Don’t Go Too Fast

A software development philosophy that maintains quality under pressure

Bakmeon Kim

Third Edition: 2026

Table of Contents

DON’T GO TOO FAST

A software development philosophy that maintains quality under pressure

ROD • TFD • DGTF

Integrating Kahneman, Meadows, and Altshuller

Bakmeon Kim

Third Edition

Don’t Go Too Fast

A software development philosophy that maintains quality under pressure

Third Edition: 2026

Theory Foundation:

- Daniel Kahneman: Thinking, Fast and Slow (2011)
- Donella H. Meadows: Thinking in Systems (2008)
- Genrich Altshuller: TRIZ (1946-1998)

Preface: Why I Wrote This Book

The Pattern I Kept Seeing

For over 25 years, I've been building software.

I started as a junior developer at the turn of the millennium, worked my way up to lead research institutes, and have guided countless projects across manufacturing, AI, smart factories, and security systems. I've worked with teams in Korea, Taiwan, Japan, and the United States. I've seen what works and what doesn't.

And through all of it, I kept seeing the same pattern.

Good developers making bad decisions under pressure.

Not because they lacked skill. Not because they didn't know better. But because when the deadline loomed, when the requirements changed at the last minute, when the manager asked "when will it be done?"—something happened. Their careful thinking gave way to rushed reactions. Their systematic approach collapsed into quick fixes.

I watched talented engineers reach for global variables when they knew better. I saw well-designed systems become tangled messes in the final weeks before delivery. I witnessed the same mistakes repeated across different companies, different technologies, different decades.

And I realized: this wasn't a skill problem. It was a *thinking* problem.

The Question That Haunted Me

In 1996, after returning from military service, I took a course on software engineering. Unlike database or networking courses that focused on specific technologies, software engineering asked deeper questions:

What makes software good? How do we build systems that last?

These questions became my obsession.

"What does it mean to build good software?" has been my lifelong inquiry. Not just software that works—any programmer can make software that works. But software that remains good when requirements change. Software that welcomes new features instead of fighting them. Software that survives contact with reality.

I completed my master's degree in software engineering. I led research institutes for over 15 years. I conducted software engineering studies with my teams for over a decade. I kept asking the question.

And slowly, through countless projects and failures and successes, I began to find answers.

The Three Discoveries

The first breakthrough came when I encountered Daniel Kahneman's work on how humans think.

Kahneman, a Nobel laureate, revealed that we have two modes of thinking: System 1 (fast, automatic, error-prone) and System 2 (slow, deliberate, accurate). Under pressure, System 1 dominates. This explained everything I had observed. When deadlines pressed, developers weren't making *conscious* decisions to use global variables—their fast-thinking System 1 was grabbing the first solution that came to mind.

The second breakthrough came from Donella Meadows and systems thinking.

Meadows taught me to see the whole, not just the parts. She showed me that the most powerful intervention point in any system is at the design level—before implementation begins. A complete service chain designed upfront could prevent chaos during implementation. "More is better than missing" became a core principle.

The third breakthrough came from Genrich Altshuller and TRIZ.

Altshuller, who analyzed millions of patents, discovered that most problems involve contradictions that people try to compromise around. "Fast development" versus "high quality" seems like a tradeoff. But TRIZ taught me: don't compromise—resolve the contradiction. By separating design time (slow and careful) from implementation time (fast but guided), we could achieve both speed and quality.

From Theory to Practice

Understanding these theories was one thing. Applying them was another.

Over the years, working across industries—SCM systems, AI platforms, smart factories, AR/VR solutions, security applications—I developed three practical methodologies:

ROD (Responsibility-Oriented Design): Design complete service chains before implementation, eliminating the "missing pieces" that trigger System 1 panic responses.

TFD (Test-First Development): Treat requirements as tests, creating feedback loops that catch errors early and guide implementation.

DGTF (Don't Go Too Fast): Maintain System 2 thinking even under pressure, recognizing triggers and pausing before reacting.

These methodologies worked. Teams that adopted them produced better software with fewer bugs. Projects became predictable. The frantic last-minute scrambles disappeared. Engineers went home on time.

I taught these methods to my teams. I conducted studies. I refined the approaches based on real-world feedback. But I kept them internal, shared only with colleagues.

Until now.

The Proof I Lived It

Some people ask: “Don’t Go Too Fast? Is that even possible in the real world?”

My answer is my career.

I worked at one company for 19 years out of my 25-year career. I served as research institute director for over 15 years.

In an era where developers change jobs every 2-3 years, I chose depth over breadth. I stayed and built. I refined and improved. I watched solutions evolve over decades, not quarters.

Why?

Because going properly beats going fast. Because depth beats breadth. Because the tortoise, in the end, beats the hare.

This book is the result of those 25 years. Not theory from an ivory tower, but wisdom forged in the daily reality of shipping software, meeting deadlines, and—most importantly—maintaining quality when everything pushed me to compromise.

When I say “Don’t Go Too Fast,” I’m not offering advice I read somewhere. I’m sharing how I’ve worked for a quarter century.

Why Now?

For years, I was reluctant to publish these ideas.

Perhaps it was laziness. Perhaps it was the feeling that more refinement was needed. Perhaps it was the classic engineer’s hesitation to share work that wasn’t “perfect.”

But recently, something changed.

In 2025, while developing an AI-based security solution, I’ve been collaborating intensively with AI tools. From direct experience, I’ve found that when you use AI well, you can achieve productivity similar to working with 3-4 developers.

But did this make DGTF unnecessary? The opposite.

The more AI accelerates coding, the more important thoughtful design becomes. The faster AI builds things, the better you need to think about what to build. The more powerful the tools become, the more critical the judgment of the person using them.

The methodologies in this book are not threatened by AI—they become more essential.

What This Book Does NOT Solve

Let me be honest:

- It cannot fix toxic organizational culture

- It doesn't work magically when you're exhausted
- It won't turn juniors into instant experts
- Quick reading cannot replace months of deliberate practice

I include this because honest limitations make honest methodology. The "Hard Questions" section in the Practical Guide addresses these limitations and how to cope with them in more detail.

What This Book Offers

This book offers three things:

First, theoretical foundation. You'll understand *why* ROD, TFD, and DGTF work, grounded in cognitive psychology, systems thinking, and innovation theory.

Second, practical methodology. You'll get concrete, actionable techniques for *what* to do—not vague advice like "think better," but checklists, workflows, and patterns you can use tomorrow.

Third, proven results. These aren't laboratory ideas. They're methodologies refined over decades in real projects, tested with real teams and real deadlines.

Who This Book Is For

This book is for:

- **Senior developers** who have burned out before
- **Junior developers** who can code but wonder *why* things work
- **Technical leaders** who need predictable outcomes
- **Teams** who want to build sustainable culture

It's for anyone seeking:

- Quality maintenance under pressure
- Reduced technical debt
- Predictable development schedules
- Sustainable pace without last-minute crises

A Personal Note

Writing this book, I thought a lot about my past self. The junior developer struggling to learn quickly and prove himself. The manager frustrated watching his team burn out and bugs multiply.

If only I had known this earlier.

That's why I'm publishing. To reach across the barrier of awareness. To speak to myself 25 years ago, and to you who are looking for the same thing sooner.

Our industry celebrates “fail fast, fail often.” But that’s not the only way. You can move thoughtfully, build it right the first time, construct rather than destroy.

Going slow is faster. Going thoughtful wins. The tortoise wins.

Let’s begin.

Part 1: Core Concepts and Values

The Problem: What vs How

Have you read Clean Code?
Do you know SOLID principles?
Have you heard of TDD?

Probably "yes".

Then why do you use global variables on Friday afternoon before a deadline?

```
Many developers:
"Clean Code? I know it"
"SOLID? Sure"
"TDD? I've heard of it"

Reality:
Friday afternoon 5 PM
Manager: "When will this be done?"
→ "Let me just use a global variable..."
→ "Hardcoding would be faster..."
→ "Tests? Later..."
```

The problem isn't not knowing "What". The problem is not knowing "How".

- What: Clean Code, SOLID, TDD (you already know)
- How: How to follow them under pressure? (this is what you need)

ROD, TFD, DGTF are the answers to "How".

This methodology is not a new "What". It's the answer to **"Why"**, **"When"**, and **"How"** to do good development.

Theoretical Foundation

This methodology is built on three validated theories. Understanding these theories is essential—not as academic knowledge, but as the foundation for why DGTF++ works.

Daniel Kahneman: Why We Fail Under Pressure

The Nobel Prize-winning insight:

Daniel Kahneman received the 2002 Nobel Prize in Economics for demonstrating that human decision-making is not rational. His research revealed two distinct systems of thinking:

System 1 (Fast Thinking)

```
Characteristics:
- Automatic and unconscious
- Requires no effort
- Always running
- Pattern-matching based
- Emotional

Strengths:
- Instant response
- Energy efficient
- Good for familiar situations
- Keeps us alive (fight or flight)

Weaknesses:
- Vulnerable to cognitive biases
- Jumps to conclusions
- Cannot handle complexity
- Makes mistakes under pressure
- Overconfident
```

System 2 (Slow Thinking)

```
Characteristics:
- Deliberate and conscious
- Requires effort and focus
- Must be activated intentionally
- Logic and reasoning based
- Analytical

Strengths:
- Accurate judgment
- Can handle complexity
- Considers multiple factors
- Long-term thinking
- Self-aware

Weaknesses:
- Slow
- Tiring (depletes mental energy)
- Lazy (avoids activation)
- Shuts down under pressure
- Limited capacity
```

The Critical Discovery:

> System 2 is lazy. It will not activate unless forced to. Under pressure, System 2 shuts down completely, leaving System 1 in full control.

