

Article

BLOW: A Systematic Approach to Behavior-Driven Development in a Layered Organization of Work-Centers

Nicolas Afonso-Alonso ^{1,2} , Juan A. Holgado-Terriza ^{1,*} , Miguel A. Oltra-Rodríguez ^{1,2} 
and Paul Stonehouse ² 

¹ Software Engineering Department, Research Centre for Information and Communication Technologies (CITIC-UGR), University of Granada, 18071 Granada, Spain; nicolas.afonso@se.com (N.A.-A.); miguel.oltra@se.com (M.A.O.-R.)

² Schneider-Electric, 41092 Seville, Spain; paul.stonehouse@se.com

* Correspondence: jholgado@ugr.es

Abstract

Agile teams often struggle to translate business requirements into maintainable, high-quality software due to the persistent ambiguity in the roles and relationships of behavior-driven development (BDD), Acceptance Test-driven Development (ATDD), and Test-driven Development (TDD). These approaches are frequently misunderstood, inconsistently applied, and only loosely connected within a unified delivery lifecycle. This article introduces BLOW (Behavior-Driven Development in a Layered Organization of Work-Centers), a systematic approach that establishes BDD as the coordinating methodology between ATDD (business-focused) and TDD (technology-focused). BLOW structures scenario-driven development across layered domains of accountability with clearly defined roles and responsibilities, organizing delivery through nested work-centers that transform user stories into executable specifications and production code. This approach integrates two complementary collaboration practices: the Three Amigos for discovering and formulating business scenarios, and the proposed Technical Three Amigos for linking those scenarios to Technical Domain Contexts, identifying required Enablers, and deriving technical scenarios when additional architectural support is needed. The proposed operating model emphasizes observability through executable scenarios as first-class artifacts, introducing native, test-anchored metrics that support reasoning about progress, technical effort, and value delivery within scenario-driven development. An exploratory longitudinal case study, consisting of a single-sprint proof of concept followed by an 18-month production deployment, reports patterns in which technical enablement precedes business value delivery and reusable infrastructure supports sustained growth of business scenarios over time. The findings also indicate that changes in the applied operating model are associated with measurable shifts in scenario evolution and internal quality indicators. Overall, BLOW provides a governance-compatible, end-to-end approach for organizing scenario driven development and improving alignment between stakeholder intent and technical implementation in complex software systems.



Academic Editors: Amit Kumar Mishra and Yan Liu

Received: 28 April 2026

Revised: 8 June 2026

Accepted: 18 June 2026

Published: 25 June 2026

Copyright: © 2026 by the authors.

Licensee MDPI, Basel, Switzerland.

This article is an open access article

distributed under the terms and

conditions of the [Creative Commons](https://creativecommons.org/licenses/by/4.0/)

[Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

Keywords: Behavior-Driven Development (BDD); Acceptance Test-Driven Development (ATDD); Test-Driven Development (TDD); systematic approach; software development; software engineering governance; executable specifications; empirical software engineering

1. Introduction

The effective translation of complex business requirements into high-quality, maintainable software remains a significant and persistent challenge in modern software engineering, particularly within dynamic agile development contexts [1,2]. To address this challenge, practices such as behavior-driven development (BDD), Acceptance Test-driven Development (ATDD), and Test-driven Development (TDD) have been widely adopted. These approaches are valued for their potential to bridge the gap between business intent and technical implementation, fostering collaboration, and improving software quality [3,4]. To anchor these paradigms within their established historical foundations, we leverage their canonical definitions: TDD as popularized through the test-first disciplined cycle by Beck [5], BDD as a communication and specification methodology originated by North [6], and ATDD as formalized through business-driven acceptance requirements within agile governance frameworks [7].

Despite their widespread adoption, the practical integration of BDD, ATDD, and TDD within a coherent development lifecycle remains unclear. Both the academic literature and industrial practice reveal substantial ambiguity regarding their definitions, scope, and interrelationships. Several studies describe BDD as a direct evolution or refinement of TDD, emphasizing its roots in test-first development and executable specifications [8–10]. In contrast, other authors portray BDD as a synthesis of acceptance-level and unit-level practices, positioning it closer to ATDD and focusing on behavioral specification over technical design [11,12].

Additional perspectives further complicate this landscape. Fontela and Garrido [13] treat ATDD and BDD as largely interchangeable, while also describing ATDD as a specific form of TDD. Garner [14] distinguishes ATDD from TDD primarily by its focus on acceptance criteria rather than code structure, thereby viewing the approaches as complementary but loosely coupled. Ferguson et al. [15] advance a clearer hierarchical interpretation by identifying ATDD as a related practice that enables, but does not replace, BDD. Meanwhile, comparative empirical studies such as those by Cui et al. [4] implicitly assume the existence of distinct models for BDD and TDD without resolving their operational overlap within a unified lifecycle.

Existing definitions of BDD also exhibit significant variability. While some definitions emphasize its relationship with test-first development, others foreground collaboration and specification as executable examples [3,8–10]. Other definitions adopt a practice-centric perspective, describing BDD primarily as a set of practices aimed at deriving automated tests from team communication rather than as a coordinating methodology spanning the development lifecycle [16]. Although these definitions are valid, they tend to focus on observable characteristics of BDD rather than on its methodological role within software development. This limitation is particularly evident in practice-oriented studies that emphasize the use of ubiquitous language in BDD, which, while valuable, are often not situated within a clearly defined end-to-end development model [17].

The absence of a shared conceptual and operational framework has tangible consequences. Although TDD has achieved a relatively stable position within the software development lifecycle and aligns naturally with DevOps practices, the positioning of ATDD and BDD remains less clear. As a result, organizations frequently adopt these practices in isolation, apply them selectively, or abandon them altogether due to unclear responsibilities, coordination overhead, and tooling mismatches [9,18]. Empirical studies repeatedly show that the benefits attributed to acceptance- and behavior-driven approaches, such as improved collaboration, faster feedback, and higher quality, depend more on disciplined coordination between people, processes, and tools rather than on test automation itself [19,20].

Taken together, these observations suggest that the core challenge is not the absence of techniques, but the absence of a unifying operating model that explicitly situates ATDD, TDD, and BDD within a shared organizational and methodological structure. Without such a model, scenario-driven practices struggle to scale and to remain sustainable over time.

In response to these challenges, this paper introduces BLOW (Behavior Driven-Development in a Layered Organization of work-centers), a systematic approach that establishes an operating model for scenario driven development. BLOW positions BDD as a coordinating methodology that explicitly integrates ATDD as a business-focused framework and TDD as a technology-focused framework within a layered structure of accountability. The proposed approach organizes development through nested work-centers and complementary collaboration practices, making technical enablement explicit and observable alongside business value delivery.

To clarify the nature of the contribution, we distinguish three levels of novelty. First, BLOW introduces or formalizes BLOW-specific constructs for scenario-driven governance: (a) scenario-anchored metrics that use executable scenarios as formal units of value for tracking cost, progress, and completeness; (b) the Technical Three Amigos, a dedicated collaboration practice for technical scenario discovery; (c) a layered operating model with nested work-centers that distributes BDD, ATDD, and TDD across distinct organizational boundaries with explicit push-pull dynamics; and (d) the formal distinction between business scenarios and technical scenarios as disjoint unit-of-value types with separate accountability paths. Second, BLOW adapts and scopes established concepts, including SAFe Enablers, Lean-inspired work-center flow, and the Three Amigos practice, integrating them within a unified governance structure. Third, BLOW provides a positioning contribution by situating BDD as a coordinating methodology between ATDD and TDD, helping to clarify the definitional ambiguity documented in the literature and in the empirical survey presented in Section 2.

Beyond conceptual clarification, BLOW introduces native, test anchored metrics that leverage executable scenarios as first class artifacts for reasoning about progress, technical effort, and value delivery. The proposed operating model is evaluated through a longitudinal case study, spanning a single sprint proof of concept and an 18 month production deployment, to explore its practical implications for coordination, observability, and internal software quality.

The remainder of the paper is structured as follows: Section 2 presents an empirical survey examining current perceptions and adoption patterns of BDD, ATDD, and TDD among practitioners and academics. Section 3 establishes a unified conceptual foundation by analyzing the fragmentation of knowledge between business and technical domains and by delineating the roles of TDD and ATDD within a BDD-led development lifecycle. Section 4 introduces the BLOW systematic approach, detailing its layered organizational structure, core principles, and collaboration practices. Section 5 defines the native metrics associated with BLOW, providing a formal basis for reasoning about cost, value, and progress through executable scenarios. Section 6 reports the longitudinal case study used to evaluate the proposed approach. Section 7 discusses the limitations of the study. Section 8 interprets the empirical findings and discusses their methodological and organizational implications. Finally, Section 9 concludes the paper and outlines directions for future work.

2. BDD Perception: An Empirical Survey

In order to validate the hypothesized conceptual confusion identified in the literature, an empirical perception survey was conducted. The objective of this study was to assess the extent of ambiguity surrounding BDD, ATDD, and TDD among active practitioners and experts in the field.

The survey was designed to capture both the theoretical understanding and the practical application of these methodologies. To ensure a representative sample of the industry, we targeted two distinct populations:

- Population 1 (industry): This cohort consists of active software professionals surveyed during an international software development summit. It represents a broad cross-section of the industry, including roles such as software architects, DevOps engineers, and technical leads. This population provides empirical insight into the interpretation and application of BDD, ATDD and TDD in the industry.
- Population 2 (academia): This group comprises postgraduate students enrolled in a Master's degree program in Software Development. Participants had undergone rigorous training in modern agile frameworks and formal development methodologies. This population serves to capture the perception of BDD from a theoretical and structured educational perspective, relatively untainted by legacy industrial habits.

All the participants were assured that their personal data would be anonymized, and their consent was formally obtained prior the assessment.

The questionnaire was organized into two blocks: (i) audience composition and (ii) opinions on the definition and adoption of BDD. The BDD-related items were organized into two categories: (i) BDD adoption and understanding and (ii) BDD perception. The reference total for the number of potential responses was determined by the first question in the survey, resulting in 52 expected responses for Population 1 and 20 expected responses for Population 2.

For the respondents who answered the audience composition questions but omitted responses in the adoption and perception blocks, we implicitly assigned the default option "I do not have an opinion". This conservative imputation treats non-responses as uncommitted to avoid artificially inflating support or consensus for any particular viewpoint during the baseline data organization.

2.1. Audience Composition

Two questions (RQ1-RQ2) were conducted in order to characterize the respondents according to their roles and levels of experience. Table 1 shows a comparison between both target populations.

Table 1. Comparison of Target Populations 1 and 2: Survey demographics.

Question/Category	Population 1		Population 2	
	Percentage	Answer	Percentage	Answer
RQ1: Your role fits better in:				
Development team member	71.15%	37	90.00%	18
Product owner	15.38%	8	5.00%	1
Stakeholder	13.46%	7	5.00%	1
RQ2: Years of working experience:				
More than 10 years	69.39%	34	15.00%	3
5–10 years	14.29%	7	0.00%	0
0–5 years	16.33%	8	85.00%	8

Specifically, Population 1 included an audience of 52 participants and was composed mainly of senior technologists with extensive software development experience. In this population, 71.15% identified as members of the development team, 15.38% as product owners, and 13.46% as stakeholders. Regarding experience, 69.39% reported more than 10 years, 14.29% reported 5–10 years, and 16.33% reported 0–5 years. Three individuals did not answer and are not considered in the table.

In contrast, Population 2 consisted of 20 participants, primarily individuals who had only recently completed their grade studies and therefore possessed an up-to-date academic background. Most participants identified as development team members (90%), with very few product owners or stakeholders. As expected, the vast majority (72.73%) reported 0–5 years of working experience. In this case, nine individuals preferred to avoid answering and are not considered in the table.

A direct comparison of role composition is provided in Figure 1, which offers a visual overview of the observed outcomes. Across both conditions, the predominant portion of the audience consists of members of the development team, allowing us to regard them as the principal represented group.

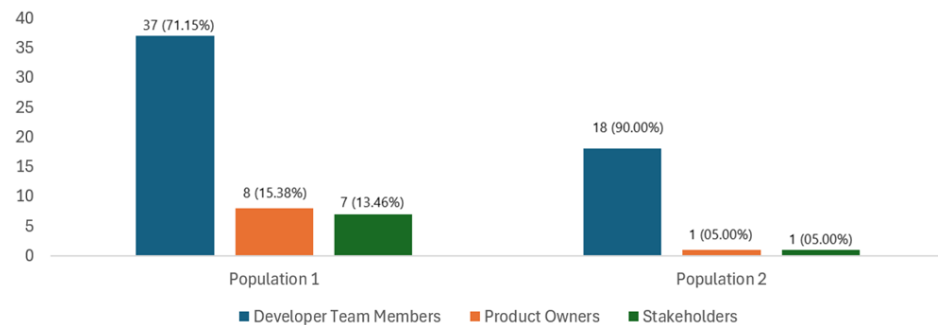


Figure 1. RQ1: Graphical representation of the audience composition.

The distribution of experience is illustrated in Figure 2. The distribution spans a broad range of experience levels, suggesting substantial heterogeneity within the sample. This pattern indicates that both senior and junior practitioners are well represented in the dataset, rather than the data being skewed toward a particular experience group.

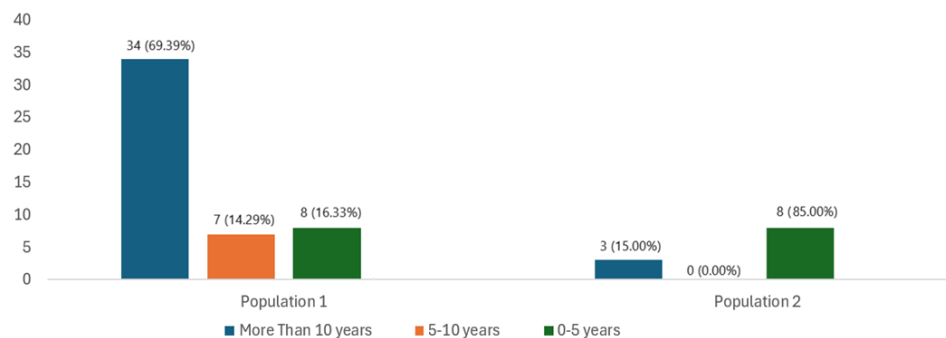


Figure 2. RQ2: Experience distribution. When the survey responses were aggregated, the resulting sample was primarily composed of individuals with substantial professional experience, with the majority having more than five years in their respective roles.

2.2. BDD Adoption and Understanding

Two questions were formulated to evaluate the participants' familiarity with BDD and the extent to which BDD is applied in their development processes, as it is shown in the Table 2.

For Population 1, most participants categorized BDD either as a development methodology or as both a testing and development methodology, plausibly influenced by the explicit presence of the word “development” in the term BDD. Nonetheless, as reported in Table 2, most respondents had little or no direct exposure to BDD in professional settings. Furthermore, when combining those who did not select any predefined option with those who left the question unanswered, a substantial proportion can be classified as not having a clearly defined position regarding BDD. Overall, the collected data suggests for Population

1 that, although BDD is generally regarded by the respondents as a valuable driver for software development practices, its actual, systematic application in real-world projects remains limited.

Table 2. Comparative analysis of BDD adoption between P1 and P2.

Question/Category	Population 1 (P1)			Population 2 (P2)		
	Answer	with Default	Percentage	Answer	with Default	Percentage
RQ3: Your perception of BDD is:						
Development methodology	5	5	09.61%	3	3	15.00%
Both testing and dev. methodology	30	30	57.69%	3	3	15.00%
Testing methodology	0	0	00.00%	0	0	00.00%
Neither testing nor dev. methodology	0	0	00.00%	2	2	10.00%
(DEFAULT) I don't have an opinion	4	17	32.69%	12	12	60.00%
RQ4: Have you ever applied BDD in your daily work?:						
Occasionally read	8	8	15.38%	2	2	10.00%
Tried but abandoned	3	3	05.77%	1	1	05.00%
Used to work on it	5	5	09.61%	0	0	00.00%
Trying but not able to apply	4	4	07.69%	0	0	00.00%
Key stone in dev. practices	0	0	00.00%	0	0	00.00%
(DEFAULT) Never	20	32	61.54%	17	17	85.00%

In Population 2, 60% of the respondents reported having no clear opinion on the topic, while the remaining 40% were distributed across the other response options. Only a small fraction had experimented with or read about BDD. The detailed distribution is presented in Table 2. Comparing the “Answer” and “With Default” columns serves as a sensitivity check. Excluding the default-imputed entries leaves the overall distribution of substantive choices unchanged, confirming that the descriptive findings are not driven by imputation strategy. These findings suggest that most students, who had only recently graduated, had not received formal academic instruction on BDD principles, which likely contributes to the lack of a structured perspective on BDD within their educational programs.

Figure 3 comparatively displays BDD perception assuming default counts.

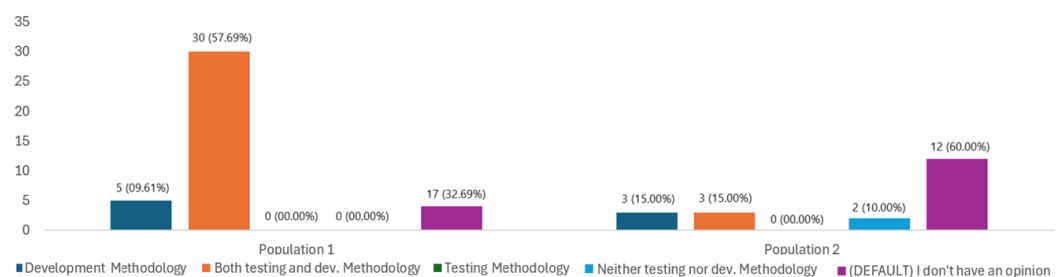


Figure 3. RQ3: BDD perception comparison with default counting displaying the percentage.

Similarly, Figure 4 displays the BDD adoption comparison among the two populations.

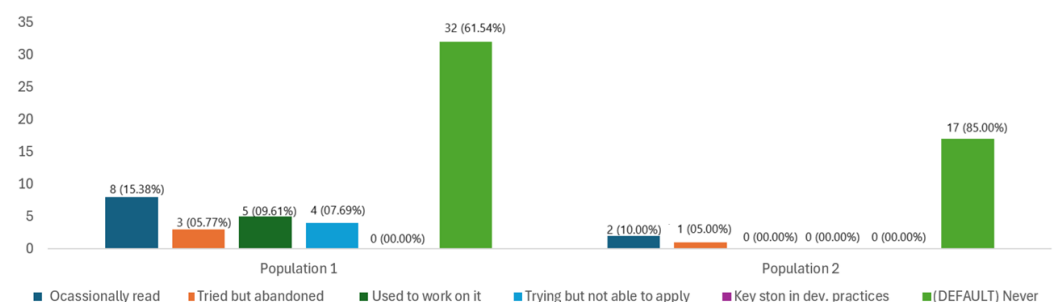


Figure 4. RQ4: BDD adoption comparison between both populations.

2.3. BDD Perception

Two questions assessed how the audience perceived BDD in relation to TDD and ATDD: the potential relationship between TDD and BDD (Figure 5), and the potential relationship between ATDD and BDD (Figure 6). The RQ questions and results are shown in Table 3.

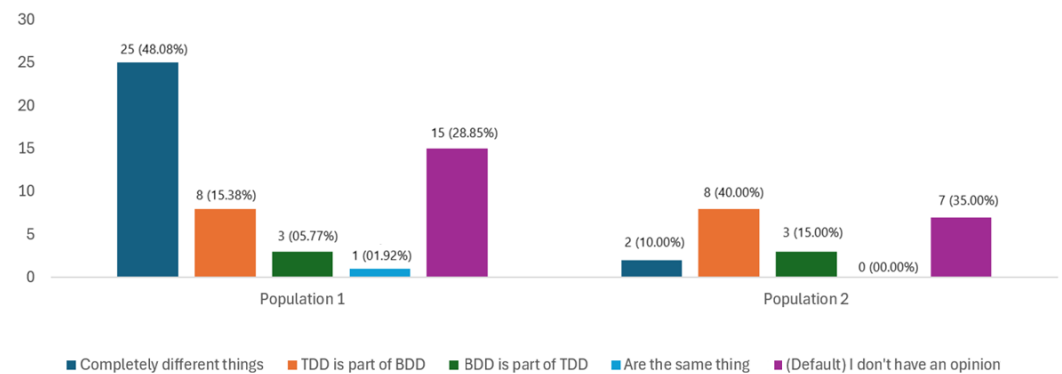


Figure 5. RQ5: How BDD and TDD are related one to the other?

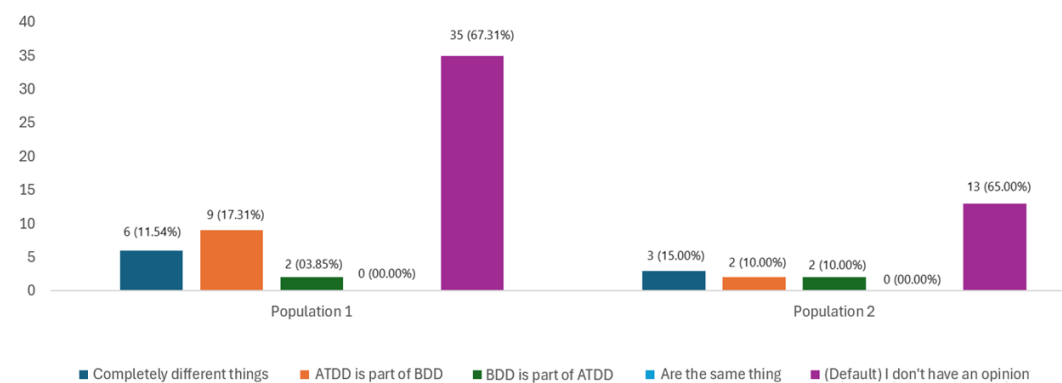


Figure 6. RQ6: How BDD and ATDD are related one to the other?

Table 3. Comparison of BDD, TDD, and ATDD perceptions between P1 and P2.

Question/Category	Population 1 (P1)			Population 2 (P2)		
	Answer	with Default	Percentage	Answer	with Default	Percentage
RQ5: Your perception of BDD and TDD are:						
Completely different things	25	25	48.08%	2	2	10.00%
TDD is part of BDD	8	8	15.38%	8	8	40.00%
BDD is part of TDD	3	3	05.77%	3	3	15.00%
Are the same thing	1	1	01.92%	0	0	00.00%
(DEFAULT) I don't have an opinion	6	15	28.85%	7	7	35.00%
RQ6: Your perception of BDD and ATDD are:						
Completely different things	6	6	11.54%	3	3	15.00%
ATDD is part of BDD	9	9	17.31%	2	2	10.00%
BDD is part of ATDD	2	2	03.85%	2	2	10.00%
Are the same thing	0	0	00.00%	0	0	00.00%
(DEFAULT) I don't have an opinion	6	35	67.31%	13	13	65.00%

In Population 1, the results show a lack of consensus on the understanding of TDD, ATDD, and BDD and their relationships in Population 1. Approximately 50% of the Population 1 respondents perceive a disconnection between TDD and BDD, while a significant proportion either do not consider this or have no opinion. This may be due to intuitive

responses or confusion, as indicated by the high number of non-responses to the question about BDD and ATDD.

Regarding Population 2, the limited presence of structured instruction in academic settings is reflected in the responses from master's degree students. When asked about the relationship between BDD and TDD, the uncommitted default rate was 35.00%, while the relative majority (40.00%) actively indicated that TDD is part of BDD. However, when queried about the relationship between BDD and ATDD, the proportion reporting no opinion remained high (65.00%). These findings are summarized in Table 3.

2.4. Discussion and Research Implications

The survey analysis presented in this section is descriptive and exploratory in nature and does not incorporate formal inferential statistical procedures. The primary objective of the survey is to characterize the extent of conceptual ambiguity surrounding BDD, ATDD, and TDD, rather than to identify statistically significant differences between populations. Given the sample sizes of $N = 52$ and $N = 20$, conventional tests of group differences would be associated with limited statistical power, and assessing convergence (rather than divergence) would necessitate equivalence-testing designs that fall beyond the scope of this exploratory investigation.

The findings from the survey underscore a pronounced absence of consensus concerning the specific definitions and interconnections among BDD, TDD, and ATDD. Notably, approximately two-thirds of the participants affirm their recognition of BDD as a pivotal methodology for driving development processes. However, the majority of these attendees seem to lack substantiated practical experience in applying BDD effectively. This deficit in practical exposure is hypothesized to contribute significantly to the prevailing confusion surrounding how these development approaches relate to one another. Furthermore, even though nearly half of the respondents perceive BDD and TDD as distinctly separate frameworks, there persists a noteworthy segment of respondents who either hold divergent viewpoints or abstain from expressing any opinion. This observation implies that their perceptions could be driven by intuitive inclinations or by an incomplete understanding of the intricate details characterizing these methodologies.

