

TDD vs. SDSE

From Code Correctness to Behavioral Confidence

The central proposition

Test-Driven Development helps engineers build the implementation correctly. Simulation-Driven Software Engineering helps the organization validate that it has selected the right behavior to build.

By Marc Hornbeek, a.k.a. DevOps_the_Gray

Originator of Simulation-Driven Software Engineering

August 2026

Executive Summary

Test-Driven Development (TDD) is one of the most influential practices in modern software engineering. It integrates coding, testing, and design through a repeating red–green–refactor cycle: write a failing automated test, create the minimum implementation needed to pass, and improve the design while preserving behavior. Used skillfully, TDD provides rapid feedback, supports modular design, protects refactoring, and builds a durable regression suite. It remains especially valuable as AI accelerates code generation because generated code still requires precise, executable expectations.

TDD nevertheless begins with an expectation. Someone must decide what result the test should demand. If that expectation is incomplete, locally correct but globally incompatible, or based on a misunderstood behavioral promise, TDD can implement and protect the wrong behavior with impressive discipline. A green test demonstrates agreement between an implementation and a selected test. It does not independently prove that stakeholders selected the right behavior, that interacting systems will produce the intended end-to-end outcome, or that failure and recovery behavior is acceptable.

Simulation-Driven Software Engineering (SDSE) addresses that earlier problem. SDSE begins with stakeholder intent and unacceptable outcomes, creates a Candidate Specification, represents consequential behavior in an executable model, simulates normal and adverse scenarios, reviews traces and counterexamples, and establishes a versioned Validated Behavioral Baseline before production implementation hardens. Behavior Simulation Notation (BSN) and the BSN Workbench provide the notation and working environment for creating, executing, reviewing, and maintaining that behavioral evidence.

Core conclusion

TDD helps engineers build the implementation right. SDSE helps the organization determine the right behavior to build. The strongest engineering system uses both.

1. TDD creates fast implementation feedback and supports clean, maintainable code.
2. SDSE moves executable system-level behavioral evidence ahead of costly commitments.
3. SDSE can supply validated scenarios and expectations from which TDD tests are derived.
4. TDD can confirm that implementation increments conform to the accepted behavioral baseline.
5. Neither practice eliminates the need for integration, security, performance, resilience, deployment, and operational validation.

Contents

Executive Summary

- 1. The Continuing Importance of TDD**
- 2. The Problem TDD Was Not Designed to Solve**
- 3. What Simulation-Driven Software Engineering Adds**
- 4. The Fundamental Comparison**
- 5. Unit Behavior Is Not System Behavior**
- 6. The Difference Between a Test and a Simulation**
- 7. Comparing the Development Loops**
- 8. How SDSE Strengthens TDD**
- 9. How TDD Strengthens SDSE**
- 10. TDD and SDSE in Agile Delivery**
- 11. TDD and SDSE for AI-Native Systems**
- 12. Common Misconceptions**
- 13. When TDD May Be Sufficient**
- 14. When SDSE Should Be Added**
- 15. An Integrated SDSE–TDD Operating Model**
- 16. Practical Adoption Guidance**
- 17. Measuring the Combined Value**
- 18. Leadership Implications**

Conclusion

Appendices, References, and Author

1. The Continuing Importance of TDD

TDD is not simply the act of writing unit tests before production code. It is a programming discipline in which tests, code, and design evolve together. Agile Alliance describes TDD as a style of programming that tightly interweaves coding, testing, and refactoring [2]. Kent Beck's canonical treatment presents a small-step rhythm that uses a failing automated test to create focus, a passing implementation to create progress, and refactoring to remove duplication and improve design [1].

The cycle is powerful because it shortens the distance between a programming decision and evidence about that decision. A developer does not wait for a large integration phase to learn that a small behavior is absent or broken. The test suite also creates a safety net that supports continued change. The result is not merely more tests; it is a way to design through feedback.

The red–green–refactor cycle

1. **Red.** Write a small test that expresses the next desired behavior and verify that it fails for the expected reason.
2. **Green.** Write the minimum production code required to satisfy that test, then run the relevant suite.
3. **Refactor.** Improve names, structure, duplication, interfaces, and design while keeping the suite green.

What disciplined TDD provides

4. Immediate feedback about a narrowly defined implementation expectation.
5. Pressure toward small interfaces, separation of concerns, and testable design.
6. Executable regression evidence that travels with the code.
7. Safer refactoring and more confident incremental change.
8. A repeatable mechanism for turning one expectation at a time into working software.

