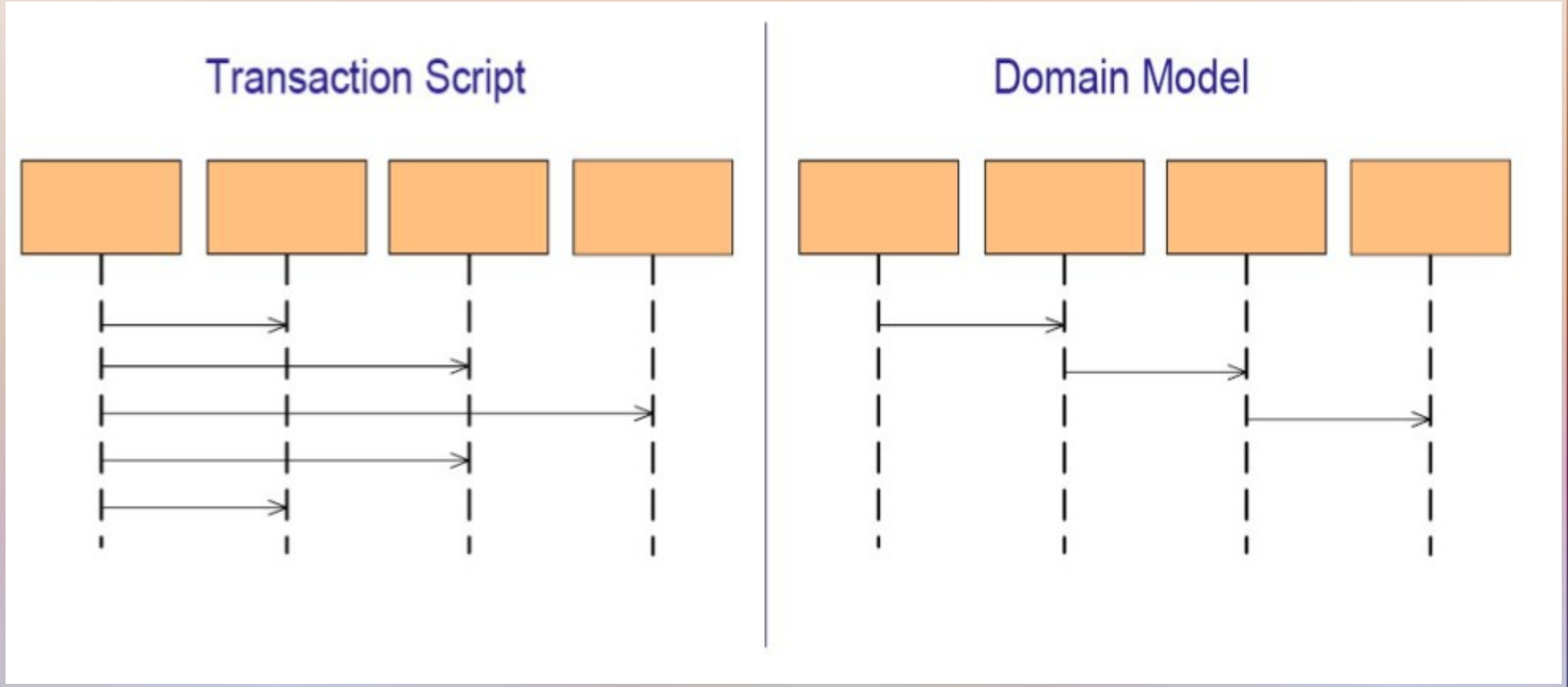
Domain-Driven Design

- Politics Wang - DerbySoft

What Is Domain-Driven Design

- An approach to developing software for complex needs by deeply connecting the implementation to an evolving model of the core business concepts
- Domain is business of an industry
- Talking with the domain experts to learn about a domain
- Synthesize domain into model, communicate the model

Transaction Script
vs
Domain Driven



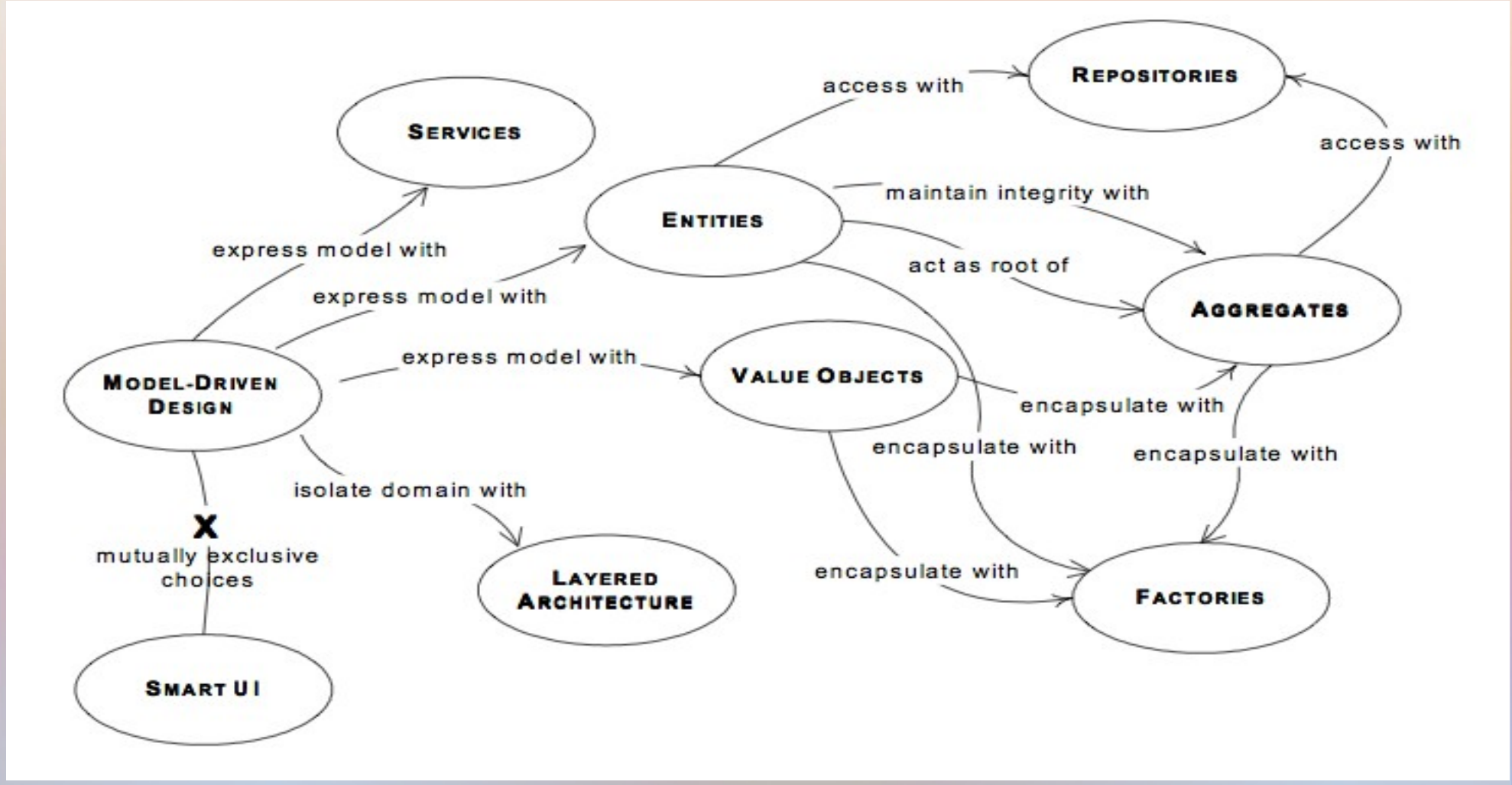
Ubiquitous language

- Developers and domain experts always thinking in different ways
- Use a language based on the model
- Make sure this language appears consistently in all the communication forms used by the team
- It takes hard work and a lot of focus to make sure that the key elements of the language are brought to light
- The model and the language are strongly interconnected with one another

