



agilepartner

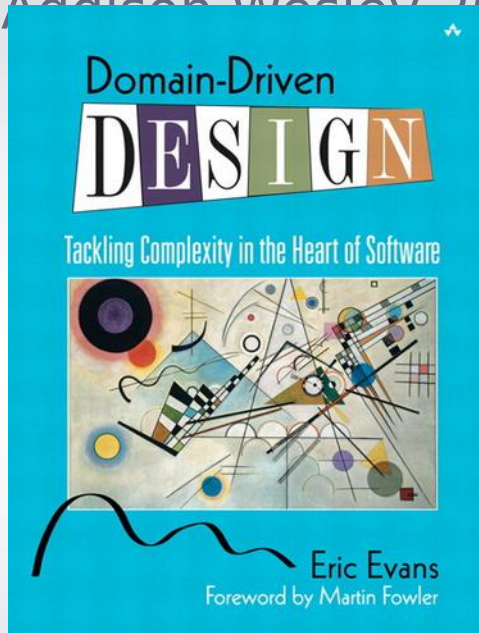
Domain Driven Design Tackling Complexity in the Heart of Software

▲ Eric Evans

- ▲ Since the early 1990s, he has worked on many projects developing **large business systems** with **objects** with many different approaches and many different outcomes.
- ▲ Eric now leads Domain Language, a consulting group which coaches and trains teams applying **domain-driven design**, helping them to make their development work more productive and more valuable to their business.

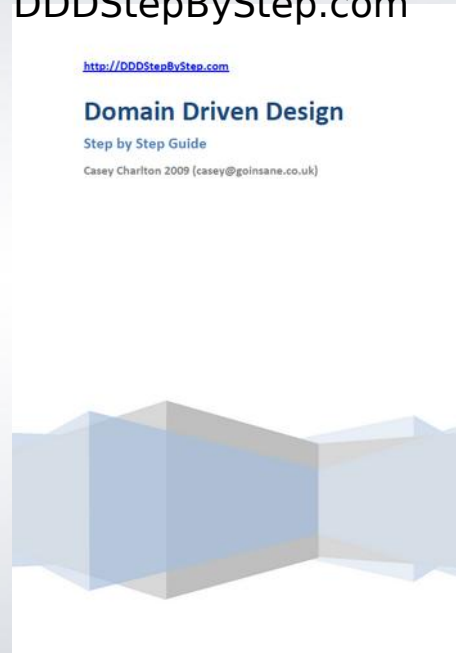


"Domain-Driven Design:
Tackling Complexity in
Software",
Addison-Wesley 2004.



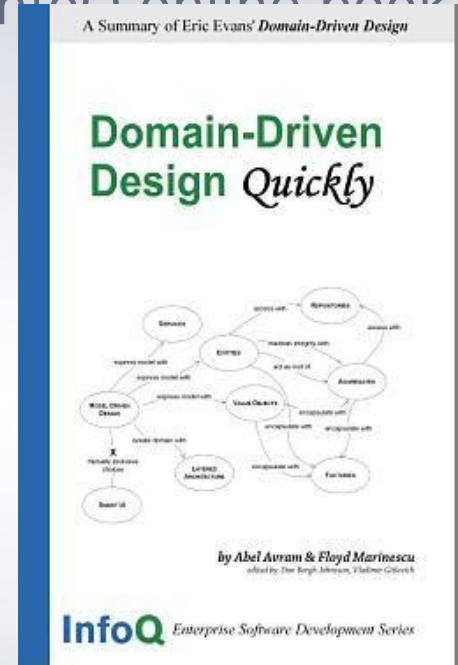
[http://
domaindrivendesign.org/books](http://domaindrivendesign.org/books)

[http://
DDDStepByStep.com](http://DDDStepByStep.com)



"Domain-Driven Design:
Step by Step Guide",
Casey Charlton.

"Domain-Driven Design
Quickly",
InfoQ online book



[http://www.infoq.com/
minibooks/domain-driven-
design-quickly](http://www.infoq.com/minibooks/domain-driven-design-quickly)

“There is technically nothing new or revolutionary in DDD, there is only a guide to a **better way of thinking.**”

Casey Charlton

- ▴ Domain Driven Design
 - ▴ certainly includes **software patterns**,
 - ▴ and certainly includes **coding guidance**,
- ▴ but what the book is really about is
 - ▴ a different way of **thinking**,
 - ▴ a different way of **solving problems**,
 - ▴ a different way of **communicating**.

- ▶ From Domain to Model
- ▶ Building the domain knowledge
- ▶ Model-Driven Design
- ▶ Building blocks of Model-Driven Design
- ▶ Refactoring toward deeper insight
- ▶ Strategic Design
- ▶ Best Practices
- ▶ Distributed Domain-Driven Design

Where should you start ?

FROM DOMAIN TO MODEL

- ▶ **Domain** (aka Problem Domain)
 - ▶ The **area of expertise** or application that needs to be examined to solve a problem.
- ▶ **Model** (aka Domain Model)
 - ▶ A **conceptual** model of the domain of interest which describes the **entities, attributes, relationships**, and **constraints** that govern the integrity of the model elements comprising that problem domain.
- ▶ **Design** (aka Software Design)
 - ▶ The **solution** to the problem at hand **envisioned** by the **developers/designers**.
 - ▶ **Multiple designs** possible for the same problem.

- ▶ In order to create good software, you have to **know what that software is all about**.
- ▶ The **domain** is where we should always **start**.
 - ▶ Complex domain designs should be **based on a model**.
 - ▶ The **primary focus** should be on the **domain and domain logic** (as opposed to the particular technology used to implement the system).

- ▶ How can we make the software fit harmoniously with the domain?
 - ▶ The best way to do it is to **make software a reflection of the domain.**
 - ▶ Software needs to **incorporate the core concepts** and elements of the domain, and to precisely **realize the relationships** between them.
 - ▶ Software has to **model the domain.**

- ▶ A domain is something of this world.
 - ▶ It cannot just be taken and poured over the keyboard into the computer to become code.
 - ▶ We need to **create an abstraction of the domain**.
- ▶ The model is our **internal representation** of the target domain
 - ▶ We need to **organize information**, to systematize it, to **divide it up in smaller pieces**, to **group** those pieces **into logical modules**, and take **one at a time** and deal with it.
 - ▶ We need it in order to be able to **deal with complexity**.

It's
not

just a diagram

just the
knowledge in a
domain expert's
head

It is

the idea that the
diagram is
intended to
convey

a rigorously
organized and
selective
abstraction of
that knowledge

- ▶ We need to **communicate** this model with domain experts, with fellow designers, and with developers.
- ▶ We need to do it well, precisely, completely, and **without ambiguity**.



We can and we should create a
language to communicate specific
issues about the domain : **Ubiquitous
Language**

Database

- ▶ **Persistence Ignorance (PI)** is a very important aspect of DDD
- ▶ Start from the premise “there is no database”
- ▶ Try to evolve the Domain **without concern to the database** you may or may not use in the end.

How to get acquainted with the domain ?

BUILDING THE DOMAIN KNOWLEDGE

- ▶ **Meet** with the business experts.
- ▶ **Talk** to the right person
- ▶ **Include** as many **profiles** as you need
 - ▶ Domain experts, Business analysts, Software architects, Developers, Testers, Etc.
- ▶ Work **incrementally**
 - ▶ Repeat meetings as many times and as often as necessary

- ▶ Binding the **model and the implementation**
 - ▶ Prototypes / spikes
- ▶ Cultivating a **language based on the model**
 - ▶ Everyone should **understand** what is talked about without ambiguity and without need of translation
- ▶ Developing a **knowledge rich model**
 - ▶ **Not** just a **data schema**
 - ▶ Objects have **behavior** and enforce **rules**

- ▶ **Distilling** the model
 - ▶ Keep **usefull concepts**
 - ▶ Drop concepts that **do not fit** or are **unneeded**
- ▶ **Brainstorming** and **experimenting**
 - ▶ Get together with the **right** people
 - ▶ Sketch, draw, write on a **white board**
 - ▶ Be curious and ask the **right questions**
 - ▶ Don't be ashamed to **admit your ignorance** : you are here to **learn**

- ▶ Team members and domain experts should **share a common language**.
 - ▶ Team members must **understand and use** the terms used by domain experts.
 - ▶ Domain experts should **get familiar** with the technical terminology that team members use.
- ▶ Use the **model as the backbone** of your language.
 - ▶ Use the **same language everywhere**, in diagrams, writing, speech, and code.
 - ▶ **Do not hesitate to change** a concept name if inappropriate.
 - ▶ Use **non-english language** if the domain experts do not use english, but **only for domain concepts**.

- ▶ Identify **key concepts** used in the domain
- ▶ Create classes from key concepts
i.e. : Insurance policy → Policy class
- ▶ Domain experts should **object to terms or structures** that are **awkward or inadequate** to convey domain understanding.
- ▶ Developers should **watch for ambiguity or inconsistency** that will tend to appear in design.

- ▶ Best tool : **white board** and photo camera
 - ▶ Draw only what you need
 - ▶ Be concise
- ▶ You can use UML, but...
 - ▶ Beware of giant UML diagrams
 - ▶ UML cannot convey two important aspects of a model: the **meaning** of the concepts it represents and what the **objects are supposed to do**.
- ▶ **Communicate** through code

How to transform the model into code ?

MODEL-DRIVEN DESIGN

- ▶ The better approach is to **closely relate domain modeling and design**.
 - ▶ Design a **portion** of the software system to reflect the domain model in a very literal way.
 - ▶ **Revisit** the model and modify it to be implemented **more naturally** in software.
 - ▶ Use **model terminology** in the design and the basic assignments of responsibilities.

- ▶ Developers should be **included in the modeling process**
 - ▶ So that the **design process is straightforward**.
 - ▶ So they can **give feedback** and make sure the model **can be implemented** in software.
- ▶ Those who write the code should **know the model very well**, and should feel **responsible for its integrity**.
- ▶ Support a **fluent** Ubiquitous Language

▲ Lack of intent

- ▲ You build a system but you do not find out what the intention of the user is.
 - The domain model is empty from business logic
 - There is only four verbs, CRUD which doesn't tell you anything about what the user wanted to do

▲ Anemic Domain Model

- ▲ A model that looks and smell like a real domain model, but when you are looking for behavior you can't find it
 - You have domain services that become transaction scripts on top of your domain model
 - It's not an anti-pattern but it's not DDD either

▲ DDD-Lite

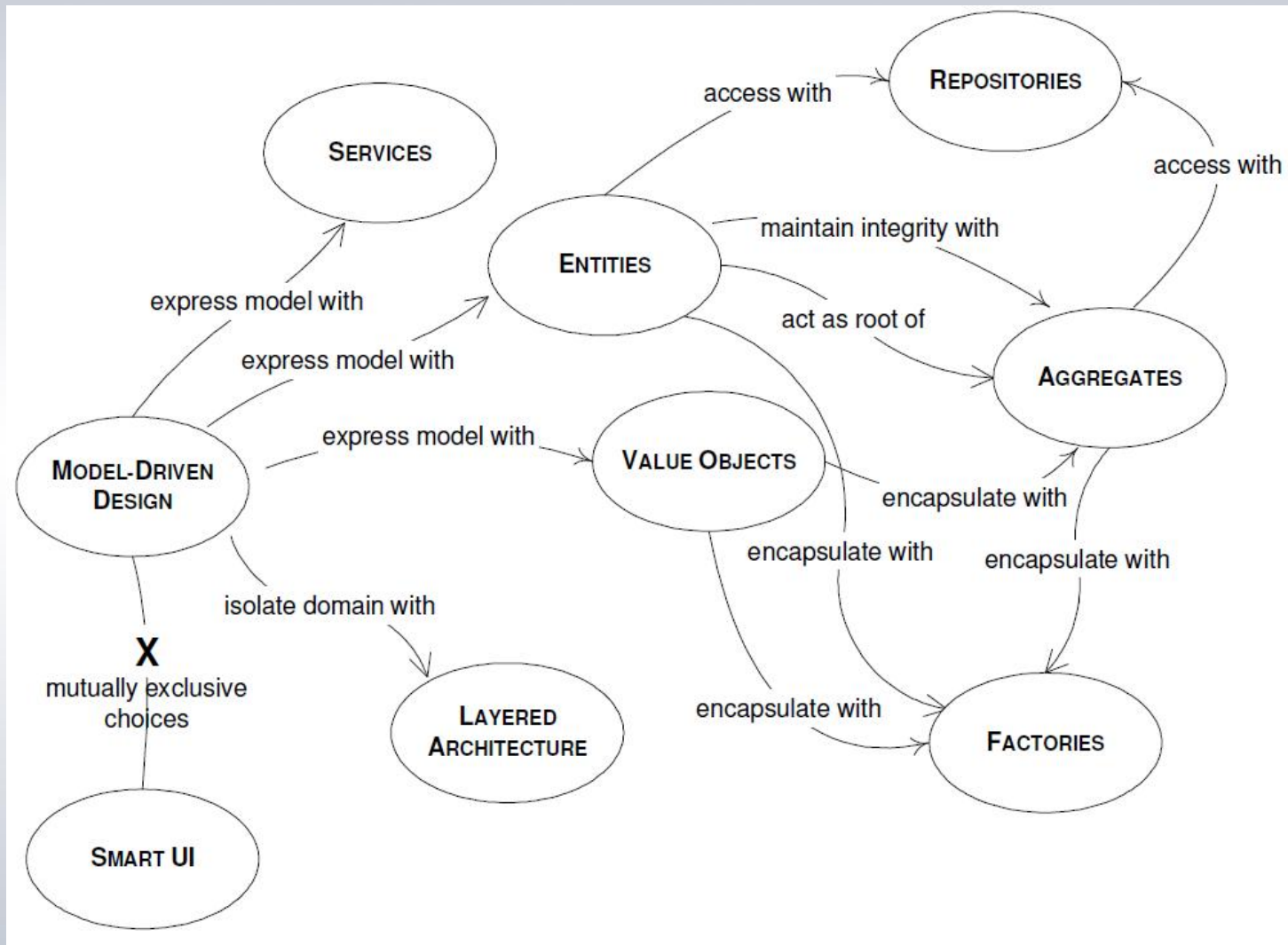
- ▲ Using DDD as a pattern language, you do not define context you just implement patterns
 - DDD-Lite will give you all of the overhead and none of the benefits
 - The domain model is only a representation of the ubiquitous language. So if the language is not there, the competitive advantage will not surface.

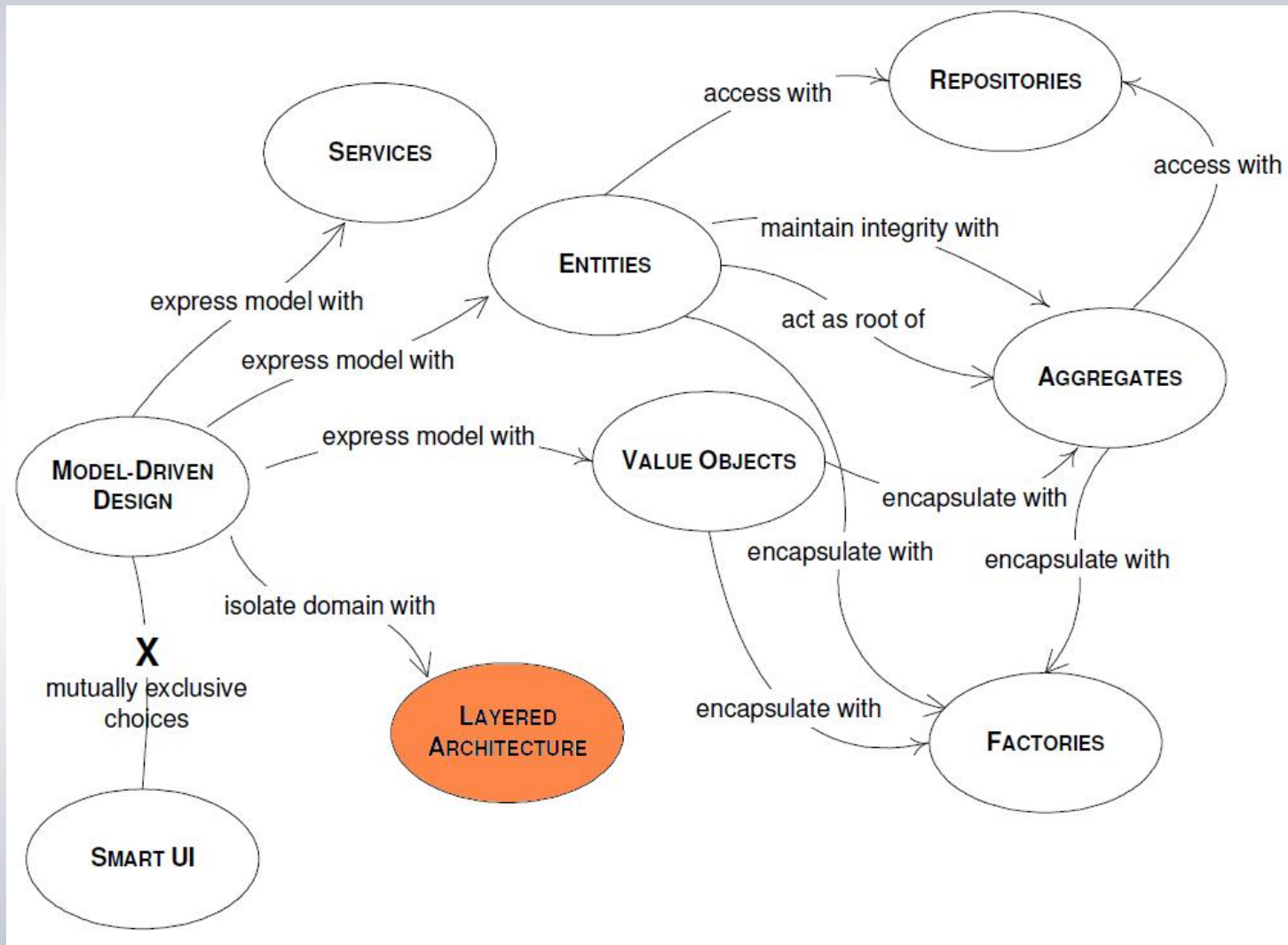
- ▲ Lack of adequation between model and design
 - ▲ That model is of little value
 - ▲ The correctness of the software is suspect
- ▲ Lack of isolation
 - ▲ Without isolation, there will be abstraction leakage left and right
 - i.e. Unit of Work is really a infrastructure concern why would you bring that into the domain ? If you have your aggregate boundaries right, you don't need a unit of work.
- ▲ Ubiquitous what ?
 - ▲ The ubiquitous language is only ubiquitous in a context
 - If you aren't using bounded context, you will fail with defining the language
 - The way to find contexts is to find places where words have different meaning.
 - We need to break things apart and focus on our context boundaries

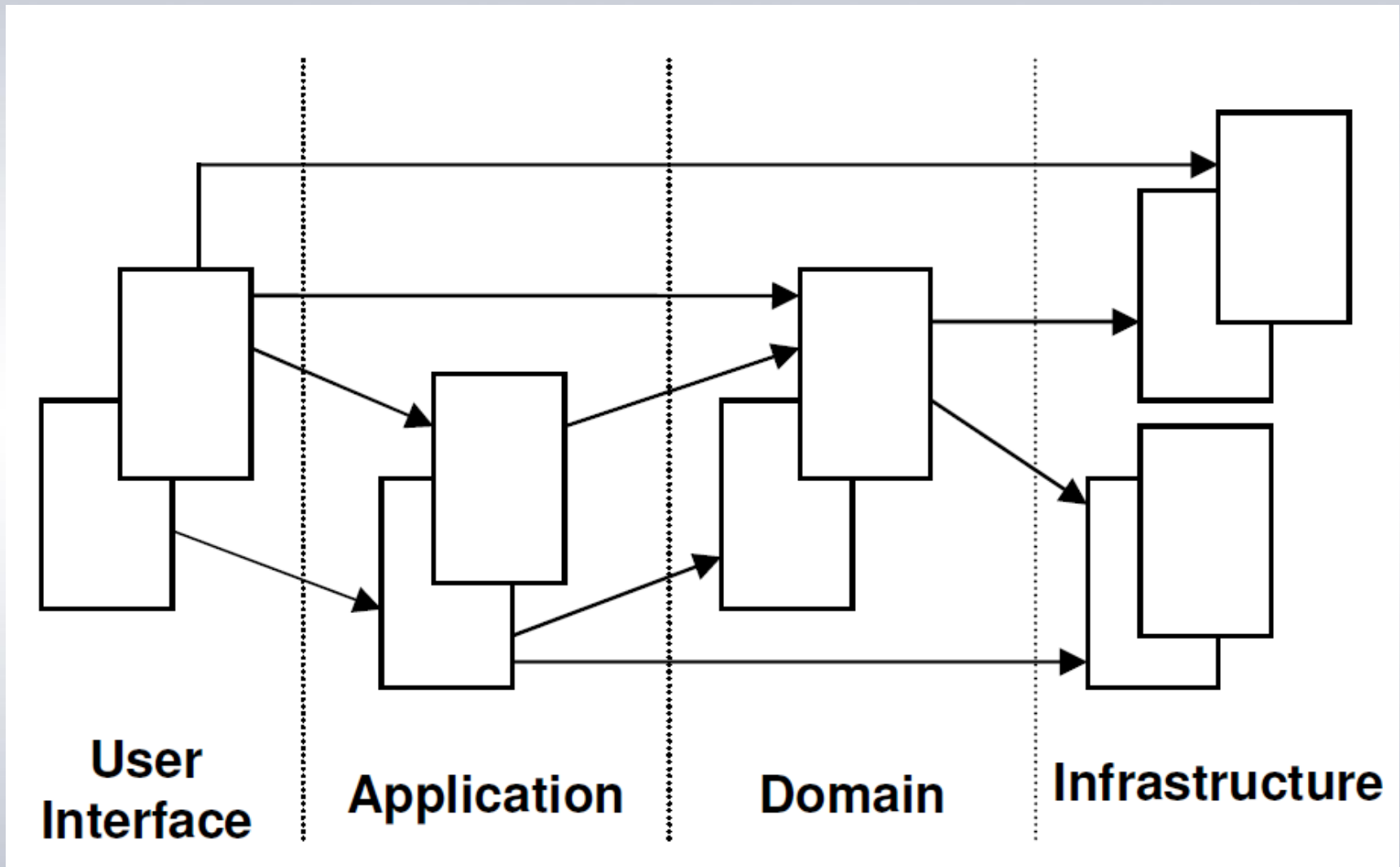
- ▲ Lack of refinement
 - ▲ The team learns more about the domain, but do not go back and update the model with the new knowledge
 - Since the domain model is a tool to analyze the domain and solve problems, it has to be clean and reflect the level of understanding we have of the domain
- ▲ Proxy Domain Expert (Business analyst)
 - ▲ Each level of translation a little more get lost
 - ▲ What's really happening is that we have two ubiquitous languages
 - One from the BA to the real domain expert
 - One from the BA to the developers. What's the chances they are the same?
- ▲ **Complex mappings** between models and design functions
 - ▲ are **difficult to understand**
 - ▲ are **impossible to maintain** as the design changes

How to build your software ?

BUILDING BLOCKS OF MODEL-DRIVEN DESIGN







Each layer is cohesive and depends only on the layers below.

User Interface

- 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.