Empirical research does not justify simplistic claims that TDD always improves quality or productivity. Results vary with task, experience, adherence, test granularity, and organizational context. Research has found that the granularity and uniformity of the process can matter as much as nominal adoption [4], while families and meta-analyses of experiments report heterogeneous effects [5][6]. The defensible conclusion is that well-practiced TDD is valuable within its proper scope, not that it guarantees every desired system outcome.

2. The Problem TDD Was Not Designed to Solve

TDD begins when a developer or team can express the next expected result as an executable test. That is an implementation advantage and an epistemic boundary. The test cannot determine whether its own expectation is complete, ethically acceptable, economically sensible, operationally recoverable, or consistent with expectations owned elsewhere. It checks the proposition supplied to it.

A team can therefore practice TDD rigorously and still deliver the wrong system. Every component may satisfy its local tests while an end-to-end sequence violates a policy, duplicates a transaction, exposes stale state, prevents recovery, or delegates too much authority to an AI agent. These are not arguments against TDD. They reveal a different engineering problem: the need to establish and challenge the behavioral promise before using it to drive implementation.

Green is evidence—not proof of intent

A passing test proves that the tested implementation produced the expected result under the tested conditions. It does not prove that the expectation was the right one or that the scenario set represents the complete consequential behavior.

TDD can faithfully preserve

- 9. An incomplete or misunderstood requirement.
- 10. A locally valid rule that conflicts with another team's rule.
- 11. A happy path that omits degraded, recovery, misuse, or intervention behavior.
- 12. A technically correct implementation of an unsafe authority model.
- 13. A component interface whose combined timing and state behavior fails at system scale.

3. What Simulation-Driven Software Engineering Adds

SDSE is a behavior-first engineering discipline. It moves executable evidence earlier by creating a model of the proposed system behavior before production implementation exists. The purpose is not to model every detail. It is to represent enough consequential behavior to expose ambiguity, incompatible assumptions, unsafe paths, difficult state transitions, and decision-relevant uncertainty while change is still affordable.

The model represents participants, interfaces, state, operating conditions, logic, interactions, scenarios, and expectations. Simulation executes these relationships so reviewers can examine traces, outcomes, omissions, contradictions, and limits. When accountable stakeholders accept the behavior, evidence, assumptions, and residual uncertainty, they establish a versioned Validated Behavioral Baseline. That

baseline becomes the behavioral promise that design, code, tests, controls, deployment, telemetry, and operations should implement and preserve.

The role of BSN and the BSN Workbench

Behavior Simulation Notation (BSN) supplies constructs for describing system participants and behavior. The BSN Workbench provides the working environment for authoring the model, running scenarios, examining traces, recording expectations, comparing versions, and preserving review evidence. The notation makes the promise explicit; the Workbench makes it executable and reviewable.

14. BOX and INTERFACE constructs identify participants and behavioral boundaries.

15. OPERATING and LOGIC constructs express state, conditions, rules, and decisions.

16. NETWORK constructs express interactions and dependencies among participants.

17. SCENARIO constructs exercise normal, boundary, failure, degraded, misuse, recovery, and intervention behavior.

4. The Fundamental Comparison

Dimension	TDD	SDSE
Primary question	Does the implementation satisfy the selected expectation?	Is the proposed end-to-end behavior the right behavior to build and accept?
Starting point	A testable implementation expectation.	Stakeholder intent, outcomes, obligations, constraints, and unacceptable outcomes.
Primary artifact	Automated test and production code.	Executable behavior model, scenario evidence, and validated baseline.
Typical scope	Small increment, unit, component, interface, or service boundary.	Cross-component, cross-team, system, operational, and sociotechnical behavior.
Feedback timing	During implementation, in short cycles.	Before and throughout implementation, deployment, and operations.
Principal evidence	Fail/pass result and regression suite.	Execution traces, states, interactions, outcomes, counterexamples, and limitations.
Design contribution	Emergent code and interface design through refactoring.	Behavioral, architectural, control, authority, and lifecycle decisions.
Principal limitation	Cannot independently validate the correctness or completeness of its expectations.	Does not prove that production code is implemented or operating correctly.

Table 1. TDD and SDSE produce different but complementary forms of executable evidence.

The comparison should not be read as a rigid boundary. Some schools of TDD work outside-in and begin at a user-visible interface. Acceptance Test-Driven Development and Behavior-Driven Development

broaden stakeholder involvement. Conversely, an SDSE model can be bounded to one service or feature. The distinction remains useful because TDD primarily drives implementation from selected expectations, while SDSE primarily validates and explores the behavioral system from which implementation expectations should be derived.

