

Practical Domain Driven Design

Domain Driven Design IS

An architectural methodology
for evolving a software system
that closely aligns
to business requirements

Domain Driven Design IS

- Domain *First*
- Focus on the Object Model
- Focus on Object-Oriented Design
- Works well with other *DD methods to:
 - Reduce Complexity
 - Increase Maintainability

Domain Driven Design IS NOT

- A silver bullet
 - A panacea for all your troubles
 - An easy path to follow
 - Always the best solution
-
- And most importantly, it is not focused on the How, but the What and Why

The Domain Vision Statement

- A shared understanding of what it is you are actually trying to create
- Should be brief, written in clear English and understood by business and tech people alike
- Should be factual, realistic, honest
- Should avoid superlatives and marketing speak
- Should avoid technical and implementation details

Domain

- A Domain is a Sphere of Knowledge, Influence or Activity
- A Domain is represented by the Ubiquitous Language
- A Domain encapsulates a Domain Model
- A Domain lives within a Bounded Context

The Ubiquitous Language

- A major reason for failure of software projects is a failure of people, the failure to communicate
- The Ubiquitous Language is a shared language between the business and the development teams

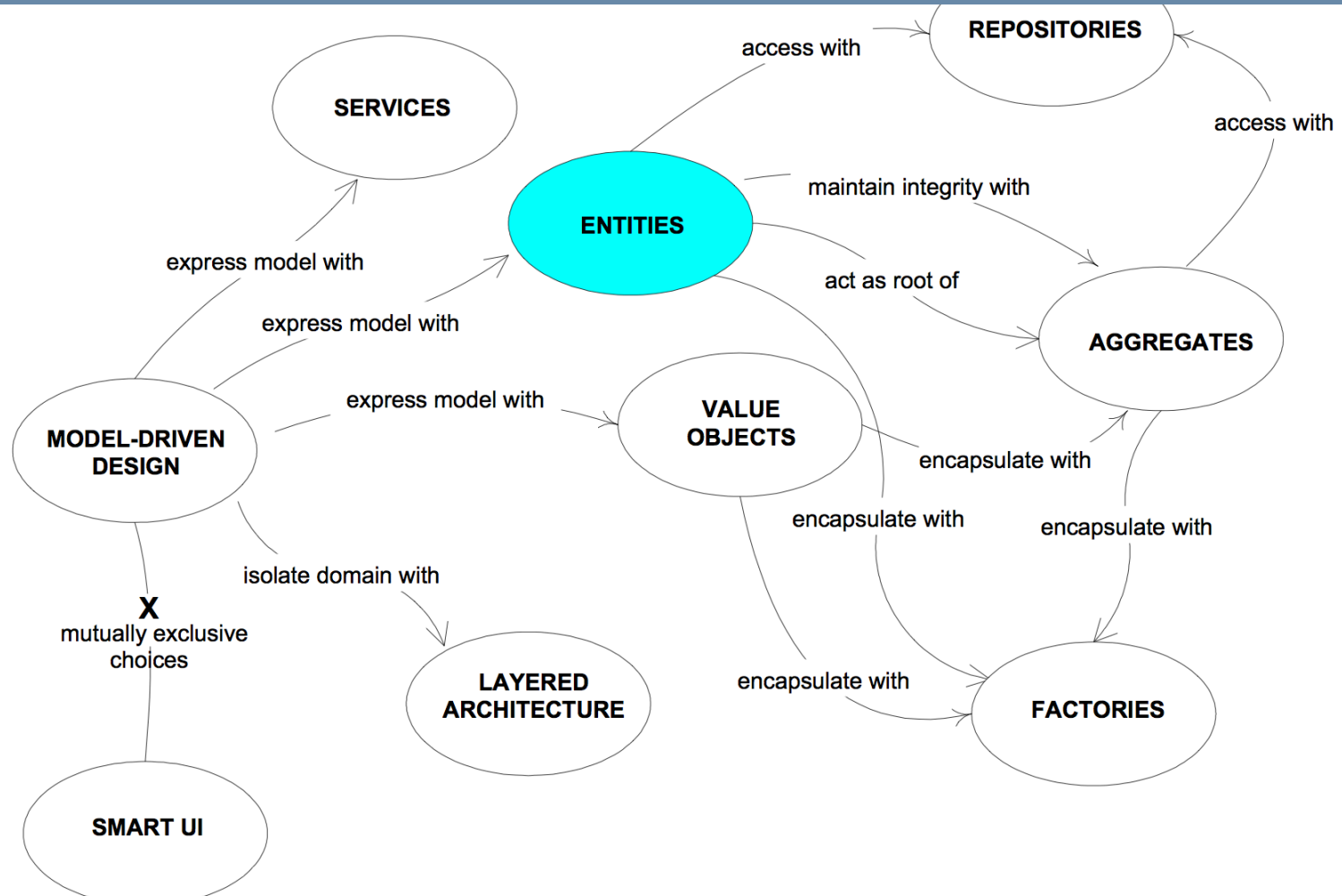
Domain Experts

- Domain Experts are the primary point of contact the development teams have with the business
- They are the Experts on their part of the business, not just users of the system
- They should have deep knowledge of the subject Domain

Domain Model

- A Domain Model is a representation of the relationships between the Entities and Value Objects in your Domain
- It may look similar to UML or a class relationship diagram, but it is not one
- The Domain Model should be recognisable and understandable by the business

Building Blocks



Entities

- Entities are the “things” within your Model
- An Entity is defined by being unique, and uniquely identifiable through time
- They have identity, state and behaviour
- E.g. Person, Account

Value Objects

- Value Objects are the “things” within your model that have no uniqueness
- They are equal in all ways to another Value Object if all their properties match
- Value Objects are interchangeable

