WHITE PAPER

Constraint-Driven Development

Reducing Recurring Bug Classes Through Structural Constraints

Damir Majer
ASE Academy, Munich

May 2026
ASE ACADEMY

Contents

Who This Paper Is For

Developers, architects, and team leads who invest heavily in testing and code reviews but still face recurring categories of bugs. This paper presents a structural model for eliminating bug categories by design, not by detection.

Abstract

Testing catches individual failures. Structural constraints can prevent some recurring defect classes and reduce others before they occur. This paper introduces Constraint-Driven Development (CDD) and its central model, the Constraint Stack: a six-level model linking defect patterns to structural interventions across code, design, architecture, domain, system, and verification. The framework draws on practitioner observation across 187 codebases and integrates ideas from type theory, paradigm analysis, and domain-driven design. Its strongest claims concern recurring software design and implementation defects rather than every possible failure mode.

Foreword

This series marks twenty years of my own firm, Majer Consulting (majcon). It is not a survey of the field but the set of ideas that two decades of practice and training have most shaped, set down plainly. Read them as a practitioner's case, offered in good faith and open to challenge.

Key Takeaways

- Testing catches instances. Constraints can prevent some recurring defect classes and materially reduce others. The distinction between analytical and constructive quality management is well established. CDD formalizes the constructive side.
- Six levels, six recurring defect areas. Code, Design, Architecture, Domain, System, and Verification. Each level introduces constraints that can prevent or materially reduce specific classes of defects.
- Paradigms converge. OOP, FP, Event-Driven, and UI paradigms often express similar structural constraints under different names.
- Same principles, deeper perception. Developers do not always learn new principles; they often learn to see familiar ones at deeper levels.
- The claim is testable within scope. If major recurring software defect classes within this scope resist structural mapping, the model is incomplete.

1. The Bug Paradox

Every team I work with has the same story. They write tests. Good tests. They follow naming conventions. They use design patterns. They invest in code reviews. And still, the same categories of bugs come back. Not the same bugs. The same categories.

85% test coverage. Still ships business logic bugs. Clean hexagonal architecture. Still finds race conditions. Domain-Driven Design with all the patterns. Still breaks integrations between services.

Testing catches instances. A test finds this specific bug. Fix it, write a regression test, move on. The category stays open. Next sprint, a different bug from the same category. Different symptom, same root cause.

> Testing catches instances. Structural constraints can prevent some recurring defect classes and materially reduce others.

Quality = Constraint Coverage at Every Level

We measure quality by how many bugs we catch. The better question: how many recurring defect classes can we address structurally before they recur? The distinction between analytical and constructive quality management [12] is well established. CDD formalizes the constructive side into a complete, leveled model.

2. Constraints, Not Tests

> Adopt Programming Habits that Constraint you, to Help You to limit Mistakes. Any Programming Paradigm only Assists with that.
>
> Dave Farley [2]

Checking for a problem and making a problem impossible are fundamentally different things. An order must have at least one item. Two ways to enforce this:

Approach A: Runtime Check
The code accepts an empty list, then checks: "Is this list empty? If yes, throw an error." The constraint exists in the logic. It can be forgotten. It can be bypassed.

Approach B: Structural Constraint
The type system requires a `NonEmptyList`. An order with zero items cannot be represented. Not "is rejected at runtime." Cannot be constructed.

In code, the difference looks like this:

Runtime check (Level 0)

```
Order createOrder(List<Item> items) {
  if (items.isEmpty())
    throw new IllegalArgumentException();
  return new Order(items);
}
```

Structural constraint (Level 4)

```
Order createOrder(NonEmptyList<Item> items) {
  return new Order(items);
}
// Empty order? Cannot be expressed.
```

Pugh's Prefactoring [13] proposed applying design experience proactively, but acknowledged: guidelines are advisory and can be forgotten or bypassed.