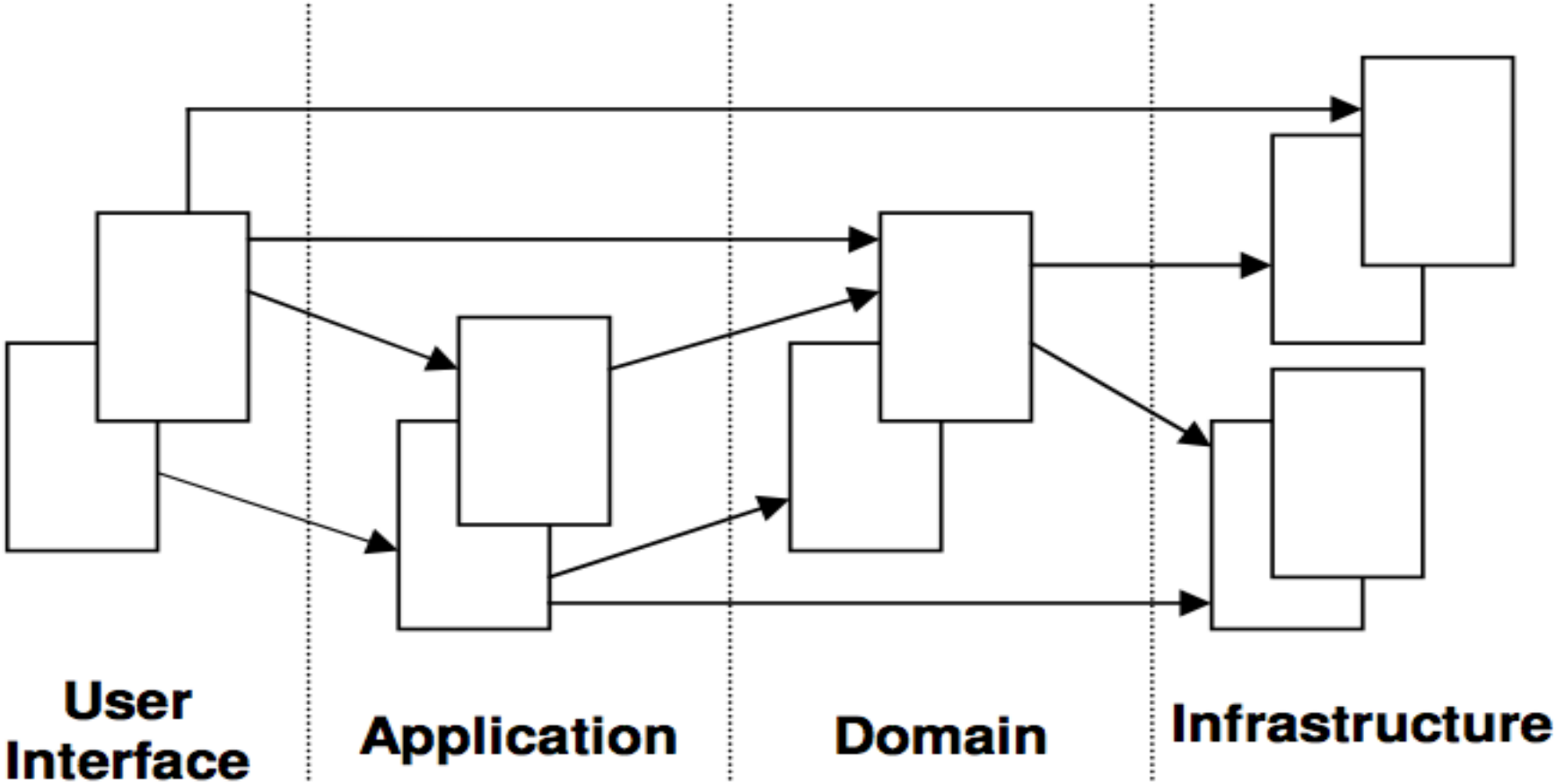
Model-Driven Design

- Ubiquitous language → Modeling → Implement the model in code
- Analysis model - cannot foresee some of the defects in their model, and all the intricacies of the domain
- Closely relate domain modeling and design
- Revisit the model and modify it to be implemented more naturally in software
- Model implementation : Object-oriented or Procedural language ?

A navigation map of the language of MODEL-DRIVEN DESIGN



Isolating the Domain - Layered Architecture



Isolating the Domain - Layered Architecture

- User Interface (Presentation Layer)

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

- Application Layer

  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, but it can hold the state of an application task progress.

Isolating the Domain - Layered Architecture

- Domain Layer

  This layer contains information about the domain. This is the heart of the business software. The state of business objects is held here. Persistence of the

  business objects and possibly their state is delegated to the infrastructure layer.

- Infrastructure Layer

  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.

A Model Expressed in Software - Entities

- Uniquely identifiable (has identity)
- Equality comparison uses identity (not attribute values)
- Usually extended longevity
- Examples