The substantial volume of unanswered questions and the wide range of conflicting opinions regarding BDD and ATDD highlight an urgent necessity for more explicit definitions and structured educational efforts. Although a sensitivity check confirms that unanswered items do not distort the study's descriptive findings, we acknowledge that this strategy cannot definitively separate genuine uncertainty from survey fatigue; thus, future studies should include explicit opt-out mechanisms to disambiguate both states. While this survey is not a formal TAM/UTAUT-based adoption study, the observed absence of structured instruction and operational clarity is consistent with broader adoption concerns related to perceived ease of use and facilitating conditions.

These outcomes reinforce the argument advocating for the development of a systematic approach to BDD that can rectify these misunderstandings and offer a more transparent framework for its adoption and integration into software development practices. This requirement is the fundamental driver behind the BLOW method proposed in this work. BLOW addresses the identified lack of structural clarity by formalizing the boundaries between business-centric requirements and technical implementation, transforming intuitive practices into a repeatable engineering process.

Finally, the survey results show a striking consistency across both professional and academic cohorts. Although responses from students might be subject to inherent cognitive biases due to limited practical exposure, their high degree of alignment with industry practitioners is significant. This convergence suggests that current educational and industrial

models lack a sufficiently robust backbone for understanding BDD and its integration into the software development lifecycle. Such a widespread lack of conceptual clarity underscores the necessity for a systematic method like BLOW. By providing a structured method that can be integrated into both academic curricula and industrial workflows, BLOW enables future and current engineers to move beyond intuitive interpretations, providing the theoretical and practical foundation required to effectively bridge the gap between business specifications and technical implementation.

3. Harmonizing Knowledge Domains Through BDD

To provide a rigorous foundation for the BLOW method, this section establishes a unified conceptual framework for behavior-driven development (BDD) and its constituent practices. We first analyze the systemic “split of knowledge” between business and technical domains, identifying the communication “gray zones” where modern agile projects often fail. Subsequently, we delineate the operational boundaries of Test-driven Development (TDD) and Acceptance Test-driven Development (ATDD). By redefining these frameworks not as isolated practices, but as integrated layers within a BDD-led software development lifecycle (SDLC), this section provides the necessary taxonomy to understand how BLOW automates the transition from natural language specifications to executable technical verification.

3.1. Problem Statement: The Split of Knowledge Domains

In 1994, Hedlund theorized about two types of knowledge [21] developing the concepts stated by Polanyi [22], articulated knowledge and tacit knowledge. Although articulated knowledge can be represented by structured ways of communication, such as written documents, or verbalized instructions, tacit knowledge represents a wider set of knowledge where intuition and experience are the main source of information. Along these two areas of knowledge, a set of different vectors or dimensions was established for the heading of knowledge, individual, group, organization, and interorganizational domain. To illustrate it, in the same work Hedlund (1994) [21] posed the following examples shown in Table 4.

Table 4. Classification of articulated and tacit knowledge at various organizational levels. The table highlights the different types of knowledge and their relevance at individual, group, organizational, and interorganizational levels.

	Individual	Group	Organization	Interorganization Domain
Articulated Knowledge	Knowing Calculus	Quality circle’s documented analysis of its performance	Organization chart	Supplier’s patent and documented practices
Tacit Knowledge	Cross-cultural negotiation skills	Team coordination in complex work	Corporate culture	Customer’s attitudes to products and expectations

Hedlund’s work is instrumental to understand the problem statement. As organizations are becoming larger, the knowledge finds difficulties in mutual understanding in the so called “gray zones” of knowledge. A “gray zone” can be defined as knowledge that cannot fit in a single domain. That means a discipline in a gray zone can not be localized in a specific domain area. In other words, as an organization size increases, there is a greater interdependence among the knowledge treasured by the different domains composing the

organization. At the same time, the speed with which knowledge is increased can limit the amount of information that those domains can share.

The separation of knowledge domains problems is widely documented even in ancient history. As an example, the transition from mythos to logos in ancient Greece could be considered. Mythos (word or speech) was the word used to describe knowledge in homeric poems. The semantic area of Homeric mythos correspond to the area that would later be covered by therm logos [23]). The Homeric poems were written in the eight century b.c. By the times of Plato the term mythos tends to specialize in the sense of fiction, subject of a tragedy. That is, as the knowledge on nature matter was increased, the mythos stopped meaning the knowledge of mankind to become in fictional not covered by the logos. So, in ancient Greece, the priests were focusing on the mythos knowledge through the religion, while philosophers were focusing on the logos knowledge through the philosophy. Several centuries later, the philosophical domain would be split into modern philosophy and science.

Then, the more a specific domain knowledge size increases, the more difficult the exchange of information can be. That also applies to computer science.

By mapping Hedlund's knowledge dimensions onto the SDLC, the structural roles of testing frameworks are clarified: ATDD formalizes the Articulated Business Domain, while TDD ensures the Articulated Technical Domain. However, a "gray zone" often exists where original business intent is lost in technical translation. In this scenario, BDD acts as the bridge for tacit knowledge, capturing the intuition of the "Three Amigos" and translating it into a Unified Language. By externalizing these tacit needs into articulated technical behaviors, BDD ensures the resulting software is both technically robust and functionally aligned with the organizational vision.

3.2. Test-Driven Development

In agile environments, Test-driven Development (TDD) is a evolutionary software development framework where unit test cases are written incrementally before the implementation of any functional code [5,24]. TDD is commonly described as a "test-first" programming practice, requiring developers to introduce new production code only in response to a failing automated test, followed by systematic refactoring to remove duplication and improve internal design [5,25]. Then, TDD is fundamentally enabled by the complementary practices of Unit Testing and Refactoring, which together form a tightly coupled development discipline.

The TDD process consists of the iterative building of unit tests and the corresponding minimal amount of code required to pass these unit tests. Each iteration includes a refactoring of resulting code to generate a clean and sustainable route towards a valid technical solution.

Conceptually, TDD is mainly intended to create defect-free code as fast as possible by developing the minimal required functional code required to pass a specific test suite [14,18]. Consequently, this method yields a comprehensive set of unit tests that validate the precision of the proposed solution (Figure 7).

This characterization is consistent with the definition provided by the Agile Alliance [26]. According to this organization, TDD refers to a style of programming in which three activities are tightly interwoven: coding, testing (in the form of writing unit tests), and design (in the form of refactoring). Design decisions emerge implicitly through continuous refactoring, rather than being specified upfront.

According to Kent Beck [5], the TDD execution follows the following steps:

- TDD starts by writing a single-unit test describing part of the required functionality. This test must fail the first time that is run as the functionality is not implemented yet.

- The developer runs the minimal and simplest amount of code to pass the test.
- When necessary, the code is refactored to optimize it, verifying that the functionality is still working through existing previous tests.
- Finally, the source code is committed to keep the value and revert to the previous state if necessary.
- The cycle is repeated for each new added functionality.

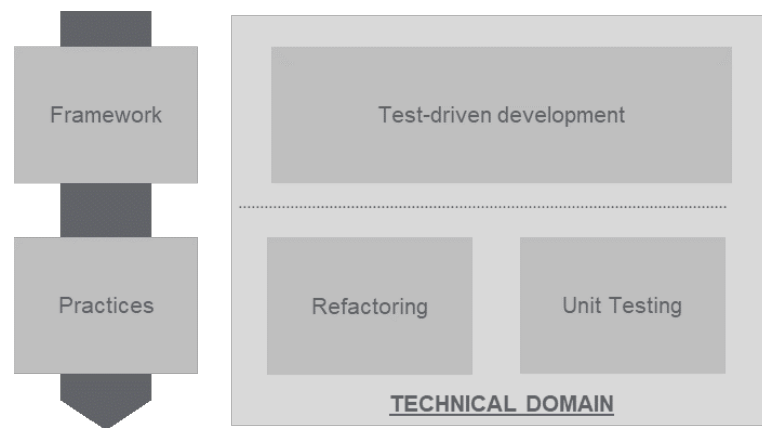


Figure 7. Test-driven Development is a development framework focusing on technical domains enabled both by Refactoring and Unit Testing practices.

This iterative and incremental cycle is depicted in Figure 8, emphasizing the role of TDD as a disciplined mechanism for constructing and validating technical solutions.

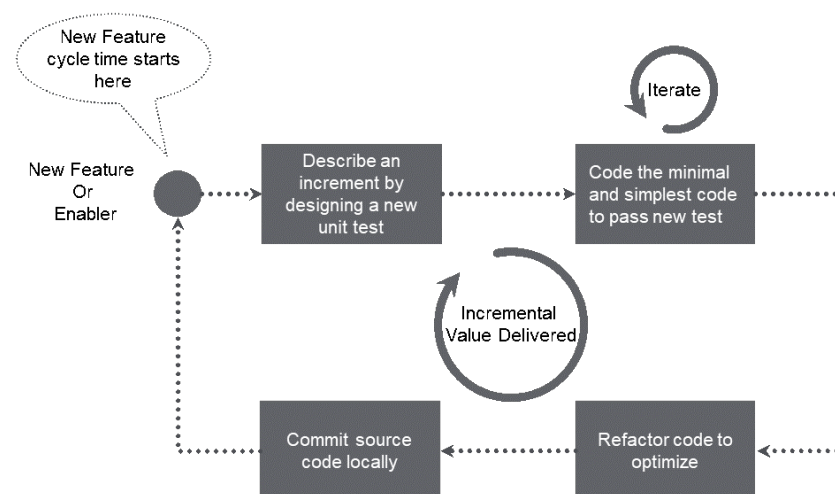


Figure 8. Iterative and incremental cycle enabled by Test-driven Development.

Consequently, TDD serves as a robust working framework for verifying and simplifying the coded solutions. However, it does not inherently address the acceptance criteria defined by stakeholder, nor does it validate whether the implemented functionality satisfies business intent [14,18]. Therefore, TDD is strictly focused on the technical domain and supports disciplined verification rather than validation to ensure that the solution is built right (verification) [15]. Table 5 summarizes the role of TDD within this context.

Table 5. Main area of TDD is the Technical Domain which enables verification.

Framework	Domain	Enabled Process
TDD	Technical Domain	Verification Process

The benefits of TDD are well-documented in the literature. Empirical studies associate TDD with improved code maintainability, enhanced program comprehension, and reduced defect injection rates, without significantly increasing overall development time when assessed over the full lifecycle [24,27]. Furthermore, controlled experiments by Williams et al. [28] report defect reductions of up to 40%, even in teams with limited prior TDD experience. These results reinforce the role of TDD as a robust technical foundation for larger and more comprehensive development method such as BLOW.

3.3. Acceptance Test-Driven Development

As TDD is primarily focused on unit testing within the technical domain, its scope is inherently limited with respect to validating business requirements. In fact, Soeken et al. assert ‘While unit tests check the correct implementation of single atomic components in the software, acceptance tests check a scenario of the system as a whole without considering how the system is actually implemented’ [29]. Consequently, TDD ensures the technical integrity of the code but lacks the necessary link to the business domain required to ensure functional validity.

Bridging this gap is the main scope of Acceptance Test-driven Development (ATDD). In ATDD, stakeholders, developers, and testers collaboratively define acceptance tests before implementation begins, thereby making business requirements explicit and executable early in the development lifecycle [30]. These acceptance tests describe system behavior from the user’s perspective and serve as a contract that guides subsequent development activity.

A central practice enabling ATDD is the Three Amigos. This collaborative practice brings together representatives from the business, development, and testing domains to collectively explore requirements and define acceptance tests prior to implementation. Wang and Gómez [31,32] emphasize the importance of the Three Amigos as a socio-technical coordination mechanism that improves shared understanding and reduces ambiguity between business intent and technical realization. Similarly, Aggarwal [33] characterizes the practice as “first writing acceptance test cases for a solution or product, and then using those tests to drive development.” Figure 9 summarizes the focus of ATDD as a framework operating primarily within the business domain.

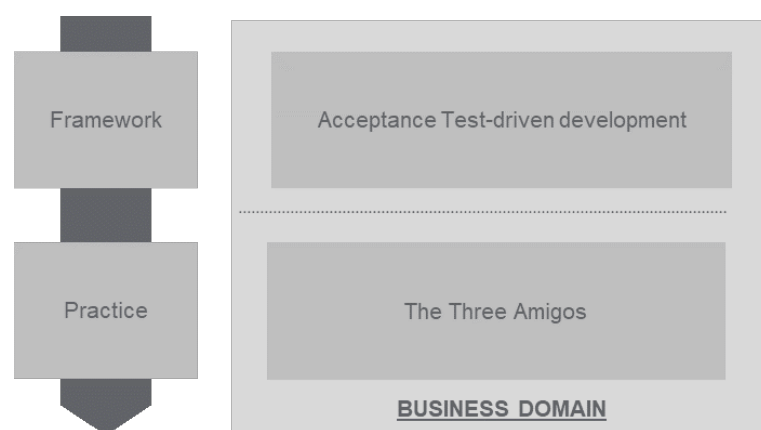


Figure 9. Acceptance Test-driven Development is a development framework focusing on business domains enabled by the Three Amigos practice.

As indicated by Soeken et al. [29], it is clear that ATDD and TDD operate at different levels of abstraction and serve complementary roles. Unambiguously, the engagement of stakeholders in ATDD aligns with a business perspective, whereas engineer involvement in TDD aligns with a technical perspective. This means that stakeholders direct ATDD towards business-centric areas, while test engineers steer TDD towards technology-centric areas. Therefore, ATDD emerges as an appropriate framework for supporting business-centric domains.

The distinction between ATDD and TDD is fundamental: the involvement of stakeholders in ATDD ensures a business-centric perspective (validation), whereas the engineer's focus in TDD ensures a technology-centric perspective (verification). Therefore, ATDD functions as the operational bridge for the articulated business domain. Table 6 details the specific roles and focus areas of each team member within this collaborative framework.

Table 6. Three Amigos roles and their main focus during the practice.

Perspective	Focus	Voice of Role
Business Oriented	To define the problem to be solved	Stakeholder
Development Oriented	To define a technical solution	Technical
Testing Oriented	To challenge the solution	Tester

By establishing acceptance tests early through the Three Amigos practice, ATDD integrates validation activities into the initial stages of development. This early incorporation reduces the number of feedback cycles required with stakeholders and limits rework caused by misinterpreted requirements. Figure 10 illustrates the application of the Three Amigos practice in integrating acceptance tests within the development lifecycle. Once the acceptance tests have been defined, the technical team (e.g., developers) can proceed with the implementation in accordance with the aforementioned tests. This integration ensures that validation is not a final, disconnected phase, but a continuous driver of the development lifecycle.

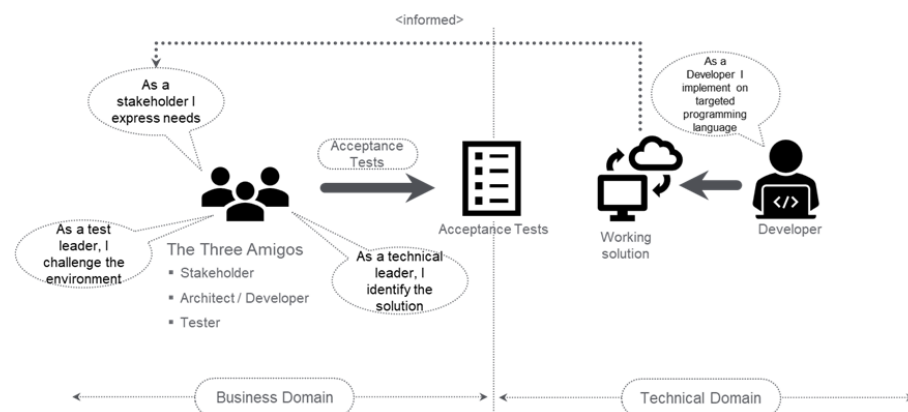


Figure 10. In Acceptance Test-driven Development, acceptance tests are defined prior the development and incorporated into the development lifecycle in an early stage.

3.4. Behavior-Driven Development

The BLOW method is primarily grounded in the definition of behavior-driven development (BDD) provided by Axelrod [34], who describes BDD as ‘a methodology that is derived from Test-driven Development but adds to it an emphasis on bridging the gap between the natural language description of the behavior of a feature [...] and the tests that verify that the feature indeed behaves according to description’. The concept, originally introduced by Dan North [6], proposes redefining tests as behaviors (structured as scenarios or examples) to describe the functional behavior of the code to be tested. Consequently, the effective adoption of BDD is achieved by defining the desired set of solution behaviors necessary to overcome a specific engineering problem.

As defined by Dan North, a scenario is intended to define the behavior of the solution in response to specific preconditions when an event is triggered. From this distinction, BDD can be considered an improvement (in Design Science, an improvement is a new solution for a known problem) of TDD. This defines a common language understandable among

the different participating roles (technical and nontechnical) to express the behavior that a software system should do.

According to the Agile Alliance, BDD, also known as Specification by Example, is defined as “a synthesis and refinement of practices stemming from Test-driven Development (TDD) and Acceptance Test-driven Development (ATDD)” [35]. This perspective is crucial as it positions BDD as the overarching methodology that orchestrates TDD within the technical domain (verification) and ATDD within the business domain (validation), as summarized in Table 7. From a technical perspective, BDD synthesizes the basic principles of TDD with object-oriented analysis and domain-driven design with the aim of optimizing and maximizing the effectiveness of the software development process [35].

Table 7. Functional domains and processes coordinated by BDD.

Behavior-Driven Development		
Framework	Domain	Enabled Process
TDD	Technical Domain	Verification Process
ATDD	Business Domain	Validation Process

In this work, the distinction between methodology and framework is used to separate the coordinating logic of BDD from the supporting structures provided by TDD and ATDD. Accordingly, BDD is treated as a methodology, while TDD and ATDD are treated as frameworks that support specific verification and validation activities [36–39].

To complete this taxonomy, two further levels are introduced. A practice denotes a specific, repeatable pattern of interaction or execution carried out by defined roles. Examples include the Three Amigos and Technical Three Amigos practices, which organize collaboration between business and technical stakeholders. A tool represents a software artifact that automates or support one or more such practices. Examples include Cucumber and SpecFlow, which support the automated execution of scenarios created during BDD practices. This four-tier structure (methodology → framework → practice → tool) forms the conceptual foundation for the organizational scheme encoded in BLOW. Consequently, BDD is treated in this work as a methodology supported by the TDD and ATDD frameworks, as it is illustrated in Figure 11.

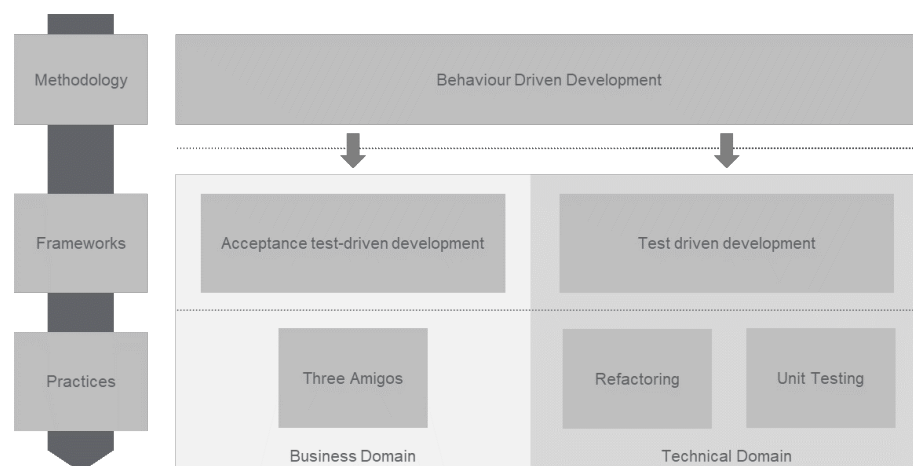


Figure 11. BDD as methodology, ATDD and TDD as frameworks, and their practices mapped against business and technical domains.

In all cases, BDD encourages the use of an ubiquitous natural language driven to cover the needs of stakeholders that can be implemented in the software system. These scenario

descriptions, due to the nature of the way in which they are written, represent a way to reduce miscommunications between stakeholders and technical teams. As a consequence, ambiguities that traditionally result in miscommunication between the business domain and the technical domain are reduced, and stakeholder satisfaction can meet expectations.

Traditional interpretations of BDD often face a dilemma similar to the chicken-and-egg situation. On the one hand, implementing ATDD scenarios derived from the Three Amigos practice necessitates a technical solution for integration. On the flip side, a technical solution requires business relevance to be impactful. Irshad et al. documented this problem [40], noting that BDD often fails at large-scale since no product development is done yet. Their analysis of theoretical BDD-related works demonstrated the absence of industrial evaluation within the community. They then propose a systematic approach whereby technical scenarios (system-level feature files) are introduced to align with business scenarios (behaviors in the original). This systemic gap—the lack of industrial evaluation for large-scale BDD—is precisely what the BLOW method seeks to address by introducing a layered organizational approach with work-centers.

The application of BDD is usually a complex task because it requires seamless collaboration of business and technical teams to derive test cases and their corresponding implementations.

The Operational Cycle: Discovery, Formulation, and Automation

The practical application of BDD is traditionally organized into three iterative stages, as proposed by Rose and Nagy [41]: Discovery, Formulation, and Automation.

Discovery

This stage focuses on collaborative exploration. Business stakeholders and technical teams engage in “Three Amigos” conversations to agree on the desired behavior of a feature [42]. The output is a set of acceptance criteria categorized by business rules. In Figure 12) an example of the outcome of a discovery session is shown for the functionality of a shopping cart.

```
Rule: A persona can check in products without purchasing
Scenario: A persona can check in a single product
Scenario: A persona can check in the same product several times
Scenario: A persona can also check in several products

Rule: A persona can purchase a previously checked in product
Scenario: A persona purchases a checked in product
Scenario: A persona shall checked in at least one product to purchase
```

Figure 12. Example of the outcome of a discovery session for the functionality of an e-commerce cart.

Formulation

Once rules and scenarios are approved, the writing of the scenario is initiated in Formulation phase. In this stage, scenarios are articulated in a specific language, such as Gherkin, which is widely recognized as the industrial standard. Due to its extensive use in BDD, many studies closely associate Gherkin with BDD. For example, Tianyao Li et al. [43] defines Gherkin as the BDD test specification language.

Gherkin provides a set of grammar rules (<https://cucumber.io/docs/gherkin/>, accessed on 17 June 2026) to define scenarios in terms of the following reusable steps:

- Given steps: These set the initial context for a scenario, for example; Given my bank account has a balance of 3400 USD.
- When steps: These define or trigger an action or sequence of actions that set the scenario in motion, for example; When I withdraw 300 USD cash.

- Then steps: These specify the expected outcomes, for example; Then after the withdrawal my bank account will have a balance of 3100 USD.

Furthermore, Gherkin utilizes background conditions to define the relevant information about the environment that can impact the scenarios within a feature. Despite background conditions can appear during the discovery stage, they are formalized here to ensure consistency across the test suite. In Figure 13 the background conditions for the execution of a specific scenario are illustrated.

```

Feature: Basket cart
As a customer
I want to get information about checked in products in cart
So I can know the total cost of my purchase
Background
  Given the catalogue has following products
    | product | price | currency | discount |
    | motion-cam-hero-09-2020 | 10 | EUR | 10% |
    | motiom cam-hero-10-2023 | 10 | EUR | 0% |
    | phone-hero-13-2024 | 1342 | EUR | 0% |
    | phone-hero-12-2021 | 999 | EUR | 5% |
  And these personas are registered
    | Persona | Fidelity discount | Preferred currency |
    | David | 0% | EUR |
    | Maria | 20% | EUR |
    | Paul | 0% | USD |
    | Ushiro | 10% | JPY |
    | Jules | 5% | JPY |
  And the following exchange rates at the time of operation
    | From currency | To currency | Rate |
    | EUR | USD | 1.134 |
    | USD | EUR | 0.881 |
    | USD | YPD | 114.386 |
    | JPY | USD | 0.009 |
    | JPY | EUR | 0.008 |
    | EUR | JPY | 129.737 |

Rule: Persona can check in products without purchasing

Scenario: A persona can check in a single product
  Given I am David
  And Iam having an empty cart
  And I check in a product 'motion-cam-hero-10-2023'
  When I list checked in products
  Then there will be a single product with code 'motion-cam-hero-10-2023'
  And cart total will be 10 EUR
  
```

Figure 13. The first scenario of Figure 12 is developed here. Notice that the background conditions include conditions from other scenarios which are not displayed.