In this paper, constraint refers to a structural property of the code that prevents or reduces a defect class – from type-system guarantees that make errors unrepresentable to architectural patterns that contain side effects.

This is the difference between a guardrail and the absence of a cliff. Guidelines build better guardrails. Strong constraints can sometimes remove the cliff entirely, especially at the level of representation. More often, at higher levels of system design, they materially reduce the probability and recurrence of a defect class rather than eliminating it in an absolute sense.

3. The Constraint Stack

Bugs do not appear randomly. They cluster by category. Each category lives at a specific level of the software. The Constraint Stack is our contribution: it synthesizes established results from type theory, paradigm analysis, and domain-driven design into a single six-level model. Each level guards a category of defects.

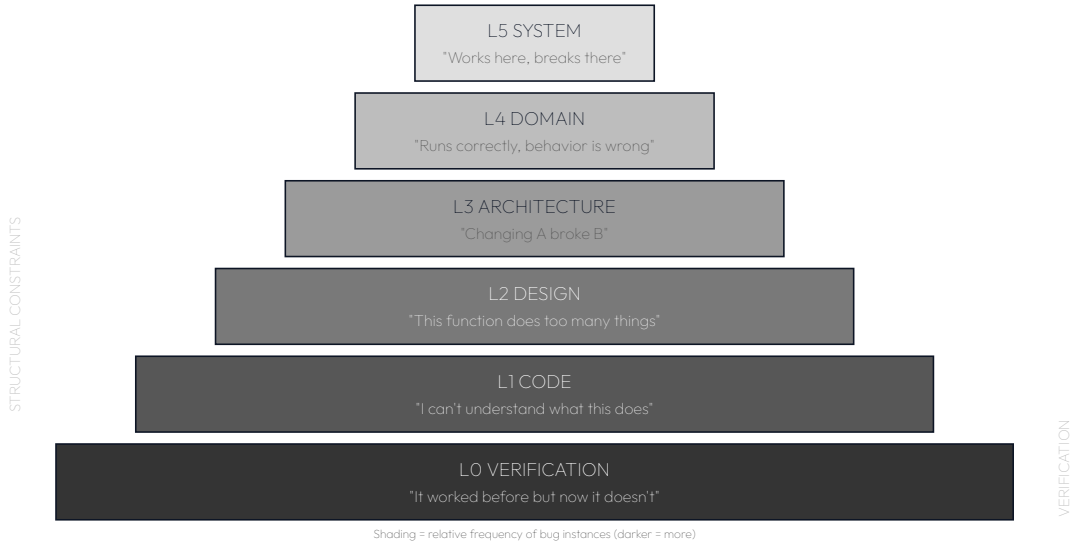

Fig. 1. The Constraint Stack. Six levels, each guarding a specific category of defects; shading encodes the relative frequency of bug instances per level (verification-level bugs are frequent, structural-level bugs rare but systemic).

Level 5: System

Guards: cross-context integration, semantic confusion, cascade failures

Each bounded context owns its own model. Communication uses explicit domain events. Conway's Law \[9\] is satisfied by design, not by accident.

"It works in our context but breaks in theirs."

Level 4: Domain

Guards: business logic correctness, domain model integrity

Aggregates \[5\] enforce invariants at construction time. Value Objects wrap domain concepts as immutable types \[4\]. The language in the code matches the language of the domain.

"The code runs correctly but the behavior is wrong."

Level 3: Architecture

Guards: component coupling, dependency direction, side effect control

Pure core, impure shell [6]. Dependencies point inward. Side effects live at the boundary only.

"Changing module A broke module B."

Level 2: Design

Guards: function-level complexity, coupling, mutation

Separation of integration and operation [11]. Communication through messages, not data inspection. Immutability by default. Command-query separation.

"This function does too many things and I can't tell which part failed."