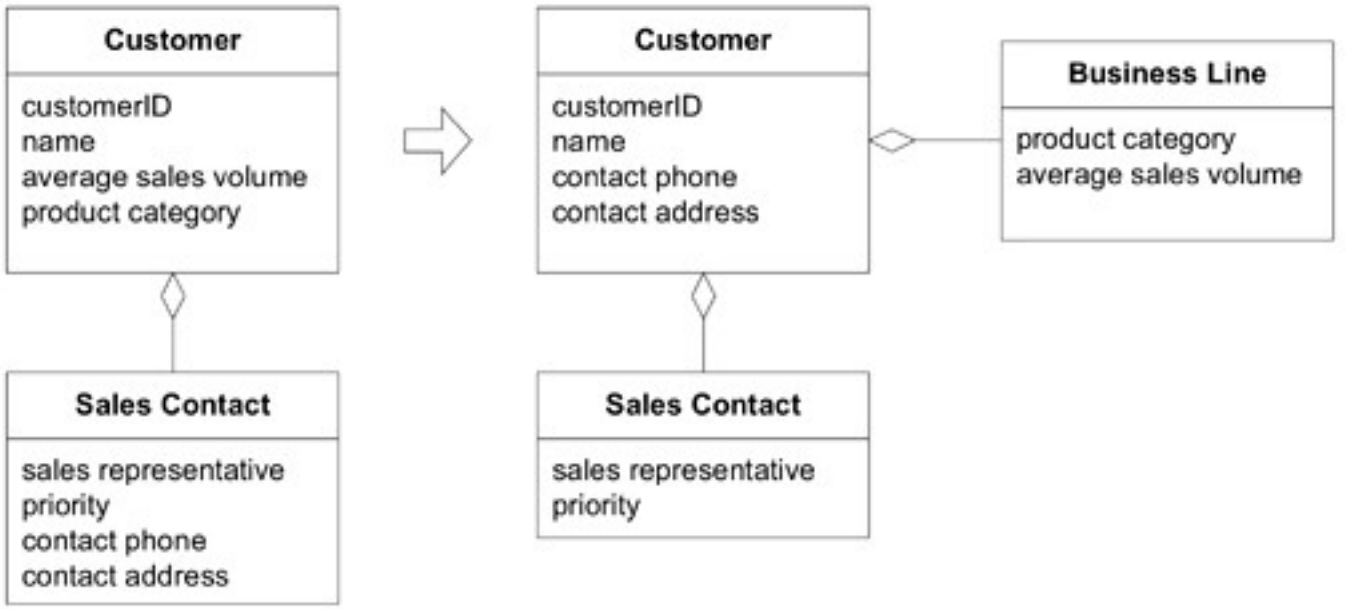
  Product

  Customer

  Order

A Model Expressed in Software - Entities

Figure 5.5. Attributes associated with identity stay with the ENTITY.



A Model Expressed in Software - Value Objects

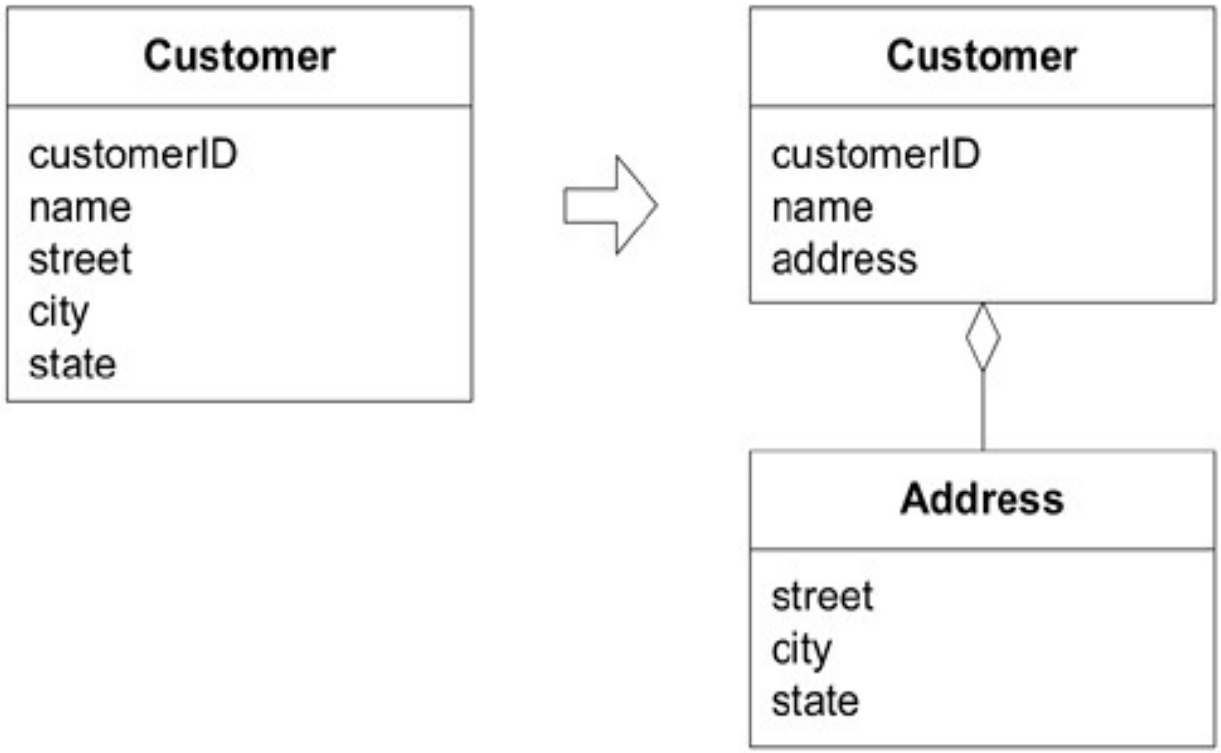
- No identity
- Equality comparison uses attribute values
- Describes certain aspects of a domain
- Should be immutable
- Example

  Address

  OrderLineItem

A Model Expressed in Software -Value Objects

Figure 5.6. A VALUE OBJECT can give information about an ENTITY. It should be conceptually whole.



A Model Expressed in Software-Services

- Some concepts from the domain aren't natural to model as objects, declare it as a Service
- An interface that stands alone in the model
- There are three characteristics of a Service:

  1. The operation performed by the Service refers to a domain concept which does not naturally belong to an Entity or Value Object.

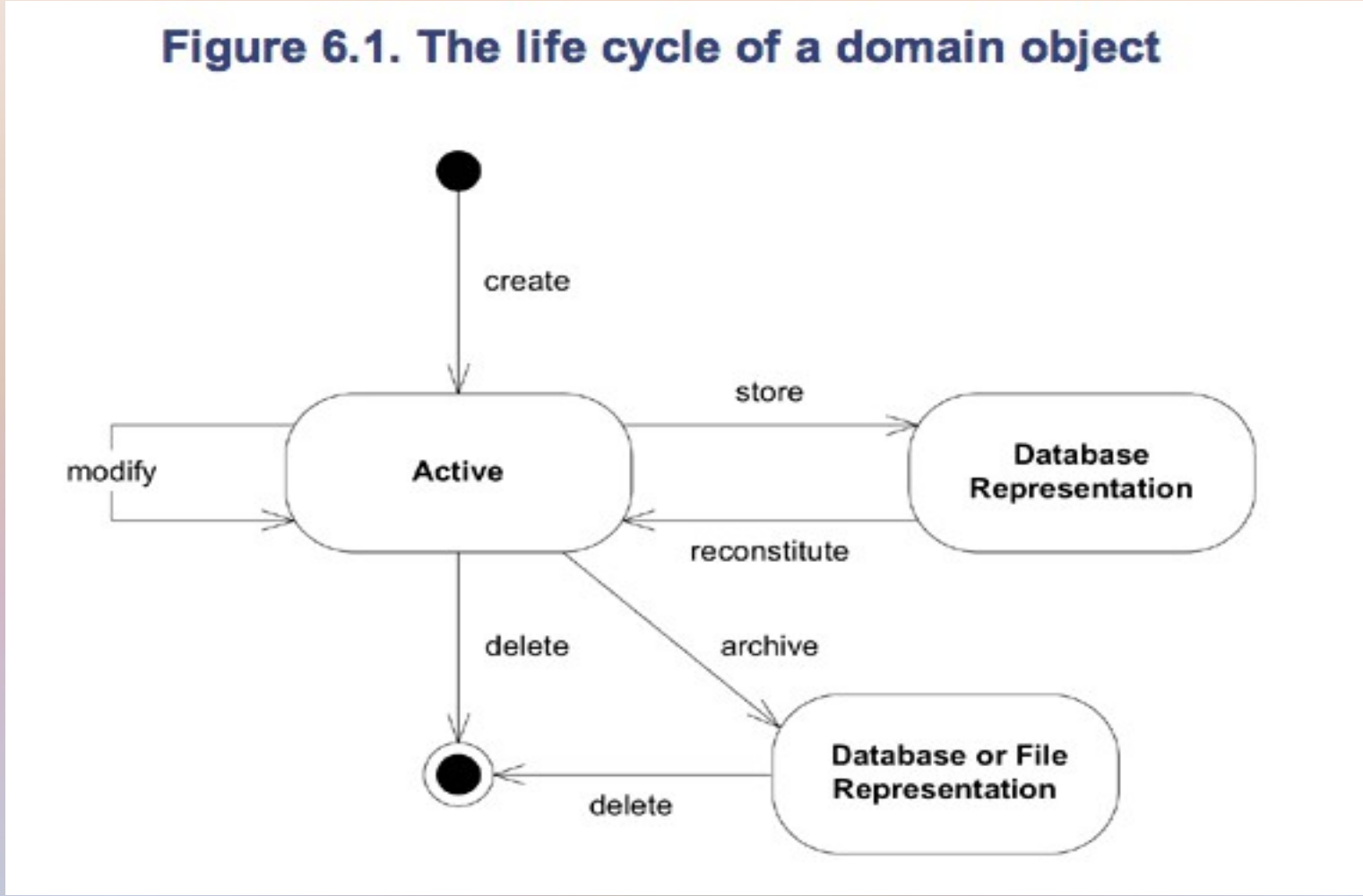
  2. The operation performed refers to other objects in the domain.

