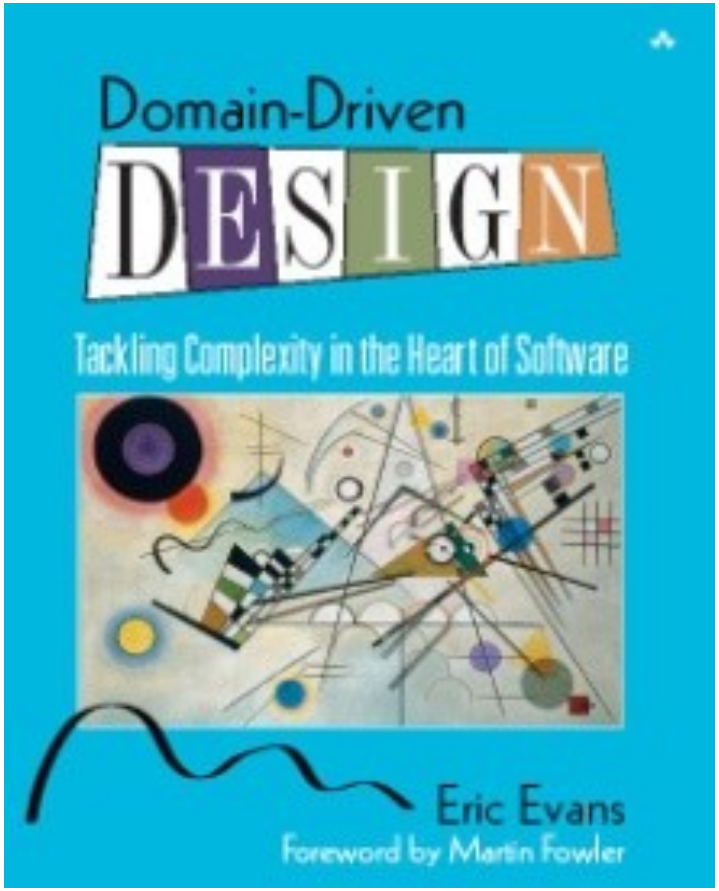
Domain Driven Design

Based on slides from (slideshare available)

Tom Kocjan



What is DDD?

- It is a way of thinking and a set of priorities, aimed at accelerating software projects that have to deal with **complicated domains**.
- A **Model driven** software design approach used to tackle the complexity of software projects.
- Collection of principles and patterns that help developers craft **elegant systems.**

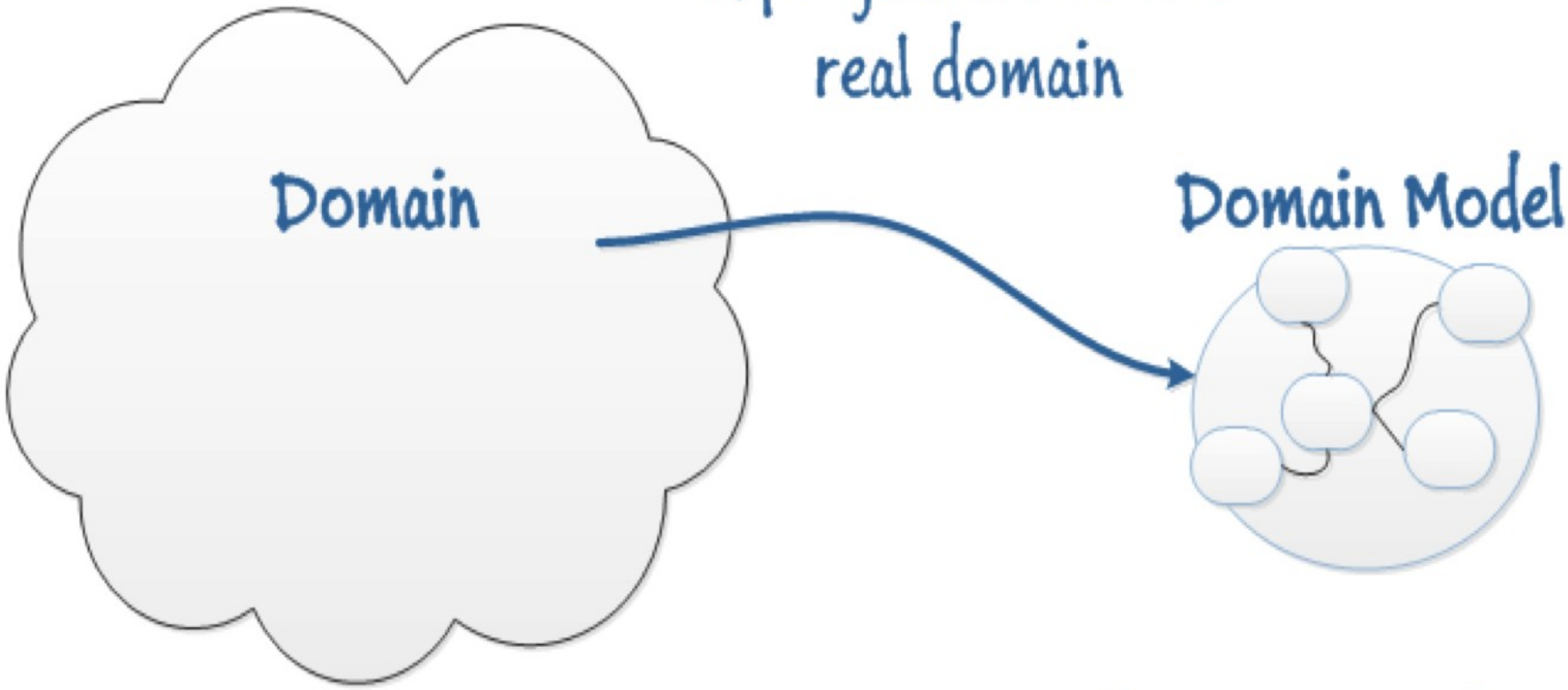
Definitions

Domain A sphere of knowledge or activity. What an organization does and the world it does it in.

Model A system of abstractions that describes selected aspects of a domain and ignores extraneous detail. Explains a complex domain in a simple way.

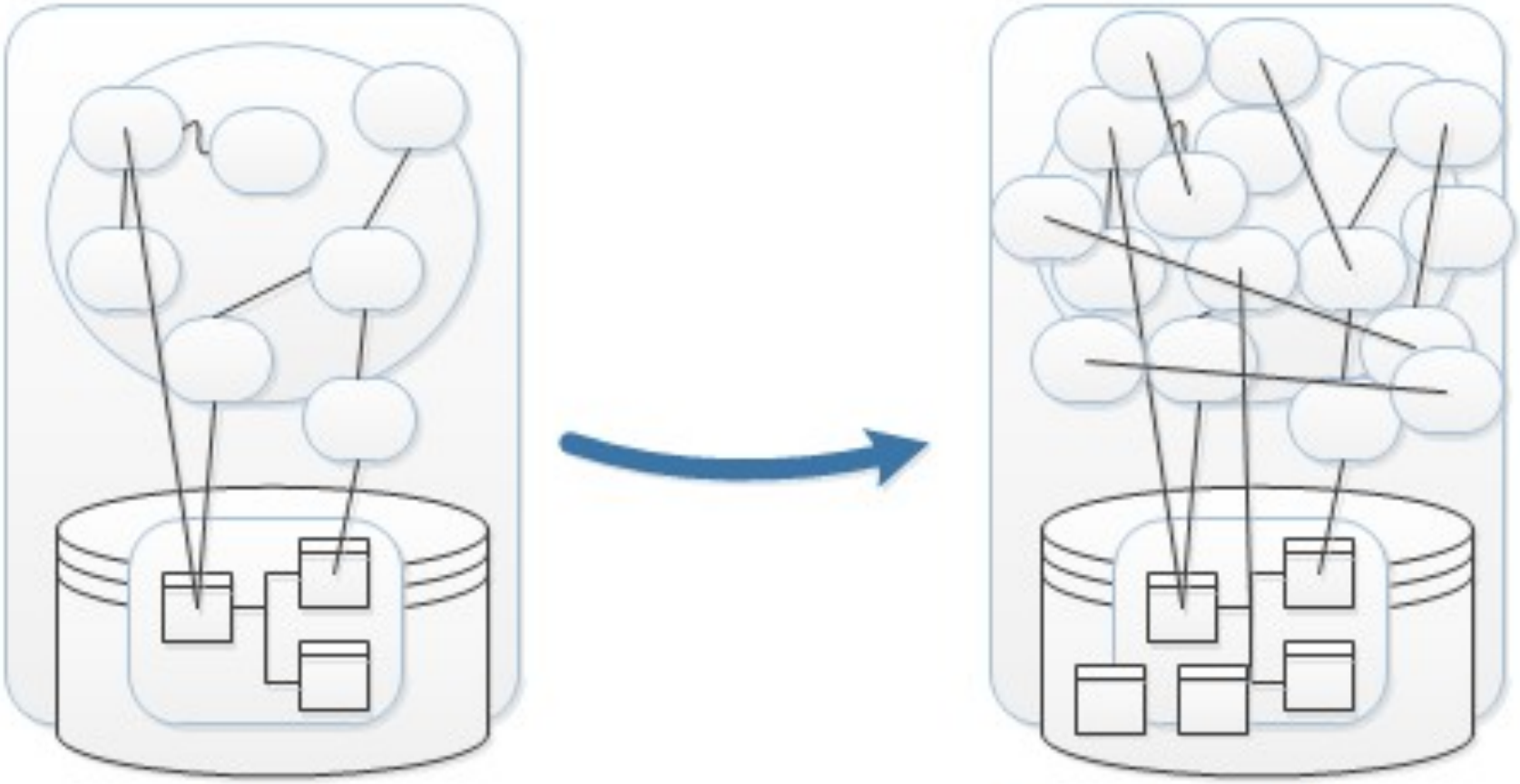
A **model** is a distilled form of **domain** knowledge, assumptions, rules and choices.