The formulation stage requires specific Gherkin syntax knowledge and automation skills. For this reason, the Gherkin formulation is usually written by testing engineers, although optionally stakeholders and customers can contribute to scenario writing. The resulting specifications must be validated by such stakeholders in two ways, by involving the stakeholder during the writing process or by getting stakeholder's approval offline, to ensure the "Executable Specification" remains faithful to the business intent.

Automation: BDD Enabled by Gherkin-Supported Tools

After developing a collection of scenarios in a designated language such as Gherkin, these scenarios are then mapped to code functions (step definitions) using BDD frameworks such as Cucumber, SpecFlow (C#), or Behave (Python), which are tailored to a particular programming language. This process automates the creation of both unit and acceptance tests, ensuring that every automated verification directly tracks back to a validated business requirement.

Figure 14 provides an example of a Python function decorated with Behave annotations. These functions execute assertions using standard unit testing frameworks (such as xUnit or PyTest). Since the Gherkin scenario prescribes a strict execution order, the corresponding functions are carried out in that exact sequence, maintaining the integrity of the behavioral flow.

```

from behave import given, when, then

@when('the search phrase "{string}" is entered')
def _the_search_phrase_is_entered(context, string):
    #TODO: Implement code to execute a search
    pass

```

Figure 14. Step definition in Python (Behave) illustrating the binding between Gherkin steps and executable code.

The reusability during the automation is one of the significant advantages of Gherkin-supported tools. Indeed, once the steps have been linked to their corresponding functions, alternative scenarios can be defined without writing additional code, as shown in Figure 15.

```

Feature: Google searching
  As a web surfer,
  I want to search Google,
  so I can learn new things.
Scenario: Searching for panda
  Given a web browsers is on the Google page
  When the search phrase "panda" is entered
  Then results for "panda" are shown
  And the related results include "Panda Express"
  But the related results do not include "pandemonium"

Scenario: Searching for chaos
  Given a web browsers i on the Google page
  When the search phrase "chaos engineering" is entered
  Then results for "chaos engineering" are shown
  And the related results include "Home of Chaos Monkey"

```

Figure 15. Reusing existing steps to create more tests to cope with different scenarios based on illustrative scenario from <https://automationpanda.com/2017/01/27/bdd-101-gherkin-by-example/> (accessed on 17 June 2026).

By securing stakeholder agreement on these automated tests, these tests are inherently correct. This suggests that the mechanism is applicable at both the verification and validation stages. Consequently, tools that support Gherkin ensure that verification outcomes can be extended to validation outcomes, thereby anticipating the validation results and reducing the risk of defects in delivered features. A significant consequence of automating these tests is the subsequent automation of acceptance tests, which in turn enables the automation of procedures for validation.

3.5. BDD Tools

3.5.1. Gherkin as Domain Specific Language

Gherkin is widely recognized for its effectiveness in the development of scenarios. The objective of Domain-Specific Languages (DSLs) is to improve the efficiency of software developers by removing the need for repetitive, low-level boilerplate code [44]. In this context, Gherkin serves as a structured DSL to outline the behaviors required to address both user stories and technical stories. Essentially, the validation criteria require the adherence to these scenario-based specifications. Numerous scholars have identified Gherkin as an essential element of BDD. For instance, Saeed Parsa's 2023 publication [45] describes BDD as a novel approach that employs the Gherkin language to deliver executable specifications. The syntax of Gherkin is notably effective for the implementation of BDD practices. It relates to the problem statement (Section 3.1) by facilitating clearer communication between technical and nontechnical stakeholders, ensuring a mutual understanding of the project objectives. Gherkin, defined within the Cucumber testing framework, enables the depiction of BDD specifications through scenarios in any language, including plain English.

Then, Gherkin is the most prevalent natural language-based format for the implementation of technology-agnostic scenarios. Consequently, Gherkin can be regarded as both a test specification language (TSL) [46] and a domain specification language (DSL) [47]. This enables technical and non-technical users to reach a consensus on a functional specification

or behavioral specification and its acceptance criteria, thereby ensuring a common understanding. Conversely, Gherkin can also be employed by technical domain actors to define the behavior of their solution, thereby facilitating the formulation of unit tests for Enablers, otherwise known as technical stories.

The structure of Gherkin consists of three primary entities: Features, Scenarios, and Business Rules (Figure 16). A feature describes the behavior of a solution to cover a need defined by a story or a technical story; in other words, it states what the software needs to do and how it should behave under different circumstances. An example of a feature is the display of the bank account balance associated to a debit card in an automated teller machine (ATM).

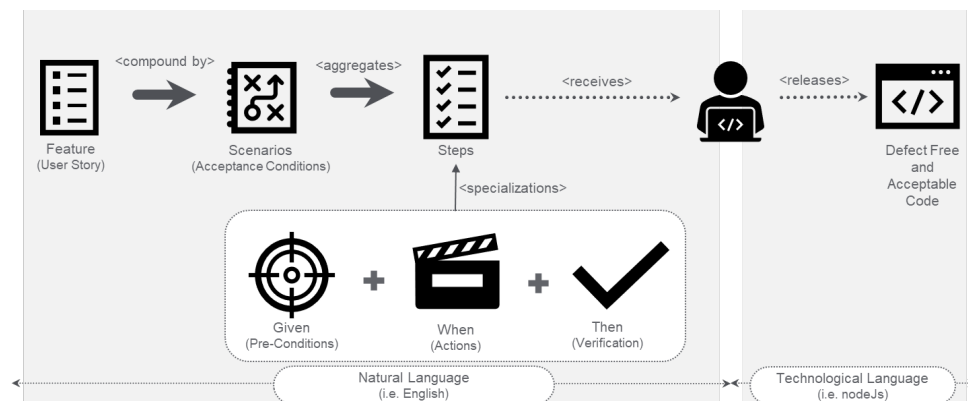


Figure 16. Hierarchical structure of Gherkin specifications: Features, Rules, and Scenarios.

A feature is typically composed of one or more scenarios, which are stored within a designated feature file. Each scenario describes the necessary acceptance conditions and the actions required to achieve the desired outcome. Consequently, a feature narrates a story, and the scenarios detail the acceptance conditions of that story related to the feature. These scenarios are often subject to the constraints of business rules. Adhering to these rules enables the organization of the resulting scenarios. For example, a business rule for ATM machines states that the machine is unable to dispense cash if it is out of money. This leads to two potential scenarios: firstly, a message is displayed to the user indicating the location of the nearest available ATM; secondly, the ATM transmits a notification to a monitoring center.

Furthermore, each scenario is structured into a series of repeatable steps that follow specific grammatical guidelines. These steps outline the necessary actions or behaviors. Given their applicability to a multitude of different scenarios and features, it is possible to reuse scenarios and steps. For instance, a step requiring the user to input a valid PIN at an ATM can be employed in both checking account balances and withdrawing cash, as both processes need PIN verification.

Steps included into a scenario have been split into the following keyword blocks:

- Given, including steps to establish preconditions of actions to be executed.
- When, expressing the conditions taken by actions to be executed.
- Then, indicating the definition of the expected outcome(s) from the executed actions.

In Figure 17 there is a specification of a Gherkin example for a simple scenario for a specific feature. The story is described by specifying a behavior or a scenario.

Gherkin specifications can be effectively automated for BDD by converting Gherkin scenarios into executable specifications through specific tools such as Cucumber. Cucumber is the most popular tool because it is closely linked to Gherkin, as it was originally released as part of Cucumber. In fact, Gherkin scenario steps can be associated with code functions that developers may enhance. These code functions transform into executable

specifications that establish preconditions with *Given* steps, perform actions in *When* steps, and evaluate the results in *Then* steps. Integrating these functions creates executable tests in a technological language, maintaining a link between the Gherkin specifications and the results of the tests. Tools compatible with Gherkin enable the automation of BDD.

Popular tools, such as Xray for testing management in Jira, include Gherkin to define tests cases and assign them into test plan, enabling traceability between Stories and their acceptance criteria.

```

Feature: User login
  As a user
    I want to log in to the application
    To access my personalized dashboard
  Scenario: Successful login with valid credentials
    Given the user is on the login page
    When the user enters a valid username and password
    And clicks on the login button
    Then the user should be redirected to the dashboard page
  
```

Figure 17. Illustrative Gherkin feature file for a login scenario.

3.5.2. Enabling BDD Through Tools

There are multiple BDD tools in the market ready to enable BDD automation. Table 8 shows an extensive list of these tools. Many of them are specific to a technology such as JBehave and JDave that is oriented specifically to Java technologies. However, BDD is mainly technology agnostic, because the methodology can be applied regardless of the supported technology. Then, BDD tools can usually be supported by different programming languages. According to the study by Binamungu et al. [49], the most popular tools best suited to BDD were Cucumber, FitNesse, JBehave, Concordion, Spock, and Specflow as shown in Table 8.

Table 8. Key BDD automation frameworks and their technological alignment.

Tool	Technology	Web Page
Cucumber	Java, nodeJs, Python, Go, C#, C++ and others	https://cucumber.io/ (accessed on 17 June 2026)
JBehave	Java	https://jbehave.org/ (accessed on 17 June 2026)
FitNesse	Java, nodeJs, Python, Go, Scala, C#, C++	http://docs.fitnesse.org/ (accessed on 17 June 2026)
JDave	Java	https://github.com/jdave/JDave (accessed on 17 June 2026)
Concordion	Java, C#	https://concordion.org/ (accessed on 17 June 2026)
TestLeft	C#, Java, Visual Basic.NET	https://smartbear.com/product/testleft/overview/ (accessed on 17 June 2026)
Spock	Java, Groovy	https://spockframework.org/ (accessed on 17 June 2026)

A trending search in Google Trends for these BDD tools [48], as shown in Figure 18, reveals similar results to the study conducted by Binamungu et al. [49]. While some of these tools (such as JDave, Concordion, and TestLeft) appeared to be marginal compared to the others, Cucumber emerged as the most popular one.

Different specification languages can be found in the BDD tools to define scenarios. However, Gherkin is usually the most used by tools such as EasyB, JBehave, TestSpec and Cucumber. For instance, Cucumber is composed of a set of packages with a common architecture which implements the principles of the BDD methodology in different programming languages, such as Java, Javascript, Python, Go, C# and others. The maintained implementations are classified as *official*, *semi-official*, and *unofficial* (<https://cucumber.io/docs/installation/> accessed on 17 June 2026). *Official* implementations are hosted directly by the Cucumber community. *Semi-official* ones, implement Cucumber standards which are recognized by the Cucumber community. Finally, *unofficial* ones claim to become a Cucumber standard implementation, but without the recognition of the Cucumber community.

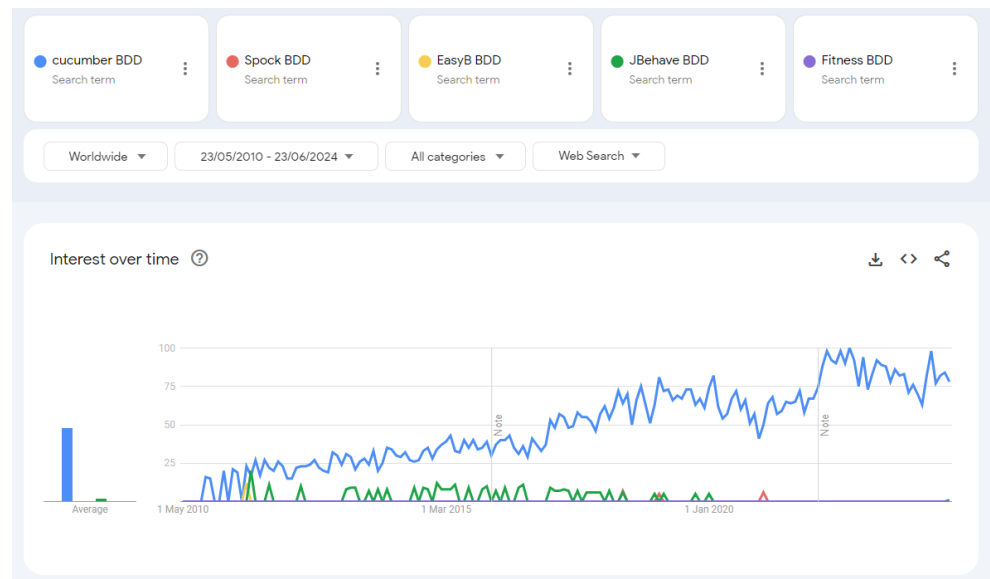


Figure 18. Trend analysis identifying the market dominance of identified BDD tools (2010–2024).

Despite the robustness and multi-language support of the tools identified in Table 8, their implementation in industrial environments often lacks a clear organizational structure, leading to the “gray zones” and “chicken-and-egg” dilemmas previously discussed. Although frameworks like Cucumber provide the “how” (automation), they do not prescribe the “where” or “who” within a complex SDLC. This is where the BLOW systematic approach intervenes. Rather than proposing a new automation tool, BLOW establishes a structured orchestration of these existing frameworks. By distributing TDD and ATDD practices across specialized work-centers and Layers, BLOW provides the necessary governance to ensure that tools like Cucumber or SpecFlow are utilized not just for testing, but as a systematic bridge for knowledge externalization.

The maintenance of large Gherkin test suites represents a practical challenge that has been widely documented in the literature. Binamungu et al. [49] observed that development teams frequently struggle with the growing volume of feature files in fast-changing agile environments, leading some organizations to gradually deprecate or completely abandon BDD practices. This maintenance bottleneck is inherent to Gherkin-based specifications and to large-scale test datasets in general, and it is not fully resolved by any current approach. However, the BLOW approach mitigates the organizational dimension of this problem by providing a structured mapping between automated scenarios, work-centers, layers, and domain contexts, facilitating their modular classification and lifecycle management. While this governance structure does not eliminate the underlying maintenance burden of large scenario suites, it reduces cross-team coordination overhead and helps teams locate, reuse, and evolve scenarios within clearly bounded responsibilities.

4. The BLOW Approach: A Systematic Approach to BDD

The BLOW approach (Behavior-driven development in a Layered Organization of Work-centers) is proposed as a systematic approach to orchestrate software delivery in high-complexity environments. Its primary objective is to address the persistent decoupling of business and technical knowledge domains, transforming BDD from an isolated testing technique into a comprehensive methodology for value-stream governance. Although BDD provides a conceptual bridge between business requirements and technology, its industrial application often suffers from a lack of organizational governance in large-scale engineering. BLOW addresses this by structuring the delivery lifecycle into specialized execution units, governed by four core pillars.

To operationalize this governance, the BLOW is structured as a layered operating model, defining how responsibilities, execution boundaries, and transformation flows are distributed across multiple levels of abstraction within the software development lifecycle.

As a structured approach, BLOW is designed to accelerate the creation of efficient and effective software systems through the application of BDD and its associated frameworks within organizational contexts. This orchestration integrates ATDD and TDD into a unified workflow, ensuring that every technical activity, from initial discovery to final implementation, directly traces back to a strategic business objective. This integration is governed through scenario-driven execution, where business and technical scenarios act as first-class artifacts that guide development progress and decision-making. This approach ensures seamless compatibility with established advanced software development models while maintaining a rigorous focus on the behavior of the system.

By organizing delivery around executable business and technical scenarios, BLOW establishes the basis for scenario-driven observability. The approach makes development progress traceable through the definition, refinement, and execution of scenarios across the layered operating model, enabling the later definition of native metrics for reasoning about project state, technical effort, and value realization. Accordingly, BLOW is presented here as a structured measurement and coordination mechanism, while the empirical behavior of quality-related indicators is examined later in the case study.

The core conceptual structure of BLOW combines layered organizational governance with scenario-driven development as the fundamental execution paradigm. This is summarized in Figure 19.

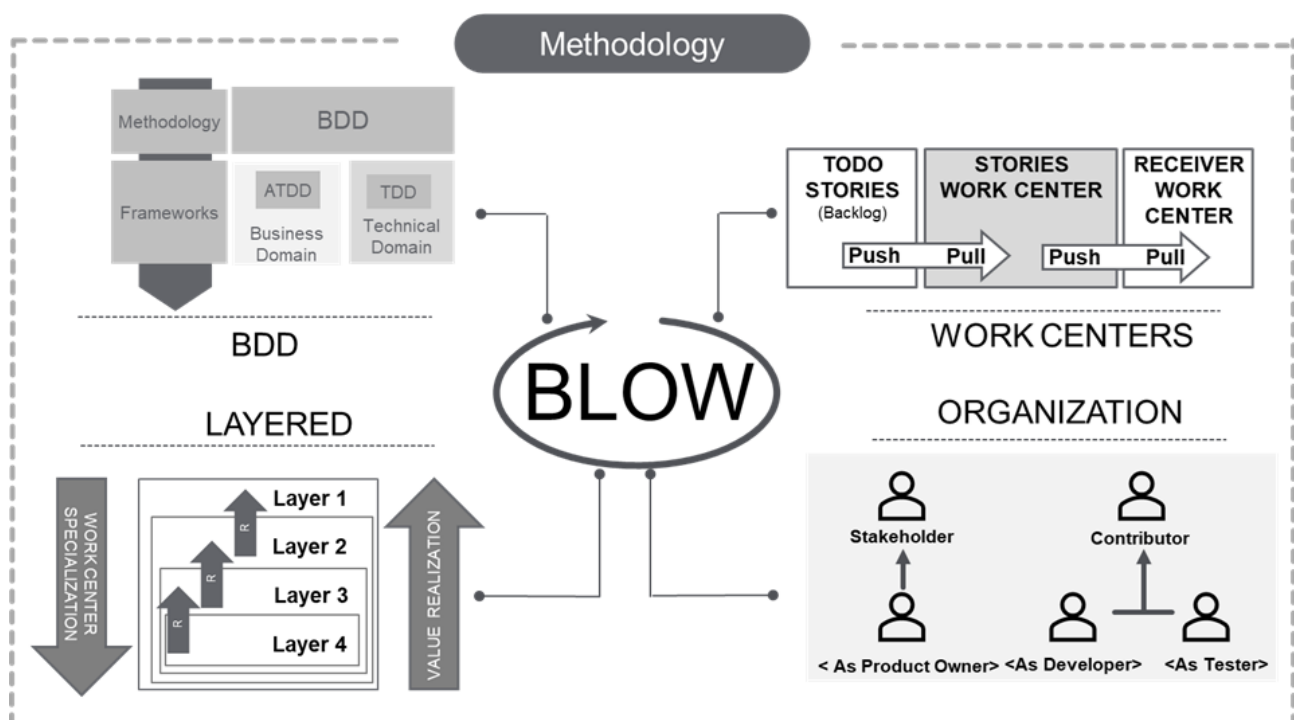


Figure 19. The foundational principles of BLOW are encapsulated in a set of core values. Central to the BLOW is the BDD approach, which serves as a guiding inspiration. Additionally, the organization incorporates a layered structure emphasizing value realization and work specialization. The establishment of work centers as distinct domains of accountability, coupled with robust organizational principles, further enhances this operating model.

4.1. Foundations and Core Principles

BLOW is based on four core principles that align development practices with organizational business goals: Behavior-driven development, Layered coordination, Organization,

and Work centers. Together, these principles provide a systematic structure for delivering software that provides demonstrable value, ensuring that every technical activity remains synchronized with the overall strategic objectives (Figure 19). Collectively, these principles define BLOW not merely as a set of practices, but as a layered operating model in which responsibilities, decision boundaries, and execution flows are explicitly structured across the software development lifecycle.

Behavior-driven (B) development is the core inspiration of the approach. It shifts the focus from isolated implementation details to the overall behavior and actions of the system from a user-centric perspective. By establishing a common, unambiguous language, BDD enhances communication and collaboration between developers, testers, and business stakeholders. This alignment ensures that requirements are not just documented but understood as executable contracts of intent, eliminating the traditional “gray zones” between business needs and technical delivery. Within BLOW, these executable contracts are expressed concretely as Scenarios, which become the primary mechanism through which behavior is specified, validated, and realized throughout development.

Layered (L) coordination in software development provides the structural hierarchy necessary to manage complexity in large-scale engineering. In BLOW, a “layer” refers to a nested organizational structure where activities and responsibilities are distributed across distinct levels of abstraction. This approach, rooted in Lean principles, allows teams to streamline processes, minimize waste, and eliminate non-essential features. By organizing development into layers, the operating model ensures that the transition from a broad business idea to a specific technical verification is governed, efficient, and supports informed decision-making at every level. These layers constitute the backbone of the BLOW operating model, ensuring that each transformation step operates at a well-defined abstraction level while remaining traceable to validated business behavior.

Organization (O) revolves around the strategic concept of “Unit of Value” (UOV). BLOW emphasizes the decomposition of complex features into the smallest self-contained units of functionality that deliver direct value to the user. By identifying and prioritizing UOVs, as integral components of the value stream, the approach ensures that the development effort is always focused on the most impactful features. This organization-oriented approach guarantees that the resulting software provides real-world utility and remains strictly aligned with evolving business objectives. In BLOW, Scenarios are identified as the fundamental Units of Value, establishing a scenario-driven development paradigm in which progress, cost, and value realization are governed by the completion of executable behavior.

Work centers (W) represent the specialized domains of accountability where value is physically realized. Derived from Lean Manufacturing, a work-center in BLOW is a designated workspace where specific activities transform inputs (termed Work Items) into outputs with incremental value. BLOW manages these work-centers within the layered coordination framework, creating an interconnected chain through which work flows via “push/pull” dynamics; the output of one work-center becomes the necessary input for the next. This structure provides a governed environment for development and testing, supported by a clear chain of command and optimized communication channels. By assigning specific scenario transformations to dedicated work-centers, BLOW ensures that scenario execution is both governed and observable across organizational boundaries.

The integration of these four pillars is intended to enhance system visibility and encourages the formation of business-oriented self-organizing teams. By incorporating specifications as executable tests early in the lifecycle, BLOW supports the continuous automation of end-to-end processes. This systematic transformation of user stories into working code (Figure 20) supports disciplined development through executable specifications, scenario-driven traceability, and test-anchored feedback. This provides an observable

structure for monitoring progress, technical effort, and value realization through scenarios and their associated tests, while the empirical behavior of quality-related indicators is analyzed later in the case study.



Figure 20. The approach consumes Stories (also known as user stories) to incrementally deliver Working Software.

4.2. Structural Components and Operational Taxonomy

To facilitate the systematic transformation of requirements into working code, BLOW operates with a structured set of fundamental entities. These components define what is being transformed, how its value is measured, and where the transformation occurs. These structural building blocks required by BLOW, which will later be composed and orchestrated by the BLOW layered operating model.

4.2.1. Work Item Taxonomy: Features, Stories and Scenarios

The fundamental unit of production in BLOW is the Work Item (W). A Work Item is defined as any entity that carries potential value and is subject to transformation within the system. To manage the complexity of high-scale engineering, Work Items are organized into a clear taxonomy based on their nature and their level of decomposition.

Within this hierarchy, the Feature (F) represents the first level of functional or technical intent. A Feature is defined as any item with a behavior that can be described in natural language terms and that is intended to increment the value of a software product. Following the Scaled Agile Framework (SAFe) [50,51], features are first classified into two disjoint sets:

- Stories or User Stories (U): Small pieces of desired functionality written in the user's language, representing an increment in business value [50].
- Enablers (E): Technical Work items intended to support the activities needed to extend the "Architectural Runway," providing the necessary infrastructure or architecture for future business functionality [51]. In BLOW, the targeted Enablers are restricted to technical items whose acceptance criteria can be formalized as technical scenarios; non-executable activities (such as research spikes) remain essential to the lifecycle but are governed outside BLOW's metrics framework.

Let P a product, and $U(P)$ the set of Stories of P :

$$\forall P, \exists U(P) \quad (1)$$

Similarly, $E(P)$ is the necessary set of Enablers enabling those Stories:

$$\forall P, \exists E(P) \quad (2)$$

The set of Features $F(P)$ is the union of $U(P)$ and $E(P)$:

$$\forall P, \exists F(P) : F(P) = U(P) \cup E(P) \quad (3)$$

However, $E(P)$ and $U(P)$ are conceptually separated sets, since they represent distinct domains of knowledge and responsibility; thus:

$$U(P) \cap E(P) = \emptyset \quad (4)$$

The taxonomy is completed by the Scenario (*S*), which represents the most granular level of a Work Item. Scenarios are generated from features during the BDD discovery stage (see Section The Operational Cycle: Discovery, Formulation, and Automation) and serve as the definitive acceptance conditions. Consequently, a Story is a feature defined by Business Scenarios, whereas an Enabler is a feature defined by Technical Scenarios. By introducing Scenarios at this stage, BLOW establishes a uniform abstraction for expressing behavior, postponing decisions about implementation flow and organizational responsibility to later stages of the approach.