  3. The operation is stateless.

A Model Expressed in Software - Modules

- A method of organizing related concepts and tasks in order to reduce complexity
- Using modules in design is a way to increase cohesion and decrease coupling
- Give the Modules names that become part of the Ubiquitous Language. Modules and their names should reflect insight into the domain.

The Life Cycle of a Domain Object



The Life Cycle of a Domain Object - **challenges**

- Maintaining integrity throughout the life cycle
- Preventing the model from getting swamped by the complexity of managing the life cycle
- Patterns to address these issues :

  Aggregate

  Factory

  Repository

The Life Cycle of a Domain Object - Aggregate

- An AGGREGATE is a cluster of associated objects that we treat as a unit for the purpose of data changes.
- Each AGGREGATE has a root and a boundary.
- The root ENTITY has global identity and is ultimately responsible for checking invariants.
- Root ENTITIES have global identity. ENTITIES inside the boundary have local identity, unique only within the AGGREGATE.

The Life Cycle of a Domain Object - Aggregate

- Nothing outside the AGGREGATE boundary can hold a reference to anything inside, except to the root ENTITY.

  The root ENTITY can hand references to the internal ENTITIES to other objects, but those objects can use them only transiently, and they may not hold on to the reference. The root may hand a copy of a VALUE OBJECT to another object, and it doesn't matter what happens to it, because it's just a VALUE and no longer will have any association with the AGGREGATE.

- As a corollary to the previous rule, only AGGREGATE roots can be obtained directly with database queries. All other objects must be found by traversal of associations.

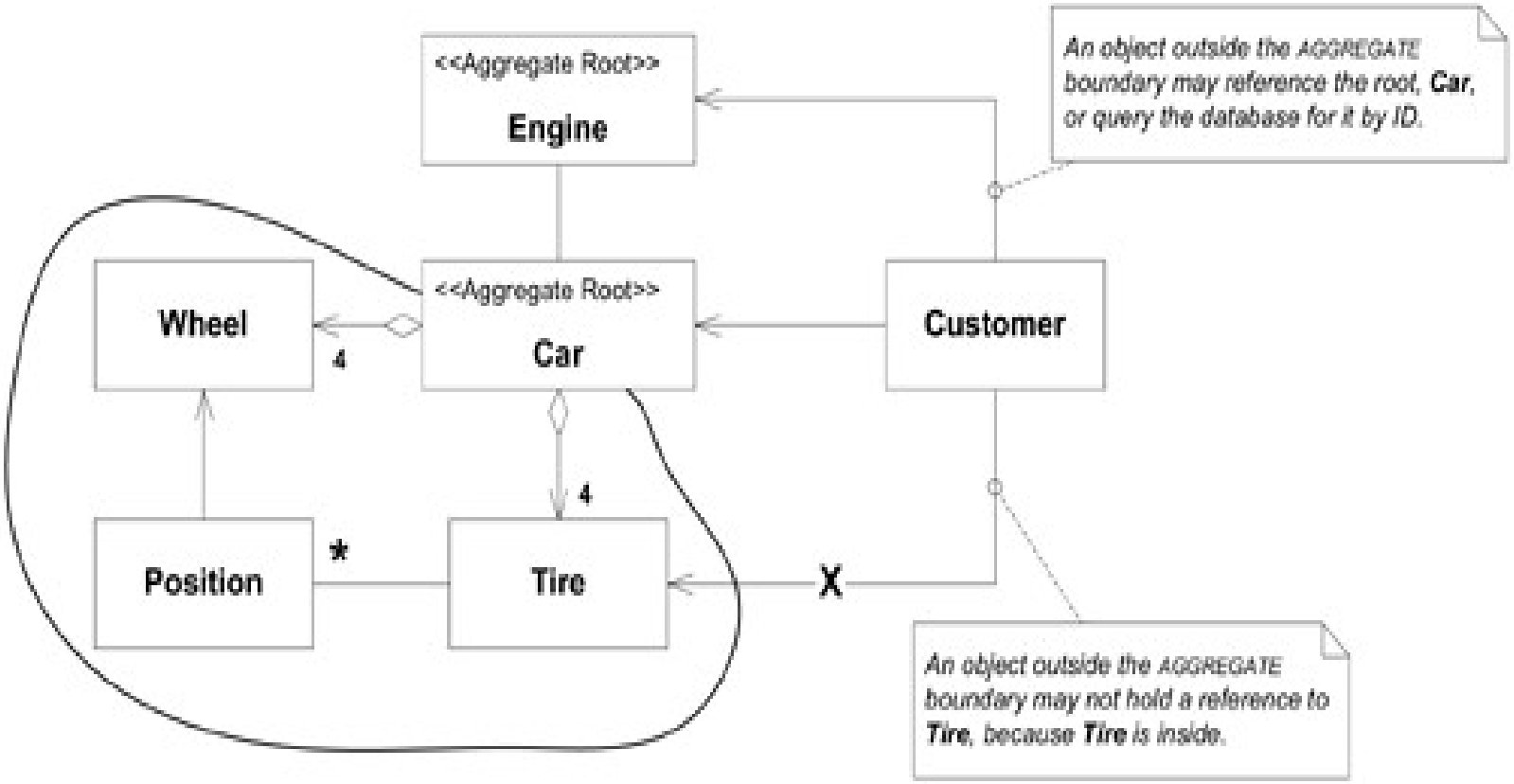
The Life Cycle of a Domain Object - Aggregate

- Objects within the AGGREGATE can hold references to other AGGREGATE roots.
- A delete operation must remove everything within the AGGREGATE boundary at once.

