Application Delivery Fundamentals 2.0 B: Java

Domain Driven Design

API Design

Goals

- What is Domain Driven Design
- The Challenges of Creating Software for Complex Problem Domains
- How the Patterns of Domain-Driven Design Manage Complexity
- The Practices and Principles of Domain-Driven Design
- Popular Misconceptions of Domain-Driven Design
-

API Design

Goals

- Knowledge Crunching and Collaboration
- Gaining Domain Insight with Domain Experts
- Patterns for Effective Knowledge Crunching
- What Is a Domain Model?
- Model-Driven Design
- Using a Ubiquitous Language to Bind the Analysis to the Code Model

API Design

Goals

- Collaborating on a Ubiquitous Language
- The Domain Layer
- Maintaining The Integrity Of Domain Models With Bounded Contexts
- Context Mapping
- Application Architecture
- Applying The Principles, Pra Ctices, And Patterns Of DDD

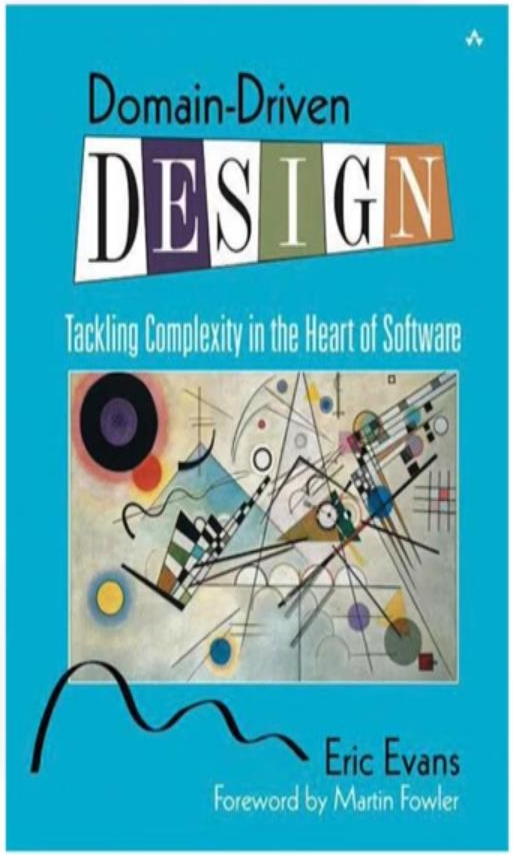
API Design

Goals

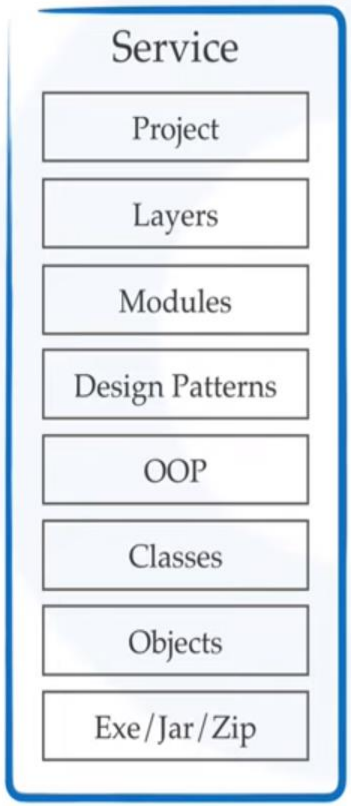
- Entities
- Domain Services
- Domain Events
- Aggregates

What is Domain Driven Design?

- Domain Driven Design is a way of looking at software from top down.
- When we are developing a software our focus shouldn't be primarily on technology, it should be primarily on business or whatever activity we are trying to assist with the software, the domain.
- Specifically we approach that by trying to develop models of that domain and make our software conformed to that.

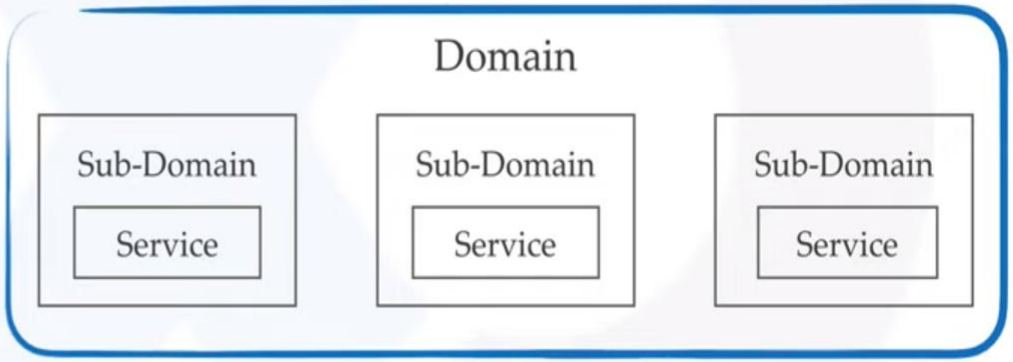


Tactical Design Tools



What does DDD give us?

Strategic Design Tools



Why should I learn DDD?

- To improve your craft
- It's not customer's job to know what they want - Steve Jobs
- Your architecture might be fabulous but its the end user ultimately who decides whether your system is fabulous or not.

Why should I learn DDD?

- To improve your craft
- It's not customer's job to know what they want - Steve Jobs
- Your architecture might be fabulous but its the end user ultimately who decides whether your system is fabulous or not.
- A system that doesn't solve the business needs is of no use to anyone, no matter how pretty it looks or how well architected its infrastructure.

Why should I learn DDD?

- To improve your craft
- It's not customer's job to know what they want - Steve Jobs
- Your architecture might be fabulous but its the end user ultimately who decides whether your system is fabulous or not.
- A system that doesn't solve the business needs is of no use to anyone, no matter how pretty it looks or how well architected its infrastructure.

'I've been a very bad girl', she said, biting her lip. 'I needed to be punished.' 'Very well', he said and installed Windows 8 on her laptop.

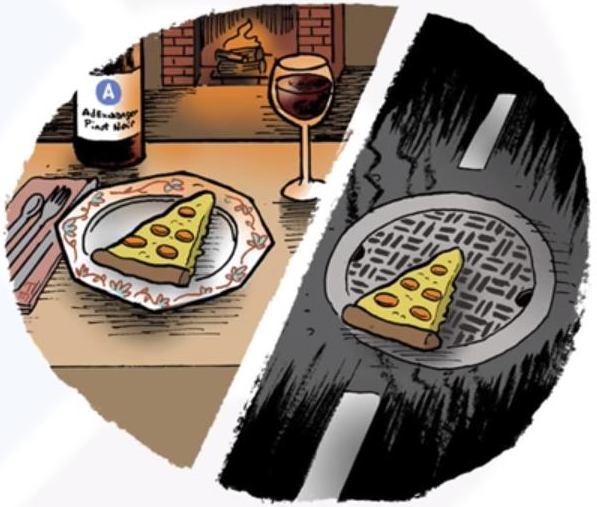
What is Strategic Design?

Object Oriented Design: Think in terms of Objects

Strategic Design: Think in terms of Contexts

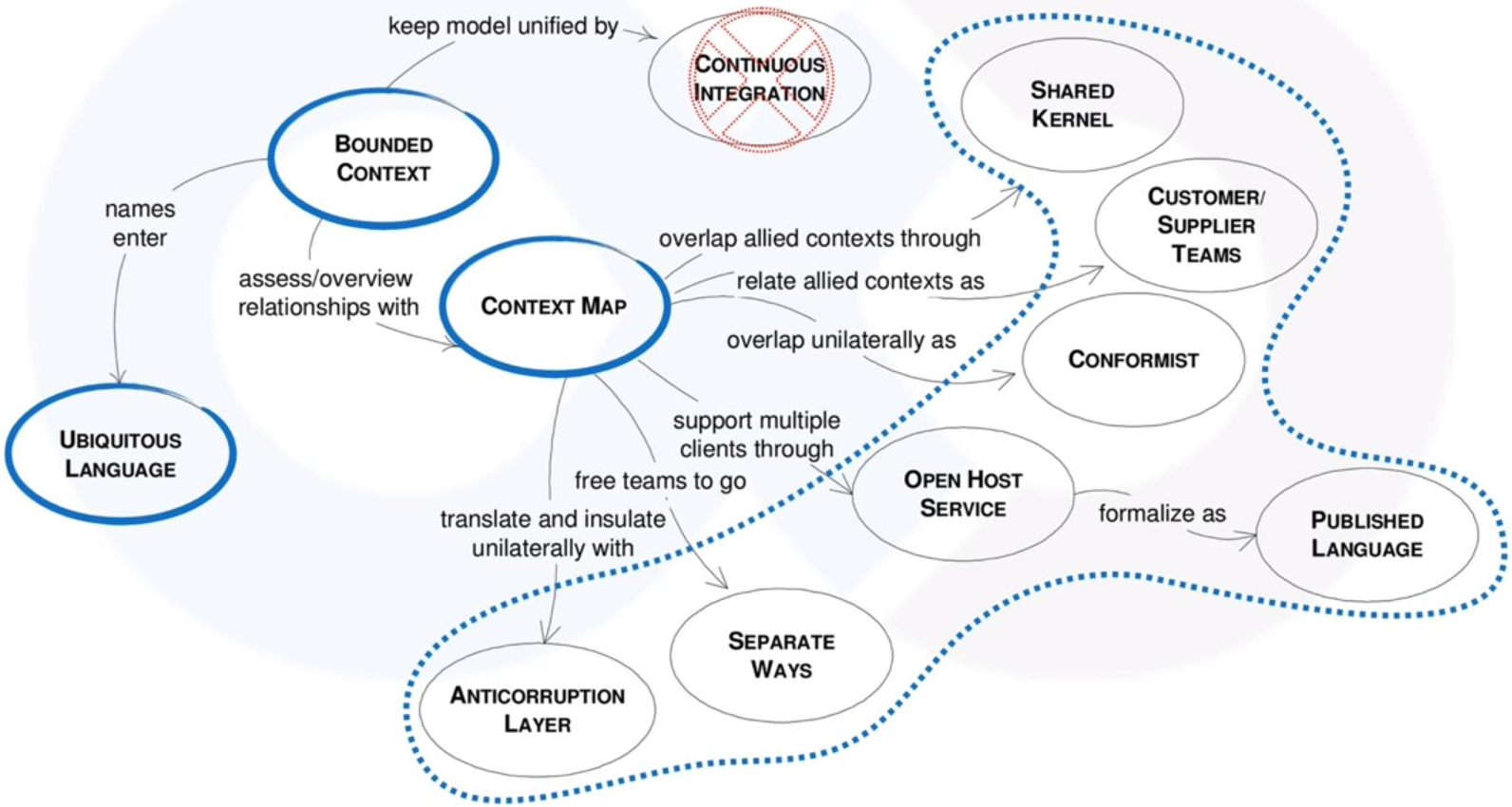
What is Context?

The setting in which a word or statement appears that determines its meaning



Context Matters

Strategic Design Tools

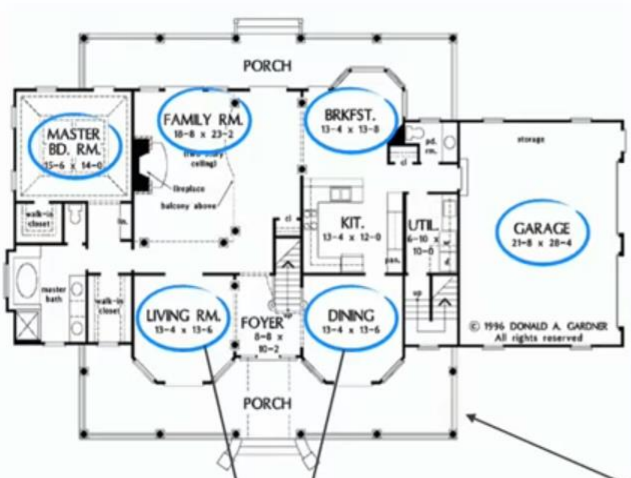


House building analogy

Build a House

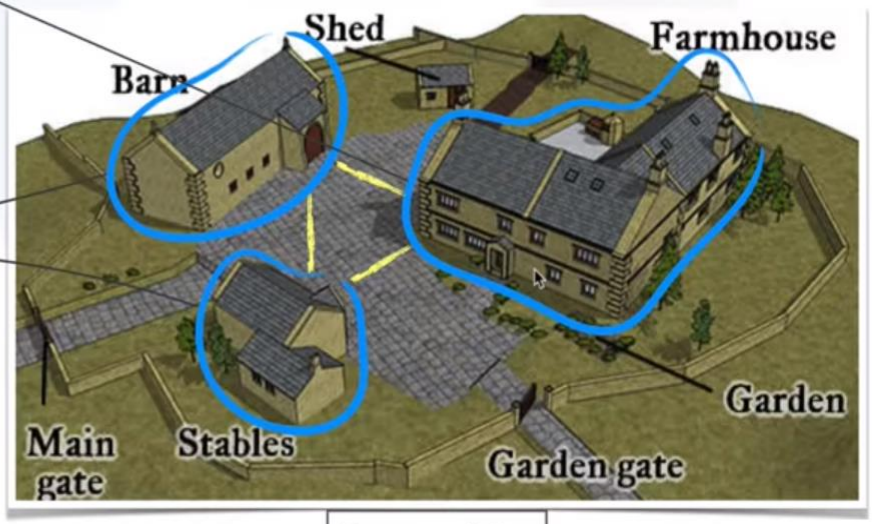
- What kind of house?
- You'll talk to the domain expert.
- You'll try to find out the core values
- You'll see what others have done

