4 · The SharedKernel

The smallest project in the solution, on purpose

Nexus.SharedKernel is the one project every other project references, which makes it the most dangerous project in the solution: anything placed here is coupled to everything forever. The admission rule is therefore brutal and worth memorising — **a type enters the kernel only if it is business-meaning-free and needed by many contexts.** Five citizens pass: the *Entity<TId>* base class, strongly-typed identifiers, the *IDomainEvent* contract, the *Result*/*Error* pair, and the *Ensure* guard clauses. Everything else — including types that *feel* universal, like *EngineeringValue* — belongs to a context and is refused entry; the end of this chapter prosecutes exactly that temptation. First, the citizens themselves, each printed in full.

Entity: identity, equality, and a mailbox for events

```
namespace Nexus.SharedKernel;

public abstract class Entity<TId> : IEquatable<Entity<TId>> where TId : notnull
{
    private readonly List<IDomainEvent> _domainEvents = [];

    protected Entity(TId id) => Id = id;

    public TId Id { get; }

    public IReadOnlyList<IDomainEvent> DomainEvents => _domainEvents.AsReadOnly();

    protected void Raise(IDomainEvent domainEvent) => _domainEvents.Add(domainEvent);

    public void ClearDomainEvents() => _domainEvents.Clear();

    public bool Equals(Entity<TId>? other) =>
        other is not null && GetType() == other.GetType() && Id.Equals(other.Id);

    public override bool Equals(object? obj) => Equals(obj as Entity<TId>);

    public override int GetHashCode() => HashCode.Combine(GetType(), Id);

    public static bool operator ==(Entity<TId>? a, Entity<TId>? b) =>
        a is null ? b is null : a.Equals(b);

    public static bool operator !=(Entity<TId>? a, Entity<TId>? b) => !(a == b);
}
```

Listing 4.1 — src/Nexus.SharedKernel/Entity.cs, complete. Identity-based equality plus the domain-event mailbox.

Two design facts carry the weight. Equality compares *GetType()* and *Id* — nothing else — because an entity that changes every property is still itself; that is the definition Chapter 6 builds on. And the event mailbox is *protected* on the way in, public and read-only on the way out: only the entity may raise its own events, while dispatching them is deliberately left to a seam Volume II owns.

Strongly-typed identifiers: making Guids unswappable

A 654-table system runs on identifiers, and raw Guids make every pair of them silently interchangeable. The kernel's answer costs three lines per context type:

```
// src/Nexus.SharedKernel/IEntityId.cs
namespace Nexus.SharedKernel;

public interface IEntityId { Guid Value { get; } }

// each context declares its own, e.g. in Nexus.Domain/AlarmManagement/:
public readonly record struct AlarmEventId(Guid Value) : IEntityId
{
    public static AlarmEventId New() => new(Guid.NewGuid());
    public override string ToString() => Value.ToString("N");
}

// the bug this abolishes -- two ids passed in the wrong order:
ackService.Acknowledge(operator.Id, alarm.Id);   // swapped
error CS1503: argument 1: cannot convert from 'OperatorId' to 'AlarmEventId'
```

Listing 4.2 — The identifier pattern, and the argument-swap bug dying at compile time.

A *readonly record struct* gives value equality, immutability, and zero allocation for the price of one line, and the wrapper evaporates at the database boundary: *From Entity to Context* already reserved the conversion seam, and Volume II fills it with EF Core value converters without the Domain noticing.

Domain events: the contract, not yet the plumbing

```
namespace Nexus.SharedKernel;

public interface IDomainEvent
{
    Guid EventId { get; }
    DateTime OccurredAtUtc { get; }
}

public abstract record DomainEvent(Guid EventId, DateTime OccurredAtUtc) : IDomainEvent;
```

Listing 4.3 — src/Nexus.SharedKernel/IDomainEvent.cs and DomainEvent.cs, complete.

