Agile Meets Formal: An Examination of Behavior-Driven Specification in Practice

Joel D. Allred[0009−0006−7286−5574], Simon Fraser[0009−0008−3224−1761], and Alessandro Pezzoni[0000−0002−2886−2943]

Anaplan Limited, York, UK
joel@allred.ch, {simon.fraser, alessandro.pezzoni}@anaplan.com
http://www.anaplan.com

Abstract. Agile methods are now widely used in software engineering organizations, whereas most formal methods are limited to niches and are perceived as inadequate in the context of agile development.
This paper presents a case study of the innovative practices used at Anaplan, a financial planning and analysis software provider, to integrate formal specification within an agile process.
The results show how Behavior-Driven Specification (BDS), by documenting behavior using executable acceptance criteria (EAC), is used to validate the design and implementation of calculation functions in Anaplan's sparse calculation engine, while keeping all stakeholders aligned on the requirements. We also show that the interaction between the specifiers and the developers allows catching implementation issues at early stages of the development, while allowing the specification to remain amenable to emerging implementation constraints. The validated requirements have enabled the development of a framework to automatically generate extensive test coverage that is used to verify the implementation. As a result, over 200 bugs were caught in the production code before release, not counting the hundreds of issues that BDS allowed developers to detect earlier in the process.
We show that BDS leads to high levels of confidence in the behavioral correctness of software while being fully aligned with agile practices, and proves to be a significant evolution in the field of software development.

Keywords: Behavior-driven specification · Agile · Software development · Requirements engineering

Note: This preprint has not undergone peer review or any post-submission improvements or corrections. The Version of Record of this contribution is published in *Requirements Engineering: Foundation for Software Quality, REFSQ 2024, Lecture Notes in Computer Science, vol 14588* and is available online at https://doi.org/10.1007/978-3-031-57327-9_21.

1 Introduction

Agile methods [3] are becoming ubiquitous in modern software engineering companies. Meanwhile, formal methods are seen as having limited use, with the perception being that they can only provide value in niche industries, such as safety-critical system or

chip development. In our experience, many software developers maintain a view that formal methods are incomprehensible, are tied to linear development practices, and are incompatible with agile methods. Changing requirements is often inevitable in the world of software, and recent efforts – based on incrementality and an iterative design loop – have been made to marry agile processes with model-checking [9].

Anaplan is a SaaS company which produces an enterprise planning platform using an agile development process. The complexity of planning models has led Anaplan to develop an alternative calculation engine suited to notionally large models that are sparsely populated with data. Development had to satisfy a consistency requirement, where the sparse engine had to be functionally equivalent to the existing one, except for intentional differences that had to be documented. When adding new functionality to the sparse engine, flawlessness also had to be ensured, because modifying the behavior of calculation functionality that has already been shipped to customers would cause significant reputational damage, as organizations rely on these numbers to make critical strategic decisions.

To satisfy these goals, we devised Behavior-Driven Specification (BDS). This approach and its technical process were introduced in [8]. We hereby elaborate on how the approach was implemented from a process and team point of view for the delivery of the sparse calculation engine, from its inception to General Availability (GA), and conduct a thorough evaluation of the effectiveness of BDS at Anaplan based on quantitative quality metrics and stakeholder feedback.

2 Background

Anaplan is renowned for its enterprise planning management platform. It has been widely adopted for business planning purposes, including budgeting, forecasting, financial planning, and analysis, as well as for operations and supply chain management. Anaplan provides an environment that customers use to model the various aspects of their business [15]. At the center of the Anaplan platform is a robust calculation engine. This engine oversees the management of OLAP (Online Analytical Processing) cubes [5], which are multidimensional arrays of data representing various business measures. Anaplan hosts a rich modeling language, offering extensive slicing, dicing, and calculation capabilities. Crucially, Anaplan's aggregation capabilities allows the construction of models that take trillions of raw data points and produce simple dashboards that provide health indicators and insights that are critical to business planning. To ensure efficient and consistent model operations, all cube data is kept in memory for rapid retrieval and calculation. When any cell's value changes within a single model, Anaplan instantly recalculates all dependent figures, thereby offering a dynamic, real-time view of business operations.