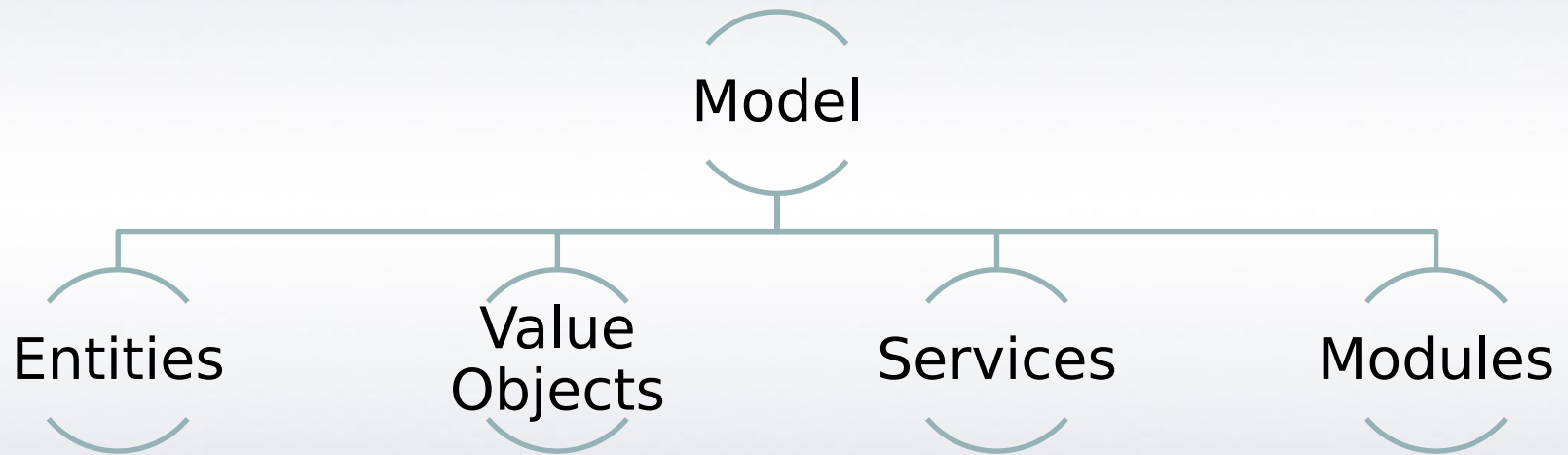
Domain Layer

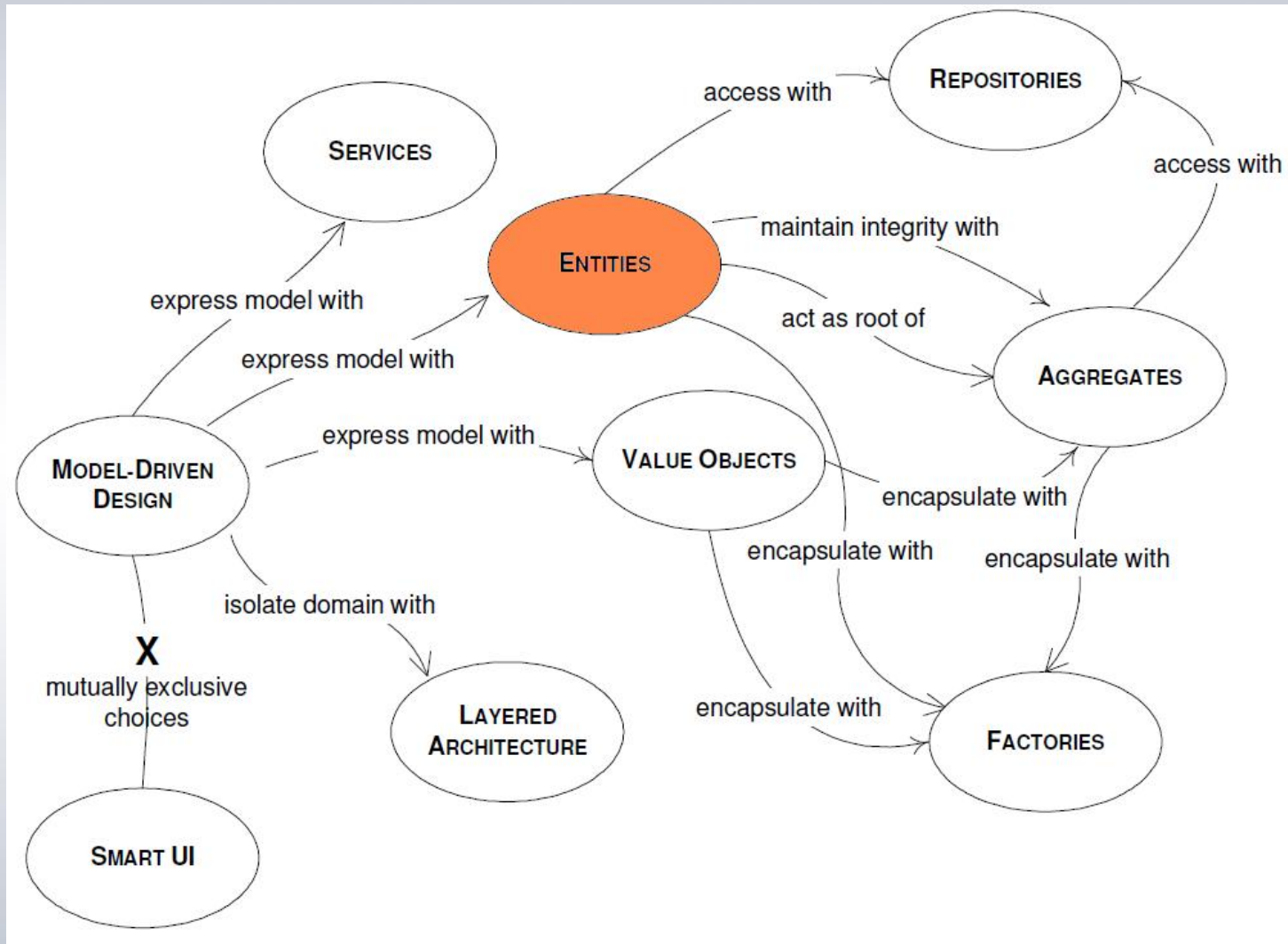
- This layer contains information about the **domain**. This is the **heart of the business software**. The **state** of business objects is held here. Persistence of the business objects and possibly their state **is delegated** to the infrastructure layer.

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.

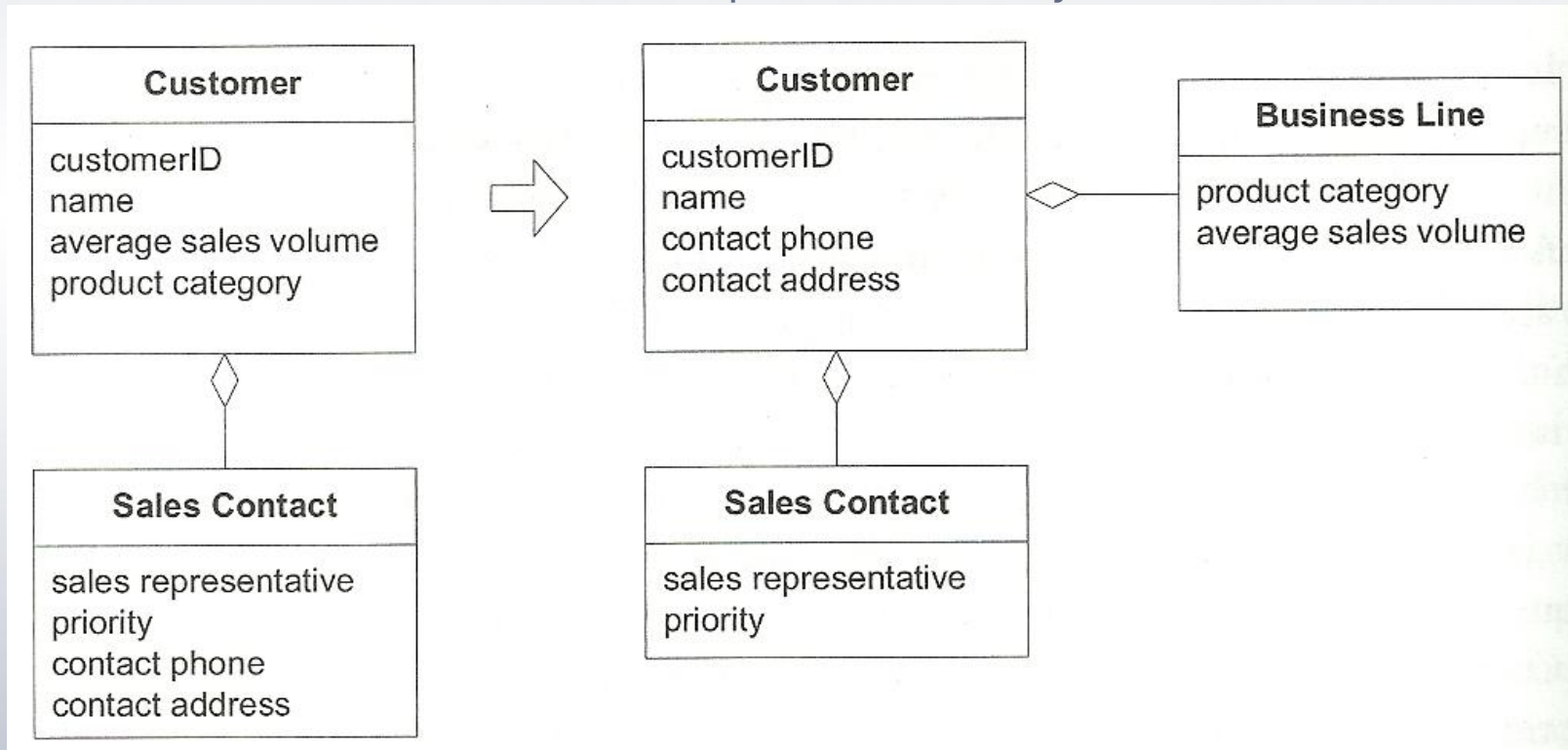
- ▲ Layers are **NOT** Tiers
 - ▲ Layers are used for **logical separation**
 - user interface logic and business logic
 - or business logic and databases,
 - ▲ but the model says nothing about deployment tiers
- ▲ We **can deploy** the same components on **client-side and server-side**
 - ▲ Think of Validation on Client Tier

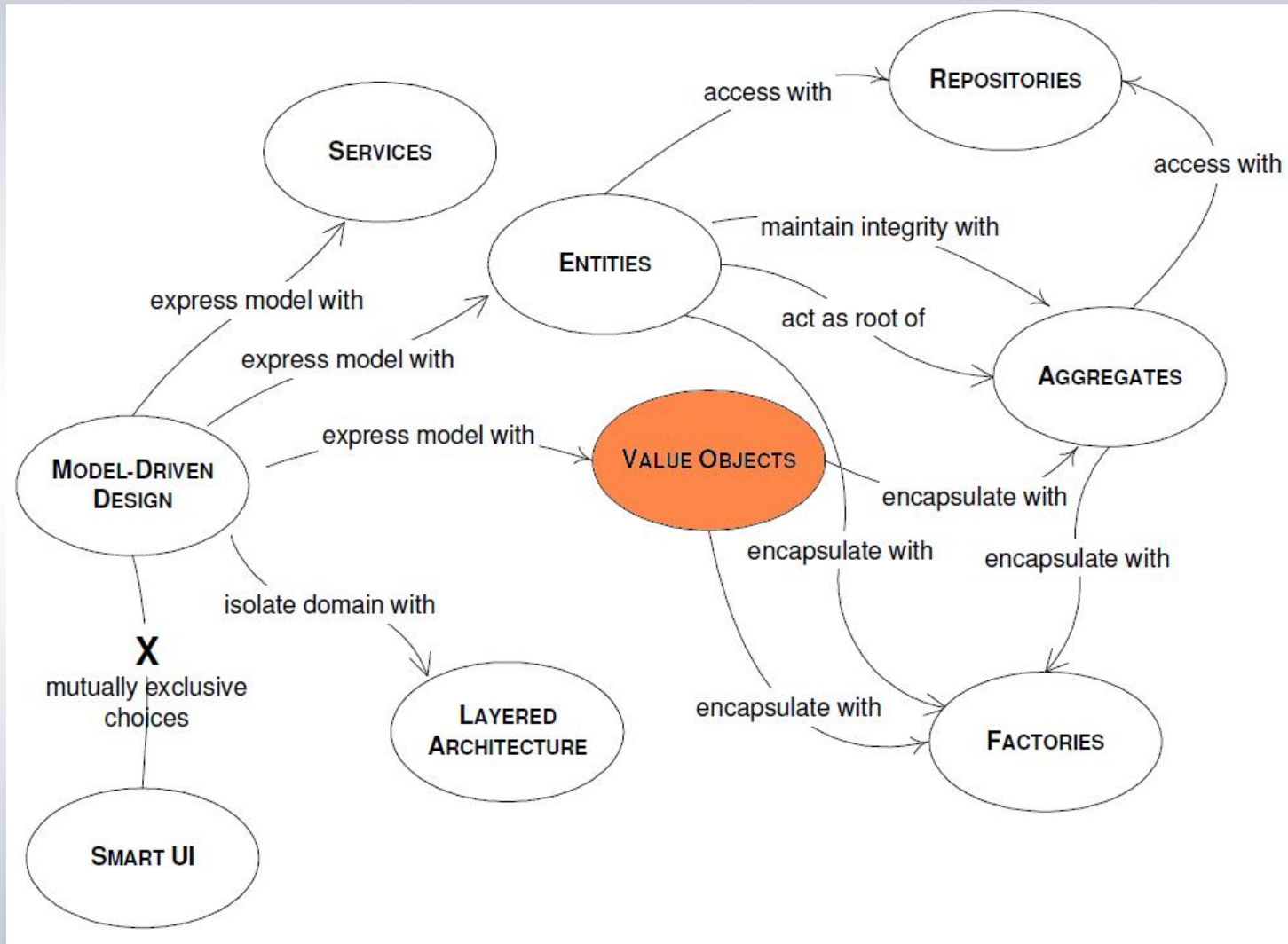




- ▶ When an object is **distinguished by its identity**, rather than its attributes.
- ▶ Keep the class definition simple and focused on **life cycle continuity** and **identity**.
- ▶ Define a **means of distinguishing** each object regardless of its form or history.
 - ▶ This means of identification
 - may come **from the outside**,
 - or it may be an arbitrary identifier **created by and for the system**,
 - but it must correspond to the **identity distinctions** in the model.

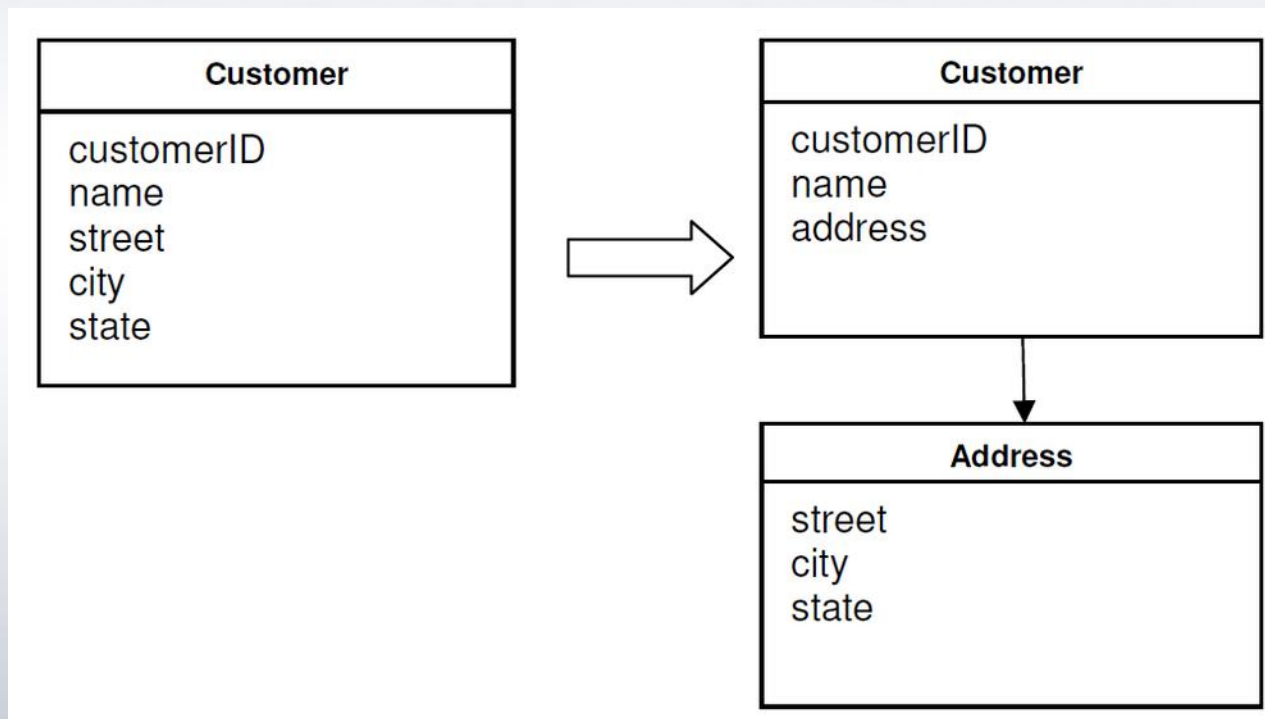
- Attributes associated with identity stay with the Entity
 - The *customerID* is the **one and only identifier** of the *Customer* Entity, but the *phone number* and *address* would often be used to **find or match** a *Customer*.
 - The *name* does not define a person's identity, but it is often used as

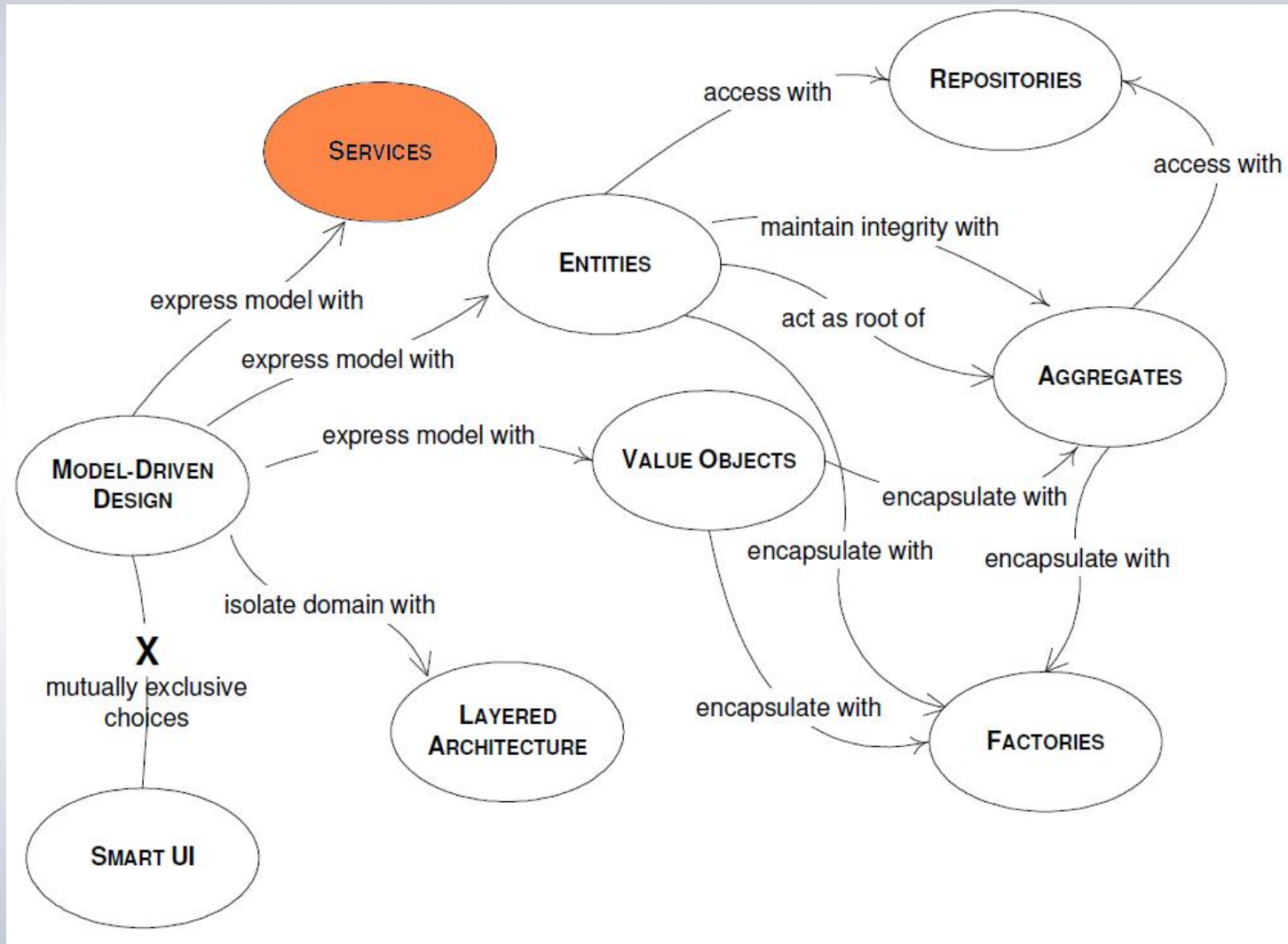




- ▶ You care **only about the attributes** of an element of the model.
- ▶ Make it **express the meaning of the attributes** it conveys and give it related functionality.
- ▶ Treat the Value Object as **immutable**.
- ▶ **Don't give it any identity** and avoid the design complexities necessary to maintain Entities.

- ▶ A Value Object can **give information** about an Entity. It should be conceptually whole.
 - ▶ We can use them to **simplify the design** or **optimize performance**.
 - ▶ This involves making choices about copying, sharing, and immutability.





- ▶ A significant **process** or **transformation** in the domain that is **not a natural responsibility** of an **Entity** or **Value Object**.
- ▶ Add an operation to the model as a **standalone interface** declared as a Service.
- ▶ Define the interface in terms of the language of the model and make sure the **operation name is part of the Ubiquitous Language**.
- ▶ Make the Service **stateless**.

Application *Funds Transfer App Service*

- Digests inputs (such as XML requests).
- Sends message to domain service for fulfillment.
- Listens for confirmation.
- Decides to send notification using infrastructure service

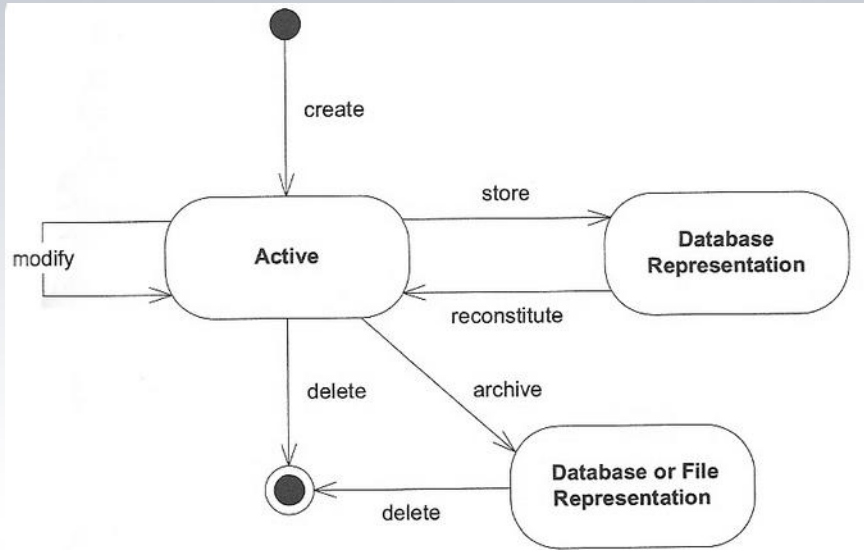
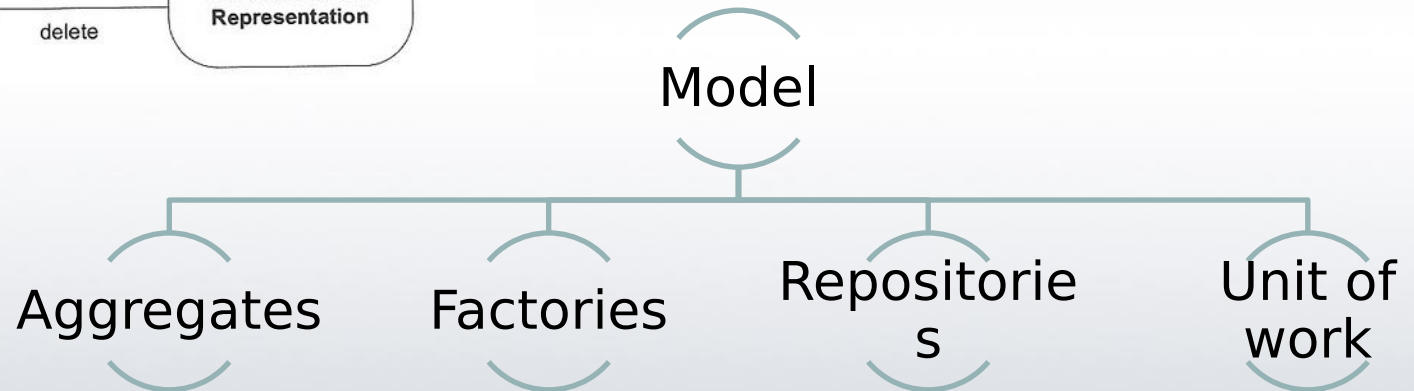
Domain *Funds Transfer Domain Service*

