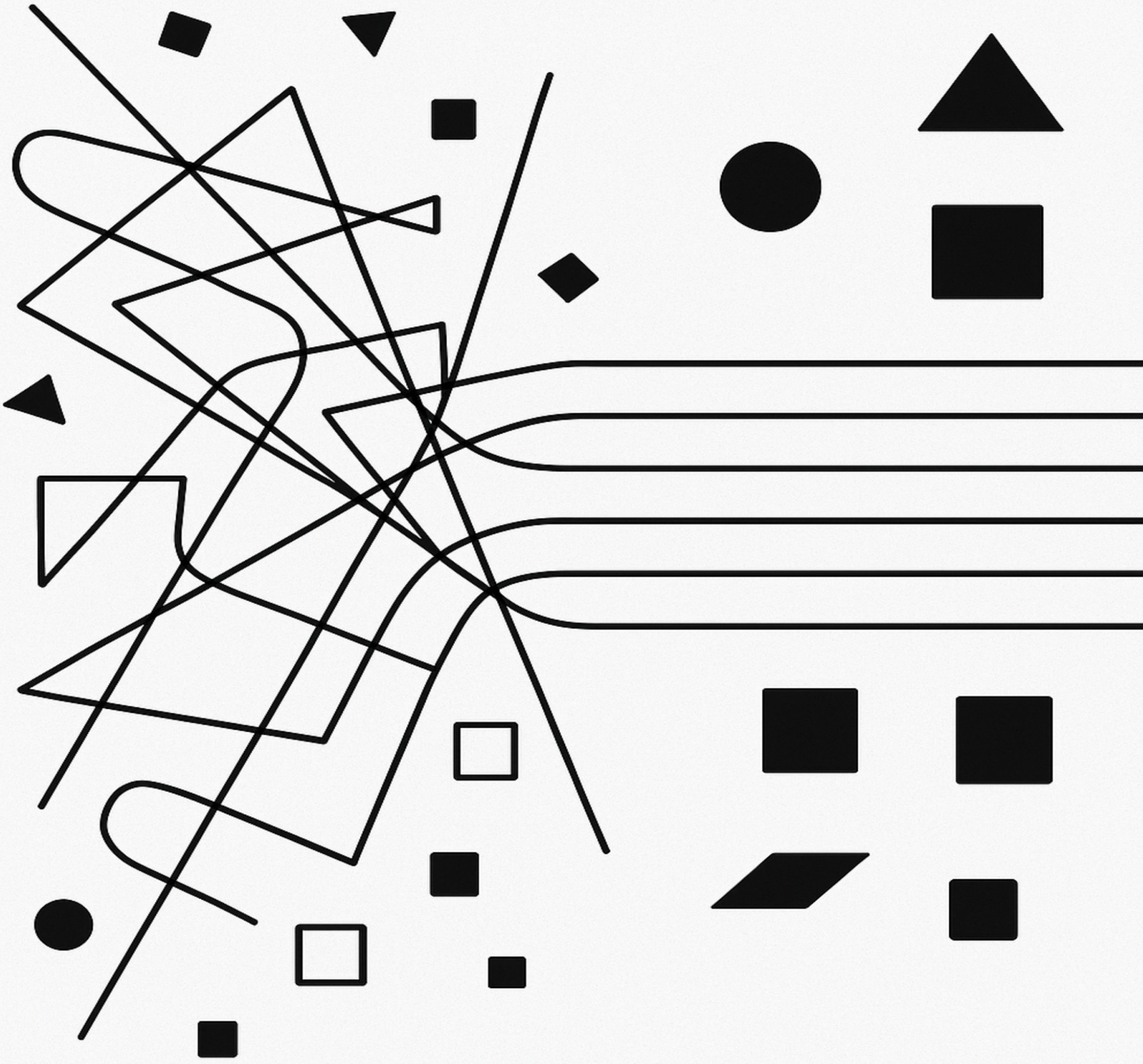
Java Backend Coding Technology

Unified Code Through Functional Composition



Sergiy Yevtushenko

Java Backend Coding Technology

Unified Code Through Functional Composition

Sergiy Yevtushenko

2025

Contents

Chapter 1: Introduction - Code Unification

Based on: JBCT v2.2.0 | **Pragmatica Core:** 0.25.0

What You'll Learn

- Why code unification matters in the AI era
- The problems JBCT solves
- The five evaluation criteria for design decisions
- Foundational concepts: side effects, composition, monads

Prerequisites: Mid-/senior Java developers comfortable with Java 21+ features (records, sealed interfaces, pattern matching), lambdas, generics, and basic functional patterns.

Code in a New Era

Software development is changing faster than ever. AI-powered code generation tools have moved from experimental novelty to daily workflow staple in just a few years. We now write code alongside - and increasingly with - intelligent assistants that can generate entire functions, refactor modules, and suggest architectural patterns. This shift creates new challenges that traditional coding practices weren't designed to handle.

Historically, code has carried a heavy burden of personal style. Every developer brings preferences about naming, structure, error handling, and abstraction. Teams spend countless hours in code review debating subjective choices. Style guides help, but they can't capture the deeper structural decisions that make code readable or maintainable. When AI generates code, it inherits these same inconsistencies - we just don't know whose preferences it's channeling or why it made particular choices.

This creates a context problem. When you read AI-generated code, you're reverse-engineering decisions made by a model trained on millions of examples with conflicting styles. When AI reads your code to suggest changes, it must infer your intentions from structure that may not clearly express them. The cognitive overhead compounds: developers burn mental cycles translating between their mental model, the code's structure, and what the AI "thinks" the code means.

Meanwhile, technical debt accumulates silently. Small deviations from good structure - a validation check here, an exception there, a bit of mixed abstraction levels - seem harmless in isolation. But they compound. Refactoring becomes risky. Testing becomes difficult. The codebase becomes a collection of special cases rather than a coherent system.

> Traditional approaches don't provide clear, mechanical rules for when to refactor or how to structure new code, so these decisions remain subjective and inconsistent.

The JBCT Approach

Java Backend Coding Technology (JBCT) proposes a different approach: **reduce the space of valid choices until there's essentially one good way to do most things**. Not through rigid frameworks or heavy ceremony, but through a small set of rules that make structure predictable, refactoring mechanical, and business logic clearly separated from technical concerns.

The benefits compound:

Unified structure means humans can read AI-generated code without guessing about hidden assumptions, and AI can read human code without inferring structure from context. A use case looks the same whether you wrote it, your colleague wrote it, or an AI assistant generated it. The structure carries the intent.

Minimal technical debt emerges naturally because refactoring rules are built into the methodology. When a function grows beyond one clear responsibility, the rules tell you exactly how to split it. When a component gets reused, there's one obvious place to move it. Debt doesn't accumulate because prevention is cheaper than cleanup.

Close business modeling happens when you're not fighting technical noise. Value objects enforce domain invariants at construction time. Use cases read like business processes because each step does one thing. Errors are domain concepts, not stack traces. Product owners can read the code structure and recognize their requirements.

Requirement discovery becomes systematic. When you structure code as validation → steps → composition, gaps become obvious. Missing validation rules surface when you define value objects. Unclear business logic reveals itself when you can't name a step clearly. Edge cases emerge when you model errors as explicit types.

Common language emerges when patterns become vocabulary. The six patterns (Leaf, Sequencer, Fork-Join, Condition, Iteration, Aspects) describe both code structure and business processes. When business says "First we verify, then we process, then we notify"—that's a Sequencer. When they say "We need profile, preferences, and history"—that's a Fork-Join. The translation is mechanical, and requirements discussions become technical design sessions.

Business logic as a readable language happens when patterns become vocabulary. The four return types, parse-don't-validate, and the fixed pattern catalog form a con-

sistent way to express domain concepts in code. Anyone who understands the domain can pick up a new codebase virtually instantly.

Deterministic code generation becomes possible when the mapping from requirements to code is mechanical. Given a use case specification - inputs, outputs, validation rules, steps - there's essentially one correct structure. Different developers (or AI assistants) should produce nearly identical implementations.

> **A Broader Movement:** JBCT is not alone in pursuing compile-time guarantees and type-driven design. Similar philosophies appear in database design (7NF type-first approaches), distributed systems, and functional programming communities. The common thread: shift errors from runtime to compile-time, make invalid states unrepresentable, and reduce cognitive load through explicit contracts.

The Five Evaluation Criteria

Before diving into patterns, understand how we evaluate every decision in JBCT. Traditional "best practices" rely on subjective "readability" - but what does that mean? JBCT uses five objective criteria:

1. **Mental Overhead** - "Don't forget to..." and "Keep in mind..." items you must track. Lower is better.
2. **Business/Technical Ratio** - Domain concepts vs framework/infrastructure noise. Higher domain visibility is better.
3. **Design Impact** - Does an approach enforce good patterns or allow bad ones? Improves consistency or breaks it?
4. **Reliability** - Does the compiler catch mistakes, or must you remember? Type safety eliminates bug classes.
5. **Complexity** - Number of elements, connections, and hidden coupling. Fewer moving parts are better.

These aren't preferences - they're measurable. When we say "don't use business exceptions," we prove it: - **Mental Overhead**: Checked exceptions pollute signatures; unchecked are invisible (+2 for Result) - **Reliability**: Exceptions bypass type checker; Result makes failures explicit (+1 for Result) - **Complexity**: Exception hierarchies create coupling (+1 for Result)

Example: Applying the Criteria

Question: Should we use `@Transactional` annotation or explicit transaction management in use cases?

