# Domain-Driven Design(DDD)

Lei Bao and Zhaochuan Shen

## What is Domain-Driven Design (DDD)?

- Domain: the problem area
- The term was invented by Eric Evans, the author of book *Domain-Driven Design*, published this famous book in 2004.
- According to Eric Evans, "DDD flows from the premise that the heart of software development is knowledge of the subject matter."
- According to Wikipedia, "DDD is an approach to developing software for complex needs by connecting the implementation to an evolving model."
- In this presentation, we introduces the principles and patterns that should be used when modeling the domain.

## Premise of DDD

The premise of DDD is as following:

- Concentrating the main goal of the project on the central domain and its logic;
- Establishing the fundamentals of complicated designs on models of the domain;
- Emphasizing  a collaboration between technical developers and domain experts to set up a conceptual model that focuses on particular domain problems.

## Core Concepts

- Model Driven Design (MDD)
- Ubiquitous Language
- Layered Architecture
- Main Domain Elements (Building blocks): Entities, Value, Services, Modules
- Solutions to maintain domain objects: Aggregates, Factories, Repositories

## Model-Driven Design(MDD)

- MDD is a natural result of DDD since developers obtain their knowledge of domain as models.

- Model refers to a set of abstractions that (partially) depicts the domain and can solve problems from that domain.

- An MDD is software organized by a set of domain concepts and requirement. For example, an MDD for an insurance software framework is one in which insurance concepts, such as quoting, auditing and billing, which also corresponds to software constructs.

## DDD and MDD

- MDD puts a domain model into the structure and design of a software system.

  This enhances the feedback between describing and learning the domain, and implementing the required system that are focusing on problems in that domain.

- Developers who implement MDD should know that a modification to the code is actually a modification to the model. A modification to the model also leads to immediate modification to the code.

## Ubiquitous Language(I)

❑ In reality:

- Technical experts (developers) speaks technical language of computer science (technical terms);
- Domain experts (clients) use terminology specified to their field of expertise (domain/business terms).

Some standard language should be built among them!

❑ The goal is: this standard language and shared vocabulary can be understood by both the technical developers and the domain experts.

## Ubiquitous Language(II)

- ❑ Ubiquitous Language refers to a common language structured around the domain model and shared by both domain experts and technical developers to communicate in activities related to the software development.

- ❑ Ubiquitous Language is defined within bounded context.
    - Context: the setting in which a word or statement appears that determines its meaning.
    - Bounded context: 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, and don’t be distracted or confused by issues outside.

## Ubiquitous Language(III)

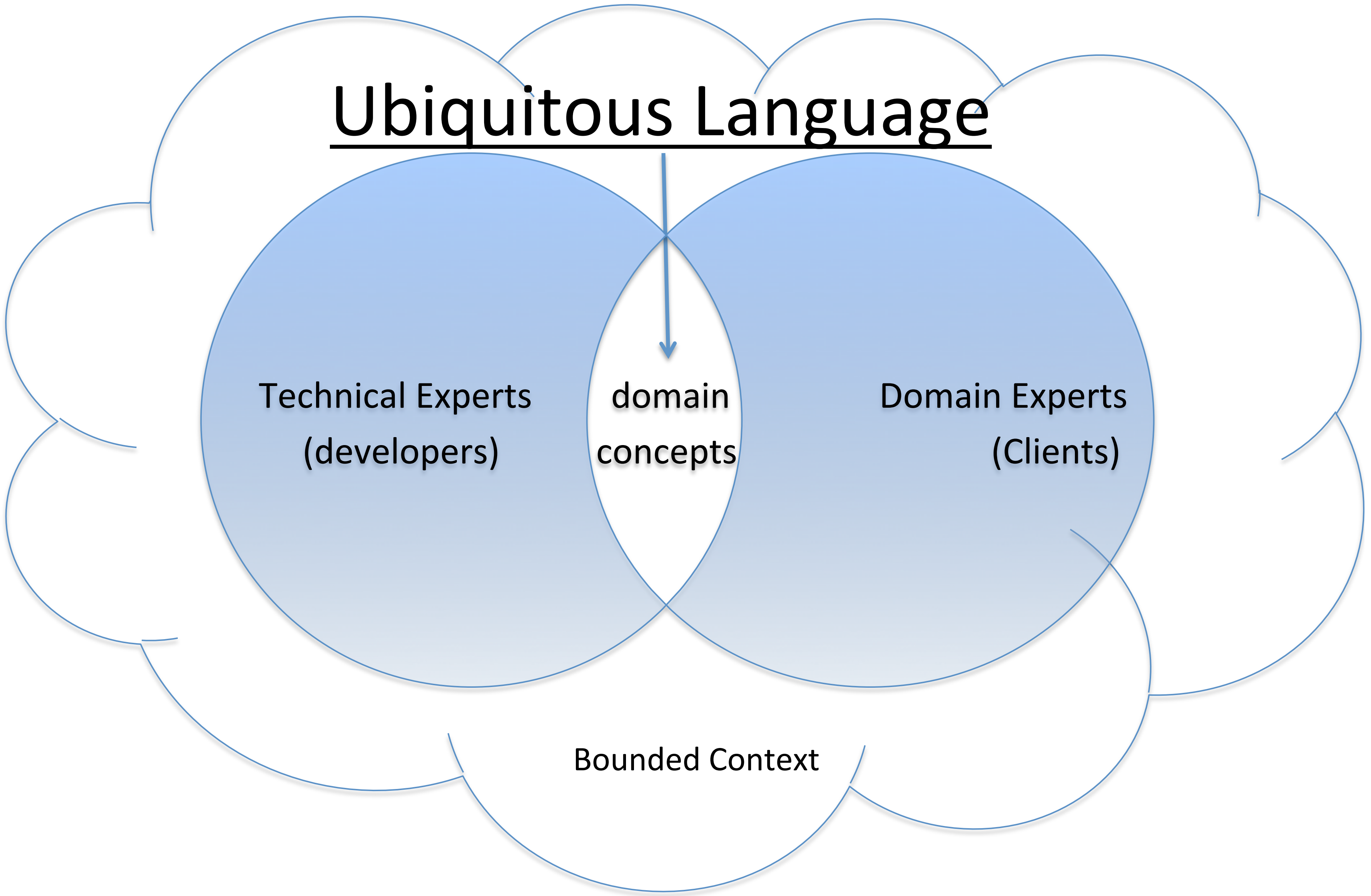
Using ubiquitous Language:

- ❑The domain model is used as the core of ubiquitous language.
- ❑Now no “translation” between developers and domain experts is usually needed. They can understand each other very well.
- ❑If there are new requirements, it usually means that new words enter the ubiquitous language.

