# A Taxonomy of Software Debugging Process

Viktor Shynkarenko[1,*,†] and Oleksandr Zhevaho[1,†]

[1] *Ukrainian State University of Science and Technologies, Lazaryana str. 2, Dnipro, 49010, Ukraine*

**Abstract**

Previous research has explored individual facets of debugging, but the field lacks a comprehensive taxonomy that systematically categorizes the debugging process and factors influencing it. This paper aims to fill that gap by proposing a taxonomy of software debugging, which classifies the process into six key dimensions: the objects of debugging ("What"), employed tools ("Which"), the applied methods ("How"), the human factors involved ("Who"), the temporal aspects ("When"), and the debugging environment ("Where"). The proposed taxonomy highlights the complexity and diversity of debugging practices. By analyzing debugging techniques, tools, and environmental influences, we present a structured framework that provides insights into the nature of debugging activities. Organizing these activities into structured categories facilitates a deeper understanding of software debugging, facilitates the identification of research gaps and limitations, and provides a foundation for future studies. Additionally, the taxonomy serves as a structured framework for teaching debugging concepts and techniques. Educators can use it to ensure comprehensive coverage of debugging approaches in software engineering curricula. To validate the taxonomy, we used data from an experiment conducted with students during a debugging olympiad. Log files containing detailed information about the debugging processes were collected. Each debugging session from the log files was mapped to the taxonomy's dimensions and categories to identify key characteristics and behaviors. We evaluated how the experimental data aligned with each dimension of the taxonomy, confirming that it encompasses all observed debugging activities. By mapping behaviors, defects, and outcomes to the taxonomy's dimensions, we confirmed its applicability to real-world scenarios.

**Keywords**

taxonomy, debugging, software, information technology, software engineering, education

## 1. Introduction

Studies consistently show that developers spend between 30% and 50% of their time debugging, highlighting its importance and the inefficiencies inherent in current practices [1-3].

Existing research has explored various aspects of debugging, including specific techniques, tool development, and cognitive factors [4-6]. However, a systematic and comprehensive taxonomy that effectively categorizes the debugging process and factors influencing it is notably lacking. The absence of a unified framework to classify and analyze the debugging process limits our ability to address its challenges systematically, identify research gaps, and teach debugging effectively. The complexity of modern software systems necessitates a more structured approach to debugging.

Taxonomies are a well-established approach in software engineering for understanding, analyzing, and identifying similarities and differences within complex domains [7]. A taxonomy provides a systematic way to organize concepts into categories based on shared characteristics.

The development of a well-defined taxonomy of software debugging is crucial for several compelling reasons.

*Workshop "Software engineering and semantic technologies" SEST, co-located with 15th International Scientific and Practical Programming Conference UkrPROG'2025, May 13-14, 2025, Kyiv, Ukraine.*

[*] Corresponding author.

[†] These authors contributed equally.

shinkarenko_vi@ua.fm (V. Shynkarenko); o.o.zhevaho@ust.edu.ua (O. Zhevaho)

0000-0001-8738-7225 (V. Shynkarenko); 0000-0003-0019-8320 (O. Zhevaho)

Firstly, it provides a much-needed structured framework for comprehending the diverse array of debugging techniques, specialized tools, and various approaches employed in the field.

Secondly, a taxonomy can significantly enhance the systematic approach to debugging education and training, providing a clear roadmap for both instructors and learners.

Thirdly, a comprehensive classification system can serve as the foundation for the future development of more sophisticated and targeted debugging tools and strategies, ultimately leading to a more efficient and effective debugging process.

Debugging is often perceived as more of an art than a science, driven by individual experience rather than standardized practices [8]. To address this, we propose a taxonomy of the debugging process – a structured classification system that delineates its multifaceted nature.

The proposed taxonomy serves three primary purposes:

1. **Systematizing knowledge:** it organizes existing insights about debugging into a coherent framework.
2. **Identifying key factors:** it highlights critical elements influencing the debugging process.
3. **Supporting education:** it serves as an educational resource for students and practitioners, providing a structured understanding of debugging concepts and practices while enhancing software engineering curricula.

## 2. Background

The concept of taxonomy, as a classification scheme, has been widely adopted in software engineering research [7]. Researchers have consistently employed taxonomic approaches to map, categorize, and interpret diverse aspects of software development [7, 9, 10]. These taxonomies serve multiple critical functions: they provide conceptual clarity, support systematic literature reviews, guide empirical research, and offer structured frameworks for understanding complex software engineering phenomena.

Over the years, prior research has made significant contributions to our understanding of debugging, exploring a wide range of topics from theoretical foundations to practical applications.

In the domain of automated debugging, researchers have surveyed fault localization techniques, classifying them based on the program analysis methods used [11]. Others have proposed a taxonomy of automated program repair techniques, focusing on search strategies [12].

More recent studies have explored the integration of artificial intelligence in debugging, including natural language processing techniques and large language model-driven scientific debugging [13, 14].

From an educational perspective, research has examined debugging behaviors among novice and expert programmers [1, 15, 16]. Studies have explored how debugging proficiency develops through education and experience.

The human factors of the software debugging process have gained attention as a key variable [17, 18]. Cognitive styles — such as preferences for visual or analytical reasoning — shape how developers tackle problems [19]. Collaborative practices, including pair or team debugging, can enhance outcomes by pooling expertise. Emotional factors, such as frustration or confidence, also affect performance, with studies suggesting that stress impairs problem-solving [18].