What This Means for Software Development:

Situation	System	Typical Decision
Calm design meeting	System 2	"Let's think about edge cases"
Friday 5 PM, deadline Monday	System 1	"Just hardcode it for now"
Code review, no pressure	System 2	"This violates SRP, let's refactor"
Production is down	System 1	"Just restart the server"
Learning new concept	System 2	"Let me understand this fully"
Third bug this hour	System 1	"Add another if statement"

The Developer's Dilemma:

```
Design Phase:
- Time available
- Low pressure
- System 2 active
- Good decisions possible

Implementation Phase:
- Deadline pressure
- Changing requirements
- System 1 takes over
- "Quick fixes" accumulate

Result:
The same developer who designed a beautiful architecture
writes spaghetti code under pressure.
Not because they don't know better.
Because System 1 doesn't care about "better."
```

How DGTF++ Uses This:

1. **ROD:** Complete the design when System 2 is active. Leave nothing for System 1 to decide.
2. **TFD:** Define tests when System 2 is active. Implementation becomes mechanical—just make tests pass.
3. **DGTF:** Recognize when System 1 is taking over. Force System 2 activation even under pressure.

Donella Meadows: Why We Must See the Whole

The Systems Thinking Pioneer:

Donella Meadows was a systems scientist who wrote “Thinking in Systems,” one of the most influential books on understanding complex systems. Her insights explain why software projects fail even when individual components are well-built.

What is a System?

> A system is a set of interconnected elements organized to achieve a purpose. The whole is greater than the sum of its parts.

Examples:

```
A car is not just parts:
- Engine + Wheels + Steering + Brakes = Transportation
- Remove any part = No transportation
- The PURPOSE emerges from connections

Software is the same:
- UserService + AuthService + Database = Login functionality
- Remove any part = No login
- The BEHAVIOR emerges from connections
```

Core Principles of Systems Thinking:

1. See the Whole First

```
Wrong approach:
"Let me build UserService first, then figure out what else I need"
→ Discover missing pieces during implementation
→ Panic → System 1 → Poor decisions

Right approach (ROD):
"Let me map the complete service chain first"
→ All pieces identified in design
→ Implementation is just building known pieces
```

2. Relationships Matter More Than Parts

```
A well-designed service with bad connections = Bad system
A simple service with good connections = Good system

ROD focuses on:
- How services connect (interfaces)
- What flows between them (data)
- Who depends on whom (dependencies)
```

3. Feedback Loops Determine Behavior

```
Positive feedback (amplifying):
Bug → Stress → Rush → More bugs → More stress → ...
→ System spirals out of control
```

```
Negative feedback (stabilizing):
Code → Test → Fail → Fix → Test → Pass
→ System self-corrects

TFD creates negative feedback loops:
Every change is immediately validated.
Problems are caught before they amplify.
```

4. Leverage Points: Where to Intervene

Meadows identified that some intervention points are far more effective than others:

```
Low leverage (expensive, low impact):
- Fixing bugs in production
- Adding tests after code is written
- Refactoring legacy code

High leverage (cheap, high impact):
- Design phase decisions
- Defining tests before code
- Establishing service boundaries early

ROD + TFD work at the highest leverage point:
The design phase.
```

5. The Danger of "Missing"

```
A system with a missing element doesn't work "mostly."
It behaves unpredictably.

Example:
Payment system without error handling:
- Works 99% of the time
- The 1% failure corrupts data silently
- Discovered months later
- Damage is catastrophic

ROD's "More is better than missing":
Better to design error handling and remove it
than to forget it and discover it in production.
```

The Iceberg Model:

```
What we see:     Events (bugs, delays, failures)
                      ↑
What causes it:  Patterns (recurring problems)
                      ↑
What shapes it:  Structures (architecture, processes)
                      ↑
```

```
What drives it:  Mental Models (how we think)

Most fixes address Events.
DGTF++ addresses Mental Models.
That's why it works at the root.
```

How DGTF++ Uses This:

1. **ROD:** See the whole system (service chain) before building parts. No missing pieces.
2. **TFD:** Create stabilizing feedback loops. Every change is validated immediately.
3. **DGTF:** Work at the highest leverage point—your own thinking patterns.

Genrich Altshuller: Why We Must Resolve, Not Compromise

The Father of TRIZ:

Genrich Altshuller was a Soviet engineer who analyzed over 200,000 patents to discover patterns in innovation. His discovery: **breakthrough solutions come from resolving contradictions, not compromising.**

What is a Contradiction?

> A contradiction exists when improving one aspect worsens another.

The Fundamental Contradiction in Software:

```
"We want FAST development"
    vs
"We want HIGH QUALITY"

Traditional "solutions" (actually compromises):
- Fast → Sacrifice quality (technical debt)
- Quality → Sacrifice speed (miss deadlines)
- Balance → Get neither (mediocre everything)

None of these are solutions.
They're surrenders.
```

TRIZ's Revolutionary Insight:

> Contradictions are not problems to balance. Contradictions are opportunities to innovate.

The 40 Inventive Principles:

Altshuller identified 40 principles that inventors use repeatedly. Three are especially relevant to software:

Principle 1: Segmentation

```
Problem: Monolithic system is hard to change
Compromise: Accept that changes are risky

TRIZ Solution: Divide into independent segments
→ Services with clear boundaries
→ Change one without affecting others
→ ROD's service chain
```

Principle 10: Prior Action

```
Problem: Problems discovered during implementation
Compromise: Accept that debugging is part of development

TRIZ Solution: Do things before they're needed
→ Design completely before implementing
→ Write tests before code
→ ROD + TFD
```

Principle 15: Dynamization (Separation in Time)

```
Problem: Need to be both fast AND careful
Compromise: Be somewhat fast, somewhat careful (neither)

TRIZ Solution: Be different things at different times
→ Design phase: Slow and thorough (100% careful)
→ Implementation phase: Fast execution (100% fast)
→ DGTF++ approach
```

Separation Principles:

TRIZ identifies four ways to resolve contradictions:

```
1. Separation in Time
   "Be fast AND careful" → Be careful now, fast later
   → Design slow, implement fast

2. Separation in Space
   "Be coupled AND independent" → Different boundaries
   → Services are independent, system is coupled

3. Separation in Scale
   "Be simple AND complete" → Different levels
   → Each service simple, whole system complete

4. Separation by Condition
   "Be flexible AND stable" → Different triggers
   → Interface stable, implementation flexible
```

The Ideal Final Result (IFR):

Altshuller asked: "What would the ideal solution look like?"

```
Ideal software development:
- Zero bugs
- Zero rework
- Zero confusion
- Instant completion

How to approach it:
- Bugs come from missing design → Complete design (ROD)
- Rework comes from unclear requirements → Tests as requirements (TFD)
- Confusion comes from System 1 → Control System 1 (DGTF)
- Slow completion comes from fixing mistakes → Prevent mistakes (all three)
```

How DGTF++ Uses This:

1. **ROD:** Prior Action + Segmentation. Design everything before implementing. Divide into independent services.
2. **TFD:** Prior Action. Define success criteria before building.
3. **DGTF:** Separation in Time. Think first (System 2), then act (can be fast).

Why These Three Together

Each theory answers a different question:

```
Kahneman → "Why do we fail under pressure?"
           Answer: System 1 takes over and makes poor decisions.

Meadows  → "What should we prepare to prevent failure?"
           Answer: The complete system, with feedback loops.

TRIZ     → "How do we get speed AND quality?"
           Answer: Resolve the contradiction through separation.
```

The Integration:

```
┌─────────────────────────────────────────────────────────┐
│                    DGTF++ Framework                     │
├─────────────────────────────────────────────────────────┤
│                                                         │
│  Kahneman: WHEN to think                                │
│  ──────────────────────                                 │
│  Design phase = System 2 time                           │
│  Implementation = Protect from System 1                 │
│                                                         │
```

```
Meadows: WHAT to think about
──────────────────────────

See the whole system
Create feedback loops
Work at high leverage points

TRIZ: HOW to resolve contradictions
─────────────────────────────────

Separate thinking and doing in time
Segment into independent services
Apply prior action
```

```
ROD  = Meadows (whole system) + TRIZ (prior action)
TFD  = Meadows (feedback) + TRIZ (prior action)
DGTF = Kahneman (System 2) + TRIZ (time separation)
```

No theory alone is sufficient:

- Kahneman without Meadows: Know WHEN to think, but not WHAT to think about
- Meadows without TRIZ: Know WHAT to prepare, but stuck in compromise
- TRIZ without Kahneman: Know HOW to resolve, but can't execute under pressure

Together, they form a complete system for sustainable software development.

ROD: Responsibility-Oriented Design

Core Principle

> **"More is better than missing"**

In the design phase, build a **complete service chain**.

When you're unsure whether to create a new service or not—**create it**.

Why? Because if the chain is missing and you discover it during implementation, **System 1 will love that situation**. And System 1's love means poor rushed decisions.