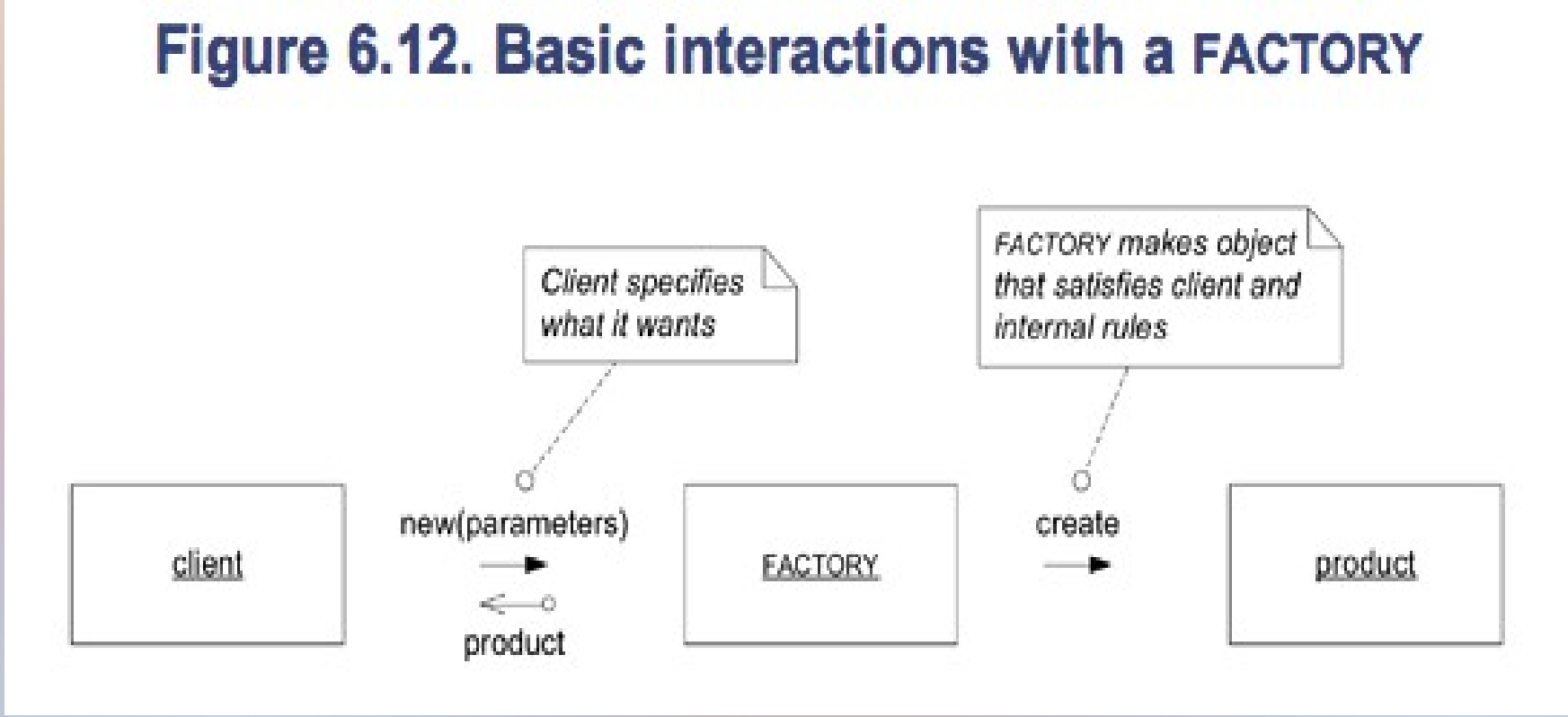
  (With garbage collection, this is easy. Because there are no outside references to anything but the root, delete the root and everything else will be collected.)
- When a change to any object within the AGGREGATE boundary is committed, all invariants of the whole AGGREGATE must be satisfied.

The Life Cycle of a Domain Object - Aggregate

Figure 6.2. Local versus global identity and object references



The Life Cycle of a Domain Object - Factory

- A program element whose responsibility is the creation of other objects is called a FACTORY.
- 

The Life Cycle of a Domain Object - Factory

- The two basic requirements for any good FACTORY are :

  1. Each creation method is atomic and enforces all invariants of the created object or AGGREGATE.

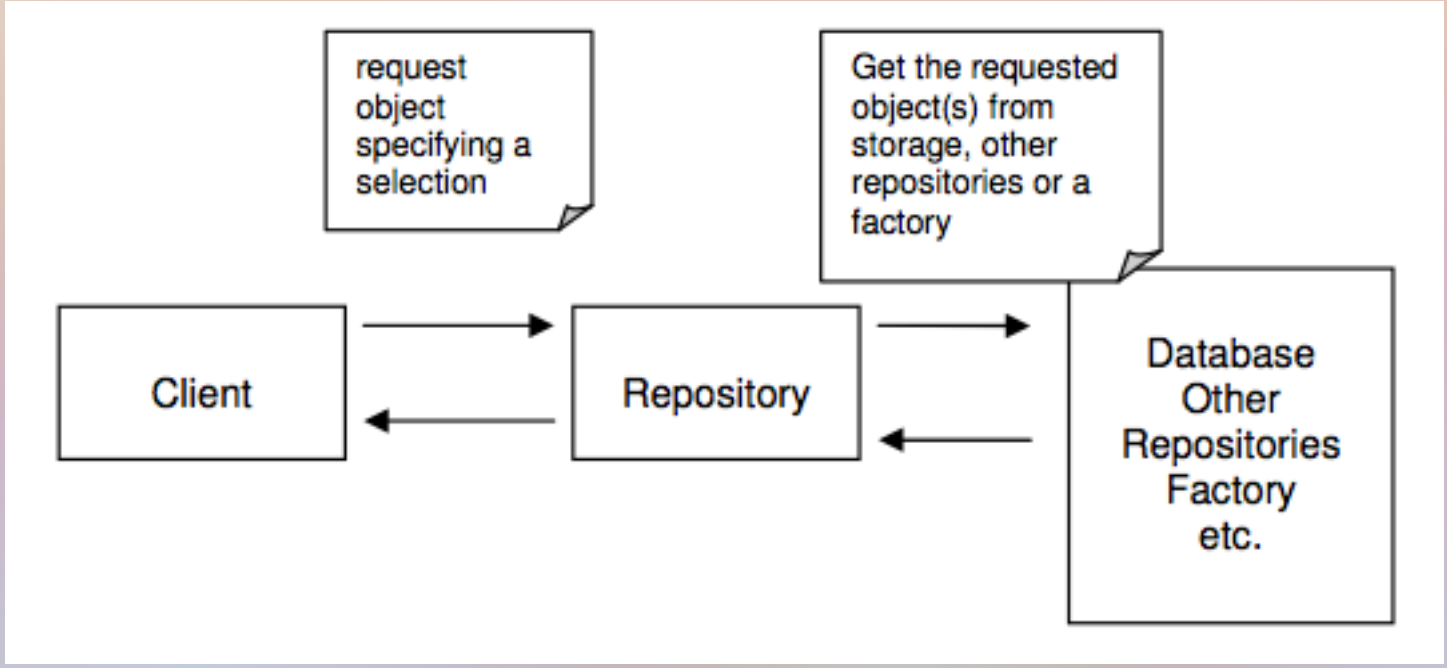
  2. The FACTORY should be abstracted to the type desired, rather than the concrete class(es) created.