By establishing the Work Item as the root concept and the Feature as the behavioral unit, BLOW ensures that every element in the backlog, regardless of its scale or domain, is treated as a traceable, executable unit of production. Under this representation, backlog management is grounded on the identification and evolution of Scenarios rather than on task decomposition.

4.2.2. Unit of Value Items vs. Incremental Items

A fundamental premise of BLOW is the formalization of how value is quantified and tracked within the software development lifecycle. To achieve high levels of observability, BLOW distinguishes between the atomic units that constitute progress and higher-level items that accumulate this progress. Work Items (*W*) are categorized based on their capacity to realize value into two distinct types: Unit of Value items and Incremental Items.

The Unit of Value (UOV) Item is an atomic, measurable item that serves as the standard to evaluate the realized value of larger elements. In BLOW, the Scenario (whether Business or Technical) is established as the fundamental Unit of Value. These are “doable” elements. This means that they possess a binary nature, either they are implemented and validated, or they are not. Once a Scenario is “Done”, it provides a fixed, non-reducible increment of validated behavior.

In contrast, Incremental Value Item is a high-level work item whose total value is the aggregate of its constituent Units of Value. In this approach, Stories and Enablers are categorized as Incremental Value Items. Unlike scenarios, they are “completable” elements; their total value is not static but fluctuates and grows as new Scenarios are discovered, refined, or modified during the discovery and implementation process.

Formally, for a software product *P*, $W(P)$ is the union of doable $WD(P)$ and completable $WC(P)$ Work Items but the intersection is the empty set:

$$\forall P, \exists W(P) \mid W(P) = WD(P) \cup WC(P) \text{ and } WD(P) \cap WC(P) = \emptyset \quad (5)$$

This categorization allows the proposed approach to shift from traditional binary tracking (e.g., “the story is 50% done”) to a precise value-realization metric. A Story is no longer viewed as a single task, with an arbitrary completion percentage, but as a container of value whose realization depends on the completion of its associated Scenarios. By separating atomic value units from aggregating structures, BLOW establishes a clear conceptual basis for value tracking that remains independent of the execution mechanisms and organizational coordination strategies.

4.2.3. Work-Centers: Domains of Accountability

A work-center is the operational environment where requirements are transformed into value. Adopted from Lean Manufacturing, a work-center in BLOW represents a specialized functional domain where specific activities transform inputs into outputs with incremental value. The primary entities flowing through these work-centers are the Work Items (*W*) categorized in Section 4.2.1.

This concept establishes the foundation for execution accountability within BLOW. Each work-center is responsible for applying a well-defined type of transformation to the Work Items it receives, ensuring that transformation outcomes can be explicitly associated with a specific domain of responsibility. As an illustrative analogy, in a manufacturing context such as a car factory, a work-center transforms individual components (inputs) into a functional subsystem or final product (output), whose value exceeds that of its isolated parts.

The transformation of a Work Item within these work-centers is viewed as a Lean-inspired process flow [52]. Work-centers in BLOW are not merely sequential. They may be organized hierarchically and decomposed into nested work-centers. A host work-center can delegate specialized aspects of its responsibilities to nested work-centers. This architectural choice necessitates different operational dynamics depending on the direction of the flow.

The process flow between sequential work-centers is managed by push–pull dynamics, following a supply chain logic. A preceding work-center “pushes” its output, which is then “pulled” as an input by the subsequent work-center. This ensures that work only moves forward when the next stage is ready to add value, minimizing “Work in Progress” (WIP) and waste. As illustrated in Figure 21, Work-center A pushes a Work Item into Work-center B, while Work-center B pulls that same item from Work-center A.

Conversely, the process flow within nested work-centers introduces the concept of specialization and value realization. Specialization refers to the specific transformation that a Work Item undergoes when it is processed by a dedicated nested work-center, enabling complex tasks to be broken down into highly focused domains of expertise. In contrast, value realization refers to the incremental value contributed by the outcome of the nested work-center back to its parent or host work-center once a specialized task is finished. For example, in Figure 21, work-center C.1 is decomposed into specialized work-centers C.1.A and C.1.B. These units operate independently within C.1, each focusing on a specific dimension of the process, and subsequently realize their value back to C.1 upon completion.

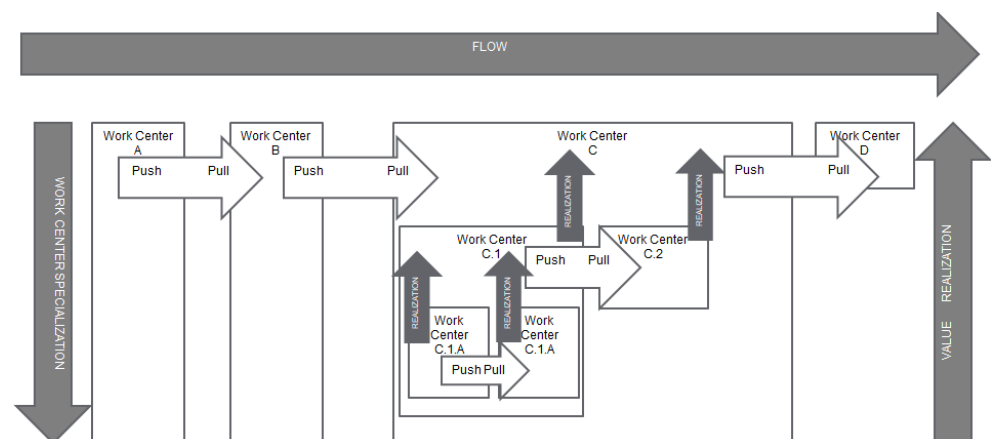


Figure 21. Work Items traverse through work-centers via Push–Pull mechanisms (horizontal flow) but achieve their value via specialized nested work-centers (vertical flow: specialization downward and realization upward).

4.2.4. Status of Work Items

A Work Item is transformed during the execution of prescribed activities within a work-center. In order to manage this transformation in a controlled and observable manner, BLOW distinguishes between the Stage in which a Work item resides and its current Status.

The Stage represents a macro-level phase of the transformation process and acts as a functional container that defines the general objective currently pursued for a Work Item

(e.g., preparation, execution, or completion). In contrast, the Status represents the granular, operational state of the Work Item at any given point within its current Stage.

This distinction is crucial because a Work Item may remain in the same Stage while its Status evolves internally to reflect operational progress.

Streamlining the transition between these Stages and managing internal Statuses is critical to minimizing downtime during production, as waiting is considered a significant source of waste in Lean production [53].

The admissible progression of Statuses depends on the category of the Work Item. In particular, BLOW distinguishes between doable Work Items (Units of Value) and completable Work Items (Incremental Value Items), each exhibiting different transformation patterns across Stages. A Unit of Value Item is actionable and yields value upon completion of a single transformation, whereas an Incremental Value Item accumulates value through the completion of its constituent Units of Value and can only be completed once all such units have reached their terminal states.

As established in Equation (5), $W(P) = W_d(P) \cup W_c(P)$ with $W_d(P) \cap W_c(P) = \emptyset$.

As illustrated in Figure 22, the BLOW operating model organizes the flow into three primary Stages, each hosting specific Statuses based on the item type:

1. Pending Stage: Identifies and prepares the item to be consumed.
 - To Do: The item is in the backlog but does not yet meet the Definition of Ready (DoR).
 - Ready: The item meets the DoR and is prepared for the Transforming Stage.
2. Transforming Stage: Where active technical or functional execution occurs.
 - In Progress: Assigned to Unit of Value Items (doable).
 - Active: Assigned to Incremental Value Items (completable), indicating the item is accumulating value from its constituent components.
3. Terminated Stage: Where the transformation process reaches its conclusion.
 - Done: Assigned to Unit of Value Items that match the Definition of Done (DoD). This is a static final state.
 - Completed: Assigned to Incremental Value Items. This status is dynamic; the item is considered completed only when all its constituent units of value are Done.

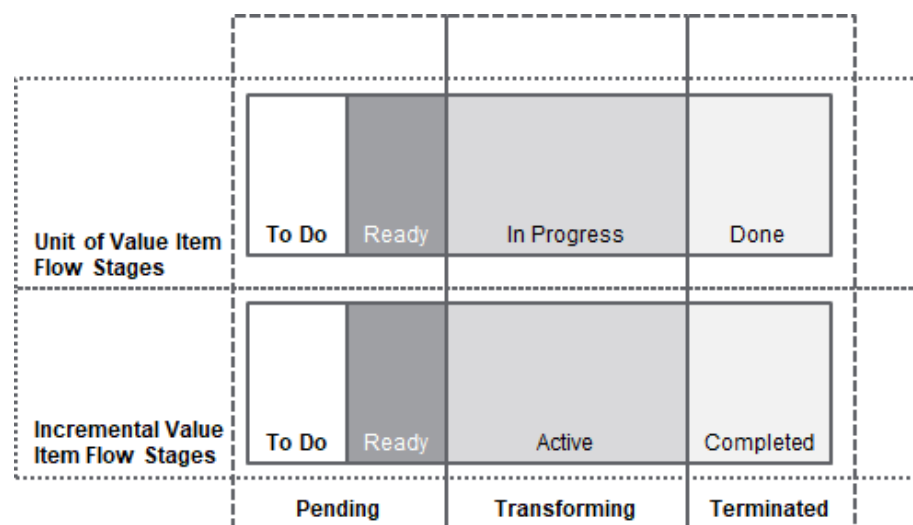


Figure 22. Graphical visualization of the hierarchy of Stages and Statuses. Each Stage functions as a container for specific statuses based on the Work Item type.

The distinction between Done and Completed is vital for managing scope uncertainty. In BLOW, a Story (an incremental value item) reaches the Completed status within the Terminated Stage only when all its associated Business Scenarios (units of value) are Done. If new Scenarios emerge as a result of stakeholder feedback or refinement, the Story transitions back to the Active Status within the Transforming Stage, while the newly introduced Scenario enters the In Progress Status. The Story can return to the Completed Status only once the added Unit of Value has reached Done.

4.2.5. Technical Domain Contexts

A Technical Domain Context (TDC) is a structural concept introduced in BLOW to organize the technological elements that contribute to the realization of a business solution. A TDC is defined as the cohesive set of technologies, skills, tools, and design patterns required to address a specific technical responsibility within the system, either partially or completely.

By defining TDCs, BLOW provides an explicit representation of the technical space in which solutions are constructed. This representation enables the decomposition of a software product according to technological responsibilities or stacks, without prescribing execution order or coordination mechanisms. For instance, consider a web application: its architectural design can be partitioned into distinct TDCs, such as a Front-end context (e.g., JavaScript/React) and a back-end context (e.g., Python/Django). Additional cross-cutting technologies, such as Docker containerization or Kubernetes orchestration, may be identified as separate or supporting TDCs depending on the required granularity of the model.

Formally, it can be expressed as follows:

$$\forall P, \exists TDC(P) \quad (6)$$

where P is the final software product and $TDC(P)$ is the set of Technical Domain Contexts composing such product or solution.

A fundamental rule in BLOW is the explicit mapping between Enablers (E) and Technical Domain Contexts. Enablers represent technical requirements or architectural capabilities necessary to support business functionality, and each Enabler must be associated with exactly one TDC. For example, in a decoupled system, an Enabler responsible for “Exposing a REST API” would be located in the back-end TDC, while an Enabler for “Consuming a REST API” would be situated in the Front-end TDC.

Formally, for every Enabler (E) in the set of Enablers ($E(P)$), there exists exactly one Technical Domain Context (TDC) in the set of Technical Domain Contexts ($TDC(P)$) such that (E) belongs to (TDC).

$$\forall E \in E(P), \exists!, TDC \in TDC(P) | E \in TDC \quad (7)$$

This strict one-to-one mapping ensures that technical infrastructure and architectural evolution are governed with the same level of accountability as business Stories. By situating Enablers within specific TDCs, BLOW enables the precise identification of the skills, tools, and technologies involved in their realization, while deferring decisions regarding execution, coordination, and flow to subsequent operational layers of the operating model.

4.3. Roles and Collaboration Practices

An instrumental part of BLOW is defining the relevant actors and orchestrating their interactions to successfully bridge the gap between business needs and technical implementation. In complex software development environments, merely assigning tasks is

insufficient to ensure alignment between stakeholder needs and executable solutions. Effective coordination in such environments requires clearly defined domains of accountability and disciplined interaction patterns centered on shared artifacts rather than on task assignment alone.

BLOW therefore establishes a clear role model, combined with structured collaboration practices, to systematically guide the discovery, formulation, and execution of scenarios.

This section is organized as follows: First, we introduce the contributing roles and their domains of accountability. Next, we describe the Three Amigos practice as the primary mechanism for discovering and formulating business scenarios. Finally, we introduce the Technical Three Amigos, a novel extension of BLOW that governs the translation of validated business scenarios into Enablers and their Technical Domain Contexts.

4.3.1. Contributing Roles

Figure 23 presents an overview of the contributing roles defined within the BLOW approach. Roles are categorized according to their primary domain of expertise—business or technical—and are associated with specific responsibilities during collaboration practices.

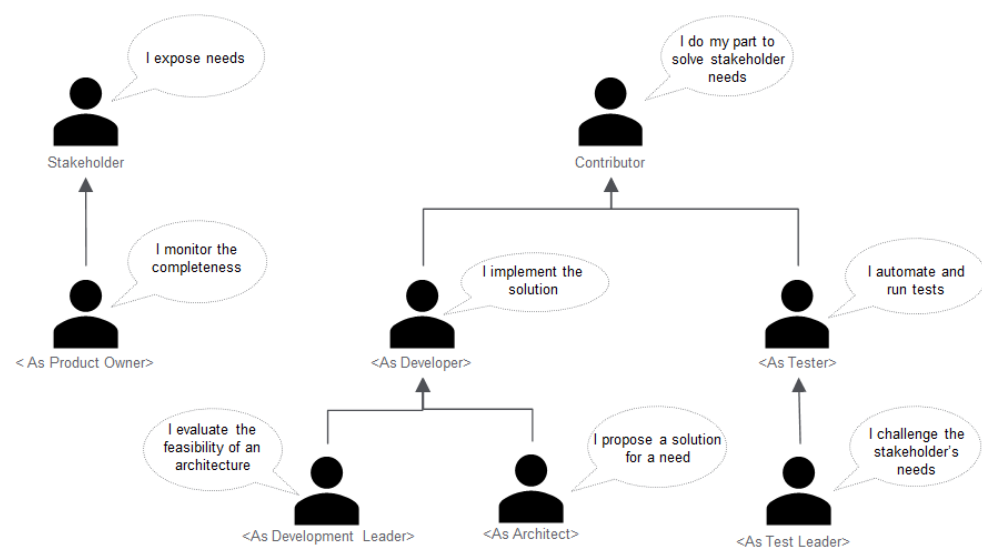


Figure 23. Overview of BLOW contributing roles.

As depicted in the Figure 23, roles and persons are not equivalent. A single person can assume multiple roles depending on the activity being performed. BLOW is therefore role-driven rather than person-driven, allowing flexibility in role assignment while preserving accountability.

For example, a contributor may alternate between Developer and Tester responsibilities within a TDD cycle, or between Developer and Architect responsibilities when addressing architectural concerns.

For example, a person can act as both a Tester and a Developer, or as a Developer and an Architect. BLOW is applied on the basis of two main types of actors: Stakeholders, who specialize in the business domain, and Contributors, who specialize in the technical domain.

The Stakeholder represents the voice of the business domain and is responsible for exposing a need. This need can be a new feature, an improvement on an existing one, or a technological Enabler. These needs are expressed in terms of desired behavior, which later becomes formalized as Business Scenarios. Examples of Stakeholders include marketing specialists, customers, or product specialists. The Product Owner is a specialized Stake-

holder who monitors the completeness of the need. A Product Owner can act on behalf of a Stakeholder, such as a product specialist.

A Contributor is an actor who contributes to the technical solution. Members of an agile scrum team are considered Contributors. Contributors are responsible for realizing Scenarios and for advancing Stories and Enablers toward completion through disciplined execution practices.

Depending on the activities being performed, the Contributor may adopt different specialization roles. A Developer, as a Contributor, is responsible for implementing Unit of Value Items in Pending Stage to Complete Incremental Value Items in Terminated Stage. For example, they will receive a set of Scenarios assigned to a Story in Ready State and release the Story in Completed State.

- Architects are Developers who have the big picture of a complex solution and the skills to break down challenging functionalities into manageable parts using existing architectural designs such as architecture patterns. Architects define Enablers as technical capabilities required to support current and anticipated Business Scenarios.
- Development Leaders challenge and validate the proposals of the Architects. This role explicitly promotes the identification of missing behaviors and failure conditions that may require additional Enablers or Scenario refinement. For example, if the software architecture involves gathering information from a database, the Development Leader could challenge the existing solution, enabling the Architect to define a behavior when the database is down.

Testers contribute to the solution by designing and executing tests. They promote the automation of test execution and must have strong skills to support test automation design. Testers play a central role in maintaining the precision and verifiability of Scenario definitions. Depending on the activity, a Tester and a Developer often have the flexibility to swap roles. For example, when TDD is applied, the Developer role switches to the Tester role when a new test is defined.

- Test Leaders challenge the scenarios during the Three Amigos practice as defined in ATDD. They are responsible for ensuring that Scenarios are formulated in a manner that supports automation, reuse, and traceability to executable tests. For example, Test Leaders should have enough experience writing Gherkin scenarios to make them easy to reuse and link to code.

This role model ensures that accountability is preserved while maintaining flexibility, enabling BLOW to adapt to diverse organizational contexts. By associating responsibilities with Scenarios rather than tasks, BLOW provides a disciplined yet adaptable mechanism for coordinating work across business and technical domains.

4.3.2. The Three Amigos Practice

Heng et al. [54] thoroughly investigated the application of Stories and behavior-driven development (BDD) scenarios for the design and development of software architecture. Their work explores how user stories (US) and BDD constructs can be systematically associated with Agent-Oriented (AO) and Object-Oriented (OO) architectural paradigms, emphasizing the advantages of high-level programming languages. Furthermore, they highlighted the potential of Natural Language Processing (NLP) techniques to transform US and BDD scenarios into a unified and coherent software architecture. However, as described by Chemnitz [55], a notable hurdle persists in the automated generation of operational code derived from these functional scenarios. Chemnitz proposed generative artificial intelligence (AI) as a potential solution. Nevertheless, Chemnitz's proposal lacks

a systematic application within a clearly defined execution structure, necessitating the development of a proposal for implementing Business Scenarios.

The Three Amigos is a practice which was introduced as part of ATDD (Section 3.3) intended to generate Business Scenarios. Subsequently, it was refined to be extended in BDD (Section 3.4). In accordance with the proposal by Seb Rose and Gáspár Nagy [41], a comprehensive recommendation for implementing the Three Amigos practice is depicted in Figure 24.

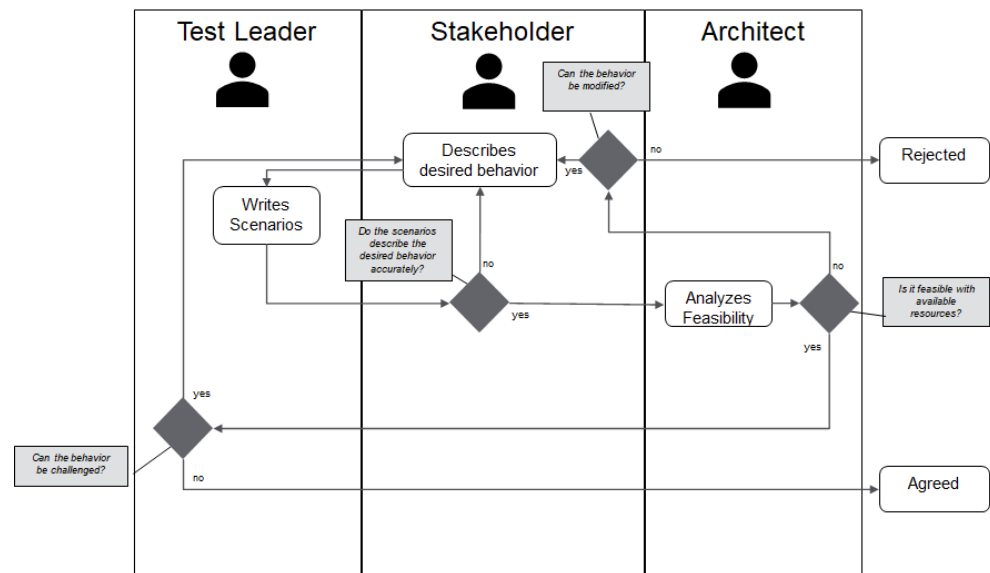


Figure 24. Overview of a flow representing the Three Amigos practice.

In BLOW, the Three Amigos is explicitly defined as a scenario-driven collaboration practice whose primary objective is to produce well-formed, agreed-upon Business Scenarios that serve as Units of Value for subsequent development activities. The practice begins with BDD Discovery activity (Section Discovery). During Discovery, the Stakeholder requests a solution to a need by exposing the expected behavior of a Story and the Business Rules. The Test Leader then writes the scenarios after the need, the behavior and the business rules are exposed. If the Stakeholder agrees that the Scenarios are describing how the software product will solve the need, then they are presented to the Architect to analyze their feasibility.

If these Scenarios cannot be implemented with existing resources, the Stakeholder is required to consider if the behavior can be modified. This may be achieved by simplifying or reducing the scope of the Story. Conversely, the request will be rejected if the behavior cannot be modified.

Once the Scenarios describe the desired behavior and their implementation is feasible, the Test Leader starts challenging the existing behavior of the Story. This process may result in the introduction of new behaviors or the refinements of existing ones where one or more challenges are identified. The Architect is responsible to evaluate the feasibility every time the Scenarios of each Story are modified. This iterative cycle continues until the participating roles reach a shared understanding and consensus on the behavior defined by the Scenarios.

It is only when Scenarios have been well formulated, declared to be feasible and immune to challenge, a consensus has been reached on the Story and it is ready to be presented to Contributors.

In a recent study, Mughai [56] advocates for the employment of checklists as a means to enhance the clarity and uniformity of behavior-driven development (BDD) scenarios, with

specific emphasis on their benefits for novice practitioners. By integrating this checklist-driven process, not only were ambiguities significantly diminished, but the overall quality of the development process experienced a marked improvement. This was mainly attributed to the promotion of a deeper comprehension and more effective evaluation of BDD scenarios, leading to a more efficient and reliable development workflow.

With all these considerations, the following responsibility matrix can be defined identifying the participation of the different roles on the practice as shown in Table 9:

Table 9. Responsibility assignment matrix. Test Leaders take an active set of responsibilities in the Three Amigos practice.

Activity/Role	Test Leader	Stakeholder	Development Leader
Need Exposition and behavior description	Informed	Responsible	Accountable
Feasibility Analysis	Informed	Accountable	Responsible
Scenario writing	Responsible	Accountable	Accountable
Behavior challenging	Responsible	Accountable	Accountable

4.3.3. The Technical Three Amigos Practice

BLOW introduces the Technical Three Amigos as a novel practice to enable the business implementation of Stories and their assigned Business Scenarios in Technical Domain Contexts. This practice arises from the need to create the necessary technological infrastructure to support the implementation of Stories. This practice is inspired by Santos et al. [57] work around the possibility of using BDD to implement non-functional requirements. In this work, Santos demonstrates the possibility of applying BDD to implement Enablers. The Technical Three Amigos complement this work by offering a systematic way to define Technical Scenarios.

The Technical Three Amigos practice follows the Three Amigos practice with an Architect acting as a unique Stakeholder that will propose a technical approach of the software product to be developed considering the overall architecture, a Developer that will analyze the feasibility of a specific approach to be delivered, and finally, the Tester that will challenge the approach ensuring that it is testable.

The Technical Three Amigos practice comprises a set of activities that address the technical workload of a software product development iteratively through each formulated Business Scenario. These activities are illustrated in Figure 25 and are as follows:

- **Context Analysis:** In this activity, the Technical Domain Contexts that are impacted by each individual Business Scenario are defined.
- **Enablement Analysis:** The aim of this activity is to identify the involved Enablers that are required to support the Business Scenario in impacted Technical Domain Contexts. If existing Enablers facilitate the implementation of the Business Scenario, then the Business Scenario can be implemented and the practice ends here for this specific Business Scenario.
- **Technical Scenario Discovery:** if the Enabler is not able to fit the Business Scenario, this activity examines the necessity of incorporating Technical Scenarios for each affected Enabler.
- **Technical Scenario Formulation:** In this activity the new Technical Scenarios are written for affected Enablers.