## Ubiquitous Language(IV)

- ❑ One team one language: try to describe and discuss problems and requirements of the model during analysis in ubiquitous language. This helps reduce possible confusion in communication.

- ❑ Benefit of Ubiquitous Language:
  - Less risk of miscommunication
  - Faster and more efficient interaction
  - Knowledge of domain can reside in codebase
  - Source code easier to understand (maintainability and extensibility)


Ubiquitous Language
Technical Experts (developers)
domain concepts
Domain Experts (Clients)
Bounded Context

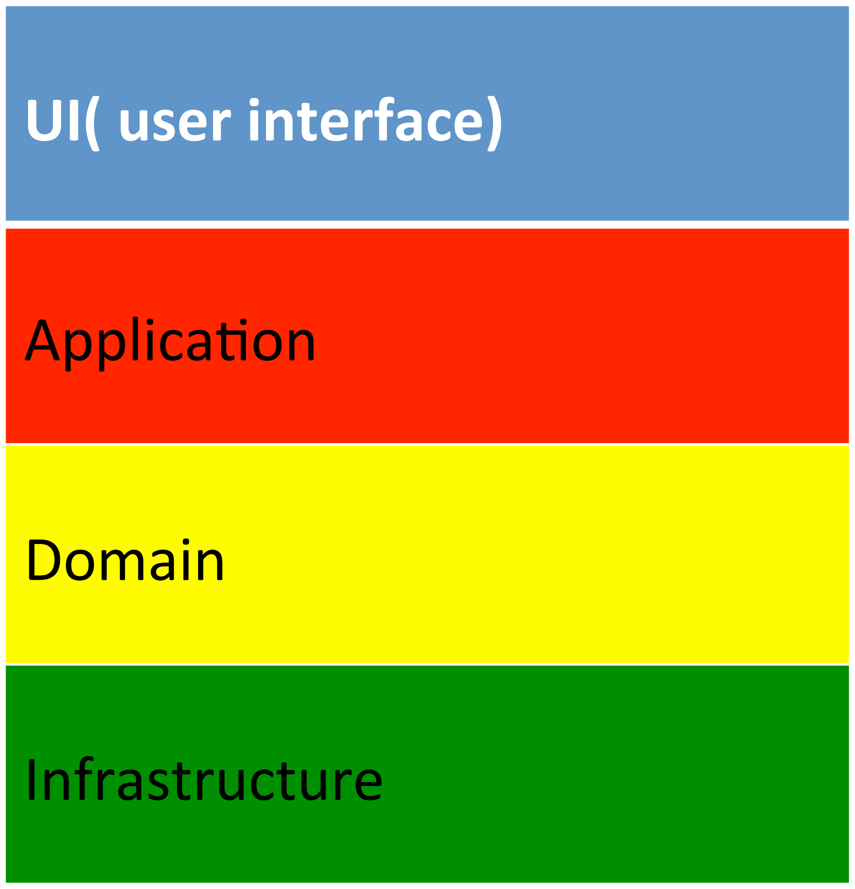
## Layered Architecture

- ❑ Software that is built with DDD takes advantage of a layered architecture. Different layers address different concerns.

- ❑ The basic principle of layered architecture:
  - Dependencies between different layers is one-way, i.e. a layer never knows anything above. Within a layer, an object can use objects in its layer and in layers below it.

  - An indirect method is usually required when an object in a lower layer really needs to have reference to an object in the layer above it.

## Typical decomposition of multilayered architecture for DDD



- Domain layer is the heart of the DDD.
- Objects within the domain layer should be maximally isolated from the other three layers.
- Domain layer is responsible for all "domain logic"(such as rules, requirements, and behaviors for the domain model).

## Main domain elements (building block)

- ❑ "Entities", "values", "services" and "modules" are the main domain elements for a domain model.

- ❑ They are connected with "associations".

- ❑ Common pattern like "repositories" and "factories" helps to complete the model.
  - "Factories" are used to create domain objects;
  - "Repositories" are used to retrieve domain objects.

## Associations between elements

- As discussed in Professor Anderson's Lecture 3, associations refers to "one domain element can reference another".

- Association may decrease maintainability: avoid unnecessary associations and use only a minimum number of them.

- Making associations more controllable:
  - Use a traversal direction association instead of bidirectional,
  - Use qualifier to minimize multiplicity,
  - Remove unnecessary associations.

## Entities(I)

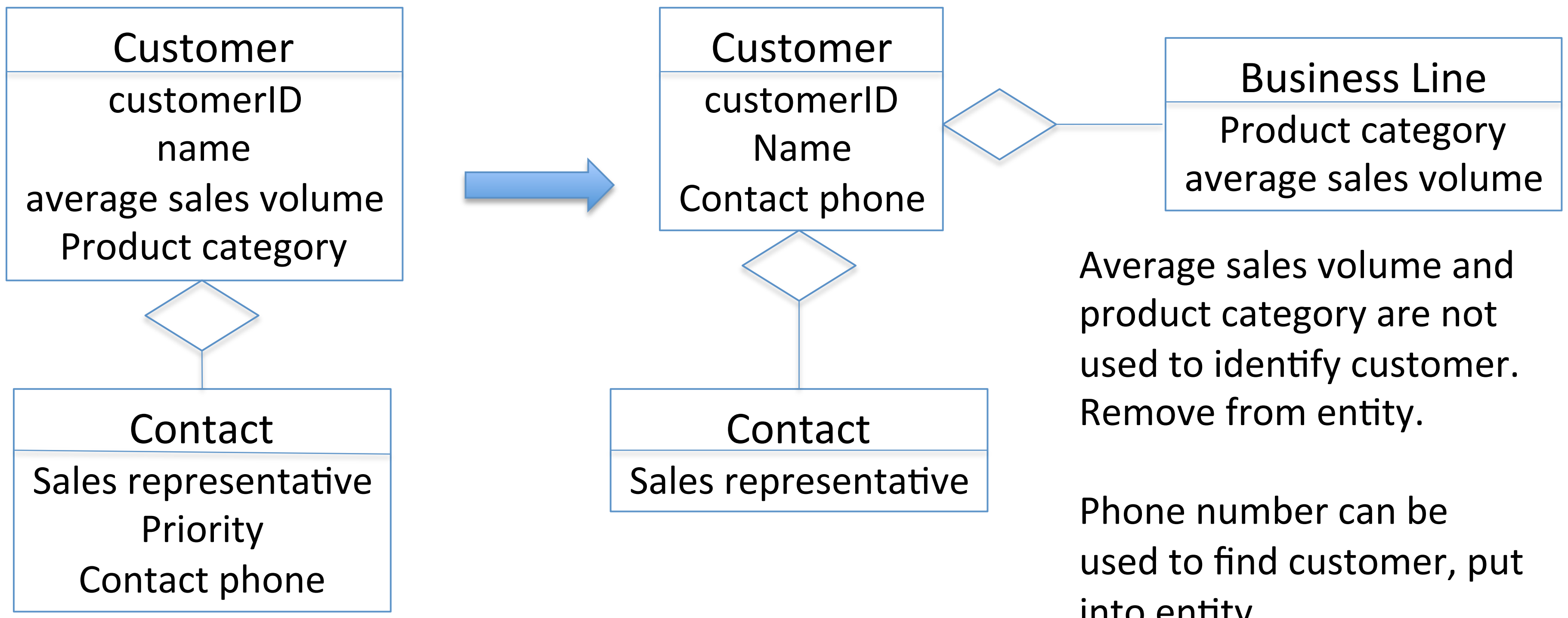
- ❑ Entity refers an object that is defined but by its identity, rather than its attributes.