Domain Model



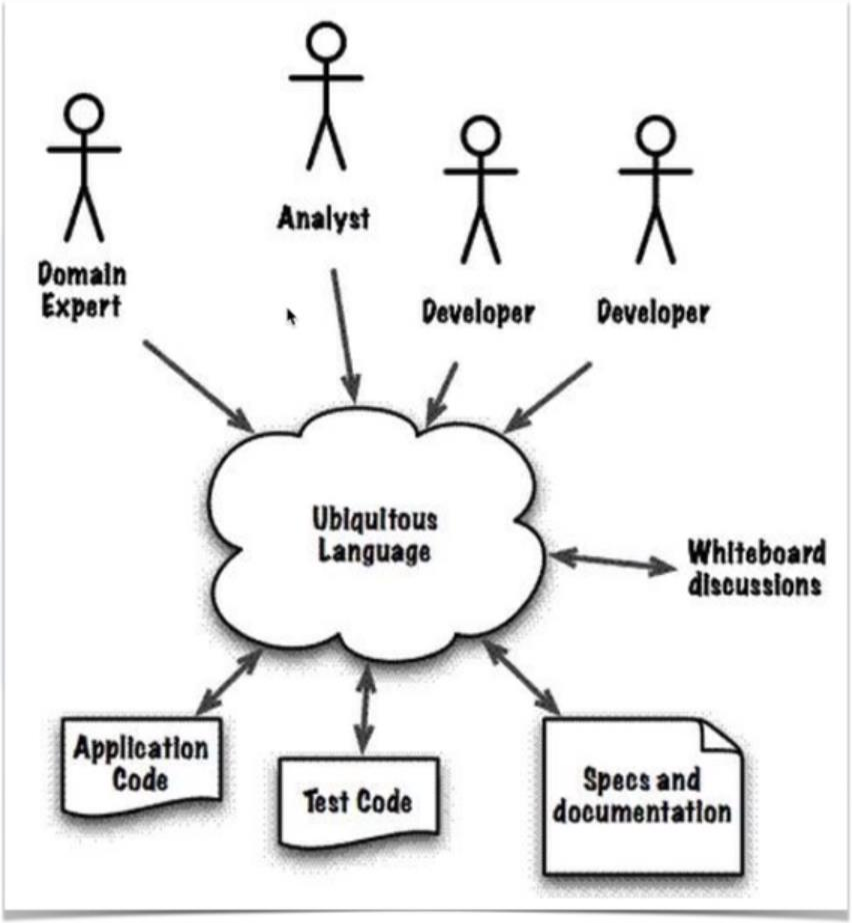
Ubiquitous Language

Bounded Context



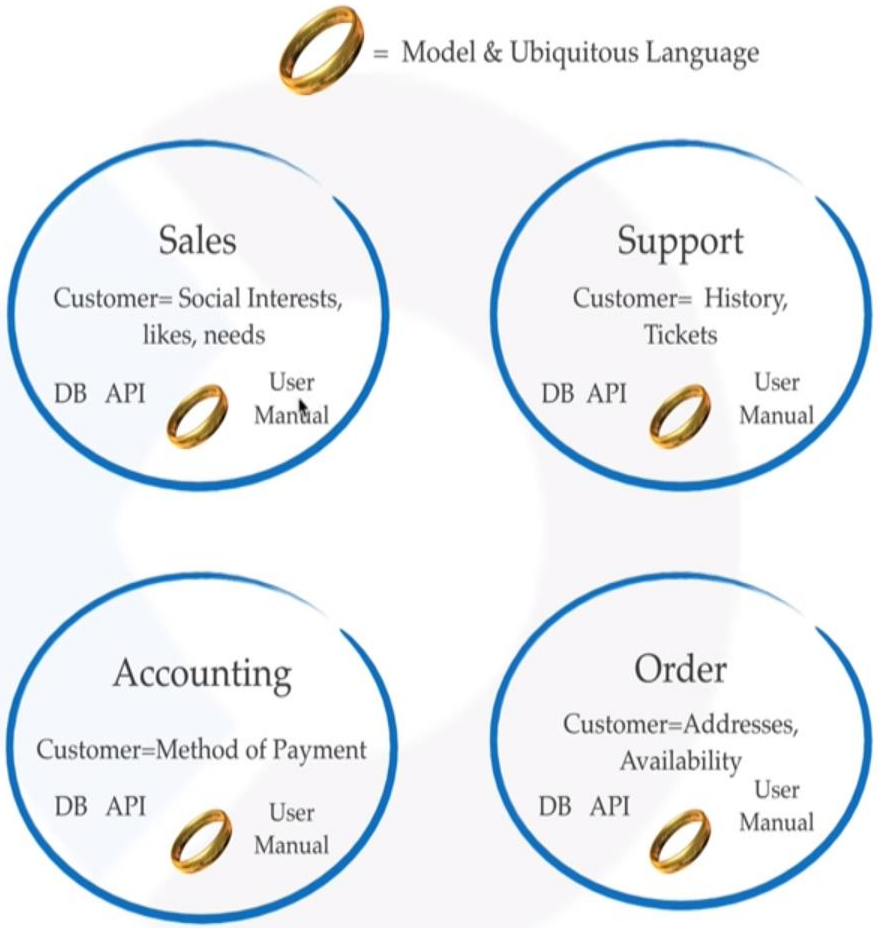
Context Map

Ubiquitous Language



e-commerce example

Bounded Context

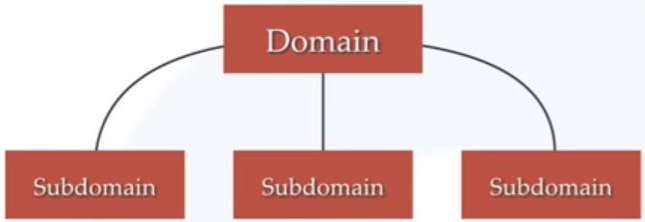


e-commerce example

Bounded Context

Context Map

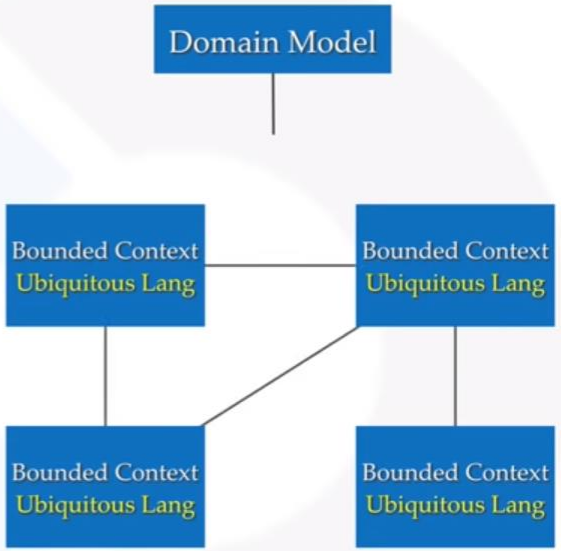
Problem Space



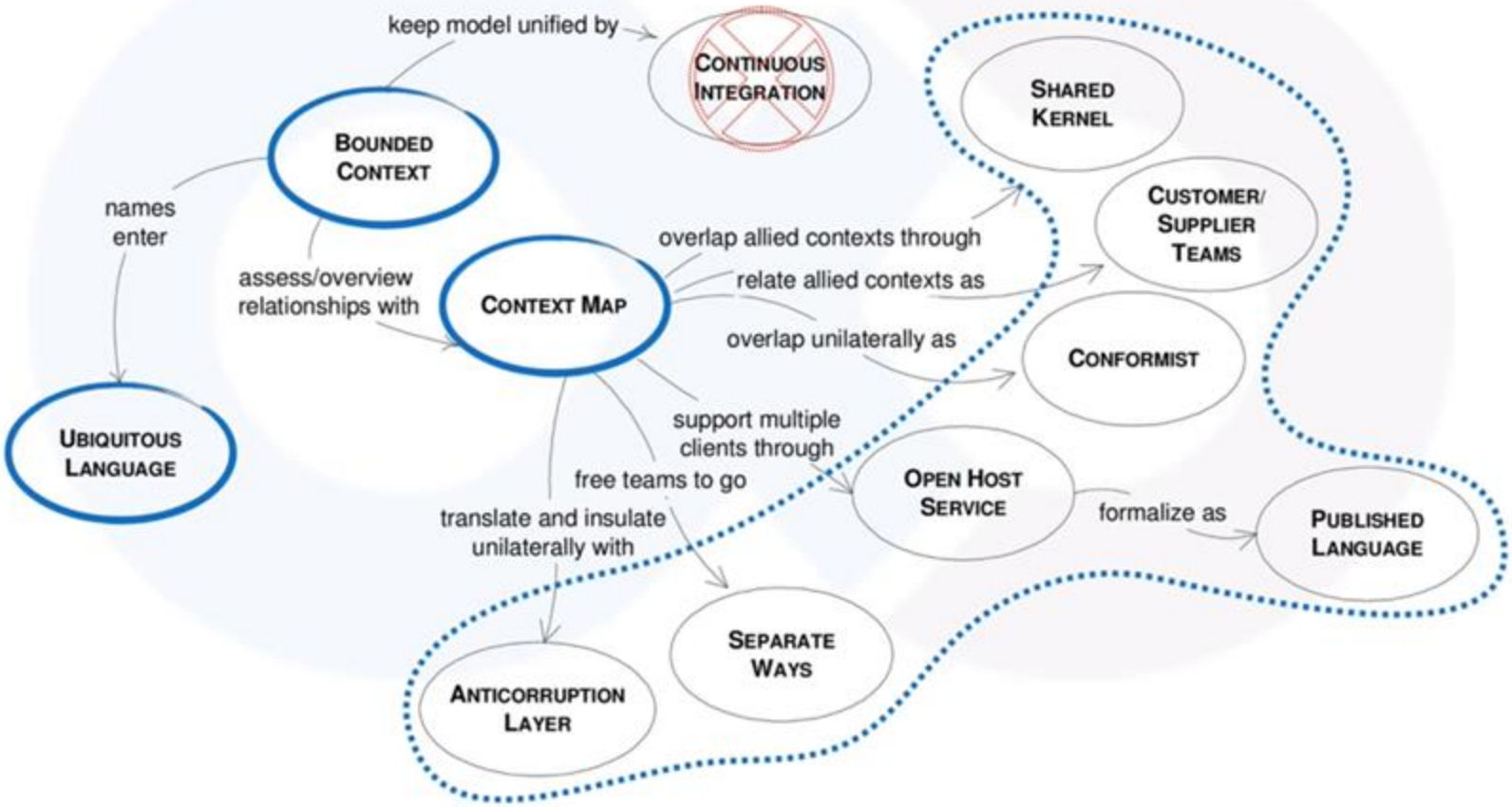
Type of Subdomains

- Core Sub-Domain
- Supporting Sub-Domain
- Generic Subdomains

Solution Space



Strategic Design Tools



Context Relationship Types

Published Language: The interacting BCs agree on a common a language (for example a bunch of XML schemas over an enterprise service bus) by which they can interact with each other

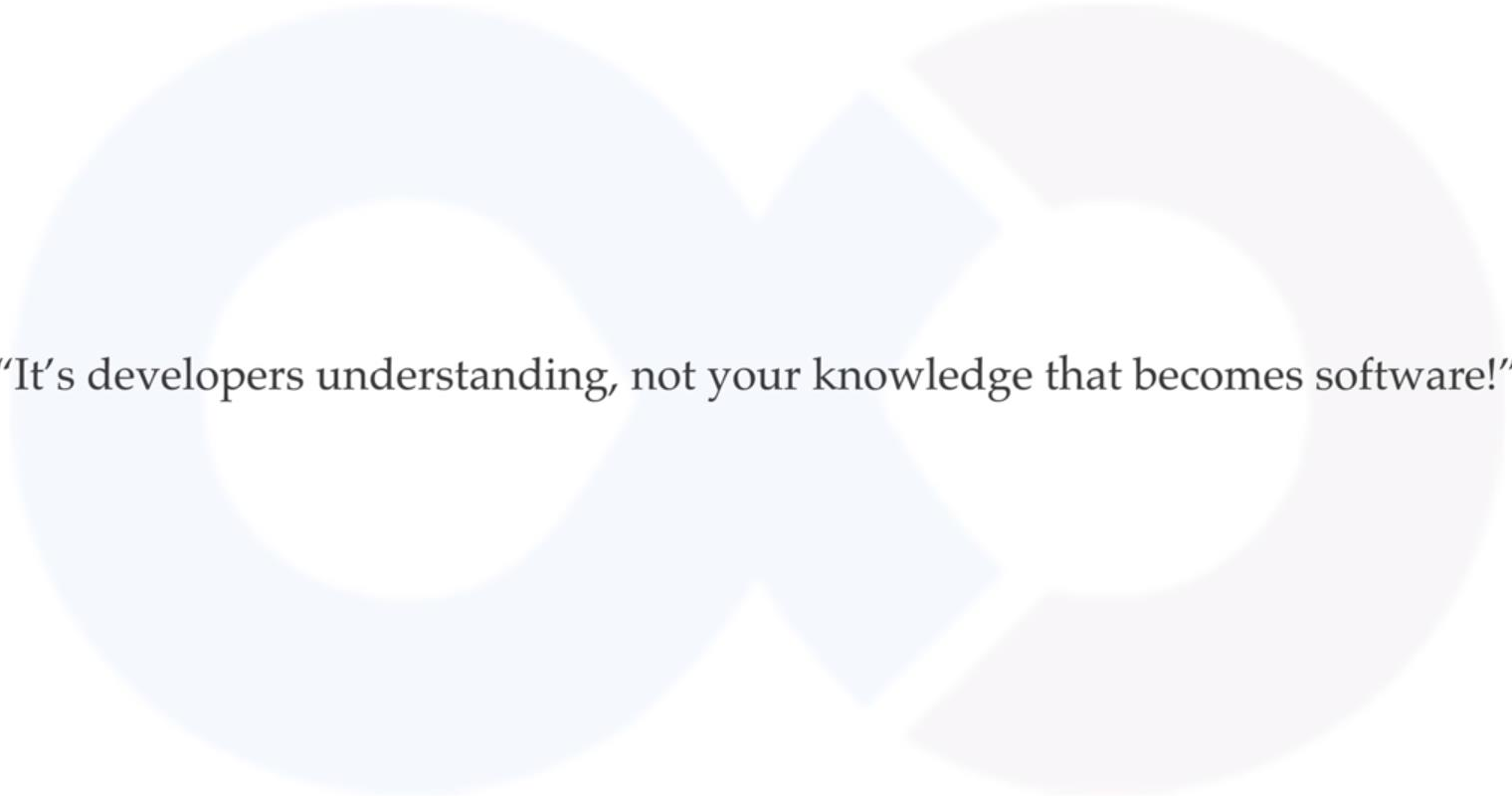
Open Host Service: BC specifies a protocol (for e.g a RESTful web service) by which any other BC can use its services

