DRAFT - IT Project Guidance

Practical & Effective Domain Driven Design

Version:

0.1

## Purpose

This document aims to reduce the failure rate of digital initiatives by improving how problems are understood and modelled before solutions are procured. Most projects fail not due to technology, tools, or code, but because they are shaped around incorrect assumptions, vague concepts, or incomplete analysis. A structured, domain-driven approach to analysis offers a clear, practical method for framing problems accurately before building solutions.

## Synopsis

Domain-Driven Design (DDD) provides a mature, proven approach for identifying, decomposing, and bounding complexity into achievable, separate concerns. By structuring systems around clear domains and stable models, DDD reduces complexity, improves modularity, and significantly lowers the risk of project failure.

While it does not guarantee success, DDD increases the likelihood of building resilient, adaptable systems that remain relevant over time. It supports collaboration across roles by establishing a clear shared understanding of the system’s structure and purpose, but its primary value lies in its disciplined method for confronting complexity early, before solutions are built.

## Contents

[Purpose 1](#_Toc196827150)

[Synopsis 1](#_Toc196827151)

[Contents 2](#_Toc196827152)

[Introduction 3](#_Toc196827153)

[Context 3](#_Toc196827154)

[XTRA 3](#_Toc196827155)

[Purpose and Scope 4](#_Toc196827156)

[Misconceptions and Misuse 4](#_Toc196827157)

[Domain Thinking and Subdomains 5](#_Toc196827158)

[Core Concepts and Platforms 7](#_Toc196827159)

[DDD Core Concepts for All Team Members 8](#_Toc196827160)

[Domain Thinking and Subdomains 10](#_Toc196827161)

[Appendices 11](#_Toc196827162)

[Appendix A - Document Information 11](#_Toc196827163)

[Versions 11](#_Toc196827164)

[Images 11](#_Toc196827165)

[Tables 11](#_Toc196827166)

[References 11](#_Toc196827167)

[Review Distribution 11](#_Toc196827168)

[Audience 11](#_Toc196827169)

[Structure 11](#_Toc196827170)

[Diagrams 12](#_Toc196827171)

[Acronyms 12](#_Toc196827172)

[Terms 12](#_Toc196827173)

# Introduction

This guidance forms part of a broader initiative to improve the consistency, reliability, and effectiveness of IT project delivery within government and enterprise contexts. It contributes to a Solution Architecture Body of Knowledge (SABOK) and is intended for reuse across programmes and delivery teams.  
The material applies to most projects seeking meaningful digital change, whether delivered by in-house teams, contracted partners, or collaborative models. It supports both decision-makers and practitioners by providing a shared structure for understanding and designing complex systems.  
By applying this guidance, projects can improve resilience, adaptability, and alignment with human, organisational, and policy realities, while reducing ambiguity and increasing modularity.

# Context

Most IT-enabled projects still fail to meet their expected outcomes. Studies such as the Standish Group’s CHAOS Report consistently report success rates below 40%. Failures are rarely technical; they typically stem from unclear problem definitions, misaligned expectations, and inadequate understanding of real-world complexity.  
This is not a new problem, but it is intensifying. Systems today must support growing regulatory obligations, diverse user needs, and frequent changes in policy and practice. Without more rigorous, structured methods for understanding and decomposing problem spaces, projects will continue to struggle.  
Domain-Driven Design offers a mature, practical response.

# Domain Driven Design

Domain-Driven Design (DDD) was developed to reduce failure in complex software projects by providing a disciplined way to confront complexity early and organise systems around clear, stable boundaries. It emphasises the identification, decomposition, and bounding of distinct domains of knowledge and activity, reducing risk by breaking large, ambiguous problems into manageable, separated concerns.

Originally formalised by Eric Evans in the early 2000s, DDD offered a structured response to disorganised system development, where technical components were often built without a coherent model of the problem space.

DDD provides a proven, practical, and accessible approach to analysis, problem definition, and solution delivery. It cannot guarantee success, but it substantially improves the likelihood of building systems that are resilient, modular, and adaptable to future change.

Although DDD supports collaboration between business and technical roles by establishing a shared understanding of each bounded domain, its core value lies in enabling teams to model what matters clearly, structure it durably, and build solutions that can evolve without destabilisation. It establishes a foundation of thinking that can be reused across initiatives, improving the clarity, stability, and coherence of delivery long before any code is written.

# Misconceptions and Misuse

Over time, Domain-Driven Design has been both overcomplicated and underutilised.

For some teams, it was misued as only résumé padding.