- ❑ Eric Evans claims that " They (entities) represents a thread of identity that runs through time and often across distinct representations."
  - The identity we talk about here is not the identity in Objected-oriented program(OOP), which is a reference or a memory address OOP languages uses to keep track of the objects instance in memory.
  - Usually, identity in DDD is either an attribute of the object, a combination of attributes, or even a behavior.

- ❑ Example: For airlines that distinguish each seat uniquely on each airplane, each seat is an entity under this context.

# Entities(II)

Implementing entities means creating identity. We should only focus on object's important characteristics that are identifying and/or can be used to find it. Core entities may associate other attributes or behaviors of the object. Eric Evans gives the following example on how to establish entities on Page 95:



Average sales volume and product category are not used to identify customer. Remove from entity.

Phone number can be used to find customer, put into entity.

## Be cautious to make all objects entities

- Entities can be tracked. Tracking and creating identity comes with a cost.
- It takes a lot of careful thinking to determine what makes an identity.
- Some performance implication in making all objects entities: one individual instance for each object.

→ This can lead to system performance degradation when dealing with thousands of instance.

So, there are cases when we just need to have some attributes of a domain element and we are not interested in which object it is, but what attributes it has.

## Values(I)

- Value (objects) refers to an object that contains descriptive attributes but has no conceptual identity. They describe the characteristic of a thing.
- Example: colors is an example of value object for the symbols displayed on the monitor.
- Value objects and entities may be different from different perspective. When people exchange dollar bills, they don't distinguish between every bill; they only care the face value of each bill. In this context, dollar bill is a value object; however, in the eye of the Federal Reserve (who prints the money), each bill is unique. In this context, each dollar bill is an entity.

## Values(II)

- Value objects can reference to other objects, and they are even allowed to reference an entity.
- Since value objects are usually sharable, they should be immutable. They are created with a constructor, and never modified during their lifetime. Since we need no care of identity: we can simply delete and create new ones if needed.
- For example, if you want to change the color of something, you destroy the old color and create a new one. Usually the creation is done through colorservice or colorfactory.

## Services (I)

Sometimes an operation or process from the domain is not natural to entity or value objects. Forcing to put such an operation into an object would either damage the definition of objects or create artificial objects. If so, in DDD, we can usually add a standalone interface, as **SERVICE**, to the model to minimize the artificial objects.

A good service meets at least the following requirements:

- it should be stateless;
- it should be defined in the common (ubiquitous) language;
- It relates to entities or value objects, but not part of them.
- It focuses on activities rather than entities, and therefore has a verb name.

## Services (II)

Most times services are within the infrastructure layer. However, domain and application layer services may work together with infrastructure services to finish operations.

Page 107, Eric gave the following example of how to partition transfer funds services into layer:

| | |
|---|---|
| Application | **Fund Transfer App Service**<br>• Got input requirement<br>• Send requirement to domain services to accomplish<br>• Wait for conformation<br>• Send notice by infrastructure services |
| Services | **Fund Transfer Domain Service**<br>• Interact with involved accounts (objects) and make transfer<br>• Send Conformation |
| Services | Send Notice Services<br>• Send emails to customers |

## Modules

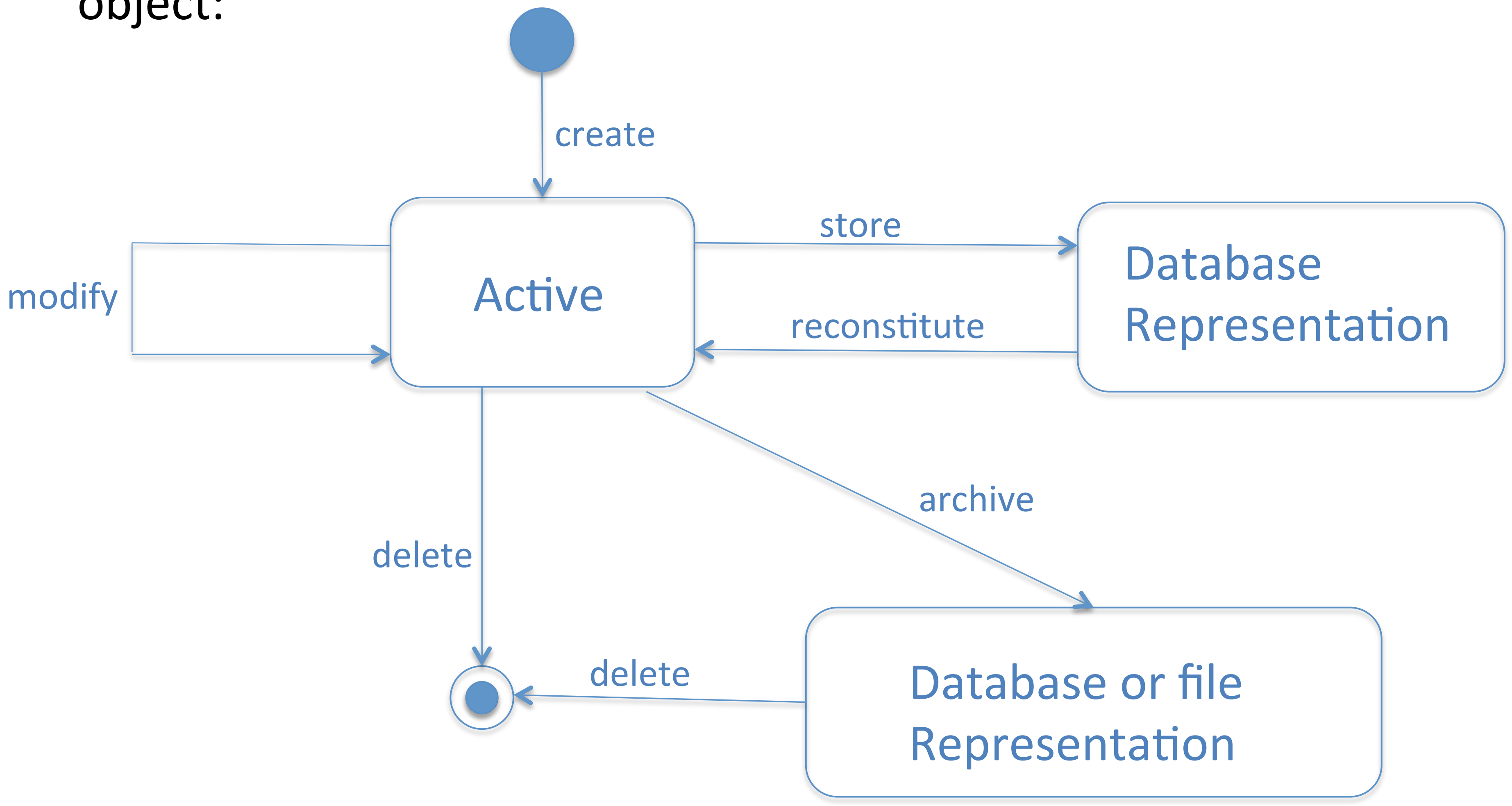
Modules are groups of model elements. If the whole system is a book, each module is a chapter.

