

Connecting Simulation and Test-Driven Development

Comparing SDSE and BSN with TDD and ATDD

Connect behavioral promises, simulation, acceptance expectations, and implementation evidence across the software lifecycle.

By Marc Hornbeek, a.k.a. DevOps_the_Gray

Originator of Simulation-Driven Software Engineering

October 2026

www.PrimaModus.com

Executive Summary

Simulation-Driven Software Engineering (SDSE), Behavior Simulation Notation (BSN), Test-Driven Development (TDD), and Acceptance Test-Driven Development (ATDD) contribute to a common engineering objective: software that fulfills its intended behavior. Their contributions differ in scope, working artifacts, and the evidence they produce. SDSE is a lifecycle discipline that uses simulation to challenge candidate behavior before design commitments and continues to connect that behavior to implementation and operation. BSN supplies a representation for the behavioral model; TDD and ATDD guide development through tests and shared acceptance expectations.

A useful comparison recognizes the strength of each approach. TDD combines test feedback with incremental implementation and refactoring. ATDD brings customer, development, and testing perspectives together to describe acceptable behavior before implementation. SDSE using BSN adds an explicit model of participants, conditions, rules, and interactions that can be exercised before the intended implementation exists. Each approach can discover specification problems, influence design, and reveal missing scenarios.

This paper follows an illustrative AI incident-response agent whose authority depends on valid human approval. The example shows how a team can clarify the promise, simulate consequential sequences, define acceptance tests, implement the rules through TDD, and compare integration and production evidence with the approved baseline. Confidence depends on the connections between these activities and the quality of their assumptions. Neither a passing simulation nor a passing test suite establishes every property of the real system.

Contents

1 A passing test suite can leave an important promise incomplete.....	4
2 Compare a lifecycle discipline, a notation, and development practices.....	4
3 Establish the approval promise and its operating assumptions.....	5
4 Use BSN simulation to challenge candidate behavior.....	6
5 Use ATDD to make acceptance expectations shared and usable.....	7
6 Use TDD to implement the rules and improve design.....	8
7 Connect expectations without duplicating every test.....	8
8 Understand the limits of simulation and test evidence.....	9
9 Combine the approaches in an iterative delivery workflow.....	9
10 Apply the same discipline to AI and multi-agent work.....	10
11 Adopt the combination where it improves decisions.....	11
Conclusion.....	11
References.....	12
Learn more and participate.....	12
About the Author.....	12

List of Figures

Figure 1 One behavioral promise across simulation and testing.....	7
Figure 2 Feedback from implementation and operation to the model.....	10

1 A passing test suite can leave an important promise incomplete

An incident-response agent proposes a change to restore a production service. A human approves the proposal, and the agent queues the action. Before execution, another engineer changes the target configuration, making the original remediation inappropriate. The queued action nevertheless proceeds because the implementation checks that approval exists without checking whether the conditions behind that approval still hold.

In this illustrative case, developer tests correctly verify the implemented approval lookup. Acceptance tests cover an approved action and a denied action. Both suites pass, yet the organization has left a consequential expectation unspecified. The missing question concerns whether an approval authorizes a command indefinitely or authorizes a particular command under particular conditions.

This example identifies a gap in the team's scenario selection and shared understanding. TDD or ATDD practitioners could discover and address the gap through careful discussion and test design. Behavioral modeling gives the team another way to expose it by making state changes and event order explicit. The engineering task is to build a connected account of intent, expectations, implementation behavior, and operational evidence.

Use this incident to begin a cross-functional discussion before choosing a technique. Ask the operator what authority the agent should retain after the target changes, and ask security how that authority must be enforced. Involve the developer who understands queued execution and the product owner responsible for the consequences. Otherwise, each discipline may solve a different version of the approval problem.

2 Compare a lifecycle discipline, a notation, and development practices

TDD is a development practice in which a test precedes the behavior being implemented. The engineer observes the expected failure, adds enough implementation to pass, and refactors while retaining the tests. Fowler emphasizes both the recurring red-green-refactor cycle and the design value of considering the interface before implementation [1]. The immediate feedback supports many small design decisions rather than requiring a complete design up front.

