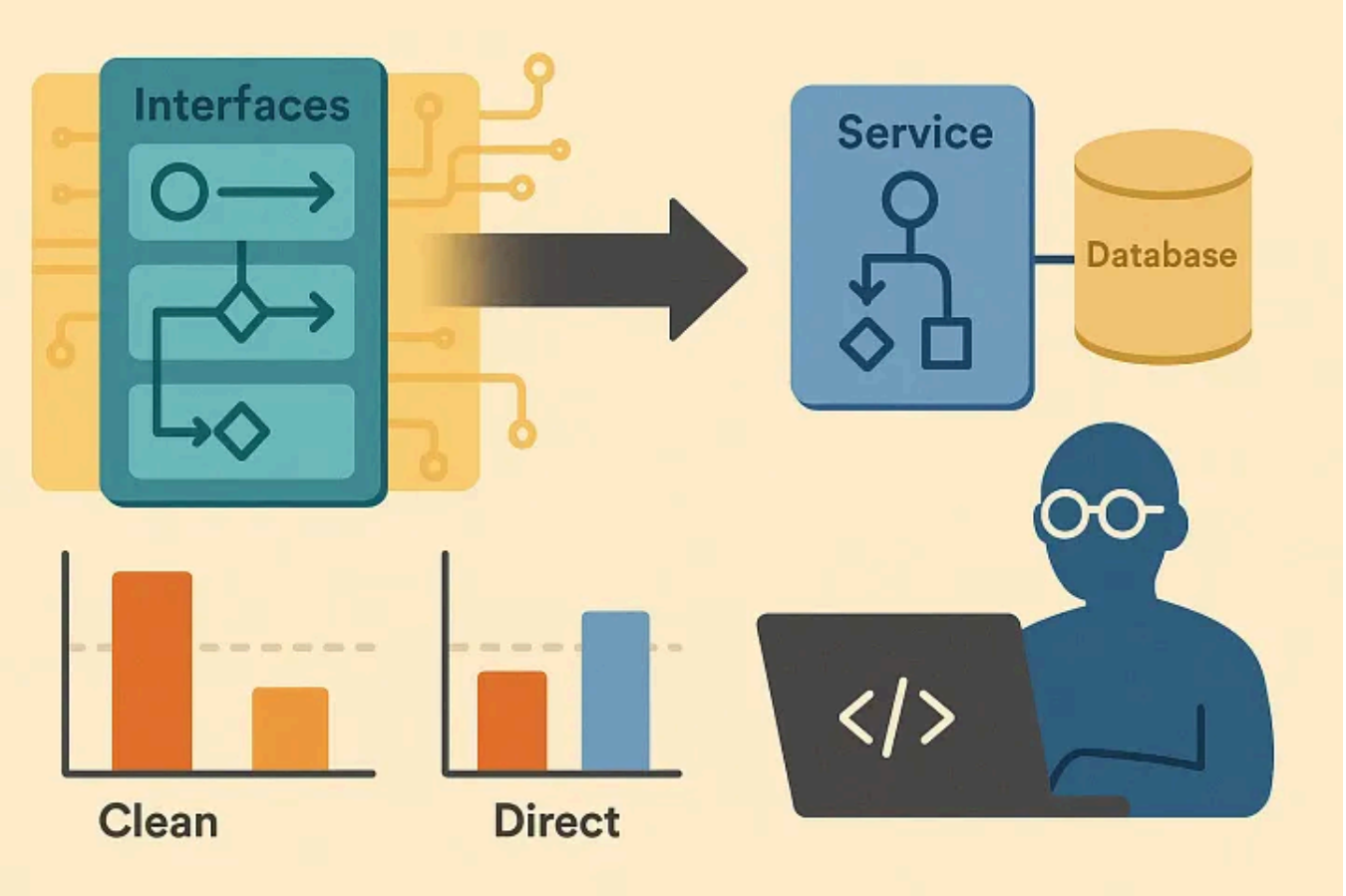
Member-only story

Why I Stopped Using Clean Code (And You Should Too)

4 min read · Aug 21, 2025

The Latency Gambler

After four years of building production systems and wrestling with legacy codebases, I've come to a controversial conclusion: the Clean Code movement has done more harm than good to our industry. Don't get me wrong , I used to be a zealot. I'd refactor perfectly working code just to make it "cleaner," create elaborate abstractions for simple problems, and judge colleagues who dared to write a 15-line method.