Level 2 in practice: Integration/Operation Segregation Principle (IOSP) [11]

Before: mixed

```
void processOrder(Order o) {
  var price = o.items().stream()
    .mapToDouble(Item::price).sum();
  var tax = price * 0.19;
  db.save(new Invoice(price + tax));
  mailer.send(o.customer(), price);
}
```

After: separated

```
// Operation: pure computation
Invoice calculate(Order o) {
  var price = sum(o.items());
  return new Invoice(price, tax(price));
}
// Integration: orchestrates only
void processOrder(Order o) {
  var inv = calculate(o);
  db.save(inv);
  mailer.send(o.customer(), inv);
}
```

Level 1: Code

Guards: readability, structural clarity, comprehensibility

Small functions. Intention-revealing names. Simple design rules [10]. The five quality dimensions (readability, maintainability, changeability, extensibility, testability) [12] are structural properties that can be constrained, not just measured.

"I can't understand what this code does."

Level 0: Verification

Guards: behavioral correctness, regression, specification gaps

Tests, property-based testing, contract verification. Static source analysis [12] automates constraint checking. This is where most teams start and stop. Necessary but reactive.

"It worked before but now it doesn't."

> Structured programming removed uncontrolled goto. OOP removed direct access to function pointers. FP removed assignment. Each paradigm restricts, not enables.
>
> Robert C. Martin [1]

Reading the Stack

LEVEL	SCALE	QUESTION	IF UNGUARDED...
5	System	Does the system reflect the domain?	Integration bugs, cascade failures
4	Domain	Does the code represent the right things?	Business logic bugs, wrong behavior
3	Architecture	Are the boundaries clean?	Coupling bugs, side effect leaks
2	Design	Are the functions well-structured?	Complexity bugs, mutation bugs
1	Code	Is the code readable and clear?	Comprehension bugs, naming confusion
0	Verification	Can we prove it works?	Regression, undetected failures

In practice, Level 4 (Domain) is where most teams have the widest gap.

Each level targets a recurring defect area. Broader coverage reduces the number of unguarded entry points.

Structural discipline is not a quest for absolute restriction, but for the Window of Optimal Constraint. The Economics of Changeability (EoC) [19] extends this concept to the economic domain. Too few constraints produce entropy – recurring defect classes that drain resources. Over-constraint produces rigidity – where evolving the design becomes prohibitive. The economic sweet spot is where constraints are strong enough to close defect classes but flexible enough to allow for the pivots that changeability requires (EoC).

4. The Depth Model

Developers do not learn new principles at each level. They see existing principles through a deeper lens. Take one principle from the object-design tradition [10]: an object should represent a single concept.

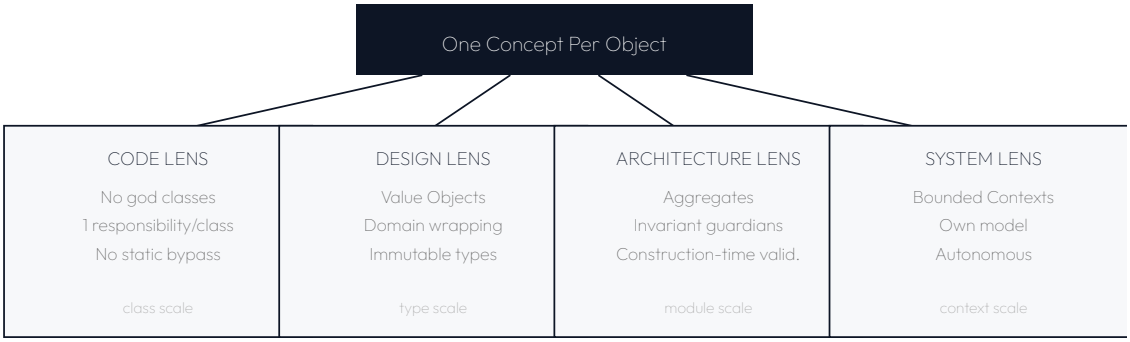

Fig. 2. The Depth Model. One principle seen through four lenses at increasing scale. The lenses accumulate.