Analysis using the five criteria:

1. **Mental Overhead:**

- `@Transactional`: Invisible behavior - must remember that methods run in transactions, requires understanding proxy mechanics
- Explicit: Transaction boundaries are visible in code
- **Score: +2 for explicit**

2. **Business/Technical Ratio:**
   - Both approaches are technical infrastructure
   - **Score: 0** (neutral)
3. **Design Impact:**
   - `@Transactional`: Couples business logic to Spring framework
   - Explicit: Business logic stays framework-agnostic
   - **Score: +2 for explicit**
4. **Reliability:**
   - `@Transactional`: Fails silently in some cases (private methods, self-invocation)
   - Explicit: Compiler errors if you forget transaction handling
   - **Score: +1 for explicit**
5. **Complexity:**
   - `@Transactional`: Hidden control flow
   - Explicit: Control flow is visible
   - **Score: +1 for explicit**

Verdict: Use explicit transaction management (Aspect pattern) - Total: +6 points for explicit

This is how every decision in JBCT is made—not based on opinion, but on measurable impact across five dimensions.

Foundational Concepts

What Are Side Effects?

A **side effect** is anything a function does beyond computing and returning a value:

- Writing to a database
- Making an HTTP call
- Writing to a file
- Printing to console
- Modifying a global variable
- Throwing an exception

Pure function (no side effects):

```
public int add(int a, int b) {
    return a + b;  // Only computes and returns
}
```

Impure function (has side effects):

```
public void saveUser(User user) {
    database.save(user);  // Side effect: modifies external state
    logger.info("User saved");  // Side effect: writes to log
}
```

Why care? **Pure functions are predictable**: same inputs always produce same output. They're easy to test (no mocking needed) and safe to run anywhere, anytime.

Impure functions are necessary - your app must interact with the world - but they're **unpredictable**: network might fail, disk might be full, database might be down.

JBCT's approach: **push side effects to the edges**. Keep business logic pure. Isolate impure operations in adapter leaves. This makes your core logic easy to test and reason about.

What Is Composition?

Composition means building complex operations by combining simpler ones.

Traditional imperative style:

```
public String processUser(String email) {
    String trimmed = email.trim();
    String lowercase = trimmed.toLowerCase();
    String validated = validate(lowercase);
    String saved = save(validated);
    return saved;
}
```

Functional composition:

```
public Result<String> processUser(String email) {
    return Result.success(email)
                 .map(String::trim)
                 .map(String::toLowerCase)
                 .flatMap(this::validate)
                 .flatMap(this::save);
}
```

The second version **chains** operations. Each step takes the output of the previous step as input. The data flows through a pipeline.

Why this matters: composition lets you **build complex logic from simple pieces** without intermediate variables or explicit error checking at each step. The structure itself handles error propagation.

Request Processing as Data Transformation

Every request your system handles follows the same fundamental process. It doesn't depend on language or framework. It mirrors how humans naturally solve problems:

take input, gradually collect necessary pieces of knowledge, and produce a correct answer.

The Universal Pattern:

```
Input → Parse → Gather → Process → Respond → Output
```

Each stage transforms data. Each stage may need additional data. Each stage may fail. The entire flow is a data transformation pipeline.

Why Async Looks Like Sync:

When you think in terms of data transformation, the sync/async distinction disappears:

```
// These are structurally identical
Result<User> user = database.findUser(userId);     // "sync"
Promise<User> user = httpClient.fetchUser(userId); // "async"
```

Both take a user ID, both produce a User (or failure). The only difference is *when* the result becomes available—an execution detail, not a structural concern.

Parallel Execution Becomes Transparent:

You don't decide "this should be parallel." You express data dependencies. The execution strategy follows from the structure:

```
// Sequential: each step needs previous result
return validateInput(request)
    .flatMap(this::createUser)
    .flatMap(this::sendWelcomeEmail);

// Parallel: steps are independent
return Promise.all(
    fetchUserProfile(userId),
    loadAccountSettings(userId),
    getRecentActivity(userId)
).map(this::buildDashboard);
```

The JBCT patterns—Leaf, Sequencer, Fork-Join, Condition, Iteration, Aspects—are the fundamental ways data can flow through any system. Once you start thinking in data transformation, implementing any processing task in close to optimal form becomes routine.

What Are Monads (Smart Wrappers)?

In JBCT, we use types that wrap values and control how operations are applied to them.

> **Note:** In functional programming, these are called *monads*. This book uses both terms - "Smart Wrapper" for intuition, "monad" for precision.

A monad controls when and if your operations run.

The Key Insight: Inversion of Control

Traditional code: **you** decide when to do something. Monad code: **the monad** decides when to do something.

Think: "Do this operation, **if/when the value is available**."

```
// Traditional: YOU check, YOU decide
String result;
if (email != null) {
    String trimmed = email.trim();
    if (isValid(trimmed)) {
        result = save(trimmed);
        if (result == null) {
            // Error: save failed
        }
    } else {
        // Error: invalid
    }
} else {
    // Error: null input
}

// Monad: WRAPPER checks, WRAPPER decides
Result<String> result = Result.success(email)
                              .map(String::trim)
                              .flatMap(this::validate)
                              .flatMap(this::save);
```

You're saying: "Here's what to do with the value... **if** you have one and **when** you're ready."

The monad decides: - **Option**: "I'll apply your operation **if** the value is present" - **Result**: "I'll apply your operation **if** there's no error so far" - **Promise**: "I'll apply your operation **when** the async result arrives"

Each monad has: - **map**: "Transform the value, if/when available" - **flatMap**: "Chain another operation, if/when the current one succeeds"

Pragmatic, Not Pure

JBCT uses *pragmatic* monads. Monad laws are not required. Purity is not a goal. **Predictability is.**

We borrow functional patterns because they make code more predictable and composable, not because we're pursuing theoretical purity. Side effects happen. I/O is necessary. The goal is to make side effects *explicit* and *isolated*, not to eliminate them.

If you're coming from Haskell or Scala, adjust your expectations: this is practical Java, not academic FP.

Immutability

All input data passed to operations must be treated as immutable and read-only. This isn't about dogmatic functional purity - it's about maintaining safety guarantees that make concurrent code predictable.

What MUST be immutable: - Data passed between parallel operations (Fork-Join pattern) - Input parameters to any operation - Response types returned from use cases - Value objects used as map keys or in collections

What CAN be mutable (thread-confined): - Local state within single operation (accumulators, builders) - Working objects within adapter boundaries - State confined to sequential patterns

Key principle: Mutability is safe when state is **thread-confined** (accessed by single thread). Parallel patterns require immutable inputs because no isolation exists.

Who Should Use JBCT?

You should use this if: - You're building backend services (REST APIs, microservices, batch processors) - You want code that's easy for new team members to understand - You're working with AI coding assistants and want generated code to match your structure - You value testability and want to minimize mocking - You're tired of architectural debates and want mechanical rules

This might not fit if: - You're building UI applications (different concerns, different patterns) - You need extreme performance optimization (some abstraction overhead) - Your team is heavily invested in a conflicting architecture - You prefer object-oriented design with mutable state

---

Key Takeaways

1. **JBCT unifies code** - Making code predictable across teams and AI collaboration
2. **Five criteria** guide all decisions - Mental overhead, business/technical ratio, design impact, reliability, complexity
3. **Side effects at edges** - Keep business logic pure, isolate I/O in adapters
4. **Composition over imperative** - Chain operations, let structure handle errors
5. **Monads invert control** - You describe transformations, the monad handles execution

---

Exercises

See Appendix B for exercises on: - Exercise 1.1: Return Type Selection - Exercise 1.2: Option vs Result

---

What's Next

Chapter 2 introduces the four return types that form the foundation of everything else: `T`, `Option<T>`, `Result<T>`, and `Promise<T>`. You'll learn when to use each one, how they compose, and why these four types are all you need.

Chapter 2: The Four Return Types

What You'll Learn

- Why exactly four return types are sufficient
- When to use each type: T, Option, Result, Promise
- How to convert between types
- Why Java's standard library isn't enough

Overview

Every function in JBCT returns exactly one of four types. Not "usually" or "preferably"—exactly one, always. This isn't an arbitrary restriction; it's intentional compression of complexity into type signatures.

Why by criteria: - **Mental Overhead**: Hidden error channels (exceptions), hidden optionality (null), hidden asynchrony (blocking I/O) force remembering behavior not in signatures. Explicit types eliminate this (+3). - **Reliability**: Compiler verifies error handling, null safety, and async boundaries when encoded in types (+3). - **Complexity**: Four types cover all scenarios - no guessing about combinations (+2).

T - Synchronous, Cannot Fail, Always Present

Use this when the operation is pure computation with no possibility of failure or missing data. Mathematical calculations, transformations of valid data, simple getters. If you can't think of a way this function could fail or return nothing, it returns `T`.

```
public record FullName(String value) {
    public String initials() {  // returns String (T)
        return value.chars()
                    .filter(Character::isUpperCase)
                    .collect(StringBuilder::new,
                             StringBuilder::appendCodePoint,
                             StringBuilder::append)
                    .toString();
```

```
    }
}
```

The signature `String initials()` tells you: this always succeeds, always returns a value, completes immediately.

Return `T` when: - Pure computation (e.g., `calculateTotal`, `formatCurrency`) - Transformation of already-valid data (e.g., `toUpperCase`, `extractId`) - Getters for required fields (e.g., `user.email()`, `order.total()`)

`Option<T>` - Synchronous, Cannot Fail, May Be Missing

Use this when absence is a valid outcome but failure isn't possible. Lookups that might not find anything, optional configuration, nullable database columns when null is semantically meaningful. The key: missing data is normal business behavior, not an error.

```
public interface PreferenceRepository {
    Option<Theme> findThemePreference(UserId id);  // might not be set
}
```

The signature `Option<Theme>` tells you: this always succeeds, but the value might be absent. The caller must handle both cases.