Building on these foundations, the proposed taxonomy addresses prior limitations by unifying various dimensions. By synthesizing previous classifications with insights from systematic literature reviews, we aim to provide a more comprehensive understanding of the debugging process and its classification within software engineering research.

# 3. Taxonomy

## 3.1. Development process

The taxonomy development of the software debugging process adheres to the widely accepted methodology for creating taxonomies in software engineering, as established by Nickerson et al. and refined by Kundisch et al. [20, 21]. This methodology is considered the de facto standard in the field, providing a systematic, iterative, and theoretically grounded approach to taxonomy construction. The iterative method allows for both conceptual and empirical approaches. Process ending conditions after each iteration serve as an assessment of the taxonomy's completion. The process results in a taxonomy with dimensions and characteristics that are mutually exclusive and collectively exhaustive.

Taxonomy development can be subdivided into four main steps:

1. Determining meta-characteristics.
2. Determining ending conditions.
3. Selecting approach.
4. Checking the ending conditions.

The process starts with selecting the meta-characteristic, which should represent the most comprehensive feature, reflect the purpose of the taxonomy, and serve as the basis for selecting characteristics.

The primary goal is to develop a taxonomy that captures the complexity and diversity of the software debugging process.

We define the "key dimensions and characteristics of the software debugging process" as the meta-characteristic.

Further, the definition of ending conditions is required. For taxonomy development, we chose three ending conditions:

- each dimension is unique and not repeated;
- each characteristic is unique in its dimension;
- no new dimensions or characteristics were added in the last iteration.

These conditions were chosen to ensure uniqueness and stability within the taxonomy structure. By specifying that each dimension must be unique and non-repetitive, we aimed to avoid redundancy and maintain a clear conceptual distinction between the different aspects.

To develop taxonomy from different perspectives, we followed the recommendation to decide in each iteration whether an empirical-to-conceptual (E2C) or conceptual-to-empirical (C2E) approach was suitable [20]. To complete one iteration, the taxonomy was evaluated against the ending conditions, and if they were not met, an additional iteration was initiated.

We drew on four primary sources of data to develop the proposed taxonomy:

1. **Literature review:** we conducted a systematic review of academic literature on debugging, covering major software engineering journals and conferences. Based on the literature review, we identified recurring themes and categories related to debugging activities. We organized these into an initial taxonomic structure, identifying six primary dimensions: **What**, **Which**, **How**, **Who**, **When**, and **Where**.
2. **Tool analysis:** we analyzed debugging tools, including traditional debuggers, specialized tools for specific domains, and recent research prototypes.
3. **Developer surveys and studies:** we reviewed empirical studies of debugging practices, including surveys, observational studies, and analyses of developer behavior during debugging tasks.

4. **Industry practices:** we examined debugging methodologies and best practices documented in technical blogs, developer forums, and practitioner-oriented publications.

We conducted four iterations to meet the ending conditions and refine our final dimensions and characteristics. Starting with a C2E approach, we iteratively refined taxonomy through E2C iterations. By applying all ending conditions, we confirmed the validity of our dimensions and characteristics from the previous iteration, finalizing taxonomy.

## 3.2. Detailed description of dimensions

### 3.2.1. What is being debugged?

The **"What"** dimension of the debugging taxonomy defines the objects of debugging, focusing on the specific entities and issues that developers encounter during the debugging process. This dimension is essential because it characterizes the variety of debugging challenges, categorizing defects, software artifacts, root causes, complexity levels, scope, abstraction layers, code ownership, persistence characteristics, and debugging objectives.

Defects in software debugging manifest in multiple forms, each presenting unique challenges:

- **"Functional Bugs":** directly impact the software's core behavior and include syntax errors, semantic errors, logical flaws, missing features, and unhandled exceptions;
- **"Non-Functional Bugs":** encompass architectural inconsistencies, performance bottlenecks, memory leaks, security vulnerabilities, usability flaws, accessibility issues, and resource mismanagement. Addressing these defects often requires specialized knowledge and tools;
- **"Integration Issues":** bugs emerging from component interactions, such as API mismatches, configuration errors, dependency conflicts, protocol failures, network latency, and compatibility issues, complicate fault localization and require cross-component analysis;
- **"Environmental Bugs":** platform-specific constraints, hardware/OS dependencies, cloud infrastructure variability, browser-specific behaviors, and time zone inconsistencies. They are particularly challenging because they often appear only under specific conditions;
- **"Concurrency Bugs":** race conditions, deadlocks, thread starvation, and priority inversion arise due to parallel execution, demanding sophisticated debugging techniques;
- **"Data Flaws":** errors in data integrity, transformation inconsistencies, I/O failures, serialization errors, and schema mismatches are elusive because they depend on runtime states and interactions with external systems.

The scope of debugging is further defined by the artifacts being examined:

- **"Code":** includes source code, scripts, and compiled binaries;
- **"Infrastructure":** covers configurations, databases, network setups, and CI/CD pipelines, all of which influence execution behavior;
- **"Dependencies":** encompasses third-party libraries, APIs, SDKs, and other external components integrated into the system;
- **"User Interfaces":** debugging graphical interfaces involves identifying layout inconsistencies, event-handling errors, and accessibility compliance issues.