For others, it was only cursorily investigated before being misconstrued and dismissed as a heavyweight or academic methodology suited only to large-scale platforms. In reality, any information system that operates within regulatory, organisational, or multi-user environments is complex. Which is practically all enterprise solutions these days. Complexity, not size, is the appropriate trigger for applying DDD.

A key misconception by many tems was an over focus on Eric Evans’ term "ubiquitous language" concept and concluding this meant mirroring stakeholder vocabulary, leading them to develop shallow models that copied business words into technical structures without using time tested development patterns of abstraction to support reuse or extensibility.

However, this was *never* the intention of DDD. While stakeholder terms are essential for communication, internal models must be more abstract, stable, and extensible. You do not build systems out of enrolments and classrooms. You build them out of foundational concepts such as groups, people, relationships, artefacts, roles, and plans. These are reusable, pre-solved domains that extend across systems and sectors. By reusing these, delivery risk is reduced, and solutions avoid being overfitted to case-specific outcomes.

Finally, modern tooling, such as .NET Core, has absorbed many DDD principles and eliminated much of the technical effort previously involved. Building services, APIs, and database models following DDD-aligned practices is now fast and scaffolded.

The slow part is no longer construction—it is thinking. This is where DDD adds enduring value: ensuring that what is built genuinely reflects the needs, structure, and language of the exposed domain.

# Domain Knowledge

Domain experts often view their area as a singular world with its own rich terminology and internal logic. They may be deeply invested in this language and the narratives that surround it. This perspective is valuable—it provides the raw material for what Eric Evans termed the "ubiquitous language"—but effective system design requires us to look beneath that surface. This is where subdomain identification and bounding begins.

## Sub Domain Identification and Bounding

A central insight of Domain-Driven Design is that no problem domain is truly monolithic or entirely unique. What appears to stakeholders as a single, novel problem space is often a composition of smaller, well-trodden subdomains. These subdomains, when recognised and bounded, reveal opportunities to apply pre-solved models and known design patterns, reducing the novelty and risk of custom development.

True system design comes from mapping business language and mental models to durable, abstract internal representations. In other words, we separate what the business does from how the business talks about it. This translation process allows us to develop stable software models that are not tightly coupled to shifting organisational terms or specific workflows. Within each bounded context, the domain expert’s language is honoured and retained—but the model underneath is shaped by enduring structural patterns that stand the test of time.

Architects and designers act as interpreters: they distil the domain’s unique expressions into stable, reusable building blocks. The resulting implementation respects the business’s reality while avoiding reinvention of generic patterns already solved in other domains.

Adopting a subdomain-oriented mindset significantly improves system design in several ways:

* **Lower Risk, Less Novelty**: Recognising recurring subdomains enables teams to reuse established solutions and architectural patterns, avoiding unnecessary invention and reducing delivery risk.
* **Adaptability**: A well-decomposed system can respond to change more easily. When policy or business requirements shift, bounded contexts can evolve independently without ripple effects across the system.
* **Extensibility**: New features can be added cleanly—either by extending an existing bounded context or introducing a new one. Modular subdomains reduce the chance of large-scale regression or rework.
* **Traceability and Alignment**: Each requirement can be traced to a specific subdomain and context. This alignment improves developer and stakeholder understanding and strengthens governance by making each slice of logic easier to audit and reason about.

# Subdomain Bounding in Practice

Most enterprise systems consist of a mix of recurring functional areas—people, relationships, roles, goals, scheduling, accounts, cost recovery, assessment, reporting, and more. These subdomains appear across industries, although the vocabulary may differ. “Enrolment” in education and “subscription” in SaaS are often functionally equivalent; “grading” and “assessment” are similar; “contact” and “person” differ only in emphasis.

DDD embraces this reality by guiding teams to identify and bound these logical parts rather than treat a domain as an indivisible whole. Each bounded context encapsulates one subdomain’s logic, allowing models to remain focused, adaptable, and independently governed. Many subdomains are well-understood and require little reinvention. Where appropriate, they can be implemented using off-the-shelf components. This allows creative energy to be focused on the genuinely novel aspects of the system—which are relatively rare in enterprise work.

## Problem Cartography

A helpful metaphor for domain mapping is cartography itself. Just as a physical landscape is composed of rivers, hills, and forests—each with distinct characteristics and requiring different management approaches—so too is a problem domain made up of discrete subdomains, each with its own logic, structure, and design needs.

The role of the architect or business analyst working with DDD is to map this landscape. Where stakeholders may describe the system as a single, undifferentiated whole, it is the designer’s job to chart where one subdomain ends and another begins. They must learn to see the "hills" of social structure, the "forests" of aspiration and goal-setting, and the "wetlands" of process-heavy workflows such as task or assessment management.

