

# *Software Architecture and Techniques*

## Architectural Trends I

# Truths (1/2)

- Agile architecture is **emergent**
- Agile architecture **evolves**
- Architecture is **technology related**
- Not all architecture aspects are technology related

# Truths (2/2)

- SOA is dead – *infrastructure, applications, and application business services coordinated through orchestration, use instead bounded domains* -
- Monolith Solutions must be handled with care
  - Through discipline you can build a **modular monolith solution**
- Applications are now mobile first
- Applications are often browser first – *in general for strange reasons* -
  - Progressive Web Application *PWA* approaches are emerging
- Browser solutions must often be rewritten – *every 18 or 24 months* -

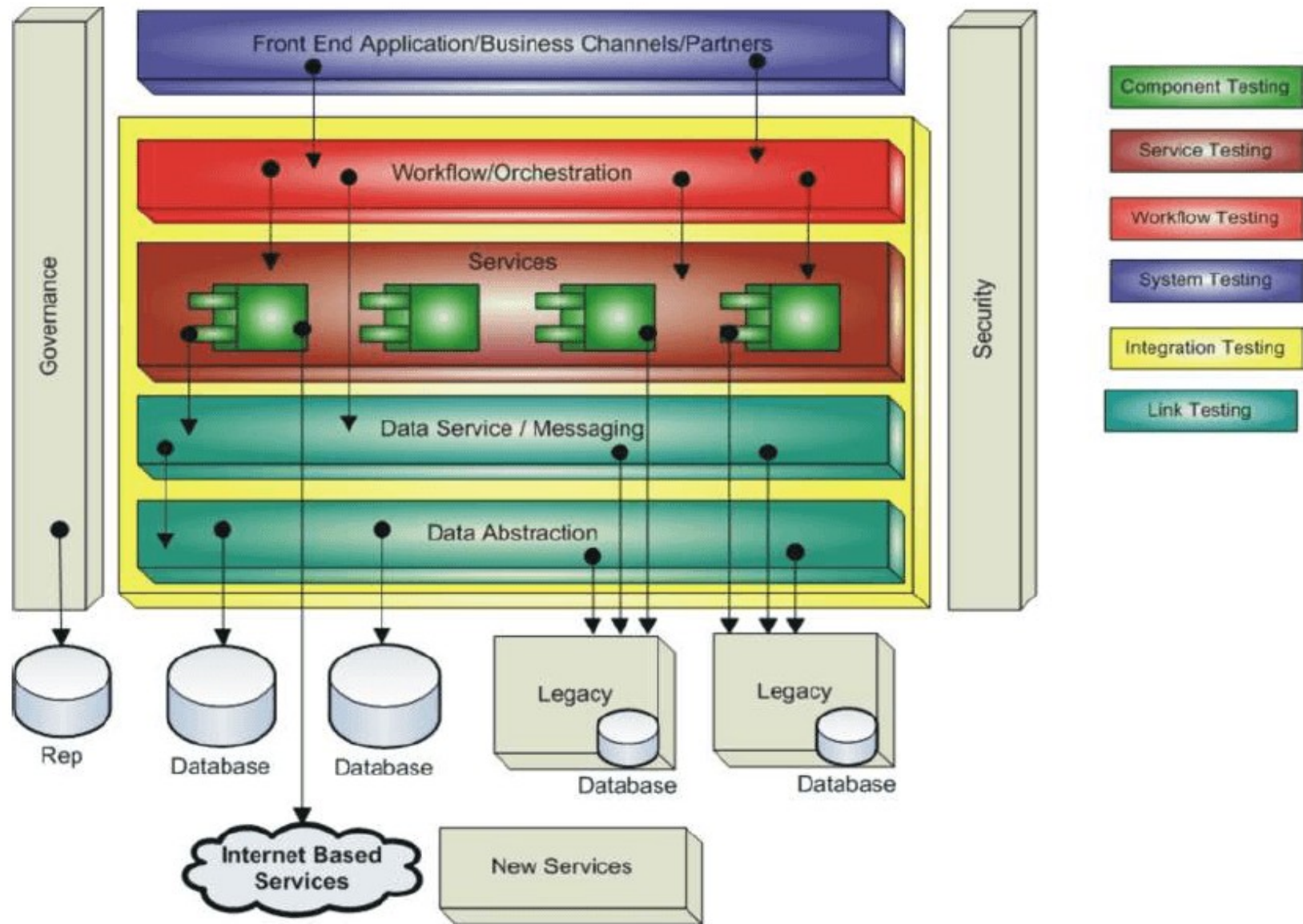
# Layered Architecture

- Unorganized source code and modules with no definite roles can become a problem for the application.
- Skipping previous layers to create tight coupling can lead to a logical mess full of complex interdependencies.
- Basic modifications can require a complete redeployment of the application.

# Event Driven Architecture

- Testing individual modules can only be done if they are independent, otherwise, they need to be tested in a fully functional system.
- When several modules are handling the same events, error handling becomes challenging to structure.
- Development of a system-wide data structure for events can become arduous if the events have different needs.
- Maintaining a transaction-based mechanism for consistency can become complex with decoupled and independent modules.

SOA  
Architecture



2000's

## SERVICE ORIENTED ARCHITECTURE

---

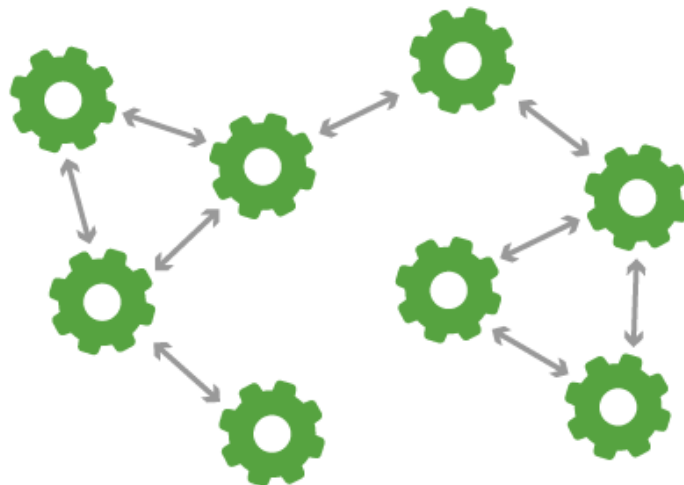


**SOA** based applications are comprised of more loosely coupled components that use an Enterprise Services Bus messaging protocol to communicate between themselves.

2010's

## MICROSERVICES ARCHITECTURE

---



**Microservices** are a number of independent application services delivering one single functionality in a loosely connected and self-contained fashion, communicating through light-weight messaging protocols such as HTTP, REST or Thrift API.