Understanding the root causes behind defects is crucial for effective debugging. These causes can be categorized into:

- **"Human Factors":** coding errors, miscommunication, and knowledge gaps introduce defects during development;

- **“Systemic Factors”:** ambiguous requirements, design debt, and technical debt contribute to persistent, structural issues;
- **“External Dependencies”:** changes in APIs, third-party library defects, and evolving platform constraints can introduce instability;
- **“Environmental Factors”:** OS/hardware limitations, cloud provider constraints, and resource availability (e.g., CPU/memory) often influence software behavior.

Defects vary in complexity, from simple errors with linear causes to multi-component failures:

- **“Complexity”:** ranges from straightforward syntax errors to emergent behaviors in distributed systems and architectural flaws;
- **“Scope”:** can be localized to a single function or extend across modules, or entire platforms;
- **“Abstraction Levels”:** bugs can manifest at different levels, including requirements, design, architecture, algorithms, and components.

The origin of the code influences debugging efficiency:

- **“Code Ownership”:** ownership affects familiarity and access. Debugging self-written code is typically more efficient than debugging team-developed code, third-party libraries, or legacy systems with minimal documentation;
- **“Persistence Characteristics”:** errors can be deterministic (consistently reproducible), non-deterministic (intermittent failures), environment-dependent, load-sensitive, or time-dependent, each requiring different debugging strategies.

The debugging process is guided by specific objectives, such as:

- **“Correctness”:** ensuring that software behaves as expected;
- **“Performance”:** optimizing speed, memory usage, and computational efficiency;
- **“Security”:** identifying and mitigating vulnerabilities;
- **“Usability & Accessibility”:** enhancing the end-user experience and compliance with accessibility standards.

Several factors influence the debugging process within this dimension:

- **“Software Characteristics”:** the size, complexity, and domain of the system dictate debugging strategies. A large-scale distributed system presents different challenges than a standalone application;
- **“Technology Stack”:** programming languages, frameworks, and architectural choices impact debugging approaches.

For example, debugging a cloud-based microservices architecture requires distributed tracing, while debugging embedded systems involves hardware-specific debugging tools. Similarly, addressing non-deterministic concurrency issues demands extensive logging and race-condition detection tools, whereas security vulnerabilities necessitate static analysis, fuzz testing, and penetration testing.

A debugging process can involve multiple aspects from this dimension simultaneously. A bug may have multiple defect types, impact various artifacts, stem from several root causes, and exhibit different complexity levels. By dissecting what’s being debugged, developers gain the insight needed to choose the right tools, methods, and strategies, transforming a potentially chaotic process into a systematic pursuit.

### 3.2.2. Which tools are used?

The **“Which”** dimension of the debugging taxonomy focuses on the technological tools used to detect, diagnose, and resolve defects. The choice of debugging tools significantly influences the efficiency and effectiveness of the debugging process, bridging the gap between identifying a defect (**“What”**) and applying the appropriate methodology (**“How”**).

Debugging tools can be classified based on their function and purpose:

- **“Static Analysis Tools”:** examine code without execution, identifying potential issues early in development. Examples include linters, vulnerability scanners, IDE plugins, code analyzers, data flow analyzers, and control flow analyzers;
- **“Dynamic Analysis Tools”:** monitor runtime behavior to uncover defects that manifest during execution. Examples include profilers, memory analyzers, concurrency analyzers, fuzzing tools, coverage tools, and execution tracers;
- **“Interactive Debuggers”:** allow developers to step through code execution and inspect states. Examples include time-travel debuggers, IDE-integrated debuggers, standalone debuggers, browser-integrated debuggers, and post-mortem debuggers;
- **“Specialized Debugging Tools”:** cater to specific domains, such as concurrency checkers, network analyzers, database debuggers, hardware debuggers, embedded system debuggers, cloud-native debugging tools, and CI/CD pipeline debugging tools;
- **“Visualization Tools”:** provide graphical representations of code execution and data structures, aiding in understanding complex behaviors;
- **“Custom Scripts”:** developers often create scripts tailored to specific debugging needs, automating repetitive analysis tasks;
- **“Remote Debugging Tools”:** enable debugging in distributed environments, allowing developers to inspect and control execution remotely;
- **“AI/ML-Driven Tools”:** leveraging machine learning, these tools enhance debugging efficiency through fault localization, automated test case generation, bug prediction, program slicing, and delta debugging;
- **“Observability Tools”:** provide real-time insights into software behavior, including log analysis, telemetry, tracing utilities, and metrics monitoring;
- **“Collaboration Tools”:** issue tracking systems, knowledge management platforms, and version control systems facilitate teamwork and debugging in large projects.

The choice of debugging tools depends on several factors:

- **“Defect Type and Artifact”:** the nature of the bug and the software component it affects steer tool choice. Memory leaks call for memory analyzers, concurrency issues demand thread checkers, and UI glitches might require browser tools. The artifact – be it source code, a database, or a network protocol – further refines the selection;
- **“Technological Context”:** programming languages, development environments, and system architectures limit or enable specific tools. Distributed systems often necessitate observability solutions, while simpler apps may not;
- **“Complexity and Scope”:** localized bugs might be tackled with interactive debuggers or print statements, but complex, systemic issues in large-scale systems require advanced tools like tracing utilities or AI-driven analyzers;
- **“Debugging Objectives”:** performance or security requirements influence tool selection;
- **“Developer Expertise”:** novices may rely on IDE-integrated tools with user-friendly interfaces, while experienced developers often use advanced analyzers and scripting;
- **“Organizational Constraints”:** budget, licensing, or policy restrictions may influence tool availability and selection.