5. Unit Behavior Is Not System Behavior

Software behavior emerges from interaction. A payment service, inventory service, notification service, retry controller, database, operator, and policy engine can each satisfy its local tests while their combined timing and state produce an unacceptable customer outcome. Component tests are necessary, but composition introduces behavior that does not exist inside any one component.

Participant	Locally correct behavior	Emergent system risk
Payment	Authorize once and return a transaction identifier.	A delayed response causes the caller to retry after authorization succeeded.
Order service	Retry when no confirmation arrives.	The retry uses a new idempotency key and creates a second charge.
Inventory	Reserve stock after payment confirmation.	One charge confirms after the reservation expires.
Notification	Send a receipt for every confirmed transaction.	The customer receives two receipts and assumes two valid orders.
Operations	Reconcile exceptions overnight.	The duplicate is discovered after settlement and customer impact.

Table 2. Local correctness can coexist with an incorrect end-to-end outcome.

TDD can implement the idempotency behavior correctly once the team knows it is required. SDSE helps the team discover and validate that need by executing the cross-service sequence, including delayed confirmation, timeout, retry, expiration, reconciliation, and recovery, before implementation choices harden.

Behavioral dimensions that often cross test boundaries

- 18.State history, sequence, timing, ordering, concurrency, and eventual consistency.
- 19.Partial failure, retry, timeout, duplication, compensation, rollback, and recovery.
- 20.Permissions, delegated authority, escalation, human intervention, and responsibility transfer.
- 21.Interactions among software, people, policy, devices, suppliers, and external services.
- 22.Environmental, data, load, and operating conditions that change the correct response.

6. The Difference Between a Test and a Simulation

A test and a simulation are both executable, but they organize evidence differently. A test asserts that a defined result should occur when an implementation is exercised under specified conditions. A simulation executes a behavioral model to reveal the consequences of interacting rules, conditions, states, and participants. The same scenario can support both activities, but the questions and artifacts differ.

Aspect	Test	Simulation
Purpose	Verify a selected expectation.	Explore and validate a proposed behavioral system.
Subject	Implementation code or an executable interface.	An explicit model of behavior before or alongside implementation.
Result	Pass, fail, error, and diagnostic output.	Trace, state evolution, interactions, outcomes, unmet expectations, and model limits.
Unknowns	Usually checks the conditions represented by the test author.	Can expose interactions and consequences that challenge the author's assumptions.
Change cost	Reduces implementation and regression correction cost.	Reduces the cost of changing intent, behavior, architecture, controls, and scope late.

Table 3. Tests and simulations answer related but distinct engineering questions.

A useful distinction

Tests ask whether the implementation did what was expected. Simulations help determine what should be expected when consequential behavior interacts over time.