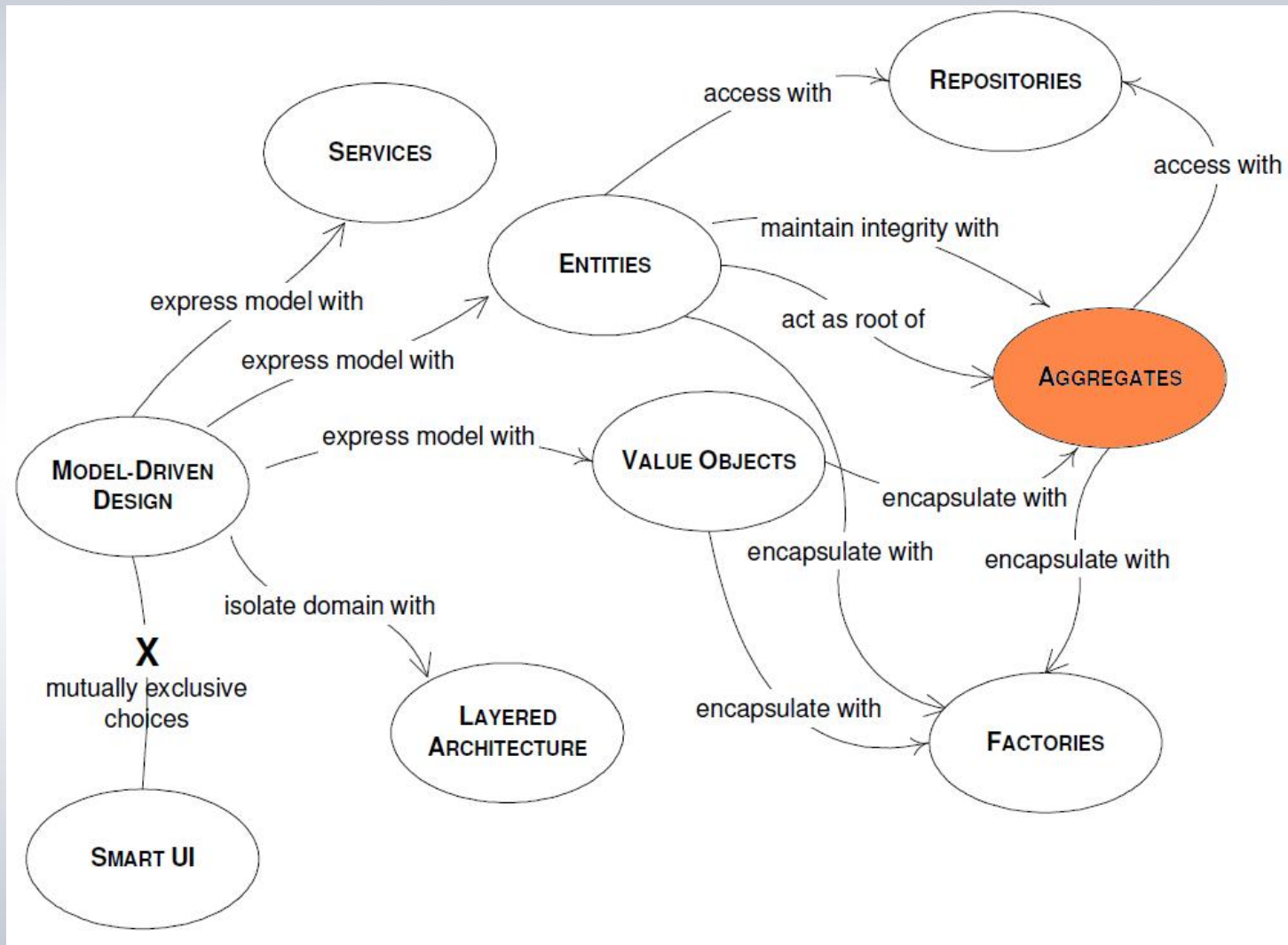
- Interacts with necessary Account and Ledger objects, making appropriate debits and credits.
- Supplies confirmation of result (transfer allowed or not,

Infrastructure *Send Notification Service*

- *Sends e-mails, letters, and other communications as directed by the application*

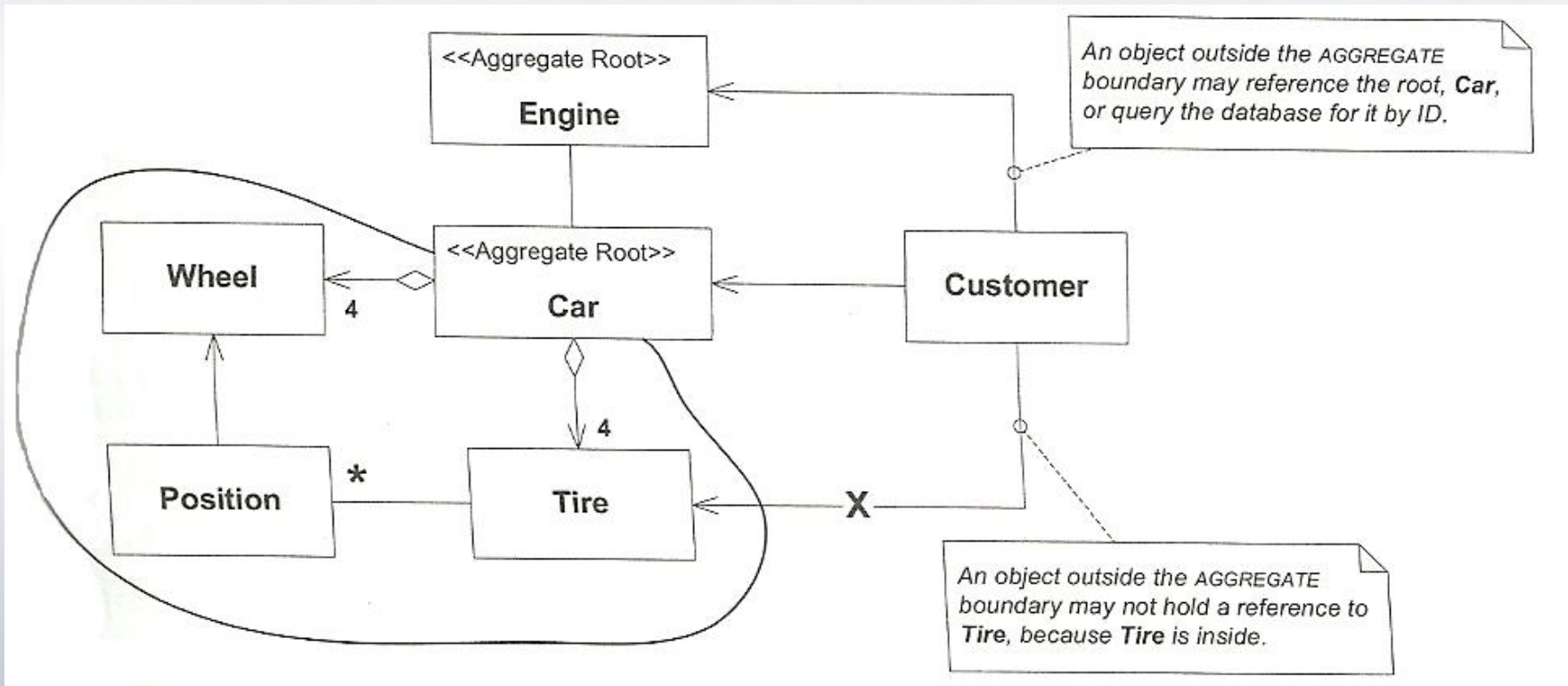
- ▶ Choose Modules that tell the story of the system and contain a **cohesive set of concepts**.
- ▶ Seek **low coupling**, in the sense of concepts that can be understood and reasoned about **independently of each other**.
- ▶ Refine the model until it partitions according to **high-level domain concepts** and the corresponding code is **decoupled** as well.
- ▶ Give the Modules **names** that become **part of the Ubiquitous Language**.
- ▶ Modules and their names should reflect **insight** into the domain.



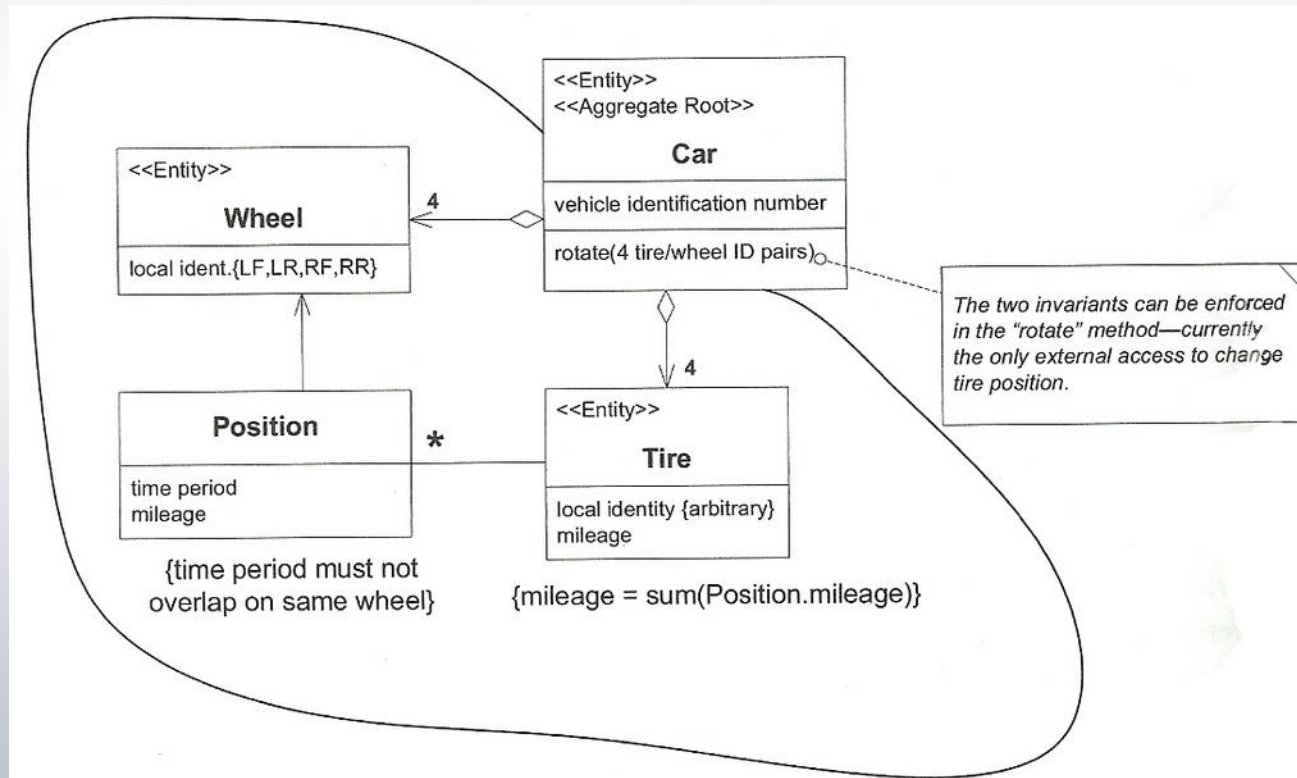


- ▶ Cluster of Entities and Value Objects
 - ▶ define **boundaries** around each.
- ▶ Aggregate root
 - ▶ **one** Entity is **root** of the Aggregate
 - ▶ **controls all access** to the objects **inside the boundary**
- ▶ References
 - ▶ External objects hold **references to the root only**.
 - ▶ **Transient references** to **internal members** can be passed out for use within a **single operation only**.
- ▶ Change of state
 - ▶ The root **cannot be blindsided** by any change because it controls all access to its internals
 - ▶ The root **enforces all invariants** for objects in the Aggregate and for the Aggregate as a whole.

- ▶ The Car is an aggregate root
- ▶ In some applications, the engine might be root of its own Aggregate

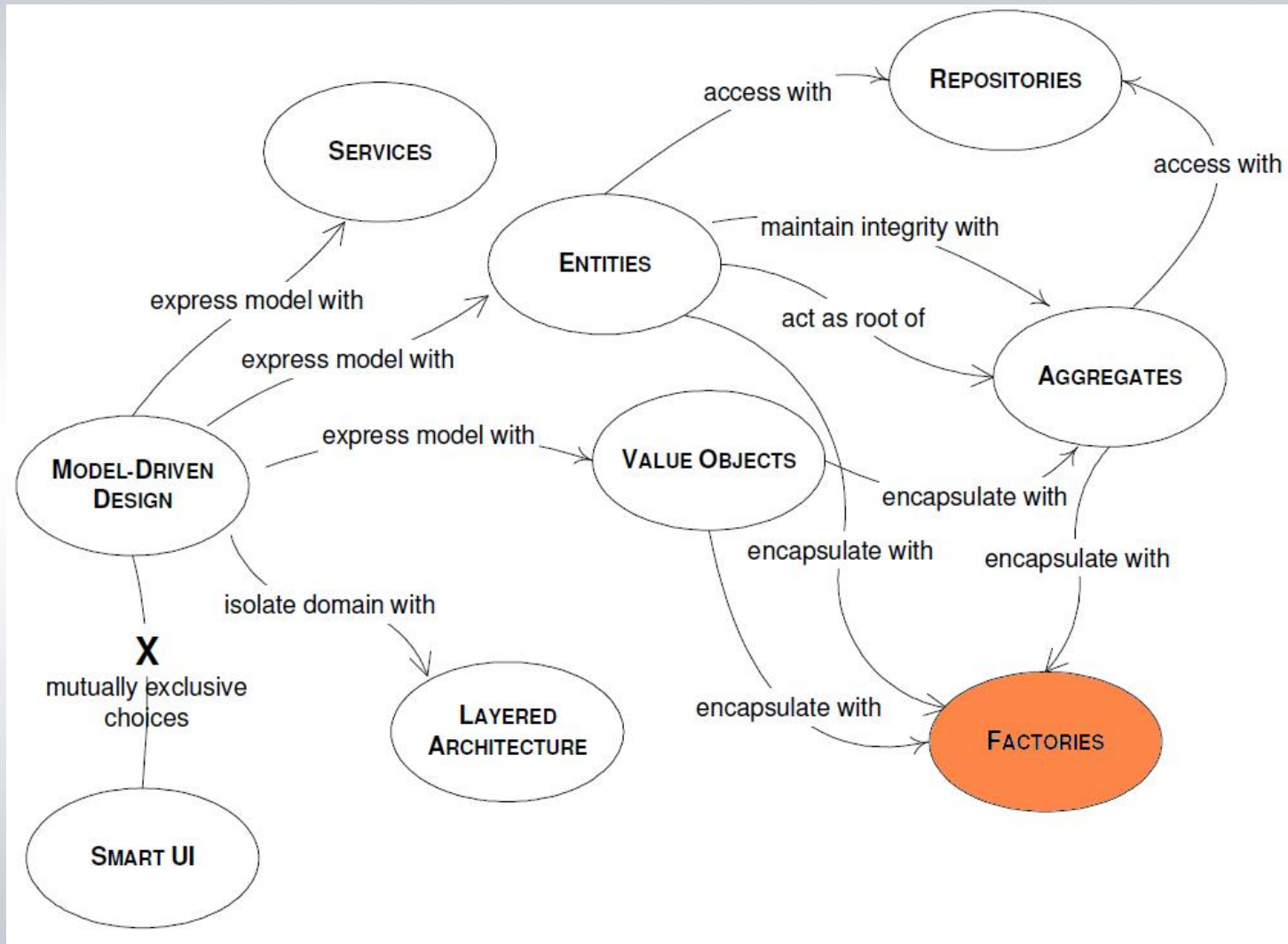


- **Invariants** are **consistency rules** that must be maintained whenever data changes
- **Invariants** involve **relationships** between the Aggregate members



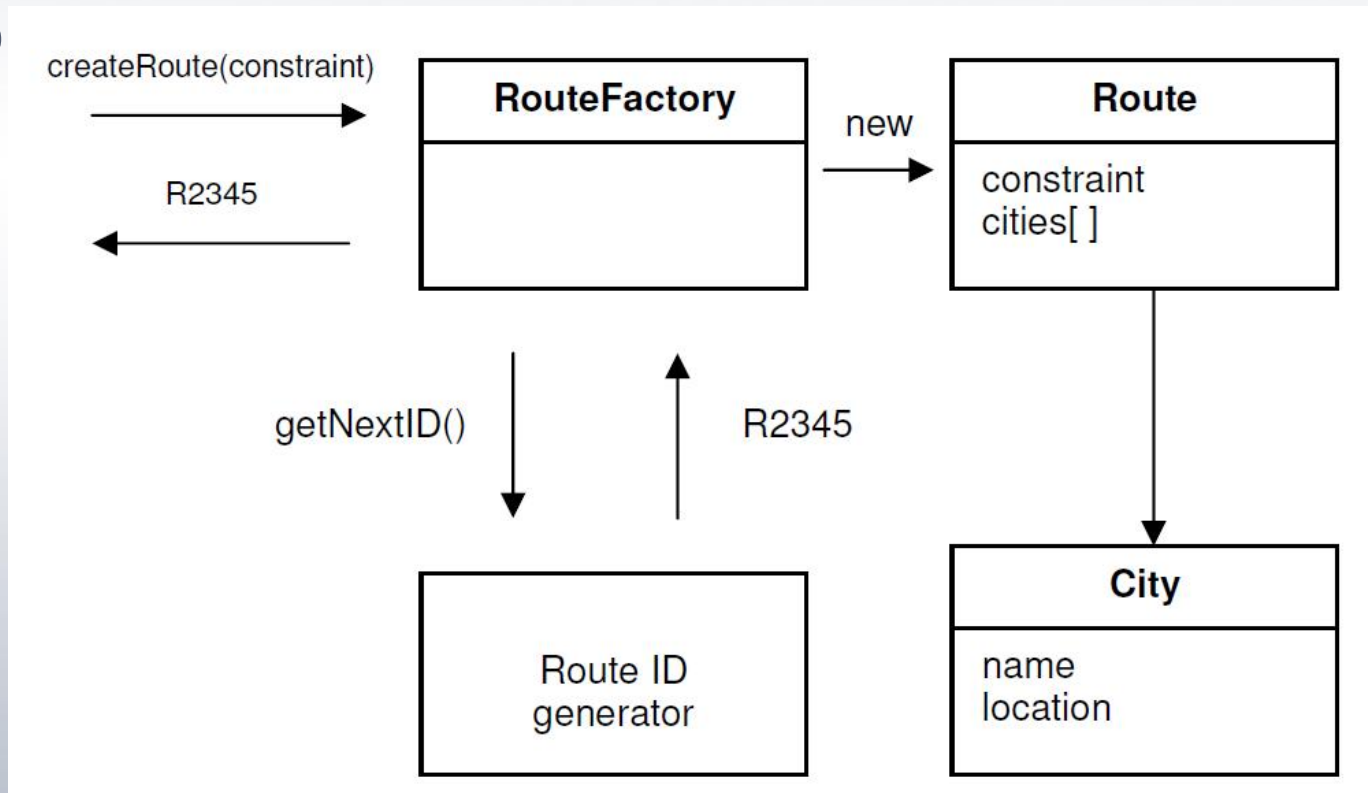
- ▶ The Aggregate root
 - ▶ has **global identity**
 - ▶ is responsible for **checking the invariants**
- ▶ Entities inside the boundary
 - ▶ have **local identity**
 - ▶ are unique only within the Aggregate
- ▶ **Nothing outside** the Aggregate boundary
 - ▶ can hold **a reference to anything inside**
 - ▶ except the root
- ▶ Only Aggregate roots
 - ▶ can be **obtained with a query**,
 - ▶ other objects must be found by **traversal of associations**

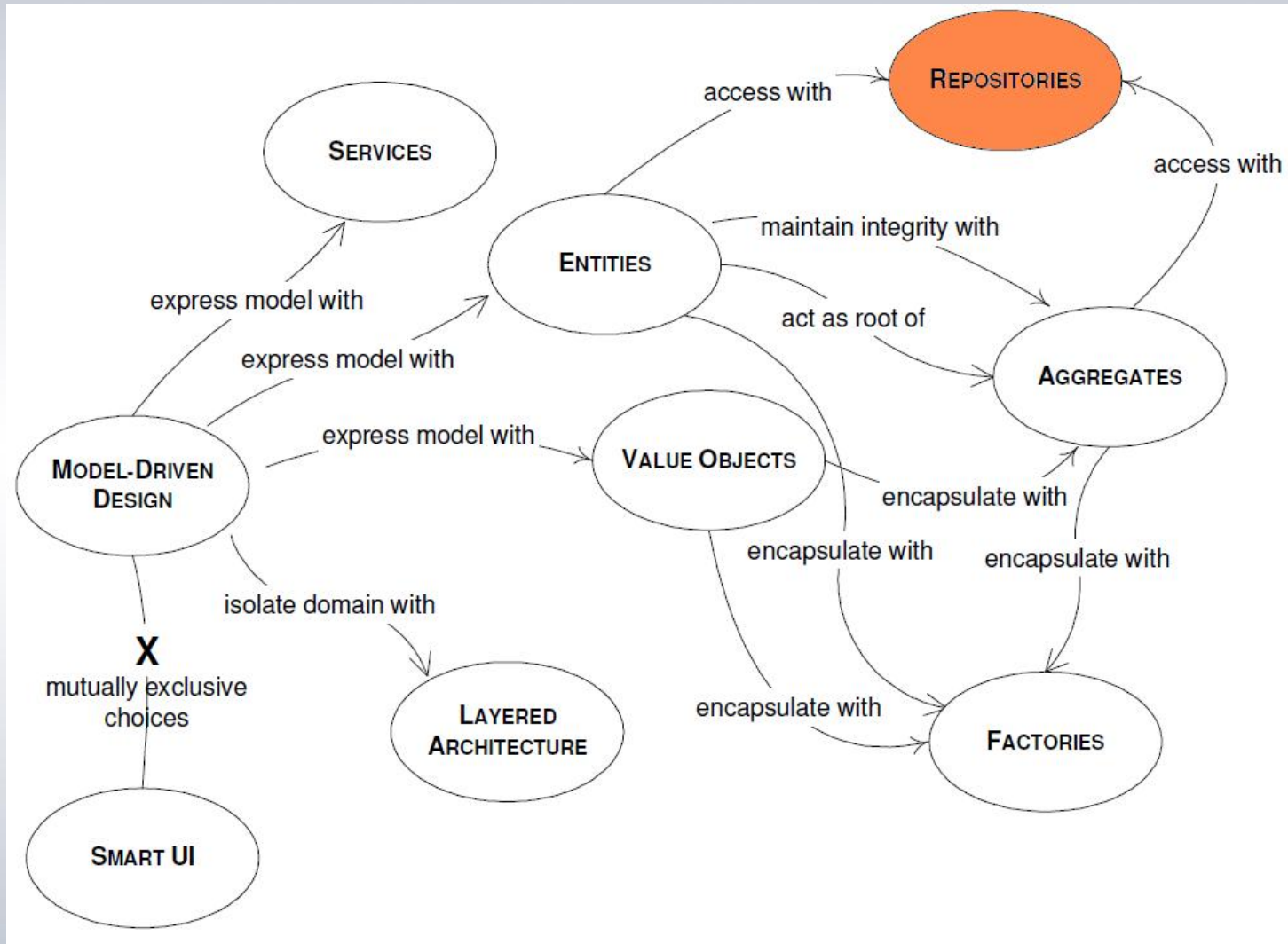
- ▶ **Objects within** the Aggregate
 - ▶ can hold references to **other Aggregate roots**.
- ▶ A **delete operation** must
 - ▶ **remove everything** within the Aggregate boundary **at once**
 - ▶ is often an **easy way to define the limits** of the Aggregate
- ▶ **All invariants** of the whole Aggregate **must be satisfied**
 - ▶ when a **change to any object** within the Aggregate boundary **is committed**



- ▶ Shift the responsibility for **creating instances** of complex objects and Aggregates to a separate object
 - ▶ which may itself have **no responsibility in the domain model**
 - ▶ but is still part of the domain design
- ▶ Provide an interface that
 - ▶ **encapsulates all complex assembly**
 - ▶ and that **does not require** the client to **reference the concrete classes** of the objects being instantiated
- ▶ Create entire Aggregates as a piece, **enforcing their invariants**

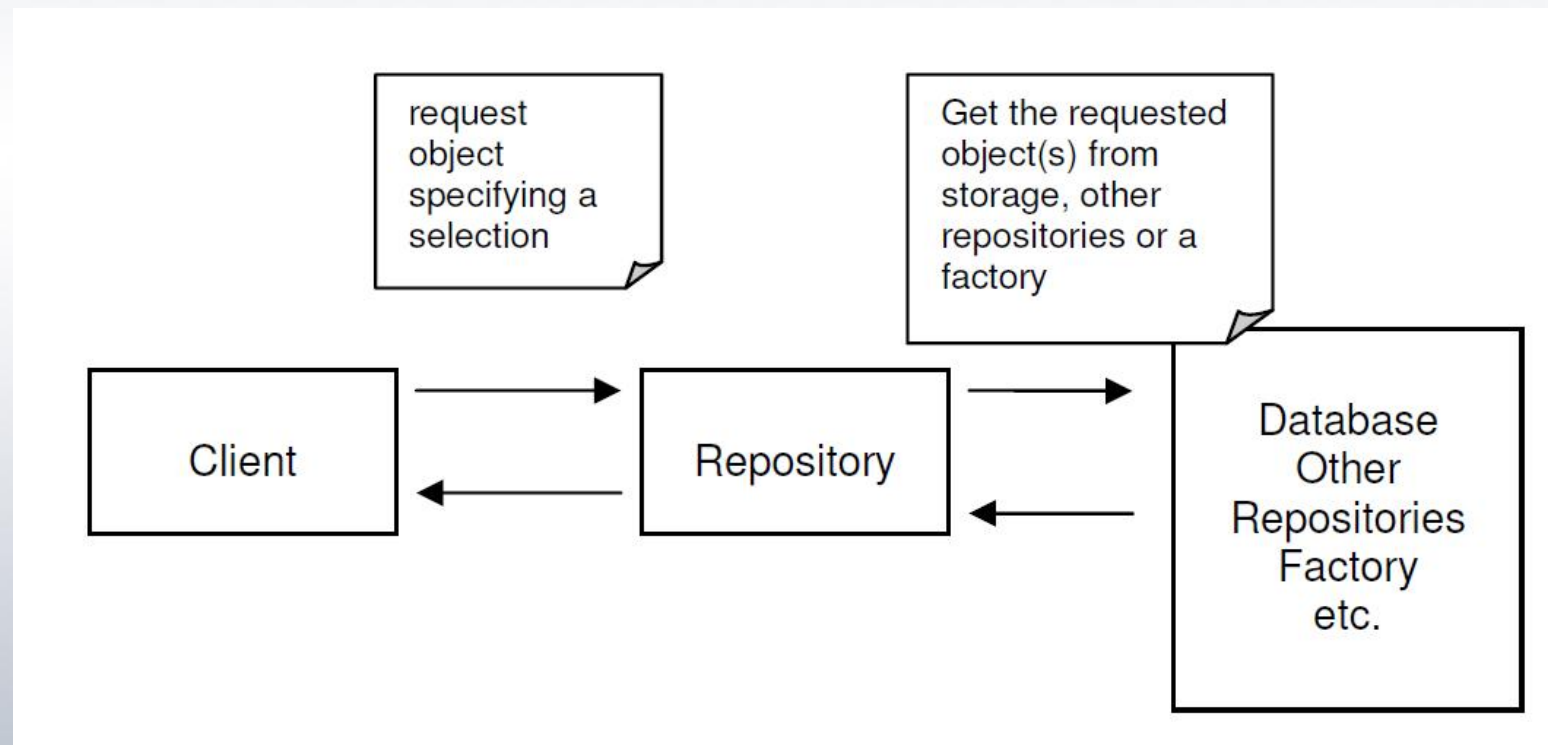
- Each creation method is **atomic** and **enforces all invariants** of the created object or Aggregate
- The Factory should be abstracted to the type desired, rather than the concrete classes created
→ Facto