Return `Option<T>` when: - Lookup might not find anything (e.g., `findByUsername`) - Field is genuinely optional in the domain (e.g., `user.middleName()`) - "Not found" is a normal outcome, not an error

---

`Result<T>` - Synchronous, Can Fail, Business/Validation Errors

Use this when an operation might fail for business or validation reasons. Parsing input, enforcing invariants, business rules that can be violated. Failures are represented as typed `Cause` objects, not exceptions.

> **Note:** `Cause` represents domain failures or error reasons. Think of it as "FailureReason" or "DomainError"—the typed representation of why a business operation failed.

```
public record Email(String value) {
    private static final Pattern EMAIL_PATTERN =
        Pattern.compile("^[A-Za-z0-9+_.-]+@[A-Za-z0-9.-]+$");
    private static final Fn1<Cause, String> INVALID_EMAIL =
        Causes.forOneValue("Invalid email format: %s");
```

```
    public static Result<Email> email(String raw) {
        return Verify.ensure(raw, Verify.Is::present)
                     .map(String::trim)
                     .filter(INVALID_EMAIL, EMAIL_PATTERN.asMatchPredicate())
                     .map(Email::new);
    }
}
```

The signature `Result<Email>` tells you: this might fail (invalid format), completes immediately, failure is typed.

Return `Result<T>` when: - Validating input (e.g., `Email.email(raw)`) - Enforcing business rules (e.g., `checkInvariant`) - Parsing or constructing domain objects (e.g., `OrderId.orderId(raw)`)

> **Why Result: Error Handling Philosophy**
>
> Error handling logic belongs where business context exists to make decisions. Sometimes that's close to where the error occurred; sometimes the error propagates unchanged because only the caller has enough context to decide.
>
> Different mechanisms have distinct trade-offs:

Mechanism	Transparency	Ergonomics	Reliability
Checked exceptions	✓ Explicit	✗ Verbose, coupling	✓ Compiler-enforced
Unchecked exceptions	✗ Hidden	✗ Mental overhead	✗ Silent failures
Errors as values (Go)	✓ Visible	✗ Manual propagation	✗ Easy to ignore
Functional (Result)	✓ In type	✓ Monadic composition	✓ Compiler-enforced

> `Result<T>` combines the best: explicit in signature, ergonomic via `map`/`flatMap`, compiler-verified handling.

`Promise<T>` - Asynchronous, Can Fail

Use this for any I/O operation, external service call, or computation that might block. `Promise<T>` is semantically equivalent to `Result<T>` but asynchronous - failures are carried in the Promise itself, not nested inside it.

```
public interface AccountRepository {
    Promise<Account> findById(AccountId id);  // async lookup, can fail
}
```

The signature `Promise<Account>` tells you: this completes later (async), might fail (network, database), failure is carried in the Promise.

> **Promise as Async Result**
>
> Think of `Promise<T>` as the asynchronous counterpart to `Result<T>`. Both represent operations that can succeed or fail with typed errors. The only difference is timing: `Result<T>` completes immediately, `Promise<T>` completes later. The same `map`/`flatMap` patterns work identically; converting is trivial (`result.async()` lifts to Promise, `promise.await()` blocks to Result). When you understand `Result<T>`, you understand `Promise<T>`.

Promise Resolution and Thread Safety:

Promise resolution is **thread-safe** and happens **exactly once**:

- Multiple threads can attempt resolution - only the first succeeds
- Resolution serves as synchronization point
- Transformations execute after resolution
- Side effects execute independently

```
var promise = Promise.<User>promise();

// Multiple threads racing to resolve - only first wins
executor.submit(() -> promise.succeed(user1));  // First to resolve
executor.submit(() -> promise.succeed(user2));  // Ignored

// All transformations see the same result (user1)
promise.map(this::processUser)
       .flatMap(this::saveToDatabase)
       .onSuccess(this::logSuccess);
```

Return `Promise<T>` when: - Any I/O operation (database, HTTP, file system) - External service calls - Operations that might block or take time

---

Why Exactly Four?

These four types form a complete basis for composition. You can lift "up" when needed (`Option` to `Result` to `Promise`), but you never nest the same concern twice.

Each type represents one orthogonal concern: - **Synchronous vs. asynchronous**: now vs. later - **Can fail vs cannot fail**: error channel present or absent - **Value vs optional value**: presence guaranteed or not

Decision table:

Sync?	Can Fail?	May Be Absent?	Return Type
Yes	No	No	`T`
Yes	No	Yes	`Option<T>`

Sync?	Can Fail?	May Be Absent?	Return Type
Yes	Yes	No	`Result<T>`
Async	Yes	No	`Promise<T>`

Return Type Matrix

Allowed Return Types

Type	Use Case
`T`	Synchronous, cannot fail, always present
`Option<T>`	Synchronous, cannot fail, might be absent
`Result<T>`	Synchronous, can fail
`Promise<T>`	Asynchronous, can fail
`Result<Option<T>>`	Optional value that can fail validation
`Promise<Option<T>>`	Async lookup that might not find anything

Discouraged

Type	Why Discouraged
`Optional<T>`	Use `Option<T>` for consistency
`CompletableFuture<T>`	Use `Promise<T>` for consistent error handling
Framework-specific types (`Mono<T>`, `ResponseEntity<T>`)	Keep business logic framework-agnostic

Forbidden (Double-Monad Nesting)

Type	Why Forbidden
`Promise<Result<T>>`	`Promise` already carries failures - double error channel
`Result<Result<T>>`	Nested failures create unwrapping ceremony
`Option<Option<T>>`	Nested optionality is meaningless
`Promise<Option<Result<T>>>`	Triple nesting - architectural smell

Rule: Each monadic concern (optionality, failure, asynchrony) appears at most once in a return type.

Why Not Java's Built-in Types?

"Can't I just use `Optional`, `CompletableFuture`, and exceptions?"

You could, but you'd hit these problems:

Java Approach	Problem	JBCT Solution
`return null`	Hidden optionality → NPE at runtime	`Option<T>` explicit in type
`Optional<T>`	Can't represent failures	`Result<T>` for typed errors
`try-catch`	Invisible control flow	`Result<T>` - errors as values
`CompletableFuture<T>`	Complex error handling	`Promise<T>` consistent patterns

Key differences: - **Explicit in types**: Signature tells you failure modes, no hidden exceptions - **Consistent composition**: `map`/`flatMap` work the same across all types - **No nesting**: One concern per type level - **Better inference**: AI can generate correct error handling from types alone

---

Type Conversions

Lifting: Moving Between Types

You can lift a "lower" type into a "higher" one:

```
// T → Option/Result/Promise
Option.option(value)
Result.success(value)
Promise.success(value)

// Option → Result/Promise
option.toResult(cause)
option.async(cause)

// Result → Promise
result.async()
```

Example:

```
public Promise<Response> execute(Request request) {
    return ValidRequest.validRequest(request)
                       .async()  // Result → Promise
                       .flatMap(step1::apply)
```

```
                        .flatMap(step2::apply);
}
```

Forbidden Nesting: `Promise<Result<T>>`

`Promise<Result<T>>` is forbidden. `Promise<T>` already carries failures - nesting `Result` inside creates two error channels.

Wrong:

```
Promise<Result<User>> loadUser(UserId id) { /* ... */ }

// Caller must unwrap twice - absurd ceremony
loadUser(id)
    .flatMap(resultUser -> resultUser.fold(
        Cause::promise,
        user -> Promise.success(user)
    ));
```

Right:

```
Promise<User> loadUser(UserId id) { /* ... */ }

// Caller just chains
return loadUser(id).flatMap(nextStep);
```

Allowed Nesting: `Result<Option<T>>`

`Result<Option<T>>` is permitted sparingly for "optional value that can fail validation."

```
Result<Option<ReferralCode>> refCode = ReferralCode.referralCode(input);
// Success(None) = not provided, valid
// Success(Some(code)) = provided and valid
// Failure(cause) = provided but invalid
```

Use `Verify.ensureOption()` (Pragmatica Core 0.9.0+) to implement this pattern:

```
public static Result<Option<ReferralCode>> referralCode(String raw) {
    return Verify.ensureOption(
        Option.option(raw).map(String::trim).filter(s -> !s.isEmpty()),
        PATTERN.asMatchPredicate(),
        INVALID_FORMAT
    ).map(opt -> opt.map(ReferralCode::new));
}
```

API Quick Reference

Type Conversions

```
// Option conversions
option.toResult(cause)      // Option → Result
option.async(cause)         // Option → Promise
option.toOptional()         // Option → Java Optional

// Result conversions
result.async()              // Result → Promise
result.option()             // Result → Option (loses error)

// Promise conversions
promise.await()             // Promise → Result (blocks)
promise.await(timeout)      // Promise → Result (with timeout)

// Cause conversions (preferred)
cause.result()              // Cause → Result failure
cause.promise()             // Cause → Promise failure
```

Creating Instances

```
// Option
Option.some(value)
Option.none()
Option.option(nullable)     // null → none

// Result
Result.success(value)
cause.result()              // Preferred for failure

// Promise
Promise.success(value)
cause.promise()             // Preferred for failure
Promise.promise()           // Unresolved
```

Key Takeaways

1. **Four types cover all cases** - T, Option, Result, Promise
2. **Signatures tell everything** - failure modes, optionality, synchrony
3. **Lift up, never nest** - Convert to higher types, never `Promise<Result<T>>`
4. **Consistent composition** - Same map/flatMap across all types

5. **Use Cause methods** - Prefer `cause.result()` over `Result.failure(cause)`

When `void` Is the Right Return Type

The four return kinds cover cases where the caller uses the result. But there's a distinct category where **no caller ever inspects the outcome**: fire-and-forget side effects.