A projection of the real domain



The reality

An abstraction of reality designed to manage complexity for specific business cases



Initial software incarnation fast to produce

Over time without care and consideration software turns to ball of mud

DDD is a Set of Driving Principles

- Speak a **Ubiquitous Language** within an explicitly Bounded Context.
- Explore **models** in a creative collaboration of domain practitioners and software practitioners.
- Focus on the **Core Domain**.
- Model and implementation are bound... developers are also responsible for the model.

Ubiquitous Language

Ubiquitous Language A language structured around the domain model and used by all team members to connect all the activities of the team with the software.

Use it consistently in speech, documentation, diagrams and code.

A Change in Ubiquitous Language <=> Change in Model and Code.

The model we want…

- Helps us solve specific problems in our domain.
- Is not necessarily “realistic”.
- Forms the basis of a language.
- Should remain current.

London Tube Map

Expressing the Model

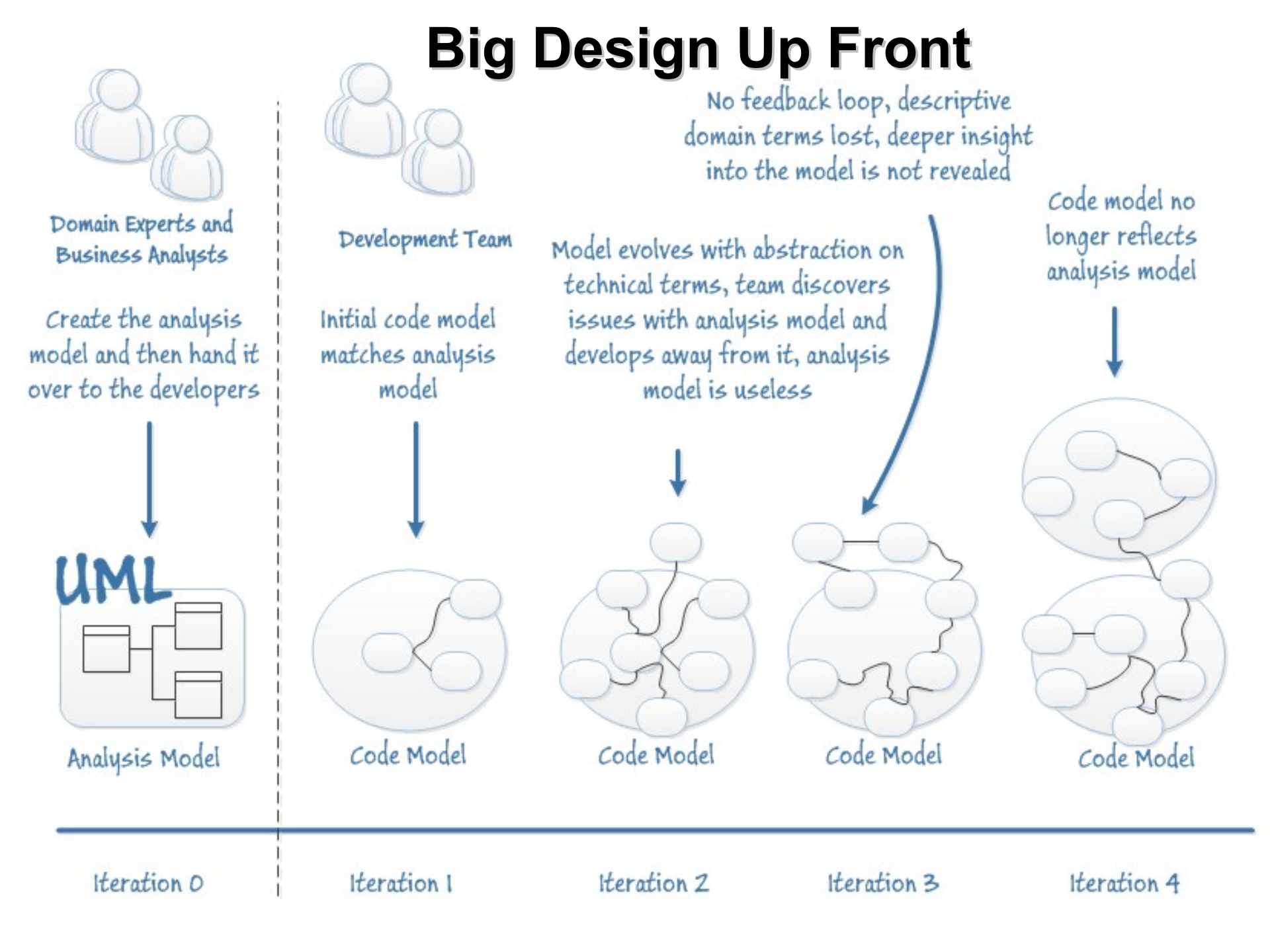
- The model can be expressed through class diagrams, explanatory diagrams, sequence diagrams or whatever conveys the model.
- **But, the design document is not the model!** The design document's purpose is to help communicate and explain the model.
- The model is ultimately expressed in the code.

DDD is Agile and Iterative

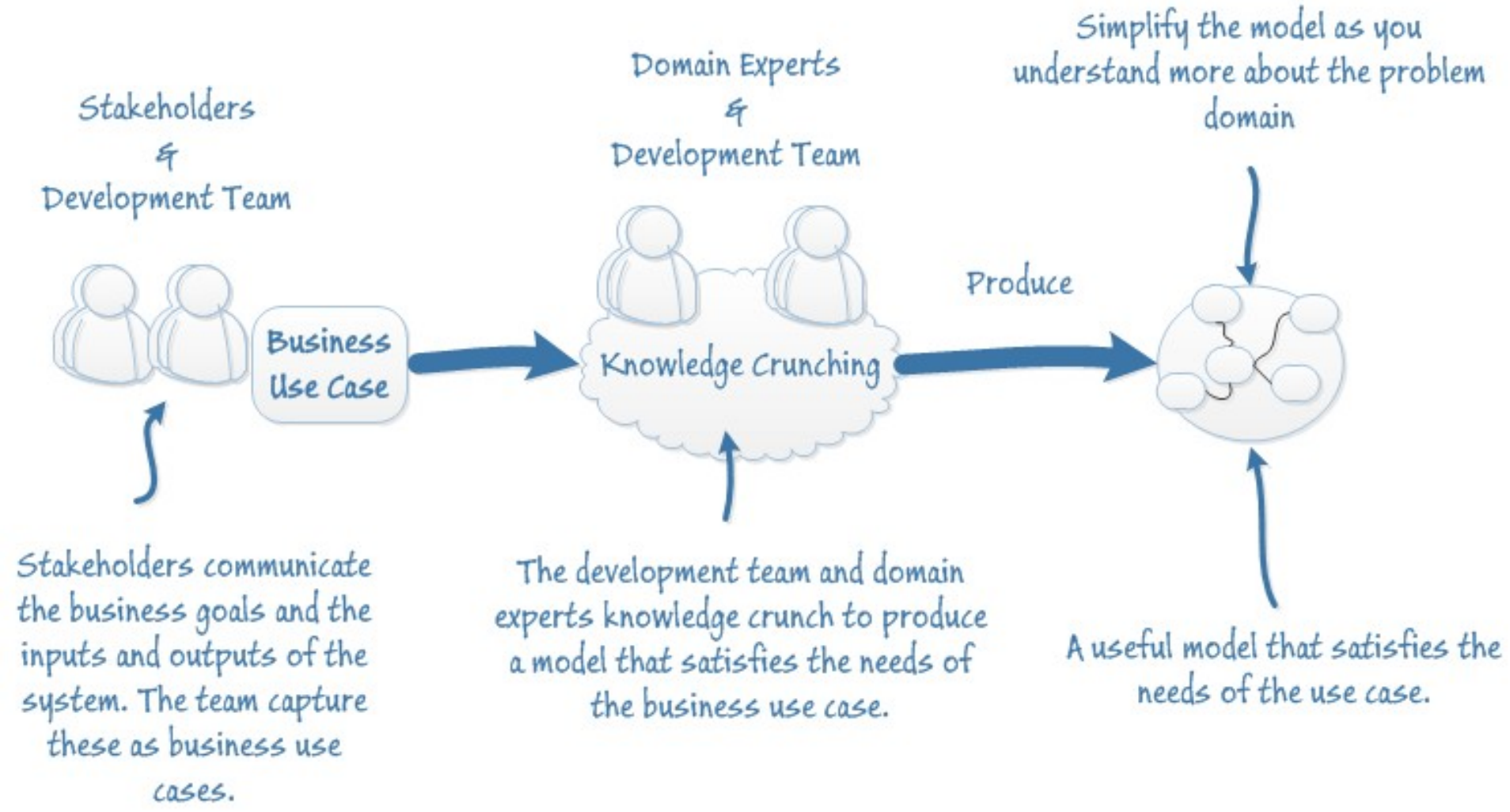
The problem with Big Design Up Front:

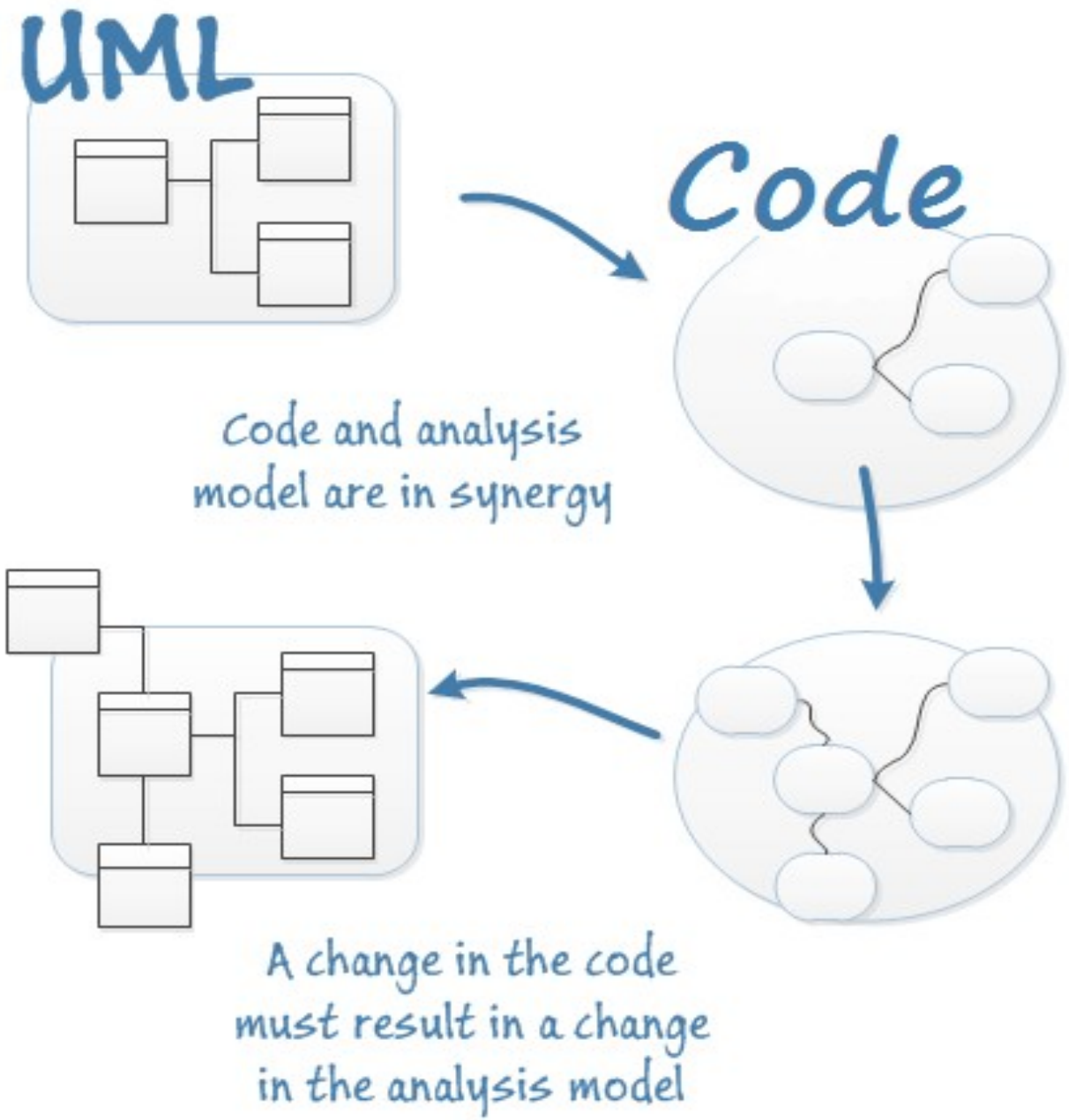
- Models are distilled knowledge.
- At the beginning of a project, the team is as ignorant as it will ever be.
- **Up Front Analysis Locks in Ignorance!**