ATDD begins with collaboration about acceptance expectations before implementing the corresponding functionality. Customer, development, and testing perspectives contribute examples and acceptance tests that express how the system should function. Agile Alliance describes automation as something teams may apply, while stressing the importance of the conversation [2]. A particular framework or scenario syntax is therefore an implementation choice rather than the defining feature of ATDD.

SDSE organizes engineering around a behavioral promise across the lifecycle. Its initial sequence proceeds from Intent to Candidate Specification, Behavior Simulation, Validated Specification, Design, Code, and Integration. Simulation challenges the candidate before architecture and detailed design commitments, then continues as design, implementation, and operational evidence become available. The validated behavioral baseline is authoritative for the current scope and version, with material changes returning for review and revalidation.

BSN is the notation used to express the model rather than a separate lifecycle competing with TDD. Its BOX, INTERFACE, OPERATING, LOGIC, NETWORK, and SCENARIO constructs support descriptions of the participants, exchanges, context, rules, relationships, and situations under examination. For the agent, that

representation can expose how approval, target state, execution, and evidence recording interact. The implementation still requires appropriate interfaces, state management, permissions, tests, and operational controls.

Compare typical emphases without making them absolute boundaries. Tests can describe system interactions, and models can represent individual component behavior. Acceptance conversations can revise intent, while developer tests can reveal contradictions in an approved specification. The distinction is how the team represents and examines the behavior for a particular decision.

Aspect	SDSE using BSN	TDD	ATDD
Primary emphasis	Behavioral exploration and lifecycle continuity	Test-first implementation and refactoring	Shared acceptance understanding
Working artifacts	Behavioral model, scenarios and approved baseline	Developer tests and evolving implementation	Acceptance examples and tests
Early evidence	Modeled outcomes under stated assumptions	Failure and passing behavior of implementation tests	Agreement on expected outcomes; later test results
Design contribution	Exposes interaction and state uncertainties	Shapes interfaces and incremental code design	Clarifies externally meaningful behavior
Principal limit	Model validity and represented scenario scope	Test scope and dependency realism	Example coverage and observable test outcomes

Table 1. Typical contributions and limits of the approaches.

3 Establish the approval promise and its operating assumptions

The agent's promise should explain what may happen, what must be prevented, and what evidence must remain. For this example, an approved action may execute only when the approval belongs to that exact action and target, remains within its lifetime, and has not been revoked. Execution must also confirm that the relevant target-state precondition remains true. If any required condition cannot be established, the system rejects or defers execution and informs the operator.

Assume an illustrative approval lifetime of five minutes. Bind approval to an action identifier, target, action parameters, and relevant configuration version. Define expiry at the boundary explicitly: at or after the expiry time, the approval is invalid. These choices are scenario assumptions for this paper; each organization must determine its own policy and consequences.

Now examine when the checks occur. A valid approval at queue entry may expire before the executor starts, and a valid target version at authorization may change before the side effect. Specify which component owns the authoritative decision and how it coordinates the state check with execution. Writing a rule without considering that interval can leave the original failure intact.

Agree on observable outcomes as well. A rejected attempt should record its incident and action identifiers, rejection reason, and the policy basis needed for investigation. An uncertain execution outcome should

trigger reconciliation before another attempt. Without those expectations, the system can prevent an unsafe action while leaving its operator unable to understand what happened or what to do next.