Two legitimate cases:

1. **API Conformance** — An external API requires `void` (JDK's `Runnable`, framework event handlers)
2. **Fire-and-Forget Side Effects** — The caller deliberately discards the outcome (metrics, event publishing, cache warming)

```
// Fire-and-forget — caller doesn't care about outcome
void recordLatency(String operation, Duration elapsed);
void publishDomainEvent(OrderPlaced event);
```

The `void` return type is a **semantic signal**: errors are handled internally, never propagated to the caller. Compare:

- `Result<Unit>` — failure matters (e.g., `deleteUser`)
- `Promise<Unit>` — async, failure matters (e.g., `sendEmail`)
- `void` — failure is irrelevant to caller (e.g., `recordMetric`)

Note: The `Void` *type parameter* (as in `Result<Void>`) remains forbidden — use `Unit`. The `void` *return type* is the fire-and-forget signal.

Exercises

See Appendix B for exercises on: - Exercise 1.3: Type Lifting - Exercise 1.4: Cause Creation - Exercise 1.5: Pragmatica Core Operations

---

What's Next

Chapter 3 covers Pragmatica Core - the library that provides these four types. You'll learn the API in depth and see common usage patterns.

Chapter 4: Parse, Don't Validate

What You'll Learn

- Why validation should be inseparable from construction
- How to use factory methods with `Result<T>` returns
- Cross-field validation techniques
- Real-world validation scenarios
- How to adopt this incrementally in existing codebases

Prerequisites: Chapter 2: The Four Return Types

The Principle

Most Java code validates data after construction. You create an object with raw values, then call a `validate()` method that might throw exceptions or return error lists. This approach is backwards.

The principle: Make invalid states unrepresentable. If construction succeeds, the object is valid by definition. Validation is parsing - converting untyped or weakly-typed input into strongly typed domain objects that enforce invariants at the type level.

Why by criteria: - **Mental Overhead**: No "remember to validate" - type system guarantees validity (+2) - **Reliability**: Compiler enforces that invalid objects cannot be constructed (+3) - **Design Impact**: Business invariants concentrated in factories, not scattered (+2) - **Complexity**: Single validation point per type eliminates redundant checks (+1)

The Traditional (Wrong) Approach

```
// DON'T: Validation separated from construction
public class Email {
    private final String value;

    public Email(String value) {
```

```
        this.value = value;  // accepts anything
    }

    public boolean isValid() {  // Caller must remember to check
        return value != null && value.matches("^[A-Za-z0-9+_.-]+@[A-Za-z0-9.-]+$");
    }
}

// Client code must validate manually:
Email email = new Email(input);
if (!email.isValid()) {
    throw new ValidationException("Invalid email");
}
```

Problems: - You can construct invalid `Email` objects - Validation is a separate step that callers might forget - The `isValid()` method returns a boolean, discarding information about what's wrong - You can't distinguish "null" from "malformed" from "too long"

The Parse-Don't-Validate Approach

```
// DO: Validation IS construction
public record Email(String value) {
    private static final Pattern EMAIL_PATTERN =
        Pattern.compile("^[A-Za-z0-9+_.-]+@[A-Za-z0-9.-]+$");
    private static final Fn1<Cause, String> INVALID_EMAIL =
        Causes.forOneValue("Invalid email format: %s");

    public static Result<Email> email(String raw) {
        return Verify.ensure(raw, Verify.Is::present)
                     .map(String::trim)
                     .filter(INVALID_EMAIL, EMAIL_PATTERN.asMatchPredicate())
                     .map(Email::new);
    }
}

// Client code gets the Result:
Result<Email> result = Email.email(input);
// If this is a Success, the Email is valid. Guaranteed.
```

The constructor is private (or package-private). The only way to get an `Email` is through the static factory `email()`, which returns `Result<Email>`. If you have an `Email` instance, it's valid - no separate check needed.

> **Note:** As of current Java versions, records do not support declaring the canonical constructor as private. Rely on team discipline and code review

to ensure value objects are only constructed through their factory methods. Direct `new Email(...)` calls stand out immediately and are easy to catch.

Library Value Objects: For common types like email, URL, and UUID, Pragmatica Core provides production-ready implementations in `org.pragmatica.lang.vo`. The examples in this chapter demonstrate the *pattern* — for production use, prefer the library VOs over hand-rolled versions.

Naming Conventions

Factory naming: Factories are always named after their type, lowercase-first (camelCase):

```
Email.email(raw)
Password.password(raw)
AccountId.accountId(raw)
```

This creates a natural, readable call site that's grep-friendly and allows static imports.

Validated input naming: Use the `Valid` prefix for types representing validated inputs:

```
// DO: Use Valid prefix
record ValidRequest(Email email, Password password) { ... }
record ValidUser(Email email, HashedPassword hashed) { ... }

// DON'T: Use Validated prefix (too verbose)
record ValidatedRequest(...)
```

Optional Fields with Validation

What if a field is optional but must be valid when present? Use `Result<Option<T>>` with `Verify.ensureOption()`:

```
public record ReferralCode(String value) {
    private static final Pattern PATTERN = Pattern.compile("^[A-Z0-9]{6}$");
    private static final Cause INVALID_FORMAT = Causes.cause("Invalid referral code forma

    public static Result<Option<ReferralCode>> referralCode(String raw) {
        return Verify.ensureOption(
            Option.option(raw).map(String::trim).filter(s -> !s.isEmpty()),
            PATTERN.asMatchPredicate(),
            INVALID_FORMAT
        ).map(opt -> opt.map(ReferralCode::new));
```

```
    }
}
```

The `Verify.ensureOption()` method (Pragmatica Core 0.9.0+) handles this pattern elegantly: - If the `Option` is empty, succeeds with `Option.none()` - no validation needed - If present and valid, succeeds with `Option.some(value)` - If present and invalid, fails with the provided cause

Caller semantics: - `Failure(cause)`: Invalid input (provided but doesn't match pattern) - `Success(None)`: No value provided (valid state) - `Success(Some(code))`: Valid code provided

---

Normalization in Factories

Factories can normalize input as part of parsing:

```
public static Result<Email> email(String raw) {
    return Verify.ensure(raw, Verify.Is::present)
                 .map(String::trim)           // Remove whitespace
                 .map(String::toLowerCase)    // Lowercase for comparison
                 .filter(INVALID_EMAIL, EMAIL_PATTERN.asMatchPredicate())
                 .map(Email::new);
}
```

Now all `Email` instances are trimmed and lowercased. Domain logic never worries about case or whitespace.

Where Validation Belongs

Clear rule for validation placement:

Validation Type	Where	Example
Single-field invariants	Value object factory	Email format, password strength, ID format
Cross-field invariants	ValidRequest factory	Password doesn't contain email, date range valid
External state invariants	Use case step	Email uniqueness (DB), credit check (external service)

Rationale: - Value objects enforce invariants that depend only on the field's own value

- ValidRequest enforces invariants that require multiple fields but no external state - Use cases handle invariants that require external lookups (database, services)

```
// Value object: single-field invariant
public record Email(String value) {
    public static Result<Email> email(String raw) { /* format check only */ }
}

// ValidRequest: cross-field invariant
public record ValidRegistration(Email email, Password password) {
    public static Result<ValidRegistration> validRegistration(Email email, Password pwd)
        // Check password doesn't contain email local part
    }
}

// Use case step: external state invariant
interface CheckEmailUnique {
    Promise<Email> apply(Email email);  // DB lookup
}
```

Cross-Field Validation with Result.all()

Use `Result.all()` to validate independent fields, then add cross-field rules:

Example: Password must not contain email local part

```
record ValidRegistration(Email email, Password password) {
    private static final Cause PASSWORD_CONTAINS_EMAIL =
        Causes.cause("Password cannot contain email local part");

    static Result<ValidRegistration> validRegistration(String emailRaw,
                                                       String passwordRaw) {
        return Result.all(Email.email(emailRaw),
                          Password.password(passwordRaw))
                     .flatMap(ValidRegistration::checkPasswordNotContainsEmail);
    }

    private static Result<ValidRegistration> checkPasswordNotContainsEmail(
            Email email, Password pwd) {
        String localPart = email.value().split("@")[0];
        return pwd.value().contains(localPart)
            ? PASSWORD_CONTAINS_EMAIL.result()
            : Result.success(new ValidRegistration(email, pwd));
    }
}
```

Pattern: 1. Validate individual fields with `Result.all()` → accumulates per-field er-

rors 2. Use `.flatMap()` to add cross-field validation → fail-fast on cross-field rules 3. Extract cross-field logic to named methods 4. Only construct if all validation passes

---

Real-World Validation Scenarios

Date Range Validation

```
public record DateRange(LocalDate start, LocalDate end) {
    private static final Fn1<Cause, LocalDate> END_BEFORE_START =
        date -> Causes.cause("End date must be after start date: " + date);

    public static Result<DateRange> dateRange(LocalDate start, LocalDate end) {
        return Verify.ensure(start, Verify.Is::notNull)
                     .flatMap(_ -> Verify.ensure(end, Verify.Is::notNull))
                     .flatMap(_ -> Verify.ensure(end, isAfter(start), END_BEFORE_START))
                     .map(_ -> new DateRange(start, end));
    }

    private static Predicate<LocalDate> isAfter(LocalDate start) {
        return end -> end.isAfter(start);
    }
}
```

Business Rule Validation

```
public record ValidOrder(OrderId id, Money total, List<LineItem> items) {
    private static final Fn1<Cause, Money> TOTAL_MISMATCH =
        Causes.forOneValue("Order total does not match line items. Expected: %s");

    public static Result<ValidOrder> validOrder(OrderId id,
                                                Money total,
                                                List<LineItem> items) {
        return total.equals(calculateTotal(items))
            ? Result.success(new ValidOrder(id, total, items))
            : TOTAL_MISMATCH.apply(calculateTotal(items)).result();
    }

    private static Money calculateTotal(List<LineItem> items) {
        return items.stream()
                    .map(LineItem::subtotal)
                    .reduce(Money.ZERO, Money::add);
    }
}
```

Collecting Multiple Errors

```
public record ValidRegistration(Email email, Password password, Age age) {
    public static Result<ValidRegistration> validate(String emailRaw,
                                                     String passwordRaw,
                                                     String ageRaw) {
        return Result.all(Email.email(emailRaw),
                          Password.password(passwordRaw),
                          Age.age(ageRaw))
                     .map(ValidRegistration::new);
        // If any field fails, Result.all() accumulates ALL errors
        // User sees "Invalid email AND password too short AND age out of range"
    }
}
```

Pragmatica Core Validation Utilities

Verify.Is Predicates:

```
// Instead of custom lambdas:
.flatMap(s -> s.length() >= 8 ? Result.success(s) : Result.failure(...))

// Use filter with standard predicates:
.filter(TOO_SHORT, s -> Verify.Is.lenBetween(s, 8, 128))
```

Common predicates: `notNull`, `notBlank`, `lenBetween`, `matches`, `positive`, `nonNegative`, `between`, `greaterThan`, `lessThan`, `contains`.