Big Design Up Front

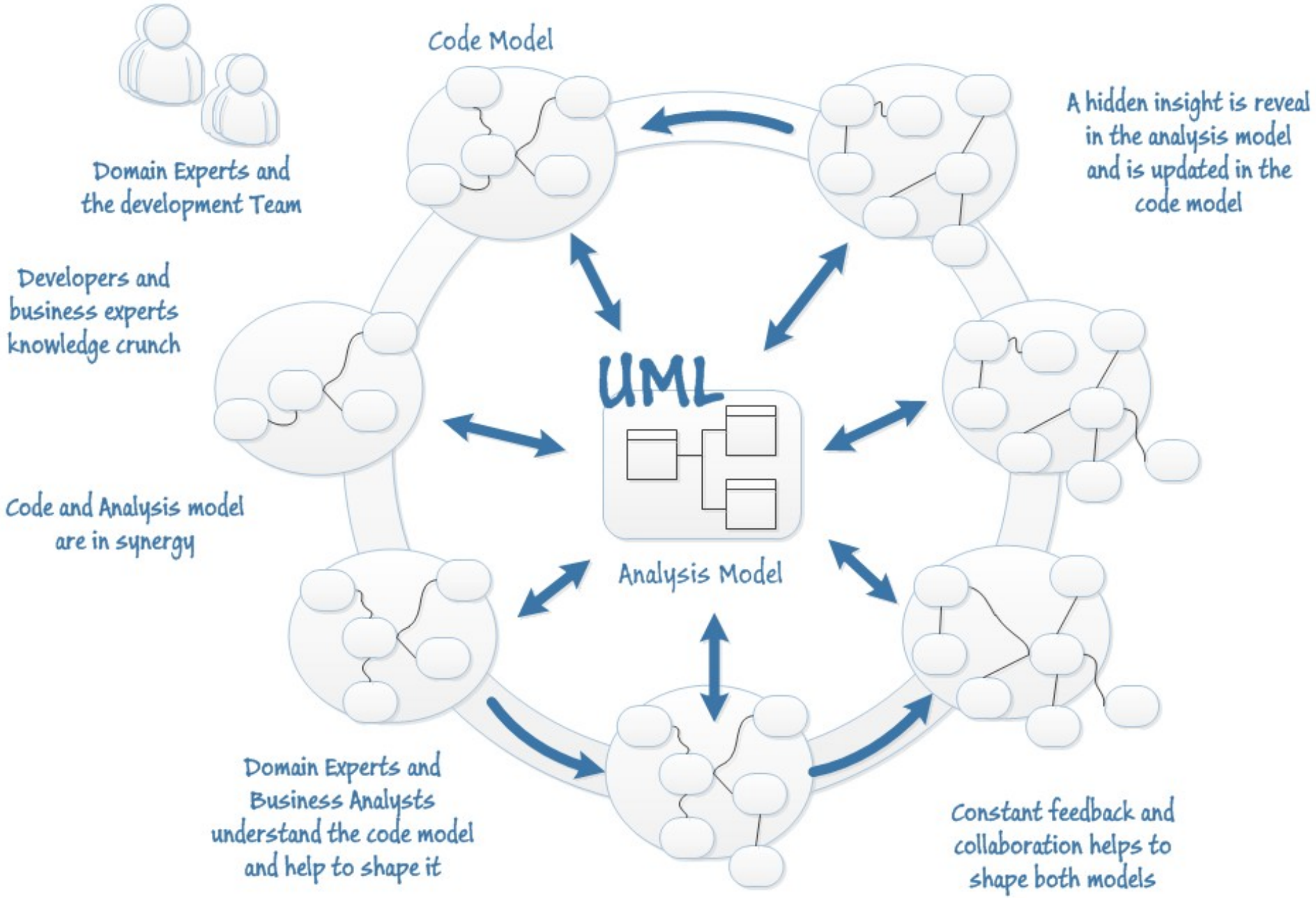


The DDD Process

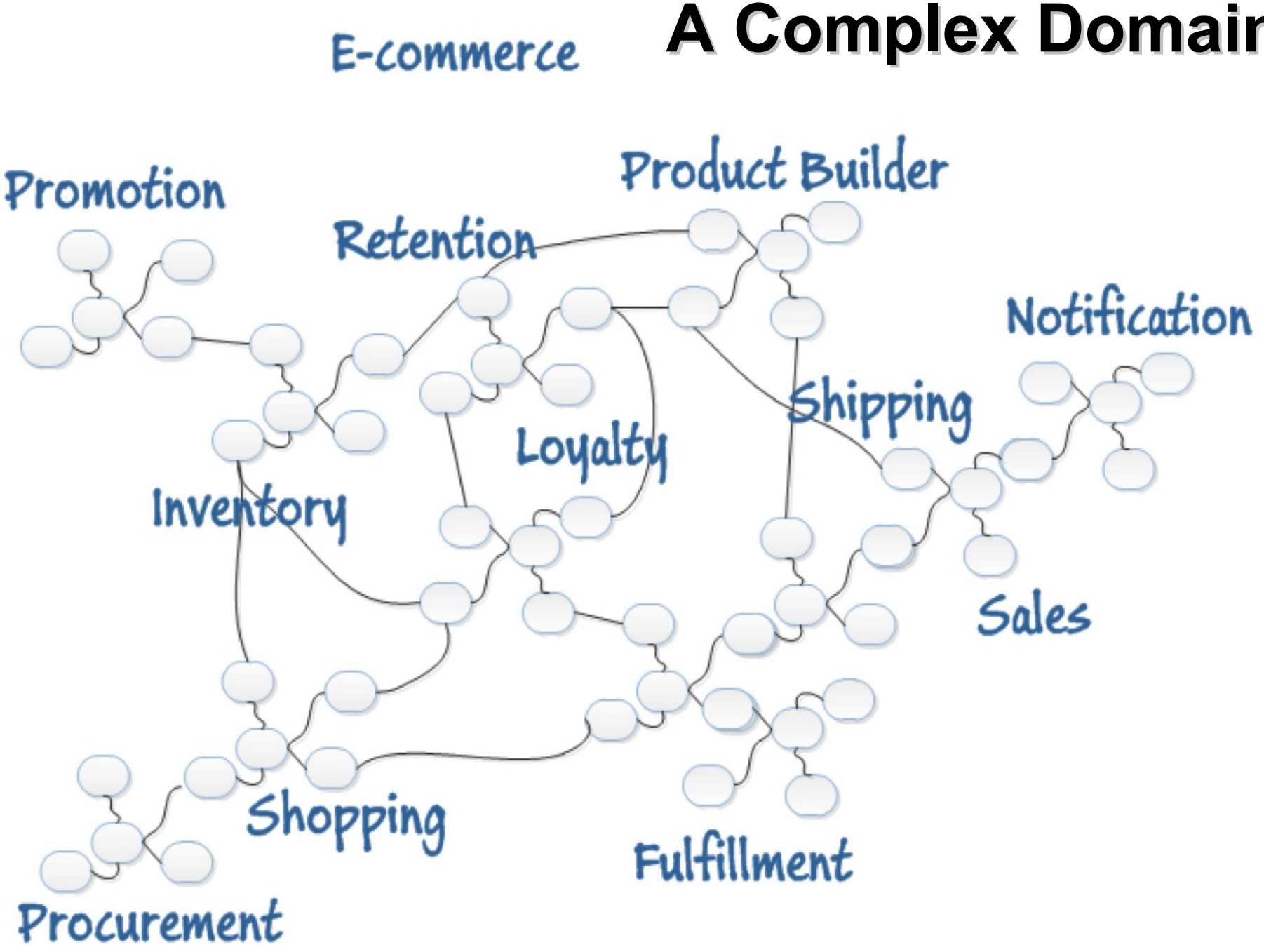



UML
Code
Code and analysis
model are in synergy
A change in the code
must result in a change
in the analysis model