2.1 Diversifying the Engine Portfolio

The Anaplan calculation engine was optimized for the dense population of a model, making use of large chunks of contiguous memory for cube data, leading to efficient calculation. However, when running in the public cloud, this approach is physically

limited by the size of available hardware. Hence, a calculation engine optimized for the sparse population model was developed and offered alongside the existing engine.

This new engine was developed in parallel to the dense engine and continuously deployed to production environments while ensuring functional consistency with the dense engine, with necessary differences documented. Due to the rapid and organic growth of the software in earlier years, the dense engine had been developed without a precise specification of the requirements. The dense engine now had to act as the 'source of truth', and its behavior needed to be captured and used as a specification for the sparse engine. This had to be done in an agile way where the sparse development could start without having a full understanding of the dense behavior, and be revisited as the specification is created. In this asynchronous way of working, implementation could even start before specification, in which case feature flags would be used to hold off the release of a feature until behavior is validated and verified.

2.2 Validating New Functionality

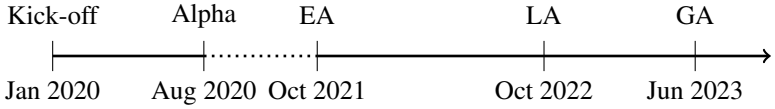
In addition to the formalization of the existing behavior, the introduction of new features such as calculation functions necessitates meticulous design and testing. Users depend on the consistency of numeric calculations embedded in their models, and thus, rolling out a new version of the Anaplan software that alters these computations, even for rectifying a prior bug, would prove significantly disruptive, principally because small changes at raw data level can lead to substantial changes in aggregated figures.

Therefore, before any new functionality can be released to users, Anaplan must:

1. validate the requirements — ensure the feature aligns with the users' needs
2. validate the implementation — confirm the feature satisfies the requirements
3. verify the implementation — ensure the implementation does not contain any fault.

2.3 Timeline

The engineering process described in this article was instrumental in the successful delivery of the sparse calculation engine, which was an ambitious undertaking that mobilized several engineering teams [1] for over three years.



The development work for the sparse engine was started in January 2020. In March 2020, a steel thread [1] was merged into the main codebase and deployed to production for selected internal users to evaluate. An alpha version with limited functionality was made available in August 2020 to selected customers and for internal use. In October 2021, a feature-incomplete version of Anaplan with the sparse engine was released in Early Access (EA) to selected customers for experimentation. A feature-complete Limited Availability (LA) version was released in October 2022 for use on customer

[1] Approximately 50 engineers were involved over the lifetime of the project.

production models. Finally, GA was announced in June 2023 and the engine is now being widely distributed to customers who have a need for high dimensionality in their models.

3 Leveraging Behavior-Driven Specification

Behavioral correctness of the Anaplan calculation engines being essential, the engineering team has embraced formal methods while also adopting agile development practices. The task of ensuring correctness can be divided into *verification* — 'doing things right' — and *validation* — 'doing the right thing'. Initially, the task was to ensure consistency between the two engines. Subsequently, there emerged a requirement to validate and verify new functionality introduced in the sparse engine. During the lifetime of the project, this latter aspect grew in importance relative to replicating the behavior of the dense engine. To realize these objectives, a formal specification was instituted to encapsulate the logic inherent in the modeling language. The formal specification is written in VDM-SL [14], a specification language that has tooling with animation capabilities.

Precisely documenting the behavior is of little use if the product requirements are unknown or misunderstood. Although a formal specification is useful for recording expected behavior in an implementation-agnostic way, it is not an adequate tool to be discussing requirements. Evans [6] describes how design and development can be problematic when language is fractured [2]. Despite the existence of a large amount of research discussing the agile engineering of requirements, the software engineering community still lacks appropriate processes to work with iteration-based specification [11], and missing or incomplete requirements is a cause of project delays [16].