Parse Subpackage:

```
import org.pragmatica.lang.parse.Number;
import org.pragmatica.lang.parse.DateTime;
import org.pragmatica.lang.parse.Network;

Number.parseInt(raw)              // Result<Integer>
DateTime.parseLocalDate(raw)      // Result<LocalDate>
Network.parseUUID(raw)            // Result<UUID>
```

Example:

```
public record Age(int value) {
    public static Result<Age> age(String raw) {
        return Number.parseInt(raw)
                     .filter(Causes.cause("Age 0-150"),
                             v -> Verify.Is.between(v, 0, 150))
                     .map(Age::new);
    }
```

```
}
```

Adopting Incrementally

Don't refactor everything at once. Adopt incrementally at boundaries.

1. **New features first** - Use parse-don't-validate for all new code
2. **Keep existing validation at controller boundaries:**

```
// BEFORE: Spring controller with @Valid
@PostMapping("/register")
public ResponseEntity<?> register(@Valid @RequestBody RegistrationRequest dto) {
    var request = new RegisterUser.Request(dto.email(), dto.password());
    return registerUser.execute(request)
                       .fold(this::errorResponse, this::successResponse);
}

// AFTER: Fully migrated
@PostMapping("/register")
public ResponseEntity<?> register(@RequestBody RegisterUser.Request raw) {
    return registerUser.execute(raw)
        .fold(this::errorResponse, this::successResponse);
}
```

3. **Gradually move validation from services to value objects:** - Find a service method with manual validation - Extract that validation into a value object factory - Update callers to use the value object - Repeat

---

Key Takeaways

1. **Validation IS construction** - If an instance exists, it's valid
2. **Factory methods return `Result<T>`** - Success means valid object
3. **`Result.all()` accumulates errors** - Show all problems at once
4. **Normalization in factories** - Trim, lowercase, etc. happen once
5. **`Result<Option<T>>`** - For optional values that must validate when present
6. **Adopt incrementally** - Start at boundaries, work inward

---

Exercises

See Appendix B for exercises on: - Exercise 2.1: PhoneNumber value object - Exercise 2.2: DateRange aggregate validation - Exercise 2.3: Error accumulation vs short-

circuit

What's Next

Chapter 5 covers error handling - how to define typed errors, compose them, and handle failures cleanly without exceptions.

Chapter 7: Basic Patterns & Structure

What You'll Learn

- When to extract functions (mechanical rules, not judgment calls)
- How to keep code at a single level of abstraction
- The three basic patterns: Leaf, Condition, Iteration
- Zone-based naming conventions

Prerequisites: Chapter 6: Null Policy & Error Recovery

Single Pattern Per Function

Every function implements exactly one pattern from a fixed catalog: Leaf, Sequencer, Fork-Join, Condition, or Iteration. (Aspects are the exception—they decorate other patterns.) Each pattern maps directly to a BPMN flow construct:

Pattern	BPMN Construct	Structural Role
Leaf	Task / Service Task	Atomic operation, one responsibility
Sequencer	Sequence Flow	Dependent steps in order
Fork-Join	Parallel Gateway	Independent concurrent operations
Condition	Exclusive Gateway	Routing, no transformation
Iteration	Multi-Instance Activity	Collection processing
Aspects	Event Sub-Process	Cross-cutting concerns wrapping logic

This is not a metaphor — JBCT grew from functional programming, BPMN grew from business process modeling, and they converged because they describe the same thing: how work flows through a system. If you can draw it as a BPMN diagram, you can write it as JBCT code. The structure is the same.

Why? Cognitive load. When reading a function, you should recognize its shape immediately. If it's a Sequencer, you know it chains dependent steps linearly. If it's

Fork-Join, you know it runs independent operations and combines results. Mixing patterns within a function creates mixed abstraction levels and forces readers to hold multiple mental models simultaneously.

This rule has a mechanical benefit: it makes refactoring deterministic. When a function grows beyond one pattern, you extract the second pattern into its own function. There's no subjective judgment about "is this too complex?" — if you're doing two patterns, split it. In BPMN terms: each method is one Task, one Gateway, or one Sub-Process — not a mix.

Why by criteria: - **Mental Overhead**: One pattern per function means immediate recognition - no mental model switching (+2) - **Complexity**: Mechanical refactoring rule eliminates subjective debates about "too complex" (+2) - **Design Impact**: Forces proper abstraction layers - no mixing orchestration with computation (+2)

The Pain of Mixed Patterns

Before showing violations, understand the **concrete problems** that mixing patterns causes:

Testing becomes brittle:

```
// Mixed pattern function
public Result<Report> generateReport(ReportRequest request) {
    // Validates, fetches in parallel, computes, formats - all in one
}

// To test, you need:
// 1. Valid request setup
// 2. Mock for fetchUserData
// 3. Mock for fetchSalesData
// 4. Verify computeMetrics logic
// 5. Verify formatReport logic
// Can't test parts independently - it's all or nothing
```

Debugging is unclear:

```
// Stack trace points to generateReport() line 45
// But which step failed? Validation? Fork-Join fetch? Compute? Format?
// You can't tell without stepping through the whole function
```

Reuse is impossible:

```
// Another use case needs the same Fork-Join fetch logic
// But it's buried inside generateReport()
// Can't reuse it - have to copy-paste or refactor
```

With single pattern per function: - Each function is independently testable - Patterns are reusable across use cases - Failures are localized (stack trace says "fetchReportData failed") - Structure is predictable and scannable

Example Violation

```
// DON'T: Mixing Sequencer and Fork-Join
public Result<Report> generateReport(ReportRequest request) {
    return ValidRequest.validRequest(request)
                       .flatMap(valid -> {
                           // Sequencer starts here
                           var userData = fetchUserData(valid.userId());
                           var salesData = fetchSalesData(valid.dateRange());

                           // Wait, now we're doing Fork-Join?
                           return Result.all(userData, salesData)
                                        .flatMap((user, sales) -> computeMetrics(user, sa
                                        .flatMap(this::formatReport);  // Back to Sequenc
                       });
}
```

This function starts as a Sequencer (validate -> fetch -> compute -> format), but `fetchUserData` and `fetchSalesData` are independent, so we suddenly do a Fork-Join in the middle. Mixed abstraction levels. Hard to test. Unclear at a glance what the function does.

Corrected

```
// DO: One pattern per function
public Result<Report> generateReport(ReportRequest request) {
    return ValidRequest.validRequest(request)
                       .flatMap(this::fetchReportData)
                       .flatMap(this::computeMetrics)
                       .flatMap(this::formatReport);
}

private Result<ReportData> fetchReportData(ValidRequest request) {
    // This function is a Fork-Join
    return Result.all(fetchUserData(request.userId()),
                      fetchSalesData(request.dateRange()))
                 .map(ReportData::new);
}
```

Now `generateReport` is a pure Sequencer (validate -> fetch -> compute -> format), and `fetchReportData` is a pure Fork-Join. Each function has one clear job.

Mechanical Refactoring

If you're writing a Sequencer and realize step 3 needs to do a Fork-Join internally, extract step 3 into its own function that implements Fork-Join. The original Sequencer stays clean.

Rule of thumb: One pattern per function. If you see two patterns, extract one.

Single Level of Abstraction

The rule: No complex logic inside lambdas. Lambdas passed to `map`, `flatMap`, and similar combinators may contain only: - Method references (e.g., `Email::new`, `this::processUser`) - Single method calls with parameter forwarding (e.g., `param -> someMethod(outerParam, param)`)

Why? Lambdas are composition points, not implementation locations. When you bury logic inside a lambda, you hide abstraction levels and make the code harder to read, test, and reuse. Extract complex logic to named functions - the name documents intent, the function becomes testable in isolation, and the composition chain stays flat and readable.

Why by criteria: - **Mental Overhead**: Flat composition chains scan linearly - no descending into nested logic (+2) - **Business/Technical Ratio**: Named functions document intent; anonymous lambdas hide it (+2) - **Complexity**: Each function testable in isolation; buried lambda logic requires testing through container (+2)

Anti-Pattern

```
// DON'T: Complex logic inside lambda
return fetchUser(userId)
    .flatMap(user -> {
        if (user.isActive() && user.hasPermission("admin")) {
            return loadAdminDashboard(user)
                .map(dashboard -> {
                    var summary = new Summary(
                        dashboard.metrics(),
                        dashboard.alerts().stream()
                            .filter(Alert::isUrgent)
                            .toList()
                    );
                    return new Response(user, summary);
                });
        } else {
            return AccessError.InsufficientPermissions.INSTANCE.promise();
        }
    });
```

This lambda contains: conditional logic, nested map, stream processing, object construction. Mixed abstraction levels. Hard to test. Hard to read.