Nine lines, and deliberately no more. The kernel defines what an event *is* — an immutable record of something that happened, named in the past tense — and stays silent on how events travel. Concrete events like *AlarmRaisedEvent* belong to their contexts (Chapter 9); dispatch belongs to the Application pipeline (Chapter 16) and, across process boundaries, to Volume II. Value objects, the other famous kernel citizen, are conspicuously absent: in .NET 8 a *sealed record* with a private constructor and a static factory does the job with no base class at all, and Chapter 7 makes that case with the kernel's blessing.

Result and Error: two failure channels, kept apart

The kernel's doctrine on failure is the most consequential decision in this chapter. **Business-rule failures are values; programmer errors are exceptions.** An operator acknowledging an already-cleared alarm is not exceptional — it is Tuesday — and modelling it with a throw turns control flow into stack unwinding and makes the rule invisible in the method signature. So domain operations that can legitimately fail return *Result*:

```
namespace Nexus.SharedKernel;

public sealed record Error(string Code, string Message)
{
    public static readonly Error None = new(string.Empty, string.Empty);
}

public class Result
{
    protected Result(bool isSuccess, Error error)
    {
        if (isSuccess != (error == Error.None))
            throw new ArgumentException("Invalid success/error combination.");
        IsSuccess = isSuccess;
        Error = error;
    }

    public bool IsSuccess { get; }
    public bool IsFailure => !IsSuccess;
    public Error Error { get; }

    public static Result Success() => new(true, Error.None);
    public static Result Failure(Error error) => new(false, error);
    public static Result<T> Success<T>(T value) => new(value, true, Error.None);
    public static Result<T> Failure<T>(Error error) => new(default, false, error);
}

public sealed class Result<T> : Result
{
    private readonly T? _value;

    internal Result(T? value, bool ok, Error error) : base(ok, error) => _value = value;

    public T Value => IsSuccess ? _value!
        : throw new InvalidOperationException("No value on a failed result.");
}
```

Listing 4.4 — src/Nexus.SharedKernel/Error.cs and Result.cs, complete.

Note where the two channels meet: reading *Value* off a failed result throws, because ignoring a failure you were handed as a value is a programmer error, not a business outcome. Error codes are strings by convention *Context.Rule* — “RootCause.CaseWithoutEvidence” — which Volume II will map onto HTTP problem details without the Domain changing a line.

Ensure: the exception channel, fenced

```
namespace Nexus.SharedKernel;

public static class Ensure
{
    public static void NotEmpty(string value, string name)
    {
        if (string.IsNullOrWhiteSpace(value))
            throw new ArgumentException($"{name} must not be empty.", name);
    }

    public static void NotDefault<T>(T value, string name) where T : struct
    {
        if (EqualityComparer<T>.Default.Equals(value, default))
            throw new ArgumentException($"{name} must not be default.", name);
    }
}
```

Listing 4.5 — src/Nexus.SharedKernel/Ensure.cs, complete. Guards protect invariants of construction, not business rules.

The first green — and the first watched failure

The kernel is small enough to prove in one sitting, and doing so establishes the test rhythm the whole book keeps. Two tests below: the first pins the equality contract; the second is the book's first *watched-to-fail* — it deliberately does the wrong thing (reading a value off a failure) and asserts that the kernel punishes it.

```
namespace Nexus.Domain.Tests.SharedKernel;

public sealed class KernelContractTests
{
    private sealed record TestId(Guid Value);
    private sealed class TestEntity(TestId id) : Entity<TestId>(id);

    [Fact]
    public void Entities_with_the_same_id_are_equal_whatever_their_state()
    {
        var id = new TestId(Guid.NewGuid());
        Assert.Equal(new TestEntity(id), new TestEntity(id));
    }

    [Fact]
    public void Reading_Value_from_a_failed_Result_throws()
    {
        var failed = Result.Failure<int>(new Error("Test.Rule", "nope"));
        Assert.Throws<InvalidOperationException>(() => failed.Value);
    }
}

$ dotnet test tests/Nexus.Domain.Tests
Passed! - Failed: 0, Passed: 2, Skipped: 0, Duration: 41 ms
```

Listing 4.6 — tests/Nexus.Domain.Tests/SharedKernel/KernelContractTests.cs, and the run.