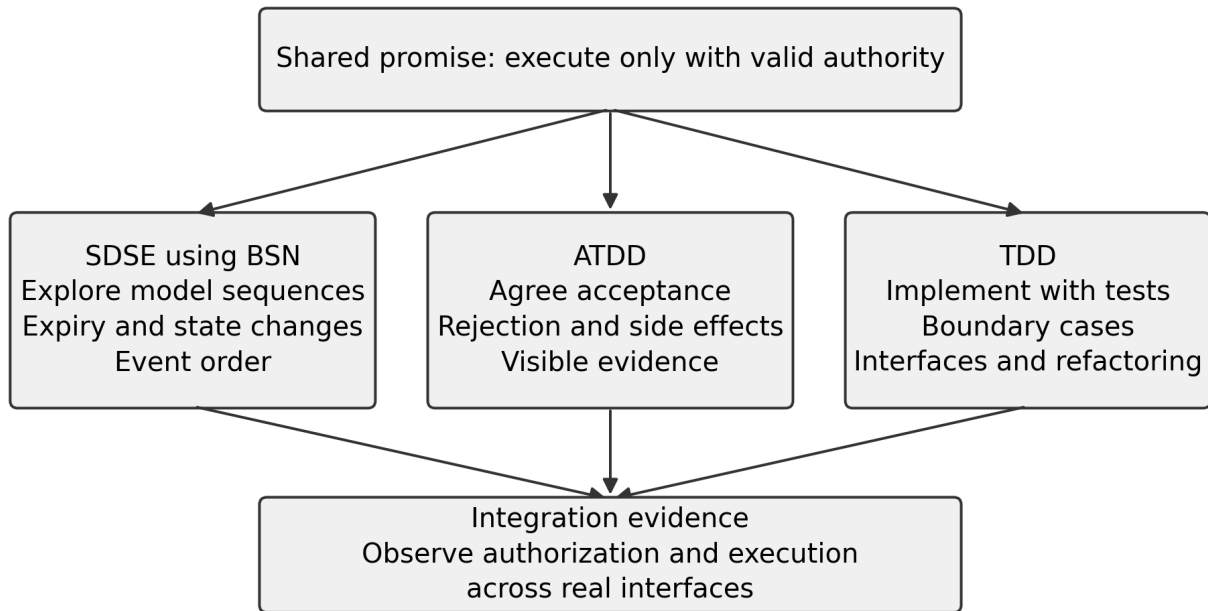
4 Use BSN simulation to challenge candidate behavior

Begin with a small behavioral model containing the agent, approver, authorization function, executor, target, and evidence recorder. Describe the relevant exchanges and states without committing each participant to its own service or process. Represent the rule that approval must still be valid when execution is authorized, along with expiry, cancellation, and target-change events. Model detail should follow the uncertainty behind the decision rather than the size of the eventual implementation.

Exercise normal execution, late approval, queue delay, changed parameters, target modification, and revocation. Include a sequence in which authorization succeeds and the target changes immediately afterward. That sequence can reveal a requirement for coordination at the execution boundary, beyond an earlier policy check. It can also expose a specification that demands guarantees the candidate architecture cannot realistically provide.

Review traces and failed expectations with the stakeholders who defined the promise. A simulation result shows what the model did under the represented conditions. If time is modeled as discrete events, document what that abstraction can establish and which real scheduling behavior still needs investigation. An uncalibrated simulation should not be used to claim production latency or throughput.

Approve a baseline only when the evidence is suitable for the intended commitment. Preserve model and scenario versions, assumptions, exclusions, residual risks, and responsible approval. The approval establishes the current behavioral reference, while subsequent discoveries remain legitimate reasons to revise it. Ignoring model suitability can replace an implicit implementation assumption with an equally weak modeling assumption.



Simulation exercises the model; tests examine implementation behavior.

Figure 1. One behavioral promise across simulation and testing.

5 Use ATDD to make acceptance expectations shared and usable

For the incident-response agent, ATDD discussions should include the people who approve changes and operate the affected service. A business-facing expectation may be that an operator's approval cannot authorize a materially different action. A security-facing expectation may be that the execution route enforces this restriction even if the agent submits altered parameters. Agree on concrete examples that make those expectations understandable and testable.

One example reads: Given approval for an action has expired, when the executor receives that action, then it rejects the request, records the reason, and leaves the target unchanged by that request. Another describes approval for configuration version 17 followed by a change to version 18 before execution. Its expected outcome is rejection and a request for renewed review. These are illustrative acceptance examples, not executable BSN syntax.

The collaboration can occur before, during, and after modeling. Acceptance examples can seed BSN scenarios, and a simulated counterexample can prompt stakeholders to revise an acceptance expectation. Agreement on examples provides requirements evidence before execution, while a passing acceptance test later provides evidence about the implementation in its test environment. Preserve that distinction in release discussions.

Choose automation interfaces that expose meaningful outcomes. For this workflow, an API or test harness should reveal whether execution occurred and which authorization basis applied. Avoid tying every acceptance test to screen positions or internal implementation steps when stable behavioral interfaces are

available. If the automation obscures the stakeholders' expectations, the team loses the collaborative value that ATDD was intended to provide.