ENTITY FACTORIES
vs
VALUE OBJECT FACTORIES

- VALUE OBJECTS are Immutable; the product comes out complete in its final form. So the FACTORY operations have to allow for a full description of the product.
- ENTITY FACTORIES tend to take just the essential attributes required to make a valid AGGREGATE. Details can be added later if they are not required by an invariant.

The Life Cycle of a Domain Object - Repository

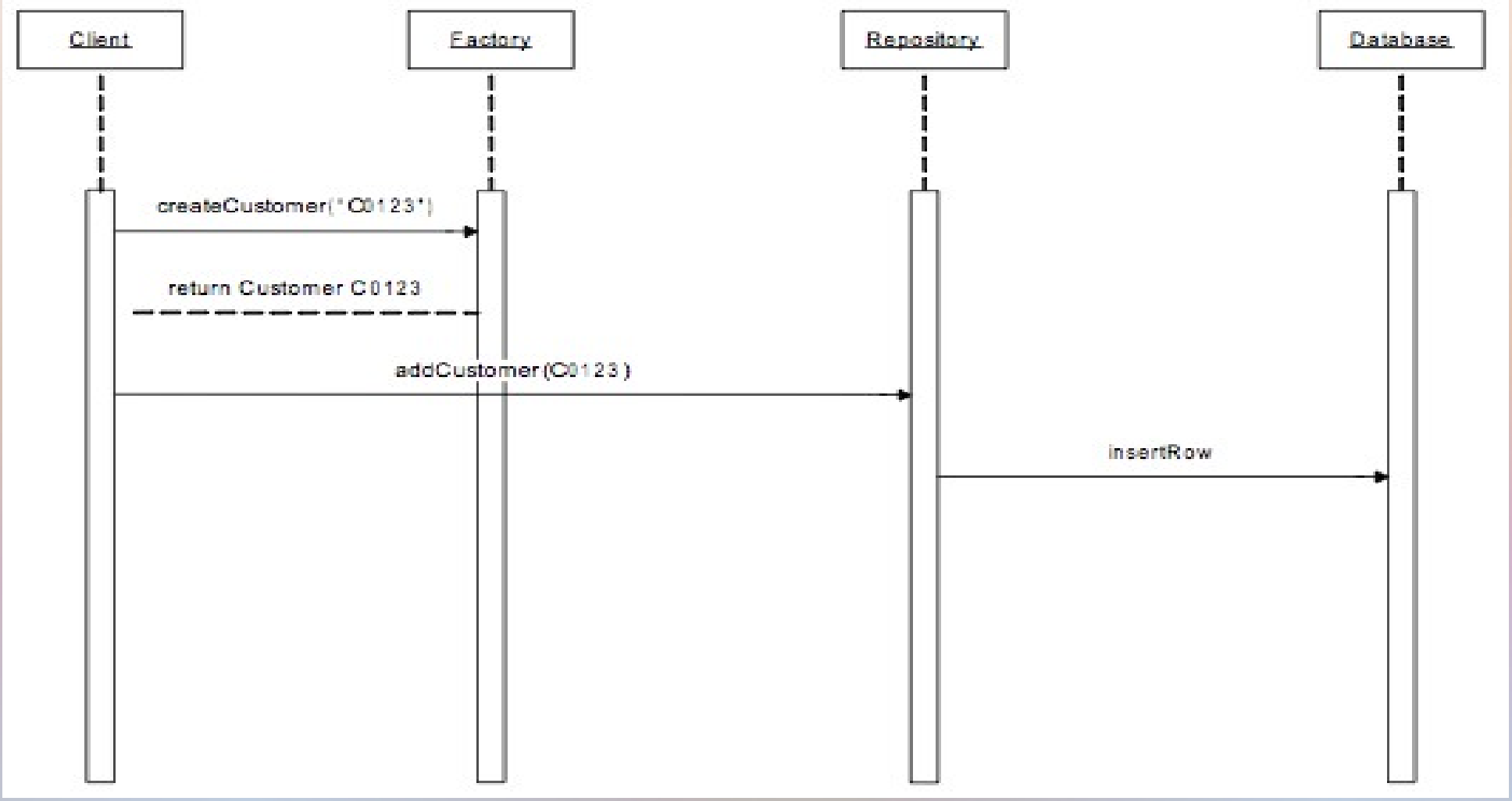
- Use a Repository to encapsulate all the logic needed to obtain object references
-



Relationship between Factory and Repository

- They are both patterns of the model-driven design, and they both help us to manage the life cycle of domain objects
- Factory is concerned with the creation of objects
- Repository takes care of already existing objects

An example


Client
Factory
Repository
Database
createCustomer("C0123")
return Customer C0123
addCustomer(C0123)
insertRow

