TigerStyle and the Rising Sea

Or some of the mathematical ideas behind TigerBeetle's approach to software

@ludwigabap - 12/4/2026

Outline

A Tour through the talk

- Smaller Worlds - TigerStyle Intro
- The Rising Sea - Grothendieck
- The Art of Not Caring
- The Unconstructible - Negative space
- The Private Act
- Yoneda
- Dissolve

Smaller Worlds

How TigerStyle shrinks the world

"Put a limit on everything because, in reality, this is what we expect—everything has a limit. For example, all loops and all queues must have a fixed upper bound to prevent infinite loops or tail latency spikes." - TigerStyle

- Bounded loops, no hidden recursion
- Strong invariants, lots of assertions (Hoare triple style)
- Domain-specific integer types
- Static memory -> minimal dynamic allocation
- Zero technical debt

Smaller Worlds

A concrete example

```zig
pub const Account: type = extern struct {
    id: u128,
    debits_pending: u128,
    debits_posted: u128,
    credits_pending: u128,
    credits_posted: u128,
    /// Opaque third-party identifiers to link this account (many-to-one) to external entities.
    user_data_128: u128,
    user_data_64: u64,
    user_data_32: u32,
    /// Reserved for accounting policy primitives.
    reserved: u32,
    ledger: u32,
    /// A chart of accounts code describing the type of account (e.g. clearing, settlement).
    code: u16,
    flags: AccountFlags,
    timestamp: u64,

    comptime {
        assert(stdx.no_padding(Account));
        assert(@sizeOf(Account) == 128);
        assert(@alignOf(Account) == 16);
    }

    pub fn debits_exceed_credits(self: *const Account, amount: u128) bool {
        return (self.flags.debits_must_not_exceed_credits and
            self.debits_pending + self.debits_posted + amount > self.credits_posted);
    }

    pub fn credits_exceed_debits(self: *const Account, amount: u128) bool {
        return (self.flags.credits_must_not_exceed_debits and
            self.credits_pending + self.credits_posted + amount > self.debits_posted);
    }
};
```

- Assertions that restrain what an Account can be
- Ensures no padding, exact 128 bytes size, 16 bytes alignment

The Rising Sea

Enters Grothendieck

> I can illustrate the second approach with the same image of a nut to be opened. The first analogy that came to my mind is of immersing the nut in some softening liquid, and why not simply water? From time to time you rub so the liquid penetrates better, and otherwise you let time pass. The shell becomes more flexible through weeks and months—when the time is ripe, hand pressure is enough, the shell opens like a perfectly ripened avocado!
>
> [Grothendieck 1985–1987, pp. 552-3]

The Rising Sea

Enters Grothendieck

> I can illustrate the second approach with the same image of a nut to be opened. The first analogy that came to my mind is of immersing the nut in some softening liquid, and why not simply water? From time to time you rub so the liquid penetrates better, and otherwise you let time pass. The shell becomes more flexible through weeks and months—when the time is ripe, hand pressure is enough, the shell opens like a perfectly ripened avocado!
>
> [Grothendieck 1985–1987, pp. 552-3]

The claim: this is what TigerStyle does to bugs.

You don't try and catch infinite loops, you don't test for them, you make them quite literally

Same with memory allocation: you don't fight use-after-free bugs, you remove the conditions that allow for them existence. If everything is statically allocated, you can't possibly free and you definitely cannot run use-after-free issues. fundamentally unconstructible.

Same with memory allocation: you don't fight use-after-free bugs, you remove the conditions that allow for them existence. If everything is statically allocated, you can't possibly free and you definitely cannot run use-after-free issues.

The Rising Sea

A morale in this story?

The work is upfront. Choosing the right definitions, the constraints, the world you choose to live in.

Once those are right, the "theorems" become obvious and the code becomes boring.

That's the ultimate goal!

The Rising Sea

A morale in this story?

The work is upfront. Choosing the right definitions, the constraints, the world you choose to live in.

Once those are right, the "theorems" become obvious and the code becomes boring.

That's the ultimate goal!