6 Use TDD to implement the rules and improve design

Once the team has a sufficiently clear expectation, developers can implement it through small test-first steps. Write a test for an expired approval and observe failure for the intended reason. Implement the expiry check, then add tests for the exact boundary, revocation, mismatched parameters, and an unavailable authorization dependency. Refactor the implementation while keeping these expectations protected by the suite.

The design questions become concrete. A controllable clock can make time-bound behavior deterministic in tests, while an explicit authorization interface can separate policy evaluation from execution. Test feedback may suggest a simpler representation of approval state or a clearer outcome type. The engineer should retain the refactoring step, because passing tests alone do not keep the design understandable [1].

TDD can also reveal a gap in the promise. A developer may discover that configuration versions change for reasons unrelated to the action's safety, making blanket rejection unnecessarily disruptive. That finding requires domain review of the relevant precondition, followed by revised scenarios and acceptance expectations if the change is material. Hiding the exception inside code disconnects the implementation from its accepted behavioral basis.

Use test doubles carefully around external services. A fake authorization service can make a test fast and repeatable, but it may omit real revocation delays or different error responses. Complement developer tests with contract and integration tests where those differences matter. Otherwise, the suite may accurately protect the simulated dependency's behavior while missing the behavior of the deployed dependency.

7 Connect expectations without duplicating every test

Use a traceability structure that connects a promise to the evidence required at different levels. Give the agent's approval promise a stable identifier, such as P-APPROVAL-01, and link it to model scenarios, acceptance cases, developer tests, and operational signals. The identifier connects meaning without requiring every artifact to contain the same wording or execution steps. Ownership should make changes visible across those links.

The expired-approval expectation illustrates the division of work. A BSN scenario explores expiry during queued execution across participants. An acceptance test verifies the externally visible rejection and lack of an unauthorized side effect. Developer tests cover clock boundaries and rule evaluation, while integration tests examine the actual executor and authorization interfaces. These checks support the same promise through different observations.

Generated artifacts can reduce clerical effort, but they need validation. A generated acceptance test must use an interface capable of observing the claimed outcome, and a generated developer test must avoid reproducing a bug in its expected result. Compare selected traces and implementation results rather than assuming that conversion preserves semantics. Any claimed automatic conversion should be assessed against the actual tool capability and artifact version.

Keep evidence proportional to consequence. A small local calculation may need TDD and a few shared examples, with little value from a separate simulation model. An approval workflow spanning agents, humans, and changing production state has more interaction uncertainty to justify modeling. Duplicating every test at every level increases maintenance without necessarily increasing confidence.

8 Understand the limits of simulation and test evidence

A model may accurately execute an incorrect promise. Acceptance tests may faithfully automate examples that omit a critical exception. Developer tests may protect a rule whose intended scope was misunderstood. Agreement across these artifacts is useful, but shared assumptions can produce agreement around the same mistake.

Independent challenge is therefore an engineering practice. Ask an operator or security reviewer to propose an unacceptable sequence before showing the team's expected outcome. Compare the model's operating assumptions with prototypes, dependency contracts, and production observations. Investigate counterexamples rather than treating them as inconveniences to remove from the suite.

Validation and verification both appear across this workflow. Stakeholder review asks whether the proposed behavior addresses the real need, while conformance checks ask whether an implementation meets specified expectations. Simulation and acceptance discussions can contribute to validation; tests and model comparisons can contribute to verification. Neither term should be assigned exclusively to one named practice.

Record unresolved uncertainty with the decision it affects. If the team has not established how target changes are coordinated with execution, mark that architectural question as unresolved. A green status for unrelated cases does not settle it. Without this discipline, test counts and coverage percentages can displace the evidence needed for a consequential release decision.