One Team, One Language, One Model



A Complex Domain

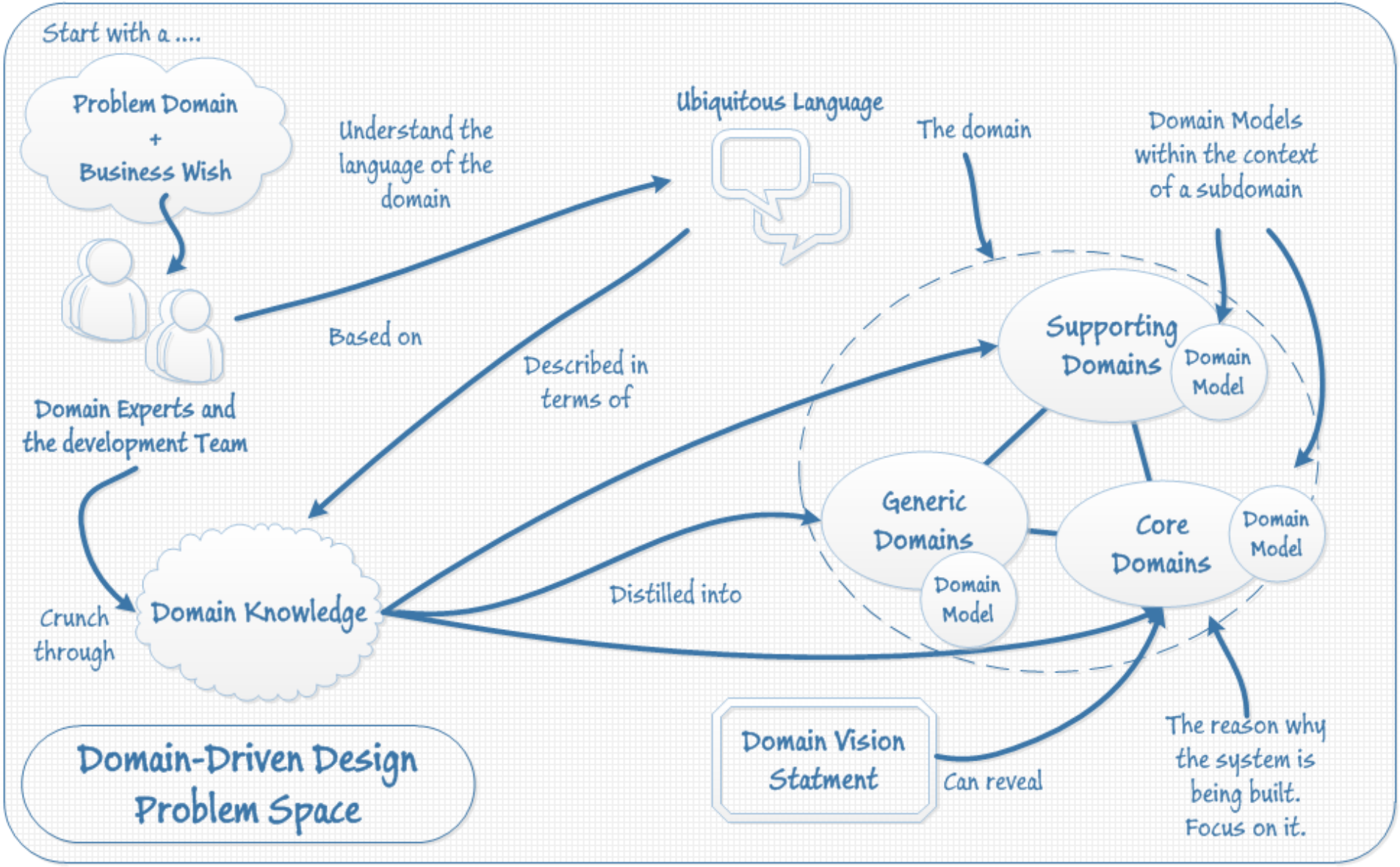


Breaking Down a Complex Domain

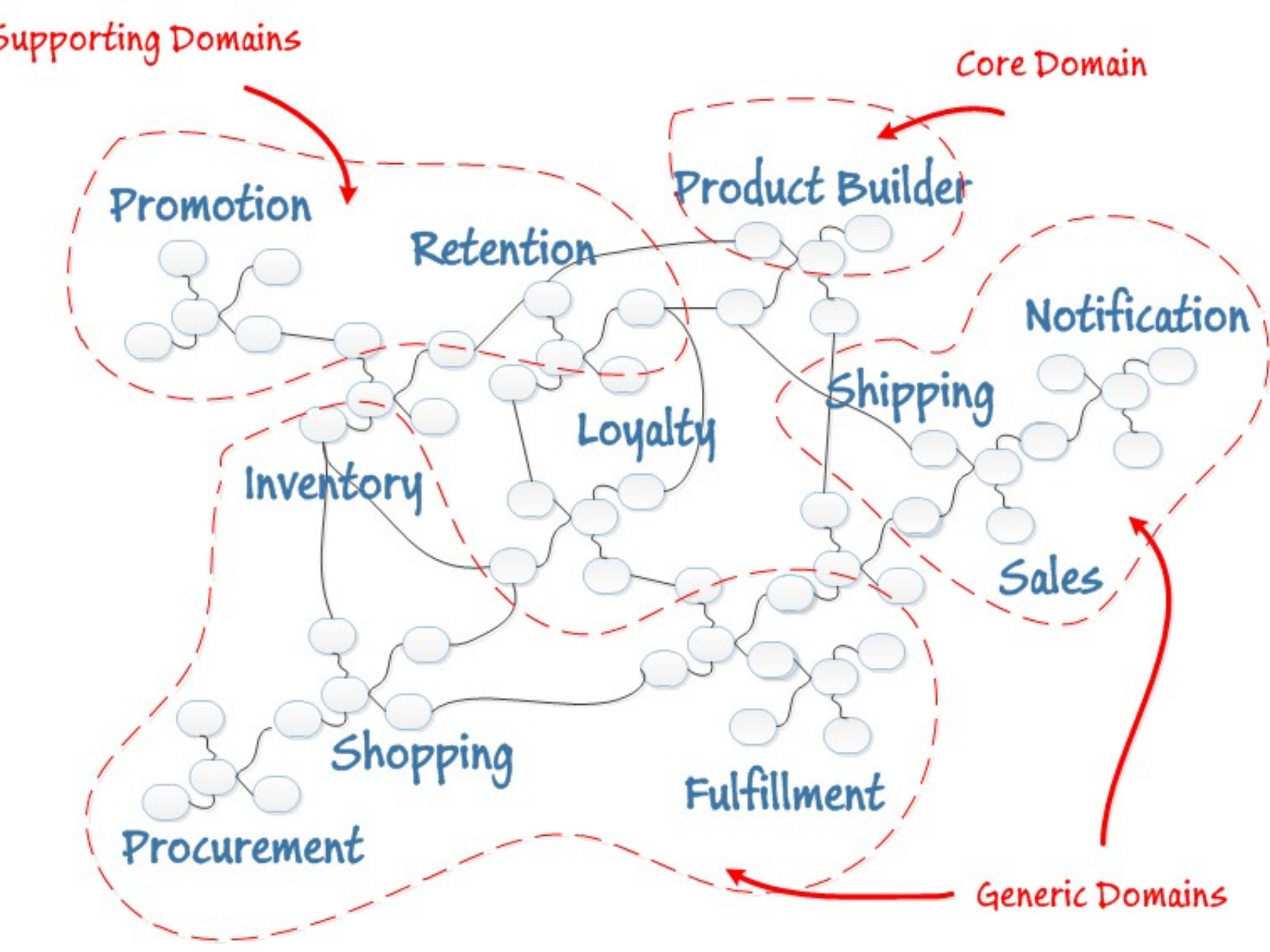
Bounded Context An operational definition of where a particular model is well-defined and applicable. (Typically a subsystem, or the work owned by a particular team).

Subdomain Part of the domain, based on a particular conceptual decomposition of the domain.