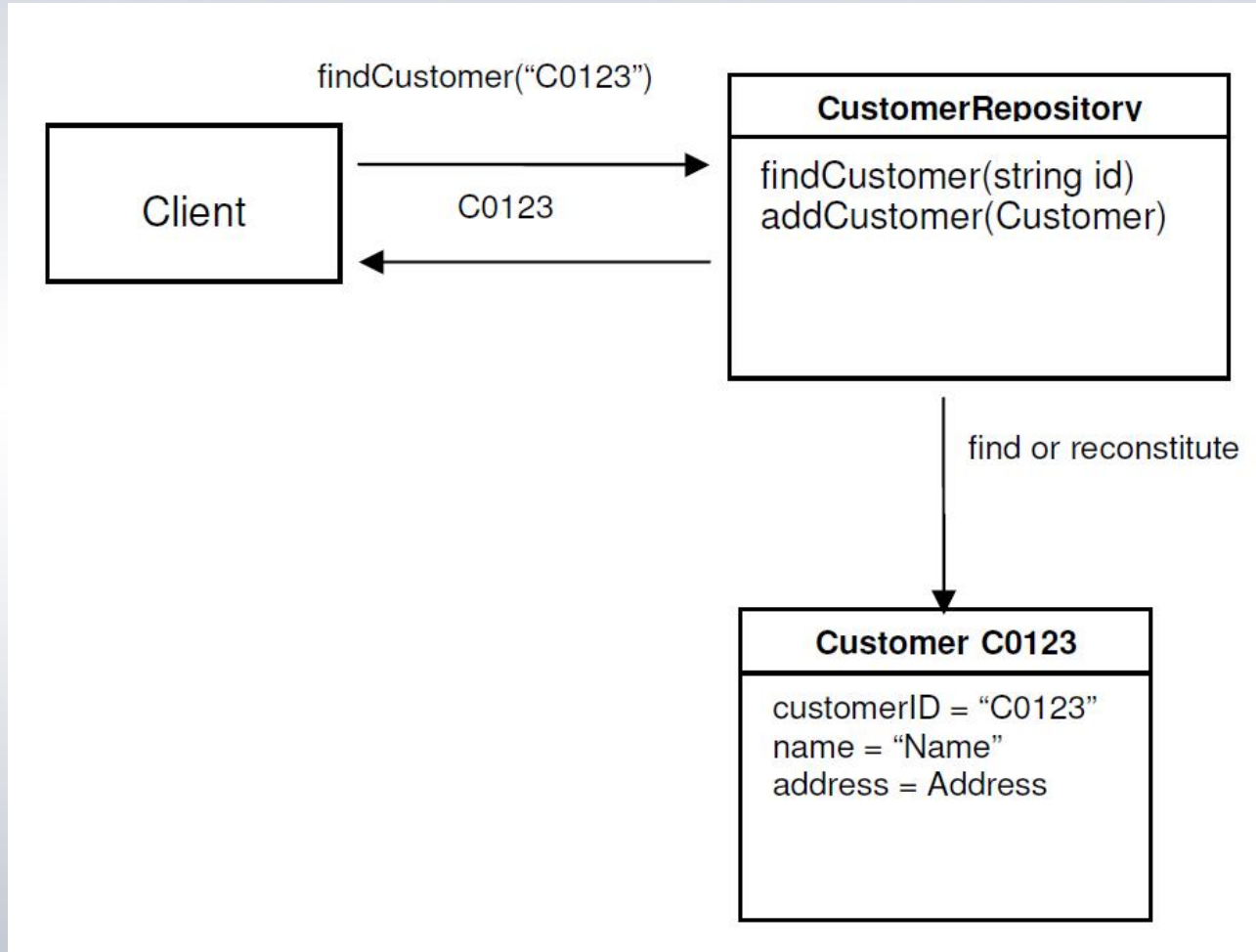


- ▶ Create an object that can provide the illusion of an **in-memory collection**.
- ▶ Set up access through a **well-known** global interface.
- ▶ Provide methods to **add and remove** objects, which will **encapsulate** the actual insertion or removal of data in the data store.
- ▶ Provide methods that **select** objects based on some **criteria** and return **fully instantiated objects** or **collections of objects** whose attribute values meet the criteria, thereby **encapsulating the actual storage and query technology**.
- ▶ Provide repositories **only for Aggregate roots** that actually **need direct access**.
- ▶ Keep the **client focused on the model**, **delegating** all object **storage and access** to the Repositories.

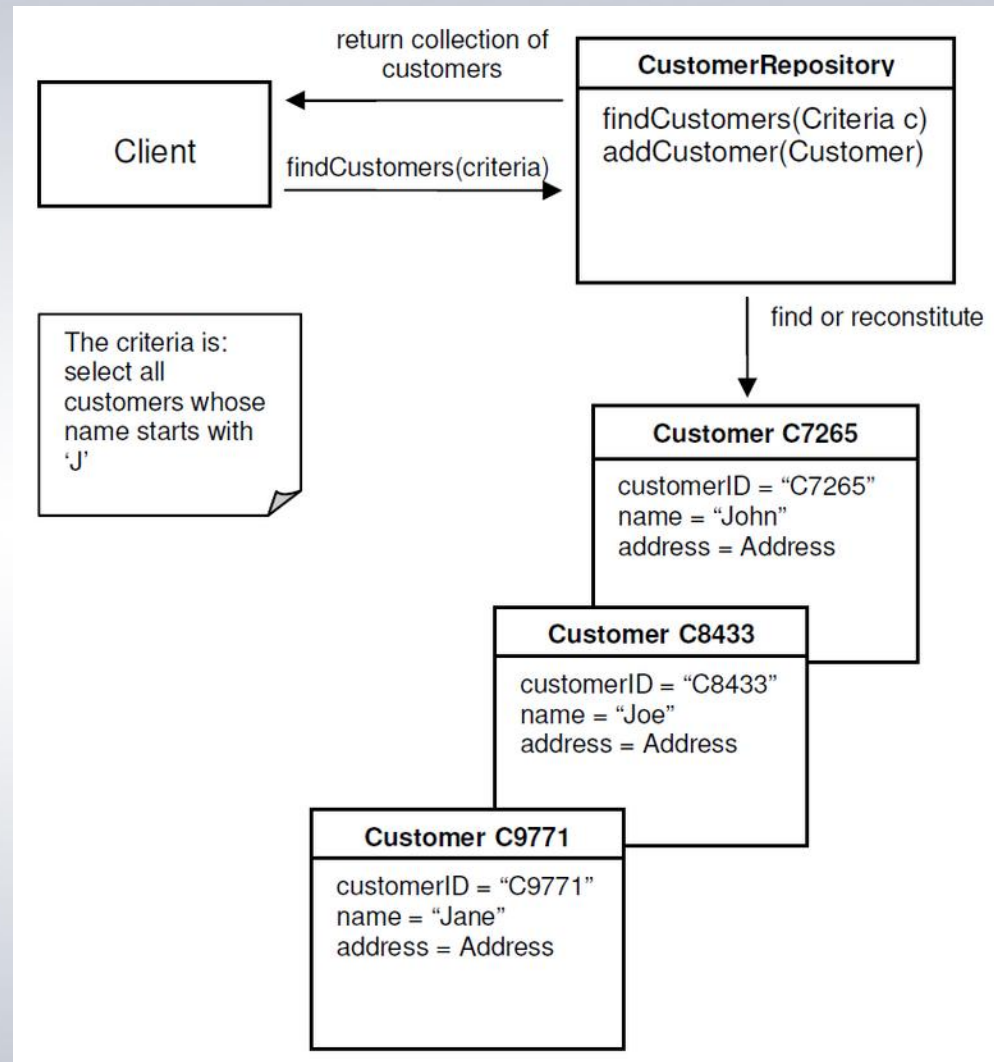
- ▶ The client/consumer requests the object to the repository.
- ▶ The repository takes care of reconstituting the object and its dependencies.



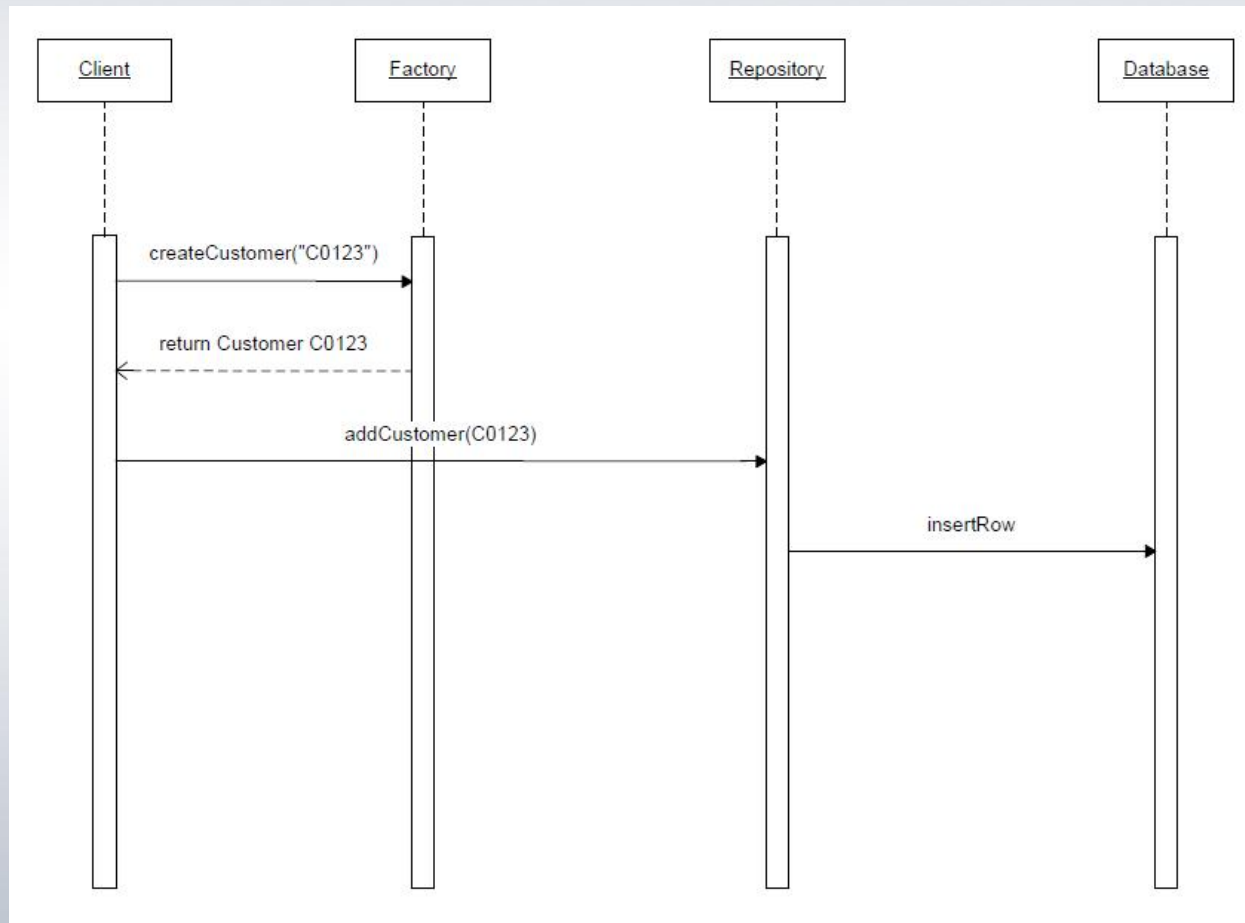
- Simple query : *Find Customer by customer id*



- Complex query :
Find customers by criteria
 - Use a **criteria** to search for multiple customers
 - Return a **collection of customers** matching the criteria



- Factories are used to create new objects
- Repositories are used to persist new objects



- ▶ Unit of Work pattern
 - ▶ Synchronize all actions performed on different repositories
 - ▶ Is usually used by the Application Layer
 - ▶ Guarantees everything is persisted at once

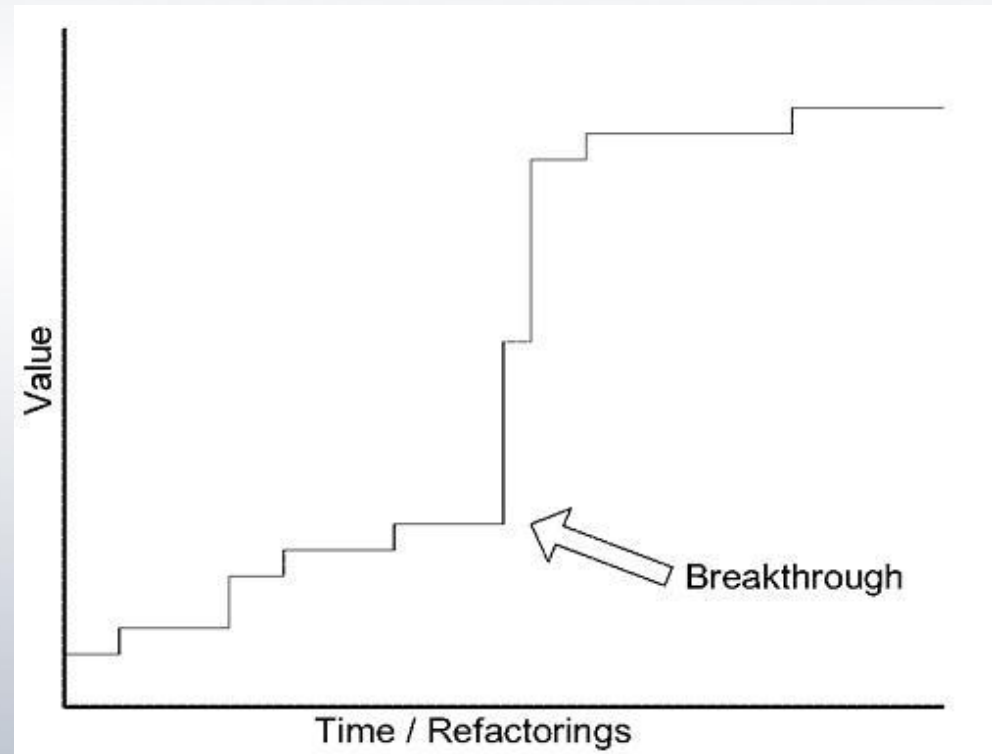
```
public interface IUnitOfWork {  
    void MarkDirty(object entity);  
    void MarkNew(object entity);  
    void MarkDeleted(object entity);  
    void Commit();  
    void Rollback();  
}
```

How to keep your software manageable ?

REFACTORING TOWARD DEEPER INSIGHT

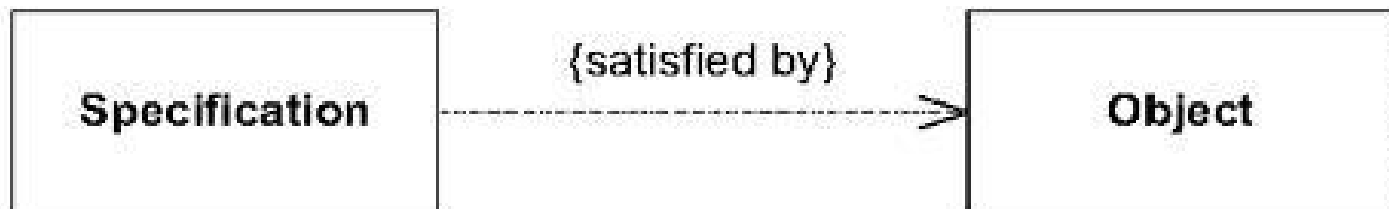
- ▶ Refactoring is described in terms of code transformations with **technical motivations**.
- ▶ Refactoring can also be motivated by an **insight into the domain** and a corresponding **refinement of the model** or its expression in code.
- ▶ Sophisticated domain are developed with an **iterative process of refactoring**.
- ▶ Includes **close involvement of the domain experts** with **developers interested in learning about the domain**.

- ▶ This sort of breakthrough is not a technique, it is an **event**.
 - ▶ The challenge lies in **recognizing** what is happening and deciding **how to** deal with it.



- ▶ **Constraints** make up a particularly important category of model concepts
 - ▶ they often **emerge implicitly**
 - ▶ and **expressing them explicitly** can greatly **improve a design**.
- ▶ **Specification** pattern
 - ▶ Used for Validation
 - ▶ Used for Selection (Quervind)

Specification



```
public interface ISpecification<T>
{
    bool IsSatisfiedBy(T sut);
}

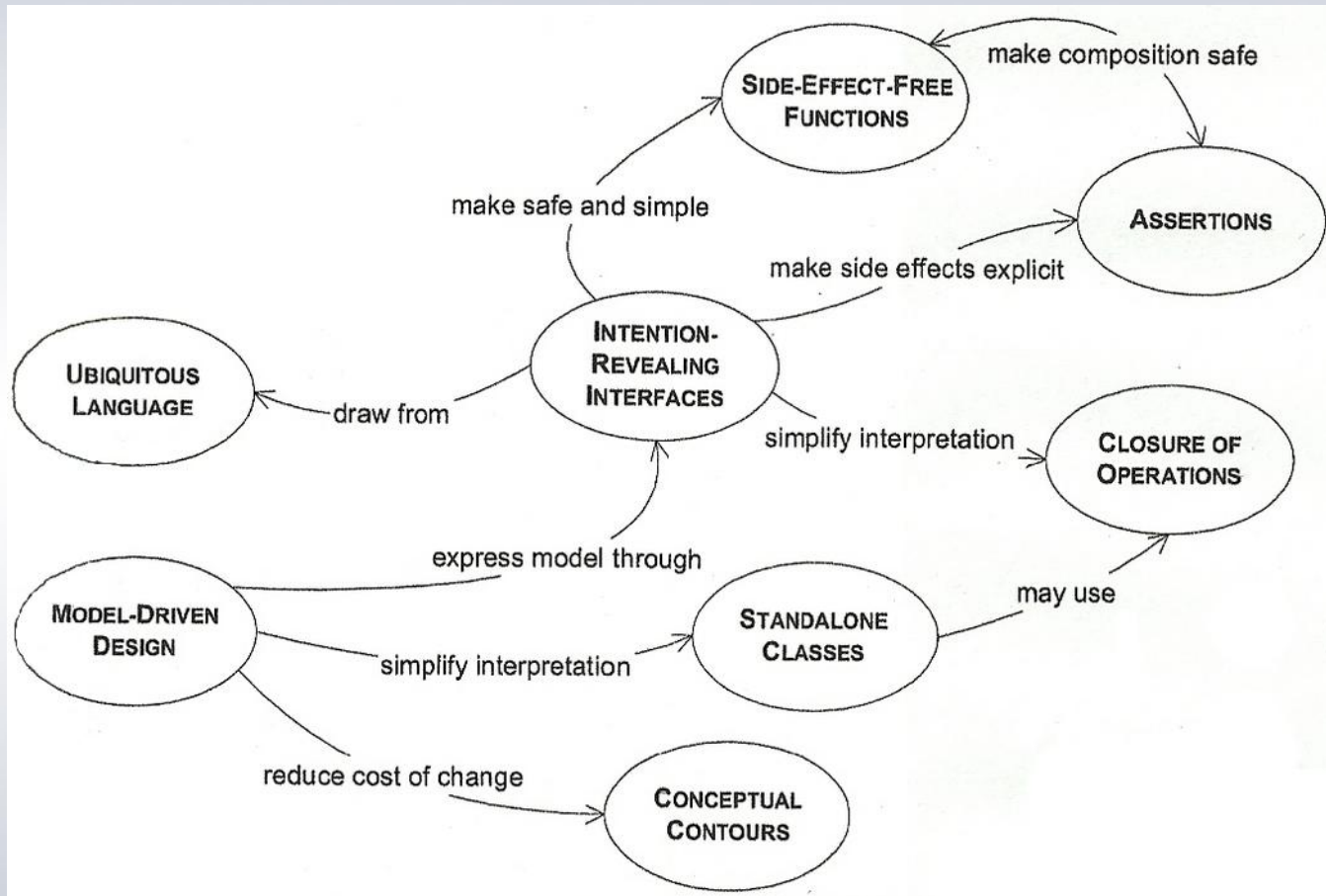
public class EligibleForDiscountSpecification : ISpecification<Customer>
{
    private readonly Product _product;
    public EligibleForDiscountSpecification(Product product)
    {
        _product = product;
    }

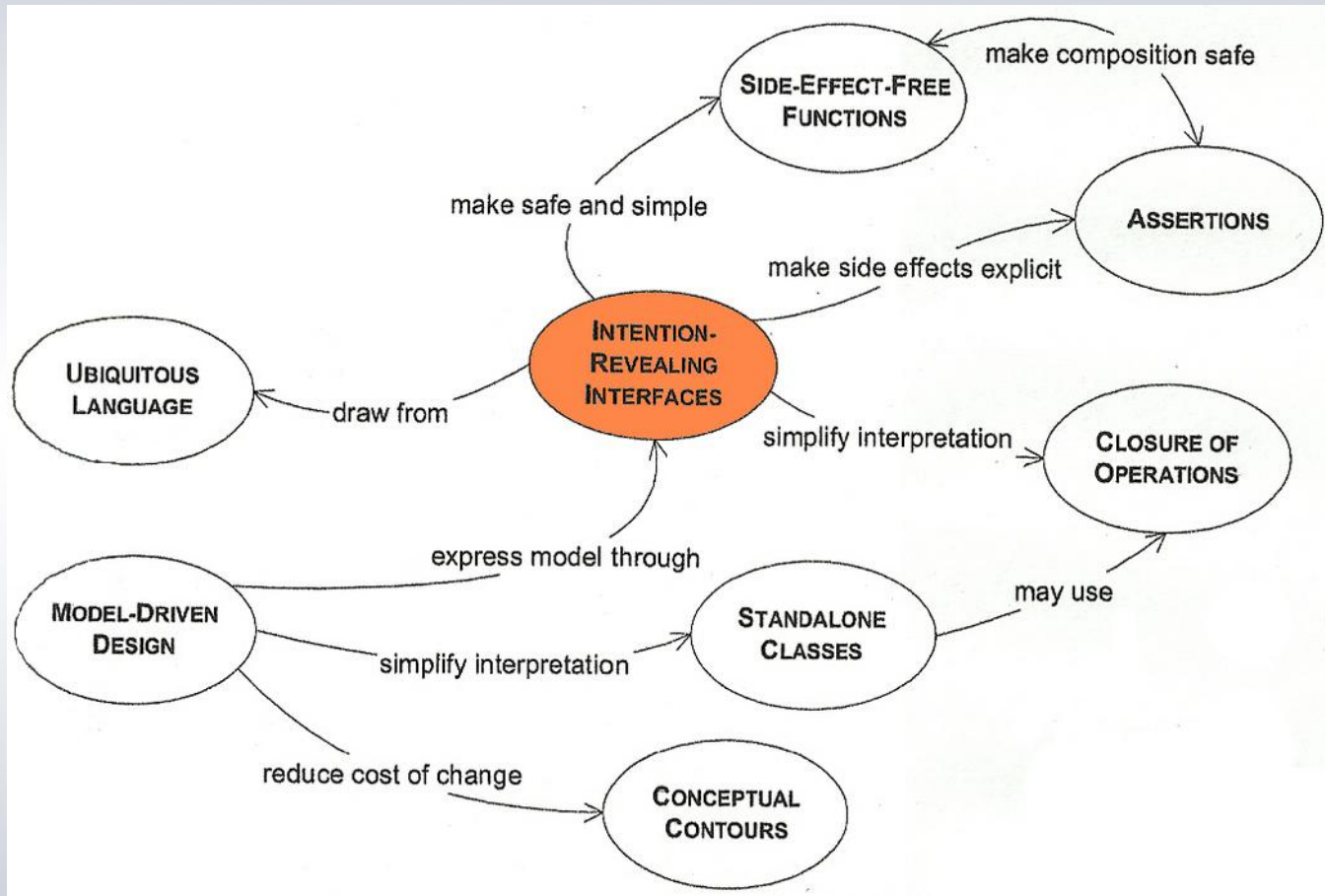
    public bool IsSatisfiedBy(Customer customer)
    {
        return (_product.Price < 100 && customer.CreditRating >= _product.Minimum
    }
}
```

```
[Fact]
public void TestSpecification()
{
    var product = new Product() { MinimumCreditRating = 3, Price = 50 };
    var spec = new EligibleForDiscountSpecification(product);
    var goodCustomer = new Customer() { CreditRating = 3 };
    var badCustomer = new Customer() { CreditRating = 1 };

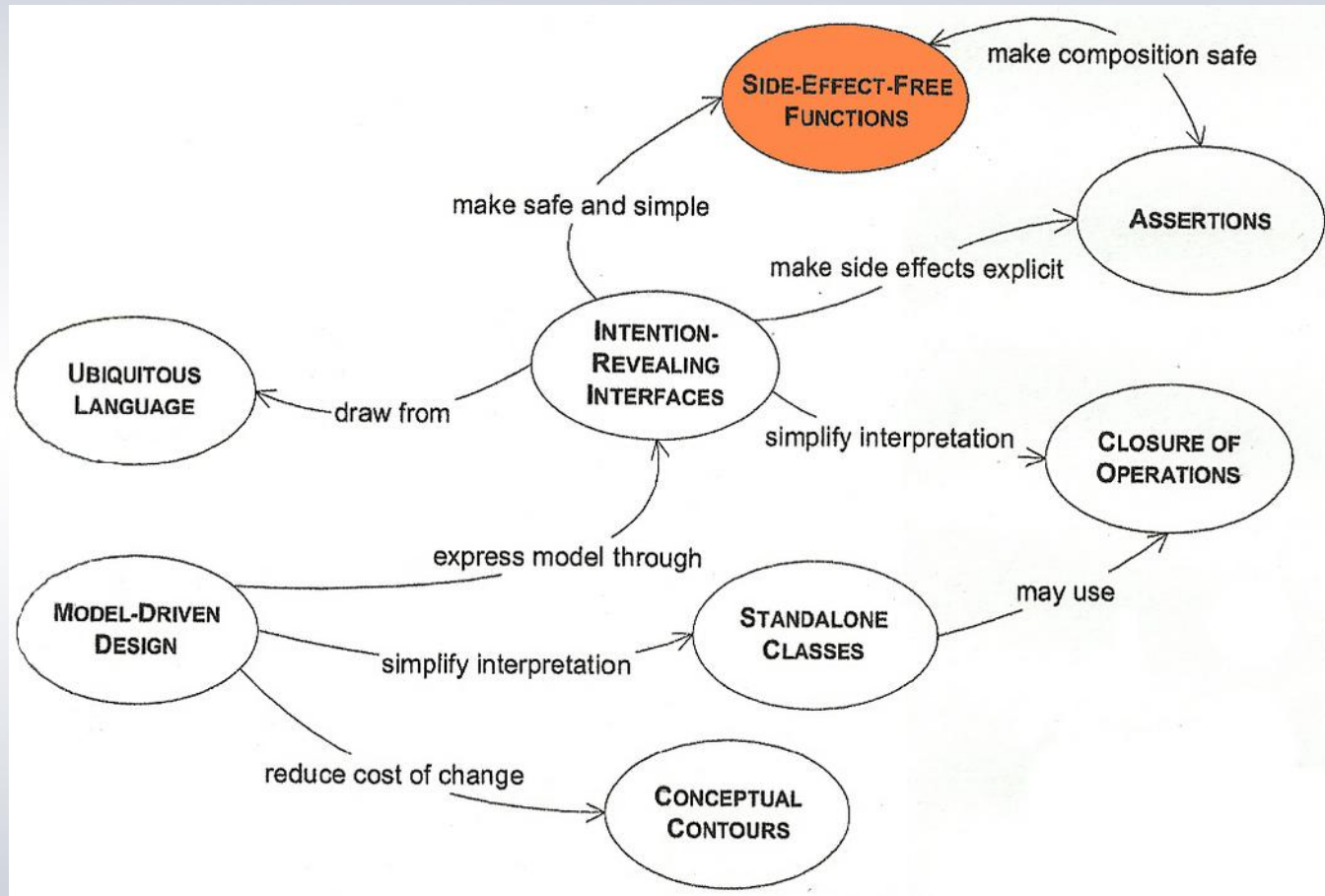
    Assert.True(spec.IsSatisfiedBy(goodCustomer));
    Assert.False(spec.IsSatisfiedBy(badCustomer));
}
```