For example, debugging a cloud-native microservices system requires distributed tracing tools, while debugging an embedded system may necessitate hardware-specific debugging interfaces. Similarly, security debugging relies on vulnerability scanners and penetration testing tools, whereas performance debugging depends on profilers and execution tracers. A debugging process often involves a combination of tools.

### 3.2.3. How is debugging performed?

The **"How"** dimension encapsulates the methodological core of debugging — the cognitive strategies, procedural techniques, and analytical approaches that developers employ to detect, diagnose, and resolve software defects. This dimension represents the intellectual problem-solving journey that bridges the gap between identifying what is broken and selecting which tools to employ. It encompasses both the tactical execution of debugging tasks and the strategic reasoning that guides the overall process.

Debugging methods can be categorized by their degree of automation:

- **"Manual Debugging":** relies primarily on human intellect and direct code interaction. Code reviews, print statements, logging, trial and error, and intuition-based approaches. These approaches are distinguished by their accessibility and minimal tooling requirements, though they demand significant cognitive effort and domain knowledge;
- **"Semi-Automated Debugging":** augments human reasoning with tool-assisted inspection and analysis. Interactive debugging (breakpoints, stepping, watches), profiling, memory inspection, and reverse debugging. These hybrid approaches balance human insight with computational assistance, offering enhanced visibility into program behavior while maintaining investigator control;
- **"Automated Debugging":** minimizes human intervention through algorithmic analysis and decision-making. These include: AI-driven repair, fuzz testing, automated test generation, anomaly detection, root cause suggestion, and self-healing systems.

Developers apply various reasoning strategies during debugging:

- **"Deductive Reasoning":** applying general rules to identify specific issues;
- **"Inductive Reasoning":** recognizing patterns in failures to infer potential causes;
- **"Abductive Reasoning":** identifying the most likely explanation for an observed defect;
- **"Heuristics & Intuition":** leveraging experience, pattern recognition, and counterfactual hypothesis generation;
- **"Hypothesis-driven debugging":** formulating and testing potential explanations;
- **"Forward reasoning":** tracing from potential causes to observed effects;
- **"Backward reasoning":** working from symptoms toward root causes;
- **"Rubber duck debugging":** articulating the problem to expose inconsistencies in understanding.

These strategies often overlap, and developers adjust their approach as new insights emerge.

Analysis techniques provide structured ways to diagnose defects:

- **"Static Analysis":** code examination without execution, including flow analysis;
- **"Dynamic Analysis":** runtime monitoring using profilers, tracers, and symbolic execution;
- **"Statistical & ML-Based Analysis":** anomaly detection, fault prediction, and comparative historical analysis;
- **"Root Cause Analysis":** identifying underlying defects through causal tracing.

Specific debugging techniques include:

- **“Isolation Strategies”:** methods to narrow the problem space. Binary search, divide and conquer, minimal reproducible example, sandboxing, and dependency mocking;
- **“Visualization-Based Methods”:** graphical representations of execution flow, dependencies, and state changes;
- **“Tracing and Monitoring”:** methods focused on monitoring system behavior. Observability tools for performance monitoring and anomaly detection;
- **“Exploratory and Systematic Debugging”:** hypothesis testing and structured investigation;
- **“Collaborative Debugging”:** techniques leveraging collective expertise. Mob debugging, pair programming, and team-based defect resolution.

The choice of debugging method is influenced by:

- **“Defect Complexity”:** simple issues may be resolved with print statements, while complex concurrency bugs require symbolic execution or automated tools;
- **“Available Tools”:** advanced debugger or AI solutions enable more sophisticated methods;
- **“Developer Expertise”:** experienced developers may use systematic or intuition-driven strategies, while novices rely on trial-and-error;
- **“Time Constraints”:** tight deadlines favor quick fixes like logging over in-depth root cause analysis;
- **“System Architecture”:** distributed systems require tracing and monitoring, while standalone applications may benefit from simpler debugging tools;
- **“Organizational Factors”:** collaboration-oriented cultures may prioritize team-based debugging over individual efforts.

A debugging approach can incorporate multiple strategies simultaneously.

Understanding this dimensional taxonomy empowers practitioners to select appropriate methodologies for specific debugging scenarios, ultimately enhancing both the efficiency and effectiveness of the debugging process.

### 3.2.4. Who is debugging?

The **“Who”** dimension explores the human element in debugging, recognizing that the characteristics, expertise, and collaborative dynamics of individuals profoundly influence the debugging process. This dimension bridges the technical and social aspects of debugging, acknowledging that while software defects exist in code, their resolution occurs through human cognition, interaction, and decision-making.