7. Comparing the Development Loops

ONE CONTINUOUS CHAIN OF ENGINEERING EVIDENCE

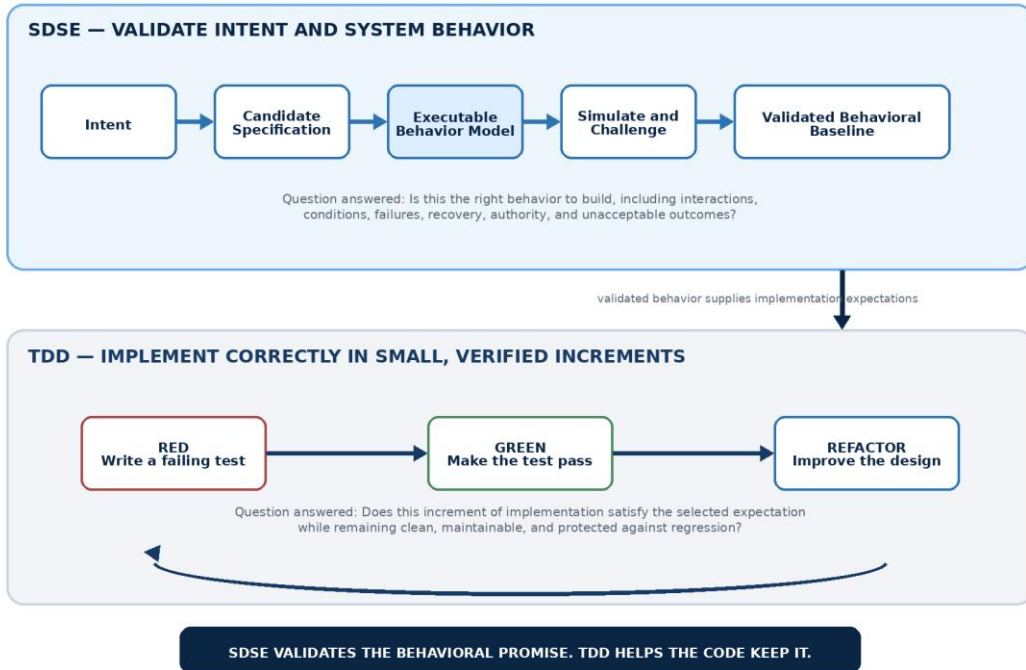


Figure 1. SDSE and TDD form one continuous chain from validated intent to verified implementation.

The loops operate at different scales and can run concurrently. SDSE does not require the organization to freeze a complete design before coding. It establishes a behavioral baseline at the fidelity needed for a consequential decision. TDD then implements selected behavior in small increments. Findings from implementation, integration, or operations can revise the Candidate Specification and trigger resimulation.

Loop	Core sequence	Primary output
TDD	Write a failing test → implement enough to pass → refactor → repeat.	A clean implementation increment protected by tests.
SDSE	Establish intent → model behavior → simulate → review → validate → learn and revise.	A justified behavioral baseline and decision evidence.
Integrated	Validate the promise → derive expectations → implement incrementally → compare → revise.	Traceable evidence from intent through operation.

Table 4. The loops reinforce one another when connected explicitly.

8. How SDSE Strengthens TDD

TDD is only as sound as the expectations that drive it. SDSE improves the source, coverage, and context of those expectations. Instead of inventing isolated tests from prose requirements, a team can derive tests from a behavioral baseline that stakeholders have already challenged through simulation.

SDSE/BSN source	Derived test family	Implementation evidence
BSN participant and interface	Unit and contract tests	Verify component responsibilities and interface promises.
State and transition	State-based unit tests	Verify allowed transitions, guards, invariants, and invalid transitions.
Interaction sequence	Integration and workflow tests	Verify ordering, handoffs, correlation, and end-to-end effects.
Failure and recovery scenario	Resilience, chaos, and recovery tests	Verify timeout, retry, rollback, compensation, and restored safe operation.
Authority rule	Security and policy tests	Verify permission, delegation, escalation, revocation, and intervention.
Scenario expectation	Acceptance and conformance tests	Compare delivered behavior with the accepted baseline.

Table 5. A validated behavior model can supply coherent expectations across the test portfolio.

- 23.Reduce contradictory tests created from inconsistent team interpretations.
- 24.Identify missing tests before implementation begins.
- 25.Preserve the reason for each expectation, including consequence and decision context.
- 26.Connect happy-path tests with boundary, degraded, failure, recovery, misuse, and intervention tests.
- 27.Provide a stable comparison point when AI generates code and tests together.

9. How TDD Strengthens SDSE

A validated model is not the production system. TDD provides the small-step implementation discipline required to realize modeled behavior without turning the baseline into a large speculative design. It also creates fast, local evidence that a component continues to honor its assigned responsibilities as the code evolves.

- 28.Turn selected behavioral expectations into executable implementation checks.
- 29.Create clean interfaces that make conformance and integration easier to observe.
- 30.Protect incremental realization of modeled states, rules, and interactions.
- 31.Detect implementation drift during refactoring and maintenance.

32. Provide rapid feedback below the fidelity and scope of the system simulation.

The reciprocal relationship

SDSE improves what the tests should mean. TDD improves how reliably the code realizes that meaning.

This relationship also protects SDSE from becoming documentation theater. If accepted behavior cannot be connected to implementation tests, controls, or telemetry, the baseline is not yet serving the lifecycle. TDD is one of the most direct mechanisms for turning behavioral intent into durable executable evidence.

10. TDD and SDSE in Agile Delivery

SDSE is compatible with iterative and incremental delivery. The alternative to code-first development is not necessarily a large up-front specification. Teams can model only the behavior needed to improve the next consequential decision, validate it, implement an increment with TDD, compare results, and revise the model as learning accumulates.