The Asymmetry: Removal vs Addition

This principle is based on a fundamental asymmetry:

Removing unnecessary service during implementation:

```
Situation: "This service is never called"
Evidence: Clear (no usage)
```

```
Mental state: Calm (System 2)
Action: Delete it
Risk: Low
Effort: Easy
```

Adding missing service during implementation:

```
Situation: "I need this but it doesn't exist!"
Evidence: Panic discovery
Mental state: Pressure (System 1 dominant)
Action: Quick hack
Risk: High
Effort: Difficult AND dangerous
```

The asymmetry is clear: - Removing = Easy, Safe, System 2 - Adding = Hard, Dangerous, System 1

Therefore: When in doubt during design, include it. You can always remove later.

What is a Service Chain?

A service chain is a complete map of all services and their relationships needed to fulfill a requirement.

```
Example: "User can purchase a product"

Service Chain:
UserService → CartService → PaymentService → OrderService → InventoryService
→ NotificationService

Each arrow = dependency
Each service = clear responsibility
Complete chain = no surprises during implementation
```

The key question: > "Can this requirement be achieved through the service chain alone?"

- YES → Chain is complete
- NO → Something is missing → Add service in design phase (not implementation!)

ROD is Not a New "What"

Let's be honest.

ROD isn't completely new: - Clean Architecture? Similar. - Service-Oriented Design? Yes. - Domain-Driven Design? Overlapping parts exist. - SOLID? Included.

But the question:

You know Clean Architecture. You know SOLID. You know dependency injection.

But why don't you follow them when deadline approaches? Why does it become "just make it work, fix later"?

ROD is not a new technology.
ROD is the answer to **"Why"**, **"When"**, **"How"** to do good design.

```
Clean Architecture:
"Structure it like this"
→ What (what to do)

ROD:
"Why to do it this way" (Kahneman: System 1 prevention)
"When to do it this way" (in design phase, when System 2 is active)
"How to keep it" (Complete service chain, no Missing)
→ Why + When + How
```

The Real Power: Change Isolation

No matter how perfectly you design, changes happen during implementation.

- Requirements change
- Discover a better way
- Need to fix bugs

ROD doesn't prevent change. ROD isolates change.

```
Without ROD (tightly coupled):
Change in A → affects B → affects C → affects D → ...
→ Fear of change
→ "Don't touch it, it works"
→ Technical debt accumulates

With ROD (service chain):
Change in A → A's interface unchanged → B, C, D unaffected
→ Change is safe
→ Confident refactoring
→ Sustainable codebase
```

The service chain creates boundaries. Each service has a clear responsibility. Changes inside a service don't leak outside.

Strict Rules

To maintain service chain integrity:

- ❌ No Constructors (new) inside services
- ❌ No Static fields or methods
- ❌ No Assumptions about implementation
- ✅ Everything expressed as services

- ✅ Dependencies injected, not created

Why these rules? Because: - `new` creates hidden dependencies (breaks chain visibility) - `static` creates global state (breaks isolation) - Assumptions become surprises (System 1 triggers)

When ROD is Complete

Verification criteria:

```
For each requirement, ask:
"Can this be achieved through the service chain alone?"

If YES for all requirements → ROD is complete
If NO for any requirement → Missing exists → Add to design
```

Signs of a complete ROD:

- Every service has exactly one responsibility
- Every dependency is visible in the chain
- No service needs to know another's implementation
- Requirements map directly to service chains
- Implementation is "just following the chain"

TFD: Test-First Development

Core Principle

> **"Requirements = Tests"**

If you can't write a test for it, you don't understand the requirement.

A test is not verification after the fact. A test is the **precise definition** of what the requirement means.

Why Requirements Must Be Tests

The problem with natural language requirements:

```
Requirement: "Users should be able to log in"

Questions this doesn't answer:
- What happens with wrong password?
- How many attempts before lockout?
- What does "logged in" mean? Token? Session? Cookie?
- What about expired accounts?
- What about unverified emails?
```

The same requirement as tests:

```
test_login_success:
  Given: valid email, correct password, verified account
  When: login attempted
  Then: returns JWT token, valid for 24 hours

test_login_wrong_password:
  Given: valid email, wrong password
  When: login attempted
  Then: returns 401, message "Invalid credentials"

test_login_account_locked:
  Given: 3 consecutive failed attempts
  When: 4th attempt with correct password
  Then: returns 403, message "Account locked for 30 minutes"

test_login_unverified_email:
  Given: valid credentials, email not verified
  When: login attempted
  Then: returns 403, message "Please verify your email"
```

Now the requirement is precise. No ambiguity. No surprises during implementation.

Tests as Completion Criteria

Without tests, "done" is subjective:

```
Developer: "It's done"
QA: "It doesn't handle this case"
Developer: "That wasn't in the requirements"
QA: "It's obvious"
Developer: "Not to me"
→ Conflict, rework, frustration
```

With TFD, "done" is objective:

```
All tests pass → Done
Any test fails → Not done
New case discovered → Add test first, then implement
```

Tests are the contract between requirement and implementation.

TFD and ROD Combined

TFD becomes powerful when combined with ROD:

```
ROD provides: Complete service chain
TFD provides: Tests for each service

Together:
- Each service in the chain has tests
```

```
- Tests define the service's contract
- Implementation just fulfills the contract
```

Example:

```
ROD Service Chain:
UserService → AuthService → TokenService

TFD Tests:
UserService:
  - test_find_by_email_exists
  - test_find_by_email_not_found

AuthService:
  - test_validate_password_correct
  - test_validate_password_wrong
  - test_check_account_status_active
  - test_check_account_status_locked

TokenService:
  - test_generate_token_valid_user
  - test_token_contains_required_claims
  - test_token_expires_in_24_hours
```

When you implement, you're just making tests pass. No guessing. No ambiguity. No System 1 decisions.

Relationship with TDD

TFD does not replace TDD. TFD makes TDD easier.

The common struggle with TDD:

```
TDD says: "Write test first"
Developer thinks: "Test for what? I don't know what to build yet"
→ Writes code first
→ TDD abandoned
```

TFD solves this:

```
ROD: Define service chain (what services exist)
TFD: Define tests for each service (what each service does)
TDD: Red → Green → Refactor (how to implement)
```

The progression:

```
Design Phase (System 2):
  1. ROD: Build service chain
  2. TFD: Write test cases for each service
```

```
Implementation Phase (DGTF protects System 2):
  3. TDD: For each test, Red → Green → Refactor
```

TFD answers "test for what?" before TDD begins.

Tests Provide Immediate Feedback

From Meadows' Systems Thinking: > "The longer the feedback loop, the harder it is to learn."

Without tests (long feedback loop):

```
Write code → Deploy → User reports bug → Debug → Find cause → Fix
Time: Days to weeks
Learning: Difficult, context lost
```

With tests (short feedback loop):

```
Write code → Run test → Fail → Fix → Pass
Time: Seconds to minutes
Learning: Immediate, context fresh
```

Tests are the feedback mechanism that keeps you on track.

When TFD is Complete

For each service in ROD:

- Every public method has tests
- Every edge case is covered
- Every error condition is tested
- Tests are readable as requirements

Signs of complete TFD:

- A new team member can understand what a service does by reading its tests
- "Happy path" and "sad paths" are both tested
- Tests don't depend on implementation details
- Tests serve as living documentation

DGTF: Don't Go Too Fast

Core Principle

> **"Slow is smooth, smooth is fast"**

Thoughtfulness is not slow—it's smooth. And smooth is ultimately fast, because there's no rework.

Prerequisite: The "Wow" Moment

DGTF has a prerequisite.

DGTF's first step is "Recognize"—recognizing that System 1 is activating. But here's the paradox:

> When System 1 is fully active, you don't know you're in System 1.

Who can use DGTF?

People who experience the "Wow" moment:

```
"Wait... is this right?"
"Hmm, something feels off..."
"Hold on, let me think about this..."
```

This brief moment of doubt—this is the "Wow".

The Critical Insight:

```
Without Wow → 100% System 1 → Cannot recognize → Cannot apply DGTF
With Wow but no DGTF → Door opens briefly → Don't know what to do → Door clos
es
With Wow AND DGTF → Door opens → Know exactly what to do → System 2 activated
```

This is why DGTF must be learned BEFORE Wow happens.

When Wow comes, you need to know what to do immediately.

The LA Analogy

A manager tells the team: "Go to LA. Fast."

```
Person D: Starts running toward LA immediately.
          "He said fast! I must go now!"
          → Most effort, slowest result

Person C: Grabs a bicycle.
          "At least I'm doing something..."
          → Compromise, still slow

Person A: Goes home to get his car.
          "Let me get the right tool first."
          → Seems like going backward, but faster

Person B: Searches for airplane schedules.
          "What's the fastest way?"
          → Seems like doing nothing, but fastest
```

From the outside: - D looks the most hardworking (running!) - A looks like he's going the wrong direction - B looks like he's just sitting there

But the result is the opposite.

This is DGTF.

When someone says "Fast!": - System 1 (D): "Yes!" → Start running - DGTF (A/B): "Wait, what's the fastest way to get there?"

The person who pauses to think beats the person who rushes to fail.

Why It Works

DGTF controls System 1 and activates System 2:

```
1. Recognize: "Am I rushing right now?"
2. Pause: "Wait, let me think"
3. Check: ROD design, TFD tests, impact analysis
4. Plan: Clear steps
5. Execute: Thoughtfully, with verification
```

Even in real emergencies, DGTF applies. The difference is 5 seconds of pause instead of 5 minutes. But it's still: Stop → Think → Act.