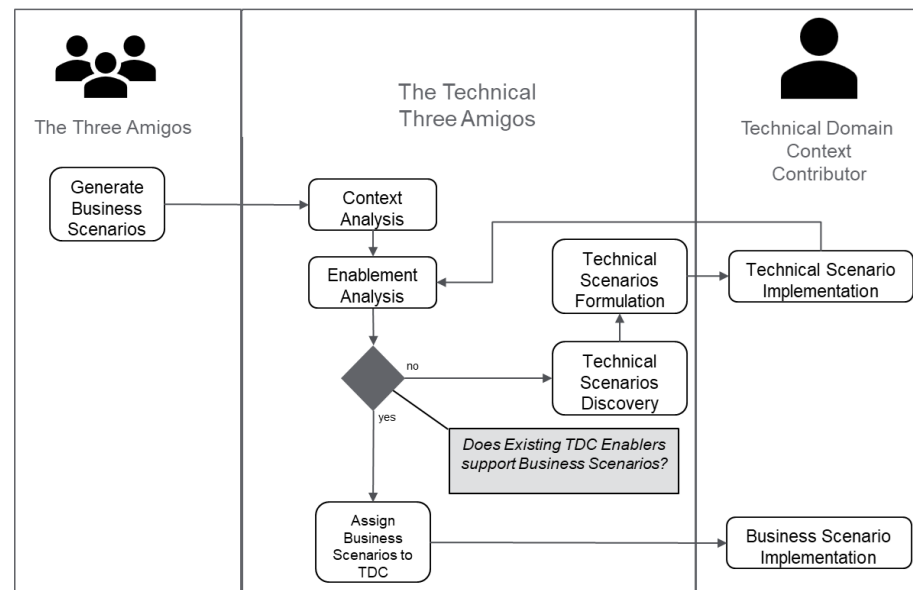


Figure 25. The Technical Three Amigos, technical scenario definition flow.

Activity: Context Analysis

Once Business Scenarios are formulated from a Story, the Technical Three Amigos can identify the involved Technical Domain Contexts that are required for development. The Context Analysis is performed on each Business Scenario to assign it to specific Technical Domain Contexts when they are defined and available, or on new ones when the need to create new Technical Domain Contexts is discovered (Figure 26). Although Architect takes the lead, Developer and Tester participate by validating the proposal.

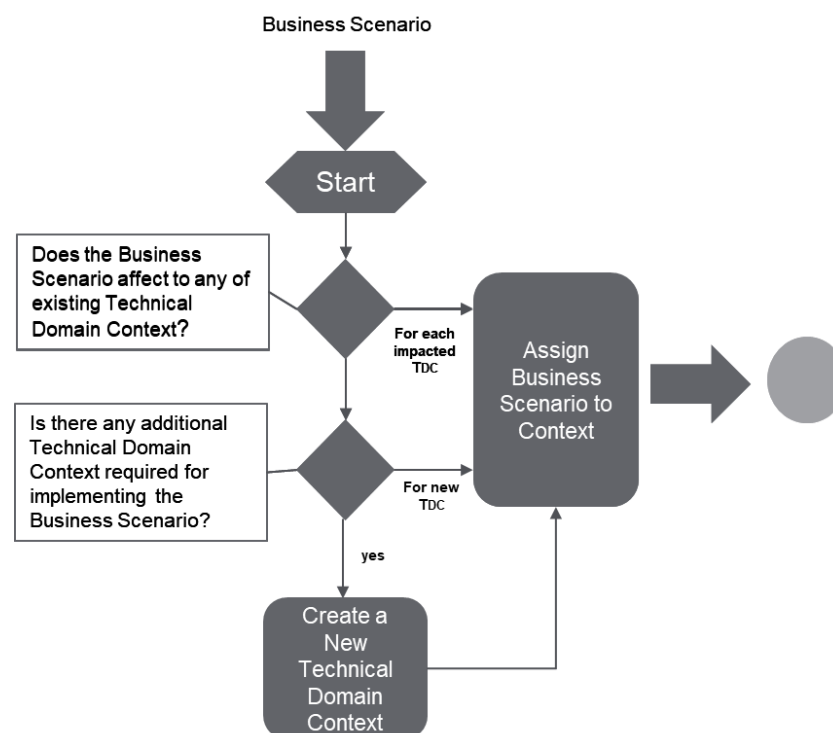


Figure 26. Workflow for context analysis executed by the Technical Three Amigos.

For example, consider a solution involving three different Technical Domain Contexts, front-end, back-end, and back-office. The front-end executes calls to back-end, which then orchestrate different calls to back-office databases to present the result to the user.

Under these circumstances, let there be a Business Scenario that requests an alarm to be triggered when specific conditions are met with data coming from back-office databases. The Architect can propose that this alarm be evaluated in the back-end and displayed in the front-end, without the need to do any change for analyzing the alarm in the back-office databases. The Developer could claim that this alarm implies modifying a REST API consumed by the front-end, and it would be easier to implement this condition on the front-end. However, the Tester could argue that it is easier to test only in the front-end. Once the Architect has determined that this approach is beneficial from an architectural standpoint, The Technical Three Amigos can conclude that this Business Scenario is applicable solely to the front-end. The outcome of a generic Context Analysis is illustrated on Figure 27. Once Business Scenarios are formulated from a Story, specific tests for the affected Technical Domain Contexts can be created for each Business Scenario.

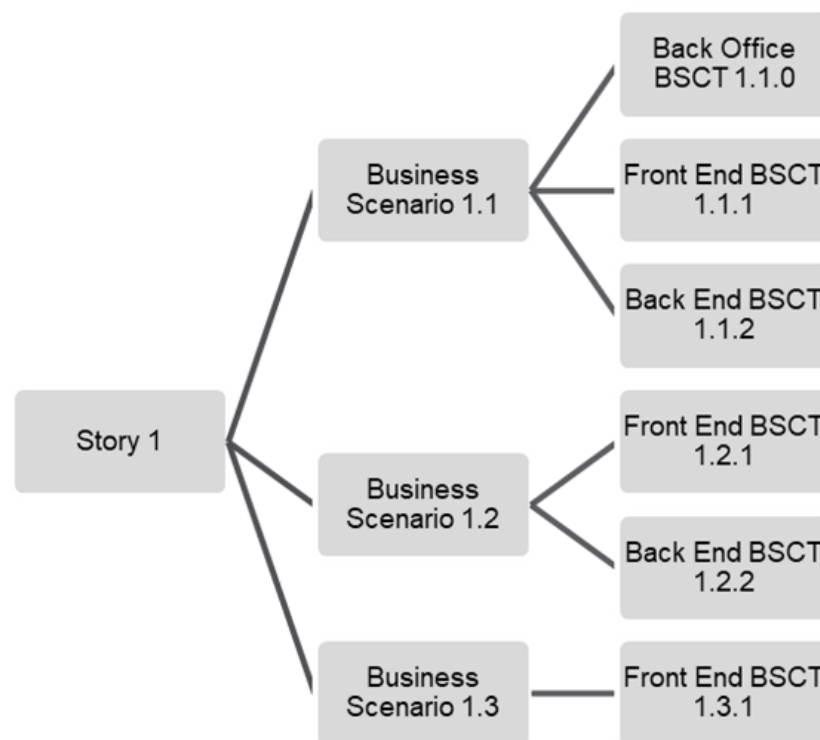


Figure 27. The tree graph elucidated the process of creating specific tests for each business scenario, exclusively on the affected technical domain contexts, as defined by The Three Technical Amigos.

Activity: Enablement Analysis

Once the Business Scenarios have been assigned to Technical Domain Context, the next activity consists on analyzing if such Technical Domain Context is able to support the Business Scenario using existing Enablers. The workflow is summarized in Figure 28. If existing Enablers facilitate the implementation of the Business Scenario, then the Business Scenario is ready to be implemented. Otherwise, two options are available:

- The proposal of a new Enabler to facilitate the implementation of the Business Scenario.
- Reactivating or modifying an existing Enabler. As with the Context Analysis, the Architect is responsible for leading this activity, while the Tester and Developer will validate the proposal.

For example, given a monitoring system with two Technical Domain Contexts, a front-end and a back-end. The front-end has an Enabler representing a widget in React which consumes a REST API from the back-end. This REST API provides information

about a system that is in an alarm state because its value is out of a defined range. The existing front-end can represent the alarm state.

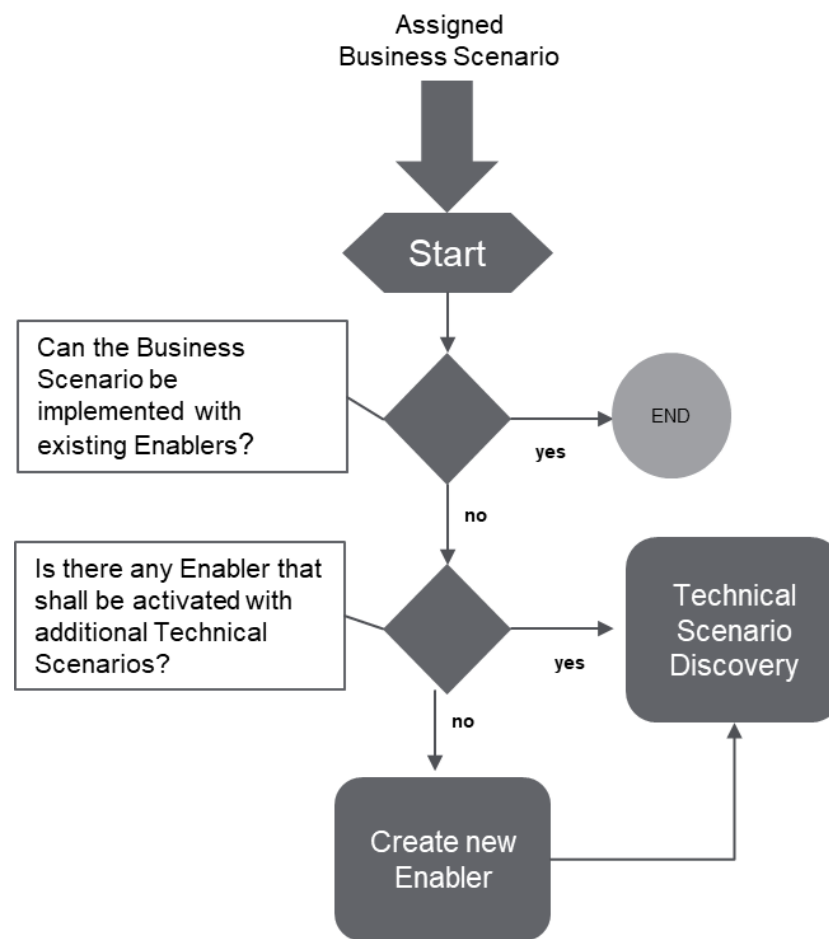


Figure 28. Workflow for enablement analysis.

Activity: Technical Scenario Discovery

This activity is required when a Technical Domain Context is not able to fit a specific Business Scenario or when a new Technical Domain Context is required. Following the Three Amigos practice, a set of Technical Rules should be established, but in this case, they will be defined by the Architect. Some examples of Technical Rules can be:

- The technical domain exposes data using gRPC.
- The database operations follow a CQRS architecture pattern.

The proposed Technical Scenarios are subject to follow these rules. Here, the Developer must approve, and the Tester can challenge these Scenarios, as in the Three Amigos practice. In this case, the Developer also has the capacity to challenge the architecture, because ultimately, it will be the developer's responsibility to implement them. The output of this activity is a set of To-Do Technical Scenarios.

Activity: Technical Scenario Formulation

To progress these Technical Scenarios to the Ready status, it is necessary to formulate them. This formulation is conducted by the Tester and the formulated Technical Scenarios are then consumed by the Developer for implementation. Consequently, once the Technical Scenarios have been formulated, they are ready to be implemented using TDD.

Figure 29 shows a real example of Technical Scenarios that has been modeled using Gherkin, in accordance with BLOW.

```

Feature: SwitchDictionary
In order to enable functional programming,
I want to provide Dictionaries with similar functionality,
So I can handle through collections

@SwitchDictionary
Scenario: Dictionary with a Lambda function uses switch to return the proper lambda
  Given a Dictionary wich maps numbers to Lambda Functions
  And A lambda function which returns string 'first lambda function' assigned to Operation '1'
  And A lambda function which returns string 'second lambda function' assigned to Operation '2'
  When I execute the extended switchOperation '1' and default returns an empty string;
  Then the result of switch should be 'first lambda function'

@SwitchDictionary
Scenario: Dictionary with a Lambda function uses switch to return default
  Given a Dictionary wich maps numbers to Lambda Functions
  And A lambda function which returns string 'first lambda function' assigned to Operation '1'
  And A lambda function which returns string 'second lambda function' assigned to Operation '2'
  When I execute the extended switchOperation '3' and default returns an empty string;
  Then the result of switch should be empty string

```

Figure 29. An illustration of a technical scenario in Gherkin from a real case.

4.4. The BLOW Layered Operating Model

Once the taxonomic components, work-centers, and contributing roles and practices have been established in the previous section, this section details the BLOW Layered Operating Model as a dynamic orchestration system. The model is realized through four nested functional layers, where each level acts as a refinement filter that progressively transforms high-level business intent into rigorous technical verification. This hierarchical organization ensures that the workflow is not merely a linear sequence of tasks, but a systematic process of specialization and value realization. In this operating model, each layer maintains governance over the acceptance contract before delegating execution to the next level of abstraction, thereby bridging the gap between abstract requirements and executable code.

The operationalization of BLOW is grounded in the systematic transformation of Stories (*U*) into functional, verified code. Similarly to a manufacturing plant, BLOW orchestrates the software development process through specialized work-centers responsible for transforming Work Items and controlling their flow across the delivery lifecycle. Within each work-center, Work items transition from the Pending stage to the Terminated stage. Value realization is achieved only upon a Work Item reaches the Terminated Stage.

BLOW is organized into a set of nested layers of work-centers that iteratively refine Work Items (see Figure 30). The primary entry point is the Stories Work-Center, which pulls user stories from the backlog and pushes the resulting validated behaviors toward receiving work-centers.



Figure 30. The BDD cycle within BLOW: User stories are pulled into the system to push deliverable code into a receiver work-center.

To facilitate this transformation, the system requires the execution of specialized work-centers in four nested layers, each contributing to the realization of value for its hosting layer (see Figure 31):

- **Layer 1: Stories work-center.** This work-center is focused on the processing of Stories until they reach the Completed status. To do that, the Three Amigos practice is applied in order to first generate the Business Scenarios and then an implementation should be

done on each Business Scenario to complete all the Stories. The ATDD is the dominant framework in this layer.

- Layer 2: Business Implementation work-center. This work-center manages the implementation of Business Scenarios until they reach the Done status. In this case, the application of The Technical Three Amigos practice determines the need to define or not to define Enablers to facilitate the implementation of Business Scenarios. The TDD is the dominant framework in this layer.
- Layer 3: Enabler work-center. In this specialized work-center the Enablers are identified and extended. The Technical Three Amigos practice is again employed to discover and formulate the Technical Scenarios (S_T) necessary to build the architectural infrastructure that supports the business requirements.
- Layer 4: Technical Implementation work-center. This work-center is specialized in the implementation of Technical Scenarios identified by The Technical Three Amigos practice. TDD is the dominant framework in this work-center.

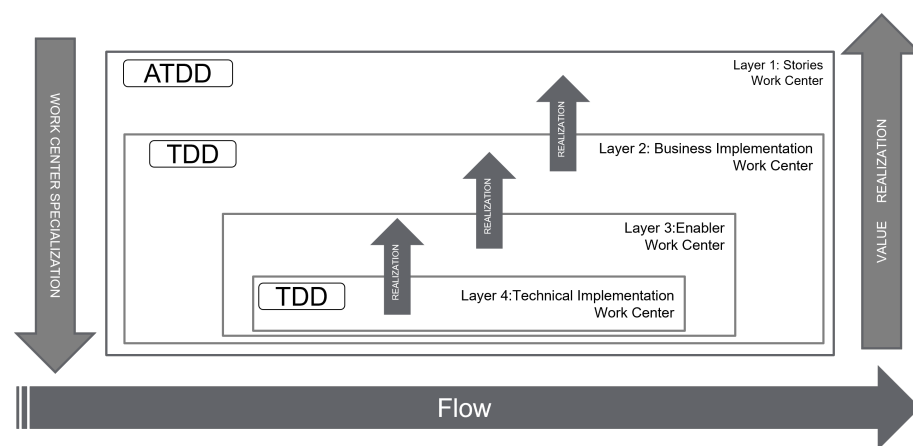


Figure 31. BLOW represented as a composition of nested work-centers and their dominant frameworks (ATDD and TDD).

The following sections provide a comprehensive exposition of how each layer operates. The process is illustrated through the implementation of a single Story. However, in a production environment, this model is applied iteratively to all the items in the backlog, encompassing every modification, refactoring, or new requirement as a new cycle of transformation through these work-centers until the final working code is obtained.

4.4.1. Layer 1: Stories Work-Center

In this layer, the Stories work-center is responsible for processing the Stories as Work Items from a To-Do status in Pending Stage until they reach the Completed status in Terminated Stage. During the processing of each Story, the associated Business Scenarios must be generated, formulated, and implemented with the support of specialized work-centers in nested layers.

The Stories work-center sets the initial To-Do status for each Story that can be processed by this work-center. This can happen in two cases:

- When a new Story is pulled from the backlog by Stories work-center.
- When the Story requires the addition of new functionality. Then, the Story is returned to the To-Do status for further development.

The processing of each Story involves a sequence of collaborative activities (Figure 32 to be carried out as follows):

- Scenario Discovery: By the application of The Three Amigos practice, the Business Scenarios are identified and associated with the specific Story. This implies that

the status of the Story changes to Ready, and the identified Business Scenarios to To-Do State.

- **Scenario Formulation:** Once the identified Business Scenarios describe the desired behavior and their implementation is feasible by the Three Amigos practice, the Business Scenarios are written in a specific language such as Gherkin to ensure they are both human-readable and machine-executable. Once formulated, the Business Scenario transitions to the Ready state.
- **Context Analysis:** This activity is performed using the Technical Three Amigos practice to determine the affected Technical Domain Contexts with each Business Scenario, which implies moving the Business Scenarios into the In Progress state.
- **Implementation of Business Scenario:** The formulated Business Scenarios are then processed to Business Implementation work-center in a nested layer (Section 4.4.2) for implementation. This work-center realizes value by pushing the working code of the Business Scenario onto specific Technical Domain Contexts.

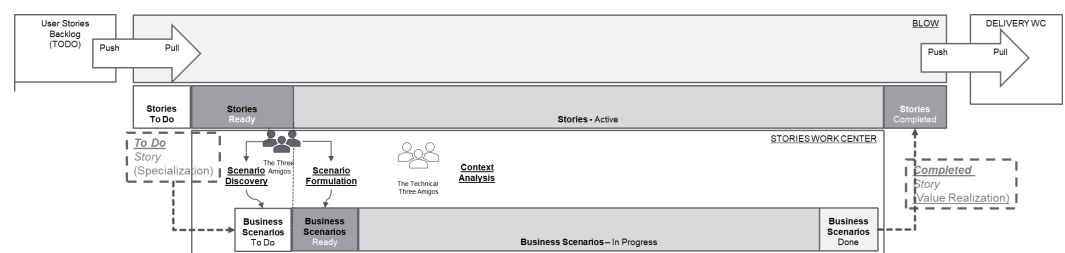


Figure 32. Story work-center activities.

Thus, in this work-center, we can consider that a Product P will have a dependent set of Stories $U(P)$ as Incremental Value Items:

$$\forall P, \exists U(P) \quad (8)$$

For each Story $U \in U(P)$, there exists a set of Business Scenarios $S_B(U)$ that describe its expected behavior and act as Unit of Value:

$$\forall U \in U(P), \exists S_B(U) \quad (9)$$

The Business Value V_B of a Story U is defined as the cardinality of its associated set of business scenarios $S_B(U)$:

$$\forall U \in U(P), \exists V_B | V_B = |S_B(U)| \quad (10)$$

A result of the Context Analysis can be an increase in the number of Technical Domain Contexts. This can happen when, as part of the technical solution, a Business Scenario requires the creation of a new Technical Domain Context. For example, a product composed of a simple front-end with a form to insert data into a database through a back-end could have a new Business Scenario in which the technical solution required the creation of a worker to export data from the database part of the solution to a third party database. That could imply new Technical Domain Context to fit this worker to appear. In other words, the outcome of the Context Analysis can be an increment of Technical Domain Contexts to realize their value.

Thus, considering that a set of Technical Domain Contexts of a product $TDC(P)$, and a subset of Contexts $T'DC(P)$,

$$T'DC(P) \subset TDC(P) \quad (11)$$

And defining the Contextual Impact of a Business Scenario as the cardinality on the number of $T'DC(P)$ affected such specific Business Scenario:

$$IC = |T'DC(P)| \quad (12)$$

In cases where the Contextual Impact associated with a particular business scenario exceeds zero, there arises the potential for an expansion within the established set of the Technical Domain Context. This expansion results from executing the Context Analysis activity. Consequently, the change (denoted as delta) will exhibit a value that is either equal to or greater than zero.

$$IC > 0 \Rightarrow \Delta|TDC(P)| \geq 0 \quad (13)$$

This situation can be visualized in a new product development context. When The Technical Three Amigos analyze the first Business Scenario, they may need to define the initial Technical Domain Contexts required for its implementation, such as a front-end and a back-end. In this example, the contextual impact is $IC = 2$.

4.4.2. Layer 2: Business Implementation Work-Center

The Business Implementation Work-Center is the layer responsible for the technical realization of the business behaviors defined in Layer 1. While the previous layer focuses on “what” the system should do, this layer focuses on “how” it is implemented within specific technical boundaries.

In this layer, the formulated Business Scenarios are implemented as work items by executing Business Implementation work-center until a valid code to support them is achieved. Depending on the Contextual Impact (IC) identified during the Context Analysis in Layer 1, this layer may activate multiple instances of the work-center, one for each affected Technical Domain Context (T_{DC}).

Initially, Business Scenarios that impact a specific Technical Domain Context are pulled into the corresponding Business Implementation work-center in a To-Do state. These Business Scenarios are named Contextual Business Scenarios to distinguish when they are applied to a specific Technical Domain Context. Then, the implementation of Contextual Business Scenarios implies the creation of a specific test in accordance to TDD named Business Scenario Contextualized Test (BSCT) and a valid code into a specific programming language.

Figures 33 and 34 illustrate how the Business Implementation work-center are activated for a single Technical Domain Context and Three Technical Domain Contexts.

During the processing of each Business Scenario into the Business Implementation work-center of each Technical Domain Context, there is a sequence of activities to be carried out as follows:

- **Enablement Analysis.** In this Business Scenario, the Enablement Analysis is performed using the Technical Three Amigos practice to determine if the Business Scenario is Ready to be implemented in this Technical Domain Context or requires further specialization using existing Enablers. In the latter case the execution of the Enabler work-center (Section 4.4.3) in a nested layer is required.
- **Test Implementation.** In this activity, the Business Scenario Contextualized Tests (BSCTs) are written using the specific testing language. Then, the Contextual Business Scenarios transition to the In Progress state.
- **Code Implementation.** In this activity, the code implementation should be completed for each Contextual Business Scenario. Then, the BSCT is passed to check the validity of the code. During this activity, the appropriate refactoring is performed to find the

minimum code implementation that satisfies the BSCT. Once all the BSCTs have been passed, the Contextual Business Scenarios are moved to Done Status.

- Commit Business Scenarios. In this activity, the Contextual Business Scenario is saved and pushed to Stories work-center to realize value.

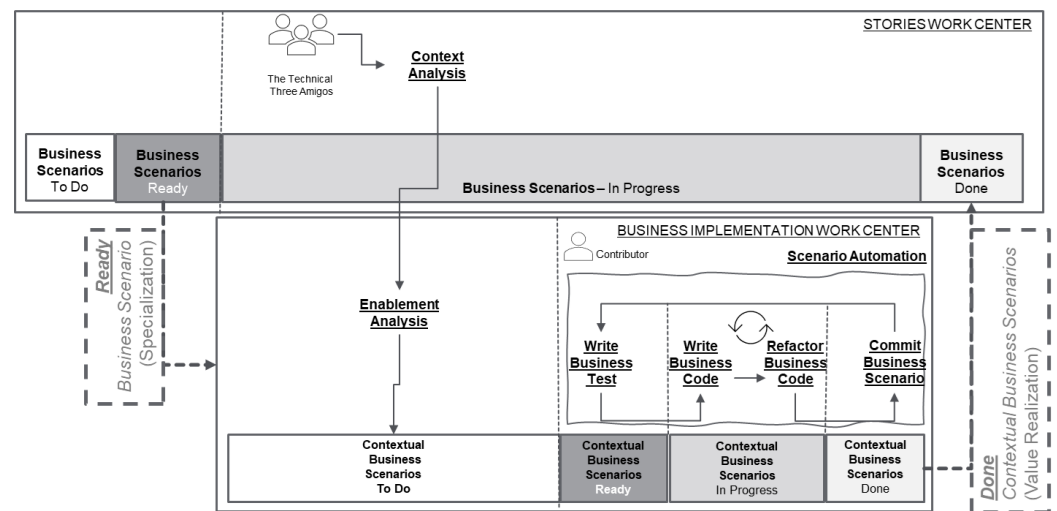


Figure 33. Business Implementation work-center. This figure represents a solution with a single domain context to visualize all activities.