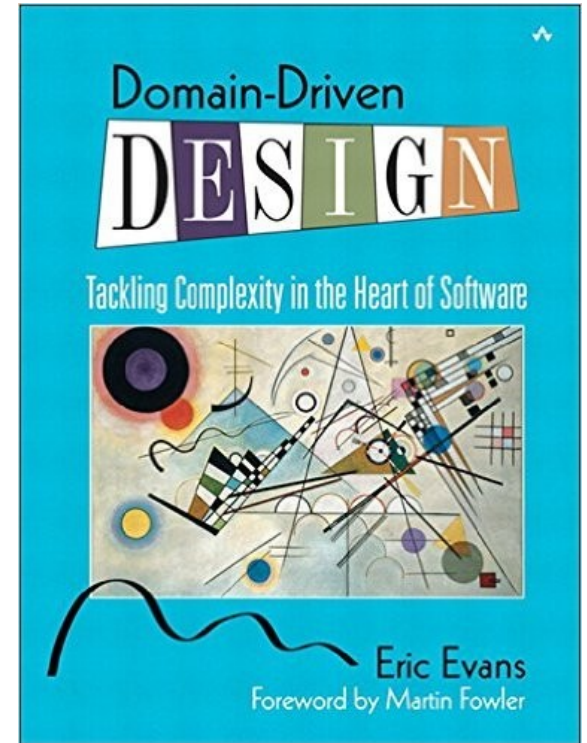
# Micro Kernel Architecture

- The plugins must have good handshaking code so that the microkernel is aware of the plugin installation and is ready to work.
- Changing a microkernel is almost impossible if there are multiple plugins dependent on it.
- It is difficult to choose the right granularity for the kernel function in advance and more complex at a later stage.

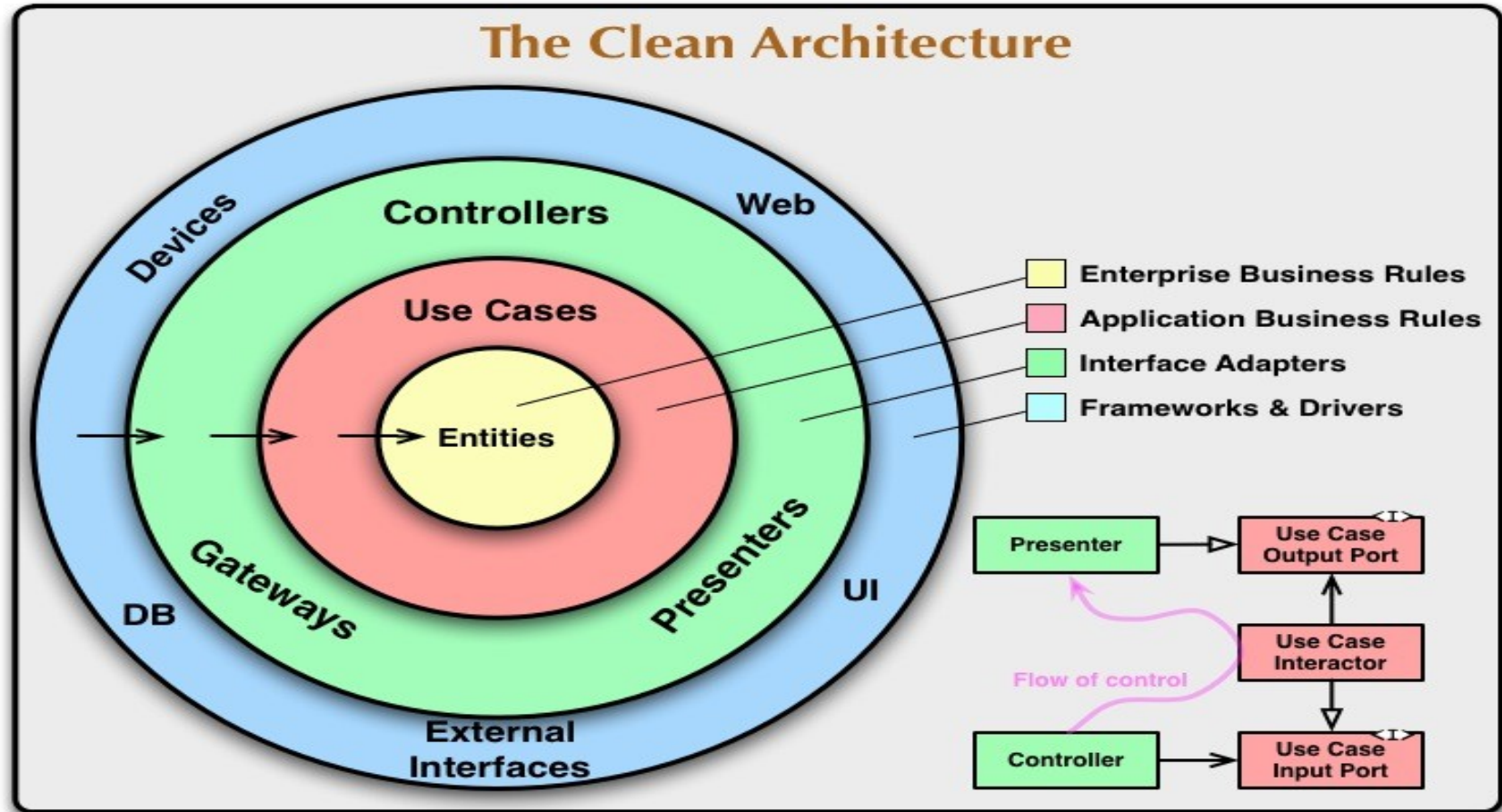


# Domain Driven Design

- Book published in 2003
- Focus on application domain instead of technical world
- Understand user and his language
- Code reflects user model
- Death of big UML models