DGTF ≠ Slow, DGTF ≠ Willpower

Common Misconceptions:

```
❌ DGTF = Work slowly
❌ DGTF = Requires willpower to resist rushing
❌ DGTF = Enduring the urge to go fast
```

Truth:

```
✅ DGTF = Work thoughtfully
✅ DGTF = More effective, not more effortful
✅ DGTF = Saves energy by avoiding rework
```

DGTF does not consume willpower. It saves energy.

```
Without DGTF:
Rush → Bug → Debug → Fix → New bug → More debug → Exhaustion

With DGTF:
Think → Implement correctly → Done → Energy saved
```

The Agile Analogy:

Some teams say: "This project is easy, let's do Agile." This is backwards. Agile is MORE effective for HARD projects.

DGTF is the same.

❌ "I have time, so I'll use DGTF"
✅ "I'm under pressure, so I NEED DGTF"

The harder the situation, the more DGTF helps.

What DGTF Does NOT Solve

DGTF is not a solution for everything.

DGTF does NOT fix:

1. **Toxic Organizations**
   - If "slow and careful" gets you fired
   - If unreasonable deadlines are the norm → This is an environment problem, not a DGTF problem.
2. **Irrational Managers**
   - If careful work is not recognized
   - If only "looking busy" matters → DGTF cannot change other people's attitudes.
3. **Fundamentally Broken Systems**
   - Legacy code that needs complete rewrite → DGTF is for prevention, not repair.

The Answer:

```
If the environment doesn't allow DGTF:
Option 1: Leave that environment.
Option 2: Apply DGTF within the scope you can control.
```

Option 2 is key:

DGTF is an internal process. - Taking 3 seconds to think before typing—nobody notices. - Checking the design before coding—invisible to others. - Pausing when you feel "hurry"—only you know.

From the outside, you look the same. Only the results are better.

How They Work Together

```
┌──────────────────────────────────────┐
│ Design Phase: ROD                    │
│                                      │
│ • System 2 active                    │
│ • Build complete service chain       │
│ • When in doubt, include it          │
│ • Eliminate "Missing"                │
```

```
Result: "What to build" is clear
```

↓

```
Test Design Phase: TFD

• System 2 still active
• Define tests for each service
• Tests = precise requirements
• Clarify completion criteria

Result: "How to verify" is clear
```

↓

```
Implementation Phase: DGTF

• Pressure increases
• DGTF keeps System 2 active
• Follow ROD design
• Verify with TFD tests

Result: "How to build" is safe
```

Synergy:

- ROD alone: Complete design, but no verification mechanism
- TFD alone: Verification exists, but design may be incomplete
- DGTF alone: Thoughtful progress, but no direction or verification
- **ROD + TFD + DGTF:** Complete design + Precise verification + Thoughtful execution = Predictable, sustainable, high-quality software

---

The Essence: Habit, Not Knowledge

ROD, TFD, DGTF are:

❌ Not Principles
❌ Not Rules
❌ Not Processes
❌ Not Knowledge

✅ A Way of Thinking

✅ Habits
✅ Training

Understanding ≠ Doing

```
Understanding:
- Read this document → "I get it"
- Nod along → "Makes sense"
- Agree with everything → "I'll do this"
→ This is System 1 saying "Got it, next!"

Doing:
- Apply tomorrow → Struggle
- Fail → Learn
- Try again → Slightly better
- Repeat → Eventually natural
→ This is building habit through training
```

You can understand DGTF in 10 minutes. You cannot DO DGTF without months of practice.

Training, Not Learning

```
Learning: Acquire knowledge (one-time)
Training: Build habit (continuous)

You don't "learn" to drive well.
You "train" to drive well.

You don't "learn" DGTF.
You "train" DGTF.
```

Crossing the Hurdle is Not the End

❌ "I applied DGTF once successfully!"
→ Finished? No. Tomorrow the pressure returns.

✅ "I applied DGTF today. I'll do it again tomorrow."
→ And the day after. And the week after.
→ Until it's no longer conscious effort.

The Driving Analogy

Basic driving principles: - Green light means go - Brake pedal stops the car - Accelerator moves forward

Does knowing these principles make you a good driver?

No.

Good drivers have: - Continuous attention - Situational judgment - Consideration for others - These as **habits** ingrained in them

Software is the same.

Knowing SOLID, Clean Code, TDD doesn't make you a good developer. Being able to practice these as **habits** makes you a good developer.

The difference:

```
Principles/Rules:
- "You should do this"
- Externally enforced
- Collapse under pressure

Way of Thinking/Habits:
- "I think this way"
- Comes from within
- Persists under pressure
```

The Hardest Part

The hardest thing about DGTF is not understanding it. The hardest thing is: - Keeping curiosity alive - Maintaining doubt ("Is this really right?") - Practicing continuously - Not stopping after the first success

This requires not technique, but attitude.

Core Values

1. Solid Theoretical Foundation

- Kahneman (Nobel Laureate): How humans think
- Meadows (Systems Authority): How systems work
- Altshuller (TRIZ Founder): How to solve problems

Theory + Practice = Reliable methodology

2. Understands Humans

We acknowledge human weaknesses: - We make mistakes under pressure (Kahneman) - We miss the whole when seeing parts (Meadows) - We're accustomed to compromise (Altshuller)

And compensate with systems: - ROD: Be complete in design phase - TFD: Verify with feedback - DGTF: Force thoughtfulness

3. Universally Applicable

- Language independent (Go, Java, Python, etc.)
- Domain independent (Web, Mobile, Backend, etc.)
- Team size independent (Solo to large teams)

4. Complementary

- Each can be applied independently
- Synergy when applied together
- Gradual adoption possible

5. Sustainability

- Individual: Prevent burnout, improve expertise
- Team: Consistent productivity, high morale
- Business: Predictable deployment, competitive advantage

Getting Started

Today: 1. ROD: When designing next feature, write a service chain first 2. TFD: Before implementing, define test cases 3. DGTF: When you feel "hurry," pause for 3 seconds

This Week: - Apply to one small feature - Observe what happens - Note what's difficult

This Month: - Apply to multiple features - Share with a teammate - Read the Practical Guide for detailed methods

Remember: > "Good programmers come from correct thinking, not fast typing"

> "Quality is built in the process, not in inspection"
>
> "Sustainable development starts from attitude, not methodology"

For detailed implementation guidance, examples, checklists, and exercises, see the Practical Guide.

Part 2: Practical Guide

How to Use This Guide

Prerequisites: - Read “DGTF++ Core Concepts” first - Understand WHY before learning HOW

This guide provides: - Practical rules and methods - Step-by-step workflows - Real examples with code - Checklists for verification - Common pitfalls and solutions

Structure: 1. Quick Reference (Theory summary) 2. ROD in Practice 3. TFD in Practice 4. DGTF in Practice 5. Complete Example 6. For Junior Developers 7. Measuring Success 8. FAQ & Hard Questions 9. Execution Plan

---

Quick Reference: Theory Summary

For detailed explanations, see “Core Concepts” document.

Kahneman (When to Think)

```
System 1: Fast, automatic, error-prone, dominant under pressure
System 2: Slow, deliberate, accurate, shuts down under pressure

→ Design with System 2 (calm)
→ Protect System 2 during implementation (DGTF)
```

Meadows (What to Think About)

```
- See the whole system before parts
- Missing elements cause unpredictable behavior
- Feedback loops enable self-correction
- Design phase = highest leverage point

→ Build complete service chain (ROD)
→ Create feedback with tests (TFD)
```

TRIZ (How to Resolve Contradictions)

```
"Fast" vs "Quality" is not a trade-off.
Resolve by separation in time:
- Design phase: 100% careful
- Implementation phase: 100% fast (following the plan)

→ Prior action: Do everything before it's needed
→ Segmentation: Divide into independent parts
```

Part 1: ROD in Practice

The Rules

Rule 1: No Constructors Inside Services

```go
// ❌ BAD: Hidden dependency
type OrderService struct{}

func (s *OrderService) CreateOrder(userId string) (*Order, error) {
    user := new(User)  // Where does User come from?
    user.ID = userId   // What data does User have?
    // ... hidden complexity
}

// ☑ GOOD: Explicit dependency
type OrderService struct {
    userFinder UserFinder
}

func (s *OrderService) CreateOrder(userId string) (*Order, error) {
    user, err := s.userFinder.FindByID(userId)
    if err != nil {
        return nil, err
    }
    // ... clear flow
}
```

Why this rule? - `new()` hides dependencies → Missing not discovered - `new()` is System 1's escape route → "Just create it!" - Explicit dependencies → Complete service chain

---

Rule 2: No Static Fields or Methods

```go
// ❌ BAD: Global state
var globalConfig *Config  // Who sets this? When?

func GetUser(id string) *User {
    db := globalDB  // Hidden dependency
    // ...
}

// ☑ GOOD: Injected dependencies
type UserFinder struct {
    db     Database
    config Config
}
```

```go
func (f *UserFinder) FindByID(id string) (*User, error) {
    // All dependencies visible
}
```

Why this rule? - Static = global state = hidden connections - Global state breaks isolation - Changes in global state affect everything unpredictably

---

Rule 3: No Implementation Assumptions

```go
// ❌ BAD: Assuming implementation
type PaymentService struct {
    // Assumes Stripe is always used
}

func (s *PaymentService) Process(amount int) {
    stripe.Charge(amount)  // What if we change to PayPal?
}

// ☑ GOOD: Interface-based
type PaymentGateway interface {
    Charge(amount int) error
}

type PaymentService struct {
    gateway PaymentGateway  // Could be Stripe, PayPal, anything
}

func (s *PaymentService) Process(amount int) error {
    return s.gateway.Charge(amount)
}
```