Debugging involves various stakeholders with distinct roles and perspectives:

- **“Developers”:** primary coders responsible for identifying and fixing defects;
- **“Testers”:** identifies and reports issues during testing phases;
- **“Security Engineers”:** address vulnerabilities and ensure system integrity;
- **“Performance Engineers”:** focus on optimization and efficiency;
- **“Domain Experts”:** provide contextual insights for specialized systems;
- **“System Administrators”:** debug deployment and infrastructure-related issues;
- **“AI Agents & Automated Tools”:** assist in defect localization, anomaly detection, and automated fixes.

Debugging effectiveness varies significantly across expertise levels:

- **“Novices”:** rely on simple tools like linters and trial and error approaches;

- **“Intermediate Developers”:** utilize systematic debugging techniques and interactive tools;
- **“Experts”:** leverage intuition, deep knowledge, and advanced debugging strategies.

Debugging has evolved from an individual effort to a collaborative activity, including:

- **“Solo”:** traditional approach where a developer works independently;
- **“Pair”:** two developers collaboratively troubleshoot code;
- **“Team Debugging (Mob Debugging)”:** a group actively resolves issues together;
- **“Cross-Functional Collaboration”:** experts from various domains contribute to debugging complex issues;
- **“Human-AI Collaboration”:** developers work alongside AI tools for enhanced efficiency.

Several factors shape the debugging process:

- **“Defect Complexity”:** simple bugs may be handled individually, while complex issues require specialized expertise or team collaboration;
- **“Bug Severity”:** critical security flaws involve security engineers, while performance issues engage optimization specialists;
- **“Time Constraints”:** urgent fixes may demand teamwork for faster resolution;
- **“Cognitive Styles & Emotional Factors”:** debugging effectiveness is influenced by problem-solving preferences, frustration tolerance, and confidence;
- **“Project Structure”:** large-scale, distributed projects necessitate collaboration tools, while smaller teams may rely on direct communication;
- **“AI Integration”:** automated debugging tools reshapes collaboration dynamics.

Understanding this dimensional taxonomy helps organizations optimize their debugging processes by recognizing the critical importance of human factors. As software systems grow in complexity, the **“Who”** dimension will likely gain increasing prominence, with successful debugging depending not just on technical capabilities but on effectively leveraging diverse human expertise and collaborative potential.

### 3.2.5. When does debugging occur?

The **“When”** dimension examines the temporal aspects of debugging, recognizing that different software lifecycle stages present unique debugging challenges and opportunities.

Debugging occurs at various stages of the software development lifecycle:

- **“Requirements and Design Phases”:** identifying ambiguities early;
- **“Implementation Phase”:** addressing compile-time and runtime errors;
- **“Testing Phase”:** debugging during unit, integration, and regression testing;
- **“Post-Release Maintenance”:** resolving production issues and vulnerabilities.

Debugging activities can be classified based on their timing relative to defect manifestation. Different strategies are employed depending on whether the goal is to prevent defects, address them as they arise, or analyze them after failure:

- **“Preventive Debugging”:** focuses on identifying and eliminating defects before they manifest in execution. It involves proactive measures to improve code quality and prevent errors from reaching production;
- **“Reactive Debugging”:** triggered when a defect is detected during execution and requires immediate intervention. This is the most common form of debugging;

- **“Retrospective Debugging”:** takes place after a failure has occurred, often when debugging live or deployed systems. The goal is to reconstruct the cause of failure using recorded artifacts.

Key influencing factors include:

- **“Development Methodology”:** Agile teams debug iteratively, Waterfall defers fixes to later phases;
- **“Bug Detectability”:** latent issues (e.g., memory leaks) surface post-release;
- **“Release Deadlines”:** time constraints prioritize critical fixes over comprehensive validation.

### 3.2.6. Where does debugging take place?

The **“Where”** dimension explores the diverse execution environments and platforms that shape the debugging process, reflecting the increasingly complex and distributed nature of modern software systems.

Debugging can occur across different platforms and environments:

- **“Platforms”:** desktop, web, mobile, embedded systems, cloud;
- **“Execution Environments”:** local, testing, production, and distributed systems.

Key influencing factors include:

- **“Deployment Targets”:** cloud debugging requires remote tools; embedded systems need hardware simulators;
- **“Resource Availability”:** production environments limit invasive debugging;
- **“System Distribution”:** distributed architectures necessitate network analyzers and telemetry;
- **“Accessibility”:** production debugging is limited by access restrictions, unlike local environments;
- **“Performance Concerns”:** debugging in production must minimize user impact, requiring lightweight tools;
- **“Data Sensitivity”:** real user data in production demands privacy compliance, unlike test environments.

The environment in which debugging takes place is a critical dimension that influences the approach and the available tools.

## 3.3. Overall structure and schema

The proposed taxonomy organizes the software debugging process into six primary dimensions, forming a faceted classification system, where each dimension represents a distinct aspect of the debugging activities (Figure 1):

- objects of debugging (**“What”**) – the entities being debugged, including software artifacts and defect types;
- tools (**“Which”**) – the debugging tools utilized;
- methods and techniques (**“How”**) – the approaches employed to detect, localize, and resolve defects;
- human factors (**“Who”**) – the experiential, and collaborative aspects of the individuals performing debugging;

- temporal aspects (**“When”**) – the timing considerations in debugging, including when debugging occurs within the development lifecycle;
- execution context (**“Where”**) – the environmental factors in which debugging takes place, including the execution platform.