Same principle. Four levels of depth. Four categories of bugs addressed. The Depth Model works like zooming in and out of a map: at each altitude, different constraints and structural features become visible. A developer operating at the Code lens is not a novice; all four lenses are active simultaneously in mature practice. A lens determines at what scale you look. A perspective determines as whom you think (developer, tester, architect, domain expert). They compose. “Architecture Lens, tester’s perspective” means: write a property-based test to verify the aggregate invariant holds for all inputs.

5. Paradigm Convergence

> OO makes code understandable by encapsulating moving parts. FP makes code understandable by minimizing moving parts.
>
> Michael Feathers [3]

The industry treats paradigms as competing camps. After examining well-constrained code across paradigms, the structural properties converge. Take the principle “separate orchestration from computation”:

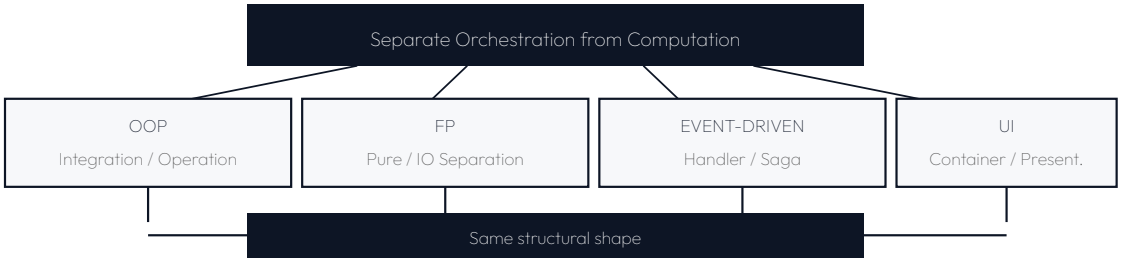

Fig. 3. Paradigm Convergence. The same constraint appears across four paradigms. Different vocabulary. Same structural shape.

Empirical signal. One pattern stood out across the curated dataset: stronger enforcement of object cohesion ("single-concept-per-object") tended to coincide with higher structural quality scores and lower observed defect pressure. This pattern appeared across paradigms and team sizes, but the data remains exploratory rather than confirmatory.

Methods Note: Analysis of 187 Codebases

The empirical observations in this paper derive from a longitudinal review (2009–2024) of 187 codebases across 12 languages (ABAP, Java, TypeScript, Python, C#, Kotlin, Haskell, and others).

- Source: 60% coaching engagements, 20% open-source repositories, 20% book/training/conference examples. The collection is curated convenience sampling, not random sampling.
- Evaluation: Each codebase was independently reviewed by two senior architects using practitioner rubrics. Inter-rater disagreements were resolved by calibration discussion.
- Interpretation: The resulting patterns should be read as exploratory practitioner research rather than confirmatory statistical evidence.

Limitations: Selection bias is inherent in convenience sampling. The correlations do not establish causation. Replication with independent datasets is needed.

6. Evidence and Honest Limits

What We Know

Immutability can eliminate state-related bugs. Race conditions cannot exist when there is no mutable shared state. This follows from the definition.

Paradigms remove specific bug categories. Martin [1]: Structured programming removed uncontrolled goto. OOP removed direct access to function pointers. FP removed assignment.

Large-scale data is suggestive but contested. 729 projects, 17 languages [7]: "small but significant" relationship between functional languages and fewer defects. A reproduction study [8] questioned the methodology and found a smaller effect size.

"Build quality in" is a foundational principle. Deming [15] argued that 85–94% of quality problems belong to the system, not the individual, and that quality comes from improving the process, not from inspecting the output. Leveson [16] extended this to system safety: accidents arise from inadequate constraints on component interactions, not from component failures alone. CDD applies both insights to software: quality is structural, not inspectional.

