International Journal of Innovative Research in Computer Science and Technology (IJIRCST)
ISSN (Online): 2347-5552, Volume-13, Issue-1, January 2025
https://doi.org/10.55524/ijircst.2025.13.1.1
Article ID IRP-1576, Pages 1-11
http://www.ijircst.org

Development of an AI-Driven Model for Advancing Software Engineering Practices

Aylin Güzel[1], and Ahmet Egesoy[2]

[1] Research Scholar, Department of Computer Engineering, Ege University, Izmir, Turkiye
[2] Assistant Professor, Department of Computer Engineering, Ege University, Izmir, Turkiye

Correspondence should be addressed to Ahmet Egesoy; ahmet.egesoy@ege.edu.tr

Received 11 November 2024; Revised 26 November 2024; Accepted 11 December 2024

ABSTRACT- This work introduces the *Fuzzy Specification Tree Model (FST)*, a general-purpose framework designed to enhance AI-assisted software engineering. The paper begins by examining the intricate interplay between software engineering and artificial intelligence (AI), emphasizing how AI technologies are reshaping software development methodologies. Building on a foundation of requirements-driven approaches, the study presents a novel adaptation of classical feature modelling to create a versatile, fuzzy logic-based requirements specification model. This model not only facilitates the definition of functionalities for partially completed software but also supports formal methods for project management, version control, and reuse. By employing separate *Fuzzy Specification Trees* for requirements and the current state of a project, developers gain a dynamic perspective on project completeness and can leverage AI assistance to prioritize tasks, ensuring efficient progression toward project completion with minimal effort.

KEYWORDS- AI, Software Engineering, Requirements Management, Fuzzy Logic.

I. INTRODUCTION

As software systems grow increasingly complex, traditional methodologies encounter significant limitations in terms of scalability and adaptability. The term "software crisis" is commonly used as an umbrella phrase to describe the persistent and ill-structured challenges associated with software development processes. This crisis is often characterized by the ongoing difficulty in meeting the ever-growing demands for software. Addressing these demands is the overarching goal of software engineering.

Artificial intelligence (AI) and formal methods are revolutionizing software engineering by addressing many of the challenges posed by the software crisis. AI techniques, such as machine learning and natural language processing, enable smarter automation of tasks like requirements analysis, code generation, testing, and debugging, reducing human error and increasing efficiency. Meanwhile, formal methods bring mathematical rigor to software development, allowing for precise specification, verification, and validation of software systems. Together, these approaches enhance the scalability and reliability of software engineering processes, enabling the creation of more complex and adaptive systems while maintaining high levels of quality.

The *Feature-Oriented Domain Analysis (FODA)* method was created by Dr. Kyo C. Kang and his colleagues at the Software Engineering Institute (SEI) of Carnegie Mellon University in 1990. They documented their work in the report titled *"Feature-Oriented Domain Analysis (FODA) Feasibility Study"*, [1] which introduced a systematic approach to domain analysis by identifying common and variable features within a software domain. It is our view that a feature-driven point of view fostered by the use of AI can provide a solution for the problems of software engineering.

Our proposal employs a fuzzy version of the *FODA tree* that is more in line with project management challenges.

The remainder of this paper is organized as follows: Section II provides an overview of the role of AI in software engineering. Section III discusses the advantages of a requirements-based approach. Section IV presents the proposed model which is an innovative diagram type (and data structure) called *Fuzzy Specification Tree Model*. Finally, Section V concludes the paper.

II. AI IN SOFTWARE ENGINEERING

Artificial intelligence (AI) is a branch of computer science focused on creating intelligent systems capable of acting and communicating in ways that resemble human behavior. AI enables computer systems to explore and perform tasks in domains traditionally driven by human labor. These systems operate with high accuracy, reduce operational costs, and enhance production processes, making them more efficient and manageable. Consequently, it is expected that AI technologies will bring comparable advancements and efficiencies to the field of software engineering.

Determining what qualifies as *intelligence* is inherently challenging, particularly in a domain already regarded as ill-structured, even for humans. Any technique that demonstrably aids in managing the inherent complexity of software systems by offering developers valuable insights or assistance can justifiably be classified as AI.

AI has a wide range of applications in software engineering. The following sub-sections will explore several key areas where AI can be utilized, including cost estimation, fault prediction, test estimation, testing, software maintenance, reuse, quality prediction, source code summarization, and the detection of design and code *bad smells*.

A. Cost Estimation

Software cost estimation is the approximate estimation of the project cost prior to the development process. Models used for cost estimation use mathematical algorithms or parametric equations. There are three main approaches: *Empirical, Heuristic and Analytical* [2].

The Empirical Approach uses data previously collected from a project, as well as some former estimates. These methods rely on historical project data and statistical models to estimate costs. This is a very data-driven approach and requires a significant amount of historical data for calibration. It uses regression or machine learning techniques to find relationships between project attributes (e.g., lines of code, team size) and cost.

The Heuristic Approach relies on expert judgment and rule-of-thumb techniques derived from past experience rather than rigorous data analysis. These techniques are flexible and intuitive when historical data is sparse or inconsistent. However, they rely heavily on expert knowledge and subjective judgment. This approach is useful in novel or poorly understood domains where data is unavailable. It is also adaptable to changing conditions. However, it can also be quite subjective and prone to bias.

The analytical approach relies on mathematical models and formal frameworks to estimate costs based on project-specific parameters. This method typically involves breaking a task into sub-parts and building estimations using principles or formal algorithms, often through deterministic calculations. It incorporates factors such as complexity, size, and team productivity. While this approach provides structured and objective results, it requires detailed upfront information about the project and can be rigid, making it less adaptable to changes during the project.

As a result, the cost estimation process delvers the estimation of the size of the software, the effort required, and the overall cost [2].

B. Fault Prediction

AI-based fault prediction is one of many branches being explored in software engineering, involving diverse techniques and applications across various domains to improve the identification and management of software faults. Classification techniques are widely applied in software fault prediction, aiming to identify faulty software modules using software metrics. *Support Vector Machines* (*SVMs*) are often employed to identify infeasible GUI test cases and to prioritize test cases in system-level testing, particularly in black-box testing scenarios where code access is unavailable. Logistic regression, Random Forest, AdaBoost, and bagging are utilized to optimize testing efforts by predicting change-prone components.

Artificial Neural Networks, Support Vector Machines, and Linear Regression are used for planning and scheduling testing activities. *Genetic Algorithms* are applied for test data generation [3]. The *K-Nearest Neighbor (KNN)* algorithm is used to identify coincidentally correct test cases.