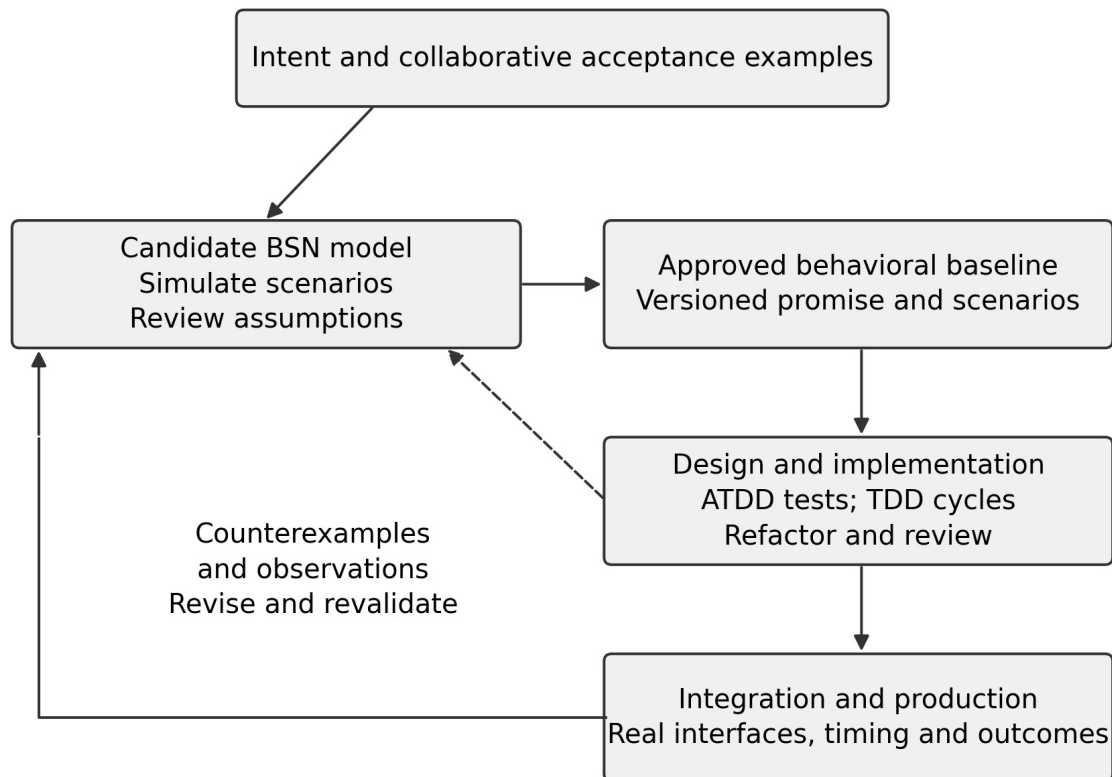
9 Combine the approaches in an iterative delivery workflow

Begin with intent and collaborative examples, then create a candidate model for the interaction risks that merit simulation. Use findings to refine expectations and approve a behavioral baseline for the next commitment. Acceptance tests can be drafted early, and TDD can guide implementation of the accepted behavior. Design and implementation discoveries return to the model whenever they materially change the promise or its assumptions.

Integration should compare meaningful observed outcomes with modeled expectations. For the agent, check ordering, rejected actions, state changes, and reconciliation across real interfaces. Include scheduling delay, dependency failures, and restart behavior that simpler models or developer tests may abstract away. Compare traces using explicitly defined equivalence criteria rather than demanding identical internal events from model and implementation.

Release assessment then examines the evidence and residual risk relevant to deployment. Verify that permissions and available tool routes preserve the intended authorization boundary. Production monitoring should detect invalid execution attempts, unexpected policy outcomes, and actions whose results remain uncertain. Those signals help determine whether the implementation, environment, or baseline needs correction.

SDSE makes this continuity an explicit lifecycle responsibility, and BSN provides a model representation for that responsibility. A named Behavior Model Owner coordinates version integrity and material revalidation with the other engineering roles. Increasing fidelity from L0 toward L3 should follow decision needs and available evidence. The progression does not eliminate the need to state what each model can and cannot establish.



Material changes return for review before the baseline is replaced.

Figure 2. Feedback from implementation and operation to the model.

10 Apply the same discipline to AI and multi-agent work

AI can help draft candidate scenarios, summarize traces, propose tests, and implement bounded changes. Review those contributions against the authoritative expectations and source evidence. If an assistant produces both implementation and tests from the same incomplete prompt, the outputs may reproduce the same omission. A passing result then provides limited challenge to the original interpretation.