The refusal that keeps the kernel honest

Now the prosecution promised on page 22. *EngineeringValue* — a number with a unit and a quality flag, defined by *From Grid to Core* and used by half the plant — looks like the perfect kernel citizen. It is refused, on the admission rule's first clause: it is saturated with business meaning. Which units exist, how quality degrades, what conversions are legal — those are Instrumentation-context decisions, and freezing them into the kernel would let every context depend on their details forever. It lives in *Nexus.Domain.Instrumentation* (Chapter 7), and contexts that need it say so in the ordinary way. The kernel stays five citizens small — and the stability contract is explicit: **within this trilogy, Nexus.SharedKernel gains no new public type after this chapter.** Volumes II and III will be the audit of that sentence.

> **HONEST BOUNDARY**
>
> Terminology, priced: in DDD strategic design, “Shared Kernel” names a negotiated subset of the *domain model* that two teams co-own — a relationship between contexts. This project is the humbler .NET usage: shared technical base types with no domain meaning at all. *From Domain to Twin* uses the strategic sense; this book uses the project sense, and the two must not be conflated. Also priced: the Result pattern is a stance, not a consensus — respected codebases model rule failures with exceptions and gain simpler call sites at the cost of invisible signatures. NEXUS-1 chooses visible signatures; Chapter 14 shows the price being paid, in full, at every handler.

> **CHECKPOINT · CHAPTER 4**
>
> Before moving on, you should be able to answer these without looking back:
>
> **1.** State the kernel admission rule, and use it to convict *EngineeringValue* in one sentence.
>
> **2.** Entity equality inspects exactly two things. Which two, why not the rest — and why does *GetType()* participate at all?
>
> **3.** Why is *Raise* protected while *ClearDomainEvents* is public? Who is each visibility aimed at?
>
> **4.** Name the two failure channels and assign each of these to one: a case closed without evidence; a Guid identifier that is all zeros; reading *Value* off a failure.
>
> **5.** What does *readonly record struct* buy an identifier, and where does the wrapper get unwrapped — in which volume, and by whom?

Five projects, five kernel citizens, two green tests. One structural question remains before Part II can pour the domain itself: who constructs all these objects, and where do interfaces meet their implementations? The next chapter builds the composition seam — dependency injection as Volume I practices it, including the discipline of a composition root that does not yet have an application to compose.

Generics, Records, and Two Kinds of No

WHERE WE ARE IN THE SOLUTION

```
  Nexus.sln
> |- src/Nexus.SharedKernel
  |- src/Nexus.Domain
  |    |- <17 context folders>
  |- src/Nexus.Application
  |    |- Abstractions (ports)
  |- tests/Nexus.Domain.Tests
  |- tests/Nexus.Application.Tests
```

Tests green: 2

The kernel is populated and frozen: five citizens, two green tests, and a promise — no new public kernel type for the rest of the trilogy — that later chapters will audit.

Generics as fill-in-the-blank contracts

Entity<TId> reads, to a newcomer, like punctuation soup. Read it as a form with one blank: “I am an Entity, and my identifier is of type ____”. Each aggregate fills the blank once — *AlarmEvent : Entity<AlarmEventId>* — and from then on the compiler holds it to the answer: *Id* is an *AlarmEventId*, not a Guid, not anyone else’s key. The *where TId : notnull* clause is a condition on the blank itself (“no nullable answers”), and Chapter 12’s *where TRoot : Entity<TId>, IAggregateRoot* will stack two conditions the same way. Generics here are not cleverness; they are how one base class serves seventeen contexts without ever confusing their keys.

Records vs classes, in one paragraph

A **class** compares by reference: two objects are equal only if they are literally the same object in memory. A **record** is a class where the compiler rewrites equality to compare *contents* — plus generated *GetHashCode*, *ToString*, and copy machinery — which is exactly the semantics Chapter 7 wants for values and exactly wrong for entities, whose whole identity is “sameness despite change”. Hence the kernel’s split: *Entity* is a class with equality hand-written to compare Id and type only, while events and value objects are records and take the generated behavior. *Equals* and *GetHashCode* travel as a pair because every dictionary and set consults the hash first — disagree between them and collections quietly lose your objects, which is why Listing 4.1 overrides both or neither, never one.