Aggregates

- “An aggregate is a collection of items that are gathered together to form a total quantity” – Wikipedia
- Provides a boundary to manage invariants
- An Aggregate Root is the root item containing a number of parts that form a whole

Services

- Some concepts or operations from the domain model are not naturally modelled as objects
- These operations often involve many domain objects and co-ordinate their behaviour

Factories

- Abstract away the creation of aggregates so objects can be simpler and focused
- Factories manage the beginning of the life cycle of some aggregates

Persistence Ignorance

- “There Is No Database”
- DDD uses the Repository pattern to create Persistence Ignorance
- A Repository represents itself as an in-memory list of items
- Repositories are specific to Aggregate Roots, not to Entities

Repositories

- **Abstraction over data access**
- Repositories manage the middle and end of the life cycle
- Repositories might delegate persistence responsibilities to object-relational mappers for retrieval of objects
- 1 Aggregate : 1 Repository

Bounded Contexts

- When you have multiple models you should consider Bounded Contexts
- Each BC is a self contained “mini application” containing it’s own model, persistence and code base
- To map between BCs you use a Context Map

Factors For Success of DDD

- Your domain is not trivial
- You have access to Domain Experts
- You have an iterative process
- You have a skilled and motivated team

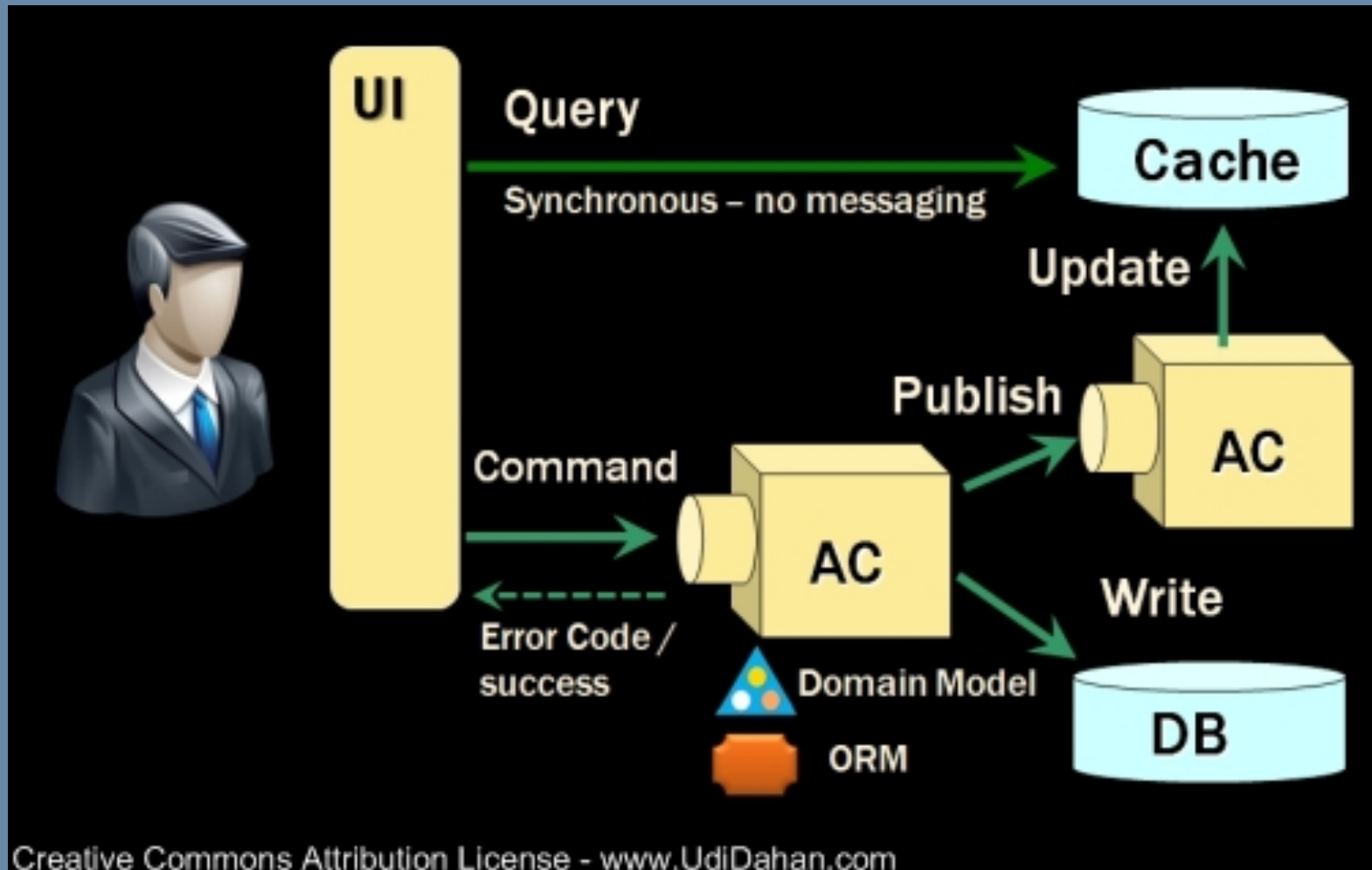
Messaging Architectures

- DDD is ideally suited to Message Based Architectures
- DDD is ideally suited to Commands and Events
- DDD is ideally suited to providing multiple autonomous systems that are loosely coupled

Command Query Responsibility Separation (CQRS)

- Bertrand Meyer principle of CQS:
every method should either be a command that performs an action, or a query that returns data to the caller
- At an architectural level this means:
either issue commands, or issue queries, but never both
- And, query from a separate source from your domain commands

CQRS in a Picture





QUESTIONS