nested properties. Fluent Validation allows the encapsulation of all validation logic within a single class for each model, making it easy to maintain and test the validation rules. It also integrates well with Dependency Injection, making it easy to use in a variety of applications [19, 20].

The same validation logic can be implemented using the classic Data Annotation approach and this can be compared with the Fluent Validation approach. Code Snippet 3 shows an example of classic Data Annotation. The code introduces a class called Customer with three properties: Name, Email, and Age. To enforce data integrity, each property is adorned with specific validation attributes. For the Name property, it requires a non-null value and imposes a maximum length of 50 characters. The Email property must be a valid email address and is also mandatory. Lastly, the Age property is subject to two conditions: it must have a non-null value, and its value must be 18 or higher. These annotations act as clear and concise rules for validating and maintaining the correctness of customer data.

```
public class Customer
{
    [Required]
    [StringLength(50, ErrorMessage = "Name cannot be longer than 50
characters.")]
    public string Name { get; set; }

    [Required]
    [EmailAddress]
    public string Email { get; set; }

    [Required]
    [Range(18, int.MaxValue, ErrorMessage = "Age must be at least 18.")]
    public int Age { get; set; }
}
```

Code Snippet 3 – Classic Data Annotation validation approach

Code Snippet 4 shows a Fluent Validation example. The code implements the same logic as in Code Snippet 3 but uses a different approach.

```
public class CustomerValidator : AbstractValidator<Customer>
{
    public CustomerValidator()
    {
        RuleFor(x => x.Name).NotEmpty().WithMessage("Name is
required.").MaximumLength(50);
        RuleFor(x => x.Email).NotEmpty().WithMessage("Email is
required.").EmailAddress();
        RuleFor(x => x.Age).NotEmpty().WithMessage("Age is
required.").GreaterThanOrEqualTo(18);
    }
}
```

Code Snippet 4 – Fluent validation approach

In both examples, the validation rules are defined in the Customer class. With data annotations, the rules are defined using attributes applied to the properties. With Fluent Validation, the rules are defined using a fluent interface in a separate validator class.

Data annotations are attributes that can be applied to model properties to specify validation rules. They are part of the 'System.ComponentModel.DataAnnotations' namespace and include attributes such as 'Required', 'StringLength', and 'RegularExpression'.

Data annotations are simple to use and easy to understand but can only provide basic validation [21].

Fluent Validation, on the other hand, is a library that provides a more flexible and powerful way to validate objects. It uses a fluent interface to define validation rules, which makes it easier to read and maintain. Fluent Validation allows for complex validation rules and can be more easily integrated into a larger validation framework.

There are often cases when predefined validators are not enough, and custom ones need to be created. Let us implement a custom validator that checks if a string property is a valid ISBN (International Standard Book Number) code.

Using the classic approach, we need to create a class that inherits from the 'ValidationAttribute' class and overrides the 'IsValid' method. In the 'IsValid' method the ISBN code is checked if it is valid using a regular expression. The details are provided in Code Snippet 5.

```
using System.ComponentModel.DataAnnotations;
using System.Text.RegularExpressions;

public class ISBNAttribute : ValidationAttribute
{
    protected override ValidationResult IsValid(object value, ValidationCon-
text validationContext)
    {
        var isbn = value as string;

        // Check if the ISBN code is valid using a regular expression
        var regex = new Regex(@"^(97(8|9))?\d{9}(\d|X)$");
        if (!regex.IsMatch(isbn))
        {
            return new ValidationResult("Invalid ISBN code.");
        }

        return ValidationResult.Success;
    }
}
```

Code Snippet 5 – Classic Data Annotation custom validation

This custom validation attribute can be used by applying it to a string property in the model:

```
public class Book
{
    [ISBN]
    public string ISBN { get; set; }
}
```

Code Snippet 6 – Data validation attribute

The 'IsValid' method of the custom validation attribute will be automatically called by the validation framework to perform the ISBN code validation. If the validation fails, a 'ValidationResult' with an error message will be returned.