Shared Kernel: Two BCs use a common kernel of code (for example a library) as a common lingua-franca, but otherwise do their other stuff in their own specific way

Customer/Supplier: One BC uses the services of another and is a stakeholder (customer) of that other BC. As such it can influence the services provided by that BC

Conformist: One BC uses the services of another but is not a stakeholder to that other BC. As such it uses "as-is" (conforms to) the protocols or APIs provided by that BC

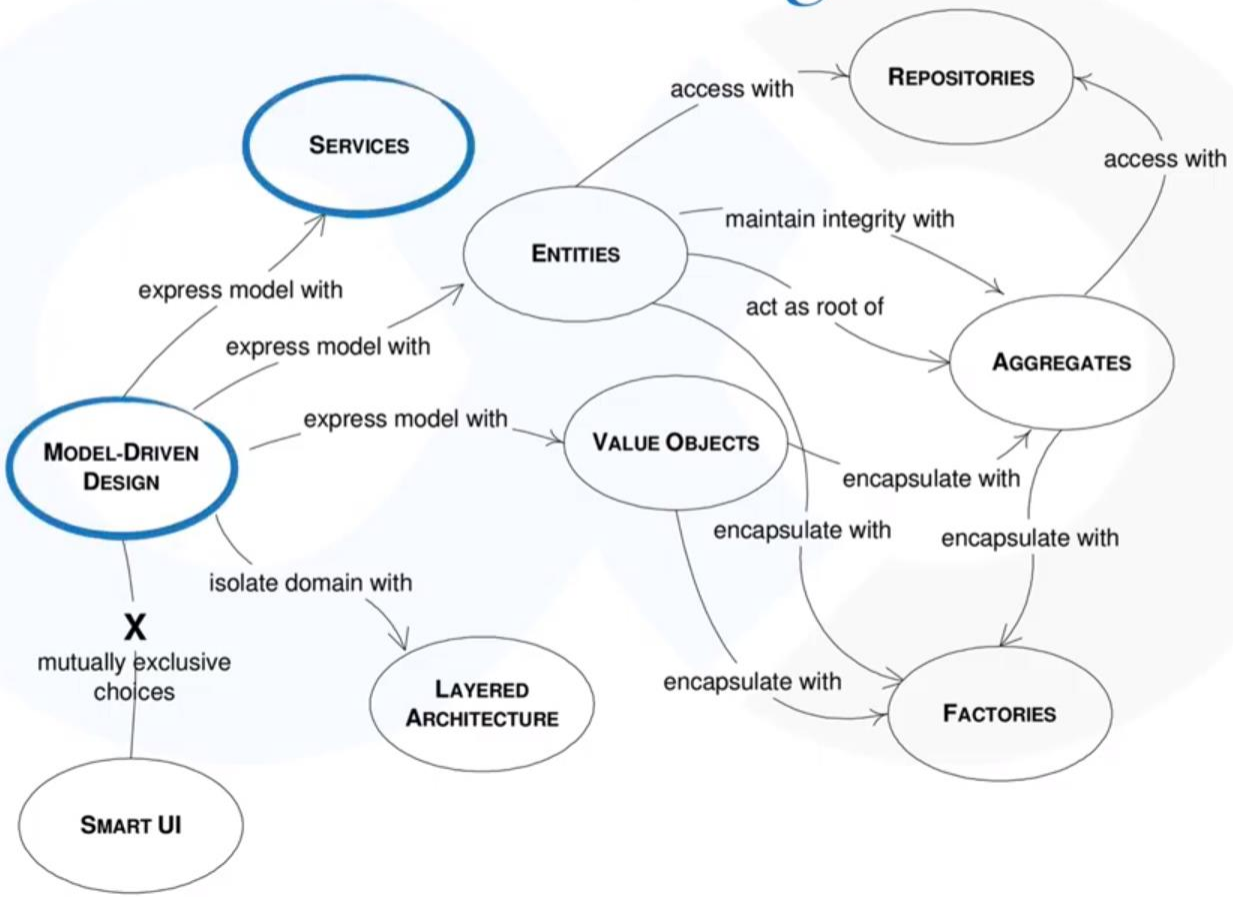
Anti-Corruption Layer: One BC uses the services of another and is not a stakeholder, but aims to minimize impact from changes in the BC it depends on by introducing a set of adapters.


“It’s developers understanding, not your knowledge that becomes software!”

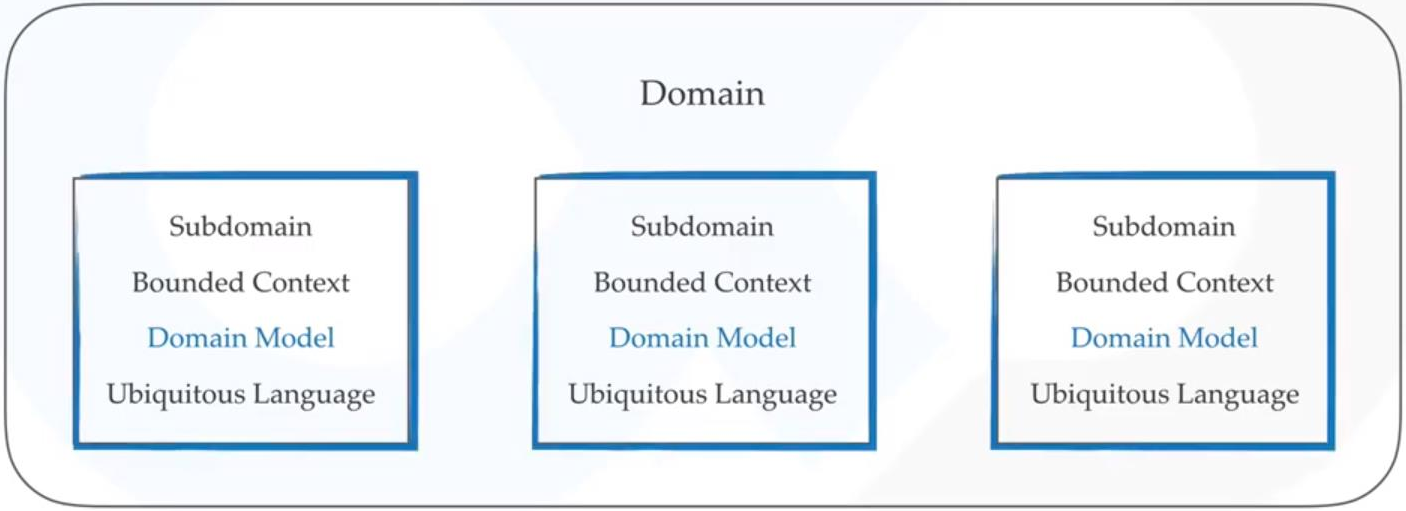
What are Tactical Design Tools?

- Tactical design tools are concerned with implementation details
- Generally takes care of components inside a bounded context
- A de facto standard in development world
- You might have heard or used things like services, entities, repositories, factories, they've all been made popular by DDD
- Contrary to strategic design, tactical design is expected to change during product development.