Why this rule? - Assumptions become surprises - Surprises trigger System 1 - Interfaces allow safe changes

---

How to Build a Service Chain

Step 1: Start with the Requirement

```
Requirement: "User can log in with email and password"
```

Step 2: Ask “What Must Happen?”

```
1. Validate input format
2. Find user by email
3. Verify password
4. Check account status
```

```
5. Create session
6. Return token
```

Step 3: Convert Each Step to a Service

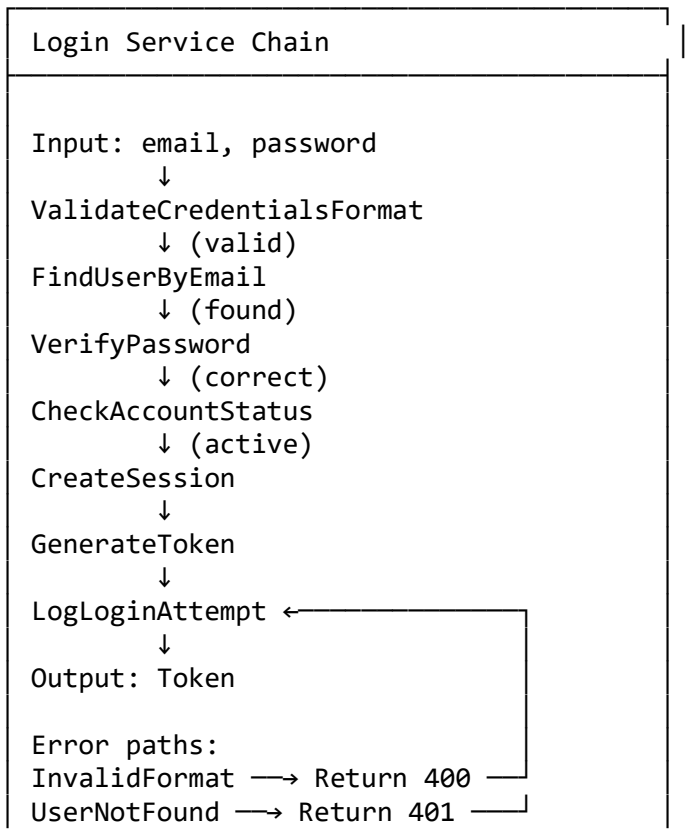
```
ValidateCredentialsFormat(email, password) → ValidationResult
FindUserByEmail(email) → User
VerifyPassword(user, password) → bool
CheckAccountStatus(user) → AccountStatus
CreateSession(user) → Session
GenerateToken(session) → Token
```

Step 4: Add Missing Elements

Ask for each service: - “Where does the input come from?” → Add service if unclear - “Where does the output go?” → Add service if unclear - “What if it fails?” → Add error handling service - “Who needs to know?” → Add notification/logging service

```
Added:
LogLoginAttempt(email, success, timestamp) → void
HandleFailedLogin(email, attemptCount) → void
NotifySecurityTeam(suspiciousActivity) → void
```

Step 5: Draw the Complete Chain



```
WrongPassword → HandleFailedLogin ─┘
AccountLocked → Return 403 ────┘
```

Step 6: Verify Completeness

For each path through the chain: - Can it complete without missing information? → YES = Complete - Does any service need something not in the chain? → Add it

---

Example: Payment System

Requirement: "User can purchase items in cart"

Service Chain:

```
Payment Service Chain

1. Input Validation
   ValidatePaymentRequest
   FindUserAccount
   CheckPaymentEligibility
        ↓
2. Calculation
   GetCartItems
   CalculateTotalAmount
   ApplyDiscounts
   CalculateTax
        ↓
3. Inventory
   CheckInventoryAvailability
   ReserveInventory
        ↓
4. Payment Processing
   ValidatePaymentMethod
   CreatePaymentTransaction
   CallPaymentGateway
   ValidatePaymentResponse
        ↓
5. Order Creation
   CreateOrder
   ConfirmInventoryReservation
   UpdateUserPurchaseHistory
        ↓
6. Notification
   SendOrderConfirmation
```

```
   SendReceipt
        ↓
7. Logging
   LogPaymentEvent

Error Handling:
   ReleaseInventoryReservation
   RefundPayment
   NotifyPaymentFailure
```

Key Design Decisions:

1. **Reservation before Payment:** If payment fails, release reservation
2. **Confirmation after Payment:** Only confirm when payment succeeds
3. **Separate Logging:** Always log, regardless of success/failure

---

ROD Checklist

```
Design Phase:
□ All requirements mapped to service chains?
□ Each service has single responsibility?
□ All dependencies explicit (no new/static)?
□ All interfaces defined?
□ Error paths included?

Verification:
□ Can each requirement complete through chain alone?
□ No service needs unlisted dependency?
□ No assumptions about implementation?

Signs of Completion:
□ Implementation feels like "just following the chain"
□ No "where does this come from?" questions
□ No "I need to add something" during coding
```

Part 2: TFD in Practice

From Requirements to Tests

The Transformation:

```
Natural Language          →    Test Cases
──────────────────────         ──────────

"User can log in"         →    test_login_success
                               test_login_wrong_password
```

```
                                test_login_user_not_found
                                test_login_account_locked
                                test_login_too_many_attempts
```

Step 1: Identify Happy Path

```
Requirement: "User can log in"

Happy Path:
- Given: valid email, correct password, active account
- When: login attempted
- Then: returns valid token
```

Step 2: Identify Failure Cases

Ask: "What can go wrong?"

```
- Invalid email format
- Email not registered
- Wrong password
- Account locked
- Account not verified
- Too many failed attempts
- System error (DB down)
```

Step 3: Identify Edge Cases

Ask: "What about boundaries?"

```
- Empty email
- Empty password
- Very long email (1000 chars)
- Password with special characters
- Unicode in email
- SQL injection attempt
- Concurrent login attempts
```

Step 4: Write Test Specifications

```go
func TestLoginService(t *testing.T) {
    // Happy Path
    t.Run("ValidCredentials_ReturnsToken", func(t *testing.T) {
        // Given: valid email, correct password, active account
        // When: Login(email, password)
        // Then: returns token, no error
    })

    // Failure Cases
    t.Run("InvalidEmailFormat_ReturnsError", func(t *testing.T) {
        // Given: malformed email
        // When: Login(email, password)
```

```
        // Then: returns validation error
    })

    t.Run("WrongPassword_ReturnsError", func(t *testing.T) {
        // Given: valid email, wrong password
        // When: Login(email, password)
        // Then: returns auth error, increments failure count
    })

    t.Run("AccountLocked_ReturnsError", func(t *testing.T) {
        // Given: valid credentials, locked account
        // When: Login(email, password)
        // Then: returns locked error
    })

    t.Run("ThreeFailedAttempts_LocksAccount", func(t *testing.T) {
        // Given: valid email, wrong password
        // When: Login fails 3 times
        // Then: account becomes locked
    })

    // Edge Cases
    t.Run("EmptyEmail_ReturnsError", func(t *testing.T) {
        // Given: empty email
        // When: Login("", password)
        // Then: returns validation error
    })

    t.Run("VeryLongEmail_HandledCorrectly", func(t *testing.T) {
        // Given: 1000 character email
        // When: Login(longEmail, password)
        // Then: returns validation error (max length exceeded)
    })
}
```

Test Structure: Arrange-Act-Assert

```
func TestVerifyPassword_CorrectPassword_ReturnsTrue(t *testing.T) {
    // Arrange: Set up preconditions
    hasher := NewBcryptHasher()
    storedHash := hasher.Hash("correctPassword123")
    service := NewPasswordVerifier(hasher)

    // Act: Execute the behavior
    result, err := service.Verify("correctPassword123", storedHash)

    // Assert: Check the outcome
```

```
    if err != nil {
        t.Fatalf("Expected no error, got %v", err)
    }
    if !result {
        t.Error("Expected true, got false")
    }
}
```

Testing Each Service in the Chain

ROD Chain → TFD Tests

```
Service Chain:                    Tests:
──────────────                    ──────
ValidateCredentialsFormat    →    test_valid_format
                                  test_empty_email
                                  test_invalid_email
                                  test_empty_password
                                  test_password_too_short

FindUserByEmail              →    test_user_found
                                  test_user_not_found
                                  test_db_error

VerifyPassword               →    test_correct_password
                                  test_wrong_password
                                  test_hash_error

CheckAccountStatus           →    test_active_account
                                  test_locked_account
                                  test_unverified_account
                                  test_deleted_account

CreateSession                →    test_session_created
                                  test_session_has_expiry
                                  test_session_stored

GenerateToken                →    test_token_generated
                                  test_token_contains_claims
                                  test_token_valid_signature
```

Integration Test:

```
func TestLoginFlow_EndToEnd(t *testing.T) {
    // Setup complete system
    db := setupTestDB()
    defer db.Close()
```

```
    // Create user
    user := createTestUser(db, "test@example.com", "password123")

    // Test complete flow
    service := NewLoginService(db)
    token, err := service.Login("test@example.com", "password123")

    // Verify
    if err != nil {
        t.Fatalf("Login failed: %v", err)
    }
    if token == "" {
        t.Error("Expected token, got empty string")
    }

    // Verify token is valid
    claims, err := validateToken(token)
    if err != nil {
        t.Fatalf("Token invalid: %v", err)
    }
    if claims.UserID != user.ID {
        t.Errorf("Token user mismatch: expected %s, got %s",
            user.ID, claims.UserID)
    }
}
```

TFD Checklist

```
Test Design:
□ Happy path tested?
□ All failure cases identified and tested?
□ Edge cases covered?
□ Error messages verified?

Test Quality:
□ Each test has single purpose?
□ Tests are independent (no order dependency)?
□ Tests use Arrange-Act-Assert structure?
□ Test names describe behavior?

Coverage:
□ Each ROD service has tests?
□ All public methods tested?
□ Integration tests exist?
□ Coverage > 80%?
```

```
Maintenance:
□ Tests don't depend on implementation details?
□ Tests serve as documentation?
□ Failing test clearly indicates problem?
```

Part 3: DGTF in Practice

The 5-Step Workflow

Step 1: Recognize

"Am I rushing right now?"