Using module has the following advantage:

- Modules help the understanding of a huge system;
- Modules expedite the independent parallel development of system.
- Using modules satisfy the requirement of Object Oriented design: weak coupling (minimum associations between modules) and strong cohesion (one module handles one thing).

# Life Cycle of Domain Object

On Page 123, Eric Evans shows a diagram for life of domain object:



## Challenges and Prescriptions

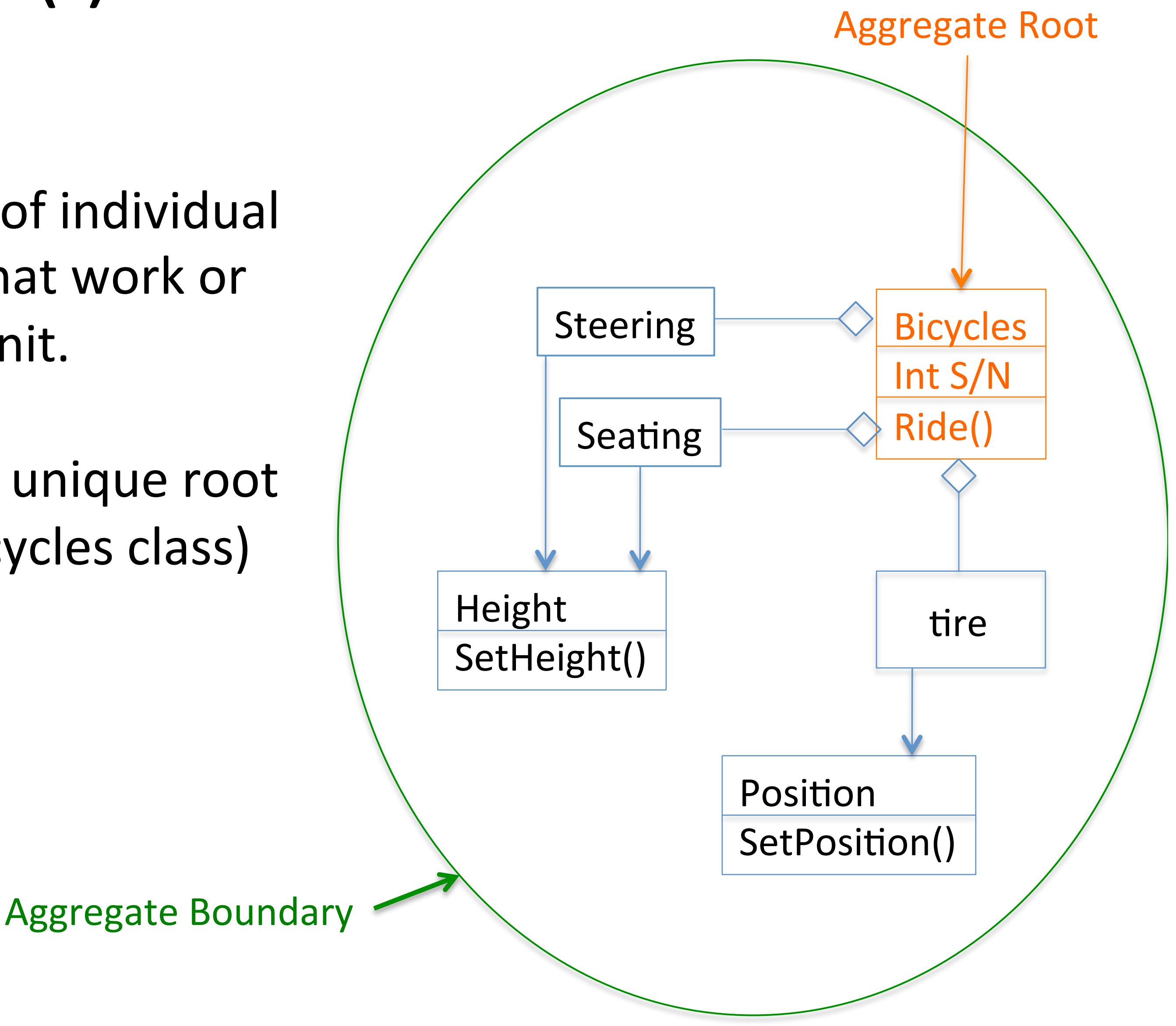
- Challenges:
  - ➢ Maintain the invariants and rules during the whole life of object;
  - ➢ Prevent the model from getting complicated by checking and handling the life cycles of objects.

- Prescriptions
  - ✓ Aggregates (tight up)
  - ✓ Factories (create)
  - ✓ Repositories (find and retrieve)

# Aggregates (I)

Aggregate is a group of individual but related objects that work or can be treated as a unit.

Each aggregate has a unique root entity (here is the bicycles class) and a boundary.