Lifecycle activity	SDSE contribution	TDD contribution
Discovery	Identify outcomes, unacceptable outcomes, stakeholders, dependencies, and the decision requiring evidence.	Define the next smallest valuable implementation increment.
Refinement	Model difficult behavior and simulate selected scenarios.	Turn validated expectations into testable slices.
Implementation	Maintain the baseline as material questions change.	Use red–green–refactor for each selected behavior.
Integration	Compare end-to-end traces and outcomes with the model.	Run contract, integration, conformance, and regression tests.
Release	Connect accepted behavior and residual risk to release controls.	Keep the implementation suite green and verify deployment readiness.
Operations	Compare telemetry and incidents with modeled assumptions; resimulate material change.	Add regression tests for confirmed implementation defects and drift.

Table 6. Agile delivery can use both loops without creating a separate waterfall phase.

The practical unit of SDSE planning is not “complete the model.” It is “resolve enough behavioral uncertainty to make the next decision responsibly.” That keeps modeling proportional and preserves the fast feedback that makes TDD and Agile effective.

11. TDD and SDSE for AI-Native Systems

AI increases the value of both practices and makes their separation more dangerous. Generative AI can produce code, tests, configuration, and infrastructure faster than humans can examine the assumptions embedded in them. If the same model generates an implementation and its tests from the same ambiguous prompt, the artifacts may be internally consistent while sharing the same misunderstanding.

Agentic systems add a second challenge. Their behavior includes permissions, tool selection, delegated action, uncertainty, escalation, external communication, monitoring, revocation, recovery, and human intervention. These are system and governance behaviors, not only function outputs. SDSE provides the earlier environment for modeling and challenging them; TDD provides precise checks for the deterministic components, safeguards, adapters, and policy enforcement mechanisms that implement them.

AI-native concern	Primary evidence combination	What must be demonstrated
Generated function or component	TDD	Precise examples, edge cases, invariants, regression, and refactoring safety.
Agent interaction and workflow	SDSE + integration evidence	Sequence, state, tool use, dependencies, time, failure, and recovery.
Permission and delegation	SDSE + security tests	Allowed actions, thresholds, escalation, revocation, and forbidden outcomes.
Human control	SDSE + control tests	Approval, explanation, intervention, stop, responsibility transfer, and evidence retention.
Learning or model change	SDSE + regression portfolio	Behavioral drift, revised assumptions, resimulation, and renewed acceptance.

Table 7. AI-native engineering requires evidence about both implementation and delegated behavior.

Human-in-Control principle

AI may propose, generate, simulate, test, or execute. Accountable humans must authorize intent, unacceptable outcomes, delegated boundaries, material change, and residual risk.

12. Common Misconceptions

“SDSE is just model-based testing.” Model-based testing derives tests from models to evaluate an implementation. SDSE uses simulation earlier to validate intent, behavior, alternatives, controls, and consequential decisions, then carries that baseline into implementation and operations.

“TDD already provides executable specifications.” TDD tests can be excellent executable implementation specifications. The question is whether their expectations represent a validated and sufficiently complete system-level behavioral promise.

“Simulation replaces testing.” A simulation cannot prove that production code, infrastructure, data, integrations, or operating procedures are implemented and functioning correctly. Testing remains essential.

“SDSE requires modeling everything.” Modeling depth should be proportional to consequence, uncertainty, interaction complexity, and the decision the evidence must improve.

“TDD is only unit testing.” TDD can be practiced outside-in and at different boundaries. Its defining discipline is the test-first implementation loop, not one mandatory test size.

“A high coverage number proves correct behavior.” Coverage shows what code was exercised, not whether expectations were correct, scenarios were consequential, assertions were meaningful, or system behavior was acceptable.

13. When TDD May Be Sufficient

Not every change warrants behavioral simulation. The cost of SDSE should be justified by the uncertainty or consequence it can reduce. TDD, supported by ordinary review and integration practices, may be sufficient when most of the following conditions are true:

- 33.The behavior is small, familiar, and contained within a well-understood boundary.
- 34.Interfaces, operating conditions, and upstream or downstream expectations are stable.
- 35.State, timing, concurrency, failure, recovery, and authority are straightforward.
- 36.The consequence of misunderstanding is low and correction remains inexpensive.
- 37.Existing acceptance, integration, security, and operational evidence is coherent and trusted.
- 38.No consequential architectural, control, policy, release, or authority decision depends on earlier simulation evidence.

Proportionality rule

Use the lightest evidence system that can support the consequence and uncertainty of the decision. SDSE is selective engineering, not universal ceremony.