In an effort to integrate the specification work with well-proven agile processes, Anaplan implemented BDS [8], a formal specification approach aligned with agile principles. The BDS process resembles Behavior-Driven Development (BDD) [17], an agile technique that stimulates collaboration, communication, and understanding between the project stakeholders, and like the Cucumber framework that supports BDD, requires a domain-specific language (DSL) for writing scenarios. However, unlike most BDD use cases, BDS clearly separates verification from validation.

The descriptive specification language introduced in [8] and materialized in the form of EACs, was brought forth to facilitate discussions and validate requirements coherently among all stakeholders. This language was formulated to ensure the convergence of understanding, enabling precise, unambiguous communication and agreement on the requirements and functionalities discussed. EACs are structured as a Hoare Triple [10] of the form:

- *given* an initial context (precondition)
- *when* the stakeholder performs an action (command)
- *then* the outcome is as expected (postcondition)

[2] Language fracture exists when stakeholders use different formalisms.

These scenarios are written using a DSL, with an adapter created by the developer to enable scenario execution. Examples of some trivial EACs are given in listings 1.1 and 1.2[3].

```
@Eac("A created list is empty")
fun create() {
  whenever {
    createAList("list")
  }
  then {
    listContains("list")
  }
}
```

Listing 1.1. EAC for creating a list

```
@Eac("When an entity is added,
  it is contained by the list")
fun addToAList() {
  given {
    thereIsAList("list", entities = "a")
  }
  whenever {
    addEntityToList("list", "b")
  }
  then {
    listContains("list", entities = "a", "b")
  }
}
```

Listing 1.2. EAC for adding to a list

EACs are abstract and can be validated against any system which provides an adapter for the abstract modeling language used. Adapters were developed for both the dense and sparse engines, as well as the formal specification. Importantly, this means that the EAC is — by default — validated against all three systems every time it is run.

Since requirements engineering is inherently a human-centric process [2], maintaining the requirement specification of the software in a readable form allows stakeholders to continuously refer to an up-to-date expected behavior. This living documentation acts as a unique source of truth and provides a starting point to discuss behavior while avoiding redundant exchanges that would arise from incomplete or outdated information, thus improving the shared understanding of requirements [4,12] The various artifacts of BDS, as well as the framework [7] that allows their integration in the development process are all maintained by a dedicated team of engineers called *System Specification*.

3.1 The Agile Process for New Functionality

On the sparse engine project, the traditional requirement engineering process of elicitation, analysis, documentation, and validation [13] of a newly introduced feature is adapted to follow an agile workflow. In the typical initiation phase of an agile project, user stories are created and prioritized, forming an initial backlog under the guidance of a PO. Once the user story is prioritized, the development process starts with conception and concludes with the feature's release to the user, as depicted in Figure 1.

The conception phase involves interactions between the PO, user group representatives, and the System Specification team to outline the product requirements for the feature. This stage also addresses how the new feature aligns with the existing constructs from a modeling standpoint, and whether it is more appropriate to introduce a new concept or expand existing functionality. The acceptance criteria at this stage are written in natural language.

Once the high-level concept is approved by all stakeholders, the feature advances to the refinement stage, during which development teams investigate the feasibility of

[3] Note that the keyword `whenever` is used in the scenario as `when` is a Kotlin keyword.

the feature and estimate the implementation effort. For each behavior, the initial step involves writing a natural language description of the AC. A scenario exemplifying the AC is then crafted, potentially extending the DSL if new language expressions are needed. This scenario then serves as an EAC used to validate the criteria against both the formal specification and the target implementation. The specifier then adjusts the formal specification as needed to satisfy the EAC, taking care not to invalidate other EACs. This process is repeated until all behavior ACs are established and agreed.

EACs keep being updated as they are handed to developers for behavior implementation. Iterative validation and refinement are necessary to keep stakeholders aligned on changing requirements [13]. During the functional review phase, stakeholders deliberate on the EACs and potential implementation limitations. During the implementation phase, the development teams will run the EACs and ensure that they pass before considering the implementation as complete.

The agility of the process makes it possible for the implementation to start before the specification is fully agreed, or even started. Importantly, the System Specification team remains separate from the development teams and has no involvement in the implementation.