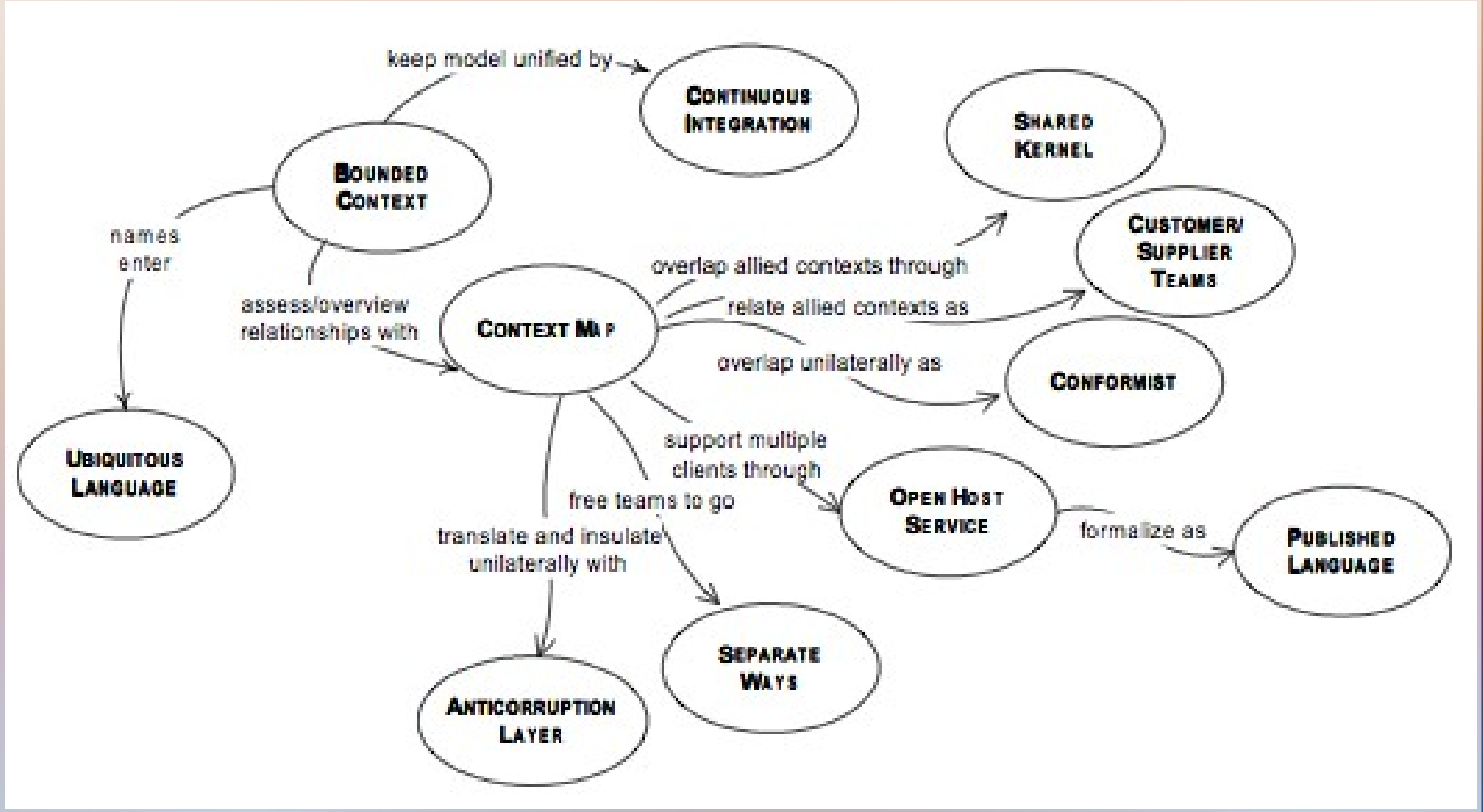
Refactoring Toward Deeper Insight

- Continuous Refactoring
- Bring Key Concepts Into Light

Strategic Design - Maintaining Model Integrity

- Large projects which require the combined efforts of multiple teams.
- Divide one big model into several models
- Present a set of techniques used to maintain model integrity

Maintaining Model Integrity



Bounded Context

- A model should be small enough to be assigned to one team
- Define the scope of a model, to draw up the

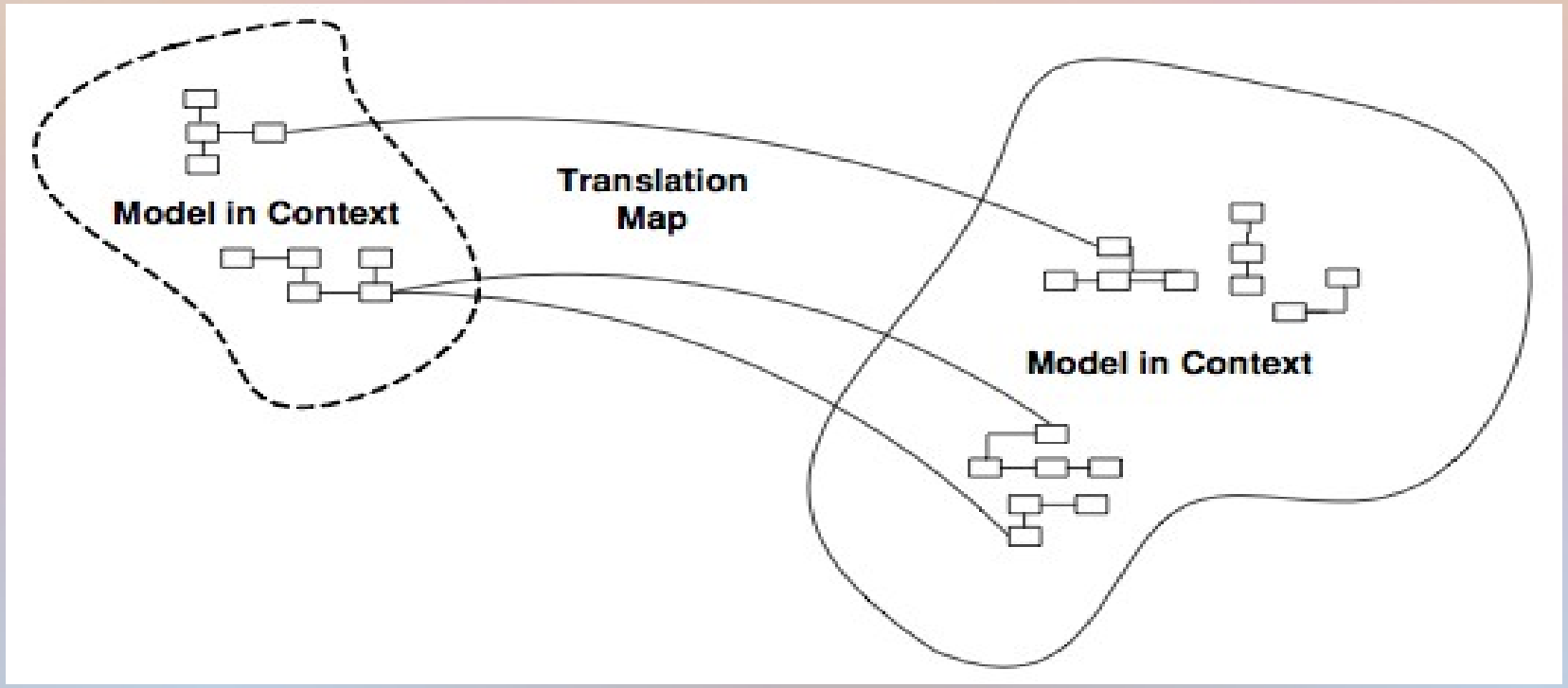
  boundaries of its context, then do the most possible to keep the model unified.
- Explicitly define the context within which a model applies. Explicitly set boundaries in terms of team organization, usage within specific parts of the application, and physical manifestations such as code bases and database schemas. Keep the model strictly consistent within these bounds, but don't be distracted or confused by issues outside.

Continuous Integration

- Make sure that all the new elements which are added fit harmoniously into the rest of the model, and are implemented correctly in code.
- The sooner we merge the code the better
- Automatically built, automatically test