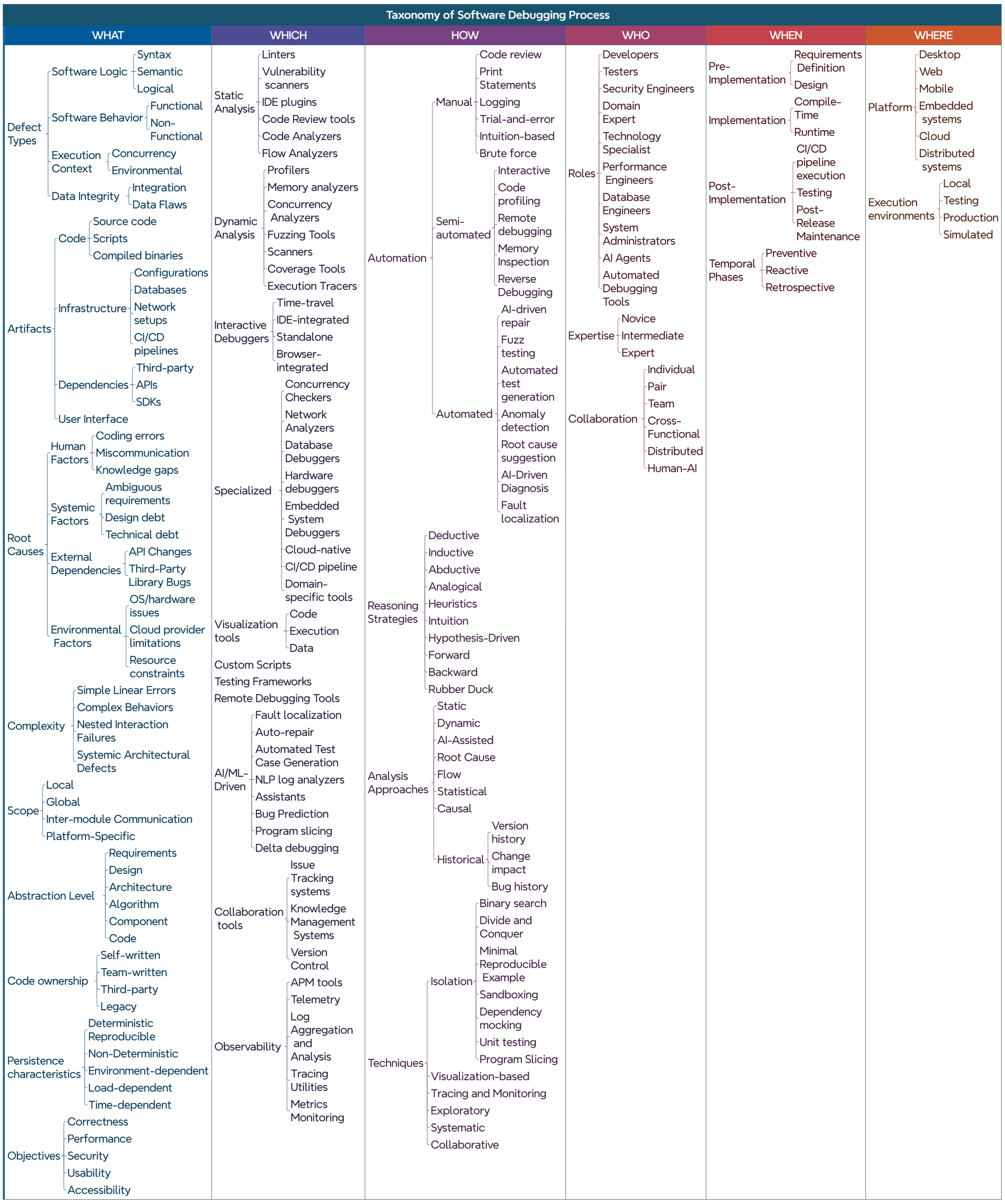

**Figure 1:** Taxonomy of the software debugging process.

Each dimension consists of multiple elements that capture key aspects of debugging, and are orthogonal to the others, meaning that a debugging activity can be categorized across all dimensions independently. However, these dimensions are interdependent. For example, the choice of tools (**“Which”**) may depend on the defect type (**“What”**), debugging technique (**“How”**), and the developer’s expertise (**“Who”**).

Multiple elements from the same dimension can apply simultaneously, reflecting the complexity and multi-faceted nature of debugging. Debugging processes often involve multiple tools, techniques, and defect types simultaneously.

This structure enables researchers and practitioners to analyze debugging activities from multiple perspectives, facilitating a deeper understanding of how various factors combine to shape the process.

## 4. Validation

To assess the validity of our proposed taxonomy for the software debugging processes, we utilized data from an experiment conducted with students from the "Software Engineering" specialty at the Ukrainian State University of Science and Technologies [22]. This experiment, structured as a debugging olympiad, provided detailed insights into student debugging behaviors, enabling us to evaluate whether the taxonomy comprehensively and accurately captures the dimensions of debugging observed in a real-world educational setting.

The experiment analyzed the debugging behaviors observed in 41 students during 15 logical-error tasks, generating 487 event logs, 2,415 debugging sessions, and 16,536 events [22].

The data from the experiment was analyzed through the lens of the proposed taxonomy. Each debugging session from the log files was mapped to the taxonomy's dimensions and categories to identify key debugging characteristics and behaviors.