Natural Language Processing (NLP) techniques are employed for multiple purposes, including test case prioritization, predicting manual test case failures, generating test cases, creating test cases from software requirements, automatically documenting unit test cases, and detecting duplicate defect reports.

C. Test Estimation

Test estimation is a technique which approximates how long a task would take to complete. Estimating effort for the test is one of the important tasks in test management. *Test Effort Estimation* is the process of forecasting how much effort is required to develop or maintain a software application. There are four methods for estimating the effort: *Expert Estimation, Top-down Estimation, Bottom-up estimation and Parametric Estimation.*

Expert estimation is a technique in which an expert estimates how much effort a project requires [4]. Expert estimation is a widely used technique in automated software test estimation, relying on the knowledge and experience of domain experts to predict the effort, cost, or time required for testing activities. Unlike algorithmic approaches, expert estimation leverages human intuition and contextual understanding, allowing it to accommodate complex, ambiguous, or project-specific factors that may not be easily quantifiable. *Expert estimation* is not inherently automated, as it fundamentally relies on human judgment and expertise. However, aspects of the process can be augmented or facilitated by automation. While the core predictions are provided by experts, automation can assist in various ways such as data retrieval and analysis, expert collaboration, and bias mitigation.

Top down estimation technique use experience from the past to make estimates for the future. The technique involves deriving high-level estimates based on the overall scope of a project and breaking it into smaller components. The advantage of using top down estimation methods is that they are basically more objective and repeatable than expert estimation. This task presents partial feasibility for automation. Automation in *top-down estimation* is feasible for supporting activities like data retrieval, initial estimate generation, and scenario modeling. However, achieving fully automated top-down estimation is currently unrealistic because of the need for strategic judgment, abstract reasoning, and adaptability.

Bottom-up estimation methods involve analyzing the specific activities required to achieve a project's objectives [4]. Each task is broken down into smaller components, typically requiring less than two weeks of effort, to ensure a manageable level of detail. Individual estimates are then assigned to each component, and these are aggregated to produce the overall project estimate. The primary advantage of bottom-up estimation is its clarity and transparency, as the detailed breakdown makes the estimates more comprehensible and justifiable compared to high-level expert estimates.

Parametric estimation methods utilize algorithms to generate project estimates based on specific inputs, such as the required functionality and expected quality [4]. These algorithms apply predefined computational steps to produce an estimate exclusively from the provided inputs. The key advantage of parametric estimation is its objectivity, offering a systematic and data-driven approach that can deliver highly reliable results. However, this method is typically more complex and time-intensive compared to other estimation techniques, which can be a drawback in scenarios requiring rapid or simplified estimation processes.

D. Testing

Artificial Intelligence plays a pivotal role in modern software testing, improving accuracy and saving time [5].

Techniques such as machine learning, deep learning, and natural language processing are integral to enhancing various aspects of the testing process. AI is increasingly applied to analyze and optimize code during software testing.

Artificial intelligence is particularly well-suited for black box testing due to its ability to analyze patterns, predict outcomes, and adapt to varying inputs without requiring knowledge of the internal system structure.

Black box testing is a method where the system is tested without any prior knowledge of its internal architecture or source code. Testers interact with the system by providing inputs and observing the outputs to evaluate its functionality. This approach helps identify issues related to usability, reliability, and system behavior in response to both expected and unexpected user actions.

By leveraging AI algorithms, *black box testing* can efficiently handle complex testing scenarios, automate repetitive tasks, and uncover hidden issues that might be missed through traditional methods. This compatibility makes *black box testing* a prime candidate for incorporating AI-driven techniques to enhance testing accuracy and reliability.

In *black box testing*, advanced algorithms are utilized to improve efficiency and outcomes. For instance:

- *C4.5*, a decision tree algorithm, is employed to support decision-making in black box testing.
- *Huber Regression*, Support Vector Regression (*SVR*), and multi-layer perceptron are used to predict test coverage in automated testing.
- *Hybrid Genetic Algorithms* (*HGA*) automate GUI testing.
- *K-Means Clustering* is applied to classify test cases, enhancing the effectiveness of regression testing.

These AI-driven techniques demonstrate the transformative impact of AI in advancing the scope and precision of software testing practices.

E. Software Maintenance

AI plays a significant role in the maintenance of software. *Predictive maintenance* is one of the most prominent techniques applied in conjunction with AI-driven development methods. The primary goal of *predictive maintenance* is to anticipate system failures and issue timely warnings, enabling preventive actions to avoid disruptions [6]. This approach helps identify anomalies and potential defects in processes, allowing corrective measures before these issues escalate into critical failures. Machine learning techniques, including supervised and unsupervised learning, are commonly utilized to power predictive maintenance.

Predictive maintenance uses various algorithms to analyze large volumes of operational data, identifying patterns and trends that signal potential failures. This proactive approach not only minimizes downtime but also reduces maintenance costs by addressing issues early.

F. Reuse

Reuse refers to the application of previously developed features, concepts, or objects in new situations, enhancing efficiency and innovation. *Reusability*, on the other hand, describes the ability of these components to be effectively adapted for new applications. In the context of software development, *reuse* significantly improves productivity, reduces time and costs, enhances reliability, and simplifies maintenance [7].

Various data mining techniques, such as knowledge discovery, classification, and clustering, play a crucial role in the domain of software reuse. *Classification* methods are employed to identify reusable software components, while *clustering algorithms* predict the reusability of software elements by grouping similar components. Additionally, methods like *neural networks* and *classification* algorithms are applied to further refine the identification of reusable components.

Reusability not only optimizes development time and costs but also improves the reliability and overall quality of software systems. In *classification-based* approaches, software components are categorized using two key methodologies: *Coverage-based Classification* and *Proximity-based Classification*.

In the *Coverage-based Classification* technique, operations are evaluated for their degree of generality and adaptability. In the *Proximity-based Classification* technique, the similarity between operations is measured by a proximity value (or similarity distance). A smaller proximity value indicates greater similarity between processes, suggesting a higher potential for reuse.

By leveraging these classification strategies, organizations can systematically identify and integrate reusable components, leading to more efficient and reliable software development practices.

G. Quality Prediction

Software quality prediction involves identifying software modules that may present potential quality issues, helping to ensure overall system reliability and stability. Techniques such as *Bayesian belief networks*, *neural networks*, *fuzzy logic*, *support vector machines*, *expectation-maximization algorithms*, and *case-based reasoning* are widely used in software quality estimation [8]. Software quality is defined by compliance with requirements and the absence of defects, with reliability and stability being critical criteria. Accurately predicting these attributes simplifies the process of assessing software quality.