## Aggregates (II)

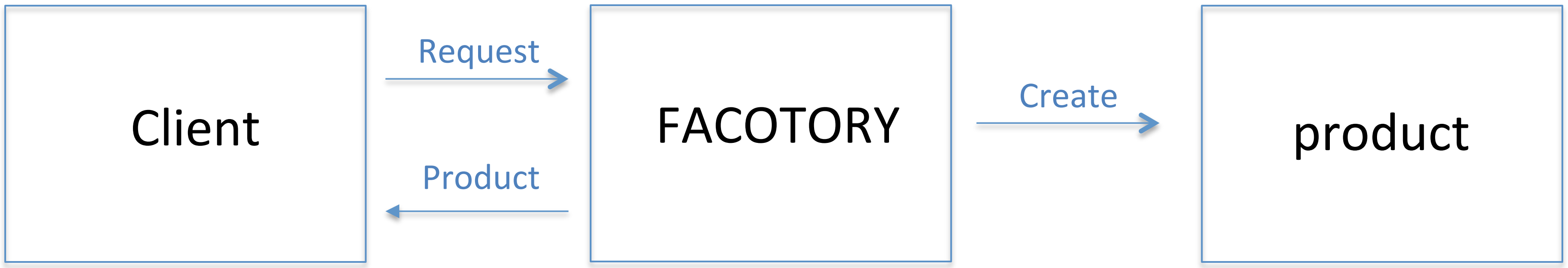
- The root of aggregate is the only accessible entity to the client. All changes to the aggregate within the boundary are maintained through the root. Enforcement of encapsulation!
- If the client deletes the aggregate, all the entities within the boundary will be deleted together.
- If there is a change to any object within the aggregate boundary, all the invariants and rules must be maintained. The root entity often takes care of these requirements.

## Factories(I)

Factories are a separate object (or interface) that in charge of creation for the instances of complex objects, and particularly aggregates. The bottom diagram shows the interactions of factory By Eric Evans, page 138.

Three common design patterns related to factories:

- Abstract factory
- Factory method
- Builder



## Factories(II)

A good factor has the following characteristics:

- It should be atomic.
- It is not allowed to give wrong results. If the required object can not be initialized, an exception message should be passed.
- It should give an abstract type of product if possible, not a concrete type.
- Arguments are usually necessary for factories.

There are also factories to reconstitute objects.

## Factories (III)

Entity Factories *VS* Value object Factories:

- Entity Factories have to assign an id to its product; this is not true for value object factories;
- Entity factories may only finish the required attributes for its product, and have other details added later. Value objects from its factory is in its final state.

Factories *VS* Factories for reconstitution

- Reconstitution factories do not assign a new id for entity factory;
- Reconstitution factories need to deal with the violation of invariants and rules, rather than simply give an exception.

Domain-Driven Design, Eric Evans, Page 144-145

## Repositories (I)

To get access to the existing domain objects, the developer of client may:

- Use traversable associations (if too many, muddle the model);
- Or search the object from database, if too many
  - Domain logic degenerates into queries and searchings;
  - Entities becomes simple data container;
  - Developer may eventually have to give up the domain layer…

Neither way works perfectly. Repository here helps!

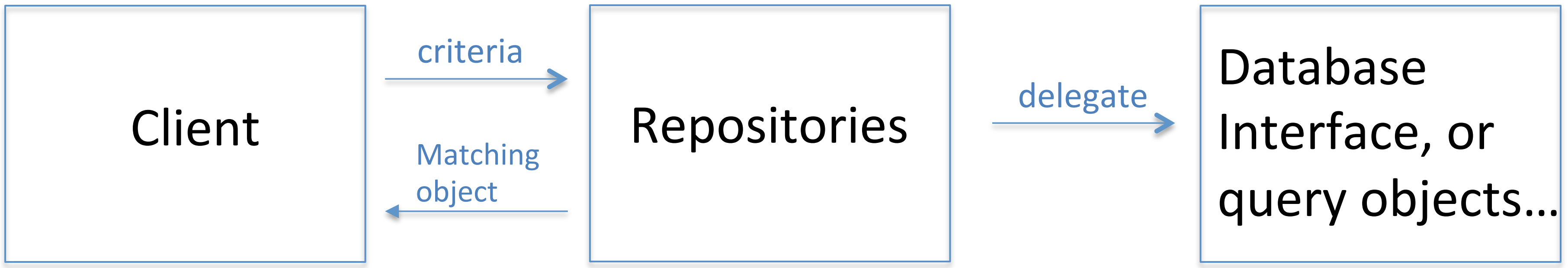
For repositories, please see Domain-Driven Design, Eric Evans, Page 260-261

## Repositories (II)

Repositories represents all objects which meets certain requirement as a conceptual collections with advanced searching capability.

Addition and removal of certain objects are achieved by the algorithm behind the repository, which inserts to or deletes from the database.

Here is a diagram of repository on Page 151 from Eric Evan's book.