Code Snippet 7 shows an equivalent example of ISBN validation using Fluent Validation:

```
using FluentValidation;
using System.Text.RegularExpressions;

public class BookValidator : AbstractValidator<Book>
{
    public BookValidator()
    {
        RuleFor(x => x.ISBN).NotEmpty().WithMessage("ISBN is required.")
            .Must(IsValidISBN).WithMessage("Invalid ISBN code.");
    }

    private bool IsValidISBN(string isbn)
    {
        // Check if the ISBN code is valid using a regular expression
        var regex = new Regex(@"^(97(8|9))?\d{9}(\d|X)$");
        return regex.IsMatch(isbn);
    }
}
```

Code Snippet 7 – Fluent validation custom validation

In this example, the validation rules are defined using Fluent Validation's fluent interface in a separate 'BookValidator' class. The 'RuleFor' method is used to specify the validation rules for the 'ISBN' property. The 'Must' method is used to specify a custom validation method

'IsValidISBN' that will perform the ISBN code validation that is the same as in Code Snippet 5. If the validation fails, an error message is returned using the 'WithMessage' method.

The Fluent Interface approach requires less code, and a developer does not need to know the structure of the .Net validation hierarchy [22].

As the final step in our overview, we create unit tests for both variants of ISBN validation. An example of a unit test for an ISBN validator using custom validation attributes is shown in Code Snippet 8. This code defines a set of unit tests for validating ISBN attributes in C#. The tests cover scenarios such as checking if a valid ISBN returns true, an invalid ISBN returns false, and a null ISBN returns false. The tests create instances of an 'ISBNAttributeTestModel', representing a model with an ISBN property, and use the 'Validator.TryValidateObject' method to perform validation based on attributes. Assertions such as 'Assert.IsTrue' and 'Assert.IsFalse' confirm the expected validation outcomes.

```
[TestClass]
public class ISBNAttributeTests
{
    [TestMethod]
    public void ISBNAttribute_ValidISBN_ReturnsTrue()
    {
        var isbn = new ISBNAttributeTestModel { ISBN = "0735619670" };
        var validationContext = new ValidationContext(isbn);
        var results = new List<ValidationResult>();
        var isValid = Validator.TryValidateObject(isbn, validationContext,
results, true);

        Assert.IsTrue(isValid);
    }

    [TestMethod]
    public void ISBNAttribute_InvalidISBN_ReturnsFalse()
    {
        var isbn = new ISBNAttributeTestModel { ISBN = "invalidCode" };
        var validationContext = new ValidationContext(isbn);
        var results = new List<ValidationResult>();
        var isValid = Validator.TryValidateObject(isbn, validationContext,
results, true);

        Assert.IsFalse(isValid);
    }

    [TestMethod]
    public void ISBNAttribute_NullISBN_ReturnsFalse()
    {
        var isbn = new ISBNAttributeTestModel { ISBN = null };
        var validationContext = new ValidationContext(isbn);
        var results = new List<ValidationResult>();
        var isValid = Validator.TryValidateObject(isbn, validationContext,
results, true);

        Assert.IsFalse(isValid);
    }
}
```

Code Snippet 8 – Unit tests coverage for classic Data Annotation custom validation

An example of a unit test for an ISBN validator using Fluent Validation is shown in Code Snippet 9.

```
[TestClass]
public class ISBNValidatorTests
{
    [TestMethod]
    public void ISBNValidator_ValidISBN_ReturnsTrue()
    {
        var validator = new ISBNValidator();
        var result = validator.Validate(new ISBN { Value = "0735619670" });

        Assert.IsTrue(result.IsValid);
    }

    [TestMethod]
    public void ISBNValidator_InvalidISBN_ReturnsFalse()
    {
        var validator = new ISBNValidator();
        var result = validator.Validate(new ISBN { Value = "invalidCode" });

        Assert.IsFalse(result.IsValid);
    }

    [TestMethod]
    public void ISBNValidator_NullISBN_ReturnsFalse()
    {
        var validator = new ISBNValidator();
        var result = validator.Validate(new ISBN { Value = null });

        Assert.IsFalse(result.IsValid);
    }
}
```

Code Snippet 9 – Unit tests coverage for Fluent validation

This code snippet introduces a different approach to ISBN validation using a custom 'ISBNValidator' class. Instead of relying on attribute-based validation as in the Conde Snippet 8, it uses a separate validator class that takes an 'ISBN' object and returns a validation result. The tests instantiate the 'ISBNValidator', call its Validate method with instances of 'ISBN', and then use assertions to verify the validity of the 'ISBN' values.

For unit testing purposes, Fluent Validation can be more convenient, as it provides a fluent interface for defining validation rules that can be easily unit tested. It also separates the validation logic from the model, making it more flexible and easier to test individual validation rules in isolation [23].

With custom validation attributes, unit testing can be a bit more cumbersome as it requires the creation of a validation context and the use of the 'Validator' class to test the validation logic [24]. However, it is still possible to write unit tests for custom validation attributes [25] and the tests will be generally like those for Fluent Validation.