By drawing clear boundaries around each subdomain, we can apply the most appropriate—and often pre-solved—approach to each. While some subdomains require custom modelling due to their centrality or complexity, others can be addressed using existing products, services, or known design patterns.

For example, a billing subdomain may be managed with a proven billing platform—reflecting a generic, replaceable concern. In contrast, a policy calculation engine in an insurance system is likely to be core and organisation-specific, and so requires careful, bespoke design. A clear domain map guides these decisions, ensuring each part of the landscape is approached with the right tools, governance, and focus.

A more complex example can be found in education. What is often described in sector-specific terms—learners, schools, classes, enrolment, scheduling, attendance, curriculum, homework, assignments, grading—can often be reinterpreted through more general and reusable domains. These include social structures (people and roles), subscriptions (enrolment and access), artefacts (resources), aspirations (learning objectives), scheduling (timetables), work (tasks), and assessments (grading and moderation).

While the designer continues to engage with the customer in their own “ubiquitous language” to preserve clarity and trust, the underlying system is mapped onto well-defined, well-trodden, and previously solved subdomain schemas and logic. This approach significantly reduces project unknowns and delivery risks by replacing novelty with clarity and reuse.

## Misapplication Risks

Beyond the misuse and misconceptualisation of DDD outlined earlier, there remain several other risks that must be called out explicitly.

### Non-Abstraction

One of the most common mistakes is to model internal system objects directly on external business concepts without sufficient abstraction. This practice encodes a narrow and rigid understanding of the domain—one that is rarely extensible or adaptable. Domain-Driven Design demands careful work to separate the surface structure of human language and business terms from the underlying logic required for robust, maintainable systems. The role of DDD is to create this abstraction layer between what the business articulates and how it is implemented in technology.

### Too Little Separation

Related to the above is the risk that developers may consider the division between "Business" and "System" as sufficient. However, in DDD, this is only the starting point. Real understanding comes from decomposing the problem space into core subdomains that reflect recurring human and organisational structures.

Both of these domains can be subdivided further. We demonstrated subdivision of a Business domain earlier, but “System” also isn’t a sufficiently fine-grained concept to demonstrate thought, understanding, and therefore achievability. Web systems are composites of subdomains such as diagnostics, configuration, settings, routing, sessions, and access control. While development frameworks often provide these as bundled features, understanding their boundaries improves predictability and reduces the risk of design error and delivery failure.

### Platform Based Domain Design

Another critical risk is the misbelief that recognising a domain is sufficient—and that solving it simply involves building on top of a platform that is “approximately right.”

A frequent failure point in domain thinking is the assumption that enterprise platforms provide core domain concepts out of the box. Managers and vendors may reasonably believe that tools like Salesforce, Microsoft Dynamics, or other enterprise products deliver architectural alignment, especially given their rich configuration capabilities and marketing claims. But there is an important distinction between what platforms support and what they define.

Most enterprise platforms were not built from first principles of domain modelling. Instead, they evolved from narrow business cases—sales pipelines, resource scheduling, or content collaboration. Their growing generality is a result of added programmability, not a foundational commitment to abstract domain representation.

Salesforce, for example, is grounded in customer relationship logic. It manages interactions well but treats people as contacts, leads, or accounts. Microsoft Dynamics is shaped by its service and resource origins. Confluence, while powerful as a wiki, is centred on page creation and linking, not modelling goals or structured activity. Even Microsoft 365, despite its breadth, is not a system in the architectural sense. It offers tools for document management, messaging, and task tracking, but lacks any unifying concept of aspiration, membership, or institutional structure—though it can store artefacts about these things.

Organisations are right to adopt such platforms. Their operational stability, vendor support, and flexibility provide real value. But it is a mistake to treat them as architectural foundations. At best, they can host carefully designed models. At worst, they distort mental models to fit tool constraints rather than the actual problem.

This does not invalidate existing investments. These platforms should be seen as transitional stages in a maturing capability journey. Low-code and configuration-first approaches are often pragmatic when technical capacity is limited. However, the true failure point in such projects is rarely the platform itself—it is the absence of a shared, rigorous understanding of what the system is meant to represent. That failure is equally dangerous whether implemented in a commercial product, custom codebase, or no-code tool.

**The path forward is not to discard platforms, but to outgrow dependence on them.** Projects should mature from reliance on products, to reliance on partners who can shape them, and eventually to internal capability. The essential first step is analysis: the ability to define, decompose, and validate models that reflect enduring concerns. This is where Domain-Driven Design begins to deliver strategic value.