Neural networks are a commonly used method for software quality prediction. In this approach, a three-layer feed-forward neural network is trained using historical data. Once trained, clustering genetic algorithms are applied to extract intelligible rules from the network. These rule sets are then used to identify error-prone software modules, enabling the classification of modules as *faulty* or *non-faulty*.

Fuzzy logic offers another method for software quality estimation. This technique either fuzzifies an existing rule-based estimation model or creates a new fuzzy model using software metrics. The Sugeno inferencing method is frequently employed to predict the number of faults in the training data, providing an adaptable and granular approach to quality prediction.

By leveraging these advanced techniques, software quality prediction helps developers proactively address potential defects, ensuring more stable, reliable, and high-quality software systems. Each approach provides unique advantages, from the interpretability of fuzzy logic models to the powerful pattern recognition capabilities of neural networks, making them valuable tools for improving software quality.

H. Detecting Bad Smells in Design and Coding

In software engineering, *code smells* refer to patterns in the code that, while not necessarily incorrect, indicate deeper problems in the software's structure, such as poor design or maintainability issues. These issues if left unaddressed, can lead to more significant problems.

Martin Fowler introduced the concept of *bad smells* in software development, referring to indicators of potential issues within the codebase [9]. A *bad smell* arises when developers make incorrect analyses of system requirements, take poor decisions regarding system design, or disregard fundamental principles of software development. Additionally, it may occur when developers write overly complex, hard-to-read, or poorly comprehensible code to address immediate needs without considering long-term maintainability.

In essence, *bad smells* in code result from errors made during the software development process whether in analysis, decision-making, or implementation. These smells act as warning signs that a part of the code might require refactoring or further scrutiny to avoid deeper issues.

Common scenarios where bad smells appear include:

- Ignoring fundamental software development principles,
- Faulty or incomplete analysis,
- Poor decision-making,
- Writing overly complex or unintelligible code,
- Incorrectly integrating new modules into the system,
- Misjudging the requirements or goals of the system.

By recognizing and addressing these bad smells early, developers can improve the code's readability, maintainability, and overall quality, ensuring a more robust and scalable software product. Bad smells are indications of potential problems in the system. Also, design problems in the code are seen as a bad smell.

Common bad smells in code include:

Duplicated Code: Identical or highly similar code structures appearing in multiple locations. Duplicated code should be consolidated to avoid redundancy and simplify maintenance.

Long Methods: Methods that attempt to perform too many tasks, making them harder to understand and reducing functionality. The **Extract Method** approach can be applied to break these methods into smaller, more focused ones.

God Class: A class that contains an excessive amount of information and responsibilities, leading to high complexity and redundancy. Refactoring techniques such as *Extract Class* or *Extract Subclass* can redistribute responsibilities and streamline the class.

Long Parameter Lists: Methods or functions with unnecessarily long parameter lists reduce clarity and usability. These should be shortened by using objects or grouping related parameters, improving code readability and simplicity.

Switch Statements: Excessive use of switch statements can make code harder to maintain and extend. Alternative approaches, such as polymorphism or strategy patterns, should be employed to reduce their frequency.

Comments: While comments can be helpful for documentation, they are often used to obscure bad code practices instead of addressing underlying issues. Over-reliance on comments to explain poorly written code is itself considered a bad smell.

Lazy Classes: Classes that do little or no meaningful work should be removed. Eliminating lazy classes reduces code size and improves clarity.

By addressing these bad smells through refactoring, developers can create cleaner, more efficient, and easier-to-maintain codebases. This process not only enhances the software's performance and quality but also ensures better scalability and adaptability for future requirements.

Bad smells in code can be detected either manually or through automated tools, which enhance the process by leveraging *metric-based* and *visualization-assisted* analysis. These tools assist developers in identifying, visualizing, and analyzing various code smells, making the detection process more efficient and systematic [10], [11], [12]. *DECOR* [13] is a tool commonly used for detecting spaghetti code, a type of code smell characterized by poor readability and tangled structure. Spaghetti code arises when the flow of the code becomes overly complex, making it difficult to follow or maintain. *DECOR* helps reduce the overall costs of development and maintenance by providing effective detection of these issues.

JDeodorant [14] is another tool designed to automatically detect *Type-Checking code smells* in Java source code. It identifies bad smells and suggests appropriate refactoring techniques.

Bad smells in code also help identify design problems within software systems, enabling developers to address underlying architectural issues. Tools like *JSpIRIT* [15] allow developers to define new detection rules for code anomalies and prioritize the identified smells. Code anomalies indicate design flaws in the source code and should be eliminated to enhance overall system quality. The detection process begins with scanning the code, followed by the automated identification of smells using predefined rules. Developers can then prioritize the identified smells based on customizable criteria, ensuring that the most critical issues are addressed first.

JSNose [16] is a JavaScript-specific code smell detection technique that uses a metric-based approach combining static and dynamic analysis to identify smells in client-side code. JavaScript, being a highly flexible scripting language for interactive web applications, can exhibit various code anomalies such as *lazy objects, long methods/functions, closure smells (nested functions), and excessive use of global variables*. By detecting these anomalies, *JSNose* improves the maintainability and quality of JavaScript codebases.

InCode [17], implemented within the Eclipse environment, is another tool for code smell detection. It identifies four common bad smells: *Feature Envy, God Class, Duplicate Code, and Data Class*. *InCode* uses a metrics-based approach to detect anomalies, providing actionable insights to improve code structure.

Machine learning algorithms are frequently employed to analyze large codebases and identify patterns associated with specific types of code smells, such as *duplicated code, large classes, or excessive coupling.* These tools are trained on datasets containing examples of code with known smells and their corresponding refactoring solutions. By learning

from these examples, AI models can predict potential smells in new code and even suggest appropriate refactoring actions. For example, *neural networks and decision tree algorithms* are often used to detect complex patterns that traditional static analysis tools might overlook. AI tools have the ability to continuously learn and adapt to new code smells as they emerge, providing up-to-date recommendations aligned with the latest coding standards and best practices. These adaptive capabilities ensure that AI-driven solutions remain effective in evolving software development environments. Additionally, AI tools can prioritize which code smells to address based on their impact on the software's overall quality and performance. This prioritization enables developers to focus on resolving the most critical issues first, optimizing both time and resources in the refactoring process.