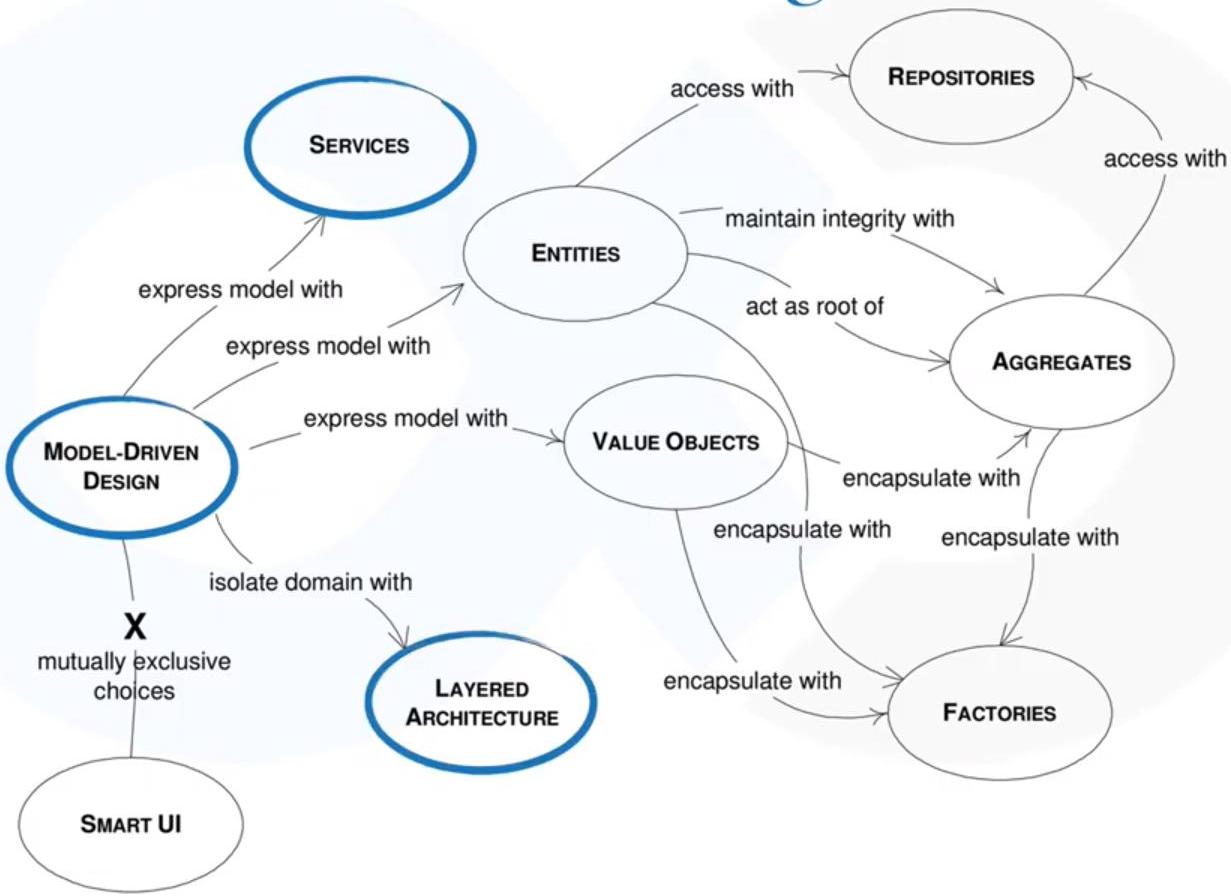
Tactical Design



Model Driven Design

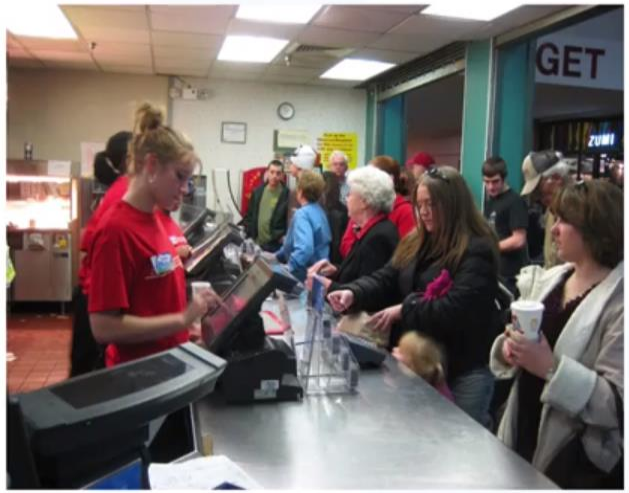


Tactical Design



Layered Architecture

McDonalds analogy



Request Handlers

Controllers

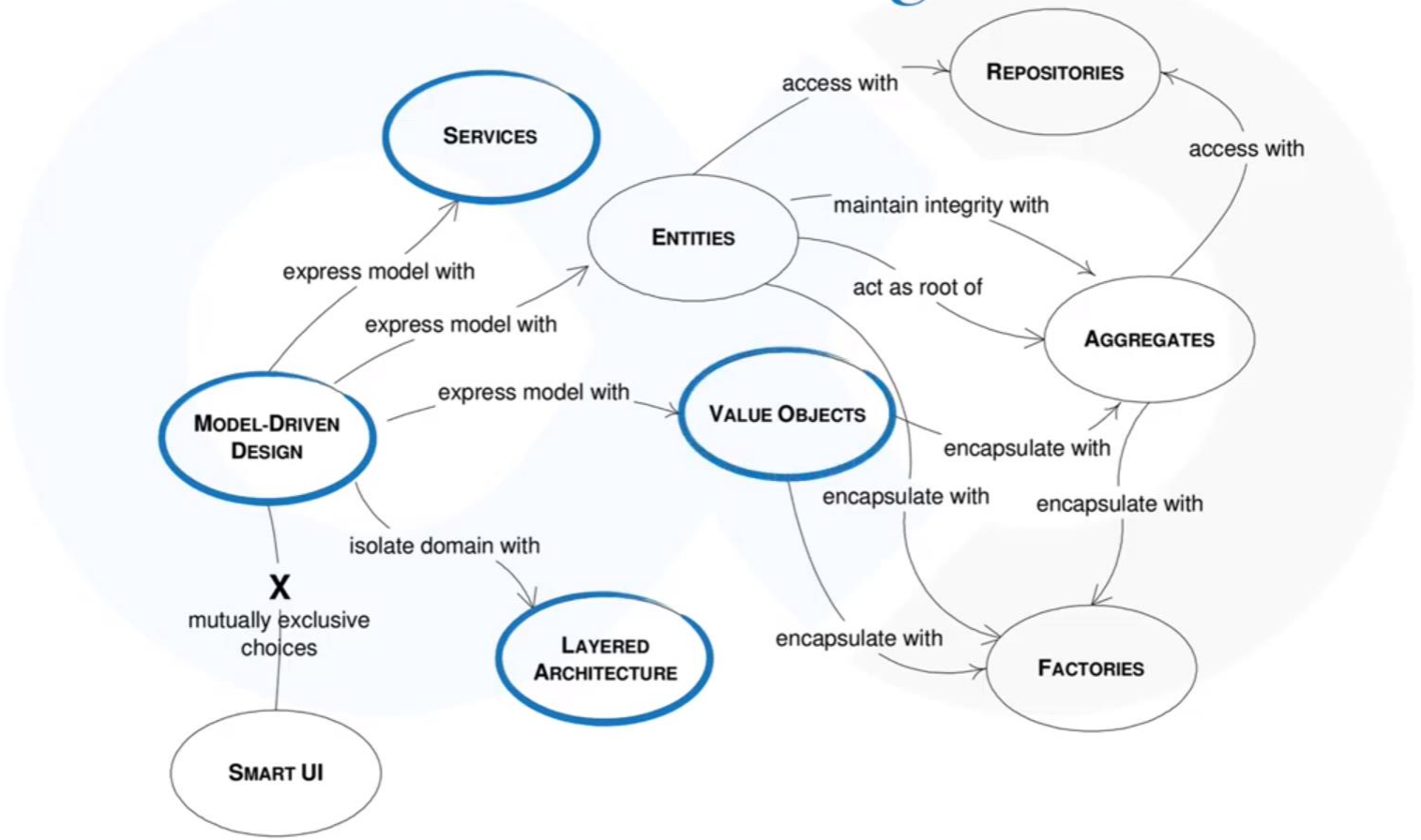
Business

Persistence

Advantages

- You can accept requests faster
- Everything is organised and well defined
- Increases flexibility, maintainability.
- Reusable components

Tactical Design



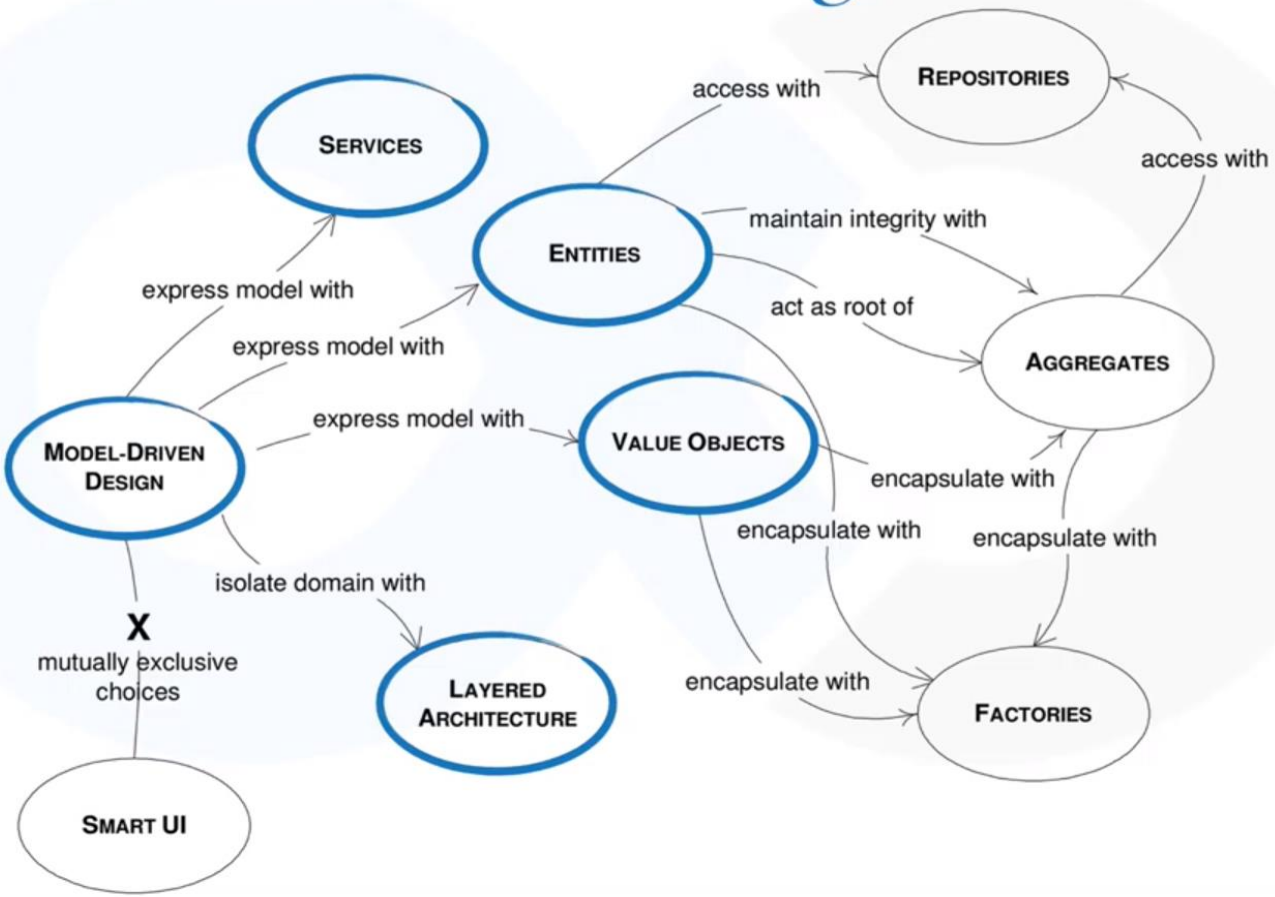
Value Objects

- One of the best things about good design
- Eg. What is a String?
- It's a general purpose value object designed to handle complexities of char arrays
- Value objects reduces complexity and forces ubiquitous language

Features of a Value Object?

- Don't care about uniqueness
- Always immutable
- Rich domain logic
- Auto-validating
- Strong equality
- Thread safe

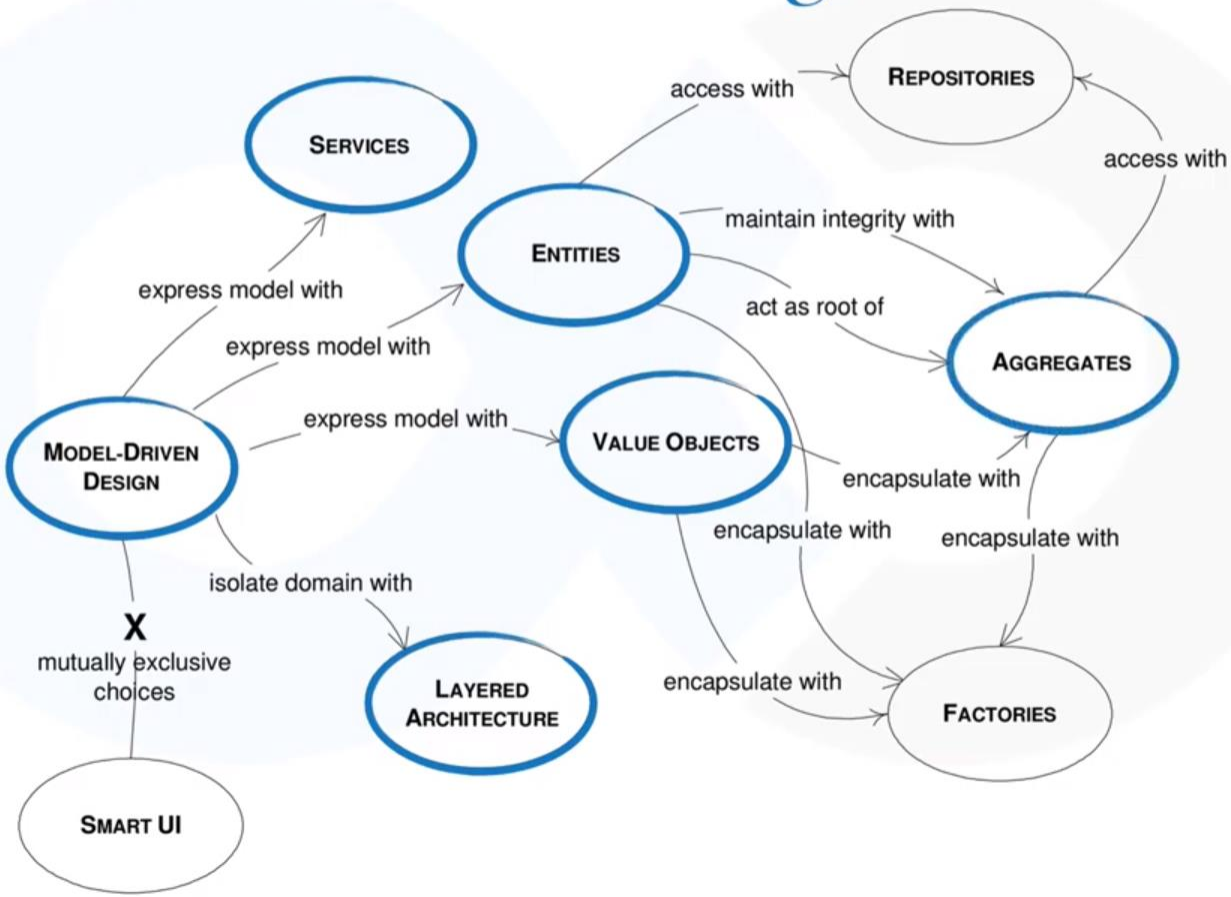
Tactical Design



Entities

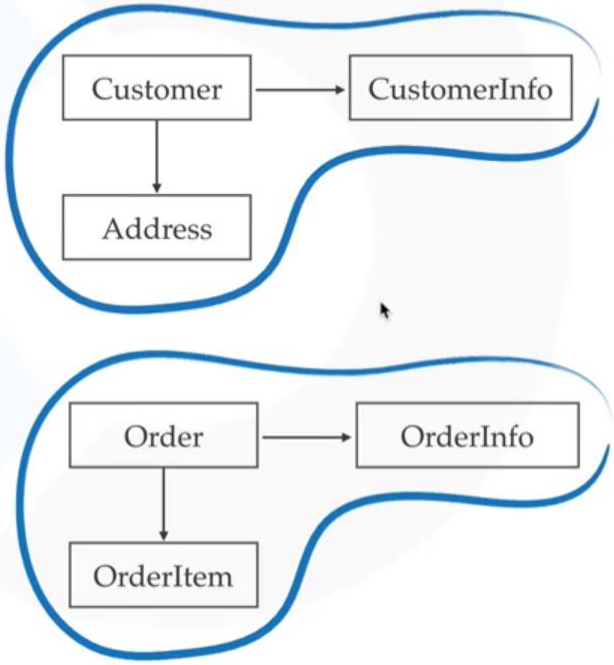
- Can be uniquely identified using an ID
- Consists of value objects
- Generally persisted as a row in DB
- Typically mutable
- Generally implements some business logic