A powerful complement to BDS is the verification framework. This is used to generate regression tests that combine many features[4]. Many levers are offered that enable targeted testing, but the basic process involves:

- generation of a scenario using the DSL of the EACs
- validation of the scenario using the specification; that is: is the scenario meaningful?
- querying of the implementation for scenario results
- verification of said results by ensuring they satisfy the specification.

Once verified the scenario is easily transformed into a test case that can be added to a suite of regression tests. This process has now generated hundreds of thousands of tests that provide significant coverage, but which take significantly less time to execute than mechanisms used with the dense engine.

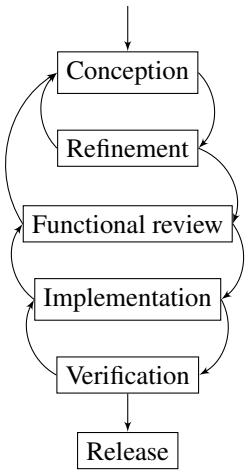


Fig. 1. Development process for new functionality

4 Evaluation

Since its introduction in early 2020, the engineering process described in this article has allowed the product and engineering teams to find an efficient and safe path to releasing the Anaplan platform whose functional correctness is critical. Leveraging BDS offers many advantages: it sharpens product requirement clarity, proactively identifies deviations from these requirements, effectively catches bugs pre-production, and reduces

[4] The optimization of said combination being a common source of bugs.

dependence on traditional verification methods like manual quality assurance (QA) and exhaustive user model testing.

In this section, we recall the benefits of BDS already exposed in [8] and elaborate using recent experience, examples, and metrics from our issue-tracking system. We also give the results of a survey carried out among the different stakeholders involved in the BDS process and gather insights that pinpoint the various aspects that differentiate BDS from other development methods.

4.1 Requirement Clarification

The fact that functional requirements are embodied in a unique source of truth – the EAC – is a fundamental shift in how features are designed and engineered. The precise documentation of the behavior makes discussing the requirements more straightforward than in areas where this process has not been applied yet.

Previously, when the PO wished to add a feature to the product, the development teams would have had to rely on the natural language requirements given by the PO, which were usually not precise enough to fully drive the implementation, leaving room for interpretation. Resolving these imprecisions would lead to long discussions because of the lack of a precise specification language. Furthermore, the resulting decision may have not always been clear to all stakeholders, introducing a risk of decisions that later need to be reverted.

Using BDS, the fact that requirements are immediately translated to acceptance tests means that our developers have been able to remove any ambiguity that would have existed. It also means that limitations coming from the implementation — for instance, choices that were made for calculation performance reasons — were surfaced and discussed appropriately and promptly and the requirements were adapted accordingly.

4.2 Early Detection of Requirement Deviation

Having a formal specification with EACs means that the implementation can be accurately validated against the requirements before a feature is released, even if the release cycle is short. The sparse calculation engine is released every two weeks, and ensuring that each version of the code satisfies the high-level requirements of the product would be very hard to achieve without this automatic validation step.

Crucially, BDS ensures that implementation and specification stay consistent with each other. As a point of comparison, during the analysis of the behavior of the dense engine, 218 difference (Diff) tickets were raised that identified a defect in the dense engine. These defects were then corrected in the sparse engine and the Diff tickets kept as documentation of differences between the two engines. Diff tickets also document the justification for the differences, as those rationales are useful for later reference when discussing adjacent topics, or when explaining behavior differences to customers.

It is worth emphasizing that we use *defect* in the broad sense. Some defects are simply errors in the code, stemming from misunderstandings of the requirements, but bugs can exist in the specification itself. Analyzing the behavior of the dense engine using BDS led us to uncover many issues in the requirements that were caused by not having an independent source of truth.