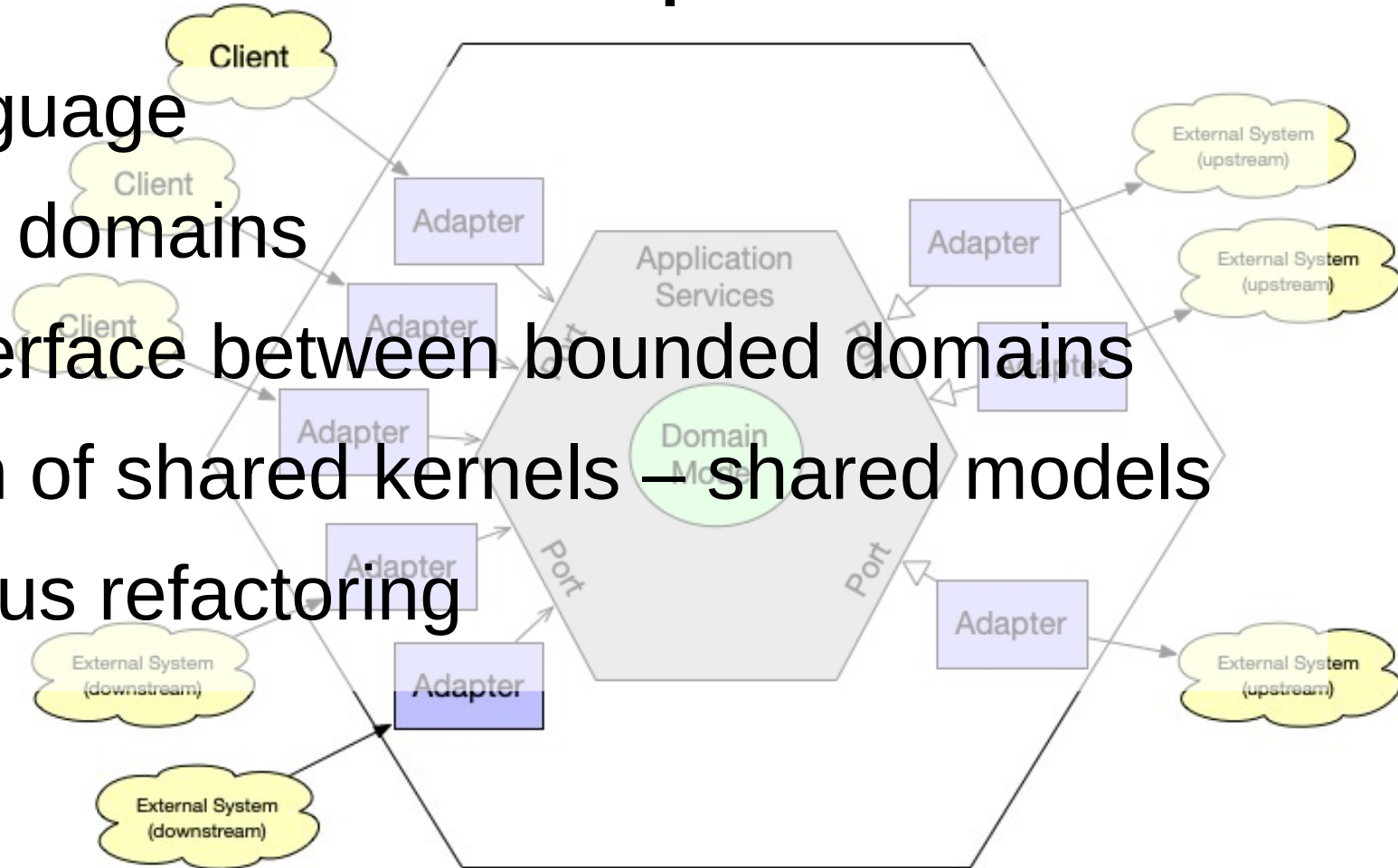


# Hexagonal or Onion Architecture



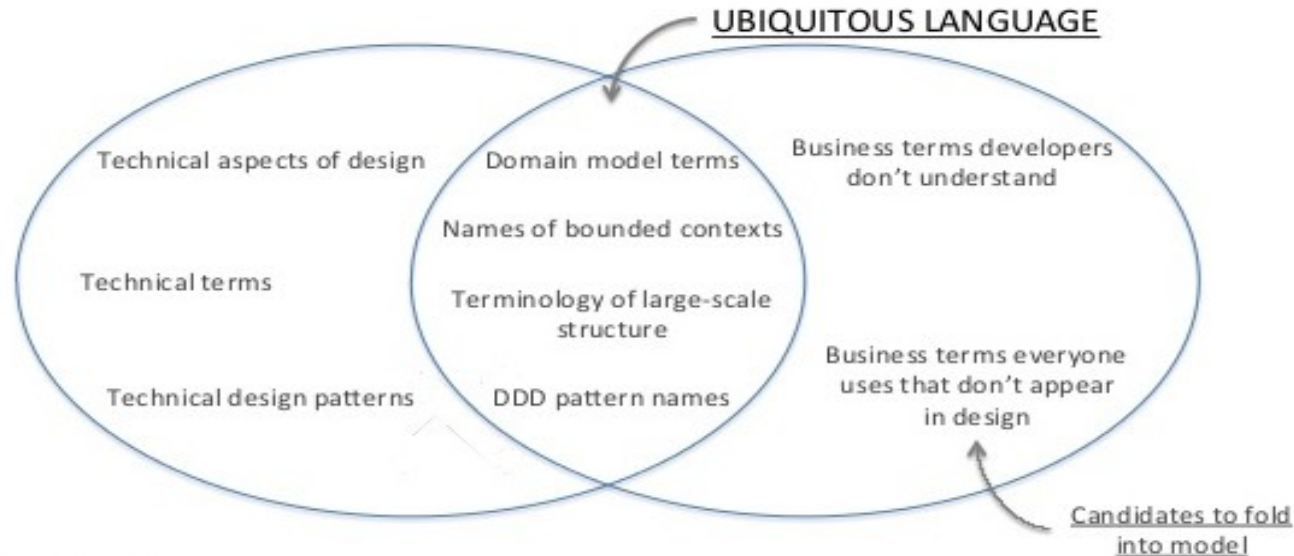
# Domain Driven Development

- User language
- Bounded domains
- Clear interface between bounded domains
- Definition of shared kernels – shared models
- Continuous refactoring

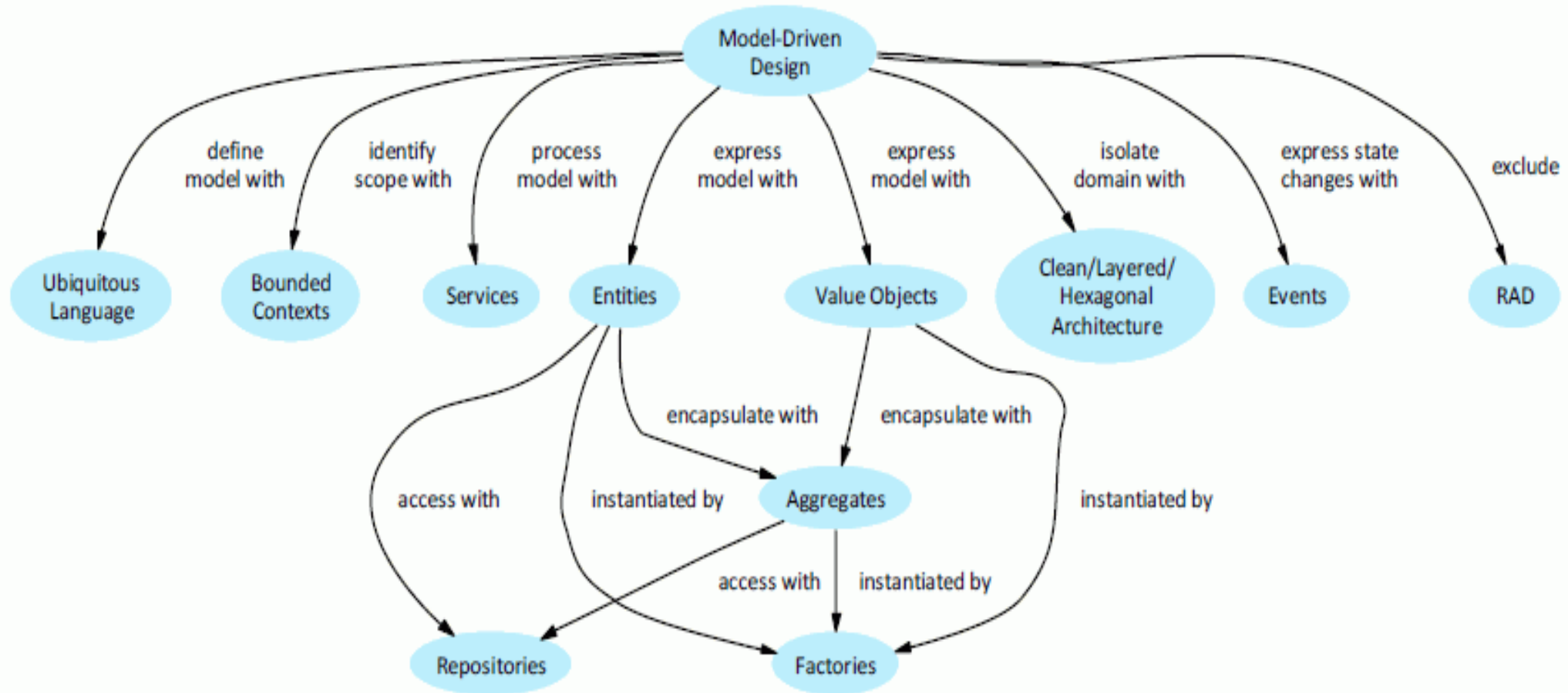


# Domain Ubiquitous Language

- Value is generated by and through the customer
- Use his domain language through your design and in the source code