Structural practices, not tools, drive the payoff. Forsgren et al. [20] surveyed 219 developers and found that teams with faster responses to developer questions report 50% less technical debt, and that developers who understand

their code well feel 42% more productive. DORA's research on AI-assisted development [21] sharpens the point: AI is "an amplifier, magnifying an organization's existing strengths and weaknesses," and the greatest returns come "not from the tools themselves, but from a strategic focus on the underlying organizational system." Where AI-accelerated generation outpaces a team's foundational practices (automated testing, CI/CD, code review), delivery stability declines. CodeScene's 2026 peer-reviewed study points the same way from the code itself: across 5,000 AI-refactored programs, defect risk rose sharply in less-healthy code while healthy code acted as a buffer, leading the authors to conclude that code quality is now "a prerequisite for safe and effective use of AI" [22]. More code without more structure makes things worse, not better. That is the CDD claim stated in empirical terms.

Constructive quality management prevents structurally. Balzert [14] established the distinction between analytical (testing) and constructive (structural) quality management. The formula Professionality = Awareness + Principles [12] builds on this: quality is a structural property of how work is done, not a metric applied after the fact.

AI-generated code amplifies the need for constraints. GitClear's analysis of 211 million lines of changed code found that code cloning quadrupled after AI coding tools became widespread [17]. When generation is effortless, structural discipline is the only defence against duplication debt.

> Professionality = Awareness + Principles. Quality is a structural property of how work is done, not a metric applied after the fact.
>
> After Balzert [14] and Arlitt et al. [12]

What We Do Not Know

Nobody has studied the full stack. Each study covers one level in isolation. The combined effect of stacking all six levels is unmeasured. The argument is mechanistic: coherent and plausible, but not yet demonstrated at scale.

Zero-bug is asymptotic. Requirements ambiguity, novel failure modes, deployment context, and human error will always remain. Domain constraints are the least studied. Level 4 is where most teams have a gap.

> The model predicts that many recurring software defect classes can be mapped to structural weaknesses. If major recurring defect classes within that scope resist structural classification and prevention, the model is incomplete.

7. The Diagnostic

To make the Constraint Stack practical as a diagnostic tool:

1. **Map Current Constraints**
   For each level (0-5), list the constraints structurally present today. Not aspirational. Present.

2. **Identify Weakest Level**
   Categorize your last 20 production bugs by constraint level. Where do they cluster? That is your weakest level.

3. **Add One Constraint**
   Not five. One. If bugs cluster at L4: introduce Value Objects. If L3: enforce one boundary. If L2: separate integration from operation.

4

Observe and Repeat

Each constraint targets a recurring defect class. Track which classes shrink. The defect surface reduces by category, not linearly.