4.3 Reduced Reliance on Manual Testing

The verification framework has allowed the generation of hundreds of thousands of test cases. Extending this suite to test specific aspects of the functionality requires very little work. This innovative tool ensures a regression coverage that would be unthinkable to achieve with manually-written tests. During the development of the project, the regression suite has caught dozens of bugs that would have otherwise been deployed to production [5]. Also, having an extensive regression suite allows the development teams to keep developing at speed to implement improvements in the engine while keeping the risk of introducing regressions low. Compute performance is a critical aspect of the Anaplan platform and the engineering roadmap for the sparse engine contains much performance enhancement work. This means that the code will incur many changes in the future and the regression suite will ensure that functionality remains unaffected.

Parts of the functionality that have been fully specified can be released without any manual checks. The existence of the specification and the fact that the EACs and the generated regression suite pass remove the need for manual testing at release time. QA resources can thus be redirected to areas that are not yet covered by specification.

4.4 Catching Code Defects: A Quantitative Evaluation

During development of the sparse engine, a number of defects were discovered in the production code. We call *defect* an observable deviation from the intended behavior. In total, 307 calculation logic defects have been identified in the production code over the course of the development [6]. These defects are partitioned into three categories, depending on which agent identified them:

1. System Specification process and tooling:
2. Internal stakeholder (e.g. Customer Support)
3. Customer

In addition, we distinguish whether the defect was found in a released or an unreleased version of the software. The following table gives us how many issues were detected in each category between September 2020 and August 2023.

	Unreleased	**Released**
System Specification Process and Tooling	296	11
Internal Stakeholder (e.g. Customer Support)	0	2
Customer	-	1

Table 1. Calculation issues detected between September 2020 and August 2023

[5] The development teams have their own unit and functional tests, so the regression suite catches issues that the existing testing mechanisms have failed to detect.

[6] To underline the significance of this number, less than 500 calculation bugs have been raised against the dense engine over the past 5 years.

It is worth mentioning that the above only counts bugs that have been caught after the code was pushed to the production branch of the repository, either because the specification was written after the implementation was completed, or because the issue was caught by the regression suite. Many more bugs, perhaps the majority, are caught by EACs before the implementation reaches production because EACs run on the continuous integration system and can immediately be fixed by the developers. In sum, in the vast majority of cases, existing defects were caught before users had a chance of detecting them, and only one calculation bug was ever reported by a customer.

4.5 Qualitative Results

A central aspect of BDS is the cross-team interactions and the various stakeholder's perception of the specification process. To understand people's experience with the process, a questionnaire was sent to representative stakeholders to understand how people interact with the specification and what kind of value they see in the process. This was not a widespread survey, but rather a collection of insights from a selection of people in each functional area of the project. We interrogated the PO, the Engineering Lead in charge of the project, two Engineering Managers, two Technical Leads, and four Software Engineers, of which two are in the System Specification team. This non-anonymous survey consisted of a mix of graded and open questions. Responses were analyzed and collated to give a synthesized account of everyone's experience with the process.

Awareness and Understanding To understand the extent to which people are exposed to and understand BDS, we asked stakeholders to self-report whether they were aware of the process and whether they understood it. One respondent was unaware that BDS was being used. The other stakeholders understood the process with various degrees of familiarity. Engineers outside of the System Specification team reported that they had a practical understanding – allowing them to contribute to the decision-making and the implementation – but not a formal one, which demonstrates that all feel the benefits despite different levels of understanding.

Communication and Collaboration All respondents agreed that BDS improves communication between teams. It is considered that the document provided by BDS is far superior to ad-hoc requirement documents as it removes much of the guess-work around areas that have not been defined with sufficient clarity. EACs are considered a useful tool as they are much more readable for engineers than the formal specification and require no knowledge of any particular code. Even junior engineers can translate EACs into appropriate unit tests at the appropriate level of abstraction. Having a common language and process has helped understand and document the differences between the sparse and dense engines. Bug reports are also enhanced as they are supported by clear and unambiguous test cases that also use the EAC formalism. The clarity allowed many issues to be agreed as 'working as intended' very early in the bug triage process, concluding that there were missing or incorrect requirements.

Crucially, the PO noted that BDS provides much greater definition of detailed requirements, as subject matter experts can own the precise requirement specification. In

addition, there were many instances where the PO had to think through the implications of their initial proposals and revise them as part of the process. This forced the PO to make explicit decisions about the product where, without BDS, the developer would have had to hazard a best guess.