III. REQUIREMENTS BASED APPROACHES

Requirements form the foundation for the entire development process, guiding design, implementation, and validation. Accurate requirements ensure that the software aligns with the stakeholders' needs and business objectives, minimizing the risk of delivering a product that fails to solve the intended problems. Poorly defined or misunderstood requirements often lead to uncontrolled expansion of a project's scope (scope creep), increased costs, and extended timelines, as developers may need to revisit and revise their work. Clear and precise requirements help prevent miscommunication between stakeholders and the development team, by providing a shared understanding of project goals. By addressing ambiguities and prioritizing critical functionalities early, teams can manage risks and allocate resources more effectively.

A. MoSCoW Method

The MoSCoW method is a widely used prioritization technique in software engineering that categorizes requirements into four groups: *Must Have*, *Should Have*, *Could Have*, and *Won't Have*. This method helps stakeholders clearly distinguish between critical and less critical functionalities, with the aim of ensuring that the most essential requirements are delivered first. For example, "*Must Have*" requirements are vital for the system's operation, while "Could Have" requirements are only implemented if time and resources permit. The method is particularly effective in agile environments where the prioritization needs to be dynamic and responsive to project constraints [18].

B. Kano Model

The Kano Model is a user-centered approach to prioritizing requirements based on their impact on customer satisfaction. It categorizes features into Basic Needs, Performance Needs, and Excitement Needs, helping teams understand which functionalities are critical to meet user expectations and which might delight users unexpectedly. For instance, while a website's login functionality is a basic need, a personalized greeting might be an excitement need. This model supports decision-making in product development by aligning features with customer value [19].

C. Weighted Scoring

Weighted scoring is a systematic approach to prioritizing requirements by assigning scores to various criteria such as business value, implementation cost, technical risk, and stakeholder urgency. Each requirement's total score determines its priority, ensuring that decisions are data-driven. For example, a requirement with high business value and low cost might score higher than one with medium value and high risk. This approach supports transparency and alignment among stakeholders [20].

D. Value-Based Prioritization

Value-based prioritization focuses on delivering the maximum value to customers and stakeholders by ranking requirements based on their potential impact, such as *return on investment* (*ROI*), customer satisfaction, and market differentiation. This approach emphasizes delivering high-value features early in the development process, allowing for quicker feedback and market adaptation. For instance, implementing features that significantly increase user engagement might be prioritized over those with marginal gains [21].

E. FODA Approach

Feature-Oriented Domain Analysis (FODA) is a methodology used in software engineering to analyze and model the common and variable features of a domain. It is particularly useful in domains where families of related software systems share a core set of functionalities but also exhibit variability in certain features. FODA aims to improve the development and maintenance of software by enabling systematic reuse and customization.

Introduced by Kang et al. in the 1990s [1],[22], FODA emphasizes the identification of features, which are end-user-visible characteristics of a system. The methodology consists of three main stages:

a) **Context Analysis:** Understanding the problem domain and its boundaries.
b) **Domain Modeling:** Capturing the commonalities and variabilities among the systems in the domain.
c) **Feature Modeling:** Representing these commonalities and variabilities explicitly through feature models, often depicted using FODA Trees.

A FODA Tree, or Feature Model, is a hierarchical structure used to represent the features of a domain. It organizes features into:

- **Mandatory Features:** Features that are always included in every product.
- **Optional Features:** Features that may or may not be included, depending on the system configuration.
- **Alternative Features:** A set of features where only one can be selected.
- **Or Features:** A set of features where one or more can be selected.

The tree is rooted in an *overall domain feature*, which represents the overarching functionality of the domain. From this root, branches represent relationships between features, using specific notations to show dependencies and constraints. A filled circle indicates a mandatory feature. An *empty circle* signifies an *optional feature*. An arc connecting sibling nodes indicates a group of alternative features where a *filled arc* is a *regular OR* connective (*one*

or more) and an *empty arc* is an *EXCLUSIVE OR (XOR)*connective (*only one*).
A typical example of a *FODA* tree, as described by [23], is shown in Figure 1, illustrating the decomposition of a Mobile Phone design task into sub-tasks: *Calls*, *GPS*, *Screen*, and *Media*. The *Calls* and *Screen* sub-tasks represent mandatory features, whereas *GPS* and *Media* are classified as optional features. The Screen requirement is further decomposed using an *XOR* connective, which specifies three mutually exclusive implementation options: A *Basic Screen*, a *Color Screen*, or a *High-Resolution Screen*. In contrast, the *Media* requirement can be satisfied by incorporating either or both of the following features: A *Camera* and an *MP3* player.

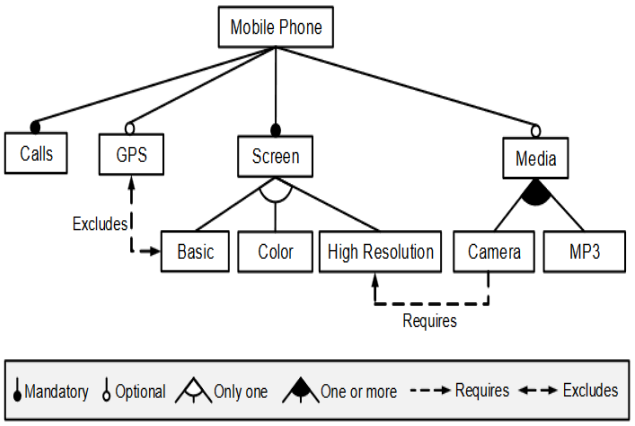

Figure 1: FODA Tree example [23]

FODA Tree approach provides many benefits. Firstly, with its graphical representation it provides a clear and intuitive understanding of the domain's feature landscape. Secondly it facilitates tailoring software systems to meet specific requirements and lastly it enables the structured reuse of features across systems.
FODA and its associated trees are widely applied in *Software Product Line Engineering (SPLE)*, where multiple software products are derived from a shared codebase. The methodology's emphasis on features and variability makes it ideal for managing complexity in evolving systems.

IV. FUZZY SPECIFICATION TREE

A. General Design

The *Fuzzy Specification Tree Model (FST)* that we propose differs from the conventional *FODA* Tree approach for several reasons. First, it integrates both discrete and fuzzy elements, which can lead to more complex models. Second, it combines structural and logical elements within the same framework. While these design choices have the potential to create an overly complicated model, the complexity was carefully managed by restricting combinations to those that are meaningful within the domain and by designing a user-friendly set of linguistic elements.
The *Fuzzy Specification Tree Model* is aimed to perform some basic functions through automation:

- Facilitate interface and implementation of configurations.
- Facilitate subtyping relations between configurations.
- Check how functional each feature is.
- Make it possible to decide on the optimum module to be developed as the next task.