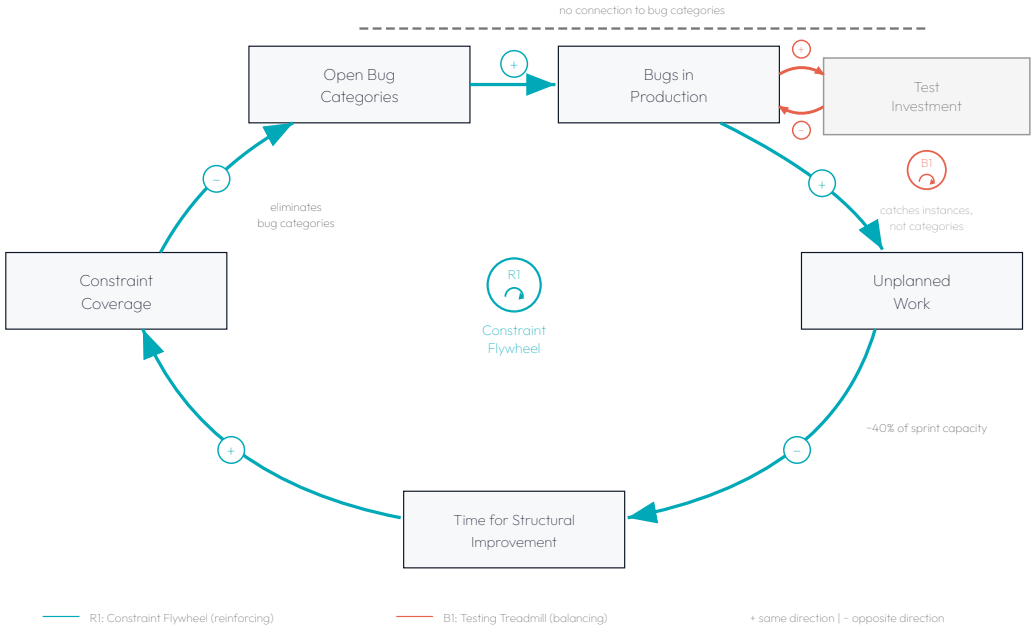

Fig. 4. Causal Loop Diagram. R1 (teal): the Constraint Flywheel. More coverage eliminates bug categories, freeing time for further structural work. B1 (coral): the Testing Treadmill. Testing catches instances but never closes the category gap. Testing reduces individual bug counts (Level 0), but the arrow does not reach Open Bug Categories — that requires structural constraints at Levels 1-5.

8. From Theory to Practice

The question is not whether your team will encounter bugs. The question is whether those bugs are accidents or recurrent structural patterns.

Most teams operate inside the B1 loop (Fig. 4): more bugs lead to more testing, which catches instances, which temporarily reduces bugs. The loop never closes on the category level. Sprint after sprint, the same categories return.

CDD offers a different loop. The Constraint Flywheel (R1) starts with one structural intervention: identify your weakest constraint level, add one constraint, and observe whether the defect class shrinks. Each constraint added frees capacity previously consumed by unplanned work. That freed capacity enables the next structural improvement. The flywheel accelerates.

The Constraint Stack is not a methodology to adopt wholesale. It is a diagnostic lens. Map your current constraints. Find the gap. Close it. The model predicts that recurring defect patterns often cluster around the weakest structural level. The six levels are fixed: Verification, Code, Design, Architecture, Domain, System. A counterexample would be a substantial recurring software defect class within scope that cannot be mapped to any of these levels and cannot be materially reduced by structural intervention. If such cases accumulate, the model needs revision. Start with one question: Where do our defects cluster? The answer points to the level. The level points to the constraint. The constraint targets the recurrence mechanism.

Not only fewer bugs, but fewer recurring classes of avoidable defects. That is the shift.

This paper is part of a trilogy. CDD addresses how to enforce quality structurally. The ACE Model [18] addresses who develops software well (the Analytical, Creative, and Emotional dimensions). The Economics of Changeability [19] addresses what to optimize for. Fig. 5 shows the connections.