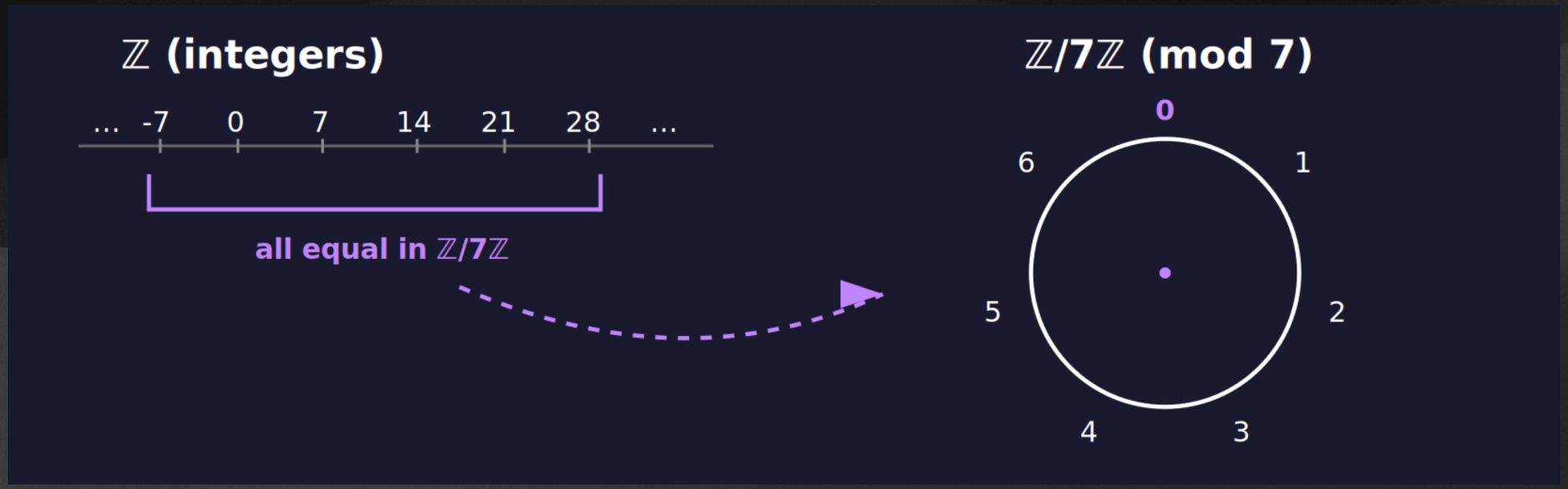
"Think about performance from the outset, from the beginning. The best time to solve performance, to get the huge 1000x wins, is in the design phase, which is precisely when we can't measure or profile. It's also typically harder to fix a system after implementation and profiling, and the gains are less. So you have to have mechanical sympathy. Like a carpenter, work with the grain."

- TigerStyle

HAMMER & CHISEL	RISING SEA
Find the bug	Remove its preconditions
Test for edge cases	Make edge cases unconstructible
Catch at runtime	Reject at compile time
Fight the problem	Dissolve the problem

The Art of Not Caring

Equivalence classes, or when not caring pays off



$\mathbb{Z}/7\mathbb{Z}$

$\{0, 1, 2, 3, 4, 5, 6\}$

The Art of Not Caring

Good types, bad types

```zig
pub const Account: type = extern struct {
    id: u128,
    debits_pending: u128,
    debits_posted: u128,
    credits_pending: u128,
    credits_posted: u128,
    /// Opaque third-party identifiers to link this account (many-to-one) to external entities.
    user_data_128: u128,
    user_data_64: u64,
    user_data_32: u32,
    /// Reserved for accounting policy primitives.
    reserved: u32,
    ledger: u32,
    /// A chart of accounts code describing the type of account (e.g. clearing, settlement).
    code: u16,
    flags: AccountFlags,
    timestamp: u64,

    comptime {
        assert(stdx.no_padding(Account));
        assert(@sizeOf(Account) == 128);
        assert(@alignOf(Account) == 16);
    }

    pub fn debits_exceed_credits(self: *const Account, amount: u128) bool {
        return (self.flags.debits_must_not_exceed_credits and
            self.debits_pending + self.debits_posted + amount > self.credits_posted);
    }

    pub fn credits_exceed_debits(self: *const Account, amount: u128) bool {
        return (self.flags.credits_must_not_exceed_debits and
            self.credits_pending + self.credits_posted + amount > self.debits_posted);
    }
};
```

This is exactly what a good type does. A well-designed Amount type doesn't expose whether it's stored as cents or whatever. The internal representation is an implementation detail you've ***defined out of existence*** at the interface level.

The Art of Not Caring

Good types, bad types

```
if (a.flags.debits_must_not_exceed_credits and a.flags.credits_must_not_exceed_debits) {
    return .flags_are_mutually_exclusive;
}
```

You cannot construct an Account with both flags. The invalid state has no representation.