14. When SDSE Should Be Added

SDSE trigger	Observable condition
--------------	----------------------

SDSE trigger	Observable condition
Behavioral complexity	State, sequence, timing, concurrency, or conditions make combined behavior difficult to infer.
Cross-boundary dependency	Important outcomes span teams, systems, suppliers, policies, or organizations.
Cost of late discovery	Misunderstanding after architecture or implementation creates material rework, delay, or customer impact.
Consequence	Incorrect behavior affects safety, security, privacy, finance, compliance, operations, or trust.
Failure and recovery	Timeout, retry, degraded mode, compensation, rollback, recovery, or intervention is consequential or unclear.
AI autonomy	Software can select tools, delegate, communicate, transact, modify environments, or act externally.
Evidence gap	Requirements, tests, controls, and telemetry do not share one coherent behavioral reference.
Decision leverage	Executable evidence can still change scope, architecture, controls, release conditions, or authority.

Table 8. SDSE is justified where earlier behavioral evidence can change a consequential decision.

One critical trigger can justify action even when the aggregate profile appears moderate. Irreversible safety consequences, material external AI action, recurring incidents caused by conflicting assumptions, or unexercised recovery behavior should receive direct leadership attention.

15. An Integrated SDSE–TDD Operating Model

1. **Establish intent.** Identify stakeholders, outcomes, unacceptable outcomes, obligations, constraints, and the decision that needs evidence.
2. **Create a Candidate Specification.** State proposed behavior clearly enough to expose assumptions, boundaries, and unresolved questions.
3. **Build the behavior model.** Represent consequential participants, interfaces, state, conditions, logic, interactions, scenarios, and expectations using BSN.
4. **Simulate by consequence.** Exercise normal, boundary, timing, failure, degraded, misuse, recovery, and intervention behavior proportional to risk.
5. **Review and validate.** Examine traces, contradictions, counterexamples, limitations, and residual uncertainty; approve a versioned baseline.
6. **Derive the implementation portfolio.** Map baseline expectations to unit, contract, integration, acceptance, security, resilience, and conformance tests.
7. **Implement with TDD.** Use red–green–refactor to realize selected behavior in small, clean, protected increments.
8. **Integrate and compare.** Verify that combined implementation behavior conforms to the baseline under consequential conditions.
9. **Release with evidence.** Tie release criteria, controls, rollback, telemetry, and residual risk to the accepted behavioral promise.
10. **Learn and revise.** Use incidents, drift, usage, environmental change, and new obligations to update the model, tests, code, and decisions.

Single promise thread

The model, test portfolio, implementation, controls, and telemetry should remain connected to one versioned behavioral promise rather than becoming independent sources of truth.

16. Practical Adoption Guidance

The strongest adoption strategy is a bounded engineering experiment, not a broad tool rollout. Select a use case important enough to demonstrate value but narrow enough to complete before the target decision. Establish a baseline, train the team through the real work, and decide in advance what evidence will justify stopping, revising, repeating, integrating, or scaling.

Pilot element	Leadership question
---------------	---------------------

Pilot element	Leadership question
Decision	What consequential choice can still change because of the evidence?
Behavioral uncertainty	Which interaction, state, timing, failure, recovery, or authority question is unresolved?
Boundary	What behavior is included, excluded, and assumed?
Baseline	What rework, clarification, delay, defect, incident, or risk evidence exists today?
Participants	Which product, domain, engineering, assurance, security, operations, and governance roles are required?
TDD connection	Which validated scenarios will drive implementation tests and conformance evidence?
Continuation rule	What evidence will cause the organization to stop, revise, repeat, integrate, or scale?

Table 9. A credible pilot is bounded around a decision, not around a demonstration of a tool.

Adoption traps

11. Selecting trivial work because it is politically safe, then concluding that SDSE has little value.
12. Beginning after scope and architecture are fixed, reducing simulation to documentation.
13. Modeling broadly without naming the decision the evidence must improve.
14. Treating model count, scenario count, test count, or code coverage as the business outcome.
15. Allowing the model and test suite to diverge into competing specifications.
16. Automating every integration before proving that the combined practice improves a worthwhile decision.

17. Measuring the Combined Value

The business case should connect engineering activity to decisions, delivery outcomes, and business consequences. It should include the full investment in training, facilitation, modeling, testing, review, tooling, integration, and maintenance. It should also avoid claiming that every counterexample certainly would have become a production incident. Credibility comes from a comparison baseline, explicit assumptions, and measurable outcomes.

Value layer	Measures to consider
Capability	Time to express behavior, simulate scenarios, derive tests, interpret evidence, and approve a baseline.
Leading evidence	Questions, contradictions, missing expectations, unsafe paths, and recovery gaps found before implementation.
Decision quality	Material scope, architecture, control, release, or authority decisions changed by evidence.