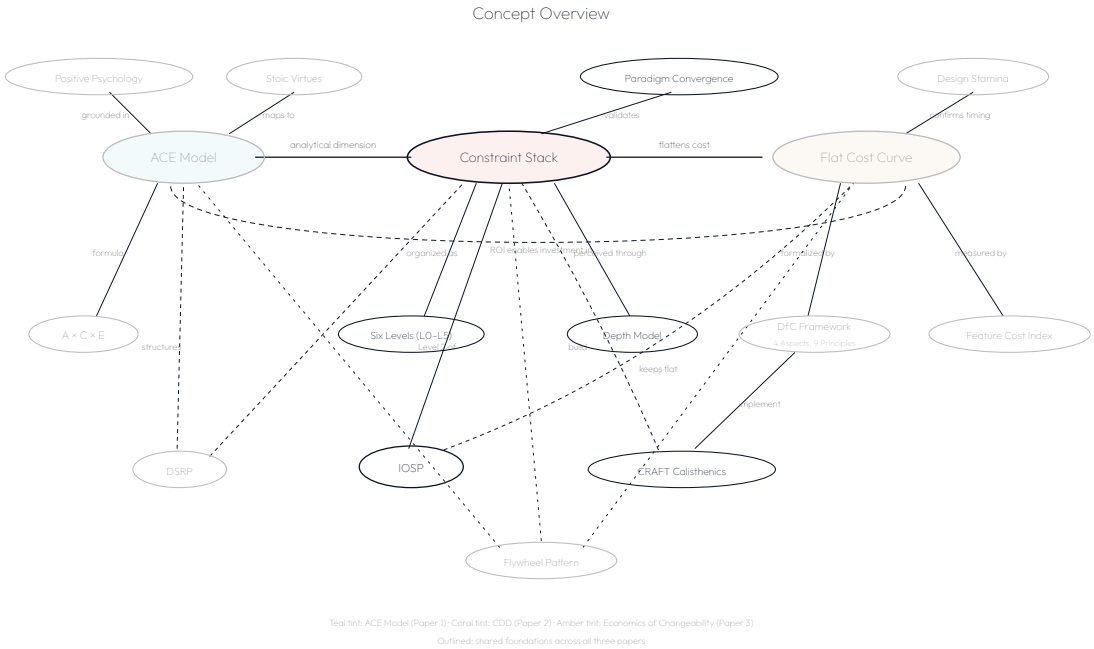

Fig. 5. Concept overview of the trilogy. Teal: ACE Model. Coral: Constraint-Driven Development (this paper). Amber: Economics of Changeability. Navy: shared foundations.

Appendix: Bug Taxonomy Mapping

The following taxonomy maps each constraint level to a representative bug category, a concrete example, and the structural fix. This codebook enables independent raters to classify production bugs by constraint level for diagnostic purposes.

LEVEL	CATEGORY	REPRESENTATIVE BUG	STRUCTURAL FIX
L0	Verification	Regression: “A broke B.”	Automated tests
L1	Code	Comprehension: “I used the wrong variable.”	Small functions, intention-revealing names
L2	Design	Mutation: “List was changed unexpectedly.”	Immutability, IOSP separation
L3	Architecture	Side-effect: “DB change broke the UI.”	Pure core / impure shell
L4	Domain	Invariant: “Order created with zero items.”	Value Objects, Aggregates
L5	System	Semantic: “Timezone mismatch across APIs.”	Bounded Contexts, domain events

Usage: Take your last 20 production bugs. For each bug, identify the constraint level where the structural fix belongs. The level with the most bugs is your weakest constraint level. Target that level first.

Appendix: Methods

Data Collection

The 187-codebase dataset is described in the Methods Note in Section 5.

Instruments

Bug Taxonomy: The six-level classification (L0–L5) above serves as the coding guide. Two raters independently classify each bug. Disagreements are resolved by discussion, not averaging. A minimum sample of 20 recent production defects per project (or the last 6 months, whichever is smaller) is recommended.

CRAFT Calisthenics Rubric: Codebases are scored on structural properties (function size, IOSP compliance, type system usage, module boundaries, domain modeling) derived from the six CRAFT principles (P1 Communication Flow, P2 Cohesive Units, P3 IOSP, P4 Reveals Intention, P5 Dimensions, D1 Software Aesthetics). The rubric produces an ordinal score per constraint level.

Falsifiability

The model makes one central claim: many recurring software defect classes can be mapped to structural weaknesses and reduced through explicit constraints. The claim is strongest for implementation, design, architectural, and domain-representation defects. A strong counterexample would be a substantial recurring defect class (e.g., accounting for more than 10% of observed defects in a mature codebase) within that scope that (a) cannot be mapped to any of the six levels, and (b) cannot be reduced in frequency across subsequent development cycles despite targeted structural intervention.