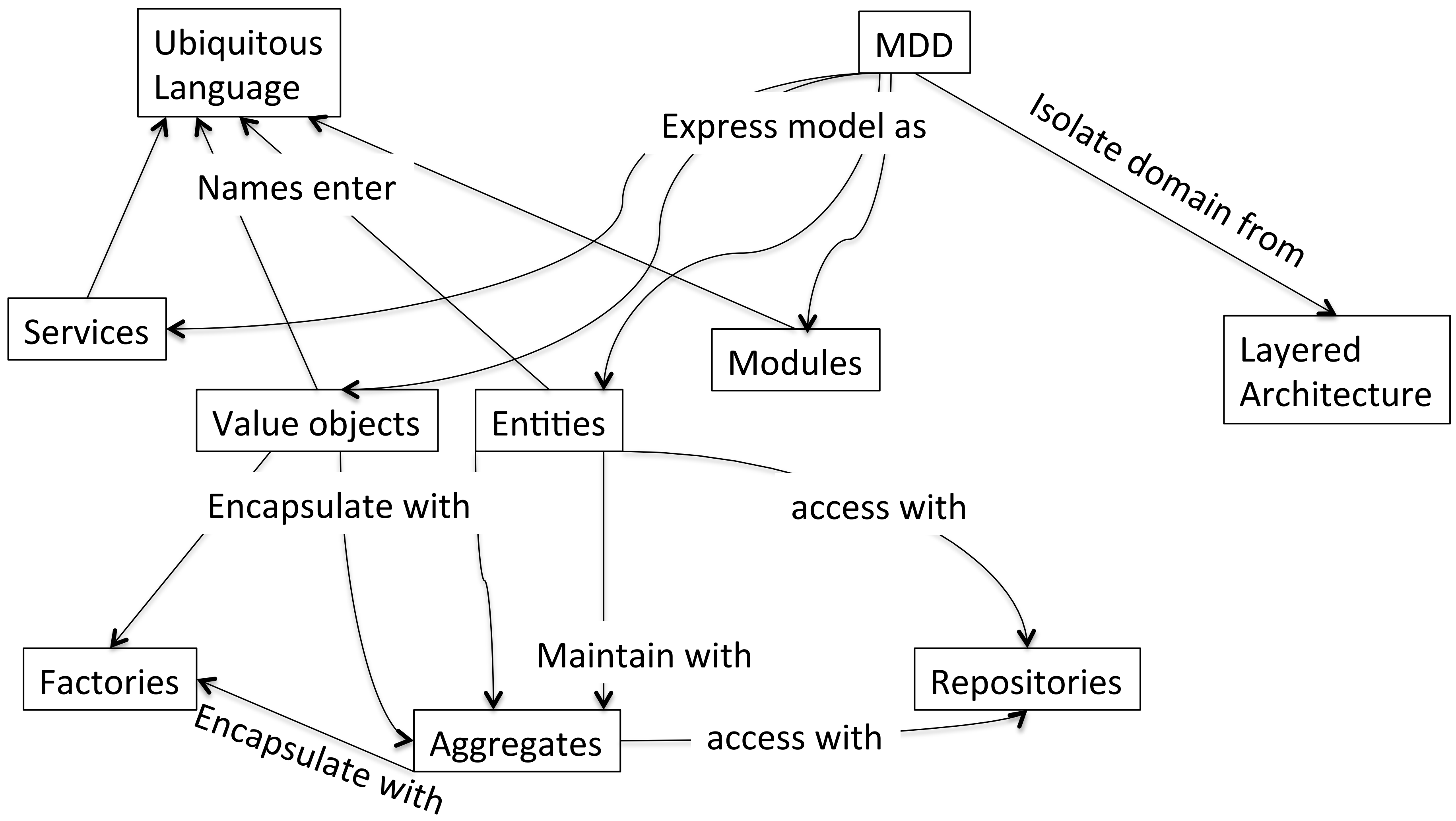


## Repositories (III)

Repositories have the following advantages:

- Repositories provide the clients an easy interface to acquire an existing object, and to maintain its life cycle.
- Repositories can communicate design ideas related to data access.
- Repositories decouple the client from the model from technical storage, or database storage;
- With repositories, dummy implementation for testing is simple and possible.

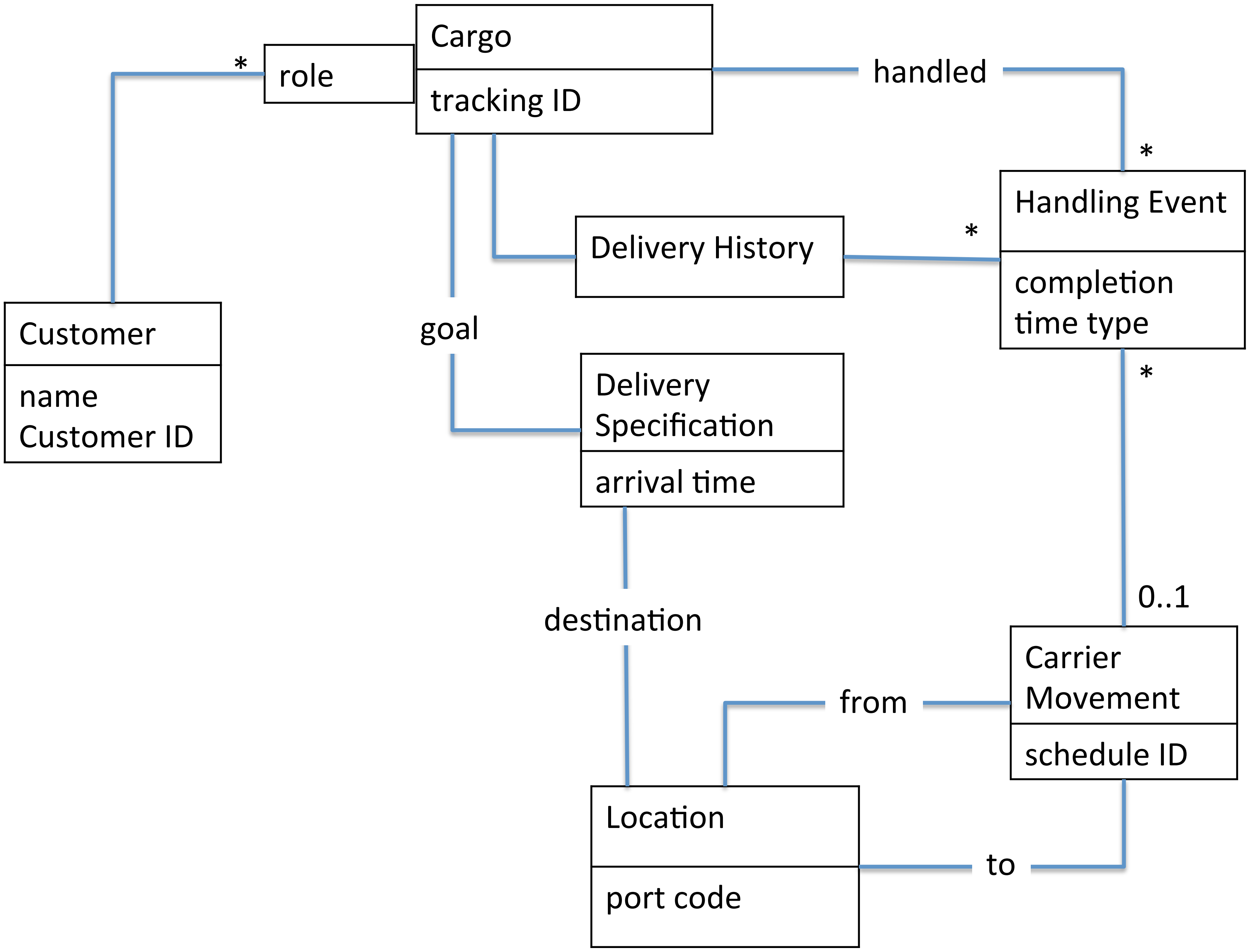
# Diagram of building blocks for DDD



Domain-Driven Design, Eric Evans, Page 65

## An example of Cargo Shipping

Whole Chapter 7 of Eric Evans' book gives an example of DDD. Here we give a very short review of this example. If you are interested, please refer to his book. His starting point is an established model, shown by the following diagram (Fig. 7.1 on Page 164. )


Cargo
tracking ID
role
*
handled
*
Handling Event
completion time type
*
Delivery History
*
Customer
name
Customer ID
goal
Delivery Specification
arrival time
destination
0..1
Carrier Movement
schedule ID
from
Location
port code
to

## Example Implementation (I)

First Eric isolated the domain by applying layered architecture. Here he introduced 3 application layer classes:

- Tracking Query
- Booking application
- Incident logging application

These three classes will not be included in the domain layer.