The Art of Not Caring

"The usual suspects for off-by-one errors are casual interactions between an index, a count or a size. These are all primitive integer types, but should be seen as distinct types, with clear rules to cast between them."

An index and a count are both integers. But in TigerStyle, you treat them as different types. You simply do not allow them be confused.

The Unconstructible

Negative space

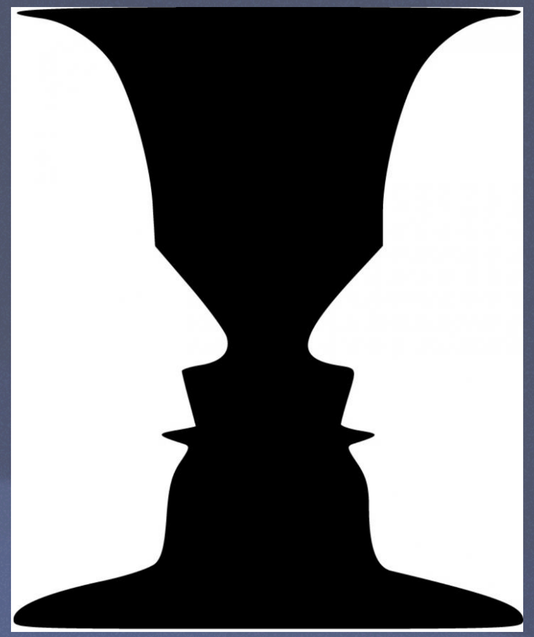
"The golden rule of assertions is to assert the positive space that you do expect AND to assert the negative space that you do not expect because where data moves across the valid/invalid boundary between these spaces is where interesting bugs are often found."

it isn't about never allowing mistakes: you want to see mismatches between your intuition and the formal model. you just want to see them early:

- at construction time

- in simulation

- as an assertion failure



Assertions, type errors, simulation failures are all ways that your model of the problem gets to tell you that "the world you thought you were in and the world you have modeled do not match."

The Private Act

The code is not the system.

Your mental model of the system is the system.

The code is a shadow.

The Private Act

The code is not the system.

Your mental model of the system is the system.

The code is a shadow.

"Take your time in the design phase. Simplicity can only come from truly understanding the problem." - TigerStyle

"Consider the inputs and outputs of the system... before you write code. Consider the inputs and outputs of subsystems... before you write code. Understand the data flow through the system... before you write code." - lobste.rs discussion on TigerStyle

Yoneda

A very Zen-like perspective

"[Mathematical] objects are completely determined by their relationships to other objects"

- Tai-Danae Bradley

Yoneda

A concrete example (forgive the reach)

```
pub const Operation: type = enum(u8) {
    /// Operations exported by TigerBeetle:
    pulse = constants.vsr_operations_reserved + 0,

-   get_change_events = constants.vsr_operations_reserved + 9,

    create_accounts = constants.vsr_operations_reserved + 10,
    create_transfers = constants.vsr_operations_reserved + 11,
    lookup_accounts = constants.vsr_operations_reserved + 12,
    lookup_transfers = constants.vsr_operations_reserved + 13,
    get_account_transfers = constants.vsr_operations_reserved + 14,
    get_account_balances = constants.vsr_operations_reserved + 15,
    query_accounts = constants.vsr_operations_reserved + 16,
    query_transfers = constants.vsr_operations_reserved + 17,
```

Dissolve

Powerful choices with powerful consequences

- Bounded loops with explicit limits -> make termination provable

- Assertions on positive and negative space -> make boundaries explicit

- Domain-specific integer types (index vs. count vs. size) -> refuse to confuse things that shouldn't be confused

- Static allocation, no dynamic memory -> remove entire categories of bugs by removing their preconditions

Closing thoughts

Thank you!

Tigerstyle, or the rising sea approach:

Do not attack your problems head on but dissolve them by simply scaffolding them out of existence.

You raise the water level by choosing the right definitions, the right constraints, the right world and your problems turn from islands to islets to nothing.

The art of not caring:

A good definition *defines away* the distinctions that don't matter.

Restriction creates capability.

A smaller world is easier to reason about.

The private act:

The code is a shadow. Your mental model is the reality. Build the model first. Name things. Describe your intuitions. Do the napkin math.

When the design is right, the code falls out of it. The water has risen and the problem has dissolved. What remains is simple, inevitable and hopefully boring (in a deeply satisfying way).