> *But real-world software development taught me harsh lessons that Robert Martin's book never prepared me for.*

The Over-Abstraction Trap

Clean Code preaches that abstraction is always good. More layers, more interfaces, more indirection. Here's what I used to write:

```
// "Clean" version - looks professional, right?
interface PaymentProcessor {
  process(amount: Money): Promise<PaymentResult>;
}

class StripePaymentProcessor implements PaymentProcessor {
  async process(amount: Money): Promise<PaymentResult> {
    return this.gateway.charge(amount);
  }
}
class PaymentService {
  constructor(private processor: PaymentProcessor) {}

  async processPayment(order: Order): Promise<void> {
    const result = await this.processor.process(order.total);
    if (!result.isSuccess) {
      throw new PaymentFailedException(result.error);
    }
```

```
  }
}
```

Versus what I write now:

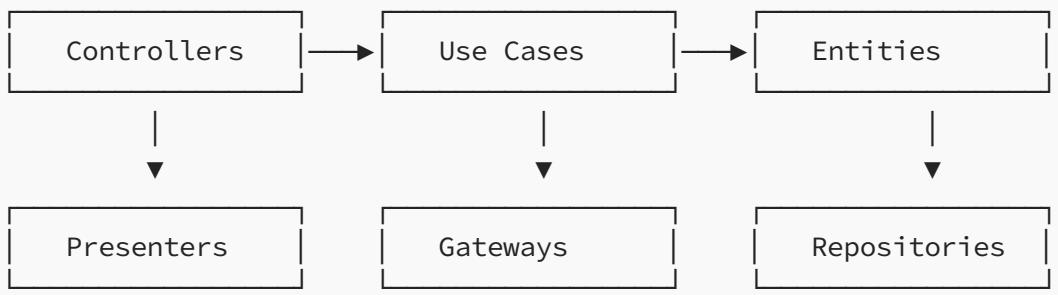
```
// Pragmatic version
class PaymentService {
  async processStripePayment(order: Order): Promise<void> {
    const result = await stripe.charges.create({
      amount: order.total.cents,
      currency: 'usd',
      source: order.paymentToken
    });

    if (!result.paid) {
      throw new Error(`Payment failed: ${result.failure_message}`);
    }
  }
}
```

> *The second version is immediately understandable. A junior developer can debug it in seconds. The first version? Good luck tracing through three layers of abstraction when Stripe's API changes.*

Architecture Reality Check

Here's how Clean Code advocates think systems should look:



Here's what actually works in most applications:

> *Three layers. That's it. Every successful startup I've worked with follows this pattern. The ones that tried Clean Architecture? They're still refactoring while their competitors ship features.*

The Performance Penalty

Clean Code's obsession with small functions and excessive abstraction has real costs. I benchmarked this simple operation on Node.js 20:

```
// Clean Code approach
function calculateTotal(items) {
  return items
    .map(item => item.price)
    .filter(price => isValid(price))
    .reduce((sum, price) => sum + price, 0);
}

function isValid(price) {
  return price > 0;
}
// Direct approach
function calculateTotalDirect(items) {
  let total = 0;
  for (let i = 0; i < items.length; i++) {
    if (items[i].price > 0) {
      total += items[i].price;
    }
  }
  return total;
}
```

Benchmark results (1M items):

- Clean approach: 847ms
- Direct approach: 23ms

> *That's a 37x performance difference. In a checkout flow processing thousands of transactions per minute, this matters.*

When "Readable" Code Isn't

Clean Code claims shorter functions are more readable. But context switching between functions destroys comprehension. Compare these:

```
# "Clean" version
def process_user_registration(user_data):
    validate_user_data(user_data)
    user = create_user_entity(user_data)
    save_user_to_database(user)
    send_welcome_email(user)
    log_registration_event(user)

def validate_user_data(data):
    if not data.get('email'):
        raise ValidationError('Email required')
    # ... more validation
def create_user_entity(data):
    return User(
        email=data['email'],
        password=hash_password(data['password'])
    )
# ... 3 more functions
```

Versus:

```
# Pragmatic version
def process_user_registration(user_data):
    # Validate input
    if not user_data.get('email'):
        raise ValidationError('Email required')
    if not user_data.get('password'):
        raise ValidationError('Password required')

    # Create user
    user = User(
        email=user_data['email'],
        password=bcrypt.hashpw(user_data['password'].encode(), bcrypt.gensalt()
    )

    # Save to database
    db.session.add(user)
    db.session.commit()
```

```
    # Send welcome email
    email_service.send_template('welcome', user.email, {'name': user.name})

    # Log the event
    logger.info(f'New user registered: {user.email}')
```

> *The second version tells a complete story. I can understand the entire flow without jumping between functions. When debugging a registration failure at 2 AM, which would you prefer?*

The Pragmatic Alternative

After years of fighting over-engineered codebases, here's what I've learned works:

1. **Write boring code** — If a junior developer can't understand it in 30 seconds, it's too clever
2. **Optimize for debugging** — Code is read during crises, not casual browsing
3. **Embrace redundancy** — DRY is less important than locality of behavior
4. **Measure before abstracting** — Most abstractions solve problems you don't have

Real-World Evidence

The most successful codebases I've maintained follow these principles:

```
GitHub (Ruby on Rails):     Monolithic, straightforward
Stack Overflow (ASP.NET):   Pragmatic, performance-focused
Shopify (Ruby):             Modular but not over-abstracted
```

> *Meanwhile, every project that strictly followed Clean Code principles became an unmaintainable mess of interfaces and abstractions.*

Conclusion

Clean Code isn't evil, but treating it as gospel is dangerous. It optimizes for the wrong things: making code look professional rather than making it work reliably. After four years of maintaining production systems, I've learned that readable, performant, debuggable code matters more than architecturally pure code.

> *Your future self, debugging a critical issue at 3 AM, will thank you for writing obvious code instead of clean code.*

The best code isn't clean , it's pragmatic.

Clean Code | Design Patterns | Software Development | Programming | Software Engineering

Follow

Written by The Latency Gambler

12.3K followers · 2 following

Tech critic exploring tools, systems & languages beyond hype. Linkedin- https://www.linkedin.com/in/kanishk-singh-140059189/

Responses (69)

Bgerby

What are your thoughts?

Wchasroth
Aug 21

OK, them's fightin' words! :-)

Seriously, though, the whole argument is a "strawman" logical fallacy. Any methodology taken to ridiculous extremes will be bad. So what?

But when you say "Clean Code preaches that abstraction is always good", that's... more

415 5 replies Reply

Luiz Armesto
Aug 24

> I'd refactor perfectly working code just to make it "cleaner," create elaborate abstractions for simple problems, and judge colleagues who dared to write a 15-line method.

At least you made it clear from the beginning of the post that you had no idea what you were doing.

174 2 replies Reply

Bruce Rosner
Aug 24

Medium digest frequently contain two type of articles: Those promoting the latest magic bullet in software development and those explaining why a magic bullet of a few years ago is not.

137 2 replies Reply

See all responses

More from The Latency Gambler

The Latency Gambler

Every Bug I Ever Fixed Made Sense Only After I Understood These 7 Layers

Three years into my career, I spent two weeks debugging why our API randomly returned 502s. The logs were clean. The application was fine...

Oct 20 665 20

The Latency Gambler

I Interviewed 20+ Engineers. Here's Why Most Can't Code

Over the past year as a Senior Software Engineer at a B2B SaaS company, I've conducted 20+ technical interviews for roles ranging from...

Sep 9 3K 97

The Latency Gambler

The PostgreSQL Problem No One Warns You About (Until It's Too Late)

PostgreSQL has earned its reputation as the world's most advanced open-source database. But after managing production clusters serving...

Oct 24 414 8

The Latency Gambler

Dissecting the AWS Outage: A Step-by-Step Breakdown of What Went Wrong

I tried opening Medium at 6:30 AM on Monday, October 20th. Nothing loaded. Figured it was my wifi. Switched to mobile data. Still nothing...

Oct 22 376 12

See all from The Latency Gambler

Recommended from Medium

In Towards AI by Teja Kusireddy

We Spent $47,000 Running AI Agents in Production. Here's What Nobody Tells You About A2A and MCP.

Multi-agent systems are the future. Agent-to-Agent (A2A) communication and Anthropic's Model Context Protocol (MCP) are revolutionary. But...

Oct 16 3.4K 116

The CS Engineer

Forget JSON — These 4 Data Formats Made My APIs 5× Faster

A 1.2 KB JSON payload transformed a smooth request into a 120 ms wait. Switching formats cut that to under 20 ms for some endpoints.

Sep 29 1.4K 38

In Medialesson by Marius Schröder

JSON vs TOON — A new era of structured input?

Why structure matters more than ever

Nov 3 163 9

Arvind Kumar

Building a Ticketing System: Concurrency, Locks, and Race Conditions

What happens when 100,000 fans try to book the same concert ticket at exactly 10:00 AM? Let's design a ticketing system that prevents...

Oct 30 616 8

In AI Software Engineer by Joe Njenga

Anthropic Just Solved AI Agent Bloat—150K Tokens Down to 2K (Code Execution With MCP)

Anthropic just released smartest way to build scalable AI agents, cutting token use by 98%, shift from tool calling to MCP code execution

4d ago 417 33

Abhinav

Why Senior Engineers Are Choosing Hexagonal Over Layered Architecture

Most developers start with the classic Layered Architecture — controller, service, repository, database.

Jul 21 595 45

See more recommendations