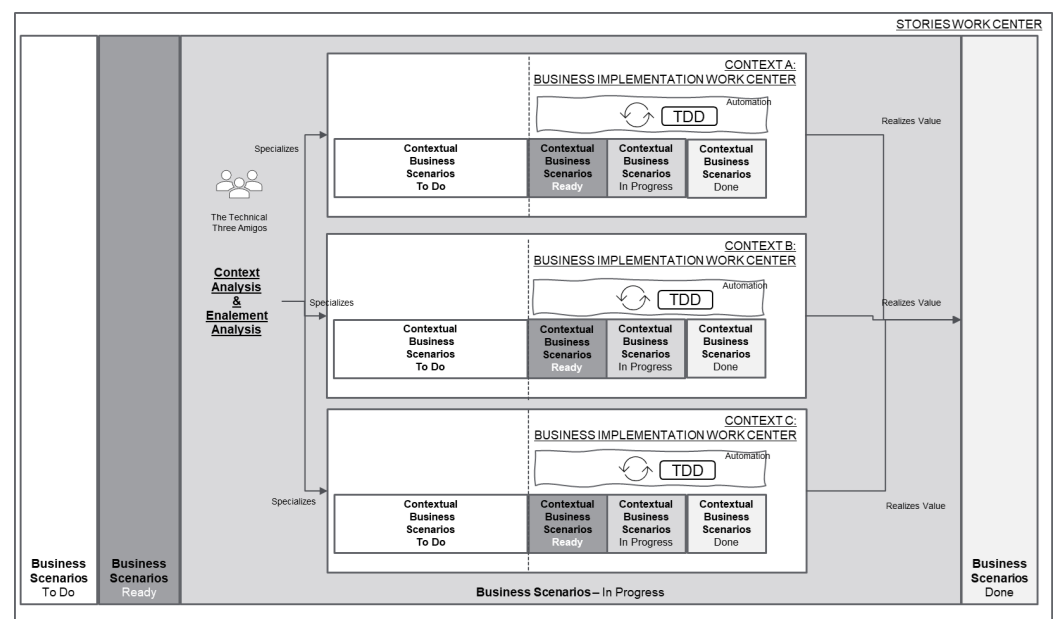


Figure 34. This figure complements Figure 33 displaying several contextualized business scenarios work-centers.

The relationship between business scenarios and their technical verification is formalized through the set of contextualized tests. Let $X_B(T'_{DC}(S_B))$ be the non-empty set of BSCTs required to verify a specific business scenario within its affected contexts:

$$\forall S_B, \exists X_B(T'_{DC}(S_B)) \neq \emptyset \quad (14)$$

A business scenario is considered Done only when all Contextual Business Scenarios in all Technical Domain Contexts are Done. This ensures that business value is not only defined but also verified through the underlying technical implementation.

For example, consider a project in which a software product is divided into three Technical Domain Contexts: the front-end, the back-end, and the back-office contexts. Additionally, assume that this product framework incorporates a function specifically designed for logging user sign-ins, which is executed on the back-end once a user has successfully logged in. Consequently, a newly introduced Business Scenario could permit users to review their login history.

Upon conducting a Context Analysis, it was determined that this Business Scenario affects both the front-end and the back-end, given that the Back-office already maintains a table to log such data. Furthermore, the back-end provides a RESTful API that reports the user's most recent login time. An Enablement Analysis determined that this REST API can be augmented to also incorporate login history data. This finding indicates that the existing RESTful API can adequately address the requirements of the Business Scenario without the need for additional Enablers. In contrast, the front-end, which currently retrieves the user's latest login time through the REST API, must introduce a new option within the profile menu. This modification will enable the display of a comprehensive registry of all the login instances. Consequently, a new Enabler is required to effectively implement the Business Scenario.

4.4.3. Layer 3: Enabler Work-Center

In this layer, the Enabler work-center is the specialization of the readiness preparation for Contextual Business Scenarios in a Technical Domain Context when the latter cannot support the implementation logic to cover a specific behavior defined by a Context Business Scenario. Its main purpose is furnishing the missing Technical Scenarios required to support the extension of the product to provide the required Context Business Scenario behavior.

These Technical Scenarios are generated by the processing of Enablers as work items within the Enabler work-center. In the case that Enablers are identified during the Enablement Analysis in the Business Implementation work-center of a specific Technical Domain Context, they are pulled to Enabler work-center in To-Do state. During the processing of each Enabler, the Technical Scenarios should be defined and formulated to support the Enabler.

The following sequence of activities must be carried out during the processing of each Enabler (Figure 35):

- **Technical Scenario Discovery.** In this activity, the application of Technical Three Amigos practice allows us to identify the Technical Scenarios of each Enabler. This results in the transition of all Technical Scenario into To-Do status. While business scenarios focus on user behavior, technical scenarios focus on system properties such as persistence, security, or cross-service communication.
- **Technical Scenario Formulation.** In this activity, the Technical Scenario are written in a specific language such as Gerkin. The formulated Technical Scenario will then transition into the Ready state as well as Enabler moving to the Active status.
- **Implementation of Technical Scenario.** The formulated Technical Scenarios are then processed by the Technical Implementation work-center in a nested layer (Section 4.4.4) for implementation taking into account the corresponding Technical Domain Context where they are coded. As result, the Technical Implementation work-center will realize the value of Enabler completing it when all Technical Scenarios are Done.

Previously, $T'DC$ was defined as the set of impacted Technical Domain Contexts by a Business Scenario. For all the impacted Contexts $T'DC$, exists a set of Technical Scenarios ST so the cardinality of the incremented set will be bigger than 0:

$$\forall T'DC, \exists ST | \Delta | ST | > 0 \quad (15)$$

Technical Cost (CT) of a Business Scenario is defined as the minimal necessary increment of value in terms of Technical Scenarios to support the implementation of such Business Scenario and represents a way to measure the necessary effort required to implement a Business Scenario.

$$CT = \sum \Delta |ST| \quad (16)$$

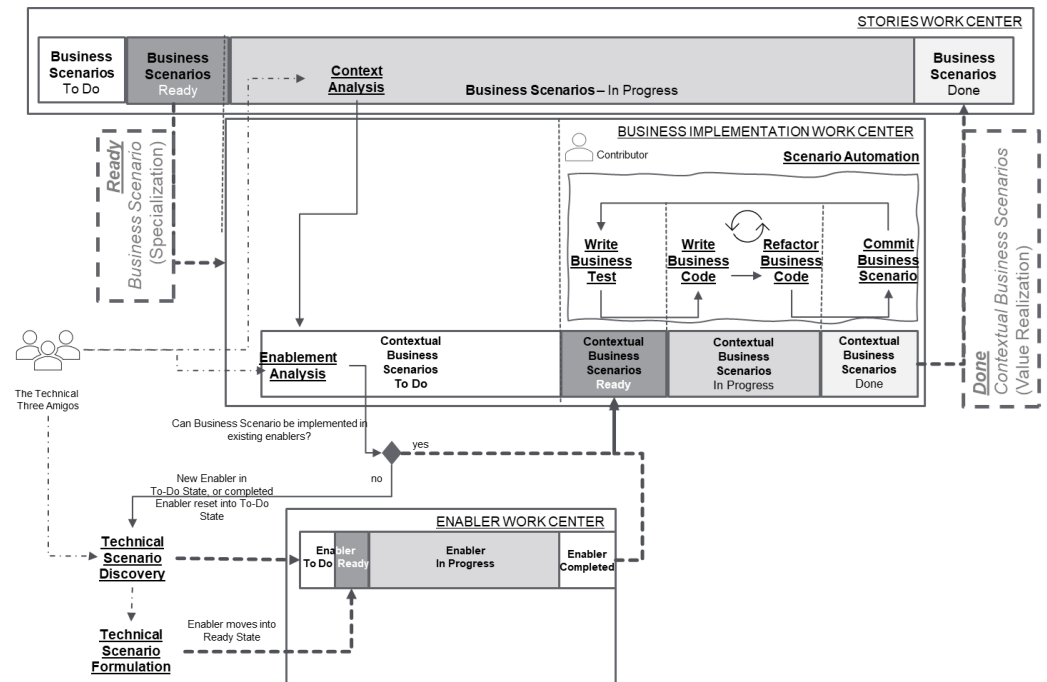


Figure 35. Enabler work-center. Because the analysis starts from a business scenario in a specific Context, there will be a work-center per Technical Domain Context.

4.4.4. Layer 4: Technical Implementation Work-Center

The Technical Implementation Work-Center is the most granular level of the BLOW layered operating model. Its mission is the rigorous execution and verification of the Technical Scenarios (S_T) defined in the previous layer. While Layer 2 deals with business logic, Layer 4 focuses on the infrastructure of the system—such as database schemas, API endpoints, or security protocols—that provide the necessary support for those business behaviors.

Specifically, this work-center processes the Technical Scenarios as work items from each Enabler for a specific Technical Domain Context. In this case, both terms of Technical Scenario and Contextual Technical Scenario are interchangeable since they are defined directly by a specific Technical Domain Context. Then, we can only build a Technical Test for a Technical Scenario with a relationship 1:1.

Initially, Technical Scenarios are pulled into the corresponding Technical Implementation work-center in a To-Do state, while the Enabler in the hosted Enabler work-center changes this state to Active status.

The operational flow in this work-center follows a strict Test-driven Development (TDD) discipline for each Contextual Technical Scenario, ensuring that no code is written without a preceding automated verification (see Figure 36):

- **Test Implementation.** In this activity, the Technical Tests for the technical scenarios are written using the specific testing language. This test is fed into the TDD to incrementally implement Technical Scenarios in the Technical Domain Context. Then, The Technical Scenarios are then moved to the In Progress state.

- **Code Implementation.** In this activity, the code implementation for each Technical Scenario should be completed. Then, the Technical Test is passed to check the validity of the code. The Technical Scenario is considered Done when the specific test is successfully executed using with the minimum amount of required code, accompanied by the appropriate refactoring process during the practice of TDD.
- **Commit Technical Scenarios.** In this activity, the technical scenario is saved and pushed to Enabler work-center to realize value of the Enabler. The Enabler is complete when all the technical scenarios have been moved to Done status.

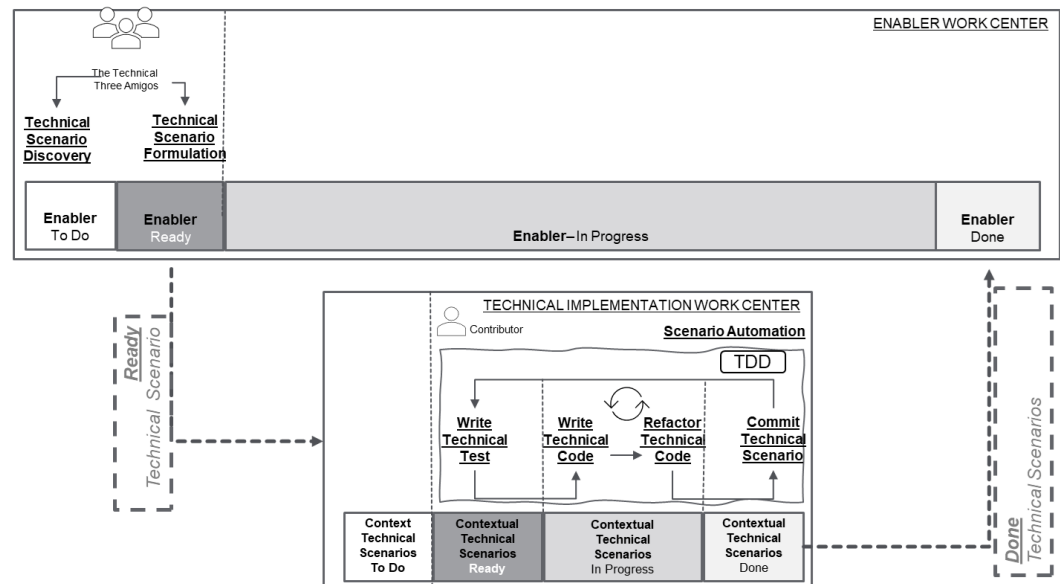


Figure 36. Technical Implementation work-center. This work-center is per Context.

The formal verification at this level is represented by the set of technical tests X_T . For every technical scenario S_T defined in an Enabler, there must be a corresponding set of tests that validate its execution:

$$\forall S_T, \exists X_T (T_{DC}(S_T)) \neq \emptyset \quad (17)$$

The completion of this layer marks the beginning of the “value realization” phase. In BLOW, value is not realized by simply writing code, but by the successful “ascent” of verified behaviors: from a passing TSCT in Layer 4, to a completed Enabler in Layer 3, to a satisfied business scenario in Layer 2, and finally, to a Completed User Story in Layer 1. This bottom-up realization ensures that by the time a feature reaches the stakeholder, it has been verified at every possible level of abstraction.

5. BLOW Metrics

The BLOW approach does not merely provide a systematic workflow; it establishes a scenario-driven measurement framework with specific metrics that provide insight into the state of the project and how software development is evolving over time. These metrics provide added value to large development teams working on a software product by enabling them to assess progress, identify potential risks, and reason about the remaining effort required to complete software delivery.

BLOW metrics are grounded in the definition of Stories and Enablers as Incremental Value Items, and the definition of Scenarios as the Unit of Value. Under this view, progress and effort are not estimated only through task completion, but are observed through the evolution of executable scenarios and their associated tests.

The metrics defined in this section are native to BLOW because they are derived from executable scenarios and their associated tests. Internal code-quality indicators, such as

Cognitive Complexity, Code Smells, and Code Violations, are not defined as native BLOW metrics; instead, they are used in the case study as external quality-related indicators to observe the behavior of the developed system over time.

5.1. Cost and Value

The application of BLOW in the software development process involves the iterative definition and implementation of different types of work items such as stories, business scenarios, Enablers and technical scenarios, across work-centers located at different nested layers, with the objective of obtaining a working code at the end of the process. In this context, Value and Cost are dual aspects of the same transformation process of work items. The processing of work items provides Value through the realization of executable behavior, while Cost is a measure of test-anchored work required to obtain the working code.

Considering a single Story, the BLOW approach involves the creation of the Business Scenarios and its implementation in specific Technical Domain Contexts defined in Contextual Analysis. The latter implies the creation of Business Scenario Contextual Tests (BSCT) for each Contextual Business Scenario and their functional implementation. These BSCT are designed to ensure the partial implementation of the Business Scenarios taking into account layers 1 and 2 of BLOW.

For instance, the example described in Figure 37 contemplates a Story that has been split into three different Business Scenarios within the Stories work-center. During the activity of Context Analysis, three Technical Domain Context were selected (Front, Back, and Backoffice) employing the Technical Three Amigos practice. Once the Business Implementation work-centers were activated for each Technical Domain Context, the corresponding BSCTs were assigned to the different contexts.

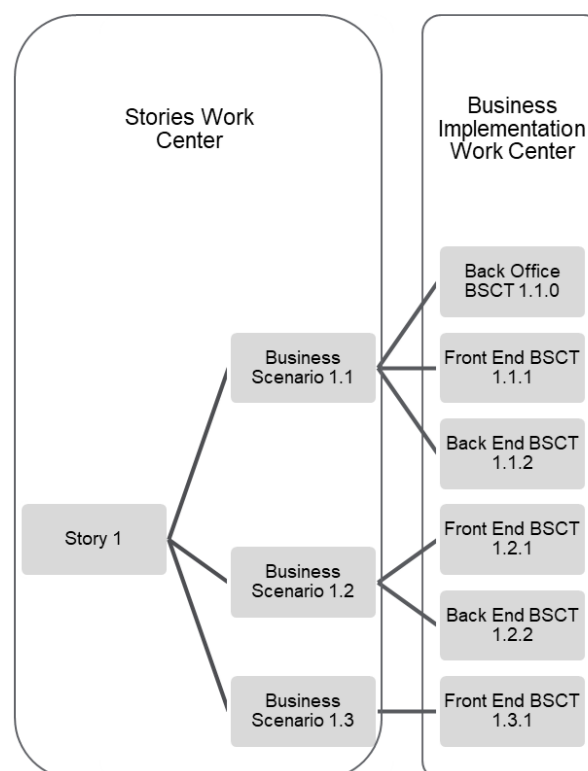


Figure 37. Business implementation of a Story through its Business Scenarios and their associated BSCT tests in layer 1 and 2 of the BLOW approach considering three Technical Domain Contexts (front, back, backoffice).

Given a Business Scenario SB belonging the set of Business Scenarios of a specific Story $SB(U)$:

$$SB \in SB(U) \quad (18)$$

For each Business Scenario SB , let $T'DC(SB)$ denote the non-empty set of Technical Domain Contexts impacted by that scenario:

$$\forall SB, \exists T'DC(SB) \neq \emptyset \quad (19)$$

For each impacted set of contexts, there exists a non-empty set of Business Scenario Contextualized Tests, denoted as $XB(T'DC(SB))$, required to verify the implementation of that Business Scenario within those contexts:

$$\forall T'DC(SB), \exists XB(T'DC(SB)) \neq \emptyset \quad (20)$$

Since every Contextual Business Scenario is associated with at least one BSCT, these tests can be used as a test-anchored proxy for the work required to implement the business behavior in the affected contexts. However, this cost only represents the business implementation effort and does not include the technical enablement required to prepare the product architecture.

Then, the Business Cost CB is defined as the set of BSCT that are depending of this Story U :

$$CB = |XB(T'DC(SB))| \quad (21)$$

Then, the Business Costs of a Story can be seen as the test-anchored effort required to implement its business behavior. Complementary, the Business value of a Story can be interpreted as the increment of the Unit of Value corresponding to Contextual Business Scenarios implemented within a Story.

Similarly, when a Business Scenario impacts one or more Enablers, there will also be Technical Scenarios linked to Technical Domain Contexts and associated tests to be implemented as it is shown in Figure 38. Notice that some BSCTs have no correspondence with any Enabler. This occurs when the Business Scenario is Ready to be implemented in the Technical Domain Context without any additional technical preparation.

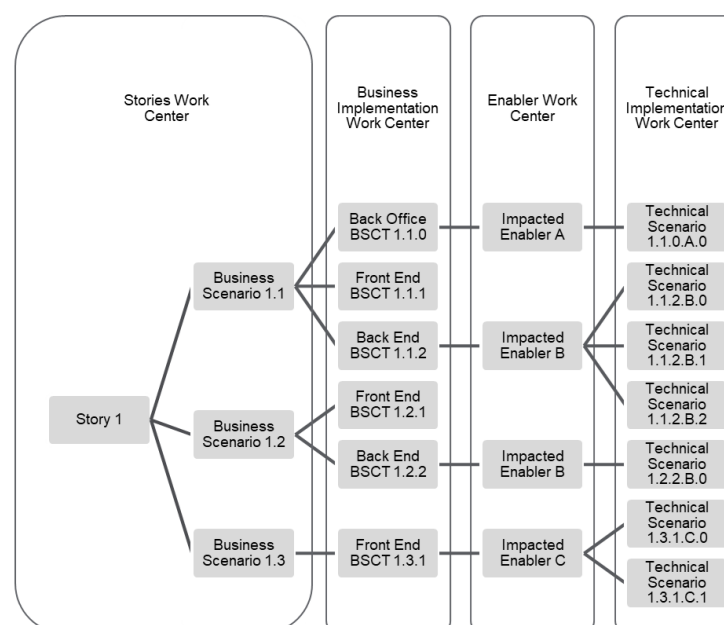


Figure 38. Extension of the business implementation of a Story including the Enabler and its Technical Scenarios in layer 3 and 4 of BLOW.

When considering the relationship between Business Scenarios and BSCTs, there is usually a one-to-many (1:N) association. However, as the Technical Scenarios depend on an Enabler in a specific Technical Domain Context, there is generally a correspondence one-to-one (1:1) between the Technical Scenarios and the Technical Tests.

Considering that each Technical Scenario ST has one and only one assigned Technical Test, the subset of necessary technical tests for a Story, denoted as $XT(T'DC(SB))$, is equivalent to the increment of technical value required for enablement:

$$CT = \Sigma \Delta |ST| \equiv |XT(T'DC(SB))| \quad (22)$$

The Total Cost (C) required to implement a single Story comprises the necessary business work to be done and the necessary technical work required to prepare the product for that aforementioned business work.

Thus,

$$CU = CB + CT = |XB(T'DC(SB))| + |XT(T'DC(SB))| \quad (23)$$

In other words, Total Cost can be measured by the total number of technical and business tests required to be implemented using TDD.

In the example shown in Figure 39, the Total Cost of User Story 1 is the sum of the number of BSCT and number of Technical Tests and, that is equal to 13. The Business Value of User Story 1 will be six, corresponding to the number of BSCT, while the Technical Value of the User Story 1 is seven.

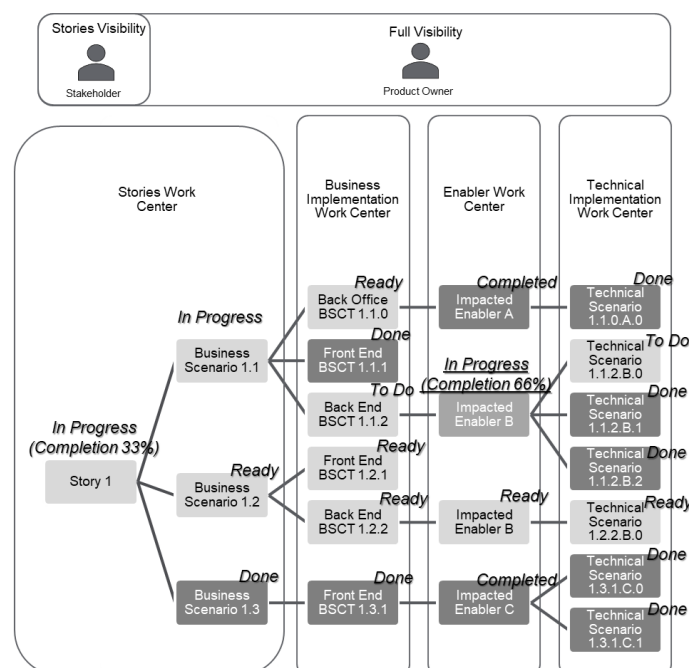


Figure 39. Status of the work items of the work-centers in all the layers for the implementation of the project described in Figure 38.

5.2. Business Scenario Done Degree

The Stakeholder can track the evolution of Story based on the successful implementation of Business Scenarios. In the case of the Figure 40, the Story progress will be at 33% of completion because only one Business Scenario has moved to Done status in Terminated stage.

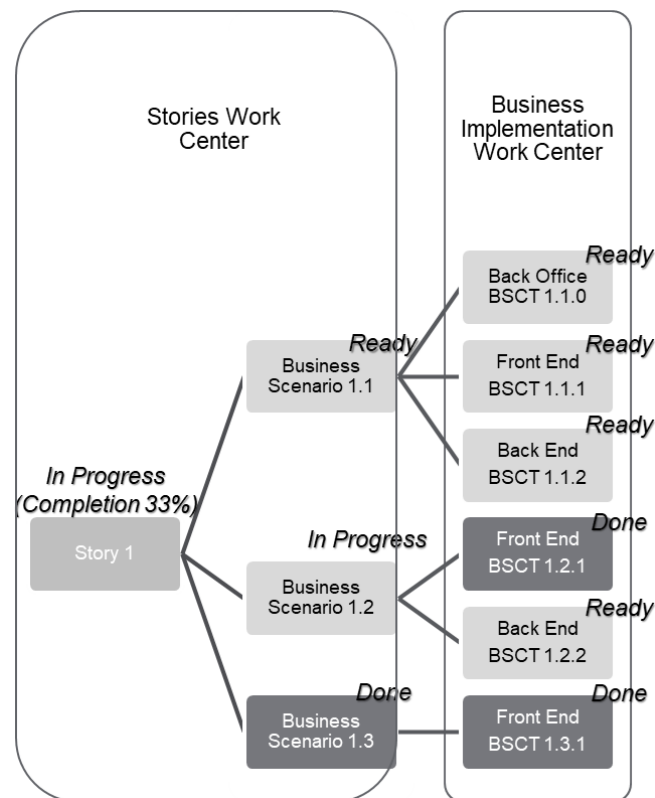


Figure 40. Status of the work items (Story and Business Scenarios) for the implementation of the example described in Figure 37 where a 33% of the Story is complete as only a third of the business scenarios are Done.