Correct Approach

```
// DO: Extract to named functions
return fetchUser(userId)
    .flatMap(this::checkAdminAccess)
    .flatMap(this::loadAdminDashboard)
    .map(this::buildResponse);

private Promise<User> checkAdminAccess(User user) {
    return isActiveAdministrator(user)
            ? Promise.success(user)
            : AccessError.InsufficientPermissions.INSTANCE.promise();
}

private boolean isActiveAdministrator(User user) {
    return user.isActive() && user.hasPermission("admin");
}

private Response buildResponse(Dashboard dashboard) {
    var urgentAlerts = filterUrgentAlerts(dashboard.alerts());
    var summary = new Summary(dashboard.metrics(), urgentAlerts);
    return new Response(dashboard.user(), summary);
}

private List<Alert> filterUrgentAlerts(List<Alert> alerts) {
    return alerts.stream()
                 .filter(Alert::isUrgent)
                 .toList();
}
```

Now the top-level chain reads linearly: fetch -> check access -> load dashboard -> build response. Each step is named, testable, and at a single abstraction level.

Allowed in Lambdas

Method references:

```
.map(Email::new)
.flatMap(this::saveUser)
.map(User::id)
```

Single method call with parameter forwarding:

```
.flatMap(user -> checkPermissions(requiredRole, user))
.map(order -> calculateTotal(taxRate, order))
```

Forbidden in Lambdas

- Conditionals (`if`, ternary, `switch`)

- Try-catch blocks
- Multi-statement blocks
- Object construction beyond simple factory calls

No ternaries (they are the Condition pattern):

```
// DON'T: Ternary in lambda
.flatMap(user -> user.isPremium()
    ? applyPremiumDiscount(user)
    : applyStandardDiscount(user))

// DO: Extract to named function
.flatMap(this::applyApplicableDiscount)

private Result<Discount> applyApplicableDiscount(User user) {
    return user.isPremium()
        ? applyPremiumDiscount(user)
        : applyStandardDiscount(user);
}
```

The Three-Zone Framework

Maintaining "single level of abstraction" becomes mechanical when you think of your codebase in three distinct zones, each with its own vocabulary:

Zone 1 (Use Case Level) - High-level business goals: - `RegisterUser.execute()`, `ProcessOrder.execute()`, `LoadDashboard.execute()` - One Zone 1 function per use case - the entry point

Zone 2 (Orchestration Level) - Coordinating steps that break down the goal: - Step interfaces in Sequencer/Fork-Join patterns - Verbs: `validate`, `process`, `handle`, `transform`, `apply`, `check`, `load`, `save`, `manage`, `configure`, `initialize` - Examples: `ValidateInput.apply()`, `ProcessPayment.apply()`, `HandleNotification.apply()`

Zone 3 (Implementation Level) - Concrete technical operations: - Business and adapter leaves - Verbs: `get`, `set`, `fetch`, `parse`, `calculate`, `convert`, `hash`, `format`, `encode`, `decode`, `extract`, `split`, `join`, `log`, `send`, `receive`, `read`, `write`, `add`, `remove` - Examples: `hashPassword()`, `parseJson()`, `fetchFromDatabase()`, `calculateTax()`

The key insight: Functions at each zone should only call functions from the same zone or one level down. Zone 2 functions call other Zone 2 steps or Zone 3 leaves. Zone 3 leaves perform atomic operations. This creates natural layering.

Example - Maintaining zone consistency:

```
// GOOD - All steps at Zone 2 (orchestration level)
public Promise<Response> execute(Request request) {
    return ValidRequest.validRequest(request)              // Zone 2: validate
                       .async()
```

```
                        .flatMap(this::processCredentials)  // Zone 2: process
                        .flatMap(this::saveUser)            // Zone 2: save
                        .flatMap(this::sendConfirmation);   // Zone 2: send
}

// BAD - Mixing Zone 2 and Zone 3 in same chain
public Promise<Response> execute(Request request) {
    return ValidRequest.validRequest(request)               // Zone 2
                        .async()
                        .flatMap(this::hashPassword)        // Zone 3 - too specific!
                        .flatMap(this::saveUser)            // Zone 2
                        .flatMap(this::fetchConfirmToken);  // Zone 3 - too specific!
}
```

In the bad example, `hashPassword` and `fetchConfirmToken` are Zone 3 operations (concrete technical details). They should be wrapped in Zone 2 steps like `processCredentials` and `sendConfirmation`.

The Stepdown Rule Test

A simple way to verify your abstraction levels: read your code aloud by adding "to" before each function. It should sound like a natural narrative:

```
// Reading this aloud:
// "To execute, we validate the request,
//  then we process payment,
//  then we send confirmation."
return ValidRequest.validRequest(request)
                   .async()
                   .flatMap(this::processPayment)
                   .flatMap(this::sendConfirmation);
```

If adding "to" makes it sound awkward or overly detailed ("to hash password, then to save to database, then to fetch from cache"), you're mixing abstraction levels.

Gap Detection Through Patterns

When you model business processes using patterns, **gaps become visible**. The patterns validate requirements, not just implement them.

How patterns reveal gaps: - **Missing validation:** Building a Sequencer but nothing validates the input before step 1? Gap found. - **Unclear dependencies:** Are these a Sequencer (dependent) or Fork-Join (independent)? If unknown, process isn't defined. - **Missing error handling:** Every Leaf can fail. What happens when this fails? - **Inefficient flows:** Sequential process described but steps are independent? Should be Fork-Join.

Discovery questions by pattern:

Pattern	Key Questions
Leaf	What does it do? Can it fail? Sync or async?
Condition	What determines the path? Mutually exclusive? Default?
Iteration	Stop on first failure? Order matters? Can parallelize?

Advanced patterns (Sequencer, Fork-Join, Aspects) covered in Chapter 8.

Pattern: Leaf

Definition: A **Leaf** (an atomic operation with no substeps) is the smallest unit of processing - a function that does one thing and has no internal steps. It's either a business leaf (pure computation) or an adapter leaf (I/O or side effects).

Rationale (by criteria): - **Mental Overhead**: Atomic operations have no internal steps to track - immediate comprehension (+2) - **Business/Technical Ratio**: Business leaves are pure domain logic; adapter leaves isolate technical concerns (+2) - **Complexity**: Single responsibility per leaf - no hidden interactions (+2) - **Reliability**: Pure business leaves are deterministic and easily testable (+1)

Business Leaves

Business leaves are pure functions that transform data or enforce business rules:

```
// Simple calculation leaf
public static Price calculateDiscount(Price original, Percentage rate) {
    return original.multiply(rate);
}

// Domain rule enforcement leaf
public static Result<Unit> checkInventory(Product product, Quantity requested) {
    return product.availableQuantity().isGreaterThanOrEqual(requested)
        ? Result.unitResult()
        : InsufficientInventory.cause(product.id(), requested);
}

// Data transformation leaf
public static OrderSummary toSummary(Order order) {
    return new OrderSummary(order.id(),
                            order.totalAmount(),
                            order.items().size());
}
```

If there's no I/O and no side effects, it's a business leaf. Keep each leaf focused on one transformation or one business rule.

Adapter Leaves

Adapter leaves integrate with external systems: databases, HTTP clients, message queues, file systems. They map foreign errors to domain Causes.

```
public interface UserRepository {
    Promise<Option<User>> findByEmail(Email email);
}

// Adapter leaf implementation
class PostgresUserRepository implements UserRepository {
    private final DataSource dataSource;

    public Promise<Option<User>> findByEmail(Email email) {
        return Promise.lift(RepositoryError.DatabaseFailure::new,
                            () -> {
                                try (var conn = dataSource.getConnection();
                                     var stmt = conn.prepareStatement(
                                         "SELECT * FROM users WHERE email = ?")) {

                                    stmt.setString(1, email.value());
                                    var rs = stmt.executeQuery();

                                    return rs.next() ? mapUser(rs) : null;
                                }
                            }).map(Option::option);
    }

    private User mapUser(ResultSet rs) throws SQLException {
        return new User(/* ... */);
    }
}
```

The adapter catches `SQLException` and wraps it in `RepositoryError.DatabaseFailure`, a domain `Cause`. Callers never see `SQLException`.

Thread Safety

Leaf operations are **thread-safe through confinement** - each invocation operates independently with its own local state. Mutable local variables (accumulators, builders, working objects) are safe within a leaf because they never escape the function scope. Input parameters must be treated as read-only.

Placement Rules

If a leaf is only used by one caller, keep it nearby (same file, same package).

If it's reused, move it immediately to the nearest `shared` package. Don't defer - tech debt accumulates when shared code stays in wrong locations.

Pattern: Condition

Definition: Condition represents branching logic based on data. The key: express conditions as values, not control-flow side effects. Keep branches at the same abstraction level.

Rationale (by criteria): - **Mental Overhead**: Conditions as expressions - evaluates to single value, not control flow scatter (+2) - **Business/Technical Ratio**: Branch logic mirrors domain rules, not imperative jumps (+2) - **Complexity**: Same abstraction level per branch - prevents tangled logic (+2) - **Reliability**: Type-checked branches ensure all cases return compatible types (+2)

Simple Conditional

```
// DO: Condition as expression returning the monad
Result<Discount> calculateDiscount(Order order) {
    return order.isPremiumUser()
        ? premiumDiscount(order)      // returns Result<Discount>
        : standardDiscount(order);    // returns Result<Discount>
}
```

Both branches return the same type (`Result<Discount>`), so the ternary is just choosing which function to call. No mixed abstractions.