Tactical Design

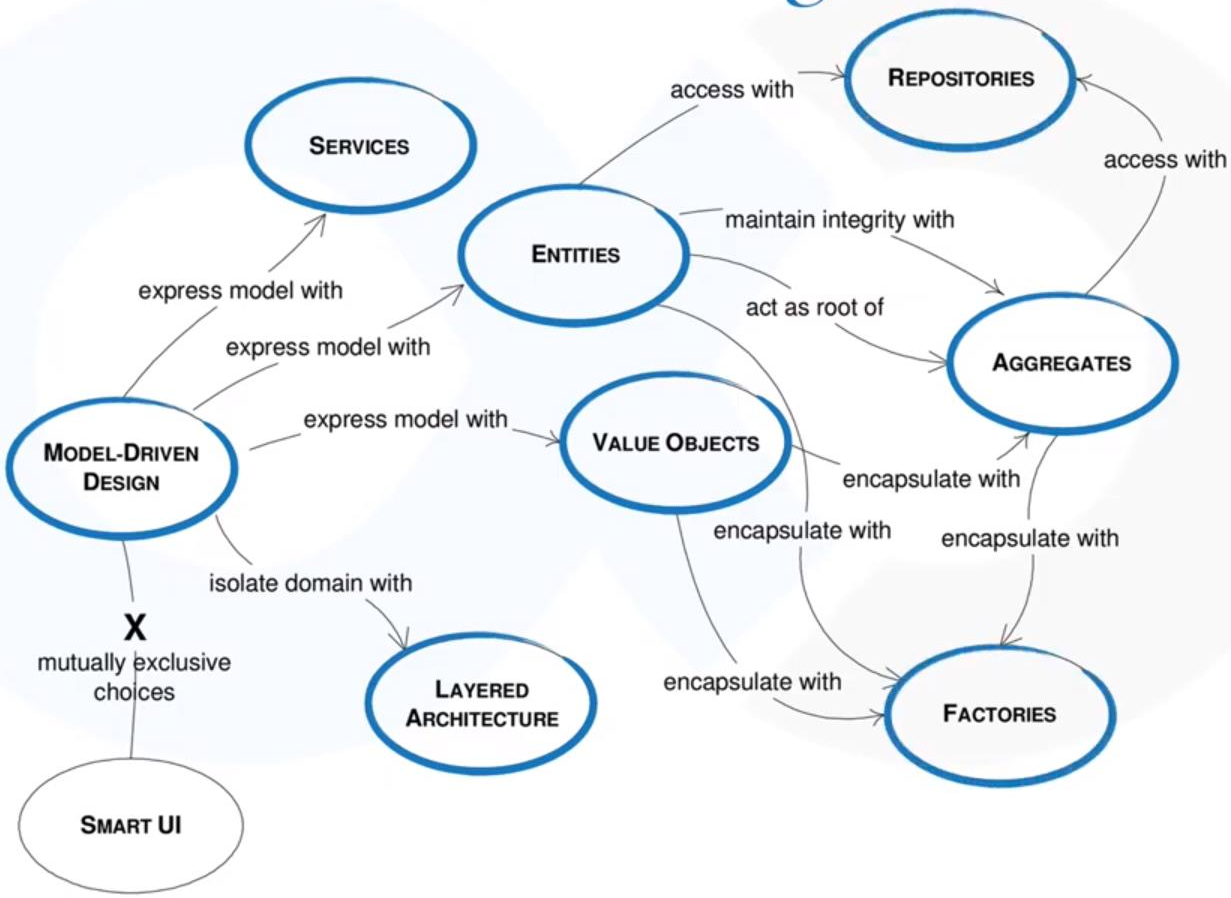


Aggregates

- When the object graph becomes big it becomes difficult to maintain
- An aggregate is collection of entities and values which comes under a single transaction boundary
- An aggregate controls the change
- An aggregate always has a root entity
- The root entity governs the lifetime of other entities in the aggregate
- An aggregate is always consistent
- Domain events are generated to ensure eventual consistency



Tactical Design



Factories & Repositories

Entities & Values

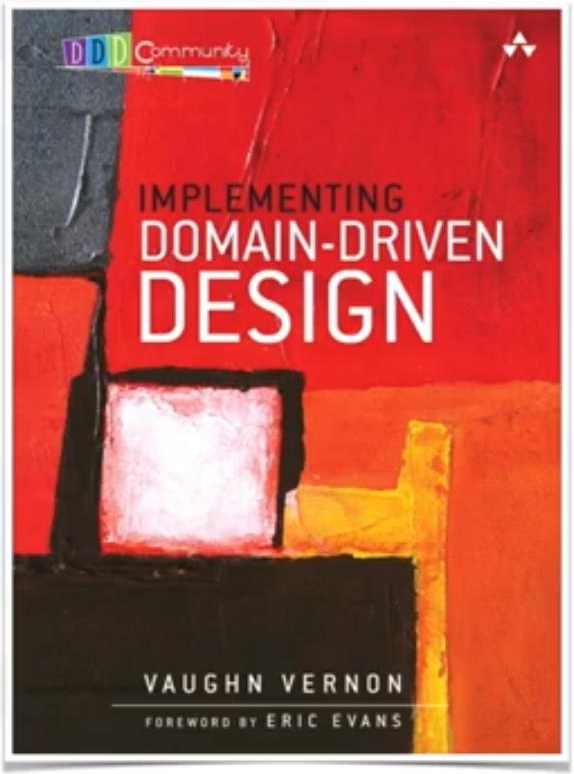
Aggregate

- Factories helps you create new aggregates
- Repository helps you get persisted aggregates
- Repository ≠ DAO

How to implement DDD?

Implementing DDD

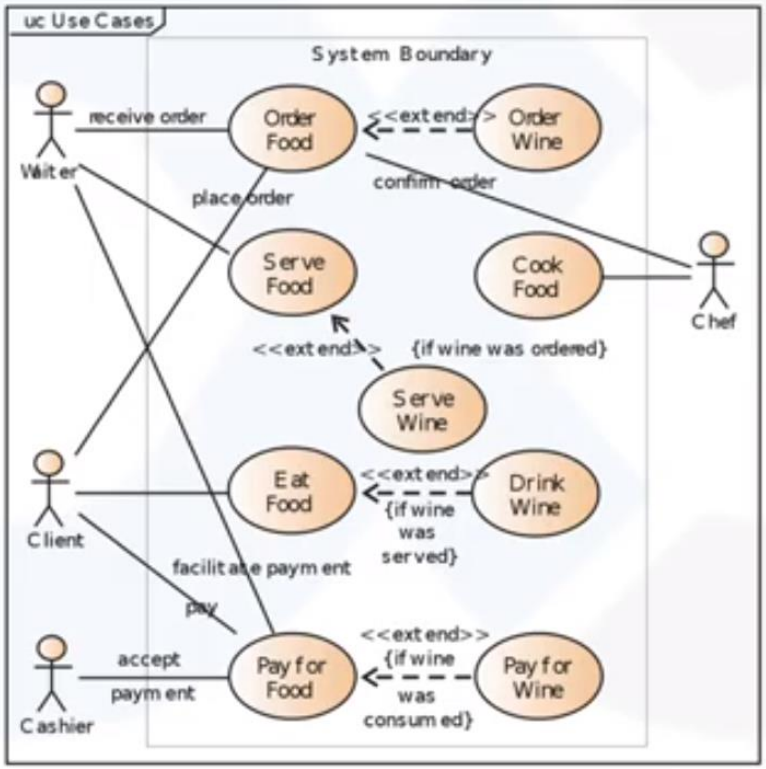
- A popular book by Vaughn Vernon
- Relatively new, published in 2013
- Next version published in 2016 known as "DDD Distilled"
- Doesn't add anything new, but adds good examples on how to implement DDD



"Building a software is a learning process, working code is a side effect"

– *Alberto Brandolini*

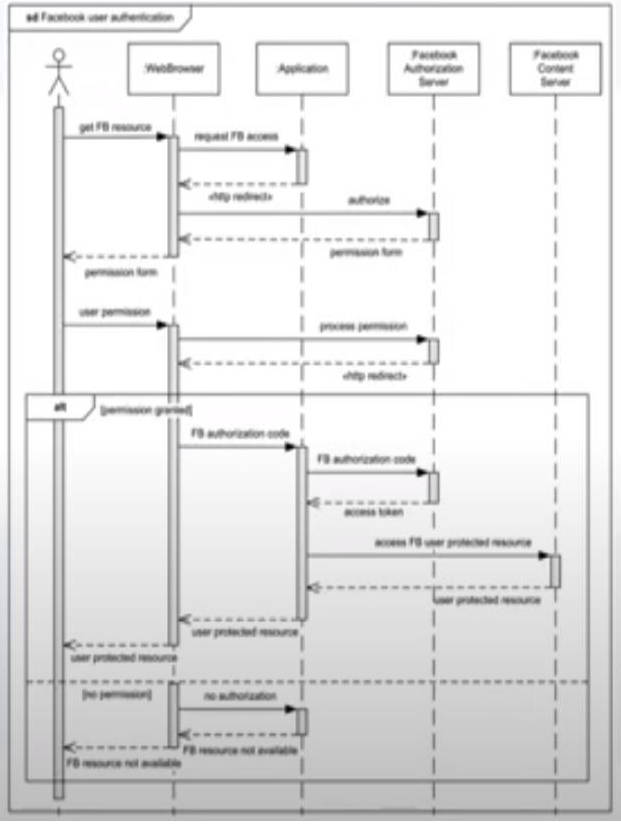
How would you discuss your understanding with domain expert?



Use Case Diagram?

Obsolete

How would you discuss your understanding with domain expert?

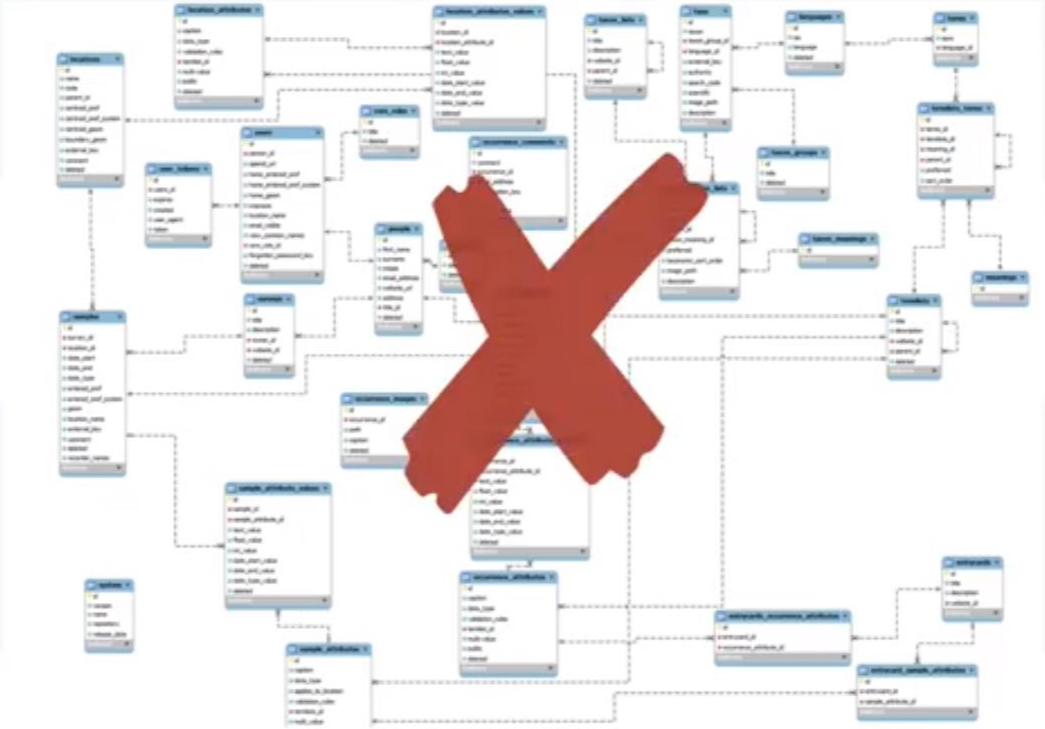


Obsolete

Sequence Diagram

Obsolete

How would you discuss your understanding with domain expert?



ER Diagram?

Event Storming

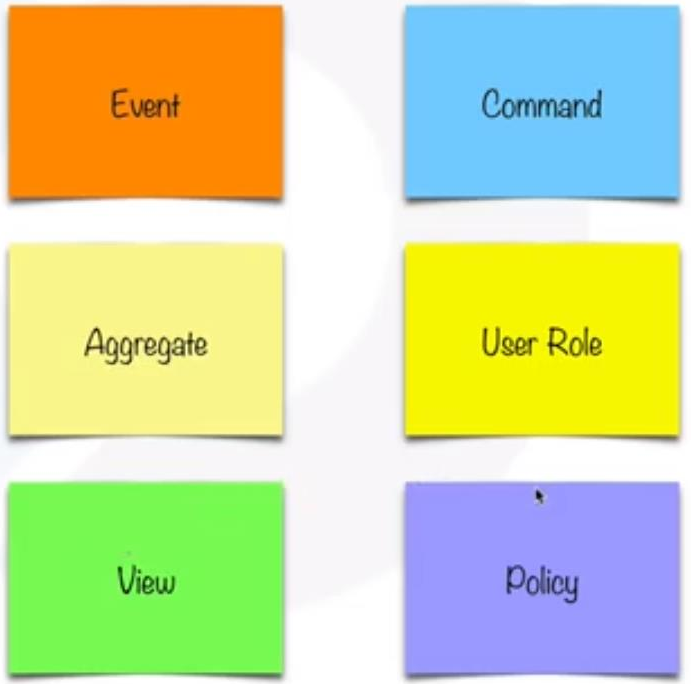
- An exercise for creating Domain Models for strategic design
- Becoming very popular in recent times
- In simple words it is a brain storming workshop among domain experts and technology people to understand the events in a system
- It is aimed to achieve a common understanding of the domain in which software must operate

How does event storming work?

- First bring the right people in

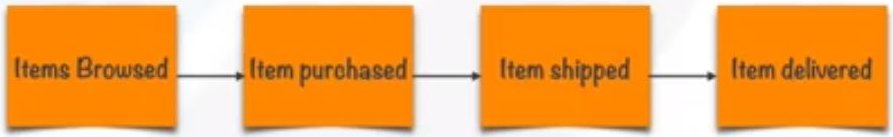
How does event storming work?