Our approach utilizes two trees, created in the same format but serving distinct purposes. One tree represents the requirements, while the other represents the implementation, which corresponds to the current structural model of the project. The *requirements tree* consists of logical nodes that indicate the degree to which specific features of the software are functional, fully leveraging fuzzy logic principles. In contrast, the *task tree* exclusively comprises aggregation relations, representing the completion levels of individual tasks. The *task tree* tracks the progress of tasks and provides fuzzy logic values that quantify task accomplishment. These values are directly linked to the leaves of the requirements tree, serving as inputs to calculate the functionality of the software's features. A bottom-up algorithm is then employed to propagate these values through the *requirements tree*, ultimately determining the functionality of higher-level features or maybe the whole project.

B. Sequence Operator

This operator combines its operands in a sequential relationship from left to right. In this relationship, the operand on the left has precedence over the operand on the right. All values are checked from left to right. If all required operands are true (1), true is returned. If more than one required operand is not true, false (0) is returned. If only one required operand has a value other than true (1), and it is the rightmost required operand, it returns the value of that operand.
This operator is used when a requirement must be fully fulfilled for another requirement to be meaningful. (That is, if the second will not have any meaning unless the first is fully fulfilled.) In this case, a partial success will only be allowed at the very last step.
Example: Let's say the task is "First go to *New York*, then find *Central Park*" In this case, if I couldn't go to *New York*, there would be no point in trying to find the *Central Park* (or claiming that I found it). Let's say I went to *New York*, but I got stuck around *Harlem*. In that case, for example, a partial success can be mentioned, such as *0.7* which is a value between *0* and *1*. However if you are not exactly in *New York* (even if you are in a nearby city) the success rate of reaching the Central Park would be 0.

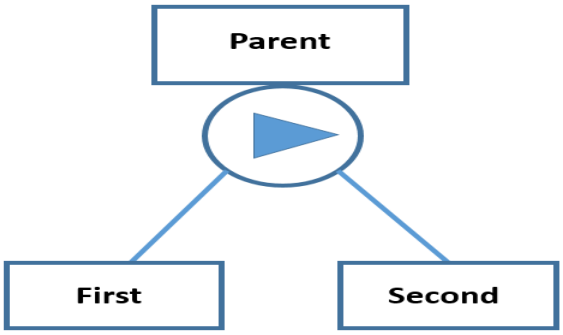

Figure 2: Sequence notation

We use this connective when one operand has a clear priority over the other and must be completed before the other even has a chance to start. In Figure 2 the graphical syntax of the *Sequence operator* can be seen. It should be noted that the operands of the sequence operator form an

ordered list and there is no *commutative property*. The operands are ordered from left to right where left has the priority.

Table 1: Sequence Operation Value Table

Sequence	**0.00**	**0.25**	**0.50**	**0.75**	**1.00**
0.00	0.00	0.00	0.00	0.00	0.00
0.25	0.00	0.00	0.00	0.00	0.00
0.50	0.00	0.00	0.00	0.00	0.00
0.75	0.00	0.00	0.00	0.00	0.00
1.00	0.00	0.25	0.50	0.75	1.00

Table 1 displays the results of the sequence operation for various sample fuzzy values. Each row represents a specific value of the first operand, while each column corresponds to a specific value of the second operand. Five values, spaced at intervals of 0.25, are selected to represent the entire range of fuzzy logic values between *0 (false)* and *1 (true).*

It is evident that the distribution of values in the table is not particularly noteworthy. The result takes on fuzzy values only when the first operand equals 1.

The *Sequence operator* was inspired by the dependency relation, software reuse, inheritance, versioning and project management with tasks arranged in a temporal order.

C. Features Operator

Features operator is a connective that combines adherence to multiple independent features of a parent specification. It is assumed that there is no overlap between the features. In other words, they are semantically orthogonal to each other. In this case, the *Product T-Norm* function is used between the operators in order to compute the result:

$$Parent = Op1 \times Op2 \quad (1)$$

The order is not important in this operation. All *mandatory operands* are multiplied and *optional operands* are ignored.

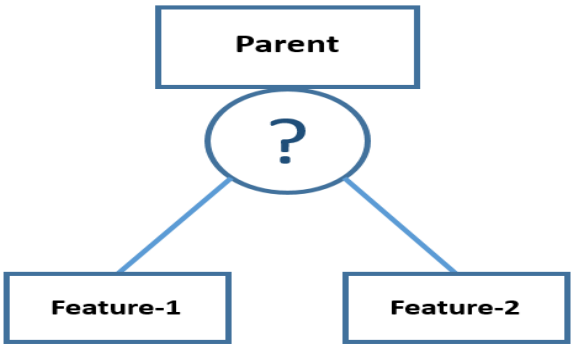

Figure 3: Features notation

Figure 3 illustrates the graphical syntax of the Features operator, represented by a circled question mark connecting two operands. Features typically refer to the capabilities or functionalities of a project that collaborate to achieve a common objective without compromising each other's performance. A classic example is the engine and transmission of a car where the overall efficiency of the vehicle can be approximated as the product of the efficiencies of these two components. Similarly, in software, features such as speed efficiency and space efficiency can be analyzed in this manner. The connection between other attributes, such as portability, generality, usability, and robustness, also fall under this categorization.

Table 2: Features Operation Value Table

Features	0.00	0.25	0.50	0.75	1.00
0.00	0.00	0.00	0.00	0.00	0.00
0.25	0.00	0.06	0.12	0.19	0.25
0.50	0.00	0.12	0.25	0.38	0.50
0.75	0.00	0.19	0.38	0.56	0.75
1.00	0.00	0.25	0.50	0.75	1.00

Table 2 presents the typical results of the Features Operation. Since the *Product T-Norm* involves the multiplication of fractional values, the resulting value can decrease rapidly when a large number of operands are less than one. It is important for developers to distinguish between Features and Aggregation relationships and to avoid incorrectly marking an aggregation as a feature.

D. Aggregation Operator

The *Aggregation operator* is employed when the relationship between the operands and the parent node represents a *Part-Whole* relationship. This operator is used to model a requirement that is composed of two or more distinct sub-requirements where partial success in one can pay for the failure in the other. In such cases, a *weighted average* operation is performed based on the relative weights of the components within the whole. Only mandatory nodes are considered in this calculation. The formula is:

$$Whole = (P1*w1 + P2*w2)/(w1+w2) \quad (2)$$

where P1 and P2 are the fuzzy operands; and w1 and w2 are relative weights of the two operands.