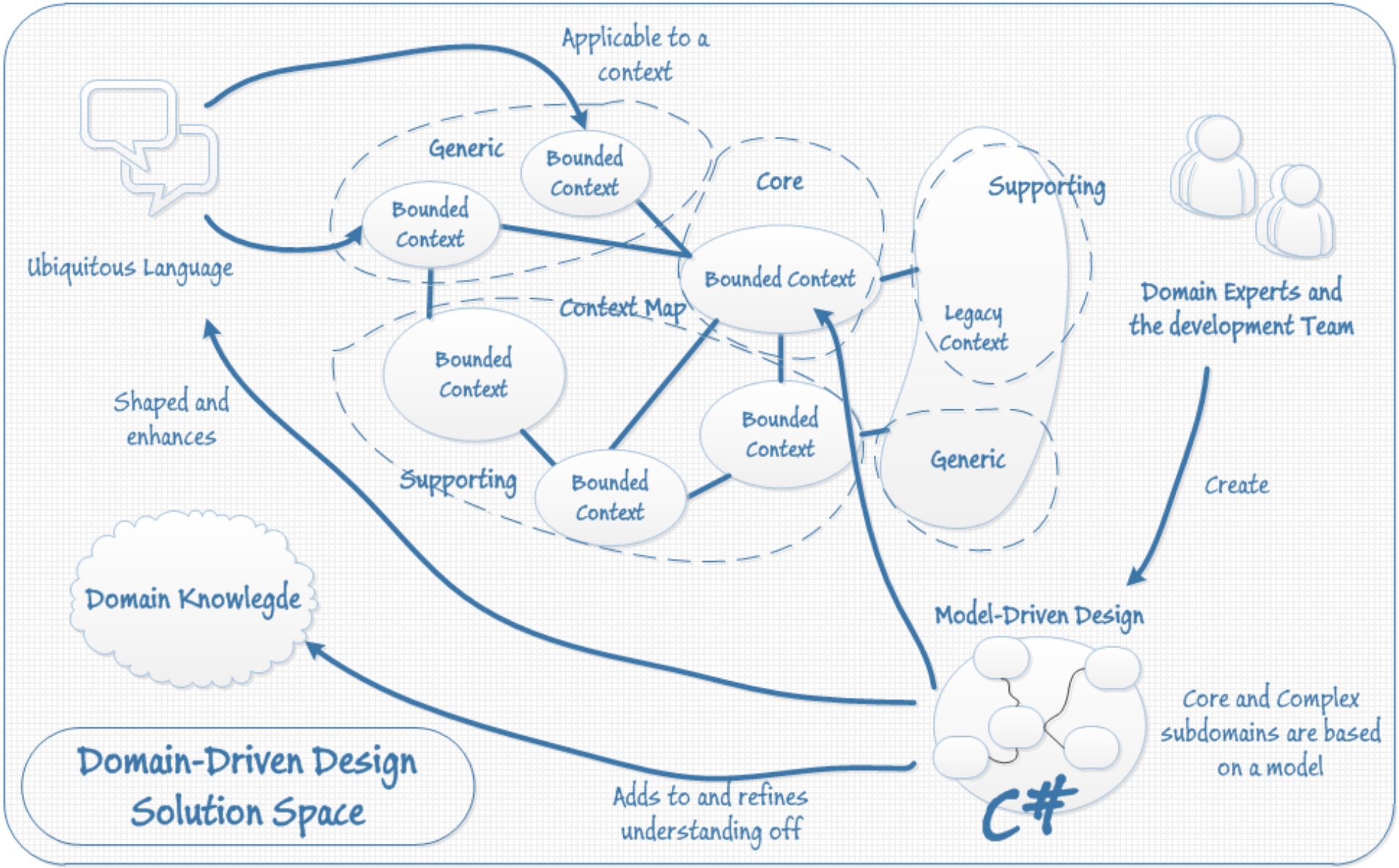
The Problem Space



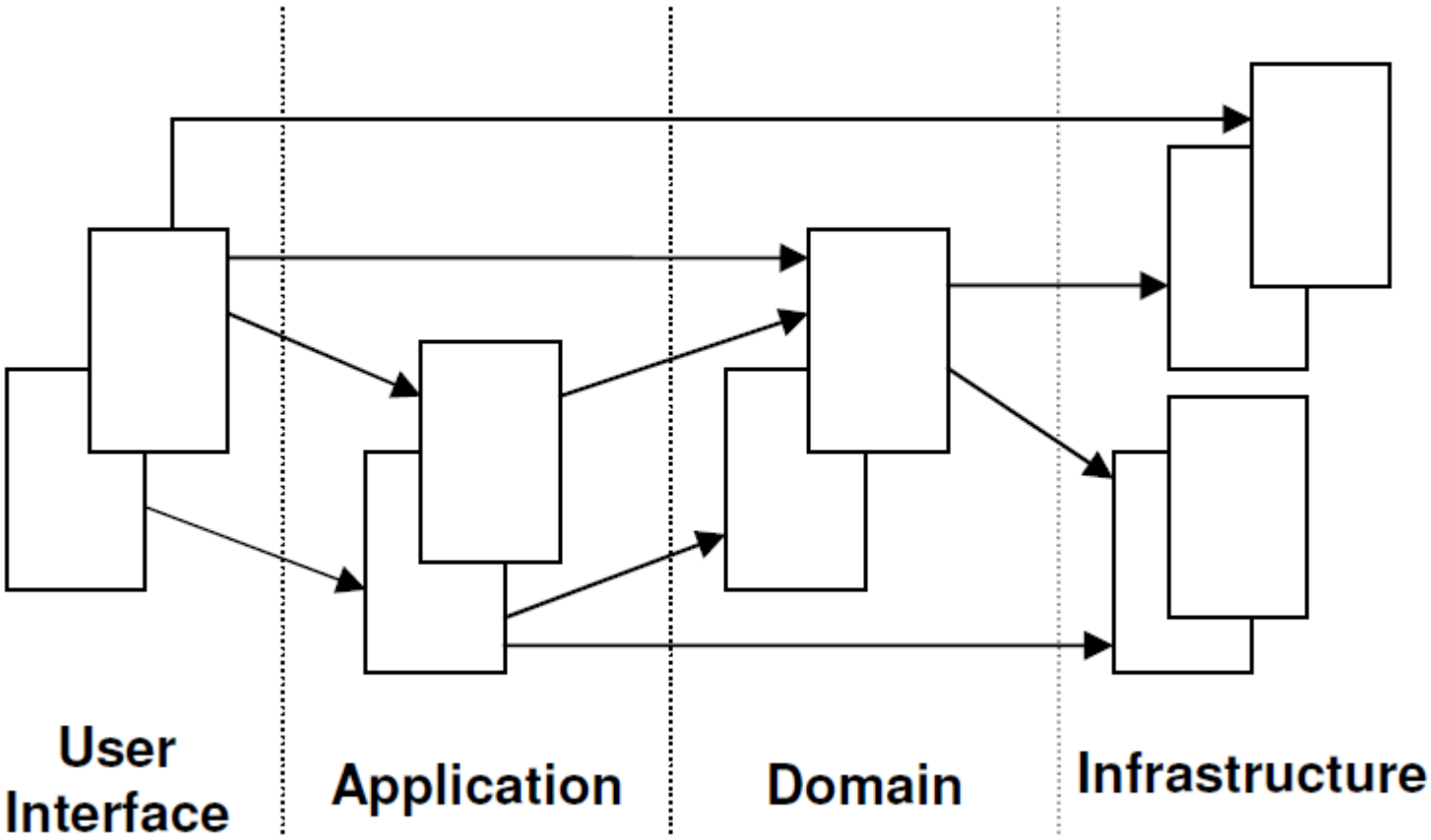

Other parts of the domain support the core subdomain
The core most important part of this domain
Promotion
Retention
Product Builder
Notification
Shipping
Loyalty
Inventory
Sales
Shopping
Fulfillment
Procurement
Generic parts of the domain distract from the core part


Supporting Domains
Core Domain
Promotion
Retention
Product Builder
Notification
Shipping
Loyalty
Inventory
Sales
Shopping
Fulfillment
Procurement
Generic Domains

The Solution Space



ayered Architecture



Layers

User Interface Responsible for presenting information to the user and interpreting user commands.

Application This is a thin layer which coordinates the application activity. **It does not contain business logic. It does not hold the state of the business objects.**

Domain This layer contains information about the domain. This is the heart of the business software. The state of business objects is held here. **Persistence details delegated to the Infrastructure layer.**

Infrastructure This layer acts as a supporting library for all the other Layers. It provides communication between layers, implements persistence for business objects, contains supporting libraries for the user interface layer, etc.

Model Expressed in Software

- Entities
- Value Objects
- Aggregates
- Services
- Associations
- Factories
- Repositories
- Modules
- Context Mapping

Entities

- Objects which have an identity which remains the same throughout the states of the software.
- Must be distinguished from other similar objects having the same attributes. (e.g. Bank Account for Jim Smith)
- Attributes of an entity can change. (Mutable)
- Entities should have behavior. (Business Logic)
- No persistence behavior!

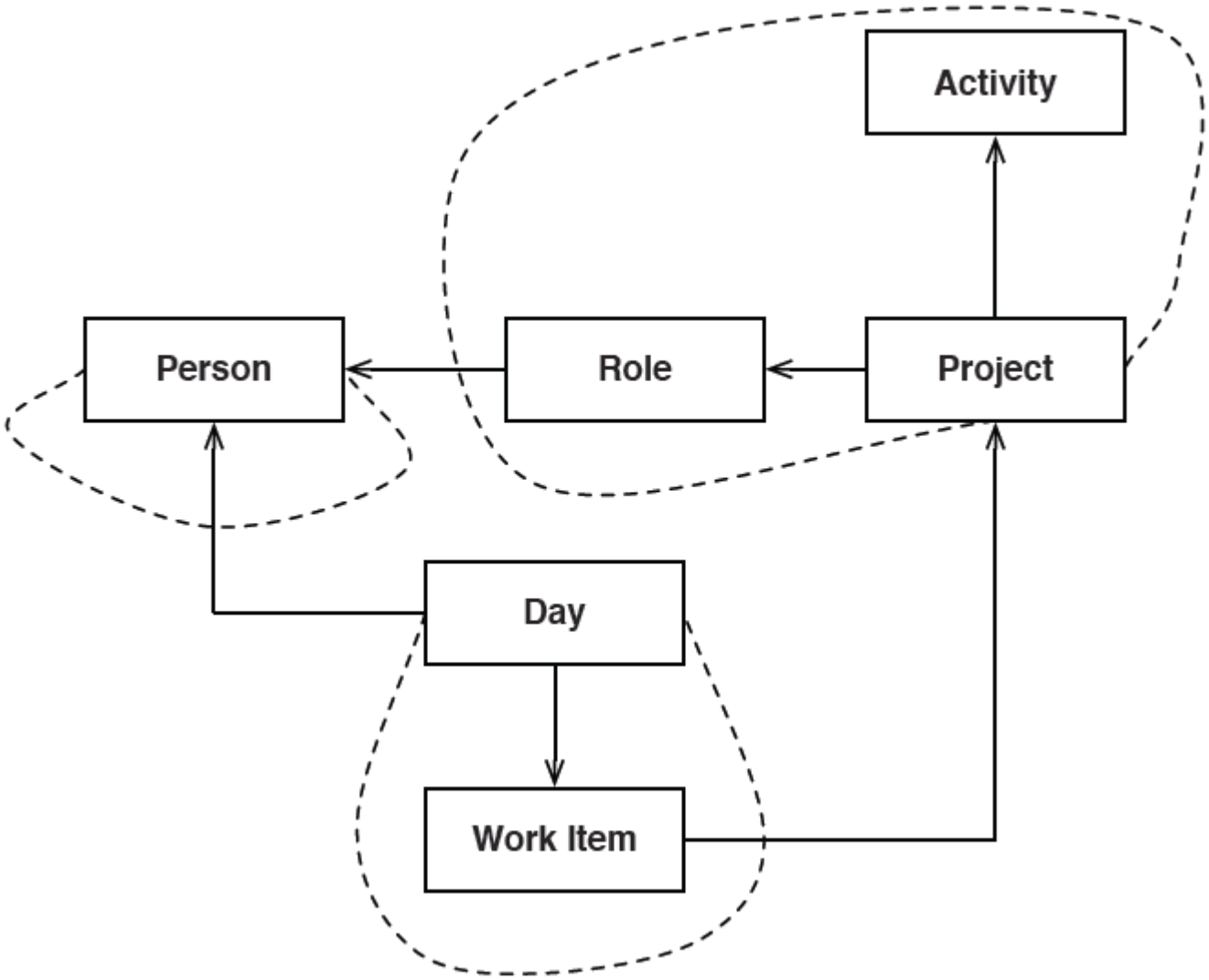
Value Objects

- Value Objects are the “things” within your model that have no uniqueness.
- They are only equal to another Value Object if all their attributes match.
- Value Objects are interchangeable.
- Attributes of a Value Object cannot change, they must be replaced. (Immutable)
- Examples: Money, Address (usually), DTO

Aggregates

- An Aggregate is a cluster of Entities and Value objects. (Object Graph) Each aggregate is treated as one single unit.
- Each Aggregate has one root entity know as the **Aggregate Root**.
- The root identity is global, the identities of entities inside are local.
- External objects may have references only to root.
- Internal objects cannot be changed outside the Aggregate.

Aggregates



Associations

- Impose a traversal direction.
- Add a qualifier, reduce multiplicity.
- Eliminate non essential associations.