- First bring the right people in
- Then take sticky notes and colour code them to events, commands, policies, processes, errors, roles and aggregates etc



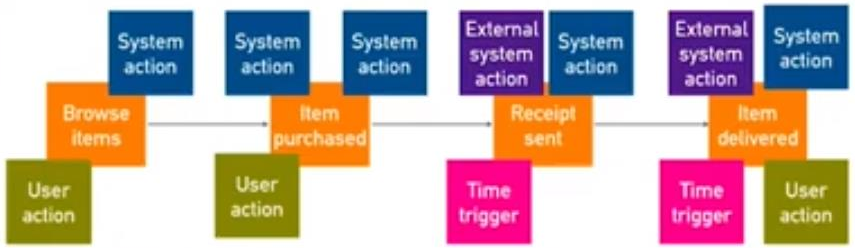
How does event storming work?

- First bring the right people in
- Then take sticky notes and colour code them to events, commands, policies, processes, errors, roles and aggregates etc
- Take a white board and start writing interesting events in sequence



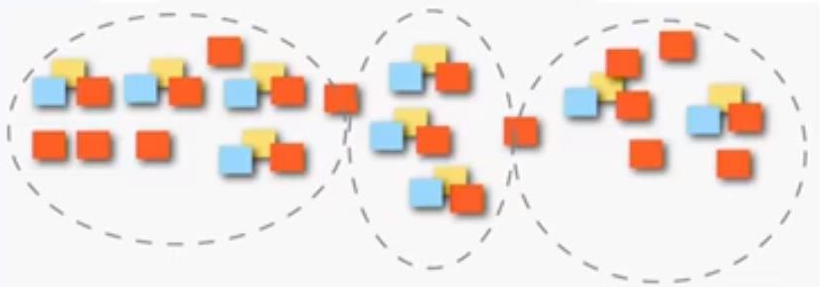
How does event storming work?

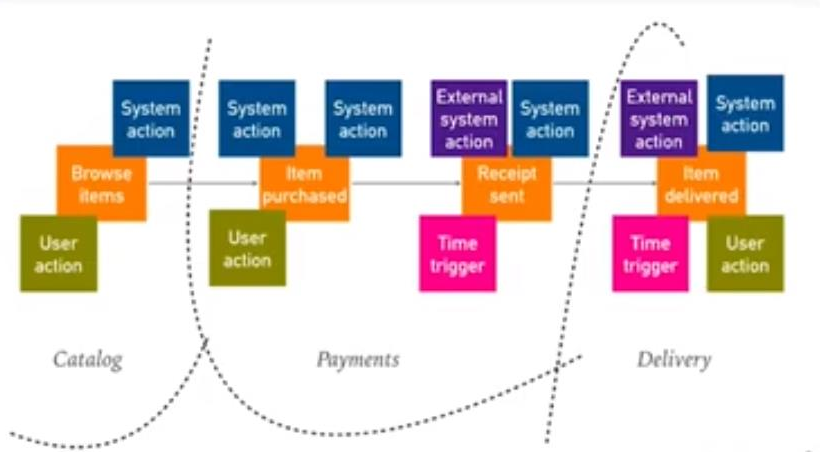
- First bring the right people in
- Then take sticky notes and colour code them to events, commands, policies, processes, errors, roles and aggregates etc
- Take a white board and start writing interesting events in sequence
- Then start adding commands, aggregates, policies etc



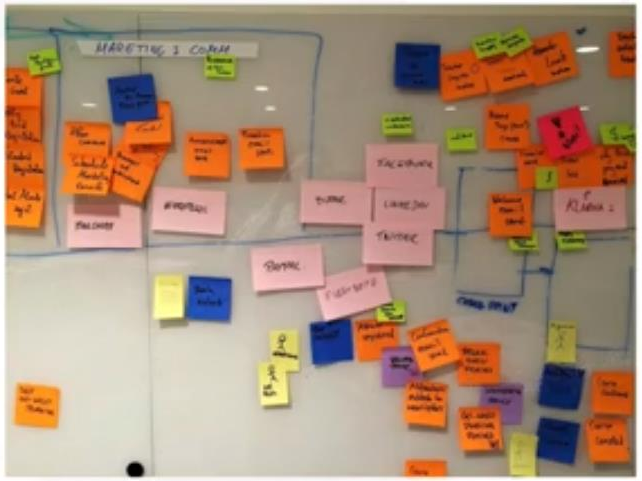
How does event storming work?

- First bring the right people in
- Then take sticky notes and colour code them to events, commands, policies, processes, errors, roles and aggregates etc
- Take a white board and start writing interesting events in sequence
- Then start adding commands, aggregates, policies etc
- After the first pass, start identifying bounded contexts





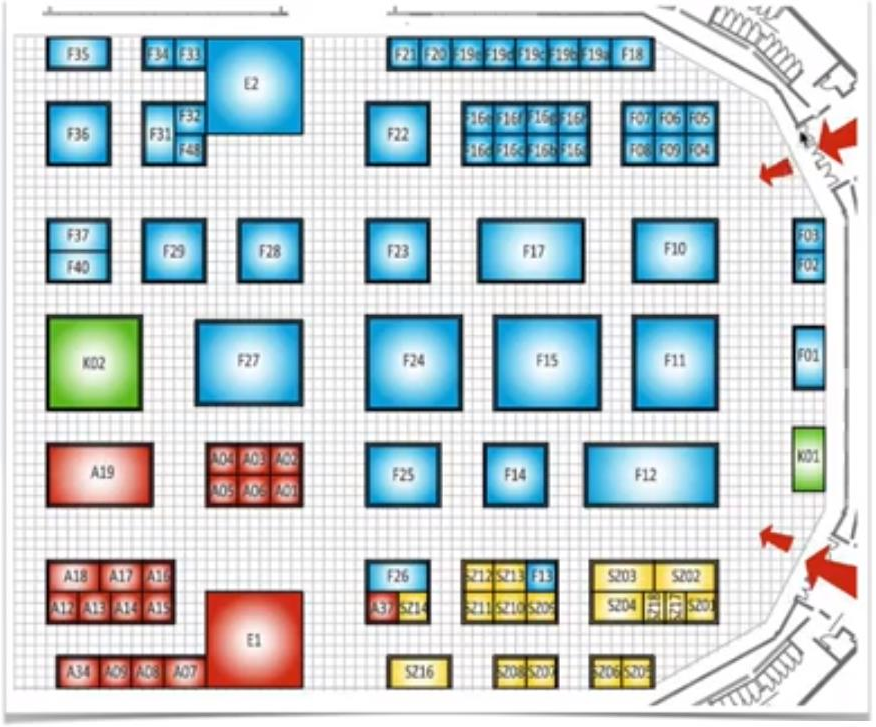
Real life examples



Trade Fair Analogy

Suppose we only check entry ticket at main gates and do profession checking in every room:

- Inefficient
- Insecure
- Complex
- Hard to maintain



Solution:

- Instead do profession checking at the gate and wrap coloured bands to each person's wrist

Value Objects aka Domain Primitives

```
public class User {
  String name;
  String email;
  String mobile;
  ...
}
```

→

```
public class User {
  Name name;
  Email email;
  PhoneNumber mobile;
  ...
}
```

```
public class Product {
  int rating;
  int quantity;
  String title;
  String description;
  BigDecimal price;
  ...
}
```

→

```
public class Product {
  Rating rating;
  Quantity quantity;
  Title title;
  Description description;
  Money price;
  ...
}
```

Implementing DDD in code

```
public class ShippingController extends Controller{
  private CartService cartService;

  public void process(ShippingForm form) throws Exception, SQLException{
    CartBean cartBean = cartService.getCart(form.getCartId());
    BigDecimal cost = null;

    if (cartBean.getTotal() <= 100){
      cost = new BigDecimal(4.99);
      if (form.getOption() == 1) {
        for (Item i : cartBean.getItems()) {
          if (i.getCat() == 'B') {
            cost.add(new BigDecimal(2.99));
          }
          if (i.getCat() == 'O') {
            cost.add(new BigDecimal(i.getWeight() / 1000)
                .multiply(new BigDecimal(2.99)));
          }
        }
      }
      if (form.getOption() == 2) {
        for (Item i : cartBean.getItems()) {
          if (i.getCat() == 'B') {
            cost.add(new BigDecimal(4.99));
          }
          if (i.getCat() == 'O') {
            cost.add(new BigDecimal(i.getWeight() / 1000)
                .multiply(new BigDecimal(2.99)));
          }
        }
      }
    }
```

Implementing DDD in code

```
public class ShippingController extends Controller{
  private CartService cartService;

  public void process(ShippingForm form) throws Exception, SQLException{
    CartBean cartBean = cartService.getCart(form.getCartId());
    BigDecimal cost = null;

    if (cartBean.getTotal() <= 100){
      cost = new BigDecimal(4.99);
      if (form.getOption() == 1) {
        for (Item i : cartBean.getItems()) {
          if (i.getCat() == 'B') {
            cost.add(new BigDecimal(2.99));
          }
          if (i.getCat() == 'D') {
            cost.add(new BigDecimal(i.getWeight() / 1000)
                .multiply(new BigDecimal(2.99)));
          }
        }
      }
      if (form.getOption() == 2) {
        for (Item i : cartBean.getItems()) {
          if (i.getCat() == 'B') {
            cost.add(new BigDecimal(4.99));
          }
          if (i.getCat() == 'D') {
            cost.add(new BigDecimal(i.getWeight() / 1000)
                .multiply(new BigDecimal(2.99)));
          }
        }
      }
    }
    form.setCost(cost.setScale(0, ROUND_HALF_EVEN));
  }
}
```

- Magic Numbers
- Duplicate code
- Primitive Obsession
- Mixed Concerns
- Fuzzy Terminology

Implementing DDD in code

```
public class ShippingController extends Controller{
  private CartService cartService;

  public void process(ShippingForm form) throws Exception, SQLException{
    CartBean cartBean = cartService.getCart(form.getCartId());
    BigDecimal cost = null;

    if (cartBean.getTotal() <= 100){
      cost = new BigDecimal(4.99);
      if (form.getOption() == 1) {
        for (Item i : cartBean.getItems()) {
          if (i.getCat() == 'B') {
            cost.add(new BigDecimal(2.99));
          }
          if (i.getCat() == 'O') {
            cost.add(new BigDecimal(i.getWeight() / 1000)
                .multiply(new BigDecimal(2.99)));
          }
        }
      }
      if (form.getOption() == 2) {
        for (Item i : cartBean.getItems()) {
          if (i.getCat() == 'B') {
            cost.add(new BigDecimal(4.99));
          }
          if (i.getCat() == 'O') {
            cost.add(new BigDecimal(i.getWeight() / 1000)
                .multiply(new BigDecimal(2.99)));
          }
        }
      }
    }
    form.setCost(cost.setScale(0, ROUND_HALF_EVEN));
  }
}
```

Implicit Concerns

- Free shipping if greater than 100. 100 What?
- Fixed shipment cost?
- Currency?
- Weight in G? Variable cost of each sent KG?
- What's option? Shipping option?
- Variable cost per category?

Implementing DDD in code

```
public class ShippingController extends Controller{
  private CartService cartService;

  public void process(ShippingForm form) throws Exception, SQLException{
    CartBean cartBean = cartService.getCart(form.getCartId());
    BigDecimal cost = null;

    if (cartBean.getTotal() <= 100){
      cost = new BigDecimal(4.99);
      if (form.getOption() == 1) {
        for (Item i : cartBean.getItems()) {
          if (i.getCat() == 'B') {
            cost.add(new BigDecimal(2.99));
          }
          if (i.getCat() == 'O') {
            cost.add(new BigDecimal(i.getWeight() / 1000)
                .multiply(new BigDecimal(2.99)));
          }
        }
      }
      if (form.getOption() == 2) {
        for (Item i : cartBean.getItems()) {
          if (i.getCat() == 'B') {
            cost.add(new BigDecimal(4.99));
          }
          if (i.getCat() == 'O') {
            cost.add(new BigDecimal(i.getWeight() / 1000)
                .multiply(new BigDecimal(2.99)));
          }
        }
      }
    }
    form.setCost(cost.setScale(0, ROUND_HALF_EVEN));
  }
}
```

Domain Driven Code

```
public Money calculateShippingCost(Cart cart, ShippingOption selectedOption) {
  if (cart.getTotalPrice().lessThan(THRESHOLD_FOR_FREE_SHIPPING)) {
    Money shippingCost = getBaseShipmentCost(cart);
    for (Item item : cart.getItems()) {
      if (selectedOption == REGULAR) {
        shippingCost = shippingCost.add(RegularShipping.getCostForItem(item));
      } else if (selectedOption == PRIME) {
        shippingCost = shippingCost.add(PrimeShipping.getCostForItem(item));
      }
    }
    return shippingCost;
  }
  return Money.ZERO;
}
```

What did we learn?

Designing a city analogy

Unplanned

Big Ball Of Mud

Planned

Domain Driven Design

What did we learn?