# Domain Driven Design

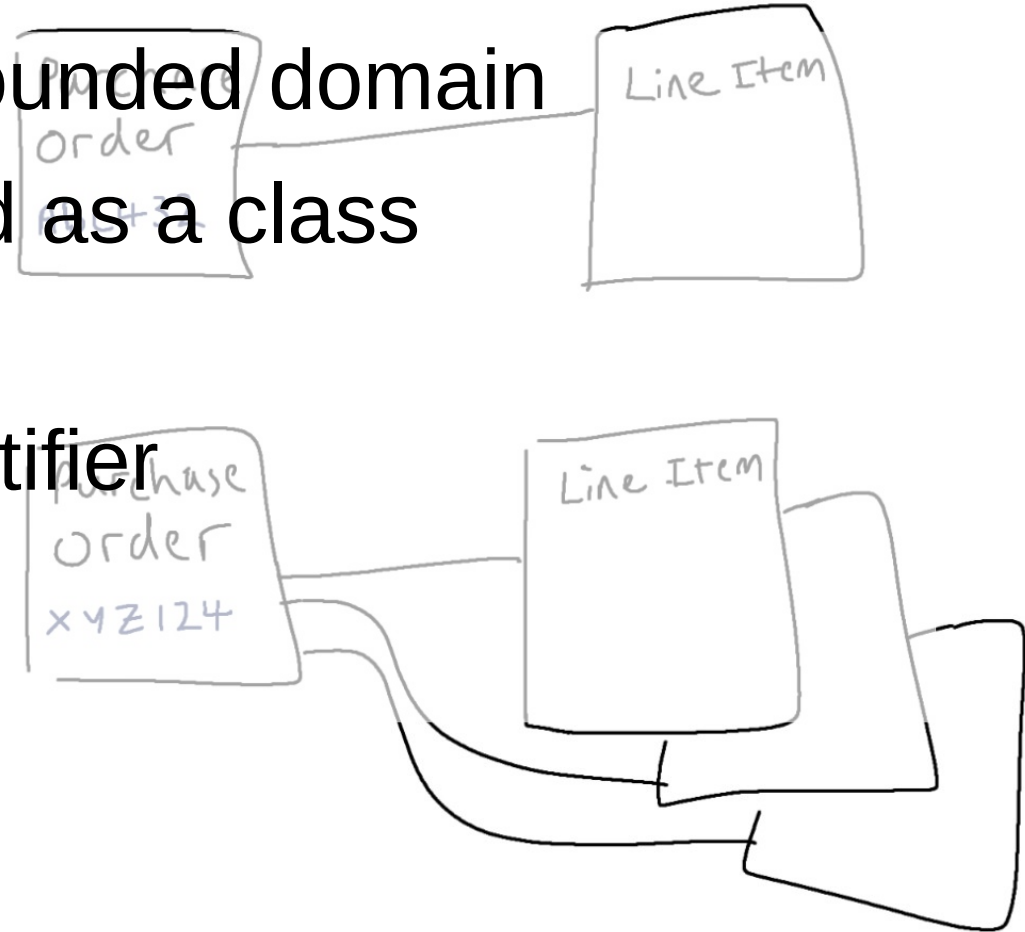


# Domain Driven Design

- **Entities** – objects having an identity
  - e.g. Person, Company
- **Value Objects** – have information, no identity
  - e.g. Address, Phone Number
- **Aggregates** – compose entities and value objects
- **Services** – business operations not belonging to objects
- **Repositories** – persist entities and value objects
- **Factories** – creates entities and value objects

# Entities

- A key concept of the bounded domain
- An abstraction modeled as a class
  - Identity, state, behavior
- Has unique visible identifier



# Value Objects

- Does not have an identity
- Easily created and discarded
- Value objects should be immutable
- Value objects are shareable
- Value objects should often be implemented as *record, sealed types, or enumeration*