The scenario SB will be done when all BSCTs belonging to $X_B(T'DC(SB))$ pass. In other words, given a non-empty set of BSCT, $X_B(T'DC(SB))$ is as follows:

$$\forall SB, \exists X_B(T'DC(SB)) \neq \emptyset \quad (24)$$

And a subset of passed Business Scenario Contextualized Tests $X'_B(T'DC(SB))$ and a subset of non-passed Business Scenario Contextualized Tests $X''_B(T'DC(SB))$ is as follows:

$$X_B(T'DC(SB)) = X'_B(T'DC(SB)) \cup X''_B(T'DC(SB)) \quad (25)$$

Because a test can only be classified as passed or non-passed in this representation, these subsets are disjoint:

$$X'_B(T'DC(SB)) \cap X''_B(T'DC(SB)) = \emptyset \quad (26)$$

A business scenario SB is considered Done when all its BSCTs have passed. This condition is expressed as

$$\exists X_B(T'DC(SB)) | X''_B(T'DC(SB)) = \emptyset \wedge X'_B(T'DC(SB)) \neq \emptyset \quad (27)$$

With these considerations in mind, the Business Scenario Done Degree DSB of a Business Scenario SB is defined as the fraction of passed BSCT:

$$DSB = \frac{|X'_B(T'DC(SB))|}{|X_B(T'DC(SB))|} \quad (28)$$

In practice, DSB enables the development team to assess whether a business scenario is approaching readiness for stakeholder review. A DSB below 1 signals that additional

contextualized tests remain pending, providing an early indicator that the Scenario requires further development or testing effort before it can be considered Done.

5.3. Story Completion Degree

As described above, Stories are completed when all the existing Business Scenarios are in Done status. Thus, the Stakeholder can track the evolution of a requested given Story by checking how many of the Scenarios have been already Done. This metric is called Business Scenario Completion Degree. Since a Business Scenario can only be in statuses In Progress or Done, only those SB with a $DSB = 1$ can be considered as Done. There be a subset $S'B(U)$ of $SB(U)$ where $DSB = 1$

$$\forall SB(U) \exists S'B(U) | \forall SB \in S'B(U), DSB = 1 \quad (29)$$

Similarly, there can be a subset $S''B$ for those Scenarios which are not done such as:

$$\forall SB(U) \exists S''B(U) | \forall SB \in S''B(U), DSB \neq 1 \quad (30)$$

The set $SB(U)$ will be composed of $S'B(U)$ and $S''B(U)$ as follows:

$$SB(U) = S'B(U) \cup S''B(U) \quad (31)$$

where

$$SB(U) \neq \emptyset \quad (32)$$

and

$$S'B(U) \cap S''B(U) = \emptyset \quad (33)$$

Story Completion Degree DU is defined as the ratio between the cardinality on the set of Scenarios in Done state, and the total number of Scenarios:

$$DU = \frac{|S'B(U)|}{|SB(U)|} \quad (34)$$

From a decision-making perspective, DU allows Stakeholders to evaluate whether a Story is sufficiently mature for release or whether remaining Scenarios represent critical functional gaps. Unlike traditional percentage-complete metrics, DU is grounded in executable specifications, ensuring that reported progress corresponds to validated behavior rather than estimated effort.

For example, for a Story with four Business Scenarios, two of the Done state will be completed at 50%. An alternative definition of a Story Completion Degree based on expected Business Value as Business Value VB of a Story U is the cardinality of $Sb(U)$. In the example of Figure 38 where the status of all the work items (Story, Business Scenarios, Enabler and Technical Scenarios) is presented of a software product, it can provide useful different information to each team role of the project. For example, Stakeholder can track the current degree of implementation for a requested Story, while a Product Owner can see the full implementation visibility including both Stories and Enablers.

5.4. Incurred Cost

As the Stakeholder can monitor the progress of a Story based on the Story Completion Degree, the Product Owner can monitor implementation progress based on the status of existing tests.

Considering the Story implementation effort or necessary cost as the addition of Technical Tests and Business Scenario Contextualized Tests:

$$CU = |X_B(T'DC(SB))| + |X_T(T'DC(SB))| \quad (35)$$

Considering the set of passed Business Scenario Contextualized Tests $X'_B(T'DC(SB))$ and the set of passed Technical Scenarios $X'_T(T'DC)$, the incurred cost CI can be defined as

$$CI = |X'_B(T'DC(SB))| + |X'_T(T'DC(SB))| \quad (36)$$

It is hypothesized that the CU per Story should decrease as the Enablers were maturing. The main hypothesis is as Enablers evolve, new need of Technical Scenarios number will lower, so in the future

$$\lim_{|Sb(U)| \rightarrow \infty} |X'_T(T'DC(SB))| = 0 \quad (37)$$

The development of CU in relation to CI is presented in this article through a proof of concept based on a single sprint, followed by its application over an extended period in a real-life project using BLOW, which was later transitioned to a traditional agile team.

As a main implication, the deployment of Technical Scenarios is conceptualized as an enabling investment intended to mitigate the growth of delivery costs for subsequent business scenarios.

5.5. Illustrative Example: E-Commerce Shopping Cart

To demonstrate these metrics, a simple case of an e-commerce shopping cart which is defined through 15 Business Scenarios. Figure 41 shows a snapshot of the implementation status for a Story in a project after the execution of the tests during the software development process. The Story (BaskerFeature) is defined through 15 Business Scenarios named as Deltas. The architecture implies three different Technical Domain Contexts with 36 Technical Scenarios where CartOperationsServices defined through 7 Technical Scenarios, CatalogueServices defined through 28 Technical Scenarios and ServiceFactory defined through 1 Technical Scenario.

After the execution of tests, the Story has passed the tests of 7 business scenarios of 15 Business Scenarios and all the 36 technical scenarios. Then, a Stakeholder can know the current status of the development process by determining the Story Completion Degree and the Incurring Costs. In fact, the Story Completion Degree is

$$\frac{7X_B}{15X_B} = 0.47 = 47\% \quad (38)$$

and the Incurred Cost is

$$CI = 36X_T + 7X_B = 43X \quad (39)$$

These metrics enable actionable decisions during development. For instance, a low DU combined with a high CT may signal substantial architectural investment relative to delivered business value, prompting the team to review the Enabler strategy. Conversely, a high DU with low CT may indicate that business functionality is being delivered with limited new technical enablement, which can be positive when existing Enablers are mature, but may also suggest the need for additional Enablers before the Story is released to production.

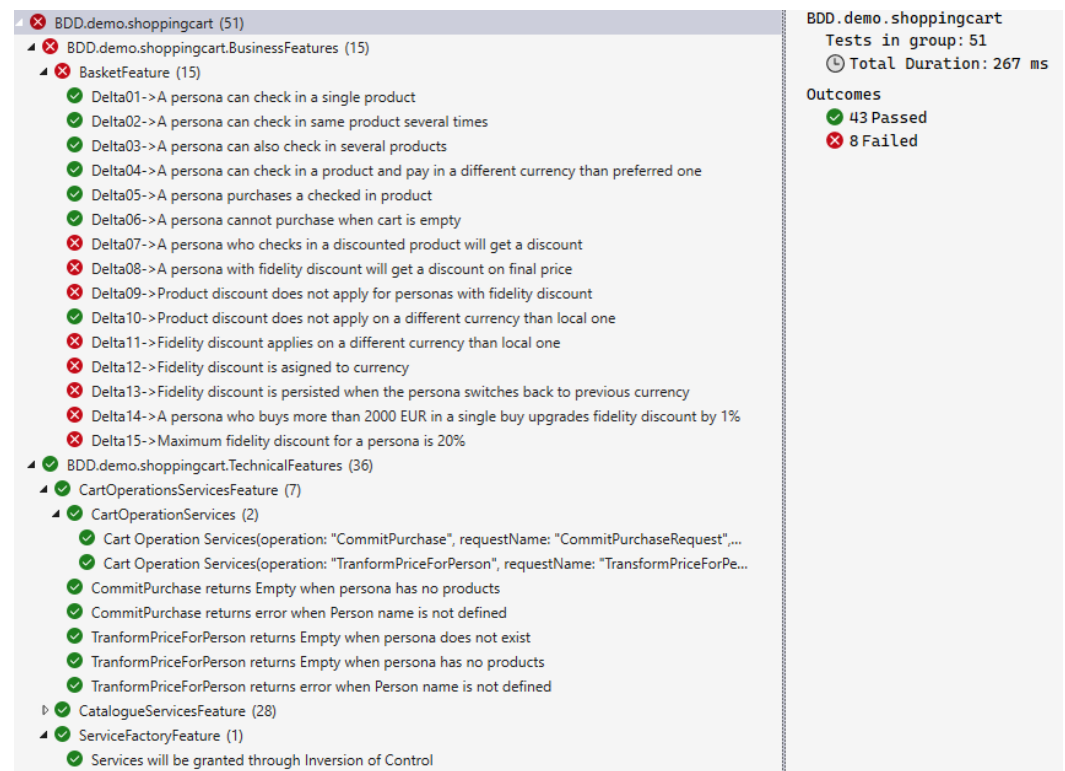


Figure 41. Snapshot of the implementation status of a Story for a project.

6. Case Study

To explore the practical application of BLOW as a systematic engineering approach and assess its scalability, a case study was conducted in two phases of system maturity: (1) an initial single-sprint proof of concept (PoC) and (2) an 18-month long-term project implementation in a production environment. The investigation aimed to validate three core hypotheses:

- BLOW can reduce the gap between business and technical domains.
- As the domain maturity increases, the value-to-cost ratio may improve through the reuse of technical Enablers.
- Internal code-quality indicators may remain more stable over time, rather than exhibiting the continuous growth commonly associated with software evolution.

6.1. Context and Setting

The case study site was selected through convenience sampling: the research team had direct access to the organization, and the development team was willing to adopt a new approach. The case represents a typical scenario in which an organization faces common challenges, fragmented BDD adoption, unclear role boundaries between business and technical stakeholders, and test automation not systematically linked to business requirements, that are representative of the problems BLOW is designed to address. Two development teams participated in the study across different phases. Both teams had comparable professional backgrounds and experience levels in software engineering. The principal distinction was that Team 1 received extensive training in BLOW practices and conducted the initial proof of concept, whereas Team 2 followed established Scrum practices without specific ATDD, TDD, or BDD training. Prior to BLOW adoption, the organization followed ad-hoc development practices without formal coordination between business requirements and test automation.

Phase 1: PoC. The PoC was conducted during a single sprint and focused on implementing one User Story within a single Technical Domain Context. This phase represents an early stage of the case study and serves to observe initial feasibility and to train participants in the core principles of BLOW, including scenario discovery, formulation, and execution.

The PoC involved three participants assuming five roles: Stakeholder, Test Leader, Architect, Contributor, and DevOps expert. Development and testing were carried out using .NET 6.0, SpecFlow for executable scenarios, GitHub for version control, and SonarCloud for static code analysis.

Phase 2: Long-term Project. The second phase applied BLOW to a live production system over an extended period (18 month) spanning multiple releases. This phase targets the study of longer-term evolutionary effects, including scalability, maintainability, and integration with enterprise processes. In addition to the original roles, two auxiliary roles were included: a Product Owner responsible for backlog management and prioritization, and a Data Scientist supporting data analytics and visualization.

The system was implemented in Python 3.12, using Behave for executable Gherkin scenarios and GitHub Enterprise for version control and CI/CD via GitHub Actions. Code-quality and security gates were enforced through an on-premise SonarQube instance. Additional tooling included Jira and XRay for story and scenario traceability and Tableau dashboards for aggregated visualization of execution and quality metrics.

During the early months of this phase, BLOW practices guided significant refactoring and test-first consolidation before full instrumentation was in place.

6.2. Study Protocol

The study observes the evolution of scenario-based development artifacts and quality indicators under a fixed operating setup.

For each Story entering development, a set of Business Scenarios and, where required, Technical Scenarios was defined before implementation. The study tracks when scenarios become passing in the shared Continuous Integration (CI) environment; local executions during development are excluded.

A Business or Technical Scenario is counted as delivered at the time it first succeeds in CI. No scenario is counted more than once. The introduction of new scenarios is recorded over time as a proxy for value (Business Scenarios), and cost (Technical Scenarios).

All commits trigger CI execution with enforced quality gates. Commits failing scenario execution or static-analysis thresholds do not contribute to the observed dataset.

Across both study phases, scenario classification, execution criteria, and observation rules remain unchanged. No parallel control process, additional estimation techniques, or alternative tracking mechanisms are introduced.

6.3. Data Collection and Instrumentation

Data collection relies exclusively on automated and tool-generated artifacts produced during the development process.

Scenario execution data are obtained from Continuous Integration (CI) pipelines, recording the timestamp at which a scenario first passes in the shared CI environment. Subsequent executions of the same scenario are not considered. Local executions during development are excluded from the dataset.

Internal code-quality indicators are collected from SonarQube at each CI execution in the PoC and SonarQube in the long-term project. The monitored indicators include Cognitive Complexity, Code Smells, and Code Violations, as defined by the active SonarQube rule set at the time of measurement. These indicators are treated as external quality-related observations rather than as native BLOW metrics.

Traceability between Stories, Scenarios, and execution results is supported through Jira and XRay. However, due to limitations in synchronizing local scenario development with Jira in real time, Jira-based artifacts are treated as auxiliary sources rather than as the primary basis for temporal analysis.

Aggregated views of scenario evolution and quality indicators are produced using Tableau dashboards. All the reported measurements are derived from CI and static-analysis outputs and are reported in relative terms consistent with the selected observation windows.

No manual data entry or post-hoc correction of collected data is performed.

6.4. Results

6.4.1. Single-Sprint Proof of Concept

Figure 42 illustrates the temporal evolution of passing scenarios during the PoC sprint. The figure reports four observed series: passing Technical Scenarios in blue, passing Business Scenarios in yellow, remaining Business Scenarios in green, and Total Cost computed as the cumulative number of scenarios introduced in red.

At the beginning of the sprint, the curves for passing technical scenarios and Total Cost curves overlap, indicating that initial development effort was devoted exclusively to technical enablement. During this early interval, no Business Scenarios reached a passing state in the CI environment.

After several commits, Business Scenarios began to pass without the introduction of additional Technical Scenarios. During this period, multiple Business Scenarios transitioned to a passing state while the technical scenario count remained unchanged, reflecting the reuse of previously introduced technical capabilities. These transitions occurred within the same sprint and correspond to distinct implementation intervals visible in the CI execution timeline.

Later in the sprint, newly identified constraints triggered the introduction of additional Technical Scenarios. This resulted in a temporary increase in Total Cost while the number of passing Business Scenarios remained stable. Following the successful execution of these new Technical Scenarios, further Business Scenarios subsequently passed in CI.

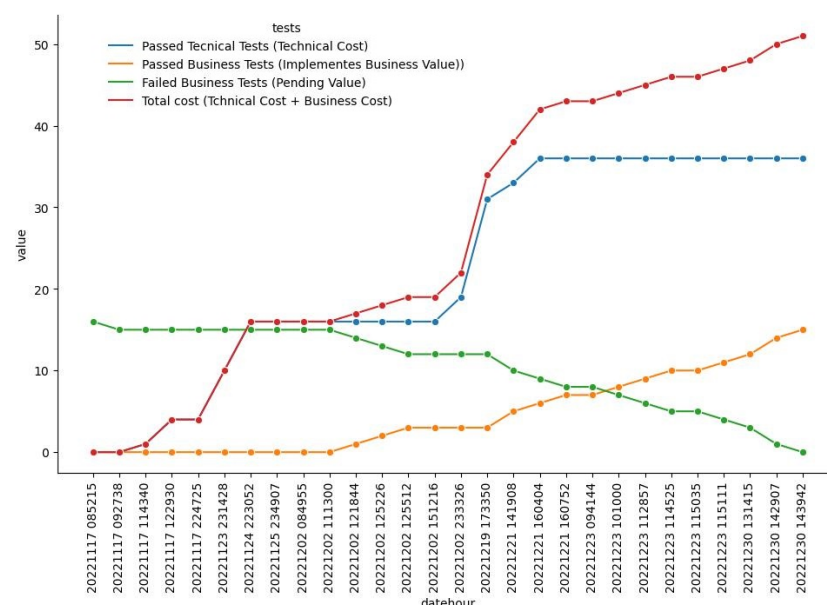


Figure 42. Evolution of passed tests and pending tests.

Overall, the PoC exhibits a repeated pattern in which technical enablement precedes business behavior, followed by periods where accumulated technical effort is progressively converted into delivered Business Scenarios within the same sprint.

In addition to scenario evolution, internal code quality was monitored throughout the sprint using SonarCloud, focusing on Code Smells and Code Violations (Figure 43). These indicators are not interpreted as direct proof of quality improvement, but as quality-related observations that complement the scenario-driven analysis.

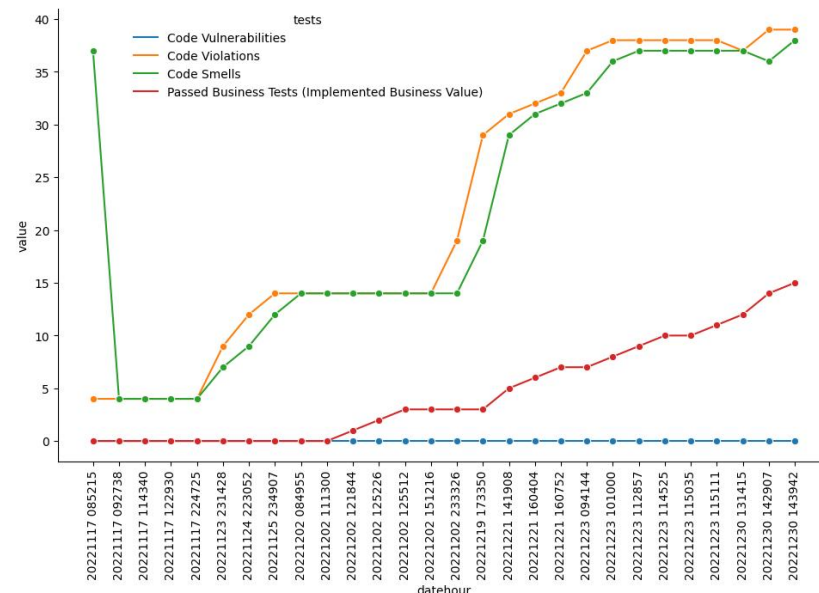


Figure 43. Timeline of the quality evolution on the solution.

Most reported issues emerged during the early commits of the sprint, corresponding to the initial construction of the solution. As development progressed, the rate of new Code Smells and Violations decreased, and their cumulative values tended to stabilize into a plateau. Within the limited temporal scope of a single sprint, functionality growth was not accompanied by a continuous increase in maintainability indicators.

Within the limited temporal scope of a single sprint, these observations indicate that early functionality growth was not accompanied by a continuous increase in maintainability issues.

6.4.2. Long-Term Project (Live Deployment)

The long-term project phase extends the observation of the same case study over an 18-month period in a live production environment. The results are derived from the accumulated evolution of passing Business Scenarios, Technical Scenarios, and internal code-quality indicators recorded throughout this interval (Figures 44 and 45).

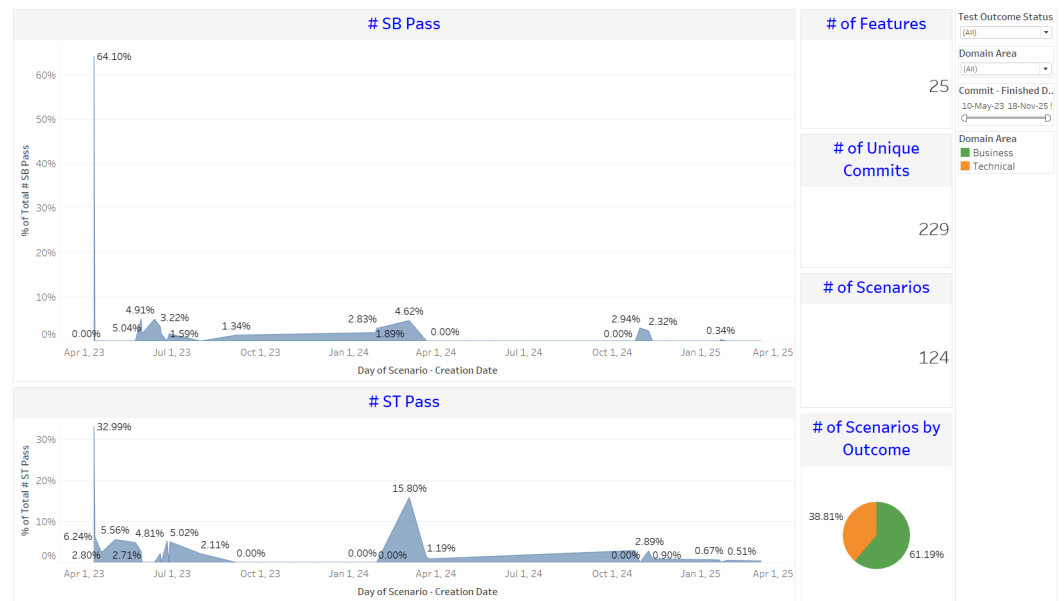


Figure 44. Tableau dashboard snapshot used during the long-term project. The upper series (# SB Pass) represents business-oriented scenarios; the lower series (# ST Pass) represents technical scenarios. The tiles on the right summarize features, unique commits, total scenarios, and the distribution of outcomes for the selected filter range.

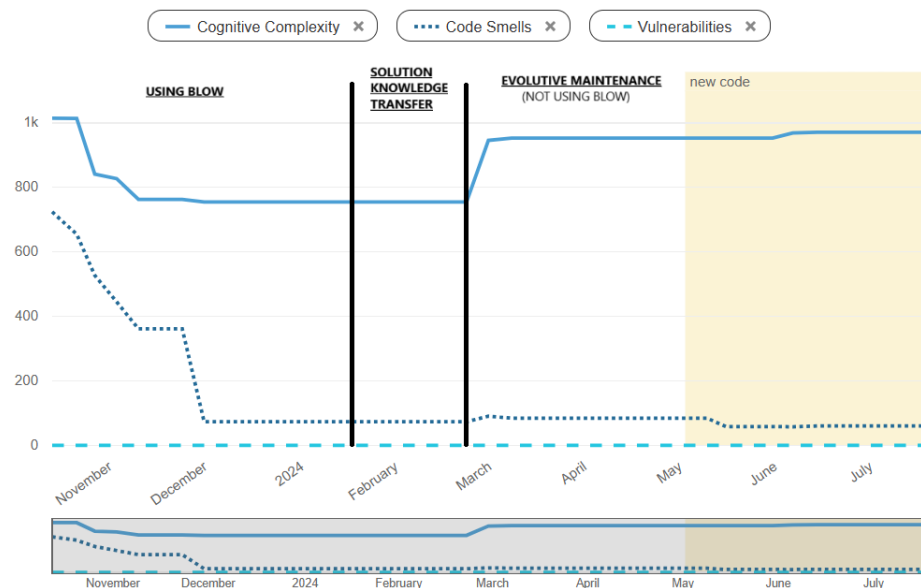


Figure 45. SonarQube time series of code quality over the long-term project, including Cognitive Complexity and Code Smells. During the period in which BLOW practices were applied, both signals stabilized or improved; after the March 2024 handover to a team following traditional Scrum practices, both metrics began trending upward.

Based on the observed scenario evolution and quality measurements, the longitudinal behavior of the system can be organized into four distinct empirical phases.

Phase 1: Initial Consolidation

During the initial months of the long-term phase, a substantial number of Technical Scenarios transitioned to a passing state in the Continuous Integration (CI) environment before a corresponding increase in passing Business Scenarios. This period coincided with intensive refactoring and test-first consolidation activities. At the first consolidated

integration snapshot on 1 April 2023, Business Scenarios represented **64.10%** of all the passing scenarios, while Technical Scenarios accounted for **32.99%**, as reported in Figure 44.

Phase 2: Sustained Growth Under BLOW

Following the initial consolidation, the rate of growth of passing Business Scenarios increased while the introduction of new Technical Scenarios progressed more gradually. Over successive releases, multiple Business Scenarios passed in CI without proportional increases in Technical Scenarios, leading to a gradual shift in their relative distribution. In the corresponding dashboard snapshot taken during this phase, the system comprised *25 features*, *229 unique commits*, and a total of *124 scenarios*, with approximately **61.19%** passing versus **38.81%** non-passing.

Throughout this interval, code-quality indicators collected via SonarQube exhibited stable or plateau-like behavior. Cognitive Complexity and Code Smells did not show a sustained upward trend despite continuous functionality growth and ongoing scenario introduction (Figure 45).