Check yourself: - Heart rate increased? - Thinking "hurry, hurry"? - Want to skip testing? - Feel pressure to "just make it work"?

If YES → You're in System 1 territory. Proceed to Step 2.

Step 2: Pause

"Wait. Let me stop."

Action: - Hands off keyboard - 3 deep breaths - Count to 5 - Ask: "What's making me rush?"

Time required: 5-10 seconds

Step 3: Check

"Check the system."

Questions: - Did I check the ROD design? - Did I check the TFD tests? - Does this change affect other parts? - What could go wrong?

Time required: 1-5 minutes

Step 4: Plan

"Plan the steps."

Write down: 1. What exactly will I change? 2. Which tests should pass after? 3. How will I verify it works? 4. Who should I ask if stuck?

Time required: 2-5 minutes

Step 5: Execute

"Proceed thoughtfully."

Method: - One small change at a time - Run tests after each change - If anything unexpected → Back to Step 2 - Commit frequently

---

Real Scenario: "Fix This Bug NOW!"

Situation:

```
Friday 4:30 PM
Manager: "Production is broken! Users can't checkout!"
You: (heart rate increases, System 1 activating)
```

Without DGTF:

```
→ Jump into code
→ Find something that looks related
→ Change it
→ Deploy
→ Different bug appears
→ Panic more
→ Weekend ruined
```

With DGTF:

```
Step 1: Recognize
"I'm panicking. System 1 is taking over."

Step 2: Pause (10 seconds)
Deep breath. "Okay, what do I actually know?"

Step 3: Check (3 minutes)
- Check error logs: "PaymentGateway timeout"
- Check monitoring: Started 30 minutes ago
- Check recent deployments: None today
- Check external status: Payment provider is down

Step 4: Plan (2 minutes)
- This is external, not our bug
- Options:
  a) Wait for provider (they say 30 min)
  b) Switch to backup provider
  c) Enable "retry later" for users
- Decision: Enable retry, notify users

Step 5: Execute (10 minutes)
```

```
- Enable retry mechanism (pre-built, tested)
- Add user-facing message
- Monitor
- Notify manager with timeline

Total time: ~15 minutes
Result: Handled professionally, no panic changes
```

Communication Templates

When Manager Asks "How Long?"

❌ System 1 Response:

```
"I'll have it done by end of day!"
(No analysis, will probably fail)
```

✅ DGTF Response:

```
"Let me check the scope and give you an accurate estimate."

[30 minutes later]

"I've analyzed the request:
- It touches 3 services
- Need 5 new test cases
- Estimated: 2 days

If we rush, we risk:
- Bugs affecting users
- More time fixing later

Recommended timeline: 2 days with proper testing."
```

When Asked to Skip Testing

❌ System 1 Response:

```
"Okay, I'll skip tests to go faster"
(Technical debt, future pain)
```

✅ DGTF Response:

```
"I understand the urgency. Let me explain the trade-off:

With tests (2 days):
```

```
- 95% confidence it works
- No weekend emergencies
- Maintainable later

Without tests (1 day):
- 50% chance of bugs
- Likely weekend fix needed
- Future changes risky

I recommend the 2-day approach.
But if we must ship in 1 day,
I'll document the risk and we should plan
for immediate follow-up testing."
```

Anti-Patterns to Avoid

Anti-Pattern 1: "Just One Quick Fix"

```
Symptom: "I'll just change this one line..."
Reality: One line becomes ten, with no tests
Result: Bug shipped, trust lost

Solution: Even "one line" gets the 5-step treatment
```

Anti-Pattern 2: "I'll Add Tests Later"

```
Symptom: "Let me get it working first, then test"
Reality: "Later" never comes
Result: Untested code in production

Solution: Write test specification BEFORE coding
```

Anti-Pattern 3: "It Works On My Machine"

```
Symptom: "Tests pass locally, ship it"
Reality: Environment differences cause failures
Result: Production bug

Solution: CI/CD with consistent environment
```

Anti-Pattern 4: "The Deadline Justifies Everything"

```
Symptom: "We HAVE to ship today, no matter what"
Reality: Shipping broken code is worse than delay
Result: Emergency fixes, lost trust, more delay

Solution: Negotiate scope, not quality
```

DGTF Checklist

```
Before Starting Work:
□ Am I calm? (If not, pause first)
□ Do I have the ROD design?
□ Do I have the TFD tests?
□ Do I understand what "done" means?

During Work:
□ Am I following the service chain?
□ Am I running tests after each change?
□ Am I committing frequently?
□ Do I feel rushed? (If yes, pause)

When Pressured:
□ Did I pause before responding?
□ Did I give honest estimate?
□ Did I explain trade-offs?
□ Did I document the decision?

At the End:
□ All tests pass?
□ Code reviewed?
□ Documentation updated?
□ Ready for next person to maintain?
```

Part 4: Complete Example - E-commerce Shopping Cart

Requirement

"User can add items to cart, view cart, and checkout"

Step 1: ROD - Build Service Chain

Sub-requirement 1: Add to Cart

```
AddToCartService Chain:
├── ValidateAddToCartRequest
├── FindUser
├── FindProduct
├── CheckProductAvailability
├── GetOrCreateCart
├── AddItemToCart
├── CalculateCartTotal
├── SaveCart
└── LogCartEvent
```

Sub-requirement 2: View Cart

```
ViewCartService Chain:
├── ValidateViewCartRequest
├── FindUser
├── FindCart
├── GetCartItems
├── EnrichCartItems (add product details)
├── CalculateCartTotal
├── ApplyPromotions
└── ReturnCartView
```

Sub-requirement 3: Checkout

```
CheckoutService Chain:
├── ValidateCheckoutRequest
├── FindUser
├── FindCart
├── ValidateCartItems (still available?)
├── CalculateTotal
├── ApplyDiscounts
├── CalculateTax
├── CalculateShipping
├── ReserveInventory
├── ProcessPayment
│   ├── ValidatePaymentMethod
│   ├── ChargePayment
│   └── HandlePaymentResult
├── CreateOrder
├── ConfirmInventoryReduction
├── ClearCart
├── SendConfirmationEmail
└── LogCheckoutEvent
```

Step 2: TFD - Design Tests

Tests for AddItemToCart:

```go
func TestAddItemToCart(t *testing.T) {
    // Happy path
    t.Run("ValidItem_AddsToCart", func(t *testing.T) {})
    t.Run("ItemAlreadyInCart_IncreasesQuantity", func(t *testing.T) {})

    // Failure cases
    t.Run("ProductNotFound_ReturnsError", func(t *testing.T) {})
    t.Run("ProductOutOfStock_ReturnsError", func(t *testing.T) {})
    t.Run("UserNotFound_ReturnsError", func(t *testing.T) {})
    t.Run("InvalidQuantity_ReturnsError", func(t *testing.T) {})

    // Edge cases
```

```
    t.Run("QuantityExceedsStock_ReturnsError", func(t *testing.T) {})
    t.Run("MaxCartItems_ReturnsError", func(t *testing.T) {})
}
```

Tests for Checkout:

```
func TestCheckout(t *testing.T) {
    // Happy path
    t.Run("ValidCart_CompletesCheckout", func(t *testing.T) {})

    // Failure cases
    t.Run("EmptyCart_ReturnsError", func(t *testing.T) {})
    t.Run("ItemNoLongerAvailable_ReturnsError", func(t *testing.T) {})
    t.Run("PaymentFails_ReturnsError", func(t *testing.T) {})
    t.Run("PaymentFails_ReleasesInventory", func(t *testing.T) {})

    // Edge cases
    t.Run("ConcurrentCheckout_OnlyOneSucceeds", func(t *testing.T) {})
    t.Run("PriceChangedDuringCheckout_NotifiesUser", func(t *testing.T) {})
}
```

Step 3: DGTF - Implementation

Day 1: Setup and Core Services

```
Morning (System 2 active):
- Review ROD design
- Review TFD tests
- Set up project structure
- Implement interfaces

Afternoon:
- Implement ValidateRequest services
- Implement FindUser, FindProduct
- Run tests: 5 pass, 15 pending
- Commit
```

Day 2: Cart Logic

```
Morning:
- Implement GetOrCreateCart
- Implement AddItemToCart
- Implement CalculateTotal
- Run tests: 12 pass, 8 pending
- Commit

Afternoon:
- Implement SaveCart
- Implement ViewCart flow
```

```
- Run tests: 18 pass, 2 pending
- Commit
```

Day 3: Checkout Flow

```
Morning:
- Implement inventory reservation
- Implement payment processing
- Run tests: 22 pass, 3 fail (edge cases)
- Debug with System 2 (not panic)

Afternoon:
- Fix edge cases
- Implement order creation
- Run tests: 25 pass
- Commit
```

Day 4: Polish and Edge Cases

```
Morning:
- Implement notification services
- Add logging
- Run full test suite: All pass
- Code review

Afternoon:
- Address review comments
- Final testing
- Documentation
- Ready for deployment
```

Result

```
Time spent: 4 days
Tests: 25 passing
Coverage: 87%
Bugs in production: 0

Compare to rushing (estimated):
Time spent: 2 days coding + 3 days fixing
Tests: 5 (added after bugs)
Coverage: 40%
Bugs in production: 8
Customer complaints: 12
Weekend ruined: Yes
```

Part 5: For Junior Developers

Where to Start

Week 1-2: Understand the Problem

Don't start with methodology. Start with understanding WHY good practices exist.