Exposure to EACs The members of the System Specification team are the only stakeholders engaged in writing EACs. The technical leads and one engineer are able to understand the meaning of most EACs. One manager has not had any exposure to EACs. All other managers, engineers, and PO have had some exposure to EACs but were unfamiliar with the syntax.

Engineers working on the kernel have little familiarity with EACs because they are written using the Anaplan concepts whereas the kernel is a much more generic OLAP engine. Also, stakeholders mainly involved in non-calculation aspects of Anaplan are unfamiliar with EACs which, for now, only cover calculation functionality. However, engineers working on the translation layer above the kernel regularly look at failing EACs to debug the implementation.

Quality and Outcomes All respondents considered that BDS significantly increased the quality of the software produced. Since Polaris was made available, a single bug was identified by a customer, while hundreds of issues — and possibly thousands if we count the occurrences of developers running EACs before pushing their code to production — were caught internally by the process and the tooling. Another outcome is that the process led to the identification and documentation of over 300 behavioral differences between the dense and the sparse engine. This documentation enables the creation of precise user-facing documentation that explains the intended differences to a user that encounters a behavior in the sparse engine that is unexpectedly different from the dense behavior they are familiar with. This improves the feeling that a given feature is working as intended, rather than being mistaken for a bug. Finally, the extensive regression suite that is generated from the specification gave the team the confidence to transition safely to fortnightly releases.

Development Speed When asked what effect BDS had on the overall development speed, 80% of respondents judged that BDS made development faster, whereas 20% considered it made it slower. Most people consider that BDS increases the wall time to release features because the specification work, although executed by a team separate from the development team, can become a bottleneck due to resourcing. Also, when writing EACs, there is occasionally the need to introduce some DSL to support the constructs used in the scenario.

However, respondents acknowledge that the reduced back-and-forth achieved by getting the implementation *right the first time* contributes to lower overall effort because fewer bugs are shipped. In addition, the cost of bug investigation, model breaks, management of dissatisfied customers and Anaplan stakeholders, fixes, and release is reduced, freeing resources that enable the development of more features.

Challenges Practicing BDS effectively requires effort and it takes time to get a feel for what constitutes a good EAC. There is also a potential interlocking between developer repositories and specification test repositories, since if either the implementation or the EACs are not ready, the test suite will not pass, which can block development[7]. This was resolved by introducing a form of deferred compliance where EACs are temporarily skipped until compatibility is achieved.

Engineers also found that the lack of a naming convention between EACs and implementation could make EACs difficult to read for engineers using different names internally. For various reasons, Anaplan constructs can have different names at different levels of abstraction. One engineer found the EAC syntax confusing because the DSL varies from the testing constructs they use in their team.

The issue of workload synchronization was also raised. When planning to implement a feature, managers need to ensure that enough workforce capacity is available in the System Specification team as well as the development team, which can complicate planning.

General Feedback All stakeholders believe that BDS should be extended beyond calculation to other areas of the platform. Among the most cited suggestions for expansion are exports, imports, filters, and generally all the grid functionality that the user can interact with. Overall, all respondents are very satisfied with the process and consider that the BDS process is essential to the quality of the Anaplan offering and is a cost-effective way of catching issues early in the process. It has been instrumental in the ability to release the calculation engine at a fast cadence with a very high confidence in its correctness, which is essential to the mission-critical operations of Anaplan's customers.

5 Conclusion

As a methodology to efficiently design, implement, and release software whose correctness is critical, BDS has proved to be remarkably effective in the development of Anaplan's sparse calculation engine. Our analysis shows that it is not only possible to integrate formal methods in an agile context, but implementing BDS brought levels of confidence in the correctness of the software that would not have been achievable by other means. From a team perspective, BDS has been successfully integrated without alienating staunch agile advocates, some of whom are now promoters of the process.