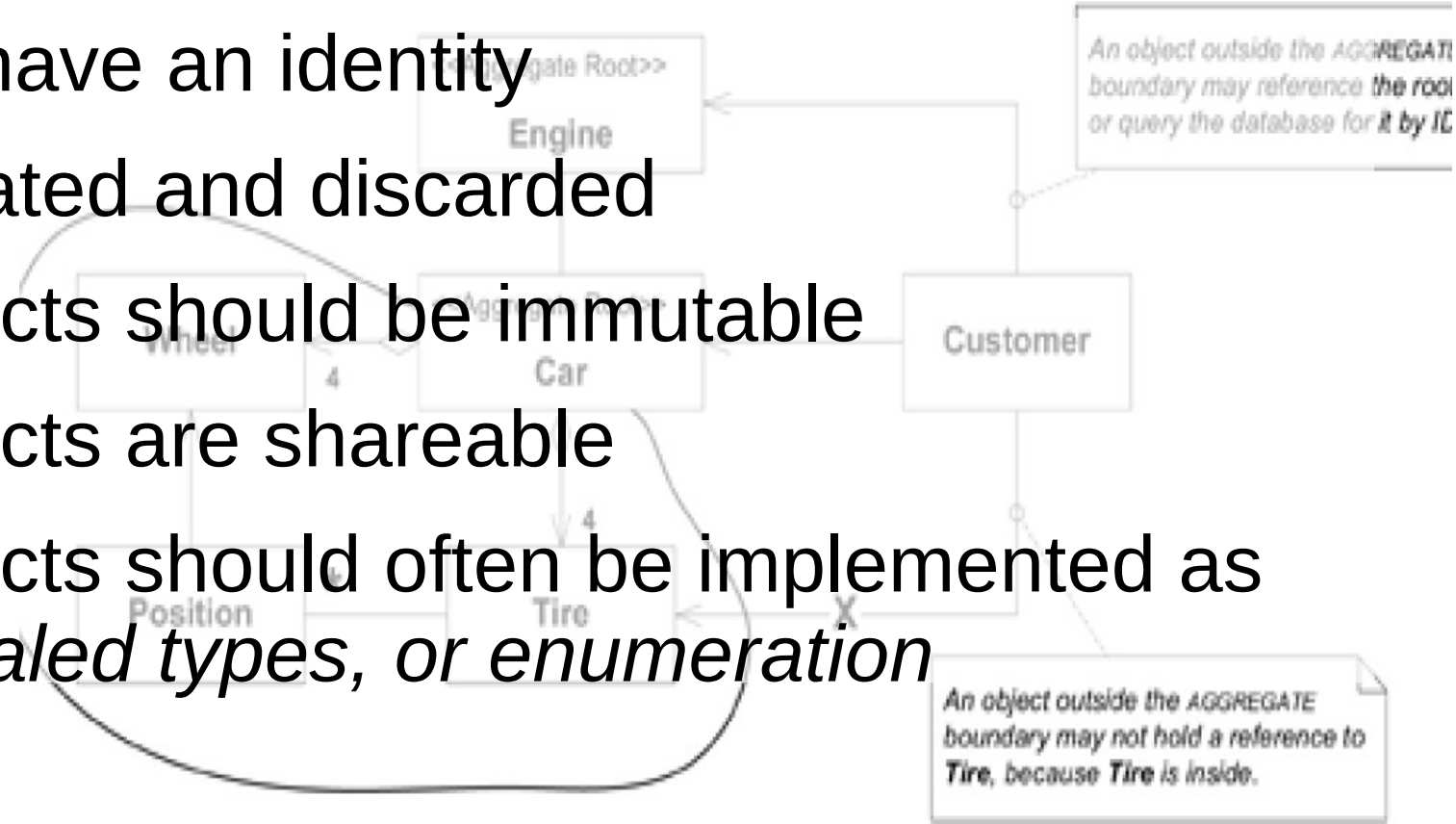
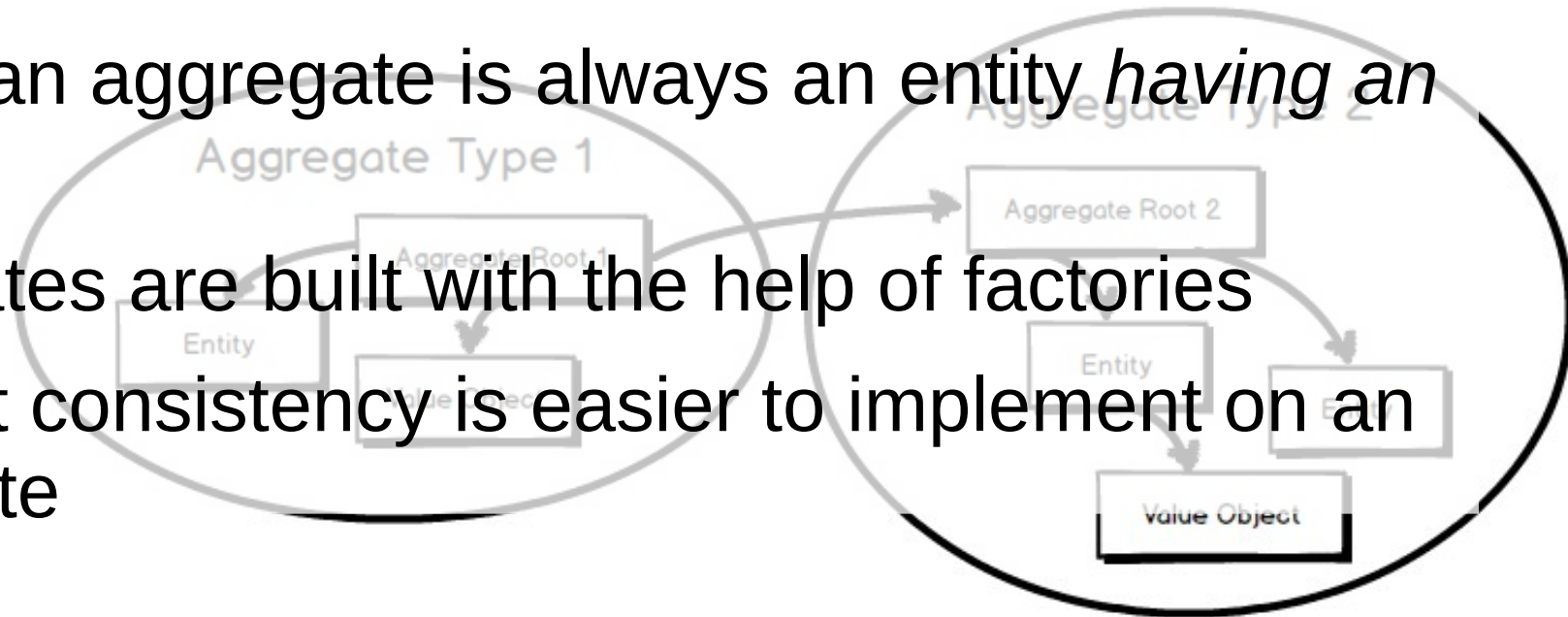


Figure 6.2. Local versus global identity and object references



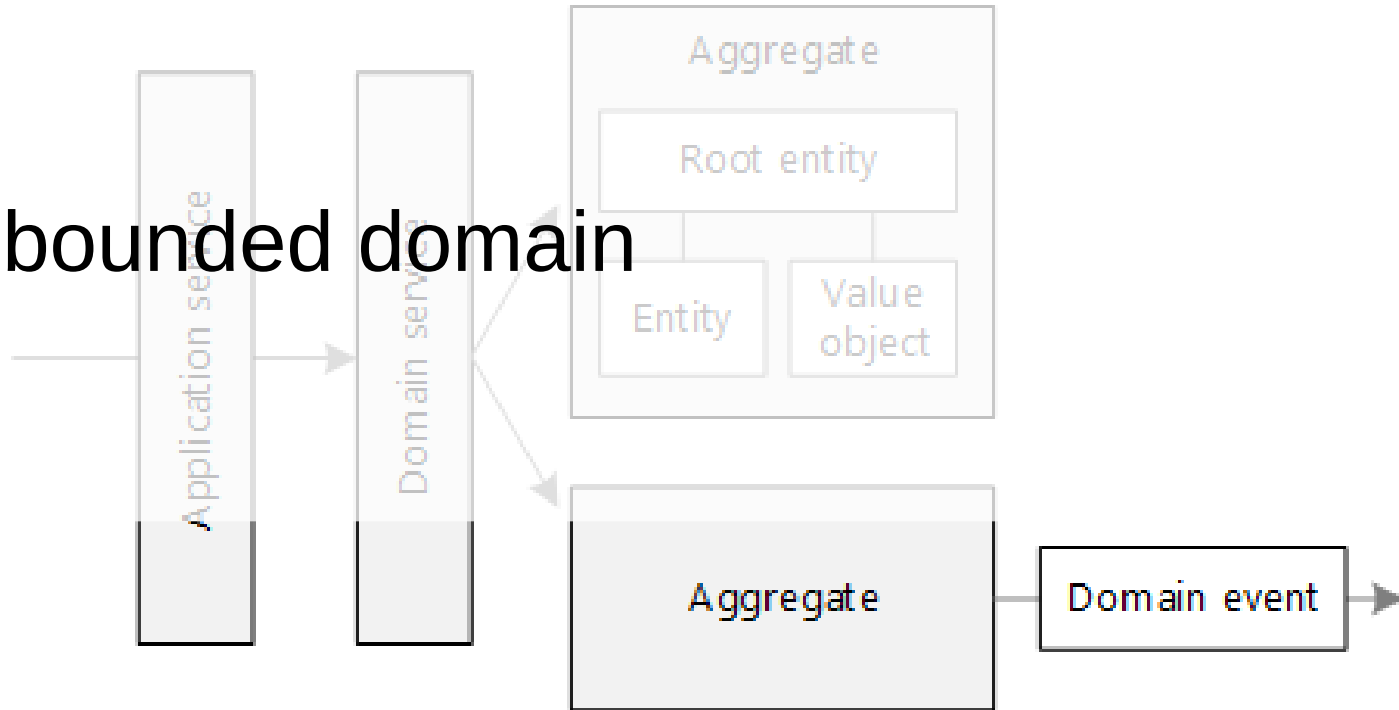
# Aggregates

- An Aggregate is a group of associated objects which are considered as one unit with regard to data changes
- Root of an aggregate is always an entity *having an identity*
- Aggregates are built with the help of factories
- Invariant consistency is easier to implement on an aggregate