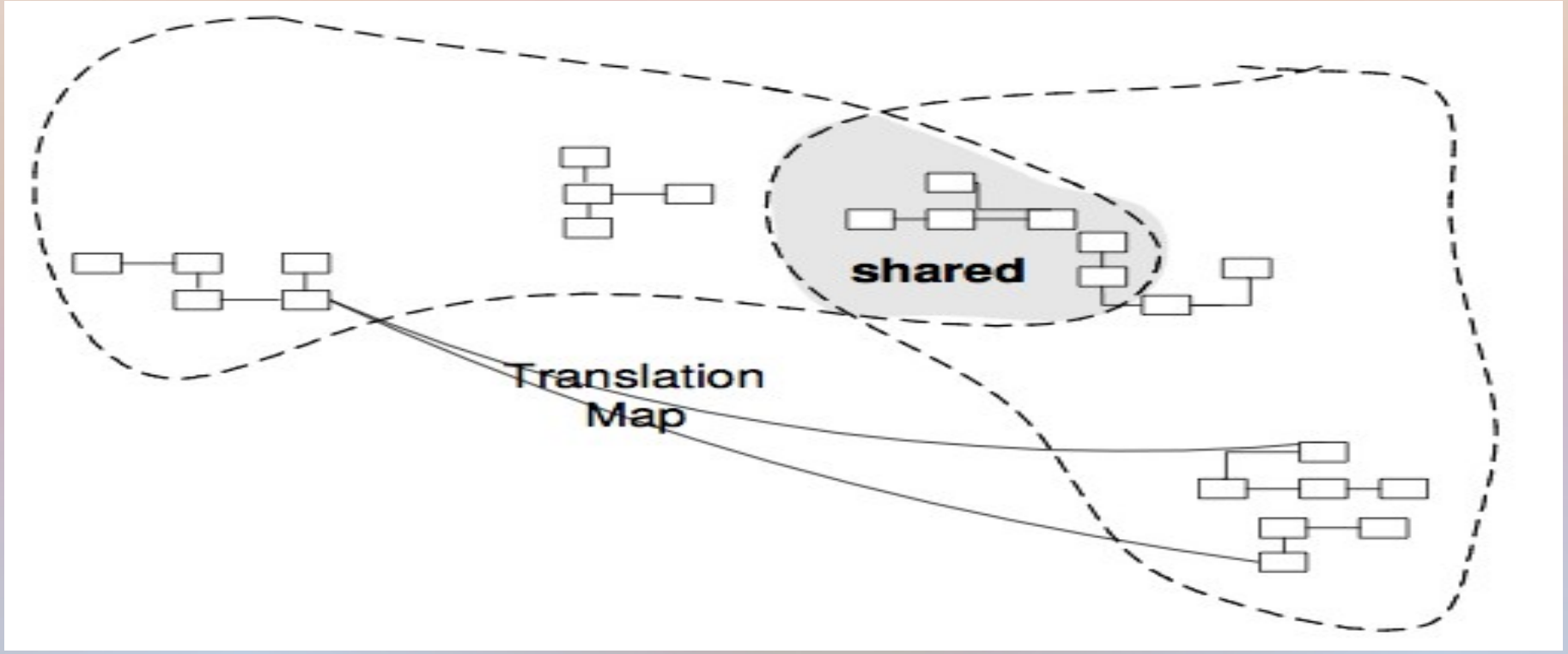
Context Map

- A document which outlines the different Bounded Contexts and the relationships between them
-



Shared Kernel

- The purpose of the Shared Kernel is to reduce duplication, but still keep two separate contexts



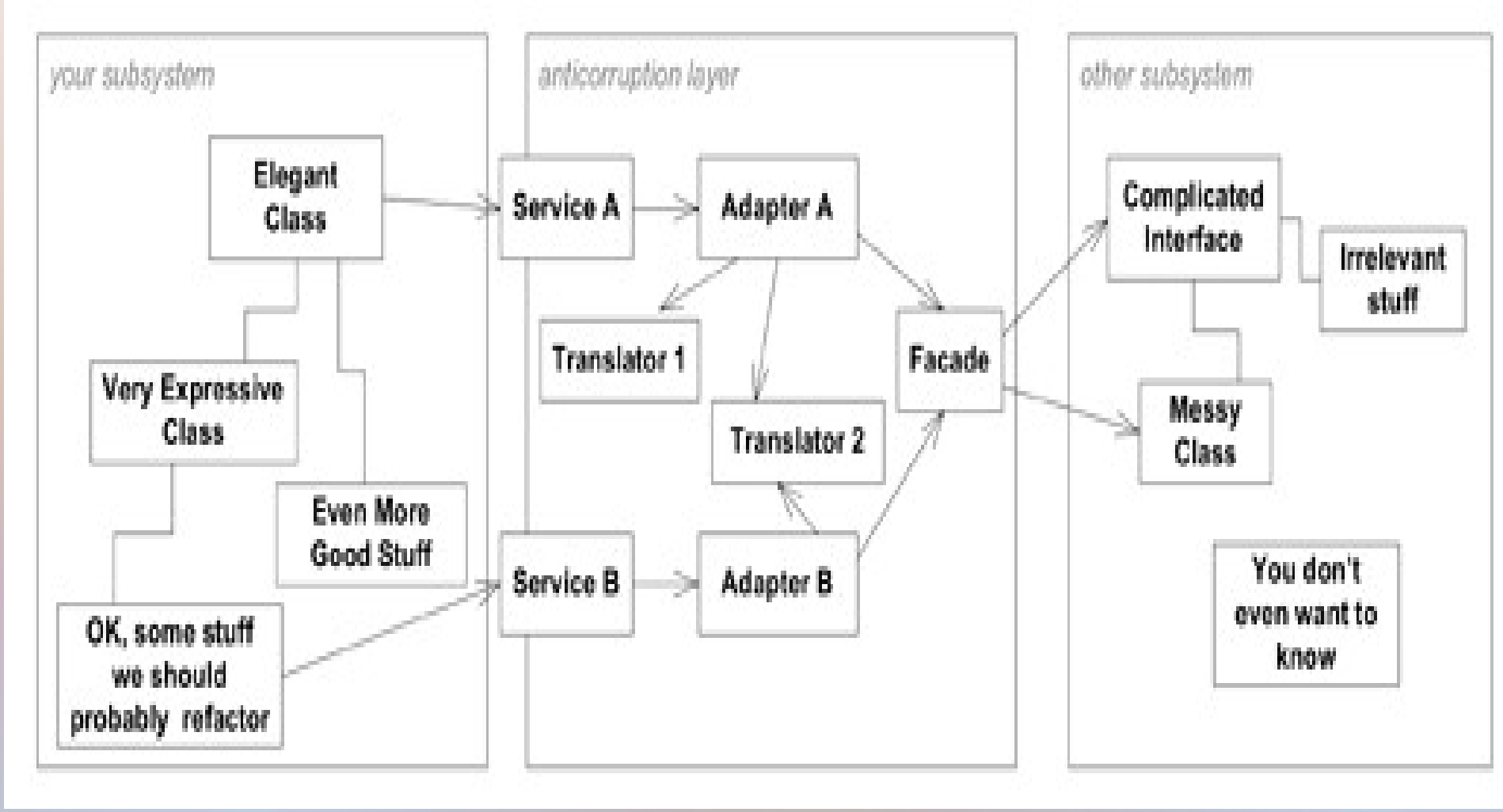
Customer/Supplier

- Customer-Supplier relationship is one subsystem depends a lot on the other subsystem
- Establish a clear customer/supplier relationship between the two teams
- Jointly develop automated acceptance tests that will validate the interface expected

Conformist

- Following isn't always bad
- If the customer has to use the supplier team’s model, and if that is well done, it may be time for conformity

Anticorruption Layer



Separate Ways

- Integration is always expensive. Sometimes the benefit is small.
- Declare a BOUNDED CONTEXT to have no connection to the others at all, allowing developers to find simple, specialized solutions within this small scope.

Open Host Service

- When a subsystem has to be integrated with many others, customizing a translator for each can bog down the team. There is more and more to maintain, and more and more to worry about when changes are made.
- Define a protocol that gives access to your subsystem as a set of SERVICES. Open the protocol so that all who need to integrate with you can use it. Enhance and expand the protocol to handle new integration requirements, except when a single team has idiosyncratic needs. Then, use a one-off translator to augment the protocol for that special case so that the shared protocol can stay simple and coherent.

Thoughts & Unsolved Problems

- Transactions

  transaction initiation and participation

  cross-repository transactions

  automatic enlisting

- Generics

  make it difficult to put repository in base class

  slightly more cumbersome than using non-generic classes

DDD Examples

- TimeAndMoney
- Strategic Design at StatOil (Norway's national oil company)

References

- [DDD] Eric Evans; Domain-Driven Design
- [DDD Quickly] Eric Evans; Domain-Driven Design Quickly
- http://www.domaindrivendesign.org/