Casey Charlton : <http://devlicio.us/blogs/casey/archive/2009/03/02/ddd-the-specification-pattern.aspx>

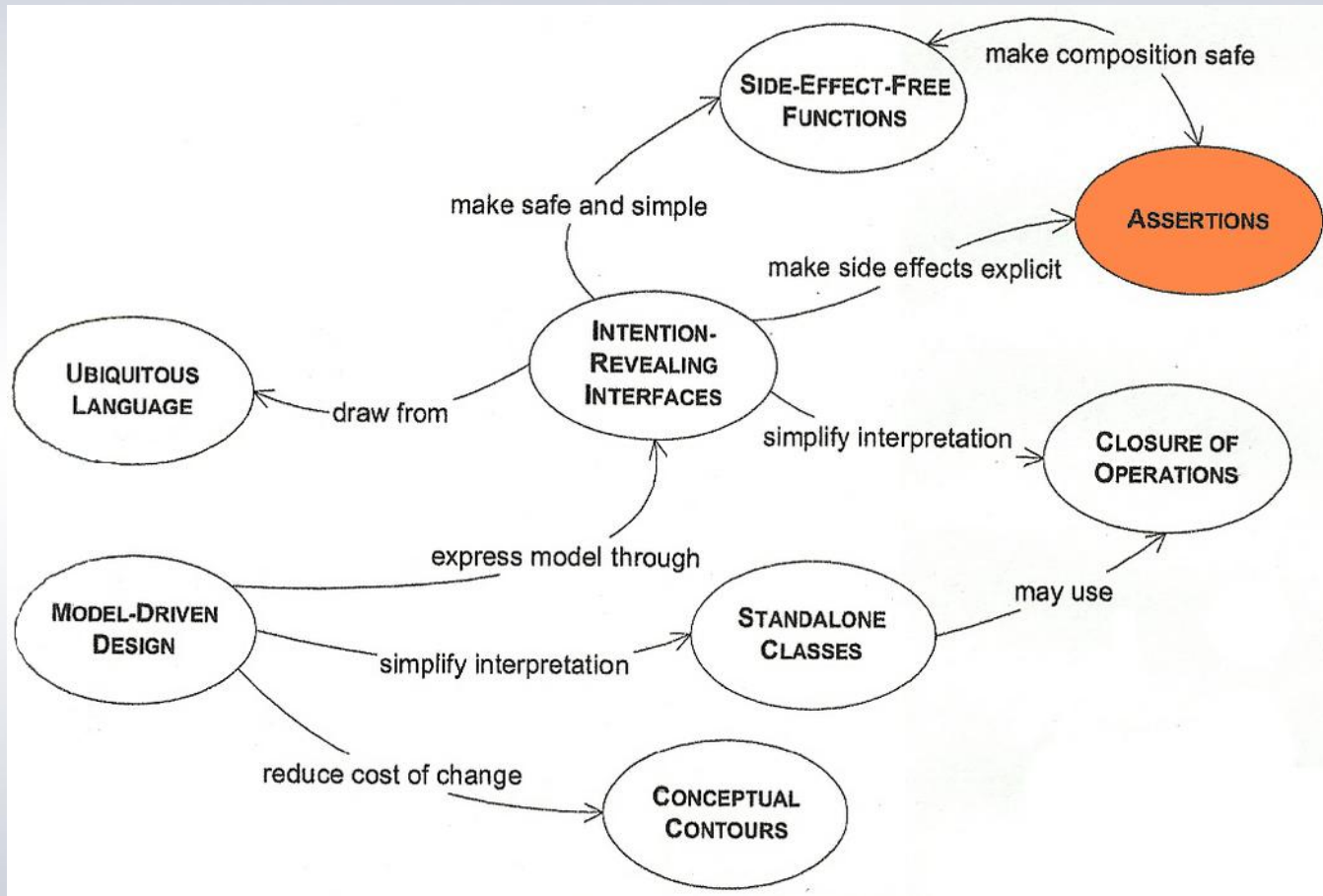




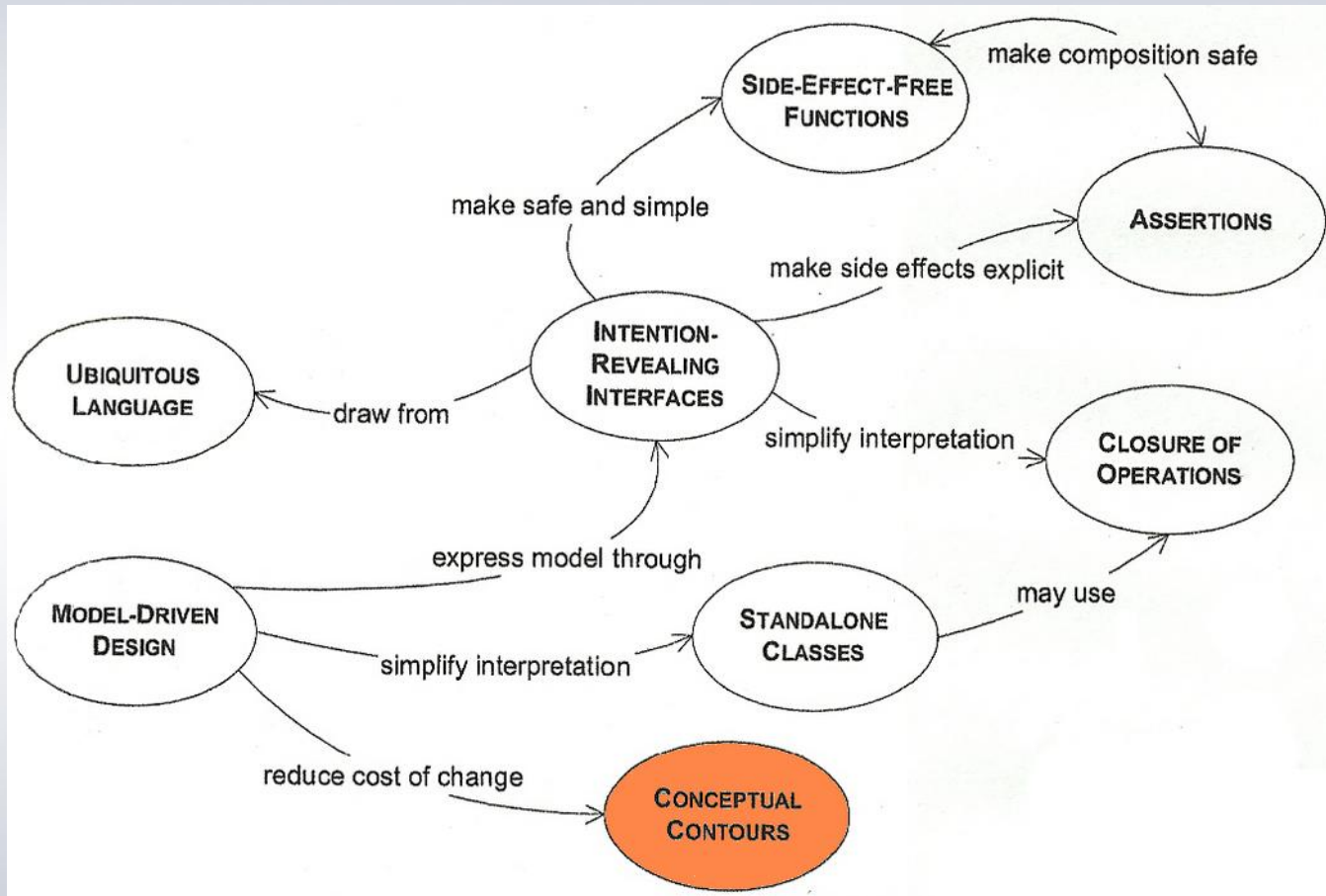
- ▶ Name **classes** and **operations**
 - ▶ to describe their **effect and purpose**
 - ▶ without reference to the means by which they do it
 - ▶ relieving the client developer of the need to understand the internals.
- ▶ Names should
 - ▶ **conform to the Ubiquitous Language**
 - ▶ so that team members can quickly infer their meaning
- ▶ Write a test for a **behavior** before creating it
 - ▶ to force your thinking into client developer mode
 - ▶ Test-Driven Design



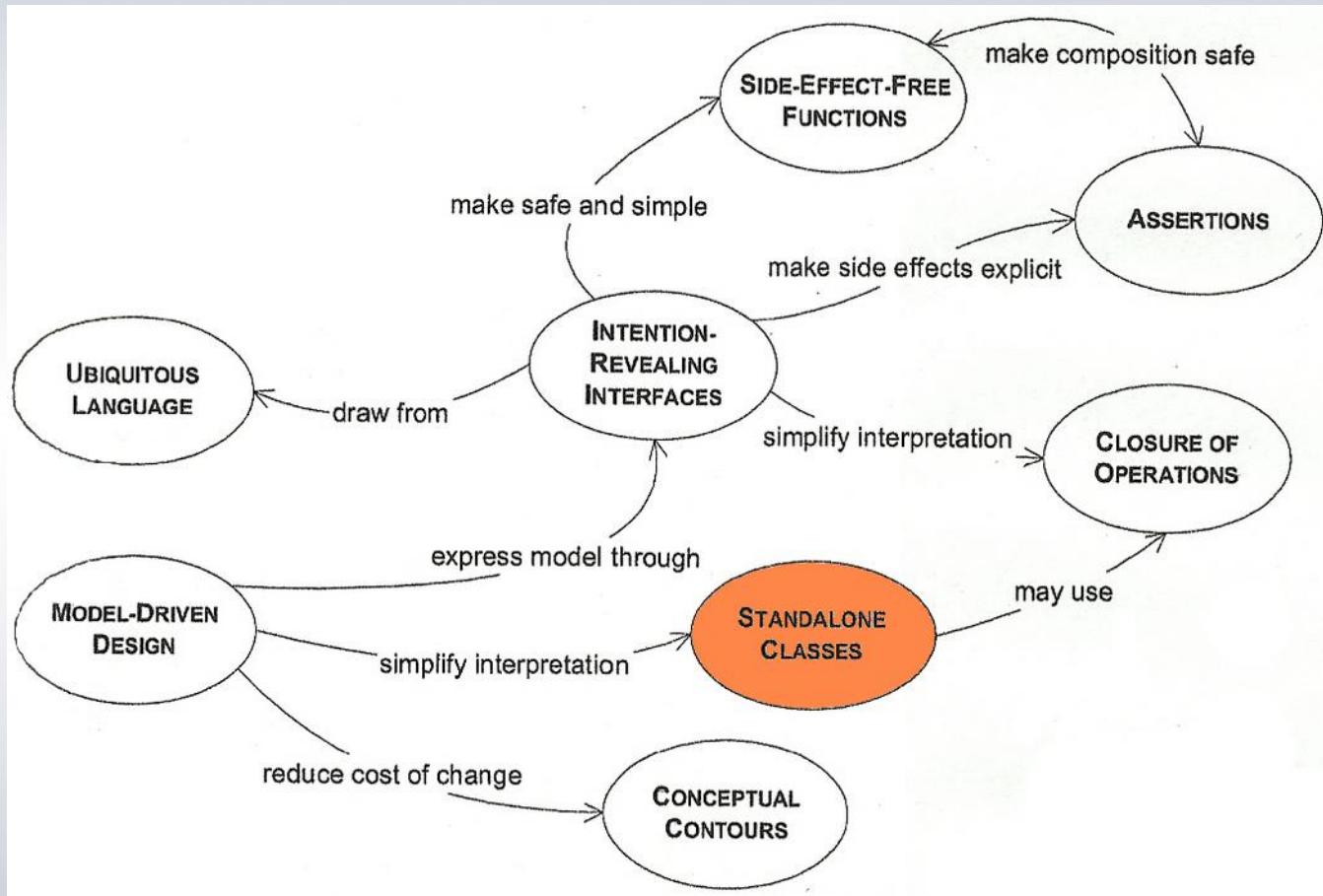
- ▶ Place **as much** of the **logic** as possible **into functions**
 - ▶ operations that return results **with no observable side effects**.
- ▶ Strictly **segregate commands**
 - ▶ methods which result in modifications to observable state
 - ▶ into very **simple operations** that **do not return** domain information
- ▶ **Move complex logic** into Value Objects
 - ▶ when a concept **fitting the responsibility** presents itself
 - ▶ to further **control side effects**



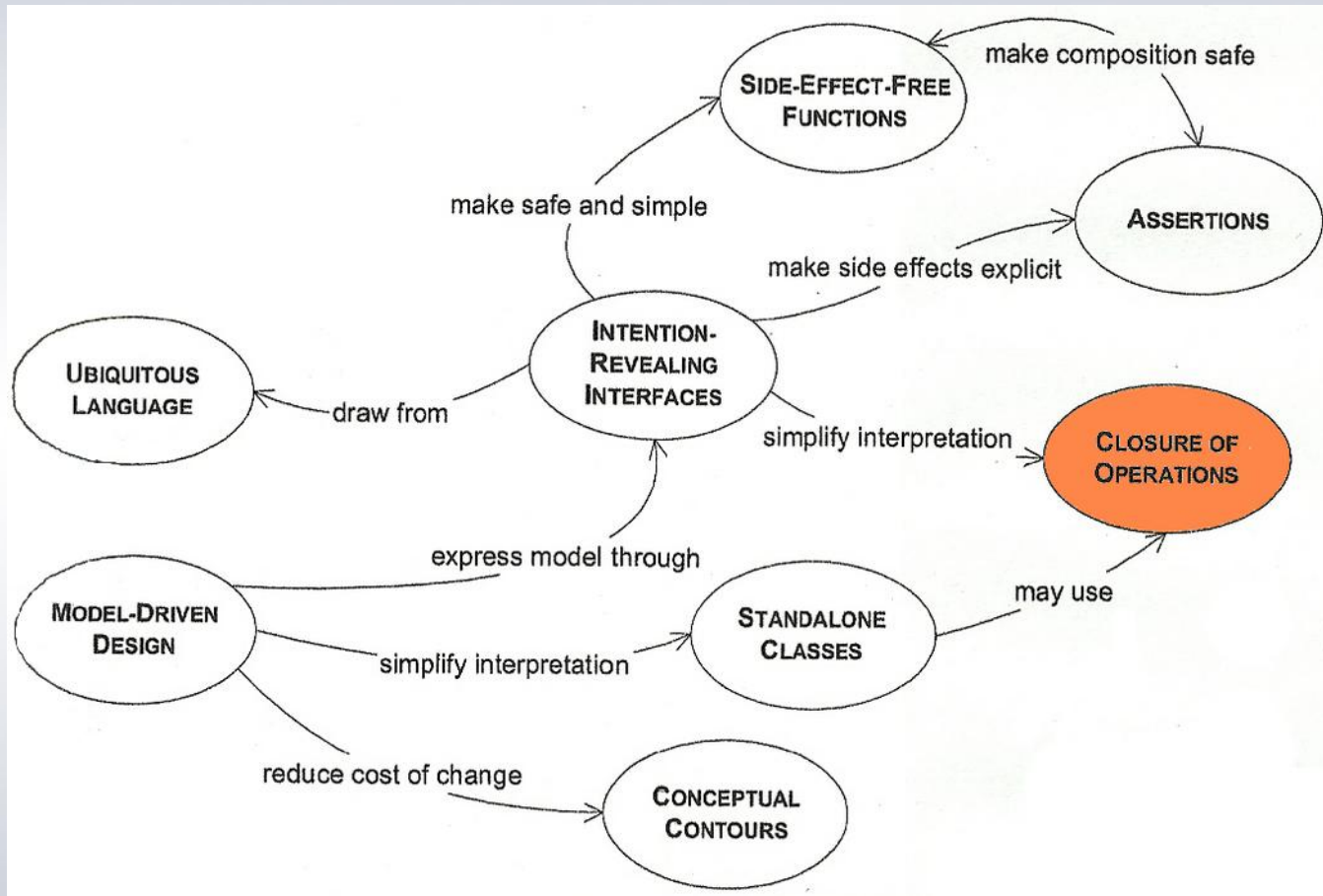
- ▶ Assertions state
 - ▶ **post-conditions** of operations
 - ▶ **invariants** of classes and Aggregates
- ▶ If Assertions **cannot be coded directly** in your programming language
 - ▶ write **automated unit tests** for them
 - ▶ write them into **documentation** or **diagrams**, where it fits the style of the project's development process
- ▶ Seek models with
 - ▶ **coherent** sets of **concepts**
 - ▶ which lead a developer to **infer** the intended **Assertions**
 - ▶ **accelerating** the **learning curve**
 - ▶ and **reducing the risk** of contradictory code



- ▶ Decompose design elements into **cohesive units**
 - ▶ Operations
 - ▶ Interfaces
 - ▶ Classes
 - ▶ Aggregates
- ▶ Take into consideration your intuition
 - ▶ of the **important divisions** in the domain
- ▶ Observe the **axes of change and stability**
 - ▶ through **successive refactoring**
 - ▶ And look for the underlying Conceptual Contours that explain these shearing patterns
- ▶ **Align the model**
 - ▶ **with the consistent aspects** of the domain
 - ▶ that make it a **viable area of knowledge** in the first place



- ▶ **Low coupling** is fundamental to object design.
- ▶ When you can, go **all the way**.
- ▶ Eliminate **all** other concepts from the picture.
- ▶ The class will be completely **self-contained** and can be studied and understood **alone**.
- ▶ Every such self-contained class significantly **eases the burden of understanding** a Module.