We evaluated how the experimental data corresponds to each taxonomy dimension:

- **“What”:** the experiment focused on “Logical Errors” within “Code” (“Source Code”). The errors were designed to be of “Simple Linear Errors” complexity, with a “Code-Level” abstraction level. This focus validates the inclusion of logical errors but suggests a need to test the taxonomy’s applicability to other defect types in future studies;
- **“Which”:** students utilized the Visual Studio IDE, employing features such as breakpoints, step-by-step execution, and variable value displays. These align with the "interactive debuggers" category in the proposed taxonomy, confirming its relevance;
- **“How”:** several debugging methods were evident, including “Manual” methods like “Trial-and-Error” (especially in the “Low” proficiency group) and “Semi-Automated” methods using “Interactive Debugging” features like breakpoints, stepping, and watches. The analysis also revealed different “Reasoning Strategies” with some students employing a more “Hypothesis-Driven” approach (especially in the “High” proficiency group);
- **“Who”:** students were classified into “High”, “Middle”, and “Low” based on performance. This classification corresponds to expertise levels in the taxonomy;
- **“When”:** the debugging activities fall within the “Implementation Phase” and “Testing Phase” as students were tasked with finding and fixing errors in given code;
- **“Where”:** the debugging occurred within the “Local Development Environment” using the “Desktop” platform.

The experimental results provide valuable insights into the validity and applicability of the taxonomy:

- **“Comprehensiveness”:** the taxonomy effectively captured the key elements of the debugging processes observed in the experiment. All major activities and tools used by the students could be categorized within the taxonomy's dimensions, confirming its completeness;
- **“Relevance”:** the taxonomy's categories proved relevant to describing the debugging behaviors of students, highlighting the importance of categories such as “Logical Errors”,

“Interactive Debuggers”, “Trial-and-Error”, and “Hypothesis-Driven Debugging” in an educational setting;
- **“Differentiation”:** the taxonomy allowed for the differentiation between debugging approaches based on student expertise.

The experimental data confirms the taxonomy’s validity. The taxonomy encompasses all observed debugging elements, from logical errors to tools (Visual Studio) and strategies (trial-and-error to systematic debugging). Behavioral patterns and expertise differences align with the taxonomy’s categories, accurately reflecting student debugging processes.

## 5. Discussion

The taxonomy we have presented is not intended to be static. Rather, it provides a foundation that can evolve as debugging practices and technologies continue to develop. We hope that this taxonomy will stimulate further research on debugging and contribute to the development of more effective debugging approaches that reduce the time and effort required to identify and fix software bugs. This integrated, multi-level taxonomy provides a structured way to classify and analyze software debugging activities based on a variety of relevant factors.

However, it is important to acknowledge the inherent limitations of any such classification system. The field of software development is constantly evolving with the emergence of new technologies and paradigms, which may necessitate periodic updates and revisions to the taxonomy to maintain its relevance and comprehensiveness. While the taxonomy aims to be generally applicable, it may not fully capture the nuances of debugging in highly specialized domains such as embedded systems, safety-critical software, or machine learning applications.

Despite these limitations, the proposed taxonomy has significant potential applications.

In research, it can provide a basis for conducting more rigorous empirical studies to evaluate the effectiveness of different debugging techniques for specific types of bugs or in particular development environments.

In education, it can provide a structured framework for developing debugging curricula in computer science and software engineering programs. Instead of presenting debugging as a collection of disparate techniques, educators can use the taxonomy to provide a comprehensive and systematic overview of the debugging landscape, helping students understand how different techniques relate to each other and when each is most applicable.

## Declaration on Generative AI

During the preparation of this work, the authors used AI program Chat GPT 4.0 for correction of text grammar. After using this tool, the authors reviewed and edited the content as needed and take full responsibility for the publication’s content.

## 6. Conclusion

This paper presents a novel taxonomy of the software debugging process, examining debugging from six complementary perspectives: **what** is being debugged, **which** tools are used, **how** debugging is performed, **when** debugging occurs, **where** debugging takes place, and **who** is debugging. These dimensions are enriched by influencing factors, offering a structured classification that bridges theory and practice. Each dimension is further divided into categories that capture the diversity of debugging approaches. By decomposing debugging into these fundamental dimensions, the proposed taxonomy aims to facilitate a deeper understanding of the software debugging process.

Future work will focus on validating and applying the taxonomy in different contexts and domains, as well as exploring the relationships between these dimensions.

## References

[1] M. Perscheid, B. Siegmund, M. Taeumel, R. Hirschfeld, Studying the advancement in debugging practice of professional software developers, Software Quality Journal (2017) 83–110. doi:10.1007/s11219-015-9294-2.

[2] A. Alaboudi, T. D. Latoza, What constitutes debugging? An exploratory study of debugging episodes, Empirical Software Engineering (2023). doi:10.1007/s10664-023-10352-5.

[3] Y. Malysheva, C. Kelleher, B. J. Ericson, Interrelation between Teaching Assistants' debugging strategies and adherence to sound tutoring practices during office hours, in: Proceedings of the 24th Koli Calling International Conference on Computing Education Research, Koli Calling '24, ACM, New York, 2024. doi:10.1145/3699538.3699562.