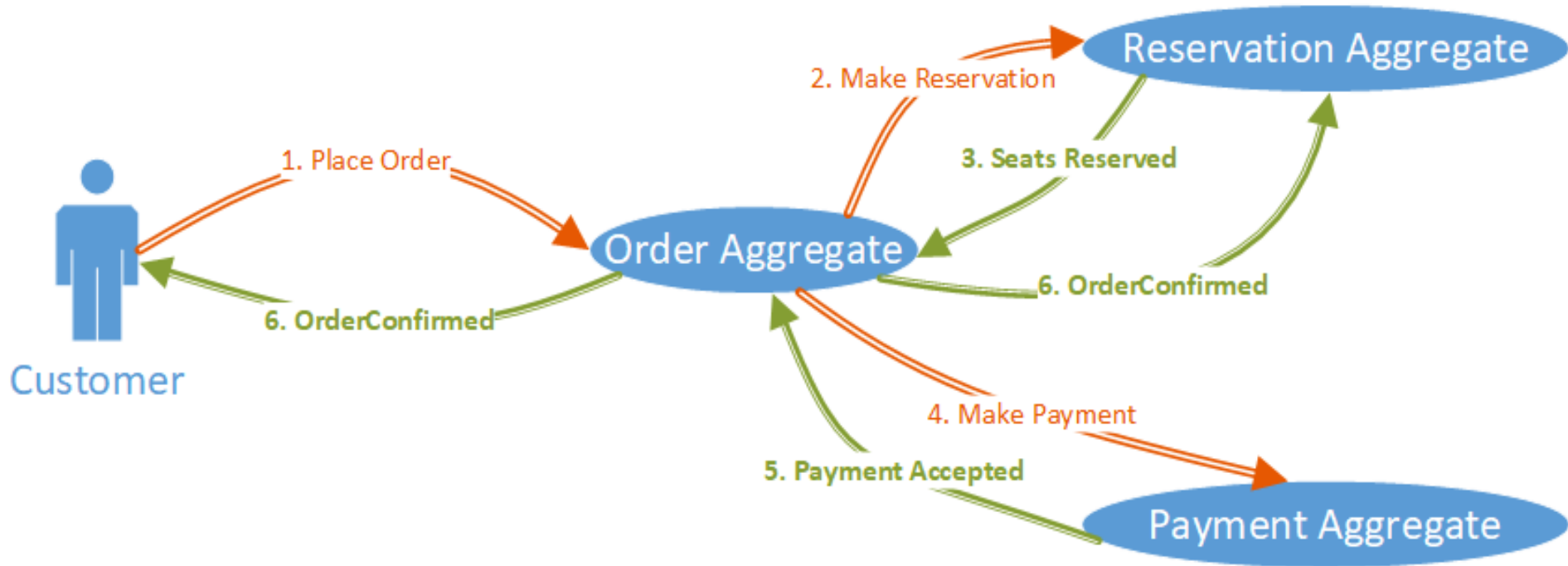
# Services

- Logic does not belong to an entity or an aggregate
- Are stateless
- Are part of a bounded domain



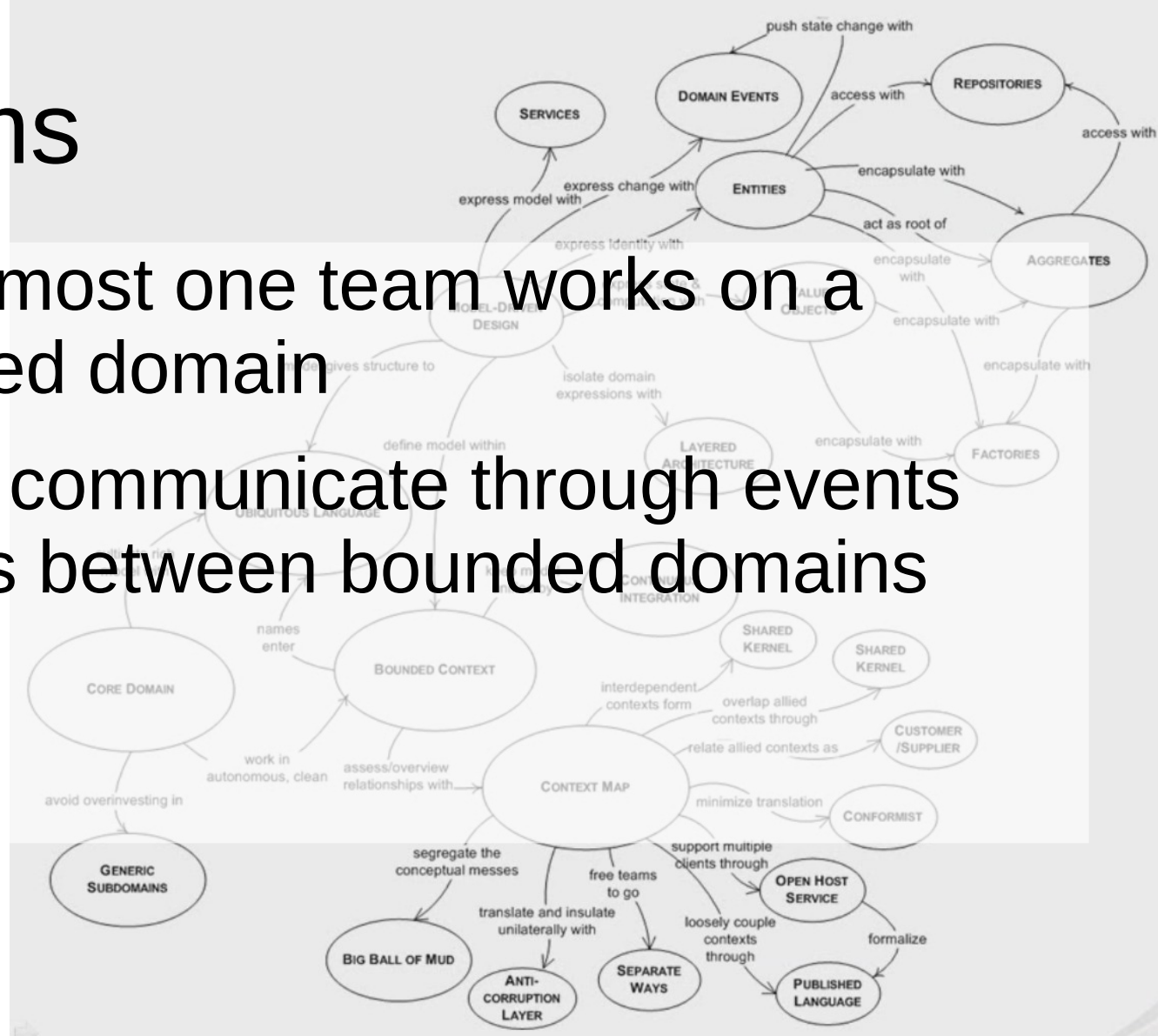
# Domain Events

- Event – Something changed in the system
  - Order Placed



# Multiple teams

- Good Rule: At most one team works on a specific bounded domain
- Multiple teams communicate through events and commands between bounded domains