Pattern Matching

Using Java's switch expressions:

```
Result<ShippingCost> calculateShipping(Order order, ShippingMethod method) {
    return switch (method) {
        case STANDARD -> standardShipping(order);
        case EXPRESS -> expressShipping(order);
        case OVERNIGHT -> overnightShipping(order);
    };
}
```

Each case returns `Result<ShippingCost>`. The switch expression evaluates to a single result.

Nested Conditions

Avoid deep nesting by extracting subdecisions into named functions:

```
// DON'T: Nested ternaries
return user.isPremium()
    ? (order.total().greaterThan(THRESHOLD)
        ? largeOrderPremiumDiscount(order)
        : smallOrderPremiumDiscount(order))
    : (order.total().greaterThan(THRESHOLD)
        ? largeOrderStandardDiscount(order)
        : smallOrderStandardDiscount(order));
```

Extract:

```
// DO: Extract nested logic
Result<Discount> calculateDiscount(User user, Order order) {
    return user.isPremium()
        ? premiumDiscount(order)
        : standardDiscount(order);
}

private Result<Discount> premiumDiscount(Order order) {
    return order.total().greaterThan(THRESHOLD)
        ? largeOrderPremiumDiscount(order)
        : smallOrderPremiumDiscount(order);
}

private Result<Discount> standardDiscount(Order order) {
    return order.total().greaterThan(THRESHOLD)
        ? largeOrderStandardDiscount(order)
        : smallOrderStandardDiscount(order);
}
```

Now each function has one level of branching. Much clearer.

Condition with Monads

Use `filter` for cleaner composition when applicable:

```
// DON'T: Ternary in lambda
return fetchUser(userId)
    .flatMap(user -> user.isActive()
        ? processActiveUser(user)
        : UserError.InactiveAccount.INSTANCE.result());

// DO: Using filter (preferred)
return fetchUser(userId)
    .filter(User::isActive, UserError.InactiveAccount.INSTANCE)
    .flatMap(this::processActiveUser);
```

Pattern: Iteration

Definition: Iteration processes collections, streams, or recursive structures. Prefer functional combinators over explicit loops. Keep transformations pure.

Rationale (by criteria): - **Mental Overhead**: Declarative combinators state intent; imperative loops require tracing (+2) - **Business/Technical Ratio**: map/filter express business logic; loops are iteration mechanics (+2) - **Complexity**: Functional composition eliminates index management and loop state (+2) - **Reliability**: Pure transformations have no side effects - deterministic and testable (+2)

Mapping Collections

```
// Transforming a list of raw inputs to domain objects
Result<List<Email>> parseEmails(List<String> rawEmails) {
    return Result.allOf(rawEmails.stream()
                                 .map(Email::email)
                                 .toList());
}
```

`Result.allOf` aggregates a `List<Result<Email>>` into `Result<List<Email>>`. If any email is invalid, you get a `CompositeCause` with all failures.

Filtering and Transforming

```
List<ActiveUser> activeUsers(List<User> users) {
    return users.stream()
                .filter(User::isActive)
                .map(this::toActiveUser)
                .toList();
}

private ActiveUser toActiveUser(User user) {
    return new ActiveUser(user.id(), user.email());
}
```

Pure transformation, no side effects, returns `List<ActiveUser>` (type `T`, not `Result`, because this can't fail).

Async Iteration

When processing collections with async operations, decide between sequential and parallel:

Sequential:

```
// Process orders one at a time
Promise<List<Receipt>> processOrders(List<Order> orders) {
    Promise<List<Receipt>> result = Promise.success(List.of());

    for (Order order : orders) {
        result = result.flatMap2(this::appendReceipt, order);
    }

    return result;
}

private Promise<List<Receipt>> appendReceipt(List<Receipt> receipts, Order order) {
    return processOrder(order)
                .map(receipt -> appendToList(receipts, receipt));
}
```

Parallel (when orders are independent):

```
// Process orders in parallel
Promise<List<Receipt>> processOrders(List<Order> orders) {
    return Promise.allOf(orders.stream()
                               .map(this::processOrder)
                               .toList());
}
```

Use parallel when operations are independent.

Common Mistakes

DON'T mix side effects into stream operations:

```
// DON'T: Side effect in the map
users.stream()
    .map(user -> {
        logger.info("Processing user: {}", user.id());  // Side effect!
        return processUser(user);
    })
    .toList();
```

DON'T use imperative loops when combinators exist:

```
// DON'T: Imperative accumulation
List<Result<Email>> results = new ArrayList<>();
for (String raw : rawEmails) {
    results.add(Email.email(raw));
}

// DO: Declarative collection
Result<List<Email>> emails = Result.allOf(
```

```
    rawEmails.stream().map(Email::email).toList()
);
```

Naming Conventions

Consistent naming reduces cognitive load and makes code self-documenting.

Factory Method Naming

Factories are always named after their type, lowercase-first (camelCase):

```
Email.email("user@example.com")
Password.password("Secret123")
AccountId.accountId("ACC-001")
UserId.userId(raw)
```

This pattern is grep-friendly: searching for `Email.email` finds all email construction sites.

Validated Input Naming

Use the `Valid` prefix (not `Validated`) for types representing validated inputs:

```
// DO: Use Valid prefix
record ValidRequest(Email email, Password password) {
    static Result<ValidRequest> validRequest(Request raw) { ... }
}

// DON'T: Use Validated prefix (too verbose)
record ValidatedRequest(...)
```

Zone-Based Naming Vocabulary

Zone 2 Verbs (Step Interfaces - Orchestration):

Verb	When to Use	Example
`validate`	Checking rules/constraints	`ValidateInput`
`process`	Transforming or interpreting data	`ProcessPayment`
`handle`	Coordinating reactions to events	`HandleRefund`
`load`	Retrieving data for use	`LoadUserProfile`
`save`	Persisting changes	`SaveOrder`
`check`	Verifying conditions	`CheckInventory`

Zone 3 Verbs (Leaves - Implementation):

Verb	Typical Use	Example
`get`	Retrieve a value	`getTimestamp()`
`fetch`	Pull from external source	`fetchWeatherData()`
`parse`	Break down structured input	`parseJson()`
`calculate`	Perform computation	`calculateTax()`
`hash`	Cryptographic transformation	`hashPassword()`
`format`	Build structured output	`formatDate()`
`send`	Transmit over network	`sendEmail()`

Test Naming

Follow the pattern: `methodName_outcome_condition`

```
void validRequest_succeeds_forValidInput()
void validRequest_fails_forInvalidEmail()
void execute_succeeds_forValidInput()
void execute_fails_whenEmailAlreadyExists()
```

Key Takeaways

1. **Single Pattern Per Function** - One pattern per function, extract if you see two
2. **Single Level of Abstraction** - No complex logic in lambdas, only method references or simple forwarding
3. **Leaf** - Atomic unit: pure computation (business) or I/O (adapter)
4. **Condition** - Branching as values, same type from all branches
5. **Iteration** - Functional combinators over collections, pure transformations
6. **Zone-based naming** - Use appropriate verbs for each abstraction level

---

Exercises

See Appendix B for exercises on: - Exercise 3.1: Pattern identification - Exercise 3.2: Leaf extraction - Exercise 3.4: Zone-based naming

---

What's Next

Chapter 8 covers advanced patterns - Sequencer, Fork-Join, and Aspects - that compose these basic patterns into sophisticated workflows.

Chapter 12: Complete Example - RegisterUser

What You'll Learn

- Complete use case from requirements to implementation
- How all patterns work together in practice
- Step-by-step evolutionary development

Prerequisites: Chapter 11: Testing in Practice

Requirements

Use case: Register a new user account.

Inputs (raw): - Email (string) - Password (string) - Referral code (optional string)

Outputs: - User ID - Confirmation token

Validation rules: - Email: not null, valid format, lowercase normalized - Password: not null, min 8 chars, at least one uppercase, one digit - Referral code: optional; if present, must be exactly 6 uppercase alphanumeric characters

Cross-field rules: - Email must not be registered yet

Steps: 1. Validate input 2. Check email uniqueness (async, database) 3. Hash password (sync, expensive computation) 4. Save the user to the database (async) 5. Generate confirmation token (async, calls external service)

Step 1: Use Case Interface

```
package com.example.app.usecase.registeruser;

import org.pragmatica.lang.*;

public interface RegisterUser {
    record Request(String email, String password, String referralCode) {}
    record Response(UserId userId, ConfirmationToken token) {}
```

```
    Promise<Response> execute(Request request);

    interface CheckEmailUniqueness {
        Promise<ValidRequest> apply(ValidRequest valid);
    }

    interface CreateValidUser {
        Promise<ValidUser> apply(ValidRequest valid);
    }

    interface SaveUser {
        Promise<User> apply(ValidUser validUser);
    }

    interface GenerateToken {
        Promise<Response> apply(User user);
    }

    static RegisterUser registerUser(CheckEmailUniqueness checkEmail,
                                     CreateValidUser createValidUser,
                                     SaveUser saveUser,
                                     GenerateToken generateToken) {
        return request -> ValidRequest.validRequest(request)
                                      .async()
                                      .flatMap(checkEmail::apply)
                                      .flatMap(createValidUser::apply)
                                      .flatMap(saveUser::apply)
                                      .flatMap(generateToken::apply);
    }
}
```

This is a **Sequencer pattern**: validate -> check uniqueness -> create user -> save -> generate token.