Second, for the objects in the previous slides, Eric identify if they are entities or value object.

- Entities: customer, cargo, handle event, carrier movement;
- Value object: Delivery specification.

## Example Implementation (II)

Third, Eric recheck the association in the class diagram. Some associations should not be bidirectional. For example, the customer should not have a direct reference to the cargo, since this could be problematic for return customer.

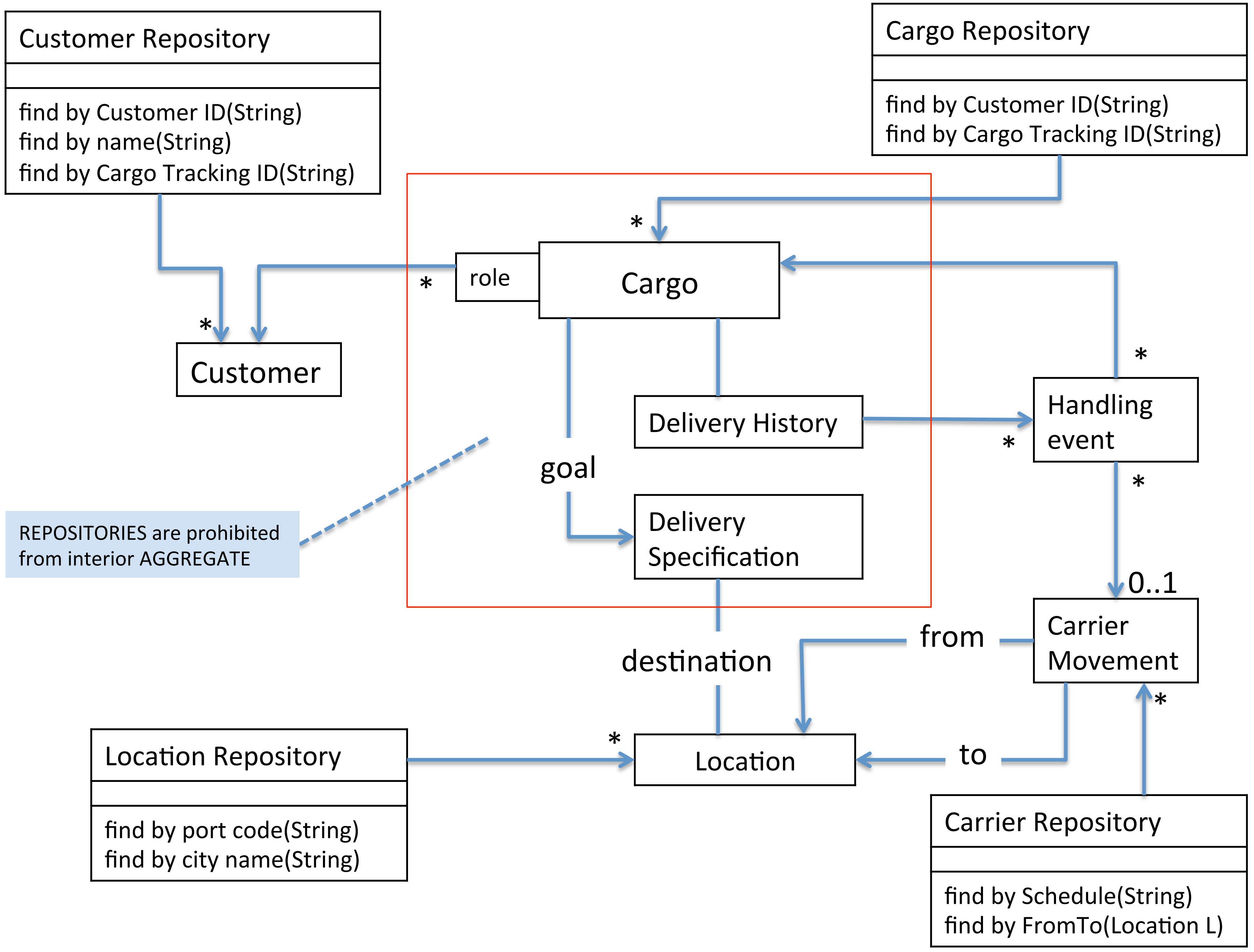
Forth, he aggregates the Delivery History and Delivery specification class into the cargo aggregate, with root of cargo. This step is to find aggregate.

Last, he finds necessary repositories for the entities. For example, the booking application to function, we need to search for customer from its repository. He found 4 repositories: customer repositories, cargo repositories, location repositories, and carrier movement repositories.

## Example Implementation (III)

Now, the class diagram looks like the following (Fig. 7.4 on Page 172. ). From here, Eric talked about modules in this model.

In the following slide, orange line indicates the boundary of aggregate.


Customer Repository
find by Customer ID(String)
find by name(String)
find by Cargo Tracking ID(String)
Cargo Repository
find by Customer ID(String)
find by Cargo Tracking ID(String)
*
role
Cargo
*
*
Customer
*
Handling event
Delivery History
*
goal
*
REPOSITORIES are prohibited from interior AGGREGATE
Delivery Specification
0..1
Carrier Movement
from
destination
*
Location Repository
find by port code(String)
find by city name(String)
*
Location
to
Carrier Repository
find by Schedule(String)
find by FromTo(Location L)

## Deep Model and Supple Design

- Deep model: a model that gets rid of superficial and captures all the essential of the problem. This results in a software that is more sensitive to the way the domain expert's think and more responsive to the their demands.

- Supple design is complement to deep model. With supple design, the software system is friendly to change of requirements. In other words, with supple design, changes in model can be realized in the code without too much extra effort.
  - A design must be easily understood.
  - A design must follow the contours of the deep model of the domain.
  - The code of a design should be extremely clear so that it is easy to anticipate the consequence of a change.

## Techniques required for Supple Design

We want to have a nice “supple design” in our implementation (easy maintenance and flexible extensibility).

❑ Making behaviors obvious to guarantee easy maintenance

- Intention-Revealing Interfaces
- Side-effect free functions
- Assertions

❑ Reducing cost of change to enforce flexible extensibility

- Standalone Classes
- Closure of Operations
- Conceptual Contours

## Intention-Revealing interface

- ❑ The interface of a class should provide the information about how the class can be used.

- ❑ During implementation, the name of classes and methods should reflect their effect and purpose, and therefore
  - these names should be from ubiquitous language if possible.
  - Think like a client of the method by writing a test case before creating the methods for each behavior.

## Side-effect-free functions (I)

- ❑ In general operations (methods) can be classified into two types:
  - Commands
  - Queries

- ❑ Commands are operations that affect the state of the system, which might cause side effect.
  - For side effects, the changes to the state of the system deviates from what we originally plan to do

- ❑ Queries are “read-only” operations. They only retrieve information from the system without changing system’s state, and of course should not cause side effect.