- ▶ An operation is **closed under the set of instances of a type** when
 - ▶ Its return type is the same as the type of its argument(s)
 - ▶ the argument(s) and return value are of the same type as the implementer, if the implementer has state that is used in the computation
- ▶ A closed operation provides
 - ▶ a **high-level interface**
 - ▶ **without introducing any dependency** on other concepts
- ▶ Most often applied to Value Objects

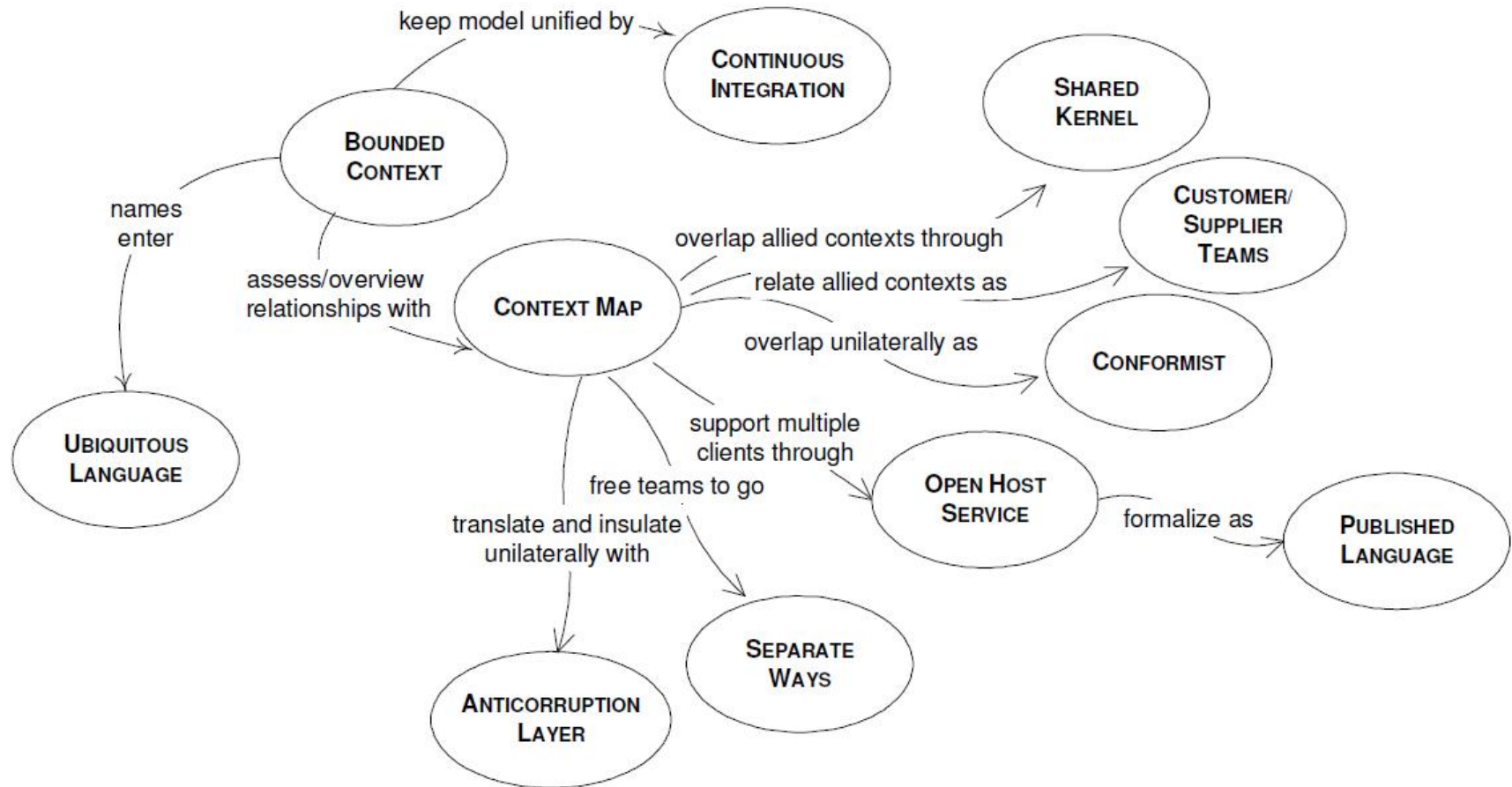
▲ Closed

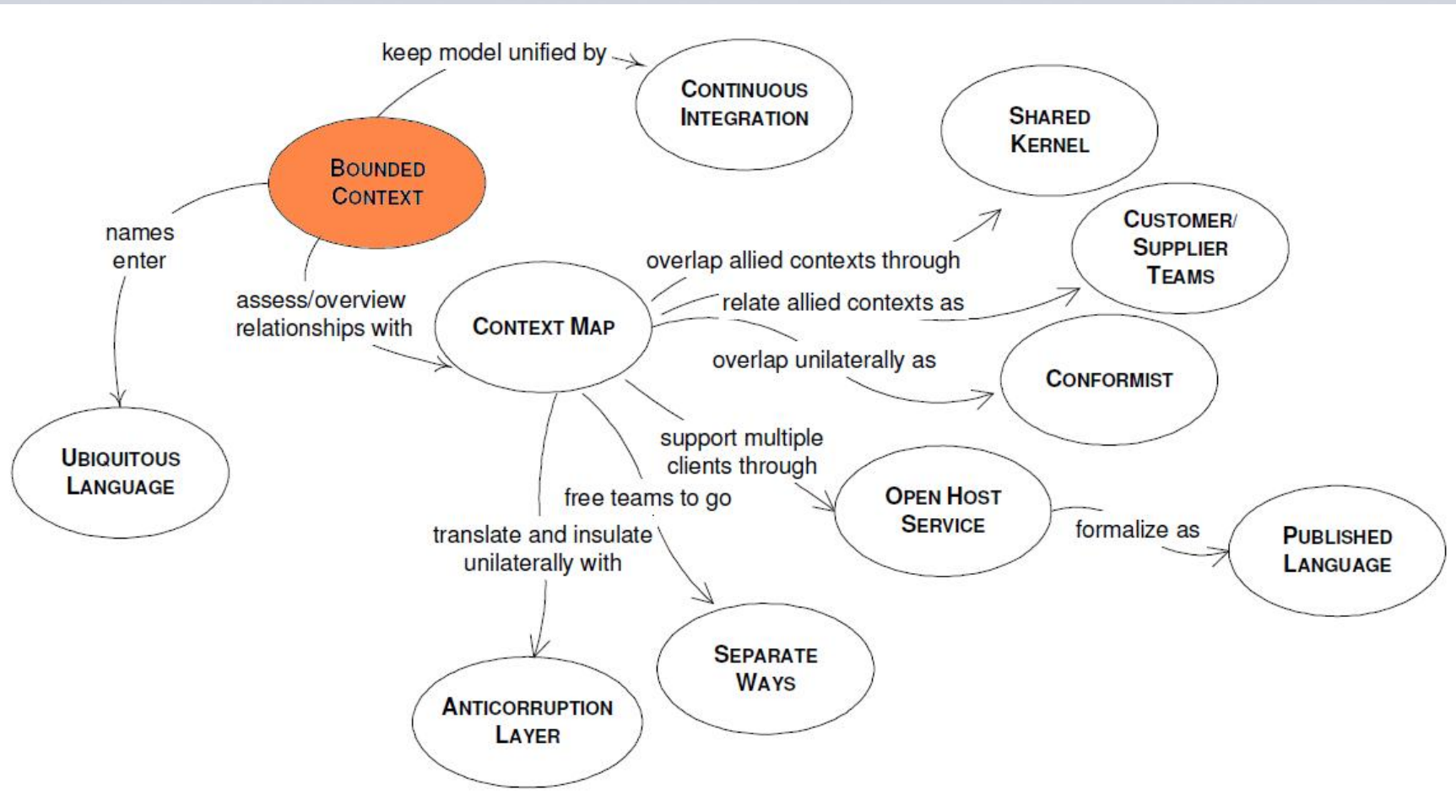
```
public int Add(int x, int y) {  
    return x + y;  
}  
  
public int Subtract(int x, int y) {  
    return x - y;  
}  
  
public int Multiply(int x, int y) {  
    return x * y;  
}  
  
public int Divide(int x, int y) {  
    return x / y;  
}
```

```
public class Person {  
  
    private int _parentId;  
    private Person _spouse;  
  
    public Person GetParent() {  
        return new PersonRepository().Get(_parentId);  
    }  
  
    public Person Wed(Person spouse) {  
        _spouse = spouse;  
        spouse.Wed(this);  
  
        return spouse;  
    }  
}
```

How to reach the enterprise level ?

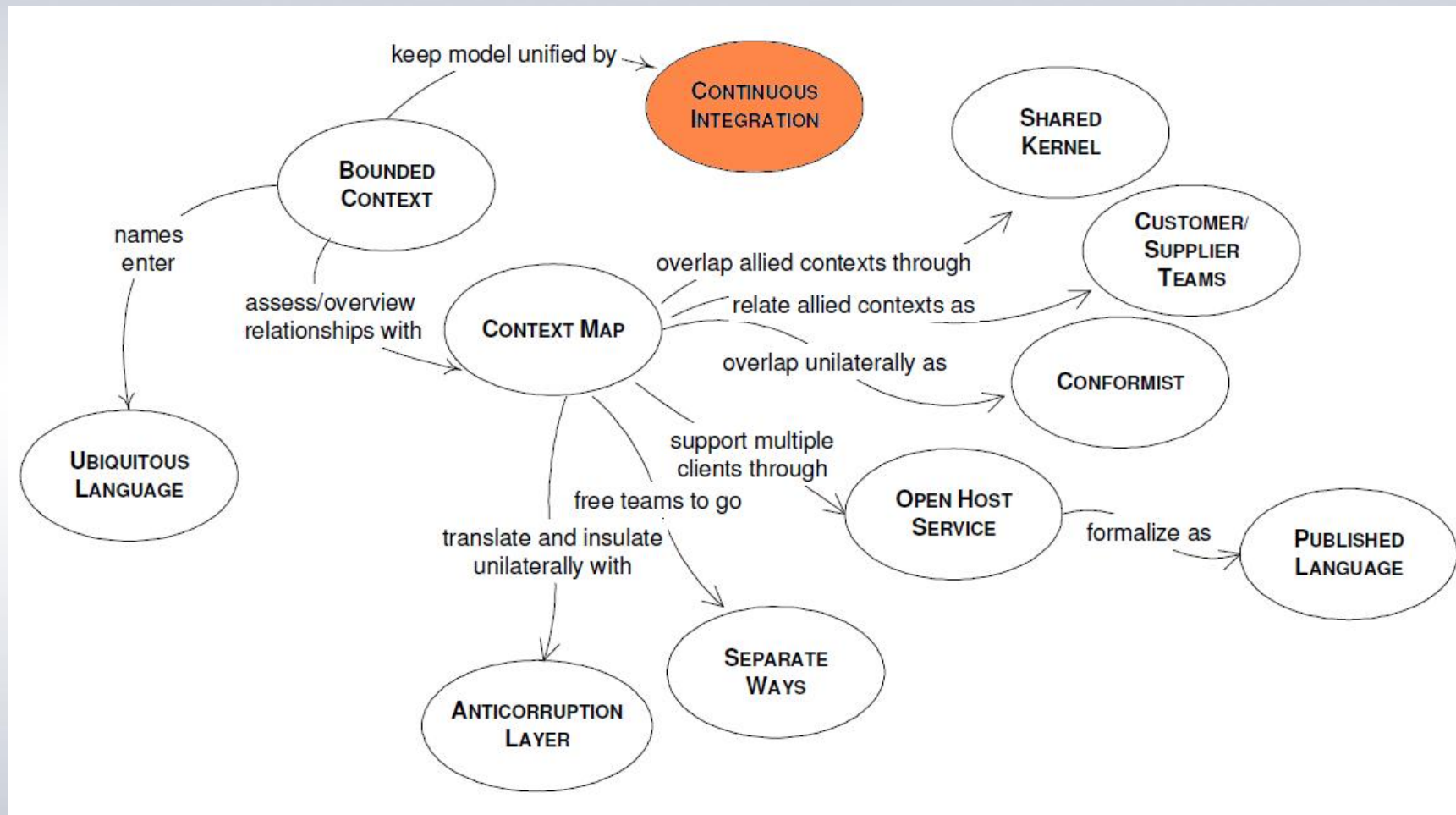
STRATEGIC DESIGN



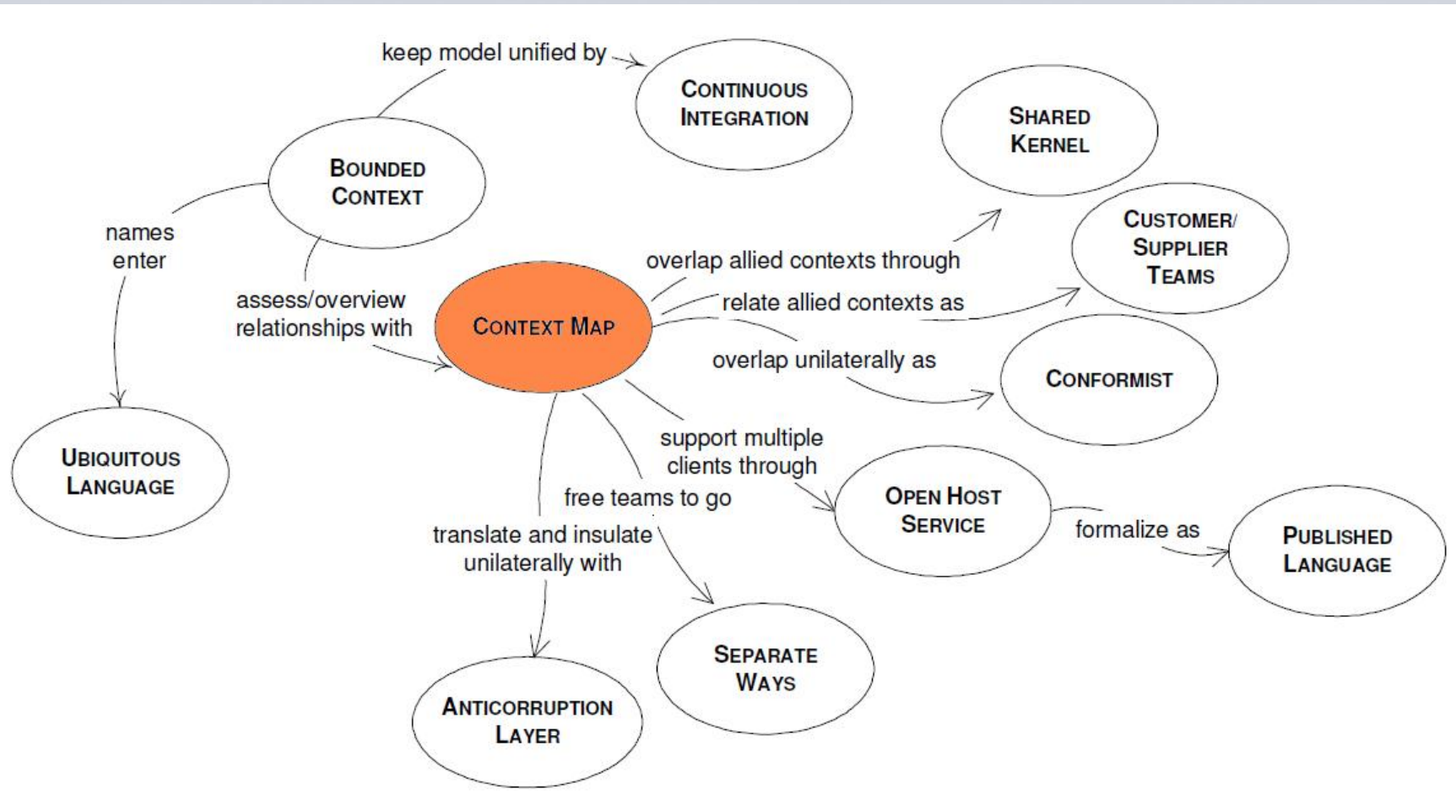


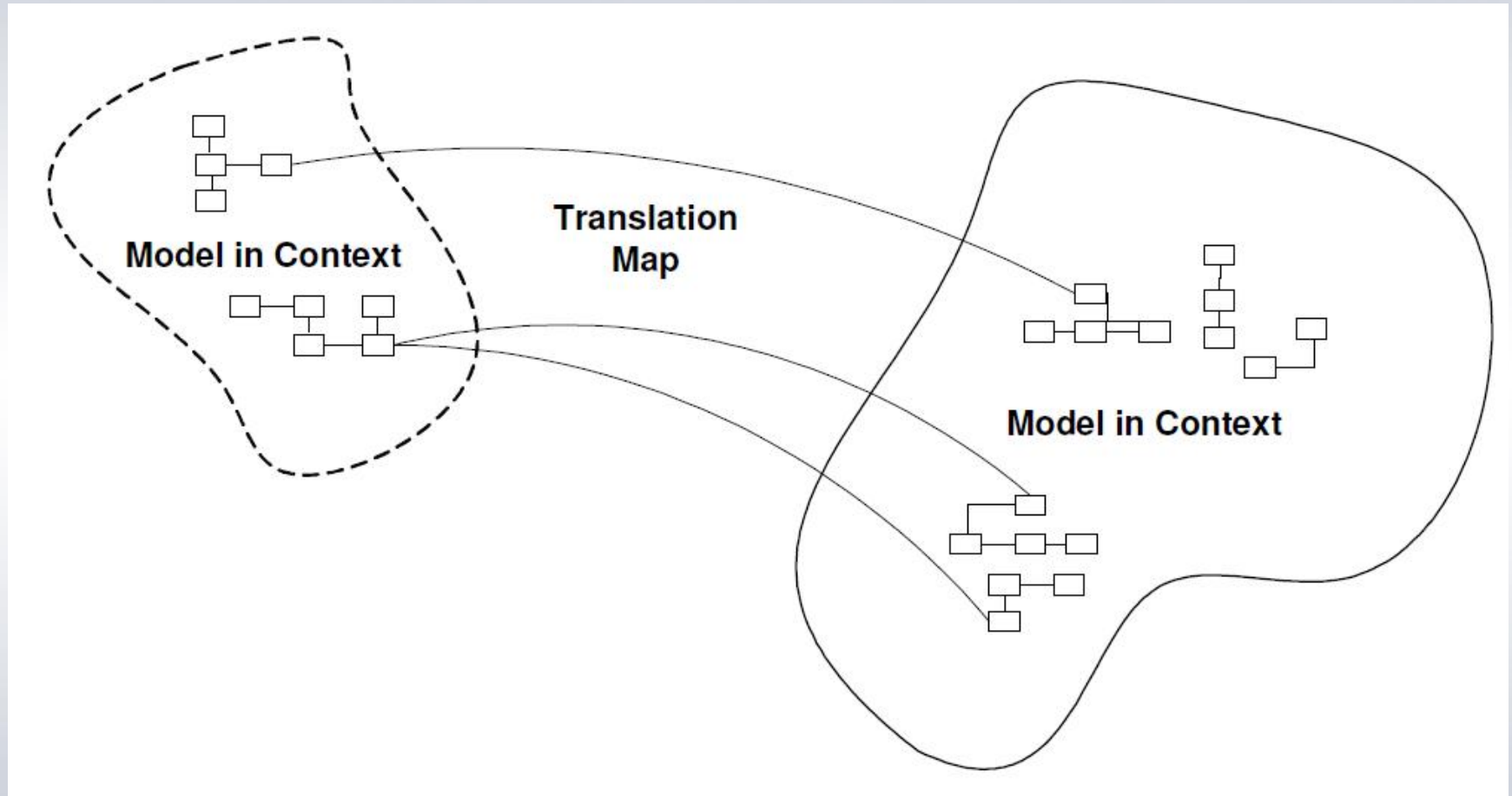
- ▶ Explicitly define the **context within which a model applies**.
- ▶ Explicitly **set boundaries** in terms of
 - ▶ **team organization**
 - ▶ usage within specific **parts of the application**
 - ▶ physical manifestations such as **code bases** and **database schemas**.
- ▶ Keep the model **strictly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

- ▶ e-Commerce application
 - ▶ Online Shopping
 - ▶ Shipping
 - ▶ Supplying
- ▶ Intellectual property business application
 - ▶ Trademark
 - ▶ Patent
 - ▶ Design
 - ▶ Domain Name

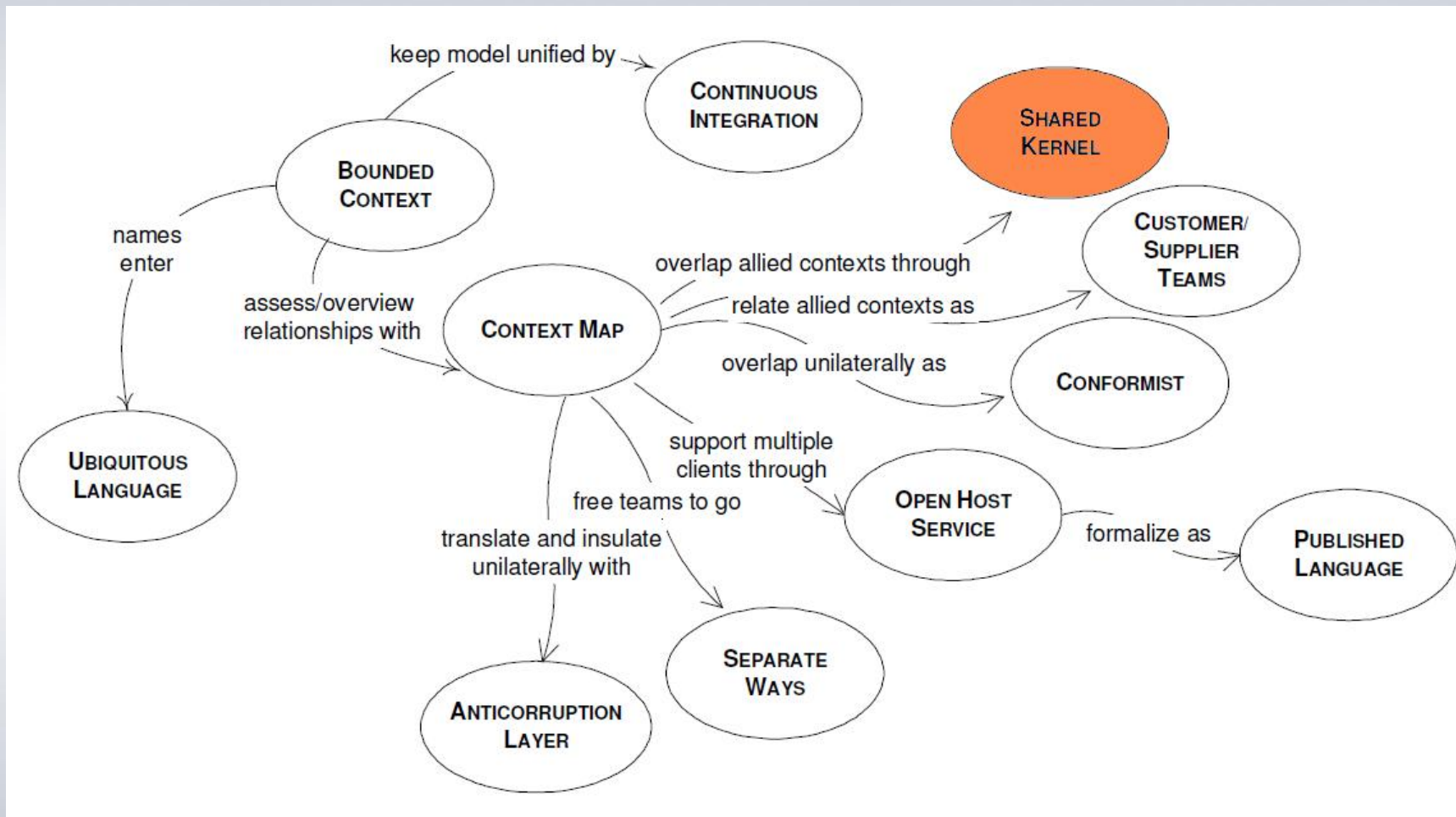


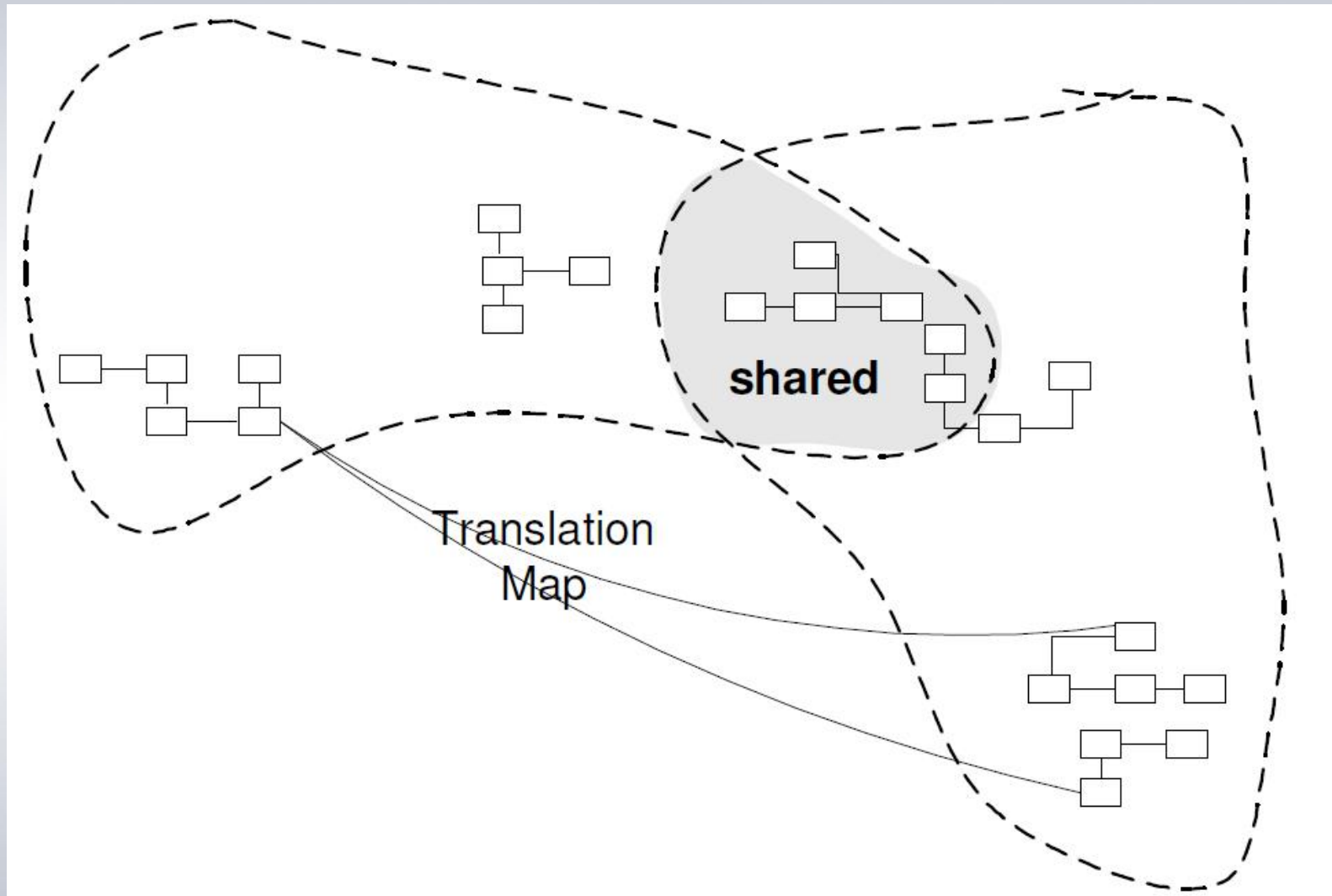
- ▶ Merging all code and other implementation artifacts **frequently**, with **automated tests** to flag fragmentation quickly.
- ▶ Relentlessly exercise the Ubiquitous Language to hammer out a **shared view of the model** as the concepts evolve in different people's heads.