Exercise: 1. Think of a time you wrote "quick fix" code 2. What happened next? 3. How much time did you spend fixing it?

This is why DGTF++ exists.

Month 1: Start with DGTF

Before learning ROD or TFD, learn to recognize rushing.

Daily practice: - Every time you feel "hurry," pause for 5 seconds - Ask: "What's making me rush?" - Write it down

This builds the habit of recognition.

Month 2: Add TFD

Start writing tests before code.

Daily practice: - Before writing any function, write one test - Just one. Not perfect coverage. - Build the habit.

```
// Before writing the function:
func TestAdd_TwoPositiveNumbers_ReturnsSum(t *testing.T) {
    result := Add(2, 3)
    if result != 5 {
        t.Errorf("Expected 5, got %d", result)
    }
}

// Then write the function:
func Add(a, b int) int {
    return a + b
}
```

Month 3: Add ROD

Start thinking about service chains.

Daily practice: - Before implementing a feature, draw the services - Just boxes and arrows on paper - Ask: "What's missing?"

Common Mistakes

Mistake 1: Trying Everything at Once

```
❌ "I'll apply ROD + TFD + DGTF perfectly from day one"
→ Overwhelm → Give up

✅ "I'll start with just pausing when I rush"
→ Small win → Build on it
```

Mistake 2: Seeking Perfection

```
❌ "My service chain must be perfect"
→ Analysis paralysis → Nothing done

✅ "My service chain is good enough to start"
→ Iterate and improve
```

Mistake 3: Applying to Everything

```
❌ "I must use ROD for this one-line script"
→ Overhead kills productivity

✅ "ROD for features, quick scripts can be simple"
→ Right tool for the job
```

Finding a Mentor

Look for someone who: - Writes tests before code (naturally) - Stays calm under pressure - Asks “what could go wrong?” often - Is willing to explain their thinking

Ask them: - “Can I see how you design a feature?” - “How do you decide what to test?” - “How do you handle deadline pressure?”

Part 6: Measuring Success

Individual Metrics

Before DGTF++:

```
- Bugs introduced per feature: 5-10
- Time spent debugging: 40%
- Weekend work: Frequent
- Confidence when deploying: Low
```

After DGTF++ (3 months):

```
- Bugs introduced per feature: 1-2
- Time spent debugging: 15%
- Weekend work: Rare
- Confidence when deploying: High
```

Team Metrics

Track monthly:

```
- Escaped defects (bugs found in production)
- Rework ratio (time fixing vs building)
- Deployment confidence score (1-10)
- Test coverage percentage
```

Signs of Progress

Week 2: - You notice when you're rushing - You pause sometimes (not always)

Month 1: - Pausing becomes habit - You write some tests before code - Fewer "emergency" fixes

Month 3: - Service chains feel natural - Tests are automatic part of workflow - Colleagues notice your code quality

Month 6: - Others ask how you stay calm - You can explain WHY you work this way - Junior developers want to learn from you

---

Part 7: FAQ & Hard Questions

Q: "This takes too much time upfront"

A: Let's compare total time:

```
Without DGTF++:
- Design: 0.5 days
- Code: 2 days
- Debug: 3 days
- Fix production bugs: 2 days
- Total: 7.5 days

With DGTF++:
- ROD: 0.5 days
- TFD: 0.5 days
- Code: 2 days
- Debug: 0.5 days
- Production bugs: 0
- Total: 3.5 days
```

The upfront time saves more than it costs.

Q: “My manager wants results NOW”

A: Communicate in business terms:

“I can ship in 2 days with testing, or 1 day without.

Without testing: - 50% chance of production bug - Bug fix will take 1-2 days - Customer impact: negative reviews - Expected total time: 3 days

With testing: - 95% confidence it works - Minimal customer impact - Expected total time: 2 days

Which do you prefer?”

Q: “How to apply to legacy code?”

A: Gradually.

```
Month 1: New features only
- Apply ROD + TFD to new code
- Don't touch legacy
- Build team skill

Month 2-3: Boy Scout Rule
- When modifying legacy, add tests
- Small improvements only
- Don't rewrite

Month 4-6: Strategic improvement
- Identify highest-risk modules
- Apply ROD redesign
- Comprehensive testing

Never: Big Bang Rewrite
- Too risky
- Too expensive
- Usually fails
```

Q: “I’m the only one using this. How to spread?”

A: By results, not preaching.

```
Month 1-2: Demonstrate
- Apply methodology quietly
- Track your metrics
- Let quality speak

Month 3: Share when asked
- "How do you avoid bugs?"
- "Why is your code so clean?"
```

```
- Then explain

Month 4+: Pair programming
- Offer to work together
- Show, don't tell
- Let them experience it
```

Q: "What about urgent production issues?"

A: DGTF still applies—just faster.

```
Normal DGTF:
- Pause: 30 seconds
- Check: 5 minutes
- Plan: 5 minutes

Emergency DGTF:
- Pause: 5 seconds (still pause!)
- Check: 1 minute (what do I know?)
- Plan: 1 minute (simplest safe fix)

The pause is never zero.
Even 5 seconds prevents panic mistakes.
```

Hard Questions (Devil's Advocate)

No methodology is perfect, and honest critique strengthens understanding. These questions address real limitations and concerns. If you're skeptical, you should be—skepticism is System 2 working correctly.

Q1: "How can I recognize System 1 when System 1 is active?"

The Paradox: DGTF Step 1 says "recognize you're rushing." But System 1's nature is unconscious. In true panic, you don't know you're panicking. "I'm fine" might be the most dangerous thought.

Honest Answer:

You often cannot recognize it in the moment. This is a genuine limitation. That's why DGTF emphasizes:

1. **External triggers**, not self-awareness
   - Deadline announced → automatic pause
   - Bug reported → automatic pause
   - "Urgent" in message → automatic pause
2. **Habits, not willpower**
   - Don't rely on recognizing panic
   - Build reflexes that trigger before panic

3. **Environmental design**
   - Pre-commit hooks that force tests
   - PR requirements that force review
   - Physical cues (sticky notes, timers)

The goal isn't perfect self-awareness. It's reducing the situations where you need it.

Q2: "What about ego depletion? Willpower runs out."

The Reality: Self-control is a finite resource. End of day, end of week, end of project—when willpower is exhausted, how do you practice DGTF?

Honest Answer:

This is a real limitation. DGTF doesn't solve fatigue. But:

1. **Reduce willpower dependency**
   - Make DGTF automatic (habit)
   - Use external forcing functions
   - When depleted, stop working—seriously
2. **Sustainable pace**
   - DGTF includes "don't overwork"
   - A 60-hour week makes DGTF impossible
   - This is a symptom of organizational failure
3. **Accept imperfection**
   - You will fail DGTF when exhausted
   - That's information: you're too tired
   - Go home

Q3: "What if my environment is toxic?"

The Reality: "Slow and careful" could get you fired. Some organizations punish quality.

Honest Answer:

DGTF assumes a minimum of organizational rationality. In truly toxic environments:

1. **Short-term: Survive**
   - Apply DGTF where safe
   - Document your quality work
   - Build a portfolio
2. **Medium-term: Decide**
   - Is this environment changeable?
   - Are there allies who want quality?
   - What's your leverage?
3. **Long-term: Exit or Change**

- Toxic environments don't fix themselves
- Your career is longer than this job
- DGTF skills transfer to better places

What DGTF cannot solve: Systemic organizational dysfunction. No personal methodology can fix a broken company. This is not a failure of the methodology—it's a recognition of its scope.

In some environments, DGTF may not be externally rewarded. However, you still benefit personally: less stress, cleaner code, better skills, and a stronger portfolio for your next role.

Q4: "DGTF is unverifiable. How do I know I'm doing it?"

The Reality: TFD has pass/fail. ROD has "missing found." DGTF measures internal state—unmeasurable.

Honest Answer:

Direct measurement is impossible. Use proxy metrics:

```
Proxy Metrics for DGTF:
- Bugs per feature (should decrease)
- "Emergency" fixes (should decrease)
- Code review iterations (should decrease)
- Personal stress level (should decrease)
- Regrettable commits (should decrease)
```

Also: **DGTF fails visibly**. When you skip it, consequences appear. Bugs emerge. Rework happens. The absence of DGTF is measurable.

Q5: "The virtuous cycle assumes rational managers"

The Reality: DGTF → Trust → Autonomy → Less Pressure → Easier DGTF… assumes managers reward quality. Many don't.

Honest Answer:

Yes, this cycle requires minimally rational management. If your manager: - Punishes quality - Rewards only speed - Ignores outcomes

Then the cycle breaks. Your options: 1. Find different management 2. Build evidence that changes their mind 3. Accept that DGTF may not be externally rewarded—while still valuable for your own growth

Not every environment supports quality practices. But the skills you build remain yours.

Q6: "ROD requires experience. Juniors can't find Missing."