Phase 3: Transition (Handover)

In March 2024, the system was transferred to a development team that did not apply BLOW practices. This event constitutes a clear transition point in the observed longitudinal data, separating two phases characterized by different development practices. Although the study was not designed as a controlled experiment, the March 2024 handover provides a natural observational contrast between two operating regimes: the period in which BLOW governance practices were active (Team 1) and the subsequent period where Team 2 maintained the system using standard Scrum workflows. During a brief transition window, Team 1 provided architectural support. This structural change supports a before-and-after analysis of scenario evolution and quality-indicator trajectories, while still requiring caution in causal interpretation due to the absence of randomization, a parallel control group, and full experimental control.

Phase 4: Post-BLOW Evolution

After the handover, the observed evolution of scenarios changed noticeably. Technical Scenarios began to accumulate at a higher relative rate than business scenarios, while the growth of passing business scenarios slowed. In parallel, code-quality indicators exhibited increasing trends in Cognitive Complexity and Code Smells, deviating from the stable patterns observed during the preceding phase (Figure 45).

Across the entire period, all the scenarios were recorded at the moment of their first successful execution in the shared CI environment, in accordance with the study protocol. The reported results therefore reflect the temporal ordering and relative accumulation of executed scenarios and quality indicators over the full lifetime of the system.

7. Limitations of the Results

This study is based on a single longitudinal case observed in a real production environment, which is subject to several limitations that must be considered when interpreting the results. First, the study was conducted without a parallel control group. This limits the strength of causal claims and the direct generalization of the findings to other organizational or technological contexts. Observed patterns are subject to potential confounding factors, such as changes in team composition and organizational shifts following the March 2024 handover. Consequently, the results represent observational associations within a specific ecosystem rather than universal effects.

The analysis relies on scenario-based and static-analysis proxies to approximate cost, value, and internal quality. Although these indirect measures are consistent with

scenario-driven development practices, they do not directly capture actual engineering effort, delivery time, or user-perceived value.

Furthermore, practical tooling constraints, particularly the lack of seamless synchronization between local development workflows and Jira/XRay, restricted fine-grained temporal observability of scenario evolution, leading the analysis to focus on aggregated trends rather than detailed event-level timing.

The longitudinal observations are descriptive in nature and do not include statistical hypothesis testing or formal time-series modeling. However, the consistency of these patterns across 18 months of production provides a relevant empirical basis for further investigation. Consequently, the reported patterns should be interpreted as indicative evidence that motivates further replicated and controlled studies rather than as definitive proof of effectiveness.

Several alternative explanations must be considered when interpreting the observed patterns. Team learning effects may have contributed to quality-related changes independently of BLOW, as the team's familiarity with the codebase and tools naturally increased over time. Tooling configuration, including evolving SonarQube rule sets and CI pipeline refinements, might have influenced the measured quality indicators. Furthermore, project-phase effects are highly relevant: the initial months involved intensive refactoring and test-first consolidation, which may have front-loaded quality-related changes independently of the operating model applied. The March 2024 handover introduced a confounding organizational shift, as the receiving team brought different skills, priorities, and working practices. Although the handover provides a natural observational contrast between two regimes, the study remains limited to a single organization and cannot support broad generalization without replication across diverse teams, domains, and organizational contexts. Consequently, these findings should be interpreted as exploratory indicators rather than definitive causal evidence.

8. Discussion

This section examines the implications of the proposed BLOW systematic approach in relation to the research problem addressed in this study: the ongoing ambiguity and inconsistent use of BDD, ATDD and TDD within agile software development. The discussion interprets the empirical evidence from the case study, positions BLOW within current research, and assesses its impact on software sustainability.

The survey results discussed in Section 2 showed that even experienced practitioners do not share a common understanding of the definitions of BDD, ATDD, and TDD, nor of how these approaches relate to one another. This conceptual fragmentation extends beyond terminology and is reflected in inconsistent adoption, partial implementations, and practices that are frequently abandoned over time.

The case study observations are consistent with the view that BLOW helps mitigate this fragmentation by explicitly defining BDD as a methodological layer that coordinates supporting engineering frameworks and practices rather than competing with them. By situating ATDD and TDD at distinct organizational and technical levels and embedding them within a layered operating model, BLOW provides a concrete structure for aligning business intent and technical execution. This explicit positioning appears to reduce ambiguity and supports more disciplined and sustained adoption, particularly in environments where earlier BDD initiatives had stalled or failed to scale.

One central empirical pattern observed across both the PoC and the long-term project is the temporal separation between technical enablement and business-facing behavior. In both phases, Technical Scenarios consistently precede the delivery of Business Scenarios, and once enabling infrastructure stabilizes, multiple Business Scenarios can be delivered

without proportional increases in technical effort. This behavior is consistent with the hypothesis that early, explicit investment in technical capabilities facilitates more efficient delivery of later features.

At a larger temporal scale, the longitudinal observations further suggest that this pattern persists as the system matures. Periods characterized by sustained growth of business scenarios without corresponding increases in technical scenarios coincide with phases in which reusable Enablers are already established. In contrast, when new constraints emerge or Enablers are absent, additional technical work is observed before further business value is realized.

A second notable pattern concerns internal quality. During periods in which BLOW practices were actively applied, both short-term and long-term observations show stabilization or plateau-like behavior in Cognitive Complexity and Code Smells despite continued functionality growth. Although this study does not establish causal mechanisms, the consistency of this behavior across different time scales suggests that continuous, scenario-driven refactoring guided by executable specifications may function as a structural constraint on uncontrolled architectural degradation. This constraint aligns with the well-documented effects of TDD practices, specifically the disciplined cycle of writing unit tests before code, producing minimal working implementations, and applying systematic refactoring [24,58,59]. These practices, which are embedded within BLOW through its integration of TDD as a supporting framework, have been independently associated with reduced defect density and improved internal code quality across industrial settings [28,60]. While this quality plateau may partially reflect the inherent benefits of sustained TDD discipline rather than effects exclusive to BLOW's organizational structure, the post-handover increase in quality-related indicators observed when the receiving team, which had not received specific BLOW or TDD training, assumed maintenance provides a natural observational contrast that reinforces the relevance of integrating technical discipline within an explicit governance structure.

From a methodological perspective, BLOW introduces a new angle by treating Scenarios as the primary Unit of Value. This departs from classic agile metrics such as story points or effort estimates, which frequently lack clear traceability to concrete business results. By linking progress, cost, and completeness directly to executable scenarios, BLOW establishes a more objective and observable basis for monitoring development.

This scenario-centered view complements existing agile and scaling approaches by making explicit the relationship between business value, technical enablement, and executable verification. In many Scrum-based implementations, progress is often tracked through abstractions such as story points, while technical quality is monitored separately through test or code-quality indicators. BLOW connects these views through test-anchored metrics across work-centers. Similarly, while SAFe introduces Enablers to represent architectural or technical work, BLOW provides a more explicit cross-layer accountability mechanism for governing how such enablement work is discovered, formulated, implemented, and related back to business scenarios. In this sense, BLOW does not replace Scrum or SAFe, but provides an operational structure for coordinating scenario-driven value delivery and technical enablement.

The explicit distinction between business and technical scenarios further enables a structured approach to reasoning about validation and verification activities. This supports the interpretation of BDD as a facilitating methodology that integrates ATDD and TDD rather than replacing them, and provides a clearer framework for analyzing the dynamics of scenario-driven development in practice.

Beyond technical considerations, the findings have implications for organizational design and cross-domain collaboration. The introduction of layered work-centers and

explicit roles contributes to reducing the “gray zones” often observed between business and technical domains. Structured collaboration practices, such as the Three Amigos and the Technical Three Amigos, act as boundary-crossing mechanisms that support shared understanding and knowledge transfer.

The layered coordination model introduces additional organizational structure that must be managed carefully. While this structure appears beneficial for large or distributed organizations where specialization and scale amplify communication challenges, it also introduces coordination overhead that, if not governed appropriately, could lead to rigidity. These trade-offs should be considered when adopting BLOW in different organizational contexts.

Prior research has variously characterized BDD as an extension of TDD, a variant of ATDD, or a collection of practices centered on executable specifications. BLOW builds on these perspectives by offering a unifying operating model that situates existing practices within a coherent organizational and methodological structure. Rather than introducing new testing techniques, the primary contribution of this work lies in the systematic organization of how established practices interact and how they support value realization and organizational accountability.

Taken together, the discussion suggests that BLOW represents a meaningful step toward clarifying long-standing uncertainties in scenario-driven development while providing a foundation for more rigorous empirical evaluation and future methodological refinement.

9. Conclusions and Future Work

This paper addressed the persistent ambiguity surrounding the practical adoption of behavior-driven development (BDD) and its relationship with Acceptance Test-driven Development (ATDD) and Test-driven Development (TDD) in real software engineering contexts. To tackle this problem, we introduced BLOW as a systematic approach that establishes an layered operating model for organizing scenario-driven software development across domains of accountability. BLOW explicitly separates business specification and technical enablement activities while preserving end-to-end traceability. By providing an operational distinction between methodology, frameworks, collaboration practices, and automated tools, BLOW helps clarify the definitional ambiguities documented in the software engineering literature and establishes BDD as the coordinating methodology between ATDD and TDD.

The primary contribution of this work is methodological and organizational rather than tool-oriented. BLOW reframes BDD as a coordinating methodology that integrates rather than replaces existing practices, providing a structured way to reason about cost, value, and quality through executable scenarios. By distinguishing Business Scenarios from Technical Scenarios and making technical enablement explicit, the approach increases observability and reduces ambiguity in large scale and long lived development contexts. Furthermore, the contribution is articulated through three main elements: (a) native, scenario-anchored metrics that utilize executable scenarios as formal units of value to support progress tracking and decision-making; (b) the Technical Three Amigos as a specialized collaboration practice for technical domain alignment; and (c) a layered structure of nested work-centers governed by explicit push–pull interaction dynamics.

The longitudinal case study provides empirical grounding for the proposal. Observations from both a single-sprint PoC and an 18-month production deployment reveal consistent patterns in which technical enablement precedes business value delivery, followed by periods where accumulated technical investment supports sustained growth of business scenarios. At a larger temporal scale, the study also shows that changes in

the applied operating model are reflected in measurable shifts in scenario evolution and internal quality indicators. Although these observations do not establish causal proof, they are consistent with the intended design goals of BLOW and provide exploratory evidence for its practical applicability.

Overall, the findings suggest that BLOW is a promising approach for organizing scenario-driven development in environments where coordination between business and technical domains is critical. By emphasizing observability, explicit enablement, and sustained operational discipline, BLOW contributes a structured response to long-standing challenges in the industrial adoption of BDD. While further replication is required, the observed results indicate that BLOW can support traceability, governance, and alignment between stakeholder intent and technical implementation in complex software systems.

Future work will focus on replicating the study across additional projects and organizational contexts, strengthening analytical rigor through complementary quantitative techniques, and improving tool support for scenario-centric observability. Further research should also investigate the integration of BLOW with established agile scaling approaches and development governance models, in order to better understand its applicability in different organizational settings. Additional empirical studies examining stakeholder collaboration, organizational impact, and long-term maintenance outcomes would help to better assess the broader applicability of BLOW across diverse development environments.

Author Contributions: Conceptualization, N.A.-A.; methodology, N.A.-A. and J.A.H.-T.; validation, J.A.H.-T., P.S. and M.A.O.-R.; formal analysis, J.A.H.-T., P.S. and M.A.O.-R.; investigation, N.A.-A.; resources, N.A.-A.; writing—original draft preparation, N.A.-A. and J.A.H.-T.; writing—review and editing, J.A.H.-T., P.S. and M.A.O.-R.; visualization, N.A.-A. and M.A.O.-R.; supervision, J.A.H.-T., P.S. and M.A.O.-R.; project administration, J.A.H.-T., N.A.-A. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by Schneider Electric and the University of Granada.

Data Availability Statement: The data presented in this study are available upon request from the corresponding author. The data are not publicly available due to privacy and confidentiality.

Acknowledgments: This article is part of a knowledge series created in strong collaboration with Schneider Electric’s community of experts, plus the collaboration with University of Granada.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

API	Application Programming Interface
ATDD	Acceptance Test-driven Development
ATM	Automated Teller Machine
BDD	Behaviour-driven Development
BLOW	BDD in a Layered Organization of work-centers
BSCT	Business Scenario Contextualized Test
DSL	Domain Specification Language
IDE	Integrated Development Environment
MVP	Minimum Viable Product
PIN	Personal Identification Number
TDD	Test-driven Development
RQ	Research Question
SAFe	Scaled Agile Framework
SDLC	Software Development Lifecycle
TSL	Test Specification Language

References

- Hoy, Z.; Xu, M. Agile Software Requirements Engineering Challenges-Solutions—A Conceptual Framework from Systematic Literature Review. *Information* **2023**, *14*, 322. <https://doi.org/10.3390/info14060322>.
- Dikert, K.; Paasivaara, M.; Lassenius, C. Challenges and success factors for large-scale agile transformations: A systematic literature review. *J. Syst. Softw.* **2016**, *119*, 87–108. <https://doi.org/10.1016/j.jss.2016.06.013>.
- Farooq, M.S.; Omer, U.; Ramzan, A.; Rasheed, M.A.; Atal, Z. Behavior Driven Development: A Systematic Literature Review. *IEEE Access* **2023**, *11*, 88008–88024.
- Cui, J. A comparative study on the impact of test-driven development (TDD) and behavior-driven development (BDD) on enterprise software delivery effectiveness. *arXiv* **2024**, arXiv:2411.04141.
- Beck, K. *Test-Driven Development: By Example*; Addison-Wesley Professional: Boston, MA, USA, 2003.
- North, D. Introducing BDD. *Better Softw. Mag.* **2006**, *8*, 27–33.
- Larman, C.; Vodde, B. *Practices for Scaling Lean & Agile Development: Large, Multisite, and Offshore Product Development with Large-Scale Scrum*; Addison-Wesley Professional: Boston, MA, USA, 2010.
- Barus, A.C. The implementation of ATDD and BDD from Testing Perspectives. *J. Phys. Conf. Ser.* **2019**, *1175*, 012112.
- Moe, M.M. Comparative Study of Test-Driven Development (TDD), Behavior-Driven Development (BDD) and Acceptance Test-Driven Development (ATDD). *Int. J. Trend Sci. Res. Dev.* **2019**, *3*, 231–234.
- Diepenbeck, M.; Kühne, U.; Soeken, M.; Drechsler, R. Behaviour Driven Development for Tests and Verification. In *Tests and Proofs*; Seidl, M., Tillmann, N., Eds.; Springer: Cham, Switzerland, 2014; pp. 61–77.
- Cavalcante, M.G.; Sales, J.I. The Behavior Driven Development Applied to the Software Quality Test. In Proceedings of the 2019 14th Iberian Conference on Information Systems and Technologies (CISTI), Coimbra, Portugal, 19–22 June 2019; pp. 1–4.
- Chandorkar, A.; Patkar, N.; Sorbo, A.D.; Nierstrasz, O. An Exploratory Study on the Usage of Gherkin Features in Open-Source Projects. In Proceedings of the 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), Honolulu, HI, USA, 15–18 March 2022.
- Fontela, C.; Garrido, A. Connection between safe refactorings and acceptance test driven development. *IEEE Lat. Am. Trans.* **2013**, *11*, 1238–1244.
- Gärtner, M. *ATDD by Example: A Practical Guide to Acceptance Test-Driven Development*; Addison-Wesley: Boston, MA, USA, 2012.
- Smart, J.F.; Molak, J. *BDD in Action: Behavior-Driven Development for the Whole Software Lifecycle*; Simon and Schuster: New York, NY, USA, 2023.
- Contan, A.; Dehelean, C.; Miclea, L.C. Test automation pyramid from theory to practice. In Proceedings of the 2018 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR), Cluj-Napoca, Romania, 24–26 May 2018; pp. 1–5.
- Pulido, N.M.A. Applying Behavior Driven Development Practices and Tools to Low-Code Technology. Master's Thesis, Universidade NOVA de Lisboa, Lisbon, Portugal, 2019.
- Adzic, G. *Specification by Example: How Successful Teams Deliver the Right Software*; Simon and Schuster: New York, NY, USA, 2011.
- Naik, H. Behavior Driven Development: An Effective Technical Practice to Develop Good Software. *Int. J. Comput. Appl.* **2016**, *149*, 23–27.
- Oliveira, G.; Marczak, S. On the Understanding of BDD Scenarios' Quality: Preliminary Practitioners' Opinions. In *Requirements Engineering: Foundation for Software Quality*; Springer: Cham, Switzerland, 2018.
- Hedlund, G. A model of knowledge management and the N-form corporation. *Strateg. Manag. J.* **1994**, *15*, 73–90.
- Michael, P. *Personal Knowledge: Towards a Post-Critical Philosophy*; Harper & Row: New York, NY, USA, 1964.
- Bottici, C. Mythos and Logos: A Genealogical Approach. *Epoché J. Hist. Philos.* **2008**, *13*, 1–24.
- Maximilien, E.M.; Williams, L. Assessing test-driven development at IBM. In Proceedings of the 25th International Conference on Software Engineering, 2003. Proceedings, Portland, OR, USA, 3–10 May 2003; pp. 564–569.
- Tahchiev, P.; Leme, F.; Massol, V.; Gregory, G. *JUnit in Action*, 2nd ed.; Manning Publications Co.: Shelter Island, NY, USA, 2010.
- Agile Alliance. What Is Test-Driven Development. 2021. Available online: <https://www.agilealliance.org/glossary/tdd> (accessed on 29 September 2021).
- Astels, D. *Test Driven Development: A Practical Guide*; Prentice Hall: Saddle River, NJ, USA, 2003.
- Williams, L.; Maximilien, E.M.; Vouk, M. Test-driven development as a defect-reduction practice. In Proceedings of the 14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003, Denver, CO, USA, 17–20 November 2003; pp. 34–45.
- Soeken, M.; Wille, R.; Drechsler, R. Assisted behavior driven development using natural language processing. In *Objects, Models, Components, Patterns, Proceedings of the International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*; Springer: Berlin/Heidelberg, Germany, 2012; pp. 269–287.
- Agile Alliance. What Are the Three Amigos. 2021. Available online: <https://www.agilealliance.org/glossary/three-amigos> (accessed on 29 September 2021).

31. Wang, Y.; Wagner, S. Combining stpa and bdd for safety analysis and verification in agile development: A controlled experiment. In *Agile Processes in Software Engineering and Extreme Programming: Proceedings of the 19th International Conference, XP 2018, Porto, Portugal, 21–25 May 2018*; Proceedings 19; Springer: Cham, Switzerland, 2018; pp. 37–53.
32. Gómez, L.A.C. Analysis of the Impact of test Based Development Techniques (TDD, BDD, AND ATDD) to the Software Life Cycle. Ph.D. Thesis, Instituto Politecnico de Leiria (Portugal), Leiria, Portugal, 2018.
33. Aggarwal, V.; Singh, M. Acceptance Test Driven Development. *J. Adv. Comput. Commun. Technol.* **2014**, *2*, 1–4.
34. Axelrod, A. *Complete Guide to Test Automation*; Apress: Matan, Israel, 2018.
35. Behavior-Driven.org. 2021. Available online: <https://www.agilealliance.org/glossary/bdd> (accessed on 29 September 2021).
36. Cambridge University Press. Methodology. 2026. Available online: <https://dictionary.cambridge.org/dictionary/english/methodology> (accessed on 4 May 2026).
37. Cambridge University Press. Framework. 2026. Available online: <https://dictionary.cambridge.org/dictionary/english/framework> (accessed on 4 May 2026).
38. Partelow, S. What Is a Framework? Understanding Their Purpose, Value, Development and Use. *J. Environ. Stud. Sci.* **2023**, *13*, 510–519. <https://doi.org/10.1007/s13412-023-00833-w>.
39. What Is the Difference Between Methodology and Framework? 2021. Available online: <https://www.linkedin.com/pulse/what-difference-between-methodology-framework-dafir-jacob/> (accessed on 29 September 2021).
40. Irshad, M.; Britto, R.; Petersen, K. Adapting Behavior Driven Development (BDD) for large-scale software systems. *J. Syst. Softw.* **2021**, *177*, 110944. <https://doi.org/10.1016/j.jss.2021.110944>.
41. BDD Discovery. BDD Formulation. 2018. Available online: <https://bddbooks.com/> (accessed on 18 January 2021).
42. Cucumber Discovery Workshop. Available online: <https://cucumber.io/docs/bdd/discovery-workshop/> (accessed on 27 December 2021).
43. Li, T.; Tsubota, S.; Hirono, K. Gherkin Syntax Extension for Parameterization of Network Switch Configurations in Test Specification. In *Proceedings of the 2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Toulouse, France, 23–26 October 2017; pp. 12–14.
44. Visser, E. WebDSL: A Case Study in Domain-Specific Language Engineering. In *Generative and Transformational Techniques in Software Engineering II*; Springer: Berlin/Heidelberg, Germany, 2007.
45. Parsa, S. *Software Testing Automation: Testability Evaluation, Refactoring, Test Data Generation and Fault Localization*; Springer Nature: Cham, Switzerland, 2023.
46. da Silva, A.R.; Paiva, A.C.; da Silva, V.E.R. Towards a Test Specification Language for Information Systems: Focus on Data Entity and State Machine Tests. In *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development*, Funchal, Madeira, Portugal, 22–24 January 2018; pp. 213–224.
47. Hesenius, M.; Griebe, T.; Gruhn, V. Towards a behavior-oriented specification and testing language for multimodal applications. In *Proceedings of the 2014 ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, Rome, Italy, 17–20 June 2014; pp. 117–122.
48. Google Trends. Search Interest for Cucumber BDD, Spock BDD, EasyB BDD, JBehave BDD, and FitNesse BDD (2010–2024). Available online: <https://trends.google.com/trends/explore?date=2010-05-23%202024-06-23&q=cucumber%20BDD,Spock%20BDD,EasyB%20BDD,JBehave%20BDD,FitNesse%20BDD> (accessed on 23 June 2024).
49. Binamungu, L.P.; Embury, S.M.; Konstantinou, N. Maintaining behaviour driven development specifications: Challenges and opportunities. In *Proceedings of the 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Campobasso, Italy, 20–23 March 2018; pp. 175–184.
50. sAFE. What Is a Story. 2022. Available online: <https://www.scaledagileframework.com/story/> (accessed on 29 September 2022).
51. sAFE. What Is an Enabler. 2022. <https://www.scaledagileframework.com/enabler/> (accessed on 29 September 2022).
52. Rahani, A.R.; Al-Ashraf, M. Production flow analysis through value stream mapping: A lean manufacturing process case study. *Procedia Eng.* **2012**, *41*, 1727–1734.
53. Schwaber, K.; Sutherland, J. The scrum guide. *Scrum Alliance* **2011**, *21*, 1.
54. Heng, S.; Snoeck, M.; Tsilonis, K. Building a Software Architecture out of User Stories and BDD Scenarios: Research Agenda. In *CEUR Workshop Proceedings*; CEUR-WS. org: Aachen, Germany, 2022; Volume 3134, pp. 40–46.
55. Chemnitz, L.; Reichenbach, D.; Aldebess, H.; Naveed, M.; Narasimhan, K.; Mezini, M. Towards code generation from bdd test case specifications: A vision. In *Proceedings of the 2023 IEEE/ACM 2nd International Conference on AI Engineering–Software Engineering for AI (CAIN)*, Melbourne, Australia, 15–16 May 2023; pp. 139–144.
56. Mughai, A.H. Promoting Quality in BDD Scenarios Using Checklist: An Investigation from the Perspective of Novice Professionals. *arXiv* **2024**, arXiv:2402.15928v1.
57. Santos, S.; Pimentel, T.; Rocha, F.G.; Soares, M.S. Using Behavior-Driven Development (BDD) for Non-Functional Requirements. *Software* **2024**, *3*, 271–283. <https://doi.org/10.3390/software3030014>.

58. Nagappan, N.; Maximilien, E.M.; Bhat, T.; Williams, L. Realizing Quality Improvement Through Test Driven Development: Results and Experiences of Four Industrial Teams. *Empir. Softw. Eng.* **2008**, *13*, 289–302. <https://doi.org/10.1007/s10664-008-9062-z>.
59. Rafique, Y.; Mišić, V.B. The Effects of Test-Driven Development on External Quality and Productivity: A Meta-Analysis. *IEEE Trans. Softw. Eng.* **2013**, *39*, 835–856. <https://doi.org/10.1109/TSE.2012.28>.
60. George, B.; Williams, L. A structured experiment of test-driven development. *Inf. Softw. Technol.* **2004**, *46*, 337–342.

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.