- Always design your softwares in Top-Down fashion, unless your domain is trivial
- Focus on what distinguishes you from competitors and understand what you must excel at
- Always focus all your energy and resources in your core sub-domain
- Don't chase shiny objects by throwing technology at business problems
- Place emphasis on naming and operations

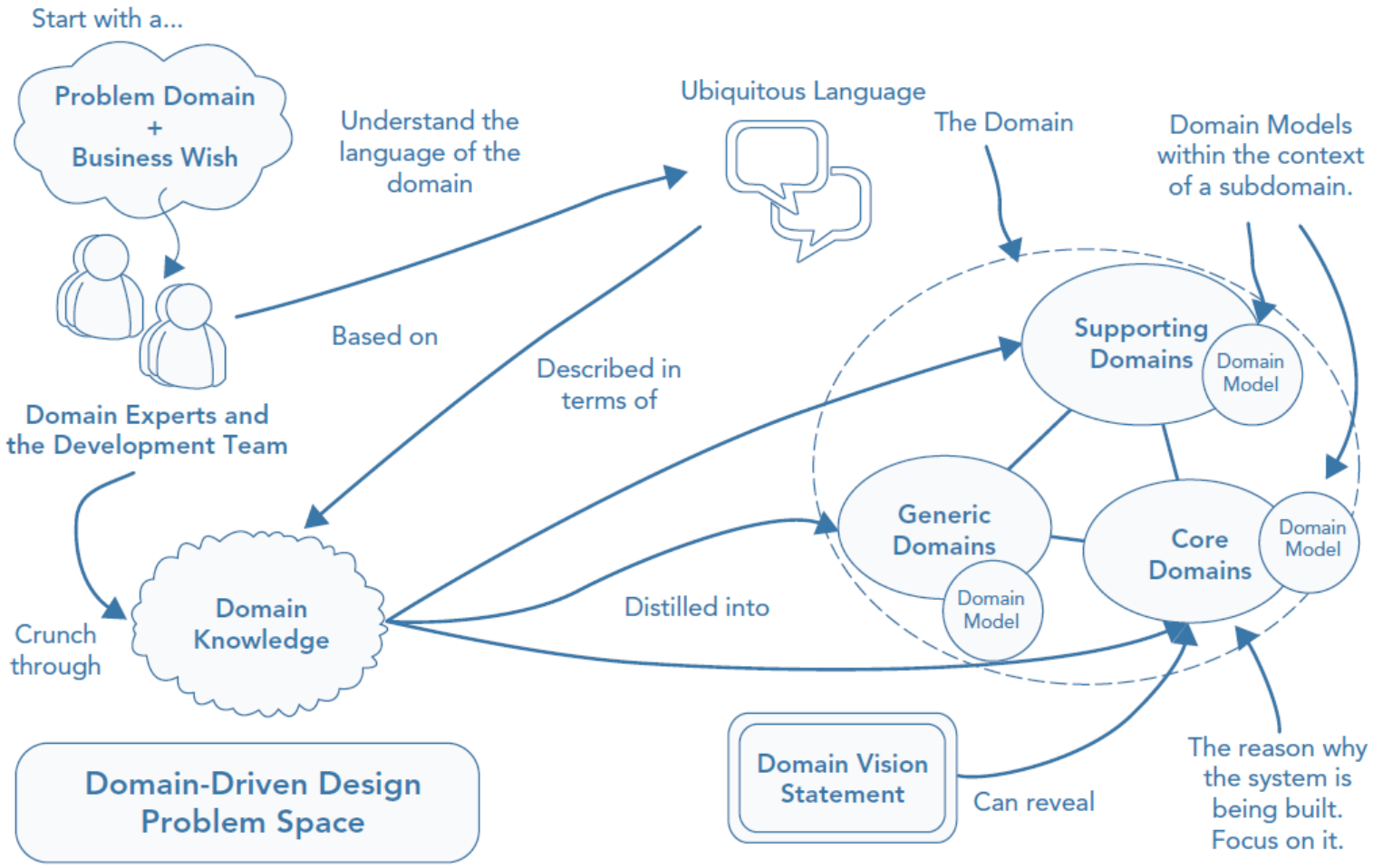
Important tips

- Don't work for a company that thinks of software as a cost centre
- DDD will look just like OOP when you get the hang of it
- Don't create heavy weight blueprints before development, follow an evolutionary approach
- Write your code and documentation so that it represents the domain
- Build a product for which you can provide long term warranties

Problem Space

- DDD emphasizes the need to focus on the business problem domain: its terminology, the core reasons behind why the software is being developed, and what success means to the business.
- The need for the development team to value domain knowledge just as much as technical expertise is vital to gain a deeper insight into the problem domain and to decompose large domains into smaller subdomains.

Problem Space



Domain Expert

Domain Expert

- Domain Expert :The domain expert is the person who has the knowledge of the system or domain for which the software has to be developed .
- They would require a product that will help them work on their projects better and hence they require Software Developers

Developer

- Developer : The developer is responsible to take in requirements from the Domain Expert and model a software according to their needs.

Domain Expert and Developer

- Domain experts and the developers together are equally responsible towards a final product when using the Domain Driven Design ideology

Clip slide

Ubiquitous Language

Clip slide

What is a **Domain Model**?

Domain == problem

Model == solution

Distilled knowledge

“The model isn’t just the knowledge in a domain expert's head; it is a rigorously organized and selective abstraction of that.”
- Eric Evans

Can be expressed in several ways:

- **Code**
- Diagrams
- Written Documentation

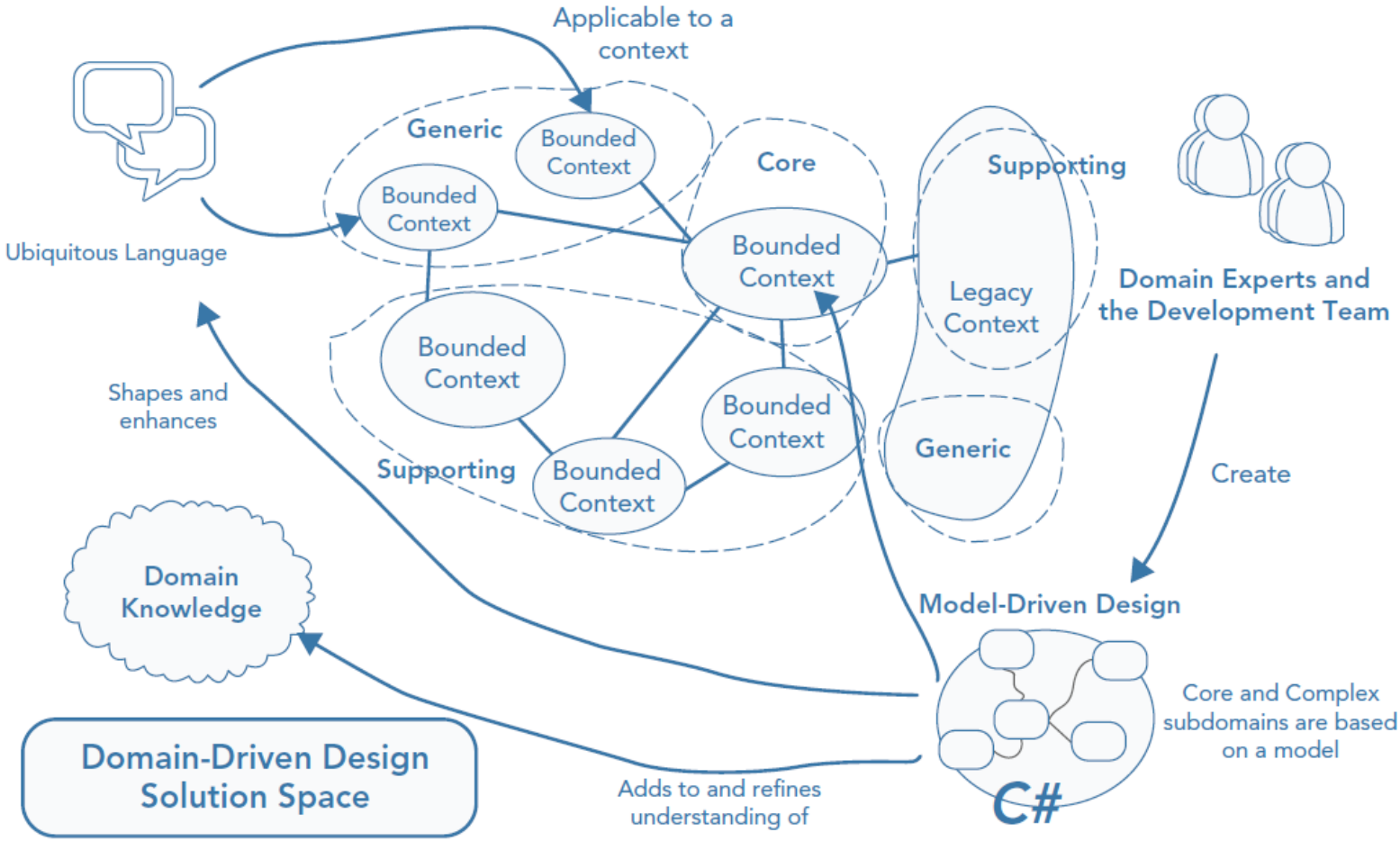
Domain Driven Design

- The Solution Space
  - When you have a sound understanding of the problem domain, strategic patterns of DDD can help you implement a technical solution in synergy with the problem space.
  - Patterns enable core parts of your system that are crucial to the success of the product to be protected from the generic areas.

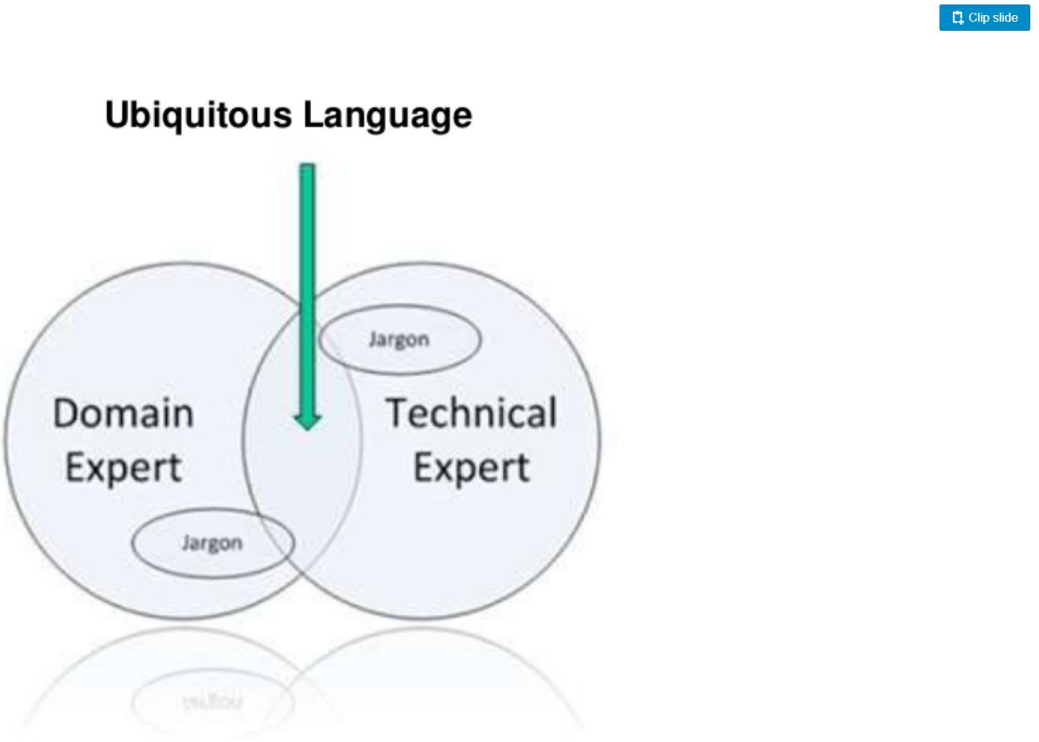
Solution Space

- Core parts of your product that are sufficiently complex or will frequently change should be based on a model.
- The tactical patterns of DDD along with Model-Driven Design will help you create a useful model of your domain in code.
- A model is the home to all of the domain logic that enables your application to fulfill business use cases.
- A model is kept separate from technical complexities to enable business rules and policies to evolve.
- A model that is in synergy with the problem domain will enable your software to be adaptable and understood by other developers

Solution Space

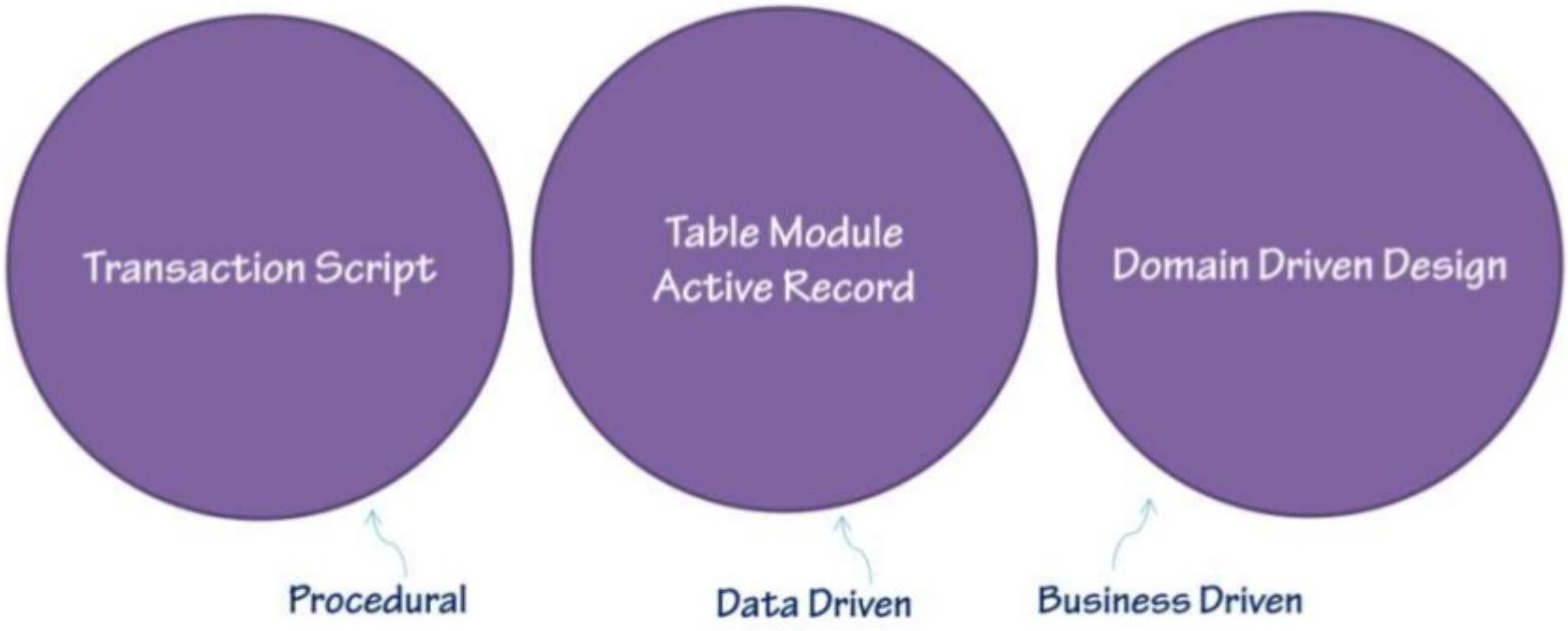


The phrase ubiquitous language was first coined in *Domain-Driven Design* by Eric Evans.
It's "the practice of building up a common, rigorous language between developers and users."
A ubiquitous language is your team's first line of defense against miscommunications.