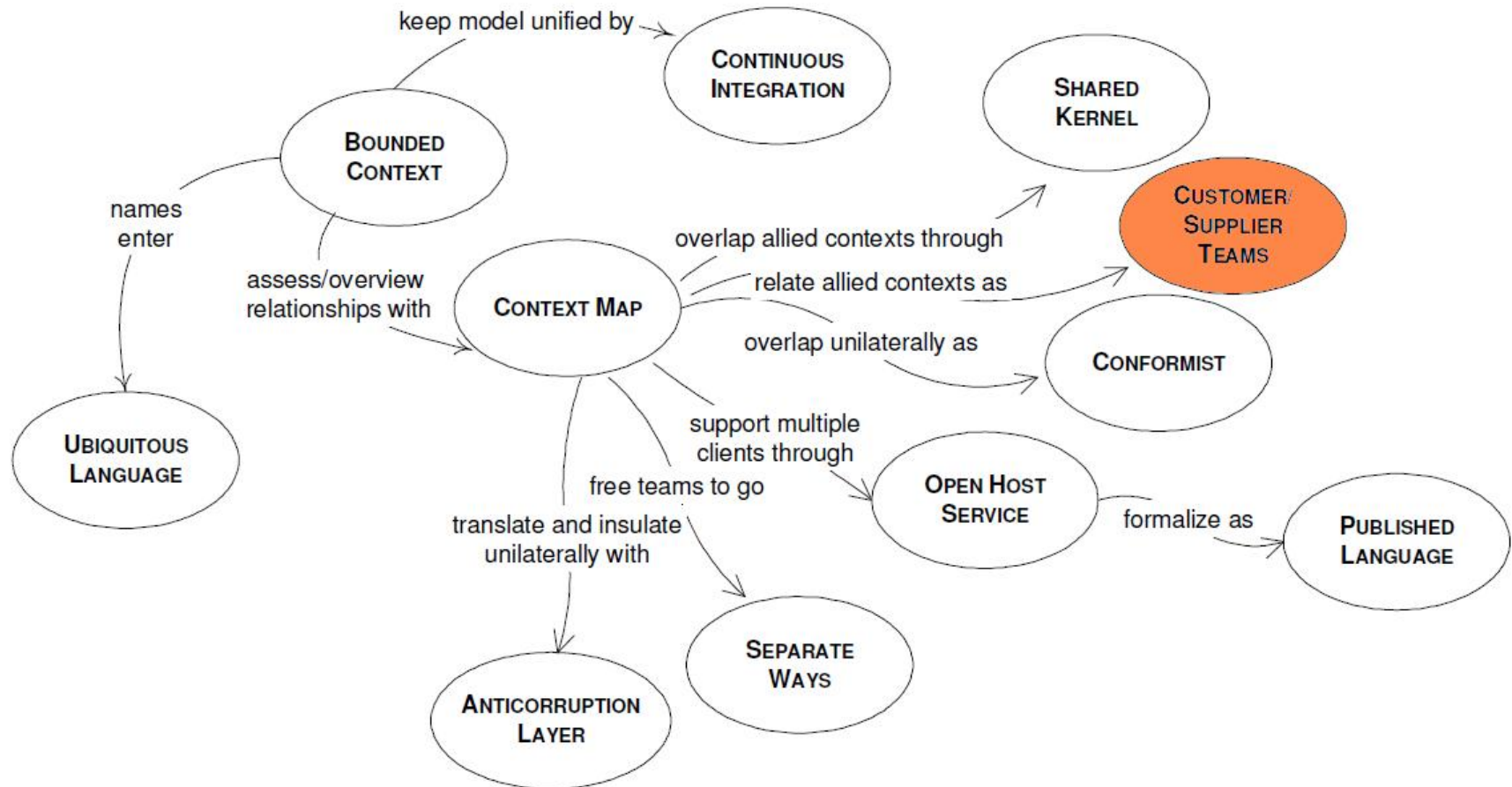


- ▶ **Identify each model** in play on the project and define its Bounded Context.
 - ▶ This **includes the implicit models** of non-object-oriented subsystems.
- ▶ **Name** each Bounded Context, and make the names **part of the Ubiquitous Language**.
- ▶ Describe the **points of contact** between the models, **outlining explicit translation** for any communication and **highlighting any sharing**.
- ▶ Map the **existing** terrain. Take up transformations later.

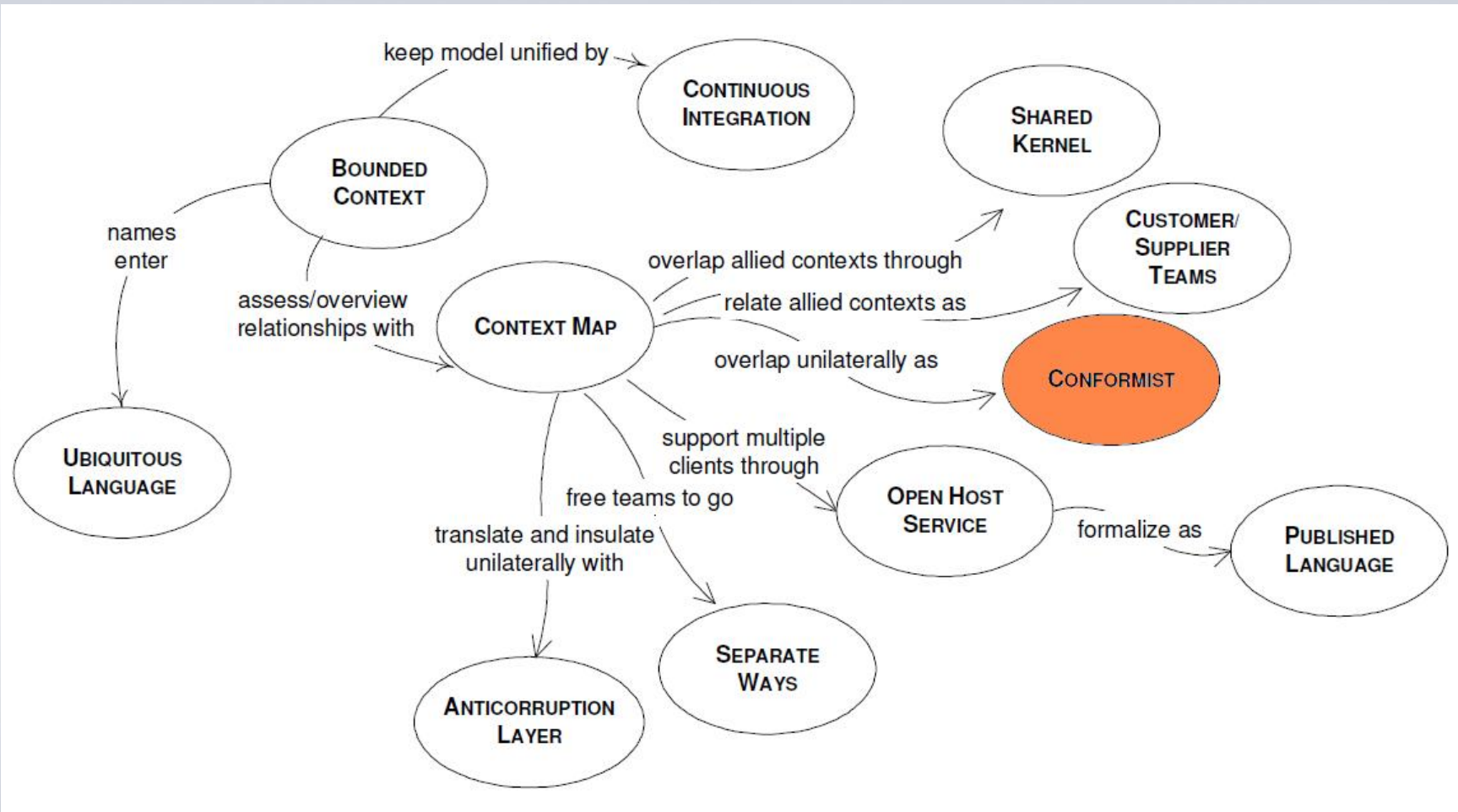




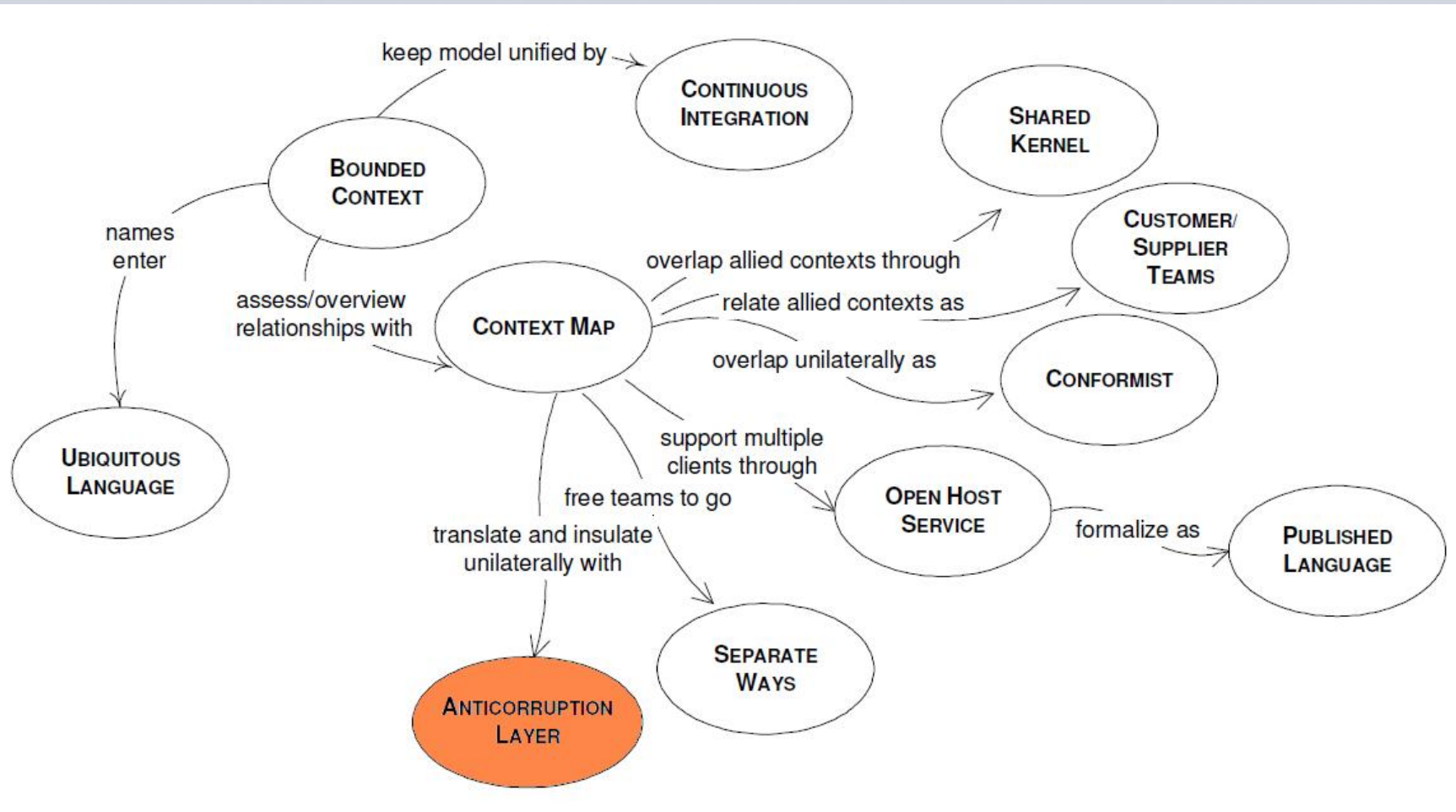
- ▶ **Subset of the domain model** that the **two teams agree to share**.
 - ▶ Of course this includes the **subset of code or of the database design** associated with that part of the model.
- ▶ This explicitly shared stuff **shouldn't be changed without consultation with the other team**.
- ▶ **Integrate** a functional system **frequently**
 - ▶ but somewhat **less often** than the pace of Continuous Integration within the teams
 - ▶ run the tests of **both** teams

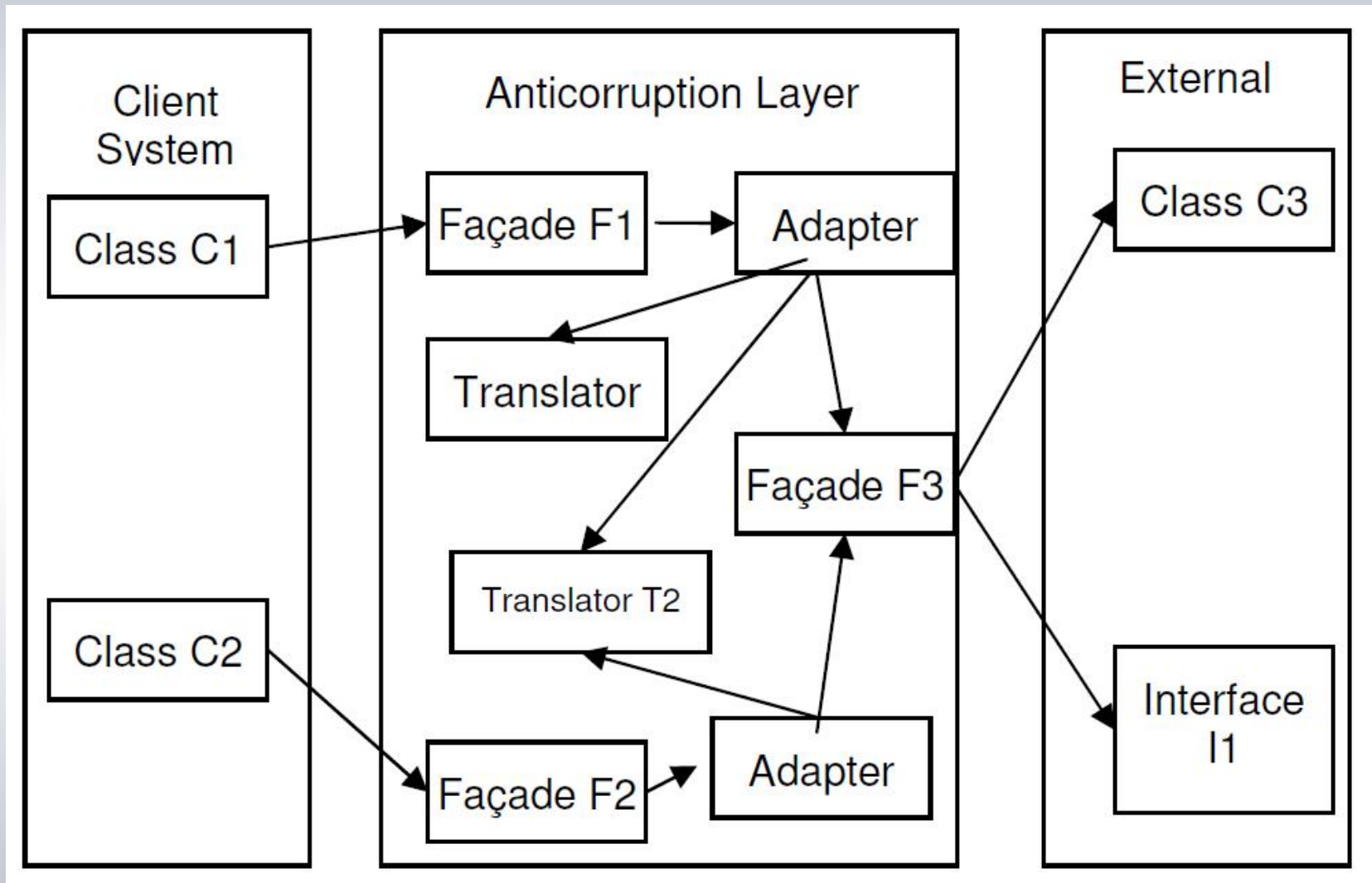


- ▶ Establish a clear **customer/supplier relationship** between the two teams
- ▶ In planning sessions
 - ▶ Make the **downstream** team play the **customer role** to the upstream team
 - ▶ **Negotiate and budget tasks** for downstream requirements so that everyone understands the commitment and schedule
- ▶ **Jointly develop automated acceptance tests** that will validate the interface expected
 - ▶ **Add** these tests to the **upstream team's test suite**, to be run as part of its continuous integration
 - ▶ This testing will free the upstream team to **make changes without fear of side effects downstream**

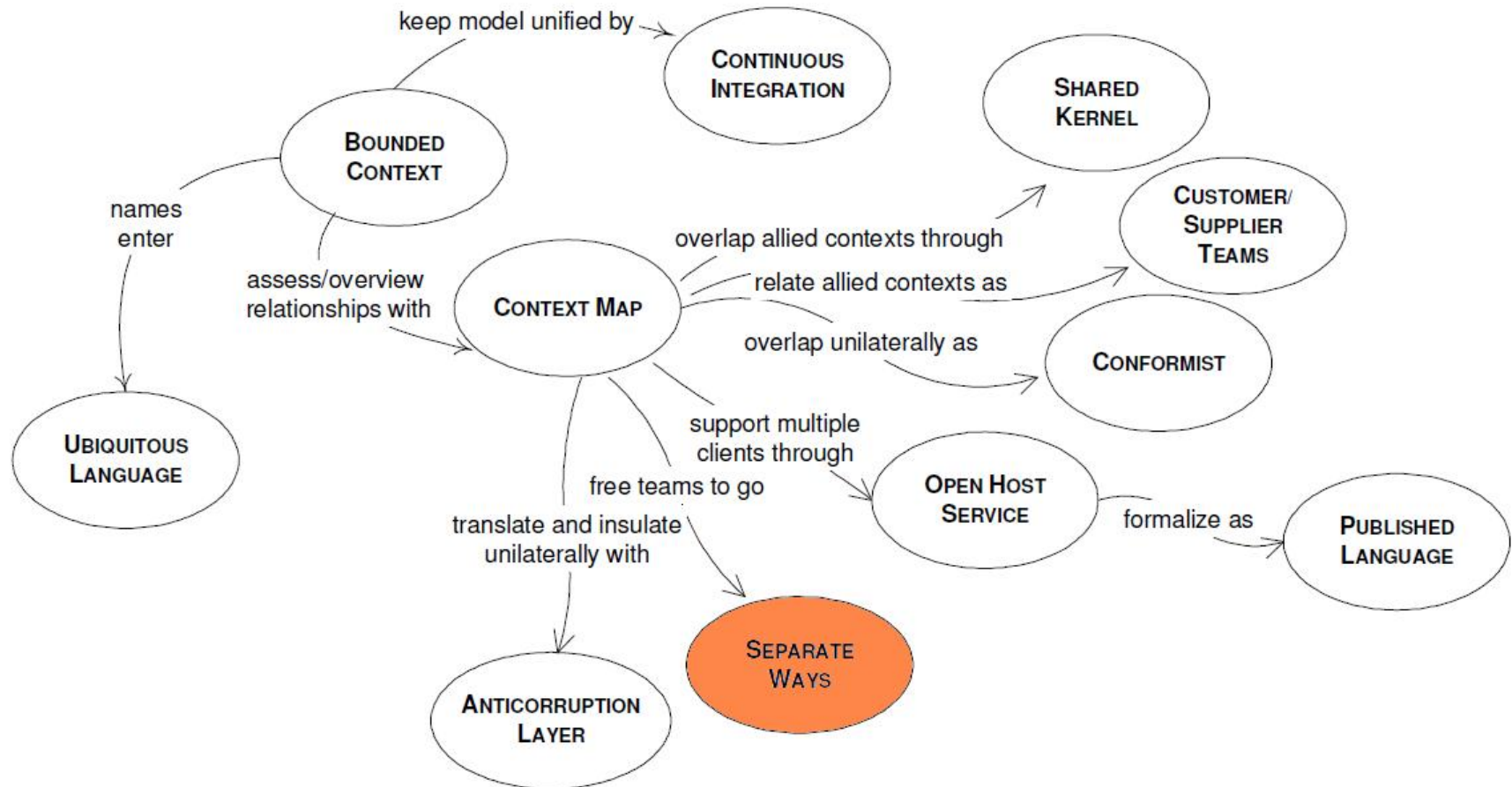


- ▶ Eliminate the complexity of translation between Bounded Contexts by slavishly **adhering to the model of the upstream team**.
 - ▶ Although this cramps the style of the downstream designers and probably does not yield the ideal model for the application, choosing Conformity **enormously simplifies integration**.
 - ▶ Also, you will **share a Ubiquitous Language** with your supplier team.
- ▶ The supplier is in the driver's seat, so it is good to **make communication easy** for them.
- ▶ **Altruism** may be sufficient to get them to share information with you.

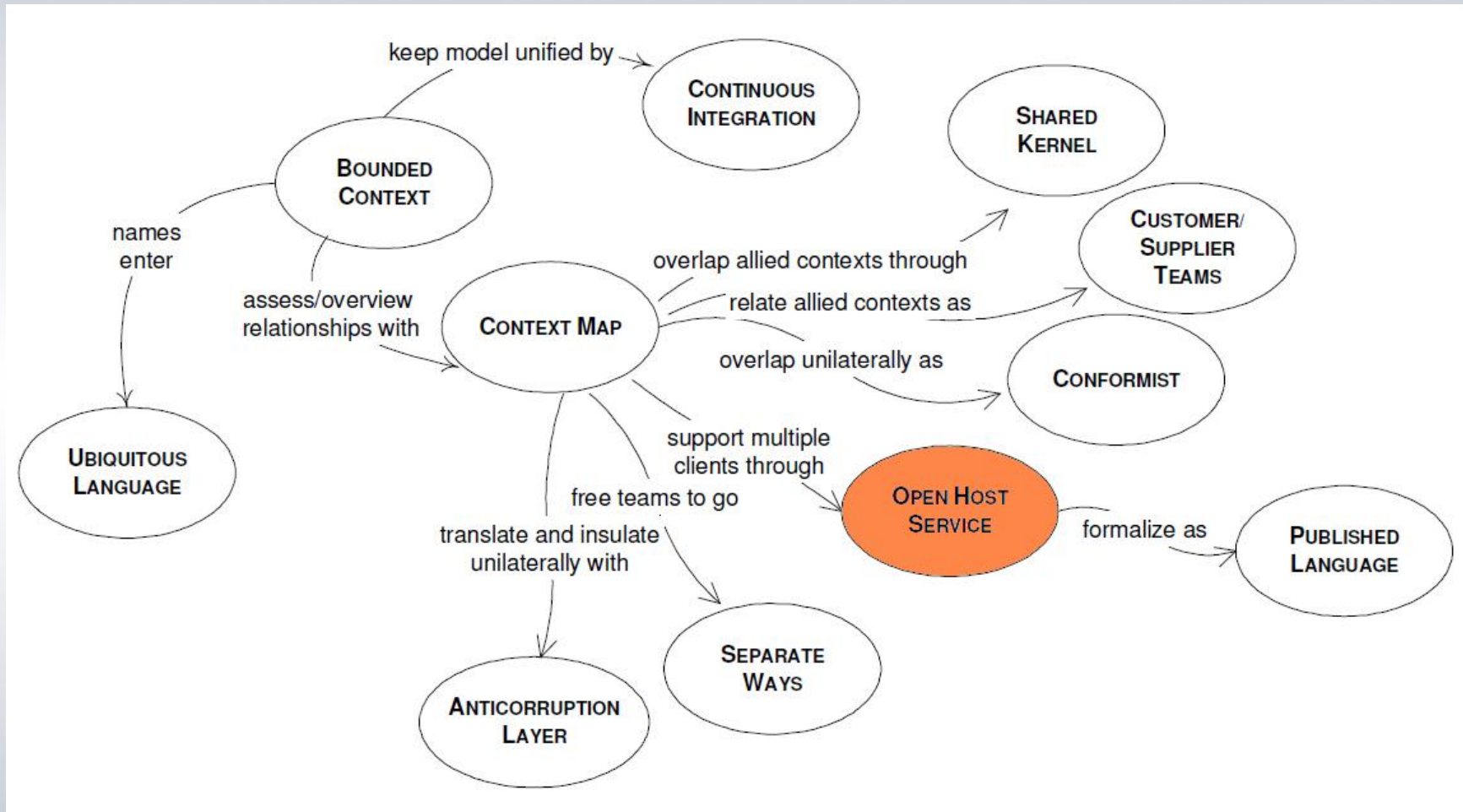




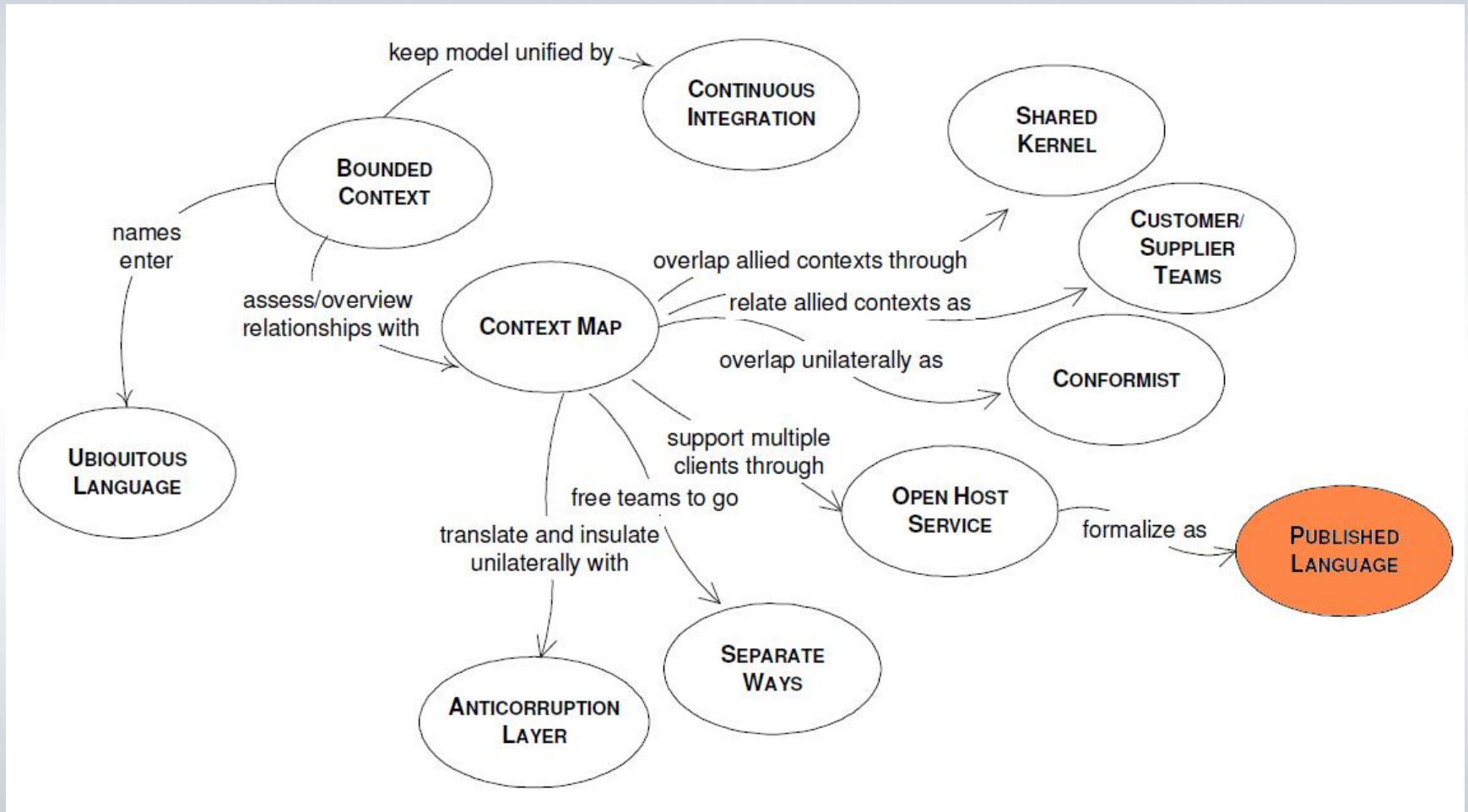
- ▶ Create an **isolating layer** to provide clients with functionality in terms of their own domain model.
- ▶ The layer **talks to the other system through its existing interface**, requiring **little or no modification** to the other system.
- ▶ Internally, the layer **translates in both directions** as necessary between the two models.