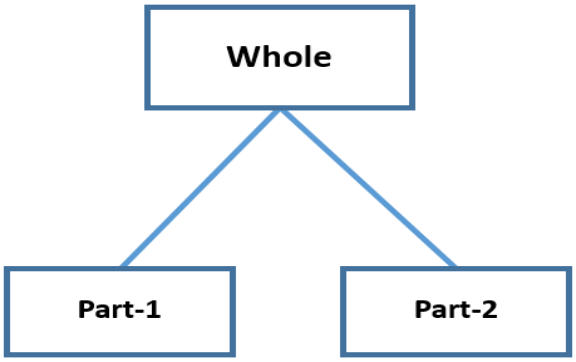

Figure 4: Aggregation notation

Figure 4 shows the syntax of the Aggregation operator. There is a similar construct in the classical *FODA* diagrams that can be interpreted as a conjunction as well as an aggregation depending on the point of view.

In requirements engineering, this connector is applied when the whole system performs two or more independent tasks, each contributing to the whole in proportion to its assigned weight. Importantly, there should be no interaction between the individual parts. In other words, these functions operate independently, performing distinct tasks that together fulfill the broader objective.

The concept of a *task* is more appropriate for the operands of *Aggregation* than the concept of a *feature*. For instance, an application can be imagined as capable of performing

multiple independent functions. Each item in the application's main menu is aggregated to contribute to the overall success of the system.

Table 3: Aggregation Operation Value Table

Aggregation	0.00	0.25	0.50	0.75	1.00
0.00	0.00	0.12	0.25	0.38	0.50
0.25	0.12	0.25	0.38	0.50	0.62
0.50	0.25	0.38	0.50	0.62	0.75
0.75	0.38	0.50	0.62	0.75	0.88
1.00	0.50	0.62	0.75	0.88	1.00

Table 3 presents the results of the *Aggregation operation*. Under normal operating conditions, weighting plays a critical role in the calculation. In our approach, weights are defined as properties of the nodes. However, to illustrate the operation of the operator, equal weights are assumed in the table.

E. Options Operator

The final operator in the *Fuzzy Specification Tree Model* is the Options operator. This operator is used when a problem has multiple potential solutions. For example, it can be applied in scenarios involving a *Java interface* with multiple implementations. If any one of the implementations is sufficient to solve the problem, this operator is employed to indicate that the best available solution determines the actual performance presented for the problem at hand. The performance of the solution is assessed using the *Fuzzy Gödel OR* function, which is defined as:

$$Problem = Max(S1, S2) \quad (3)$$

where S1 and S2 are the two operands.

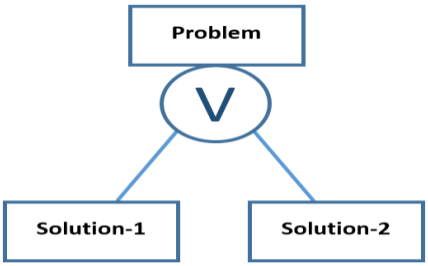

Figure 5: Options notation

As can be seen in Figure 5, the *options* operation is represented with a logical OR sign in a circle.

Table 4: Options Operation Value Table

Options	0.00	0.25	0.50	0.75	1.00
0.00	0.00	0.25	0.50	0.75	1.00
0.25	0.25	0.25	0.50	0.75	1.00
0.50	0.50	0.50	0.50	0.75	1.00
0.75	0.75	0.75	0.75	0.75	1.00
1.00	1.00	1.00	1.00	1.00	1.00

Table 4 *presents* the typical results of the Options operation. The Gödel OR function that shapes the results is quite generous in terms of the truth values.

This operation corresponds to both *OR* and *XOR* connectives in classical *FODA* trees. Notably, the *XOR* operation in a *FODA* tree does not represent a special case of a fuzzy *XOR*. For instance, in the example shown in Figure 1, the *Color* and *High-Resolution* options are not strictly mutually exclusive. Additionally, it remains debatable whether the *Basic* Screen option is extended by these two options or stands as a separate entity.

This presents an interesting example of the two-sided semantic nature of operations. A well-known approach to understanding the semantics of programming languages is axiomatic semantics. In this paradigm, every operation is characterized by two aspects: a *pre-condition* and a *post-condition*. Since operations align better with the functional programming paradigm (where side effects are absent) it may be more appropriate to refer to these aspects as *input* and *output*. When a description is provided as the semantics of an operation, it is crucial to specify whether it pertains to the input or the output of the operation.

Logical constraints on the input form a logical statement assumed to be true for the operation to be valid (or meaningful). Conversely, describing the output pertains to the nature of the transformation associated with the operation itself.

When discussing the exclusivity of an *OR* operation, it is essential to distinguish between two types of exclusivity: (1) the exclusivity imposed by the domain as a given fact, and (2) the exclusivity that may be introduced by the resultant requirement. The latter is compatible with a fuzzy *XOR*, whereas the former is not. In the classical interpretation of a fuzzy *XOR* we do not actually want the two operands to be true together. The resultant value will always be smaller than the two operands. For instance, the *XOR* correspondent of the product T-Norm is: $x+y-2 \cdot x \cdot y$ where x and y are the two operands which is always smaller than both x and y, provided that x and y are between *0* and *1* (as all fuzzy logic truth values).

This implies that when one of the two requirements (x or y) is met, it is undesirable for the other to be fulfilled as well. Such scenarios are extremely rare in software engineering, to the extent that it is challenging even to conceive of a practical use case. Consequently, we conclude that the *XOR* relation found in *FODA* trees does not represent a true *XOR* operation and should not be generalized to its fuzzy counterpart. In fact, the true fuzzy *XOR* is likely unnecessary in practice. The fuzzy generalization of the *XOR* operation, as used in classical *FODA* trees, is effectively a fuzzy *OR* operation implemented using the *Gödel co-norm*. The *Gödel co-norm*, which is the *maximum* function, is particularly suitable for cases where multiple solutions are provided for a single problem. When x and y both address the same problem, the resultant degree of success is naturally determined by the maximum value between the two.

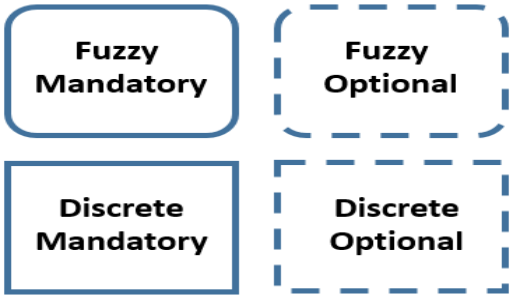

Figure 6: Node types

The *Fuzzy* Specification *Tree* notation encapsulates information about both the nodes and the branching structure. As shown in Figure 6, the syntax includes four types of nodes. Primarily, nodes are distinguished by whether they are *fuzzy* or *discrete*. Discrete nodes can only assume true or false values. However, if they contain internal details represented by child nodes in the tree, their calculated value may be a fuzzy number derived from the leaves. In such cases, a property called the *threshold* is used to determine whether the node resolves to *true* or *false*. Those that exceed the threshold become *true* and those that do not become *false*.