## Side-effect-free functions (II)

To increase the use of side-effect-free functions:

- Put the logic of the program into functions as much as possible;
- Strictly distinguish between the commands and queries to minimize the coupling between these two types of operations;
- Try to use the value objects when a concept that fits the responsibility.

Domain-Driven Design, Eric Evans, Page 250-251

## Assertions

- Assertions includes:
  - The preconditions that must be met before the operation, so the operation makes sense;
  - The post-conditions describes the state of the system after the operation;
  - Invariants is always true for a specific class.

- Therefore:
  - Clear describe the pre- and post-conditions for operations.
  - Build models on coherent sets of concepts. This makes developer easily master the model and minimize the risk of inconsistent code.

## Conceptual Contours

For a model that emerges at some point of the domain, it may show up at other parts of the domain later. Sometimes our system may require lots of effect to adjust to the new discovered model.

Therefore, we decompose the design elements into cohesive units, and looking for **conceptual contours** with which the system can easily get adapted to the new concepts.

The goal of conceptual contours is to find a collection of interfaces that are appropriate in ubiquitous language, and free (or minimized) of irrelevant objects and behaviors.

Domain-Driven Design, Eric Evans, Page 260-261

## Standalone Classes

High coupling between different parts of code contradicts the fundamental spirits of object oriented design.

Aggregates and modules are used to achieve the low coupling code.

Another technique is to use standalone classes, where one class does not take advantage of other domain concepts.

Domain-Driven Design, Eric Evans, Page 265-267

## Closure of operations

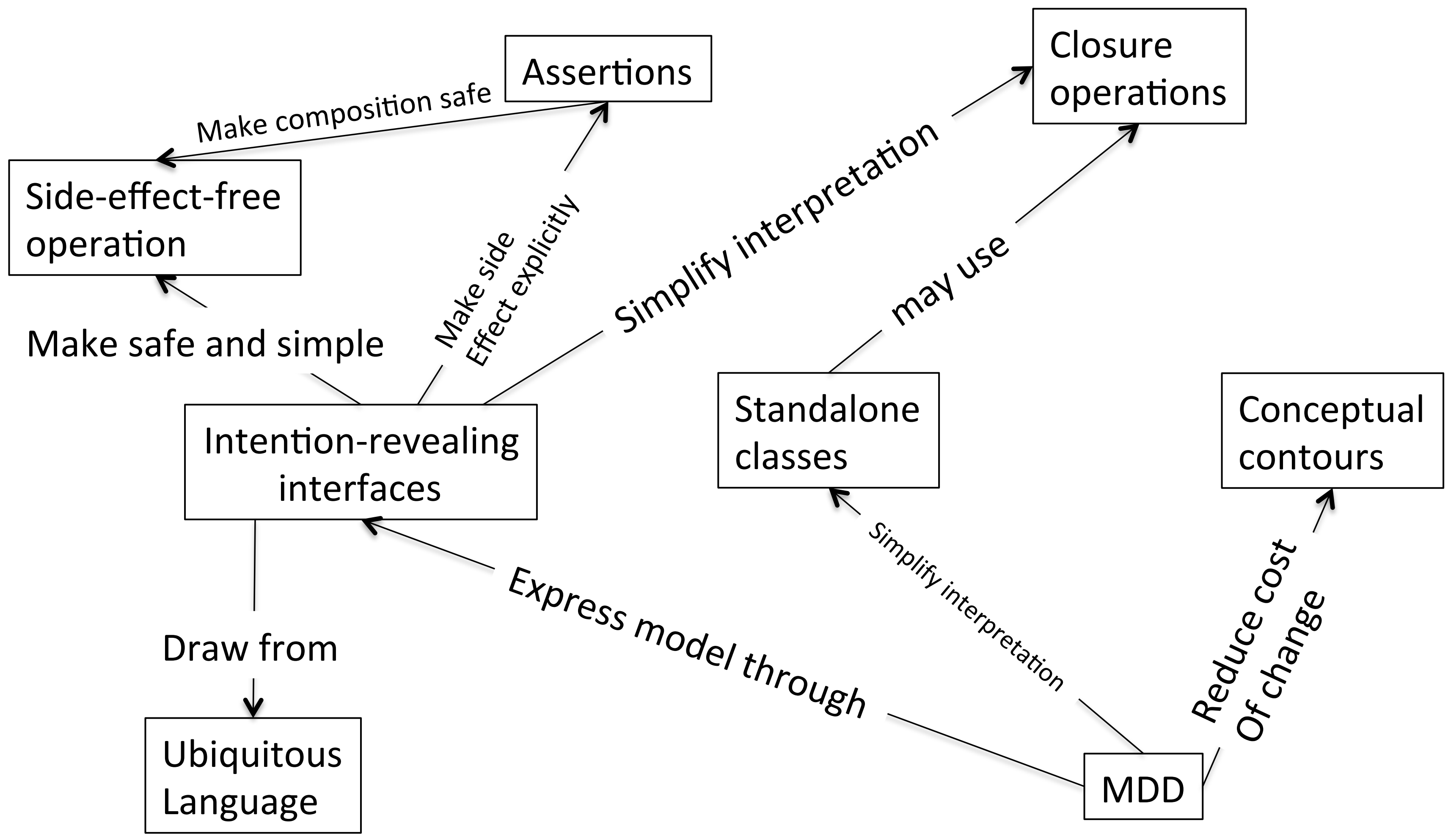
The closure of operations refer to such operations that return value is the same as its arguments. For example, addition with arguments of positive numbers is closed.

Sometimes if the implementer is used in the operations, it is essentially an argument of this operation, and hence the type of return value "should" be the same as implementer.

Closure of operations provides high cohesion and low coupling interfaces. Sometimes, even partially closed operations help!

Domain-Driven Design, Eric Evans, Page 268-270

# Supple Design Diagram



## Conclusions

- We review most important concepts and methods of domain driven design. (MDD, ubiquitous language, layered architectures, entities, value objects, services, modules, aggregates, factories, and repositories.)

- We review Eric Evan's idea about supple design, which requires intention-revealing interfaces, side-effect-free functions, assertions, conceptual contours, standalone classes, and closure of operations.

- Our main reference is *Domain-Driven Design* by Eric Evans (mainly Chapter 2, 3, 4, 5, 6, 7, and 10). This book also provides more detailed information and comprehensive review of DDD beyond we discussed. Most ideas and diagrams in this presentation are credited to this book. We also use Professor Anderson's lecture notes in Spring 2005 as another reference.