Limitations

- Convenience sampling introduces selection bias. Codebases from coaching engagements may not represent the general population of software projects.
- The “single-concept-per-object” correlation (Section 5) does not prove causation. Confounders (team experience, project maturity, management investment) are not controlled.
- The Bug Taxonomy has not been independently validated. Inter-rater reliability metrics for the taxonomy are pending formal measurement.
- “Zero bugs” is asymptotic. Requirements bugs, novel failure modes, and human error will always exist.

References

[1] Martin, R.C.: Three Paradigms. The Clean Coder Blog (2012)
[2] Farley, D.: Modern Software Engineering. Addison-Wesley (2022)
[3] Feathers, M.: OO vs FP (Nov 2010). Quoted in multiple sources incl. Martin, R.C.: Functional Programming (Clean Coder Blog, 2014)
[4] Evans, E.: Domain-Driven Design. Addison-Wesley (2003)
[5] Vernon, V.: Effective Aggregate Design. DDD Community (2011)
[6] Seemann, M.: Functional Core, Imperative Shell. blog.ploeh.dk (2020-01-06). See also: Bernhardt, G.: Boundaries. Talk at SCNA (2012)
[7] Ray, B. et al.: A Large Scale Study of Programming Languages and Code Quality in GitHub. In: FSE 2014
[8] Berger, E. et al.: On the Impact of Programming Languages on Code Quality. Reproduction study (2019)
[9] Conway, M.: How Do Committees Invent? In: Datamation (1968)
[10] Bay, J.: Object Calisthenics. In: The ThoughtWorks Anthology (2008)
[11] Westphal, R.: Integration Operation Segregation Principle. ralfwestphal.substack.com
[12] Arlitt, R., Dunz, T., Gahm, H., Majer, D., Westenberger, E.: Besseres ABAP: Schnell, sicher, robust. SAP PRESS / Rheinwerk (2015). ISBN 978-3-8362-2939-5
[13] Pugh, K.: Prefactoring: Extreme Abstraction, Extreme Separation, Extreme Reliability. O'Reilly (2005). Jolt Award 2006
[14] Balzert, H.: Lehrbuch der Softwaretechnik. Spektrum Akademischer Verlag (1998). Analytical vs. constructive quality management
[15] Deming, W.E.: Out of the Crisis. MIT Center for Advanced Engineering Study (1986); MIT Press (2000). "Build quality in"
[16] Leveson, N.: Engineering a Safer World. MIT Press (2011). STAMP: constraints on component interactions
[17] GitClear: AI Copilot Code Quality — Evaluating the 2024 Increased Defect Rate with Data. gitclear.com (2025). 211M lines analysed; 4x growth in code clones
[18] Majer, D., Drumm, C.: The ACE Model. ASE Academy White Paper (2026)
[19] Majer, D.: The Economics of Changeability. ASE Academy White Paper (2026)
[20] Forsgren, N., Kalliamvakou, E., Noda, A. et al.: DevEx in Action: A Study of Its Tangible Impacts. ACM Queue 21(6), 2024; also in CACM 67(1), 2024. n=219, PLS-SEM
[21] DORA: State of AI-assisted Software Development. Google Cloud / dora.dev (2025); ROI of AI-Assisted Software Development (2026). "AI as an amplifier"; returns from the organizational system
[22] Borg, M., Hagatulah, N., Tornhill, A., Söderberg, E.: Code for Machines, Not Just Humans — Quantifying AI-Friendliness with Code Health Metrics. arXiv:2601.02200; ACM (2026). 5,000 programs; AI defect risk markedly higher in unhealthy code

About the Author

Damir Majer coaches development teams on software quality at ASE Academy in Munich. His ACE Model (the Analytical, Creative, and Emotional dimensions) provides the broader coaching framework. CDD addresses the Analytical dimension. After working with teams across industries, one pattern became clear: the ones with persistent quality problems had incomplete constraint coverage. The Constraint Stack emerged from that observation.

Contact: d.majer@majcon.de | ase.academy