IV. CONCLUSION

Custom Validation Attribute in C# and Fluent Validation are two approaches for implementing validation logic in .NET applications. There are several key differences between these two approaches.

First off, let us talk about encapsulation. Custom Validation Attributes get slapped right onto your model properties, while Fluent Validation takes a different route by bundling all validation logic into separate classes for each model. This makes Fluent Validation a breeze for keeping things organized and testing them effectively.

Now, readability is a big deal. Fluent Validation comes out on top, offering a more user-friendly and straightforward way to lay down your validation rules. Custom Validation Attributes, on the flip side, can get a bit wordy and might not be as instantly clear.

Reusability is another factor. Custom Validation Attributes win here because you can reuse them across different models and properties. Fluent Validation, on the other hand, demands separate validation classes for each model, which could cramp your style when it comes to widespread use.

Flexibility matters too. Fluent Validation flexes its muscles with a wide range of built-in rules and the ability to craft your own using delegate functions. Custom Validation Attributes are played by a set of predetermined rules.

Lastly, testing is crucial. Fluent Validation makes it easy by clearly dividing rules into separate classes, simplifying targeted testing. Testing Custom Validation Attributes might be a bit trickier since the logic is directly embedded in model properties.

In conclusion, the choice between Custom Validation Attributes and Fluent Validation depends on the specific requirements of the project. Custom Validation Attributes can be a good choice for simple validation logic, which can be reused across multiple models, while Fluent Validation is a better choice for more complex validation logic, and for projects that prioritize maintainability and testability.

References

[1] Cart & Checkout Usability Research, [Online]. Available at: https://baymard.com/research/checkout-usability.

[2] T. Arciuolo, A. Abuzneid, "Simultaneously shop, bag, and checkout (2SBC-Cart): A smart cart for expedited supermarket shopping,"