# Immutability and Functional Style

- Entity objects shall have an immutable identity
- Value objects shall be immutable
- Immutable means that all values are set at creation
- You have quite a few immutable classes in the Java standard library

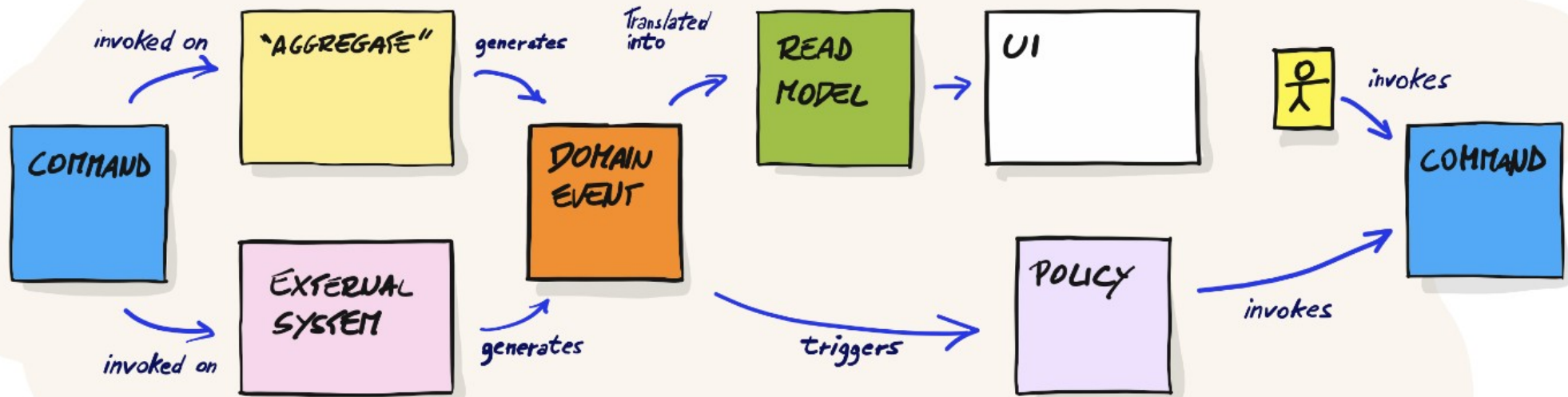
# Domain Driven Design Questions

- What is an identity?
- Why should value objects be immutable?
- Why is a bounded context important?
- Why is continuous integration a central activity?
- How can you model bounded domains in Java?
- Reflect about Domain Specific Languages DSL

# DDD Anti-Patterns

- Anemic domain objects
- Repetitive DAO's
- Fat Service Layer where service classes will end up having all the business logic.
- Feature Envy: This is one of the classic smells mentioned in Martin Fowler's book on Refactoring where the methods in a class are far too interested in data belonging to other classes.

# Event Storming – Ignite Your DDD





# Micro Service Architecture

- Designing the right level of granularity for a service component is always a challenge.
- Not all applications do include tasks that can be split into independent units.
- Performance can be affected because of tasks being spread across different microservices.

# Micro Services (1/4)

- Mapping bounded domains and micro services
- Exchange of information over JSON
  - Only if needed use a binary format – see [protocol buffers](#) -

# Micro Services (2/4)

- REST services → [openAPI](#), Swagger
- GraphQL services
- Asynchronous services
- Reactive system
  - Event based
  - [Eventual consistency](#)
  - [CAP Theorem](#)

# Micro Services (3/4)

Beware of the fallacies of distributed computing

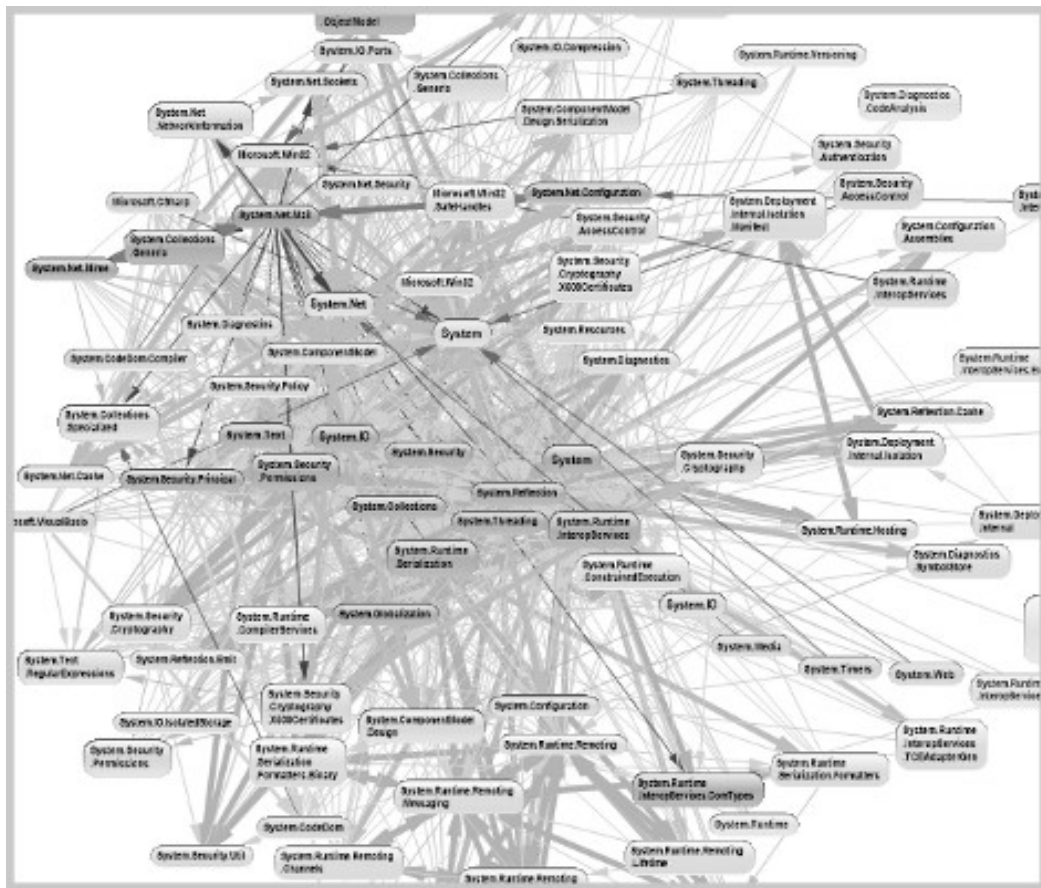
- The network is reliable
- Latency is zero
- Bandwidth is infinite
- The network is secure
- Topology does not change
- There is one administrator
- Transport cost is zero
- The network is homogeneous