DDD is not a luxury reserved for large systems. It is an essential discipline for any system that must evolve, interoperate, or endure. Its strength lies in providing clarity—clarity about what the system represents, how it is structured, and how each part can change without destabilising the whole.

# DDD Core Concepts for All Team Members

At its heart, Domain-Driven Design involves building a system around the real-world problems it is meant to solve, by separating and modelling the underlying domains of knowledge and activity.

The key concepts of DDD include:

* Domains: Areas of knowledge and activity with consistent purpose, vocabulary, and internal rules
* Bounded Contexts: Clear boundaries around each domain’s model, language, and logic, enabling it to evolve without disrupting unrelated parts
* Ubiquitous Language: Shared language between domain experts and developers within a bounded context
* Entities and Value Objects: Distinct domain concepts with identity (Entities) or state (Value Objects)
* Aggregates and Roots: Structures that maintain consistency within transactional boundaries
* Domain Services: Operations meaningful to the domain that don’t belong to a specific Entity

These ideas apply across the delivery team: not just to developers, but to analysts, testers, project leads, and stakeholders. They shape how we understand the system’s purpose and define its architecture.

Object-oriented practitioners will recognise many of the patterns they use being reapplied in DDD: the separation of concerns between layers and responsibilities, the use of stateful entities to represent continuity of identity, and stateless services that support those entities without managing their state directly. These are familiar principles, but in DDD they are applied not just at the object level, but across the entire domain model and architectural structure.

This architectural framing supports clear layering, domain isolation, and multi-schema composition. For example, a business-facing education service may coordinate interactions across subdomains such as Aspirations, Scheduling, Effort, and Assessment—each within its own schema, but aligned through shared models and contracts.

This layered view will now be developed in detail. how each major subdomain can be defined, reused, and combined. It will also demonstrate how to structure project discovery and analysis so that these subdomains emerge early, and how to use them to drive model stability, testability, and technical delivery.

# Domain Thinking and Subdomains

The key point to highlight is that recurring domains are often pre-solved. This significantly reduces the delivery risk associated with designing novel systems tailored too closely to one organisation's terminology or current operating patterns. Instead of modelling from phrasing that may be temporary or parochial, DDD encourages identifying stable subdomains that transcend individual use cases. These subdomains provide reusable frames that improve adaptability and reduce rework when policy, personnel, or practice change.

The following are a list of classic subdomains that underpin most business domains.

* The Social domain deals with people, personas, and their relationships.
* The Assets or registries domain tracks state that belong to groups or persons.
* The Artefacts or Media domain deals with the information and media used or produced by people
* The Aspiration domain models the goals, objectives, and milestones of people and groups.
* The Work domain models projects tasks and effort expended by people towards the target aspirations.
* The Scheduling domain supports coordinating work if done by more than one.
* The Assessments domain cover determining the quality of the work expended as compared to target objectives.
* (Others may be added based on context.)

These domains are not specific to education, finance, or health—they underpin all of them.

Recognising this allows thought first, then discussions using language specific to the subdomains, then systems to be built -- using stable, well-bounded logical then building blocks.

This thinking-first approach allows stakeholders to engage using language appropriate to their context, while designers work with stable, interoperable structures. The result is a system that avoids overfitting, remains maintainable, and is positioned to adapt and interoperate well into the future.

# Illustrative Domain Entities

To support practical application, the following section outlines lightweight entity examples across core subdomains. These are not implementation schemas, but indicative structures showing how common concepts repeat across contexts.

## System Domain

The System domain supports operational and technical aspects of applications. Though often supplied by frameworks, understanding these boundaries is essential to system-level modelling and governance.

* **Routing**: Route, PathPattern, TargetAction
* **Access Control**: User, Role, Permission, Policy, AccessRule
* **Sessions**: Session, SessionToken, LoginState
* **Diagnostics**: LogEntry, Event, Metric, HealthCheck
* **Configuration**: Setting, Environment, FeatureFlag
* **Accounts/Tenancy**: Account, AccountProfile, SubscriptionPlan

## Social Domain

Captures people, their contextual identities, and the roles they occupy in groups.

* **Person**
* **Persona**: Contextual role facet of a person
* **Group**: Organisation, team, class, etc.
* **Role**: Position held within a Group
* **Relationship**: Guardian-of, member-of, assigned-to, etc.

## Aspiration Domain

Represents the goals, intentions, and target outcomes of people and groups.

* **Target**: Desired future state
* **Pathway**: Strategy or route toward target
* **Objective**: Concrete step or milestone toward a goal

## Work Domain

Models effort and delivery.