[4] S. Yang, M. Baird, E. O'Rourke, K. Brennan, B. Schneider, Decoding Debugging Instruction: A Systematic Literature Review of Debugging Interventions, ACM Transactions on Computing Education (2025) 1–44. doi:10.1145/3690652.

[5] V. Shynkarenko, O. Zhevaho, Development of a Toolkit for Analyzing Software Debugging Processes Using the Constructive Approach, Eastern-European Journal of Enterprise Technologies (2020). doi:10.15587/1729-4061.2020.215090.

[6] J. Lee, A. M. Kazerouni, C. Siu, T. Migler, Exploring the Impact of Cognitive Awareness Scaffolding for Debugging in an Introductory Programming Class, in: Proceedings of the 54th ACM Technical Symposium on Computer Science Education, SIGCSE 2023, ACM, New York, 2023, pp. 1007–1013. doi:10.1145/3545945.3569871.

[7] M. Usman, R. Britto, J. Börstler, E. Mendes, Taxonomies in software engineering: A systematic mapping study and a revised taxonomy development method, Information and Software Technology (2017). doi:10.1016/j.infsof.2017.01.006.

[8] G. A. Wilkin, Debugging: From Art to Science A Case Study on a Debugging Course and Its Impact on Student Performance and Confidence, in: Proceedings of the 56th ACM Technical Symposium on Computer Science Education, SIGCSETS 2025, ACM, New York, 2025, pp. 1225–1231. doi:10.1145/3641554.3701893.

[9] O. Marchenko, D. Dvoichenkov, TaxoRankConstruct: A Novel Rank-based Iterative Approach to Taxonomy Construction with Large Language Models, in: Proceedings of the Information Technology and Implementation Workshop, IT&I 2024, Kyiv, 2024, pp. 11–27.

[10] Y. Sasaki, H. Washizaki, J. Li, N. Yoshioka, N. Ubayashi, Y. Fukazawa, Landscape and Taxonomy of Prompt Engineering Patterns in Software Engineering, IT Professional (2024) 41-49. doi:10.1109/MITP.2024.3525458.

[11] E. Scott, A. Soria, M. Campo, Taxonomy-based Approach For Fault Localization In Service-Oriented Applications, IEEE Latin America Transactions (2016), pp. 2348-2354. doi:10.1109/TLA.2016.7530432.

[12] K. Huang, Z. Xu, S. Yang, H. Sun, X. Li, Z. Yan, Y. Zhang, Evolving Paradigms in Automated Program Repair: Taxonomy, Challenges, and Opportunities, ACM Computing Surveys (2024), pp. 1-43. doi:10.1145/3696450.

[13] Y. Liang, Y. Ji, Z. Wu, A semi-supervised multilabel classification approach applied to debugging target prediction in debugging education, in: Proceedings of the 3rd International Conference on Electronic Information Engineering, Big Data, and Computer Technology, EIBDCT 2024, Beijing, 2024. doi:10.1117/12.3031156.

[14] Q. Ma, H. Shen, K. Koedinger, S. T. Wu, How to Teach Programming in the AI Era? Using LLMs as a Teachable Agent for Debugging, in: Proceedings of the 25th International Conference on Artificial Intelligence in Education, AIED 2024, Recife, 2024. doi:10.1007/978-3-031-64302-6_19.

[15] B. S. Alqadi, Enhancing Novice Programmers' Debugging Skills Through Systematic Education: A Comparative Study, IEEE Access (2024), pp. 181192-181204. doi:10.1109/ACCESS.2024.3509641.

[16] E. Park, J. Cheon, Exploring Debugging Challenges and Strategies Using Structural Topic Model: A Comparative Analysis of High and Low-Performing Students, Journal of Educational Computing Research (2024) 2104-2126. doi:10.1177/07356331241291174.
[17] L. Gale, S. Sentance, Investigating the Attitudes and Emotions of K-12 Students Towards Debugging, in: Proceedings of the 5th United Kingdom and Ireland Computing Education Research, UKICER 2023, Swansea, 2023. doi:10.1145/3610969.3611120.
[18] D. Hu, P. Santiesteban, M. Endres, W. Weimer, Towards a Cognitive Model of Dynamic Debugging: Does Identifier Construction Matter?, IEEE Transactions on Software Engineering (2024), pp. 3007-3021. doi:10.1109/TSE.2024.3465222.
[19] K. Yoshimori, T. Kano, T. Akakura, Gaze Analysis and Modeling of Cognitive Process During Debugging for Novice Programmers' Learning, in: Proceedings of the 24th HCI International Conference, HCII 2022, Swansea, 2022. doi:10.1007/978-3-031-06424-1_39.
[20] R. Nickerson, U. Varshney, J. Muntermann, A method for taxonomy development and its application in information systems, European Journal of Information Systems (2013), pp. 336–359. doi:10.1057/ejis.2012.26.
[21] D. Kundisch, J. Muntermann, A. M. Oberländer, D. Rau, M. Röglinger, T. Schoormann, D. Szopinski, An Update for Taxonomy Designers, Business & Information Systems Engineering (2022), pp. 421–439. doi:10.1007/s12599-021-00723-x.
[22] V. Shynkarenko, O. Zhevaho, A Video-Based Approach to Learning Debugging Techniques, in: Proceedings of the 14th International Scientific and Practical Programming Conference, UkrPROG 2024, CEUR, Kyiv, 2024, pp. 462–473.