Proceedings of the 2019 International Conference on Computational Science and Computational Intelligence (CSCI), Las Vegas, NV, USA, 05-07 December 2019, pp. 1162-1167. https://doi.org/10.1109/CSCI49370.2019.00219.
[3] ISO 9000:2005(EN) validation, [Online]. Available at: https://www.iso.org/obp/ui/#iso:std:iso:9000:ed-3:v1:en.
[4] R.-C. Liao, "Customers' perspectives on ISO 9001 QMS auditors' personality traits: A preliminary investigation from Taiwan's certificated companies," *Proceedings of the 2014 International Conference on Service Sciences*, Wuxi, China, 2-23 May 2014, , pp. 215-219. https://doi.org/10.1109/ICSS.2014.40.
[5] The Current State of Checkout UX – 18 Common Pitfalls & Best Practices, [Online]. Available at: https://baymard.com/blog/current-state-of-checkout-ux.
[6] R. Helmi, A. Lee, Md G. Md Johar, A. Jamal, L.F. Sim, "Quantum checkout: An improved smart cashier-less store checkout counter system with object recognition," *Proceedings of the 2021 IEEE 11th IEEE Symposium on Computer Applications & Industrial Electronics (ISCAIE)*, Penang, Malaysia, 03-04 April 2021, pp. 151-156. https://doi.org/10.1109/ISCAIE51753.2021.9431839.
[7] L. Wroblewski, Inline Validation in Web Forms, September 01, 2009, [Online]. Available at: https://alistapart.com/article/inline-validation-in-web-forms/.
[8] S. Yadav, S. Shukla, "Analysis of k-fold cross-validation over hold-out validation on colossal datasets for quality classification," *Proceedings of the 2016 IEEE 6th International Conference on Advanced Computing (IACC)*, Bhimavaram, India, 27-28 February 2016, pp. 78-83. https://doi.org/10.1109/IACC.2016.25.
[9] M. J. Price, *C# 9 and .NET 5 – Modern Cross-Platform Development: Build intelligent apps, websites, and services with Blazor, ASP.NET Core, and Entity Framework Core using Visual Studio Code*, 5th ed., Packt Publishing, 2020; pp. 558–563.
[10] C. Rippon, *ASP.NET Core 5 and React: Full-stack web development using .NET 5, React 17, and TypeScript 4*, Packt Publishing, 2021.
[11] S. Resca, *Hands-On RESTful Web Services with ASP.NET Core 3*, 1st ed, Packt Publishing, 2019, pp. 90–99.
[12] D. Damyanov, Z. Varbanov, S. Varbanova, "An improved approach of using data storage services in ASP.NET Core," *Proceedings of the 2022 International Conference Automatics and Informatics (ICAI)*, Varna, Bulgaria, 06-08 October 2022, pp. 287-291. https://doi.org/10.1109/ICAI55857.2022.9959991.
[13] M. Fowler, Fluent Interface, December 20, 2005, [Online]. Available at: https://martinfowler.com/bliki/FluentInterface.html.
[14] Q. Li, C. Jiao, C. Yang, Z. Zhang, L. Yang, "A feasible method of virtual flow field simulation – Part I: An interface from fluent to RTT," *Proceedings of the 2018 5th International Conference on Information Science and Control Engineering (ICISCE)*, Zhengzhou, China, 20-22 July 2018, pp. 25-29. https://doi.org/10.1109/ICISCE.2018.00015.
[15] M. Fowler, *Domain-Specific Languages*, 1st ed., Addison-Wesley, 2010, pp. 27–87.
[16] T. R. Silva, "Towards a domain-specific language for behaviour-driven development," *Proceedings of the 2023 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 03-06 October 2023, pp. 283-286. https://doi.org/10.1109/VL-HCC57772.2023.00054.
[17] K. Chowdhury, *Mastering Visual Studio 2019*, 2nd ed., Packt Publishing, 2019, pp. 222–244.
[18] H. Taniguchi, K. Nakasho, "Visual Studio code extension and auto-completion for Mizar language," *Proceedings of the 2021 Ninth International Symposium on Computing and Networking (CANDAR)*, Matsue, Japan, 23-26 November 2021, pp. 182-188. https://doi.org/10.1109/CANDAR53791.2021.00033.
[19] FluentValidation, [Online]. Available at: https://docs.fluentvalidation.net/en/latest/index.html.
[20] R. Chatley, S. Uchitel, J. Kramer, J. Magee, "Fluent-based Web animation: exploring goals for requirements validation," *Proceedings of the 27th International Conference on Software Engineering, ICSE*, St. Louis, MO, USA, 15-21 May 2005, pp. 674–675. https://doi.org/10.1145/1062455.1062603.
[21] A. Freeman, *Pro ASP.NET Core 6: Develop Cloud-Ready Web Applications Using MVC, Blazor, and Razor Pages*, 9th ed., Appres, 2022, pp. 847–892. https://doi.org/10.1007/978-1-4842-7957-1_29.
[22] A. Lock, *ASP.NET Core in Action*, 3rd ed., Manning Publications Co., 2023, pp. 250–261.
[23] M. Choudhari, Fluent Validation – Unit Testing the Validators. November 25, 2022, [Online]. Available at: https://thecodeblogger.com/2022/11/25/fluent-validation-unit-testing-the-validators/.
[24] A. Walker, Unit Test an ASP.NET Custom Validator Part 1. May 19, 2021, [Online]. Available at: https://www.linkedin.com/pulse/unit-test-aspnet-custom-validator-part-1-allan-walker/.
[25] R. Osherove, *The Art of Unit Testing*, 3rd ed., Manning Publications Co., 2024, pp. 19–123.

***VOLODYMYR SAMOTYY** received an M.S. in Automation from Lviv Polytechnic National University, Ukraine in 1984, a Ph.D. in 1990, and a D.S. in computers, systems and networks, elements and devices of computers and control systems in 1997. He has been a Professor since 2001. He is currently a Full Professor at the Department of Automation and Information Technologies, Cracow University of Technology, Poland, and the Department of Computerized Automatic Systems at Lviv Polytechnic National University, Ukraine. His research interests include evolutionary models, numerical methods, information security, and digital signal processing. ORCID: 0000-0003-2344-2576*

***ULYANA DZELENDZYAK** received an M.S. in Applied Mathematics from Lviv Polytechnic National University, Ukraine in 1989, and a PhD in 2006. Since 2009 he has been an Associate Professor of the Department of Computerized Automatic Systems at Lviv Polytechnic National University, Ukraine. Her research interests include evolutionary models, numerical methods, and digital signal processing. ORCID: 0000-0003-0529-8582.*

***NAZAR MASHTALER** received an M.S. in Computerized Management Systems and Automation from Lviv Polytechnic National University, Ukraine in 2014, a Ph.D. student since 2022 at the Department of Computerized Automatic Systems at Lviv Polytechnic National University, Ukraine. His research interests include computer engineering, software architecture, and data analysis.*