Services Reside in Multiple Layers

Application "if the banking application can convert and export our transactions into a spreadsheet file... that export is an APPLICATION SERVICE"

Domain "If a service were devised to make appropriate debits and credits for a found transfer, that capability would be a DOMAIN SERVICE"

Infrastructure "a bank might have an application that sends an e-mail. The interface that encapsulates the email system is an INFRASTRUCTURE SERVICE"

Domain Services

- A Domain Service is Business Logic in the domain that are not a natural part of an Entity or Value Object.
- Services usually manipulate multiple Entities and Value Objects.
- **Services are stateless!**
- A service has to be offered as an interface that is defined as a part of the model. Its parameters and results should be domain objects.

Factories

- An object whose responsibility is the creation of other objects.
- Create and manage complex domain objects.
- Especially useful for creating aggregates.

Repositories

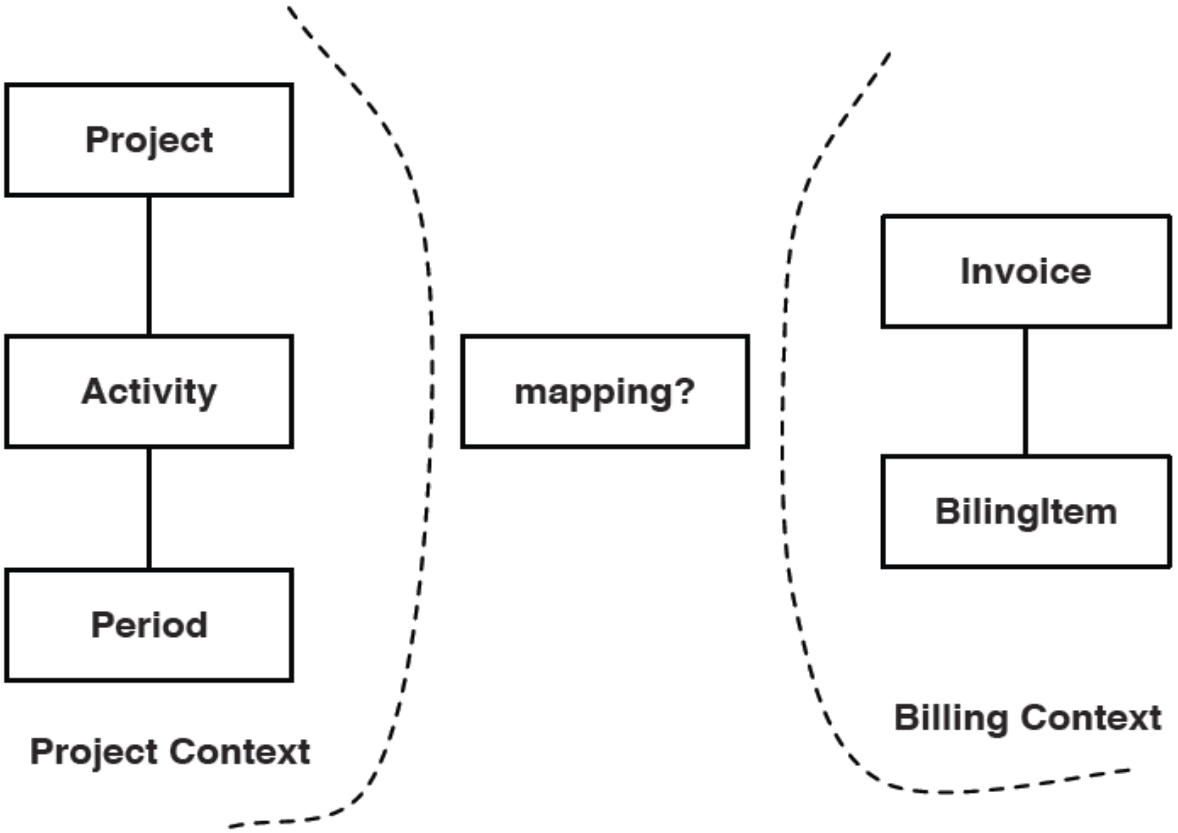
- A Repository encapsulates domain objects persistence and retrieval.
- Clean separation and one-way dependency between the domain and data mapping layers.
- May encapsulate different fetching strategies, distributed caching, NoSQL, Web Service, etc.
- Acts as a collection except with more elaborate querying capability.
- **One Repository per Aggregate!**

Modules

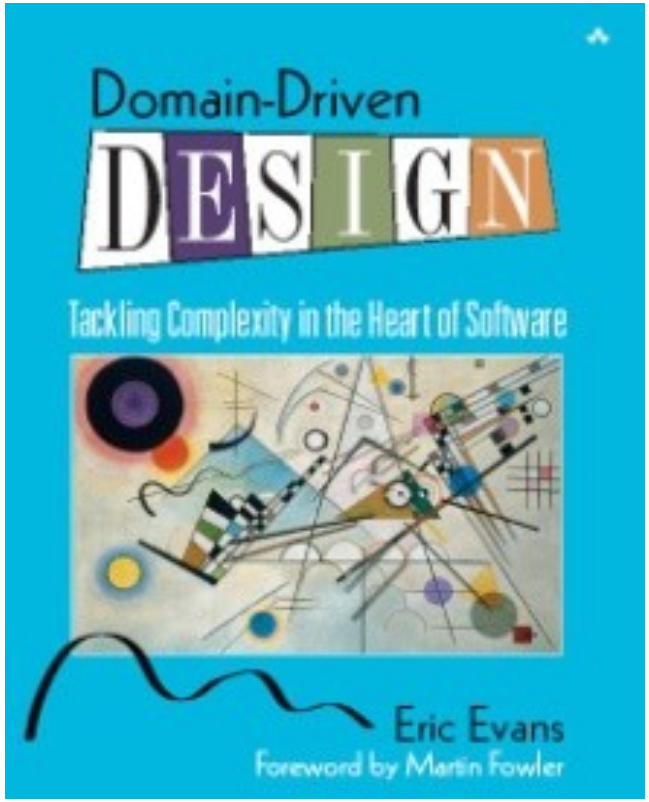
- Break up your domain to reduce complexity
- High cohesion within module, loose coupling between modules.
- Becomes part of the ubiquitous language
- Helps with decoupling
- Aids in extensibility

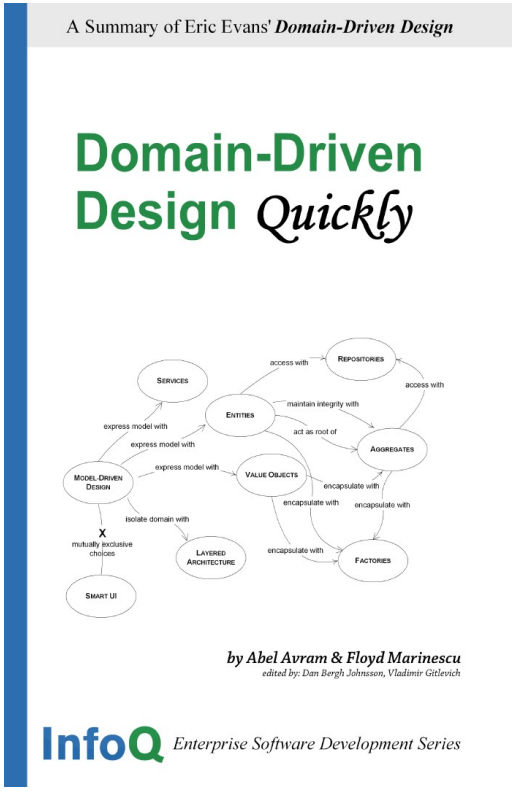
Context Mapping

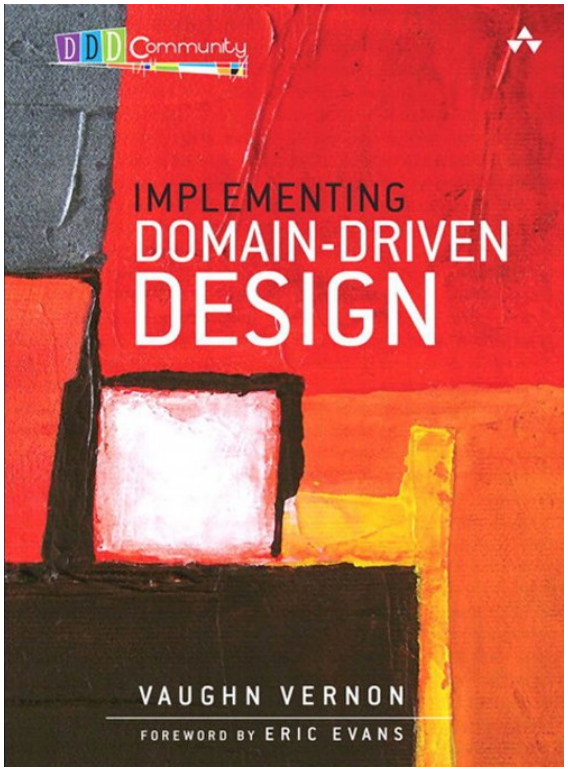
- Mapping the contact points and translations between bounded contexts.