The notation used for four different types of nodes can be seen in Figure 6. *Discrete* nodes are depicted as rectangles with sharp corners and *fuzzy* ones are depicted as rectangles with rounded corners. *Optional* nodes are drawn with dashed lines while *mandatory* nodes are drawn with solid lines.

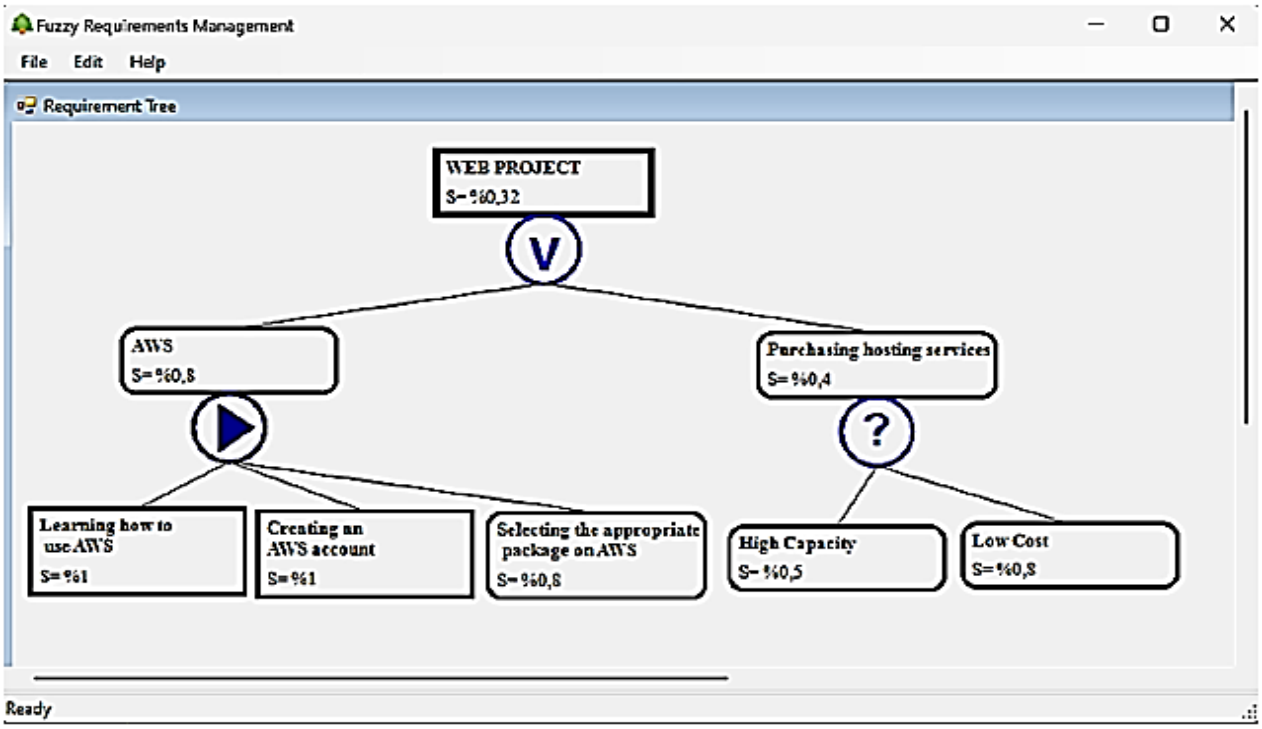

Figure 7: FST Model Tool Screenshot

The second criterion pertains to whether a node is mandatory or optional. Unlike classical *FODA* trees, where the optional or mandatory nature of contributions is treated as a parameter of conjunction (or aggregation) operands, our approach considers these attributes as intrinsic properties of the nodes themselves. This allows them to be associated with any operation in our framework.

When a node is optional, its completeness value does not contribute as an operand when calculating the completeness value of its parent. However, it can still serve as the target of a direct query from the user. Optional nodes may also include inner details in the form of sub-branches within the Specification Tree and can delegate the computation of their completeness value when explicitly queried by a user.

For instance, the user interface of an application might consist of a mandatory graphical user interface and an optional web interface. While the web interface is not essential, a user might still wish to evaluate its current level of completeness. In such cases, the optional node processes the query in the same way as any other node.

Figure 7 presents a screenshot of our *FST* tool, a graphical editor designed for creating fuzzy specification trees for both requirements and associated tasks. The tool allows users to link requirements with tasks, enabling the integration of fuzzy completeness data into the calculation of the current states of the requirements. It features dialogs for configuring the properties of nodes and connectives, along with standard graphical manipulation functionalities. Additionally, the tool can generate on-demand queries to assess the current functionality of any specified requirement. In the figure, the requirement tree for a web project is displayed on the drawing board, illustrating the use of three different connectives with relevant content assigned to their operands.

V. CONCLUSION

In this study, we introduced the *Fuzzy Specification Tree Model (FST)*, a novel approach designed to enhance the requirements specification process in software engineering. By extending the classical *FODA* framework with fuzzy logic principles, the *FST* model provides a more adaptable and precise method for analyzing requirements, accommodating the inherent uncertainties of real-world software development. This approach also supports efficient project management and task prioritization through operations such as *Sequence*, *Features*, *Aggregation*, and *Options*, which are tailored to address practical engineering challenges rather than adhering to a rigid logical structure.

To demonstrate the applicability of our model, we developed a supporting tool that implements the *FST* framework, enabling users to visualize, configure, and evaluate their requirements dynamically. This tool not only streamlines the modeling process but also facilitates better

decision-making by offering a clear representation of project progress and feature completeness.

The *FST* model and its accompanying tool provide a robust foundation for addressing the complexities of modern software engineering. Future work will explore further refinements to the model and its broader application in domains requiring adaptive and scalable requirements management solutions. It is also important to associate the requirements with costs in order to enable smart decision taking in full or semi-automatic usage.

CONFLICTS OF INTEREST

The authors declare that they have no conflicts of interest.