Why interface + factory? Every component — use case, step, adapter — is defined as an interface with a static factory method. This is not arbitrary convention:

- **Substitutability**: Anyone can implement the interface. Testing, stubbing incomplete implementations, swapping adapters — all work without framework magic or inheritance hierarchies.
- **Implementation isolation**: Each implementation is self-contained. No shared base classes, no abstract methods to override, no coupling between implementations. Each intersection between implementations is unnecessary coupling with corresponding maintenance overhead — up to needing deep understanding of two projects instead of one, with zero benefit.
- **Disposable implementation**: A local record or lambda returned by the factory can't be referenced externally. The implementation is replaceable by definition.

The interface is the design artifact; the implementation is incidental.

Step 2: Validated Request

```
record ValidRequest(Email email, Password password, Option<ReferralCode> referralCode) {

    public static Result<ValidRequest> validRequest(Request raw) {
        return Result.all(Email.email(raw.email()),
                          Password.password(raw.password()),
                          ReferralCode.referralCode(raw.referralCode()))
                     .map(ValidRequest::new);
    }
}
```

This is **Fork-Join pattern**: validate three fields independently, collect results.

Step 3: Value Objects

Email:

```
package com.example.app.domain.shared;

public record Email(String value) {
    private static final Pattern EMAIL_PATTERN =
        Pattern.compile("^[a-z0-9+_.-]+@[a-z0-9.-]+$");
    private static final Fn1<Cause, String> INVALID_EMAIL =
        Causes.forOneValue("Invalid email format: %s");

    public static Result<Email> email(String raw) {
        return Verify.ensure(raw, Verify.Is::present)
                     .map(String::trim)
                     .map(String::toLowerCase)
                     .filter(INVALID_EMAIL, EMAIL_PATTERN.asMatchPredicate())
                     .map(Email::new);
    }
}
```

Password:

```
public record Password(String value) {
    private static final Cause TOO_SHORT =
        Causes.cause("Password must be at least 8 characters");
    private static final Cause MISSING_UPPERCASE =
        Causes.cause("Password must contain uppercase letter");
```

```
    private static final Cause MISSING_DIGIT =
        Causes.cause("Password must contain digit");

    public static Result<Password> password(String raw) {
        return Verify.ensure(raw, Verify.Is::present)
                     .filter(TOO_SHORT, s -> Verify.Is.lenBetween(s, 8, 128))
                     .flatMap(Password::ensureUppercase)
                     .flatMap(Password::ensureDigit)
                     .map(Password::new);
    }

    private static Result<String> ensureUppercase(String raw) {
        return contains(raw, Character::isUpperCase)
                ? Result.success(raw)
                : MISSING_UPPERCASE.result();
    }

    private static Result<String> ensureDigit(String raw) {
        return contains(raw, Character::isDigit)
                ? Result.success(raw)
                : MISSING_DIGIT.result();
    }

    private static boolean contains(CharSequence sequence, IntPredicate predicate) {
        return sequence.chars().anyMatch(predicate);
    }
}
```

ReferralCode (optional-with-validation):

```
public record ReferralCode(String value) {
    private static final Pattern PATTERN = Pattern.compile("^[A-Z0-9]{6}$");
    private static final Cause INVALID_FORMAT = Causes.cause("Invalid referral code forma

    public static Result<Option<ReferralCode>> referralCode(String raw) {
        return Verify.ensureOption(
            Option.option(raw).map(String::trim).filter(s -> !s.isEmpty()),
            PATTERN.asMatchPredicate(),
            INVALID_FORMAT
        ).map(opt -> opt.map(ReferralCode::new));
    }

    public boolean isPremium() {
        return value.startsWith("VIP");
    }
}
```

The `Verify.ensureOption()` method (Pragmatica Core 0.9.0+) handles `Result<Option<ReferralCode>`

elegantly: if the `Option` is empty, succeeds with `Option.none()`; if present and valid, succeeds with `Option.some(value)`; if present and invalid, fails.

Step 4: Step Implementations

CheckEmailUniqueness (adapter leaf):

```
interface CheckEmailUniqueness {
    Promise<ValidRequest> apply(ValidRequest request);

    static CheckEmailUniqueness checkEmailUniqueness(UserRepository repository) {
        return request -> repository.findByEmail(request.email())
                                    .flatMap(user -> checkPresence(user, request));
    }

    static Promise<ValidRequest> checkPresence(Option<User> user, ValidRequest request) {
        return user.isPresent()
                ? RegistrationError.General.EMAIL_ALREADY_REGISTERED.promise()
                : Promise.success(request);
    }
}
```

HashPassword (business leaf):

```
interface HashPassword {
    Result<HashedPassword> apply(Password password);

    static HashPassword hashPassword(BCryptPasswordEncoder encoder) {
        return password -> Result.lift1(RegistrationError.PasswordHashingFailed::new,
                                        encoder::encode,
                                        password.value())
                                 .map(HashedPassword::new);
    }
}
```

SaveUser (adapter leaf):

```
class JooqUserRepository implements SaveUser {
    private final DSLContext dsl;

    public Promise<User> apply(ValidUser user) {
        return Promise.lift(RepositoryError.DatabaseFailure::cause,
                            () -> saveUser(user));
    }

    private User saveUser(ValidUser user) {
        String id = dsl.insertInto(USERS)
```

```
                        .set(USERS.EMAIL, user.email().value())
                        .set(USERS.PASSWORD_HASH, user.hashed().value())
                        .set(USERS.REFERRAL_CODE,
                             user.refCode().map(ReferralCode::value).orElse(null))
                        .returningResult(USERS.ID)
                        .fetchSingle()
                        .value1();

        return new User(new UserId(id), user.email());
    }
}
```

GenerateToken (adapter leaf):

```
class TokenServiceClient implements GenerateToken {
    private final HttpClient httpClient;

    public Promise<Response> apply(User user) {
        return httpClient.post("/tokens/confirm",
                               Map.of("userId", user.id().value()))
                         .map(resp -> buildResponse(user.id(), resp))
                         .recover(this::mapTokenError);
    }

    private Response buildResponse(UserId userId, Map<String, String> resp) {
        return new Response(userId, new ConfirmationToken(resp.get("token")));
    }

    private Promise<Response> mapTokenError(Cause cause) {
        return RegistrationError.General.TOKEN_GENERATION_FAILED.promise();
    }
}
```

Step 5: Error Types

```
public sealed interface RegistrationError extends Cause {
    enum General implements RegistrationError {
        EMAIL_ALREADY_REGISTERED("Email already registered"),
        WEAK_PASSWORD_FOR_PREMIUM("Premium codes require 10+ char passwords"),
        TOKEN_GENERATION_FAILED("Token generation failed");

        private final String message;

        General(String message) {
            this.message = message;
```

```
        }

        @Override
        public String message() {
            return message;
        }
    }

    record PasswordHashingFailed(Throwable cause) implements RegistrationError {
        @Override
        public String message() {
            return "Password hashing failed: " + Causes.fromThrowable(cause);
        }
    }
}
```

Sealed interface ensures exhaustive pattern matching.

Step 6: Tests

Validation tests:

```
@Test
void validRequest_fails_forInvalidEmail() {
    var request = new Request("not-an-email", "Valid1234", null);

    ValidRequest.validRequest(request)
                .onSuccess(Assertions::fail);
}

@Test
void validRequest_succeeds_forValidInput() {
    var request = new Request("user@example.com", "Valid1234", "ABC123");

    ValidRequest.validRequest(request)
                .onFailure(Assertions::fail)
                .onSuccess(valid -> {
                    assertEquals("user@example.com", valid.email().value());
                    assertTrue(valid.referralCode().isPresent());
                });
}
```

Happy path test:

```
@Test
void execute_succeeds_forValidInput() {
```

```
    CheckEmailUniqueness checkEmail = req -> Promise.success(req);
    CreateValidUser createUser = req -> Promise.success(
        new ValidUser(req.email(), new HashedPassword("hashed"), req.referralCode()));
    SaveUser saveUser = user -> Promise.success(new User(new UserId("user-123"), user.ema
    GenerateToken generateToken = user -> Promise.success(
        new Response(user.id(), new ConfirmationToken("token-456")));

    var useCase = RegisterUser.registerUser(checkEmail, createUser, saveUser, generateTok
    var request = new Request("user@example.com", "Valid1234", null);

    useCase.execute(request)
           .await()
           .onFailure(Assertions::fail)
           .onSuccess(response -> {
               assertEquals("user-123", response.userId().value());
               assertEquals("token-456", response.token().value());
           });
}
```

Failure scenario:

```
@Test
void execute_fails_whenEmailAlreadyExists() {
    CheckEmailUniqueness checkEmail = req ->
        RegistrationError.General.EMAIL_ALREADY_REGISTERED.promise();
    // ... other stubs

    useCase.execute(request)
           .await()
           .onSuccess(Assertions::fail);
}
```

Pattern Summary

Component	Pattern	Purpose
RegisterUser.execute	Sequencer	Chain dependent steps
ValidRequest.validRequest	Fork-Join	Validate independent fields
Email.email	Leaf	Atomic validation
Password.password	Leaf	Atomic validation with Sequencer internally
CheckEmailUniqueness	Leaf + Condition	Database check with branching
SaveUser	Adapter Leaf	Database write

Component	Pattern	Purpose
RegistrationError	Sealed Interface	Typed error hierarchy

Key Takeaways

1. **Use case interface** - Contains factory, step interfaces, types
2. **Sequencer for main flow** - Chain steps with flatMap
3. **Fork-Join for validation** - Accumulate errors with Result.all()
4. **Value objects as Leaves** - Parse-don't-validate
5. **Adapters implement step interfaces** - JOOQ, HTTP clients
6. **Tests use stubs** - Only adapters stubbed

Exercises

See Appendix B for exercises on: - Exercise 5.1: Extend RegisterUser with email verification - Exercise 5.3: Add premium referral validation

---

What's Next

Chapter 13 presents another complete example - PlaceOrder - demonstrating more complex patterns including Fork-Join for parallel data fetching.