Resources

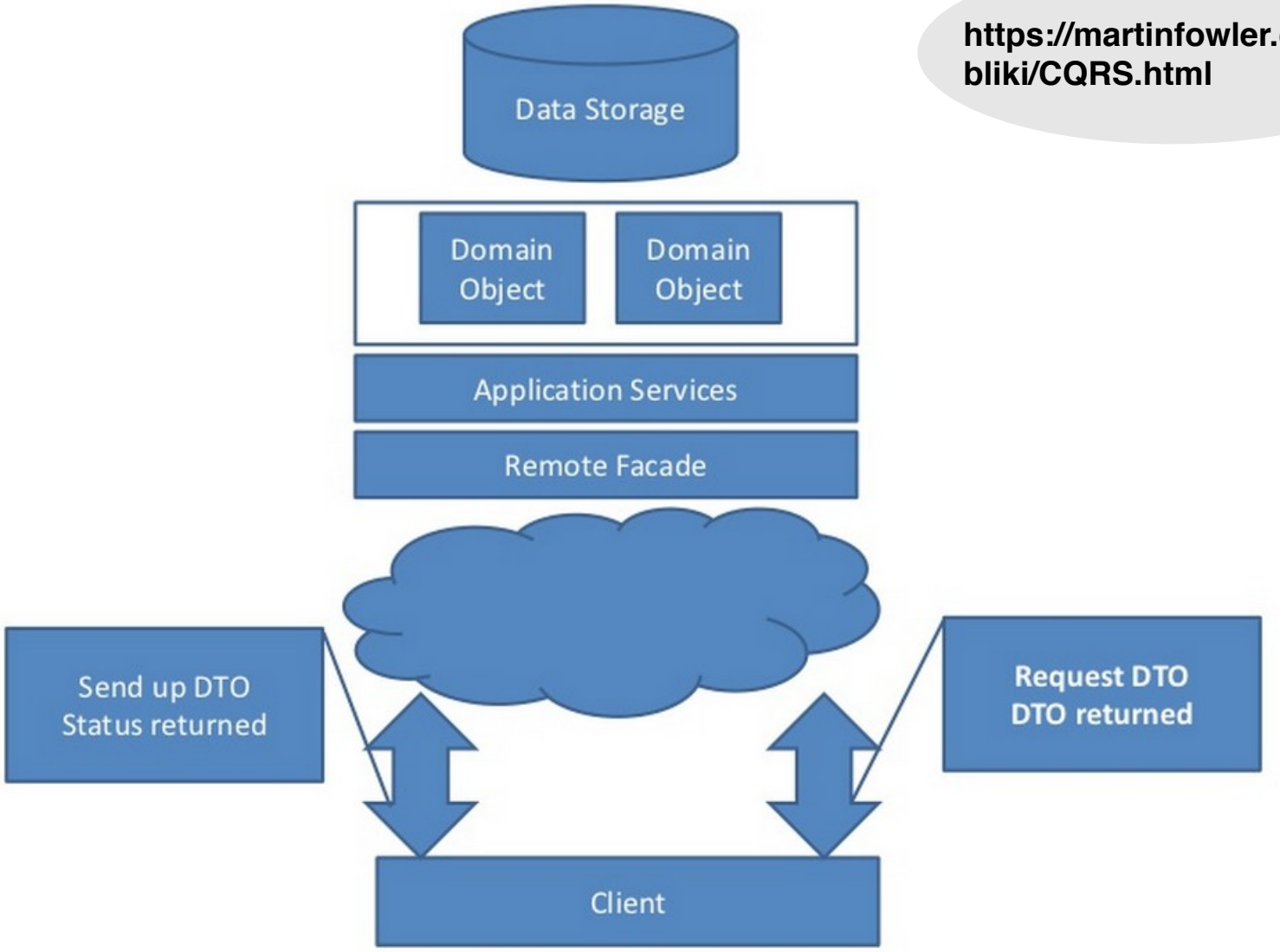

Domain-Driven
DESIGN
Tackling Complexity in the Heart of Software
Eric Evans
Foreword by Martin Fowler


A Summary of Eric Evans' Domain-Driven Design
Domain-Driven Design Quickly
Services
Repositories
Entities
Aggregates
Model-Driven Design
Value Objects
Layered Architecture
Factories
Smart UI
access with
maintain integrity with
act as root of
express model with
encapsulate with
isolate domain with
mutually exclusive choices
by Abel Avram & Floyd Marinescu
edited by: Dan Bergh Johnsson, Vladimir Gitlevich
InfoQ Enterprise Software Development Series

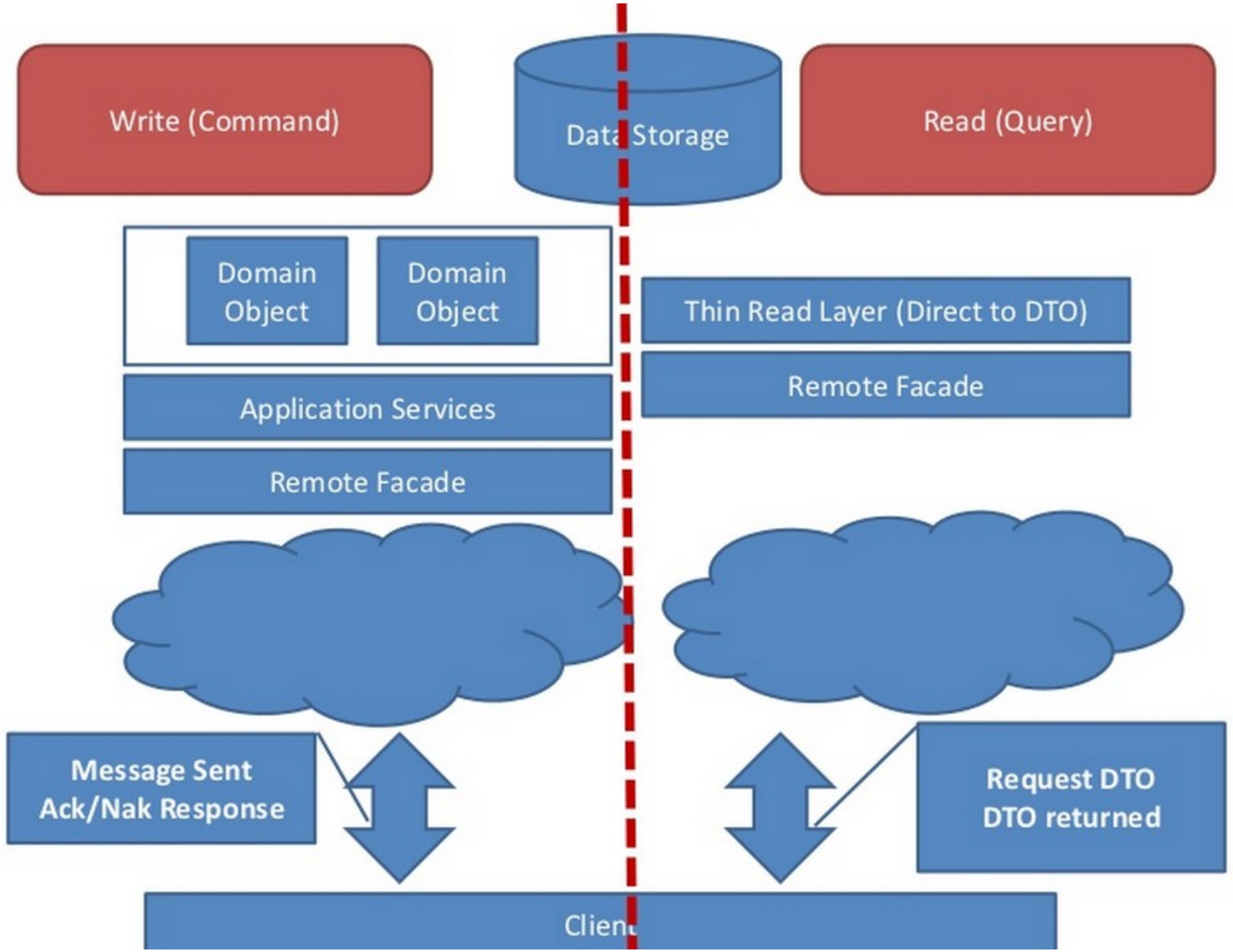

DDD Community
IMPLEMENTING
DOMAIN-DRIVEN
DESIGN
VAUGHN VERNON
FOREWORD BY ERIC EVANS

Command Query Responsibility Segregation (CQRS)

https://martinfowler.com/bliki/CQRS.html



The data that a client needs from the model is screen based and different than the domain model.



CQRS Implementation