For the running example, ask another reviewer to examine delayed approval, altered targets, and alternate execution routes. Check that tests can observe unauthorized side effects rather than merely a returned status. Evaluate AI-generated assertions and test data before accepting their results. Without this review, automation can make an incomplete interpretation easier to reproduce at scale.

Extend the behavioral model when more agents participate. One agent may propose recovery while another changes access policy or reserves the same target for investigation. Define conflict handling, delegation limits, cancellation, and authority over shared tools. Simulate consequential interactions, then validate the corresponding contracts and controls against the implementation.

Humans remain accountable for the promise and significant risk decisions. Approval interfaces should present the action, relevant state, evidence, and expiry in terms an operator can assess. People must retain practical ways to interrupt execution and investigate uncertain outcomes. Finite simulations and test suites should support these controls without being presented as universal guarantees of agent behavior.

11 Adopt the combination where it improves decisions

Select one workflow with consequential interaction uncertainty and an upcoming decision that can still change. Retain useful TDD and ATDD practices, then add a small BSN model where it can expose missing conditions or sequences. Establish the stakeholders, model owner, expected outcome, and evidence needed to evaluate the pilot. This creates a manageable learning exercise rather than a modeling program detached from delivery.

For the incident-response agent, begin with approval expiry and target modification. Capture a few acceptance examples, model the relevant participants, and challenge the interval between approval and execution. Implement the revised rule through TDD and test the real boundary through integration. Connect the promise to telemetry before the pilot is considered complete.

Measure specification gaps found before coding, consequential design changes, interaction defects found during integration, and maintenance effort. Track how quickly production evidence produces updated scenarios and a reviewed baseline where necessary. Compare these observations with an explicit prior baseline, without assuming that every improvement was caused by modeling. Scenario counts alone do not demonstrate useful engineering outcomes.

Account for costs as well as benefits. Teams need time to learn notation, preserve traceability, maintain the model, and review results. Choose abstractions that remain understandable to the participants who own the consequences. Expand only when the combined practices improve decisions or reduce material rework enough to justify their effort.

Conclusion

The agent in the opening example now rejects expired approval and checks the conditions attached to the requested action. Acceptance examples communicate those expectations, developer tests protect the rule implementation, and integration evidence examines the real execution boundary. Operators can inspect rejection reasons and reconcile uncertain outcomes. Production discoveries return to the same controlled behavioral reference.

TDD supports disciplined implementation and evolving design. ATDD supports shared acceptance understanding, and SDSE using BSN supports explicit behavioral exploration and lifecycle continuity. Their combined value depends on appropriate scope, sound assumptions, and evidence that can be traced to the promised outcome. Engineers gain confidence by examining both what the system should do and what it actually does under the conditions that matter.

References

- [1] Martin Fowler. Test Driven Development. Updated December 11, 2023.
<https://www.martinfowler.com/bliki/TestDrivenDevelopment.html>
- [2] Agile Alliance. Acceptance Test Driven Development ATDD. Agile Glossary.
<https://agilealliance.org/glossary/atdd/>
- [3] Marc Hornbeek. Where Simulation Fits in the Software Lifecycle. PrimaModus white paper, August 2026.
- [4] Marc Hornbeek, Debashis Bhattacharyya, and Jitin Dhanani. Simulation-Driven Software Engineering. 2026.
- [5] Marc Hornbeek. How SDSE and BSN Shape Software Architecture. PrimaModus white paper, October 2026.

Learn more and participate

Visit www.PrimaModus.com to learn more about SDSE, Behavior Simulation Notation, and the BSN Workbench. Architects, senior engineers, security practitioners, SREs, platform engineers, and agentic AI teams interested in evaluating these practices are invited to contact mhornbeek@engineeringdevops.com. Trial participants can help assess realistic use cases and influence the notation and workbench through structured feedback.

About the Author

Marc Hornbeek, known as DevOps_the_Gray, is an award-winning engineer, author, educator, executive consultant, and keynote speaker with five decades of experience in software and systems engineering. He is the originator of Simulation-Driven Software Engineering and leads the development of Behavior Simulation Notation and the BSN Workbench. He founded PrimaModus to help organizations model, simulate, validate, and engineer software that behaves as intended.

Prepared with assistance from OpenAI ChatGPT in drafting, source research, and diagram development.