The Reality: "Design completely" requires knowing what's missing. Juniors lack the experience to predict missing elements.

Honest Answer:

True. ROD alone is senior-dependent. But:

1. **Juniors can learn patterns**
   - Common missing elements are documented
   - Error handling, logging, validation, cleanup
   - These are predictable
2. **Juniors need mentors**
   - Design review with seniors
   - "What did I miss?" sessions
   - Learning Missing over time
3. **TFD helps juniors**
   - Tests reveal Missing earlier
   - Failure feedback is faster
   - Learning accelerates

ROD is harder for juniors. That's why DGTF++ includes "For Junior Developers" section—acknowledge the difficulty, provide scaffolding.

Q7: "How do I distinguish 'feels urgent' from 'is urgent'?"

The Reality: Production down costs money. Every second matters. Where's the line?

Honest Answer:

```
True Urgency Examples:
- Security breach in progress → Act fast (but still think)
- Data corruption spreading → Contain immediately
- Revenue loss per second → Prioritize ruthlessly

False Urgency Examples:
- Manager said "ASAP" → Probably not life-or-death
- Customer complained loudly → Important but not urgent
- Deadline is tomorrow → You had weeks; this is planning failure
```

Even in true emergencies, DGTF applies—just compressed: - 5 seconds of pause prevents wrong server shutdown - 1 minute of diagnosis prevents data loss - "Fast" doesn't mean "thoughtless"

The highest-stakes environments (surgery, aviation) have the MOST pauses and checklists, not fewer.

Q8: "DGTF is individual. What about team pressure?"

The Reality: - Pair programming partner says "just do it" - Code review asks "why so slow?" - Standup pressure to show progress

Honest Answer:

Team DGTF requires:

1. **Team agreement**
   - Discuss methodology as a team
   - Agree on standards
   - "We value quality" as shared value
2. **Explicit communication**
   - "I'm pausing to think through this"
   - "I want to get this right"
   - Name what you're doing
3. **Handling pressure**
   - "I understand the urgency. Let me check one thing."
   - "I can go faster, but we'll likely have bugs. Your call."
   - Make tradeoffs explicit
4. **Cultural change** (hard)
   - One person can model behavior
   - Results eventually speak
   - Not always possible

Q9: "What's the recovery strategy when DGTF fails?"

The Reality: You panicked. Used a global variable. Skipped tests. What now?

Honest Answer:

DGTF failure recovery:

```
1. Acknowledge (don't deny)
   - "I rushed that"
   - No shame, just recognition

2. Contain
   - Don't compound the problem
   - Stop rushing NOW
   - Prevent spread

3. Assess
   - What did I skip?
   - What's the damage?
   - What's the risk?

4. Fix or Accept
   - If fixable now: fix properly
   - If not: document as tech debt
```

```
   - Plan remediation

5. Learn
   - What triggered the rush?
   - How to prevent next time?
   - Update environment/habits
```

Key insight: DGTF failure is information. It tells you something about your environment, your state, or your triggers. Use it.

Q10: "Reading this document triggers System 1"

The Meta-Problem: Understanding DGTF conceptually feels like learning it. "Got it, next!" But reading is System 1. You haven't actually learned anything.

Honest Answer:

Absolutely true. Reading ≠ Learning. That's why this guide emphasizes:

1. **Practice, not reading**
   - "Execution Plan" section
   - Weekly practice goals
   - Deliberate application
2. **Time requirements**
   - "Month 1: Conscious incompetence"
   - "Month 3: Conscious competence"
   - "Month 6+: Unconscious competence"
3. **Failure as teacher**
   - You will fail DGTF
   - Failure teaches what reading cannot
   - This is expected and necessary

Reading this guide and feeling you "understand" is just the beginning. Real understanding comes from failed attempts, corrections, and gradual improvement over months. The document is a map; walking the territory is different.

Part 8: Execution Plan

Week 1: Understanding

```
Day 1-2:
□ Read Core Concepts document
□ Understand System 1/2
□ Understand service chains

Day 3-4:
```

□ Read this Practical Guide
□ Take notes on what resonates
□ Identify one feature to practice

Day 5:
□ Discuss with mentor/colleague
□ Answer: "Why do I want to try this?"
□ Set 4-week goal

Week 2: DGTF Practice

Daily:
□ Notice when rushing (at least 3 times)
□ Pause (even if briefly)
□ Journal what triggered rush

End of week:
□ Review journal
□ Common triggers identified?
□ Pausing becoming easier?

Week 3: TFD Practice

Daily:
□ Write at least 1 test before code
□ Keep tests simple
□ Run tests frequently

End of week:
□ Count tests written
□ Compare bug count to previous weeks
□ Feeling of testing "before" vs "after"?

Week 4: ROD Practice

Daily:
□ Sketch service chain before coding
□ Just boxes and arrows
□ Ask "what's missing?"

End of week:
□ Review sketches
□ Did chains prevent confusion?
□ Implementation easier?

Month 2-3: Integration

Weekly:
□ Apply all three to one feature
□ Track metrics (bugs, time, confidence)
□ Review and adjust

Monthly:

```
□ Compare metrics to Month 1
□ Share learnings with team
□ Adjust approach based on experience
```

Month 4-6: Mastery

```
Goals:
□ DGTF is automatic
□ TFD is default workflow
□ ROD is natural thinking
□ Metrics significantly improved
□ Can teach others
```

Final Notes

Remember

```
DGTF++ is not about:
- Being perfect
- Being slow
- Following rules blindly

DGTF++ is about:
- Making better decisions
- Sustainable pace
- Professional results
```

The Journey

```
Day 1: "This seems like a lot"
Week 1: "I noticed I was rushing!"
Month 1: "I wrote tests first today"
Month 3: "Why didn't I learn this earlier?"
Month 6: "This is just how I work now"
```

When It Clicks

You'll know DGTF++ is working when: - You feel calm during pressure - Your code needs less debugging - Colleagues trust your estimates - You look forward to deployment

Good luck. Take it slow. That's the point.

About the Author

Bakmeon Kim (김백면)

Software Engineer & Development Philosophy Practitioner

Bakmeon Kim has been building software for over 25 years.

He began his career as a developer in 2000, eventually leading research institutes and serving as CTO across multiple companies. His experience spans enterprise solutions, AI platforms, smart factories, AR/VR applications, and security systems, with projects delivered in Korea, Taiwan, Japan, and the United States.

What sets Bakmeon apart is his commitment to depth over breadth. In an industry where developers typically change jobs every 2-3 years, he spent 19 years at a single company—building, refining, and perfecting his craft. He served as Research Institute Director for over 15 years, leading teams and conducting software engineering studies for more than a decade.

His philosophy is simple but profound: **good software is not software with many features, but software where features can be easily added.** This focus on maintainability—what he calls "change responsiveness"—has been the driving force behind his work.

Professional Highlights

- *25+ years* in software development
- *19 years* at one company, demonstrating long-term commitment
- *15+ years* as Research Institute Director
- *10+ years* leading software engineering studies
- *100+ projects* delivered across multiple industries
- International project experience in *Taiwan, Japan, and USA*

Industries Experienced

- SCM (Supply Chain Management) systems
- MES-based smart factory platforms
- AI-powered demand forecasting/quality inspection/predictive maintenance
- AR/VR industrial metaverse
- AI-based security solutions (current)

Areas of Expertise

Enterprise Solutions:

- SCM (Supply Chain Management) systems
- MES (Manufacturing Execution Systems)

- Smart Factory platforms

AI & Emerging Technology:

- Deep learning for demand forecasting
- Computer vision for quality inspection
- LLM/sLLM applications
- AR/VR industrial metaverse solutions

Software Engineering:

- Test-first development culture
- Clean architecture and design patterns
- Team education and mentoring
- Development process optimization

Current (2025~)

Currently developing an AI-based security solution, applying ROD, TFD, and DGTF principles to network security.

From direct experience, he has found that when you use AI well, you can achieve productivity similar to working with 3-4 developers. But this doesn't diminish the importance of DGTF—it makes it more essential.

Origin of the Principles

In 1992, upon entering university, he felt behind among classmates who already knew how to program. He made two commitments then:

> "Never stop thinking." "Never stop being curious."

These commitments became the foundation of this book 30 years later.

Education

- *M.S. in Computer Science* (Software Engineering focus) Soongsil University, Seoul, Korea
- *B.S. in Computer Science* Soongsil University, Seoul, Korea

Philosophy

> "I hate doing the same thing twice. If I have to do something more than three times, I automate it."
>
> "Good software is not software with many options. Good software is software where options can be easily added."
>
> *"Never stop thinking. Never stop being curious."*

These principles, established during his university years and refined over decades of practice, form the foundation of the ROD, TFD, and DGTF methodologies presented in this book.

Personal

Bakmeon lives in Seoul, Korea, with his wife—whom he met in a high school English conversation club—and their daughter.

When not coding, he enjoys:

- Roasting coffee at home
- Playing clarinet (learning since 2015)
- Woodworking at local workshops
- Leather crafting
- 3D printing
- Vocal lessons (since 2023)
- Studying psychology through distance learning

He quit smoking in 2013 and now enjoys occasional drinks with his daughter.

Connect

- *GitHub:* https://github.com/bakmeon/dont-go-too-fast
- *Email:* stillblueist@naver.com

This book represents 25 years of learning, practicing, and teaching. It is written not from theory alone, but from the daily reality of building software that lasts.

Remember: Don't Go Too Fast 🐢