The kernel as a UML class diagram

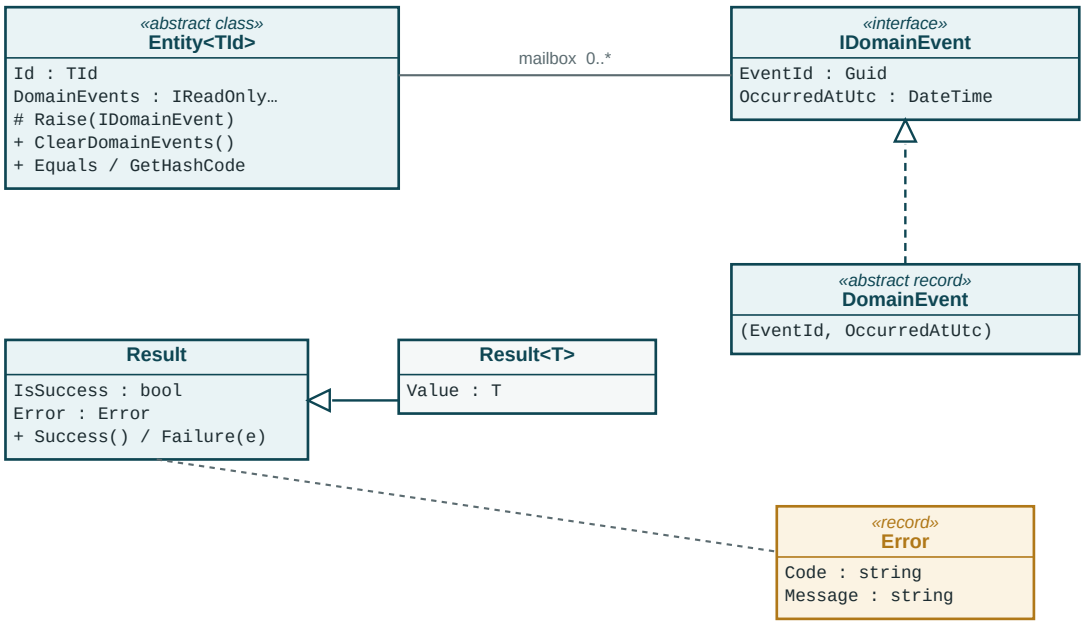

Figure D4.1 — UML class diagram of Nexus.SharedKernel. Dashed triangle: implements; solid triangle: inherits; plain line: holds.

The two channels, as a decision you can run

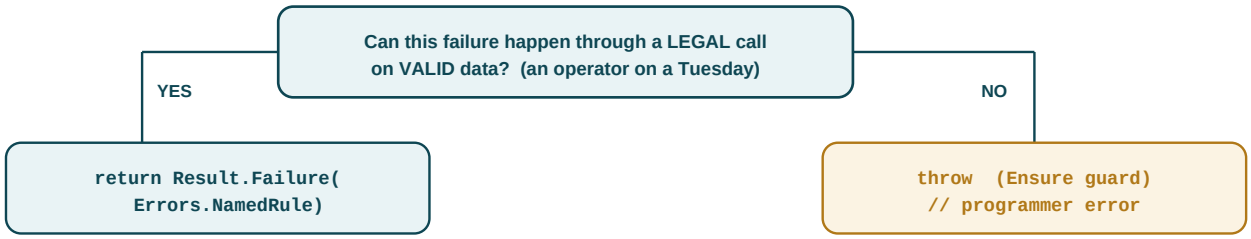

Figure D4.2 — The two-channel decision. Run it on every failure in the book; Chapters 6–17 never deviate from it.

Try the figure on three cases from later chapters before you meet them: an alarm acknowledged twice (legal call, valid data — left box, *AlarmManagement.NotRaised*); an identifier that is all zeros (no legal path produces one — right box, *Ensure.NotDefault*); reading *Value* off a failed *Result* (the caller ignored an answer it was handed — right box, and Listing 4.6 watches it throw). If you can route those three, you can route every failure in the book.