# Micro Services (4/4)

- User Interfaces are more difficult
  - Micro user interface component approach
  - GraphQL is one approach to solve some aspects
- Logging is more difficult
  - ELK is a possible answer
- Running the system is more difficult
  - Docker, Kubernetes, Helm are approaches

# Big Ball of Mud

- You will encounter often this big Anti-Pattern in Europe
- You will not get the money to rewrite the product
- You must destroy this horror



# Monolith to Modular

- Extract one big service and all associated classes
  - As a separate application
  - As a separate Java module
- Try to define your bounded domain
  - e.g. Spring has support for [modulith](#)
- Persisted data is migrated in their own schema
- Deploy as a separate micro profile service (if using Java)

# Refactor Aggressively

- Refactor your extracted big service to have clean code
- Repeat
  - Extract another big service from the ball of mud
  - Split your big service into smaller services



# Evolvable Architecture

- 1) Identify Dimensions Affected by Evolution
- 2) Define Fitness Functions for each Dimension
- 3) Use Deployment Pipeline to Automate Fitness Functions
- 4) Start evolving

# Wisdom (1/3)

- Understand the **business problem** before choosing your architecture
- The more reusable code is, the less usable it is
- **Prefer** duplication to coupling
- **Avoid COTS** solutions if you want to be agile
- **Avoid frameworks**, use libraries
- Remove needless variability

# Wisdom (2/3)

- For smaller applications start with a **monolithic modular** approach

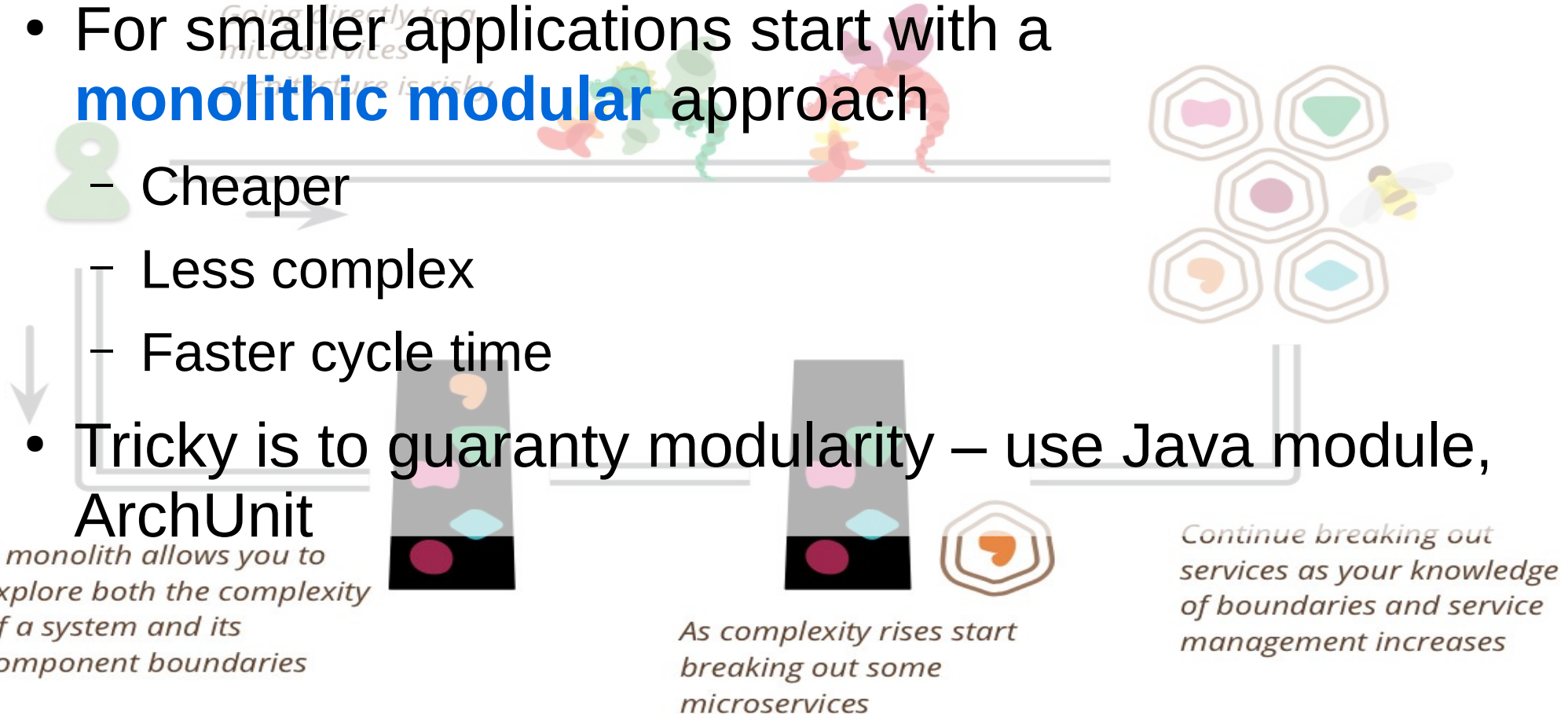
- Cheaper
- Less complex
- Faster cycle time

- Tricky is to guaranty modularity – use Java module, ArchUnit

*A monolith allows you to explore both the complexity of a system and its component boundaries*

*As complexity rises start breaking out some microservices*

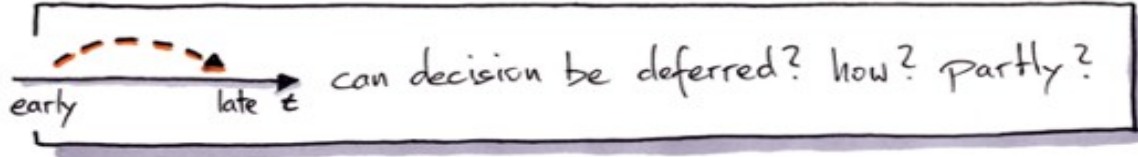
*Continue breaking out services as your knowledge of boundaries and service management increases*



# Wisdom (3/3)

- You must have a modular persistent approach
  - Multiple schemas in the database
  - Multiple databases
- You must have an agile migration approach to persistent data
  - [FlyWay](#), Liquibase

# Reflection



- persist data of your system to survive restart
- how to translate UI and data
- communication between parts of your system
- scaling (run on multiple threads, processes, machines)
- security (how to authenticate, authorize)
- journaling (Activities, data)
- reporting
- data migration / data import
- releasability
- backwards compatibility
- response times
- Archiving data

- Go back to the list of architectural themes
- For your technology have solutions to each theme

# Exercises

- Read article “Domain Driven Design Quickly”
- Make some of your Java classes immutable
  - Often implies Builder pattern, evaluate [Lombok](#) library,
  - Often implies a functional programming style
- Return only immutable collections
  - And never return null values → use **Optional<T>**
- Replace your loops and conditions with streams and filters
- Use Java modules to declare boundaries and dependencies