Value layer	Measures to consider
Delivery outcome	Clarification after work begins, estimate change, rework, test churn, integration delay, and behavioral escapes.
Code outcome	Defect detection, maintainability, refactoring safety, test stability, and implementation lead time.
Operational outcome	Incidents, rollback, recovery time, repeated failures, and behavioral drift.
Business outcome	Value realization, failure cost, safety, compliance, financial impact, customer trust, and acceptable autonomy.

Table 10. Measure the decisions and outcomes improved—not the volume of engineering artifacts.

Example value hypothesis

If a team validates the selected cross-service behavior before implementation and derives TDD tests from that baseline, then behavioral clarification, test contradiction, integration rework, and late correction should decline relative to a comparable baseline, while modeling and maintenance cost remains lower than the cost avoided or risk reduced.

18. Leadership Implications

Leaders should not frame TDD versus SDSE as a competition between engineering methods or tools. The choice is about where uncertainty resides. When the main uncertainty concerns implementation, TDD provides excellent leverage. When the main uncertainty concerns the intended behavior of an interacting system, earlier simulation may be necessary before implementation tests can be trusted as the governing evidence.

17. Protect TDD as a disciplined developer feedback practice rather than reducing it to test coverage compliance.
18. Fund behavioral discovery and validation before consequential commitments become expensive to revise.
19. Require one accountable owner for accepting the behavioral baseline and residual uncertainty.
20. Keep the BSN model, derived tests, implementation evidence, release controls, and operational learning connected.
21. Allow unfavorable simulation or test evidence to revise scope, design, controls, schedule, release, and delegated authority.
22. Scale only after a bounded pilot demonstrates improved decisions, outcomes, and reusable learning.

Leadership question

What important software decision would be better if the organization could execute and challenge the proposed behavior before committing to implementation?

Conclusion

TDD and SDSE begin at different points because they reduce different uncertainties. TDD begins with an expectation and uses rapid implementation feedback to create clean code that works. SDSE begins with intent and uses executable behavioral evidence to determine whether the proposed system should work that way in the first place. One protects the implementation. The other protects the behavioral promise.

The distinction becomes especially important in distributed, safety- or security-sensitive, regulated, and AI-native systems. In these environments, local correctness can coexist with unacceptable system behavior. Tests can pass while interacting services violate timing, recovery, authority, or customer expectations. The answer is not to weaken TDD. It is to give TDD better expectations—grounded in an explicit, simulated, reviewed, and versioned behavioral baseline.

The practical starting point is small but consequential: one bounded behavior, one accountable decision, one executable model, one validated scenario set, and one implementation team using TDD to realize the accepted promise. If the combined practice improves the decision and produces reusable learning, integrate and scale it proportionally. If it does not, revise or stop. That evidence-based discipline reflects the purpose of both TDD and SDSE.

Closing statement

SDSE validates what the system should do. TDD helps ensure that the code does it. Together they create confidence from intent through operations.

Appendix A. TDD–SDSE Comparison Checklist

Review area	Question
Expectation source	Has the expected behavior been validated with affected stakeholders and system owners?
System scope	Does the behavior remain inside one boundary, or does it emerge across interactions?
Difficult conditions	Are state, timing, failure, recovery, misuse, and authority represented?
Decision leverage	Can executable evidence still change an important decision?
Test derivation	Can each consequential modeled expectation be mapped to an appropriate test or control?
Conformance	Can implementation traces and outcomes be compared with the behavioral baseline?
Lifecycle continuity	Do telemetry and incidents trigger model, test, and implementation revision?

Table A1. Use the checklist to identify where the evidence chain is incomplete.

Appendix B. Decision Guide—TDD Alone or TDD with SDSE?

Condition	Recommended approach	Leadership implication
Low consequence, familiar, contained behavior	TDD with normal review and integration evidence	Keep the feedback loop small and proportional.
Moderate uncertainty or cross-boundary dependency	Targeted SDSE plus TDD	Model only the interactions and scenarios that affect the decision.
High consequence, complex failure/recovery, or AI authority	SDSE baseline plus full implementation evidence portfolio	Require accountable validation before commitment and conformance afterward.
High need but low organizational readiness	Prepare, then pilot	Close authority, participation, skill, information, and measurement gaps.

Table B1. Select the evidence approach according to uncertainty and consequence.