- ▶ Declare a Bounded Context to have **no connection to the others at all**, allowing developers to find simple, specialized solutions within this **small scope**.

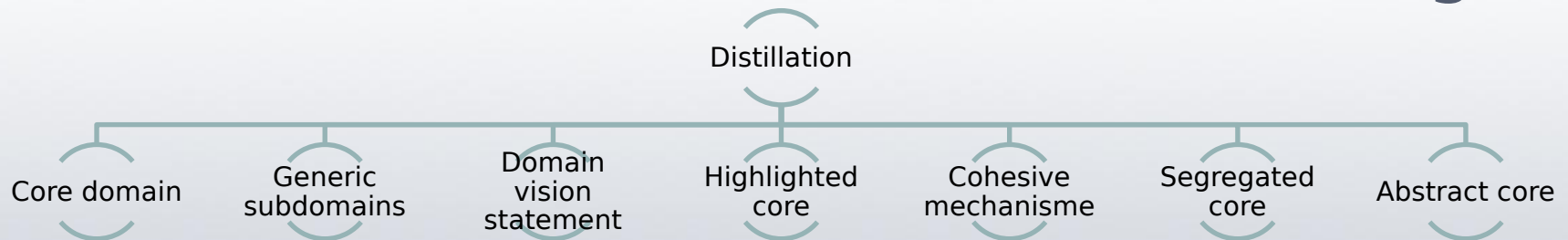


- ▶ Define a **protocol** that gives access to your subsystem as **a set of Service**.
- ▶ **Open the protocol** so that all who need to integrate with you can use it.
- ▶ **Enhance and expand** the protocol to handle new integration requirements.
- ▶ When a single team has idiosyncratic needs, use a **one-off translator** to augment the protocol for that special case so that the shared protocol can stay simple and coherent.



- ▶ Use a **well-documented shared language** that can express the necessary domain information as a **common medium of communication**, translating as necessary into and out of that language.

- ▀ Distillation is the process of separating the components of a mixture to extract the essence in a form that makes it more valuable and useful
- ▀ A model is a distillation of knowledge



▶ When

- ▶ In a large that has many contributing components
- ▶ It becomes difficult to identify the real business asset

▶ Then

- ▶ Define a core domain and provide a means of easily distinguishing it from the rest
- ▶ Bring the most valuable concepts into sharp relief
- ▶ Make the core small

- ▶ When
 - ▶ Some parts of the model add complexity without capturing specialized knowledge
 - ▶ These elements are essential to the functioning of the system
- ▶ Then
 - ▶ Identify cohesive subdomains
 - ▶ Factor out generic models
 - ▶ Place them into separate modules
 - ▶ Give their development lower priority than the core domain

▲ When

- ▲ The domain model doesn't even exist at the beginning of a project
- ▲ The critical aspects of the domain model may span multiple bounded contexts

▲ Then

- ▲ Write a short description of the core domain
- ▲ Ignore the aspects that do not distinguish this domain model from others

Keep it narrow

▲ When

- ▲ The mental labor of constantly filtering the model to identify the key parts absorbs concentration better spent on design thinking, and it requires comprehensive knowledge of the model
- ▲ The core domain must be made easier to see

▲ Then

▲ Either

- Write a very brief document (three to seven sparse pages) that describes the core domain and the primary interactions among core elements

▲ Or

- Flag the elements of the core domain within the primary repository of the model, without particularly trying to elucidate its role
- Make it effortless for a developer to know what is in or out of the core

▲ When

- ▲ A large number of methods that provide algorithms for resolving the problem obscure the methods that express the problem

▲ Then

- ▲ Partition a conceptually cohesive mechanism into a separate lightweight framework
- ▲ Expose the capabilities of the framework with an intention-revealing interface

▶ When

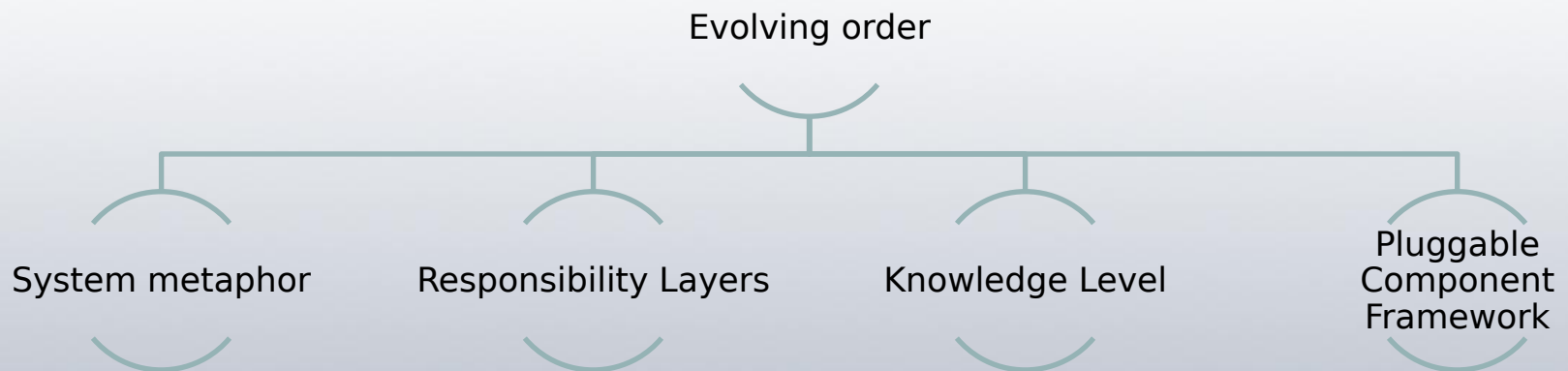
- ▶ Elements in the model may partially serve the core domain and partially play supporting roles
- ▶ Core elements may be tightly coupled to generic ones
- ▶ The conceptual cohesion of the core may not be strong or visible

▶ Then

- ▶ Refactor the model to separate the core concepts from supporting players
- ▶ Strengthen the cohesion of the core while reducing its coupling to other code
- ▶ Factor all generic or supporting elements into other objects and place them into other packages

- ▶ When
 - ▶ There is a lot of interaction between subdomains in separate modules
 - ▶ The model is obscured by references between modules
- ▶ Then
 - ▶ Identify the most fundamental differentiating concepts in the model and factor them into distinct classes, abstract classes, or interfaces
 - ▶ Design this abstract model so that it expresses most of the interaction between significant components
 - ▶ Place this abstract model in its own module, while the specialized, detailed implementation classes are left in their own modules defined by subdomain

- ▶ A large-scale structure is a language that lets you discuss and understand the system in broad strokes
- ▶ A set of high-level concepts or rules, or both, establishes a pattern of design for an entire system



▲ When

- ▲ Architectures straitjacket a project with up-front design assumptions and take too much power away from the developers/designers
- ▲ Developers dumb down the application to fit the structure, or they subvert it and have no structure at all

▲ Then

- ▲ Let this conceptual large-scale structure evolve with the application, possibly changing to a completely different type of structure along the way
- ▲ Don't over constrain the detailed design and model decisions that must be made with detailed knowledge
- ▲ Find a minimal set of structures that solves the problems that have emerged : Less is more

▲ When

- ▲ Developers and users alike need tangible ways to understand the system and share a view of the system as a whole
- ▲ A concrete analogy to the system emerges that captures the imagination of team members and seems to lead thinking in a useful direction

▲ Then

- ▲ Adopt it as a large-scale structure
- ▲ Organize the design around this metaphor and absorb it into the ubiquitous language
- ▲ The system metaphor should both
 - facilitate communication about the system
 - and guide development of it

▲ When

- ▲ Each individual object has handcrafted responsibilities
- ▲ There are no guidelines, no uniformity, and no ability to handle large swaths of the domain together

▲ Then

- ▲ Look at the conceptual dependencies in your model and the varying sources of change of different parts of your domain
- ▲ Identify natural strata in the domain, cast them as broad abstract responsibilities
- ▲ Refactor the model so that the responsibilities of each domain object, aggregate, and module fit neatly within the responsibility of one layer

- ▶ When
 - ▶ Complexity explode because the roles and relationships between entities vary in different situations
 - ▶ Objects end up with references to other types to cover a variety of cases, or with attributes that are used in different ways in different situations
- ▶ Then
 - ▶ Create a distinct set of objects that can be used to describe and constrain the structure and behavior of the basic model
 - ▶ Keep these concerns separate as two levels
 - one very concrete
 - the other reflecting rules and knowledge that a user or super-user is able to customize

▲ When

- ▲ A variety of applications have to interoperate, all based on the same abstractions but designed independently
- ▲ Translations between multiple bounded contexts limit integration
- ▲ A shared kernel is not feasible for teams that do not work closely together
- ▲ Duplication and fragmentation raise costs of development and installation, and interoperability becomes very difficult

▲ Then

- ▲ Distill an abstract core of interfaces and interactions and create a framework that allows diverse implementations of those interfaces to be freely substituted
- ▲ Allow any application to use those components, so long as it operates strictly through the interfaces of the abstract core

What should you keep in mind while coding ?

BEST PRACTICES

- ▶ TDD also stands for Test Driven **Design**
- ▶ Let the design **emerge** by writing tests before
- ▶ Writing a test before the actual code and its refactoring helps to achieve **higher quality**, because:
 - ▶ It forces developer to **refine code contracts** before working on the implementation.
 - ▶ It forces developers to **make and keep code easily testable**. This requires **loose coupling** and a **separation of cross-cutting concerns**.
 - ▶ Presence of unit tests gives developers more psychological **freedom** to experiment with the code and **refactor** it; managers get **lower risks** related to introducing high-level changes to the project.

- ▶ Don't forget the 3 rules
 - ▶ You are not allowed to write any production code unless it is to make a failing unit test pass.
 - ▶ You are not allowed to write any more of a unit test than is sufficient to fail; and compilation failures are failures.
 - ▶ You are not allowed to write any more production code than is sufficient to pass the one failing unit test.

- ▶ **SRP** → Single Responsibility Principle
 - The notion that an object should have only a single responsibility
- ▶ **OC**P → Open/Close Principle
 - The notion that software entities should be open for extension, but closed for modification
- ▶ **LSP** → Liskov Substitution Principle
 - The notion that objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program (design by contract)
- ▶ **ISP** → Interface Segregation Principle
 - The notion that many client specific interfaces are better than one general purpose interface
- ▶ **DIP** → Dependency Inversion Principle
 - The notion that one should depend upon abstractions, but not depend about concretions

- *Inversion of Control (IoC)* is an approach in software development that favors removing sealed dependencies between classes in order to make code more simple and flexible. We push control of dependency creation outside of the class, while making this dependency explicit.
- Usage of Inversion of Control generally allows to create applications that are more flexible, unit-testable, simple and maintainable in the long run.

```
public class Car
{
    Engine _engine;

    public Car()
    {
        _engine = new
        Engine();
    }
}
```



```
public class Car
{
    IEngine _engine;

    public Car(IEngine
    engine)
    {
        _engine = engine;
    }
}
```

▲ Castle Windsor

- ▲ Castle Windsor is best of breed, mature Inversion of Control container available for .NET and Silverlight.

▲ Microsoft Unity

- ▲ Lightweight extensible dependency injection container with support for constructor, property, and method call injection.

▲ Autofac

- ▲ Manages the dependencies between classes so that applications stay easy to change as they grow in size and complexity.

▲ StructureMap

- ▲ Used judiciously, StructureMap can greatly enhance the opportunities for code reuse by minimizing direct coupling between classes and configuration mechanisms.

▲ Ninject

- ▲ Ninject makes dependency injection so easy that it becomes hard not to follow good practices.

- ▶ Technique by which classes may achieve **polymorphic behavior** and **code reuse** by **containing** other classes which implement the desired functionality **instead of through inheritance**.
- ▶ Note the verb “Prefer” in the title
 - ▶ Blindly applying a “best practice” is just as irresponsible as never applying a “best practice”

```
abstract class TripBase {

    public TripBase(string from, string to) {
        From = from;
        To = to;
    }

    public string From {
        get;
        private set;
    }

    public string To {
        get;
        private set;
    }

    public abstract DateTime CalculateEstimatedArrivalTime();
}

class CarTrip : TripBase {

    public CarTrip(string from, string to)
        : base(from, to) {

    }

    public override DateTime CalculateEstimatedArrivalTime() {
        return DateTime.Now.AddHours(15);
    }
}

class PlaneTrip : TripBase {

    public PlaneTrip(string from, string to)
        : base(from, to) {

    }

    public override DateTime CalculateEstimatedArrivalTime() {
        return DateTime.Now.AddHours(2);
    }
}
```



```
interface ITransportationMode {
    TimeSpan Duration {
        get;
    }
}

class Trip {
    private ITransportationMode _transportationMode;

    public Trip(string from, string to, ITransportationMode transportationMode) {
        From = from;
        To = to;
        _transportationMode = transportationMode;
    }

    public string From {
        get;
        private set;
    }

    public string To {
        get;
        private set;
    }

    public DateTime CalculateEstimatedArrivalTime() {
        return DateTime.Now.Add(_transportationMode.Duration);
    }
}

class CarTransportation : ITransportationMode {
    public TimeSpan Duration {
        get {
            return TimeSpan.FromHours(15);
        }
    }
}

class PlaneTransportation : ITransportationMode {
    public TimeSpan Duration {
        get {
            return TimeSpan.FromHours(2);
        }
    }
}
```

- ▶ Use an Object/Relational mapper
 - ▶ Entity Framework
 - ▶ Nhibernate
 - ▶ LLBLGen
 - ▶ ObjectMapper
 - ▶ Massive
 - ▶ PetaPoco
- ▶ Learn how it works
 - ▶ its strength
 - ▶ its weakness

What kind of architecture to use ?

DISTRIBUTED DOMAIN-DRIVEN DESIGN

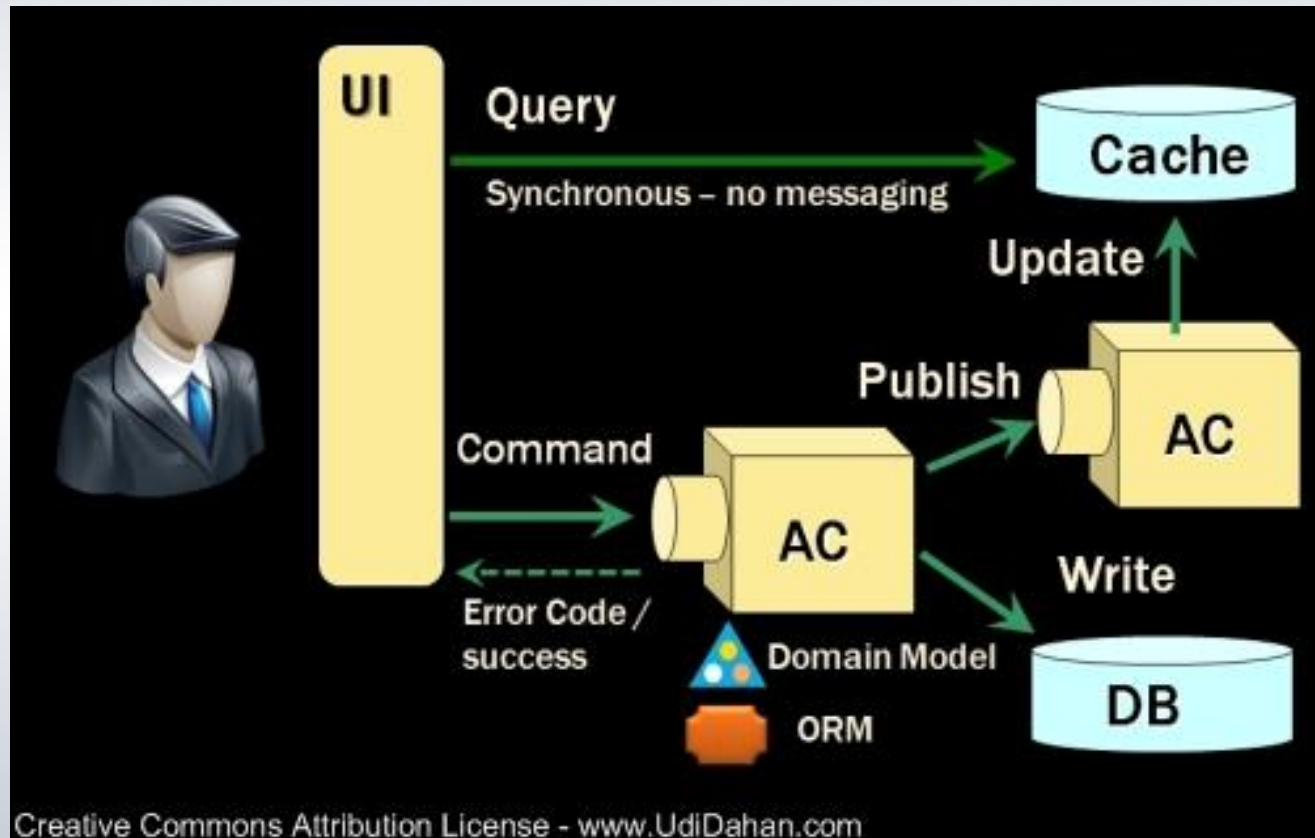
- ▶ Command-Query Responsibility Segregation
 - ▶ Style of architecture coined by Greg Young
 - Queries to read data
 - Commands to write data
 - ▶ Often uses
 - DDD
 - Message-based architecture
 - Event sourcing
 - Service bus

▴ Command-Query Separation

“Every method should either be a **command** that **performs an action**, or a **query** that **returns data** to the caller, but **not both**. In other words, asking a question should not change the answer.”

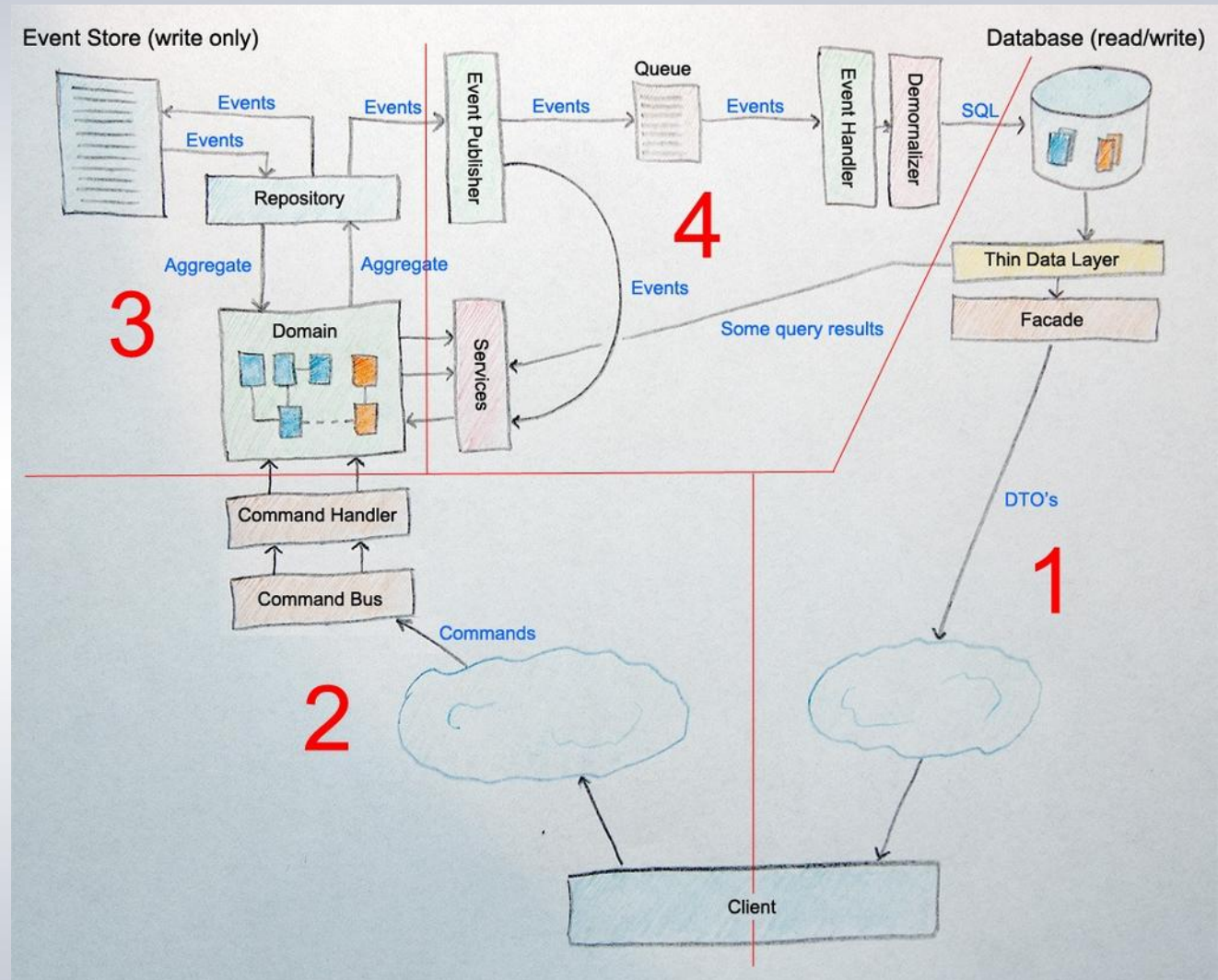
Bertrand Meyer

- AC = Autonomous Components
- Query cache not necessarily a relational database



<http://www.udidahan.com/2009/12/09/clarified-cqrs/>

1. Queries
2. Commands
3. Internal events
4. External events (Publishing)



<http://cre8ivethought.com/blog/2009/11/12/cqrs--la->

Thank You

Cédric Pontet
Software Architect /
CSM
cpontet@agilepartner.net
@cpontet