Through the establishment of EACs, the functionality validation process ensures that implementation remains constantly in line with the expected behavior as understood by all stakeholders. More importantly, the requirement specification is agile and can never go stale because it evolves with the code. Coordination is achieved by discussing requirements during their inception, as well as during subsequent modifications. Adherence to the requirement specification is enforced by the EACs acting as acceptance tests. The VDM-SL formal specification allows the automatic generation of an

[7] This only applies to changes to existing behavior such as bug fixes. New behavior can be implemented and specified in parallel.

extensive regression suite that detects faults that are not captured by the acceptance criteria, for instance when features are combined.

As our evaluation shows, BDS can be considered to have decreased overall feature development effort by catching nearly 300 issues before they reached production, thus avoiding expensive rework and impact on customer relations. To date, a single calculation bug was ever detected by an external user. The process was instrumental in bringing an entirely new engine to general availability through frequent and drama-free releases.

References

1. Alkobaisi, S., Bae, W.D., Narayanappa, S., Debnath, N.: Steel threads: Software engineering constructs for defining, designing and developing software system architecture. Journal of Computational Methods in Sciences and Engineering **12**(s1), S63–S77 (2012)
2. Alwidian, S., Jaskolka, J.: Understanding the role of human-related factors in security requirements elicitation. In: International Working Conference on Requirements Engineering: Foundation for Software Quality. pp. 65–74. Springer (2023)
3. Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R.C., Mellor, S., Schwaber, K., Sutherland, J., Thomas, D.: Manifesto for agile software development (2001), `http://www.agilemanifesto.org/`
4. Buchan, J.: An empirical cognitive model of the development of shared understanding of requirements. In: Requirements Engineering: First Asia Pacific Requirements Engineering Symposium, APRES 2014, Auckland, New Zealand, April 28-29, 2014. Proceedings. pp. 165–179. Springer (2014)
5. Codd, E.F., Codd, S.B., Salley, C.T.: Providing OLAP (on-line analytical processing) to user-analysts. An IT Mandate. White Paper. Arbor Software Corporation **4** (1993)
6. Evans, E.: Domain-driven design: tackling complexity in the heart of software. Addison-Wesley Professional (2004)
7. Fraser, S., Pezzoni, A.: Azuki framework to assist with behavior-driven specification. `https://github.com/anaplan-engineering/azuki`, accessed: 2023-10-01
8. Fraser, S., Pezzoni, A.: Behaviour driven specification. Proceedings of the 19th International Overture Workshop pp. 5–20 (2021)
9. Ghezzi, C.: Formal methods and agile development: Towards a happy marriage. The Essence of Software Engineering pp. 25–36 (2018)
10. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM **12**(10), 576–580 (1969)
11. Inayat, I., Salim, S.S., Marczak, S., Daneva, M., Shamshirband, S.: A systematic literature review on agile requirements engineering practices and challenges. Computers in human behavior **51**, 915–929 (2015)
12. Jebreen, I., Awad, M., Al-Qerem, A.: A propose model for shared understanding of software requirements (SUSRs). In: Information Science and Applications (ICISA) 2016. pp. 1045–1056. Springer (2016)
13. Kotonya, G., Sommerville, I.: Requirements engineering: processes and techniques. Wiley Publishing (1998)
14. Larsen, P.G., Battle, N., Ferreira, M., Fitzgerald, J., Lausdahl, K., Verhoef, M.: The Overture initiative integrating tools for VDM. ACM SIGSOFT Software Engineering Notes **35**(1), 1–6 (2010)
15. Sidorova, M.I., et al.: Strategic planning software as a tool for improvement of business information space. European Proceedings of Social and Behavioural Sciences (2022)

16. Wang, X., Zhao, L., Wang, Y., Sun, J.: The role of requirements engineering practices in agile development: an empirical study. In: Requirements Engineering: First Asia Pacific Requirements Engineering Symposium, APRES 2014, Auckland, New Zealand, April 28-29, 2014. Proceedings. pp. 195–209. Springer (2014)
17. Wynne, M., Hellesoy, A., Tooke, S.: The Cucumber book: Behaviour-driven development for testers and developers. Pragmatic Bookshelf (2017)