Appendix C. Example Mapping from BSN Scenario to TDD Tests

Artifact	Example
Scenario	Payment confirmation is delayed beyond the caller timeout; the caller retries; only one financial authorization may exist.
Model expectation	Repeated requests sharing the transaction intent are idempotent; late confirmation is reconciled; the customer receives one final outcome.
Unit TDD tests	Idempotency-key validation, duplicate-command handling, state transition guards, and reconciliation rules.

Artifact	Example
Contract tests	Retry retains correlation and intent identifiers; response semantics distinguish pending, confirmed, and rejected.
Integration tests	Delayed response plus retry produces one authorization and one order outcome.
Recovery tests	Reconciliation resolves late confirmation and restores consistent customer, payment, and inventory state.
Operational evidence	Telemetry identifies duplicate attempts, correlation, reconciliation outcome, and unresolved exceptions.

Table C1. One validated scenario can drive a coherent multi-level evidence portfolio.

Appendix D. Integrated Pilot Worksheet

Field	Working answer
Proposed behavior and consequence	_____
Decision to improve and decision owner	_____
Model boundary and exclusions	_____
Priority scenarios	_____
Validated expectations	_____
Derived TDD and integration tests	_____
Baseline measures	_____
Pilot participants and duration	_____
Continuation decision and date	_____

Table D1. Bound the pilot around one consequential decision and evidence chain.

Appendix E. Leadership Review Checklist

- 23.The expected behavior has an accountable owner and explicit unacceptable outcomes.
- 24.TDD is used as a design and feedback discipline, not only a coverage target.
- 25.The need for system-level simulation has been assessed separately from implementation testing.
- 26.The behavioral model is bounded by consequence and decision value.
- 27.Validated scenarios are mapped to the appropriate unit, contract, integration, security, resilience, and conformance evidence.
- 28.The organization can revise commitments when simulation or tests reveal unfavorable evidence.

29. Operational evidence can trigger model, test, control, and implementation revision.

30. The pilot will end with an explicit decision to stop, revise, repeat, integrate, or scale.

References

The following sources support the description of TDD, the red–green–refactor workflow, empirical qualifications, secure development context, software engineering practice, and the SDSE discipline. The comparative framework and integrated operating model are presented by the author.

- [1] Beck, Kent. Test-Driven Development: By Example. Addison-Wesley, 2002. [Book record](#)
- [2] Agile Alliance. “What Is Test-Driven Development?” [Official glossary](#)
- [3] Microsoft. “Set Up a Test-Driven Development Flow in Visual Studio Code.” [Official guide](#)
- [4] Fucci, Davide, et al. “A Dissection of the Test-Driven Development Process.” IEEE Transactions on Software Engineering, 2017. [IEEE record](#)
- [5] Santos, Adrian, et al. “A Family of Experiments on Test-Driven Development.” Empirical Software Engineering, 2021. [Research record](#)
- [6] Rafique, Yahya, and Vojislav B. Misic. “The Effects of Test-Driven Development on External Quality and Productivity: A Meta-Analysis.” IEEE Transactions on Software Engineering, 2013. [Research record](#)
- [7] Wang, Fanyu, et al. “Requirements-Driven Automated Software Testing: A Systematic Review.” 2025. [Research paper](#)
- [8] National Institute of Standards and Technology. SP 800-218, Secure Software Development Framework Version 1.1, 2022. [Official publication](#)
- [9] IEEE Computer Society. Guide to the Software Engineering Body of Knowledge, Version 4. [Official SWEBOK resource](#)
- [10] Hornbeek, Marc, with Debashis Bhattacharyya and Jitin Dhanani. Simulation-Driven Software Engineering: A Behavior-First Discipline for Modern Systems, 2026. [Purchase on Amazon](#)

About the Author

Marc Hornbeek, a.k.a. DevOps_the_Gray, is an award-winning engineer, author, educator, executive consultant, and keynote speaker with five decades of experience across systems engineering, packet networks, software development, behavioral simulation, testing, automation, DevOps, DevSecOps, SRE, platform engineering, and intelligent software delivery. His early engineering work included integrated tools for packet-network simulation, performance, conformance, observation, and test automation. He is the originator of Simulation-Driven Software Engineering and leads the development of Behavior Simulation Notation and the BSN Workbench. Marc received the IEEE Region 6 Outstanding Engineer Award in 2016 and has authored multiple books and professional certification courses. He founded PrimaModus® to help organizations model, simulate, validate, and engineer software that behaves as intended.

Contact: mhornbeek@engineeringdevops.com