REFERENCES

[1] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, Feature-Oriented Domain Analysis (FODA) Feasibility Study, Pittsburgh, PA, USA: SEI, Carnegie Mellon University, Tech. Rep. CMU/SEI-90-TR-021, Nov. 1990. Available from: https://www.researchgate.net/publication/215588323_Feature-Oriented_Domain_Analysis_FODA_feasibility_study

[2] P. Pandey, "Analysis of the techniques for software cost estimation," in 2012 Third International Conference on Advanced Computing & Communication Technologies, pp. 16–19, 2012, doi: 10.1109/ACCT.2013.13. Available from: https://doi.org/10.1109/ACCT.2013.13

[3] A. Haveri and Y. Suresh, "Software fault prediction using artificial intelligence techniques," in 2nd IEEE International Conference on Computational Systems and Information Technology for Sustainable Solutions (CSITSS), pp. 54–60, 2017. Available from: https://doi.org/10.1109/CSITSS.2017.8447615

[4] M. Lopez, "Machine learning techniques for software testing effort prediction," Software Quality Journal, Springer, 2020, Available from: https://doi.org/10.1016/j.jer.2023.100150

[5] H. Hourani, A. Hammad, and M. Lafi, "The impact of artificial intelligence on software testing," in 2019 IEEE Jordan International Joint Conference on Electrical Engineering and Information Technology (JEEIT), pp. 565–570, 2019. Available from: https://doi.org/10.1109/JEEIT.2019.8717439

[6] R. Jindal, R. Malhotra, and A. Jain, "Predicting software maintenance effort using neural networks," in 2015 IEEE International Conference, 2015. Available from: https://doi.org/10.1109/ICRITO.2015.7359258

[7] D. Wangoo, "Artificial intelligence techniques in software engineering for automated software reuse and design," in 2018 4th International Conference on Computing Communication and Automation (ICCCA), pp. 1–4, 2018. Available from: https://doi.org/10.1109/CCAA.2018.8777584

[8] S. Pattnaik and B. Pattanayak, "A survey on machine learning techniques used for software quality prediction," International Journal of Reasoning-based Intelligent Systems, vol. 8, no. 1/2, pp. 3–14, 2016. Available from: http://dx.doi.org/10.1504/IJRIS.2016.080058

[9] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, Refactoring: Improving the Design of Existing Code, Addison-Wesley Professional, 2002. Available from: https://silab.fon.bg.ac.rs/wp-content/uploads/2016/10/Refactoring-Improving-the-Design-of-Existing-Code-Addison-Wesley-Professional-1999.pdf

[10] E. Fenandes, J. Oliveira, G. Vale, and T. Paiva, "A review-based comparative study of bad smell detection tools," in EASE '16: Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering, June 1–3, 2016, doi: http://dx.doi.org/10.1145/2915970.2915984.

[11] M. Alenezi and M. Zarour, "An empirical study of bad smells during software evolution using Designite tool," i-manager's Journal on Software Engineering, vol. 12, no. 4, pp. 1–10, Apr.–Jun. 2018. Available from: http://dx.doi.org/10.26634/jse.12.4.14958

[12] P. Danphitsanuphan and T. Suwantada, "Code smell detecting tool and code smell-structure bug relationship," in 2012 Spring Congress on Engineering and Technology (S-CET), pp. 1–5, May 2012, doi: 10.1109/SCET.2012.6342082. Available from: https://doi.org/10.1109/SCET.2012.6342082

[13] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. Le Meur, "DECOR: A method for the specification and detection of code and design smells," IEEE Transactions on Software Engineering, vol. 36, no. 1, pp. 20–36, Jan.–Feb. 2010, Available from: https://doi.org/10.1109/TSE.2009.50

[14] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "JDeodorant: Identification and removal of type-checking bad smells," in Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR), Athens, Greece, Apr. 2008, pp. 329–331, Available from: https://doi.org/10.1109/CSMR.2008.4493342

[15] S. Vidal, H. Vazquez, and A. Diaz Pace, "JSpIRIT: A flexible tool for the analysis of code smells," in 2015 34th International Conference of the Chilean Computer Science Society (SCCC), Santiago, Chile, Nov. 2015, Available from: http://dx.doi.org/10.1109/SCCC.2015.7416572

[16] A. Fard and A. Mesbah, "JSNOSE: Detecting JavaScript code smells," in Proceedings of the 13th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM), 2013, pp. 116–125, Available from: https://doi.org/10.1109/SCAM.2013.6648192

[17] M. Ilyas and M. Hummayun, "A comparative study on code smell detection tools," International Journal of Advanced Science and Technology, vol. 60, pp. 25–32, 2013, Available from: http://dx.doi.org/10.14257/ijast.2013.60.03

[18] E. Miranda, "Moscow rules: A quantitative exposé," in Agile Planning and Delivery, Springer, 2022, pp. 11–25, doi: 10.1007/978-3-030-87515-1_2. Available from: https://www.researchgate.net/publication/356836488_MoSCoW_Rules_A_quantitative_expose_Accepted_for_presentation_at_XP2022

[19] J. Hartmann and M. Lebherz, Literature Review of the Kano Model Development Over Time (1984–2016), Bachelor's Thesis, Halmstad University, Sweden, 2016. Available from: http://dx.doi.org/10.1108/02656711311299863

[20] J. del Sagrado and I. M. del Águila, "Assisted requirements selection by clustering," arXiv preprint arXiv:2401.12634, 2024. Available from: https://arxiv.org/pdf/2401.12634

[21] D. Vavpotič, M. Robnik-Šikonja, and T. Hovelja, "Exploring the relations between net benefits of IT projects and CIOs' perception of quality of software development disciplines," arXiv preprint arXiv:1908.04070, 2019. Available from: https://arxiv.org/pdf/1908.04070

[22] K. Czarnecki and U. W. Eisenecker, "Synthesizing objects," IEEE Software, vol. 18, no. 5, pp. 70–80, Sep.–Oct. 2001, Available from: https://doi.org/10.1016/j.cad.2020.102932

[23] A. Durán, D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés, "FLAME: A Formal Framework for the Automated Analysis of Software Product Lines Validated by Automated Specification Testing," Software and Systems Modeling, vol. 16, no. 4, pp. 1219–1246, Oct. 2017, Available from: https://link.springer.com/article/10.1007/s10270-015-0503-z

ABOUT THE AUTHORS

Aylin Güzel (M.Sc. in Computer Engineering) is a Ph.D. student in Computer Engineering Department of Ege University, Izmir, Turkey. Research interests include software engineering, object oriented programming, fuzzy logic and design patterns.

Ahmet Egesoy (PhD in computer engineering) is an instructor and Assistant Professor in Computer Engineering Department of Ege University Izmir, Turkey. Research interests include object-oriented programming, design patterns, model-driven software development, artificial intelligence, programming languages, programming paradigms, philosophy of the language, semiotics and knowledge representation.