* **Project**: Bounded initiative or body of work
* **Task**: Actionable unit of effort
* **Contribution**: Relationship linking a Persona to a Task
* **Effort**: Hours, interactions, or work logs
* **Artefact**: Deliverables or work products

## Scheduling Domain

Handles coordination of work over time.

* **Event**
* **Schedule**
* **TimeBlock**
* **Availability**
* **RecurringPattern**

## Assessment Domain

Captures evaluation of effort and achievement. Applicable to learning, performance, compliance, or quality scenarios.

* **Assessment**: Evaluation instance
* **Rubric**: Criteria or framework used to assess
* **Score**: Numeric or descriptive result
* **Feedback**: Comments, suggestions, or reflection
* **AchievedLevel**: Outcome indicator (pass/fail, level reached, etc.)

Each of the example domains presented above is stable across industries and systems.

They may be named differently or used in varied combinations, but the structural roles they play are remarkably consistent. Mapping to these foundations helps create modular, interoperable, and long-lived systems.

## Conclusion

Domain-Driven Design is not a framework, methodology, or technology stack—it is a way of thinking. It shifts the focus of system design away from surface features and technical output, and instead centres on clear, stable understanding of the problem space.

The greatest value DDD provides is not in what it builds, but in what it helps avoid: unclear models, misplaced responsibilities, fragile architectures, and high-friction change. By focusing on the core domains that shape an organisation’s operations, DDD enables delivery teams to structure systems that are understandable, testable, and resilient over time.

This guidance is not just for developers. It applies equally to analysts, architects, testers, product owners, and decision-makers. Clarity of model enables clarity of purpose. And clarity of purpose reduces risk.

When applied with discipline and humility, Domain-Driven Design becomes more than an approach to system development. It becomes a shared language for meaningful digital change. One that allows teams to build less, but achieve more.

Appendices

Appendix A - Document Information

Authors & Collaborators

* Sky Sigal, Solution Architect

### Versions

* 1. Initial Draft
  2. Redo

### Images

[Figure 1: TODO Image 2](#_Toc144995112)

### Tables

[Table 1: TODO Table 3](#_Toc145048484)

[Table 2: TODO Table 2 3](#_Toc145048485)

### References

**There are no sources in the current document.**

### Review Distribution

The document was distributed for review as below:

|  |  |
| --- | --- |
| Identity | Notes |
|  |  |
|  |  |
|  |  |

### Audience

The document is technical in nature, but parts are expected to be read and/or validated by a non-technical audience.

### Structure

Where possible, the document structure is guided by either ISO-\* standards or best practice.

### Diagrams

Diagrams are developed for a wide audience. Unless specifically for a technical audience, where the use of industry standard diagram types (ArchiMate, UML, C4), is appropriate, diagrams are developed as simple “box & line” monochrome diagrams.

### Acronyms

API

: [Application Programming Interface](#Term_ApplicationProgrammingInterface).

DDD

: Domain Driven Design

GUI

: [Graphical User Interface](#Term_ApplicationProgrammingInterface). A form of [UI](#Acronym_UI).

ICT

: acronym for Information & Communication Technology, the domain of defining Information elements and using technology to automate their communication between entities. [IT](#Acronym_IT) is a subset of ICT.

IT

: acronym for Information, using Technology to automate and facilitate its management.

UI

: User Interface. Contrast with [API](#Acronym_API).

### Terms

Refer to the project’s Glossary.

Application Programming Interface

: an Interface provided for other systems to invoke (as opposed to User Interfaces).

Capability

: a capability is what an organisation or system must be able to achieve to meet its goals. Each capability belongs to a domain and is realised through one or more functions that, together, deliver the intended outcome within that area of concern.

Domain

: a domain is a defined area of knowledge, responsibility, or activity within an organisation or system. It groups related capabilities, entities, and functions that collectively serve a common purpose. Each capability belongs to a domain, and each function operates within one.

Entity

: an entity is a core object of interest within a domain, usually representing a person, place, thing, or event that holds information and can change over time, such as a Student, School, or Enrolment.

Function

: a function is a specific task or operation performed by a system, process, or person. Functions work together to enable a capability to be carried out. Each function operates within a domain and supports the delivery of one or more capabilities.

Person

: a physical person, who has one or more Personas. Not necessarily a system User.

Persona

: a facet that a Person presents to a Group of some kind.

Quality

: a quality is a measurable or observable attribute of a system or outcome that indicates how well it meets expectations. Examples include reliability, usability, and performance. Refer to the ISO-25000 SQuaRE series of standards.

User

: a human user of a system via its UIs.

User Interface

: a system interface intended for use by system users. Most computer system UIs are Graphics User Interfaces ([GUI](#Acronym_GUI)) or Text/Console User Interfaces (TUI).