Clip slide

What's your focus?

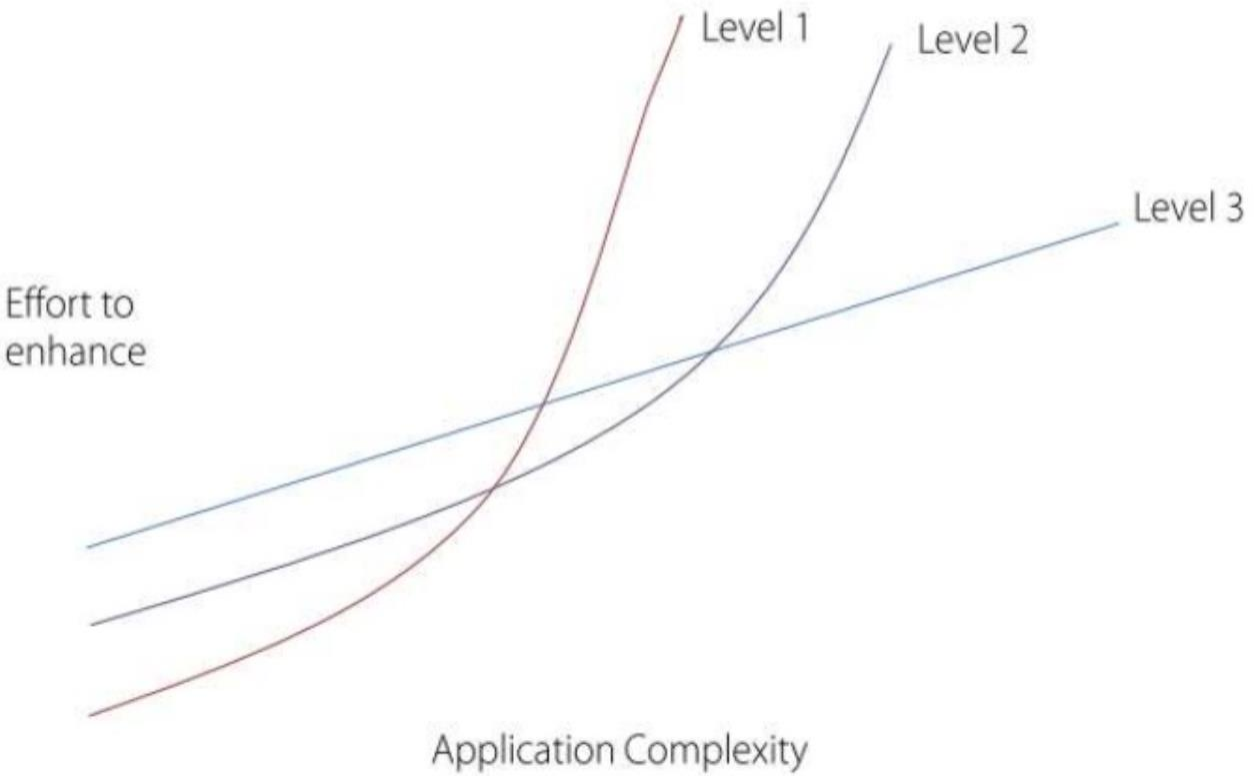


Architecture Selection

L1	L3
MVP	Flagship product
Junior team	Senior Team
Small team	Large Team
Simple Domain	Complex domain
Tight timeline	Flexible timeline
Short lifespan	Long-term
No security concerns	Security matters
Little chance for reuse	Known need for reuse

Clip slide

Choice by Complexity



Clip slide

Schedule

- ~~Basic Concepts~~
- ~~Ubiquitous Language~~
- ~~Domain Expert~~
- ~~Domain Model~~
- ~~Architectures types~~
  - ~~DDD~~,
  - ~~Smart UI~~,
  - ~~...~~
- **Strategic Design**
  - Bounded Context
  - Context Maps
  - Domain Events
  - Event Storming
- **Tactical Design** (building blocks)
  - Layered Architecture
  - Entities
  - Value Objects
  - Aggregates
  - Domain Services
  - Factories
  - Repositories
- CQRS
- SOA
  - Event Driven Architecture
- Event Sourcing
- Final Thoughts

Domain Driven Design

- Domain-Driven Design (DDD) is a development philosophy defined by Eric Evans in his seminal work **Domain-Driven Design: Tackling Complexity in the Heart of Software.**
- DDD is an approach to software development that enables teams to effectively manage the construction and maintenance of software for complex problem domains.

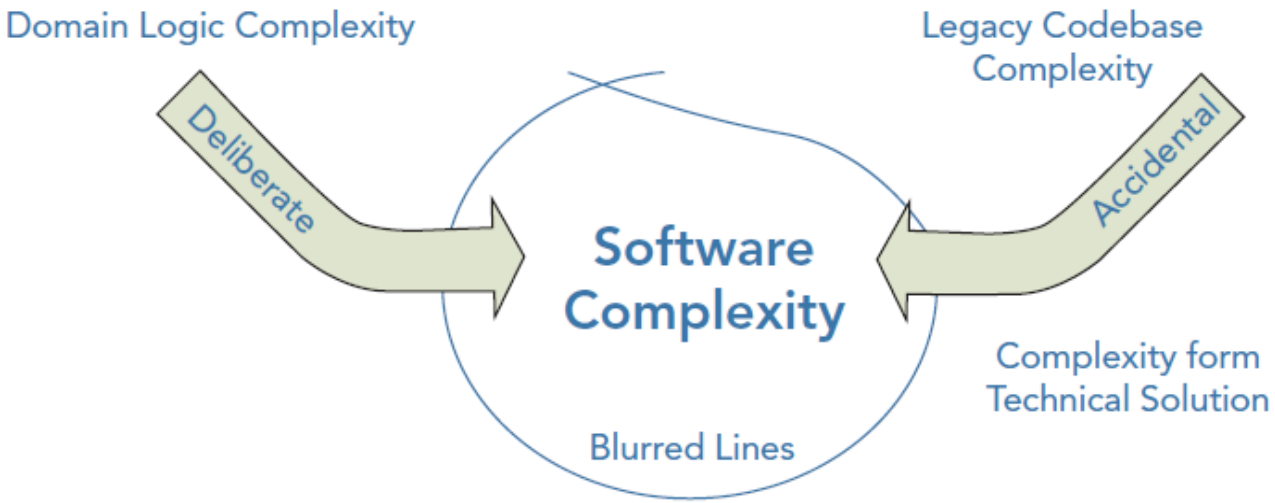
THE CHALLENGES OF CREATING SOFTWARE FOR COMPLEX PROBLEM DOMAINS

- To understand how DDD can help with the design of software for a nontrivial domain, you must first understand the difficulties of creating and maintaining software.
- By far, the most popular software architectural design pattern for business applications is the Big Ball of Mud (BBoM) pattern.
- The definition of BBoM, as defined by Brian Foote and Joseph Yoder in the paper “Big Ball of Mud,” is “… a haphazardly structured, spreading, untidy code jungle.”

THE CHALLENGES OF CREATING SOFTWARE FOR COMPLEX PROBLEM DOMAINS

- The issue with allowing software to dissolve into a BBoM becomes apparent when routine changes in workflow and small feature enhancements become a challenge to implement due to the difficulties in reading and understanding the existing codebase.
- Eric Evans describes such systems as containing “code that does something useful, but without explaining how.”
- One of the main reasons software becomes complex and difficult to manage is due to the mixing of domain complexities with technical complexities

Complexity in Software



WHAT IS AN ANALYSIS MODEL?

- An analysis model is used to describe the logical design and structure of a software application.
- It can be represented as sketches or by using modeling languages such as UML.
- It is the representation of software that non-technical people can conceptualize in order to understand how software is constructed.

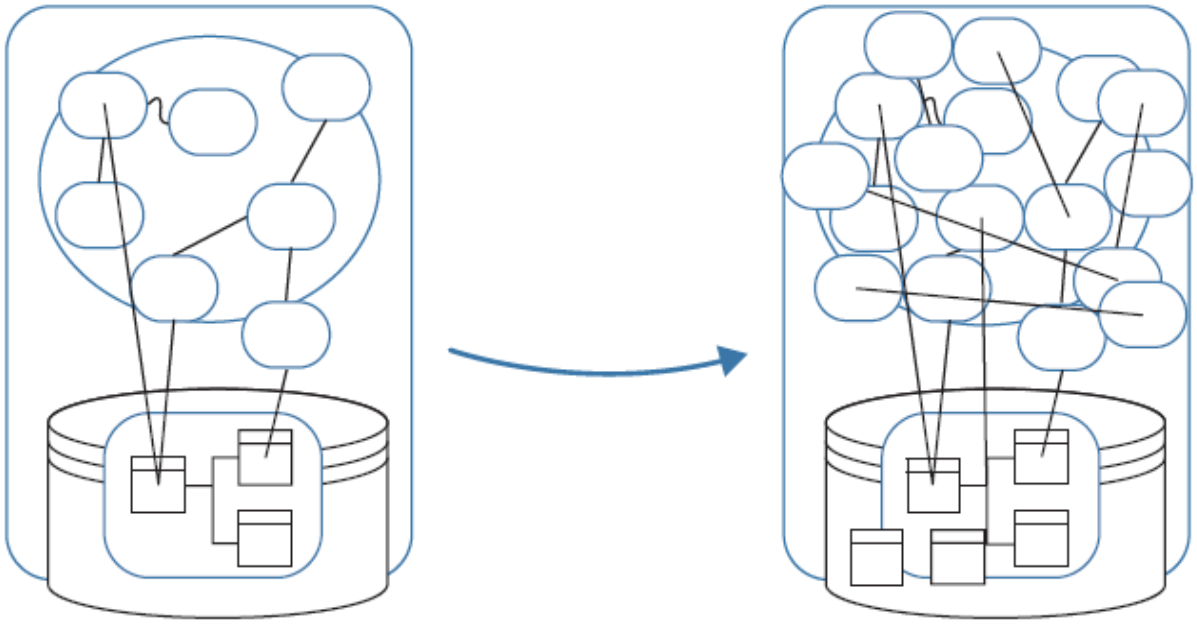
Code Created without a Common Language

- A lack of focus on a shared language and knowledge of the problem domain results in a codebase that works but does not reveal the intent of the business.
- This makes codebases difficult to read and maintain because translations between the analysis model and the code model can be costly and error prone.
- Code without a binding to an analysis model that the business understands will degrade over time and is therefore more likely to result in an architecture that resembles the BBoM pattern.
- Due to the cost of translation teams that do not utilize the rich vocabulary of the problem domain in code will decrease their chances of discovering new domain concepts when collaborating with business experts.

A Lack of Organization

- The initial incarnation of a system that resembles BBoM is fast to produce and often a well-rounded success.
- But because there is little focus based on the design of an application around a model of the problem domain, subsequent enhancements are troublesome.
- The codebase lacks the required synergy with the business behavior to make change manageable.
- Complexities of the problem domain are often mixed with the accidental complexities of the technical solution.

A Lack of Organization



Initial software incarnation fast to produce

Over time without care and consideration, software turns to ball of mud

Problem Domain

- A problem domain refers to the subject area for which you are building software.
- DDD stresses the need to focus on the domain above anything else when working on creating software for large-scale and complex business systems.
- Experts in the problem domain work with the development team to focus on the areas of the domain that are useful to be able to produce valuable software.
- For example, when writing software for the health industry to record patient treatment, it is not important to learn to become a doctor.
- What is important to understand is the terminology of the health industry, how different departments view patients and care, what information doctors gather, and what they do with it.

A Lack of Focus on the Problem Domain

- Software projects fail when you don’t understand the business domain you are working within well enough.
- Typing is not the bottleneck for delivering a product; coding is the easy part of development.
- Outside of non-functional requirements creating and keeping a useful software model of the domain that can fulfill business-use cases is the difficult part.
- However, the more you invest in understanding your business domain the better equipped you will be when you are trying to model it in software to solve its inherent business problems.

HOW THE PATTERNS OF DOMAIN-DRIVEN DESIGN MANAGE COMPLEXITY

- DDD deals with both the challenge of understanding a problem domain and creating a maintainable solution that is useful to solve problems within it.
- It achieves this by utilizing a number of strategic and tactical patterns.

The Two Sides of Domain-Driven Design

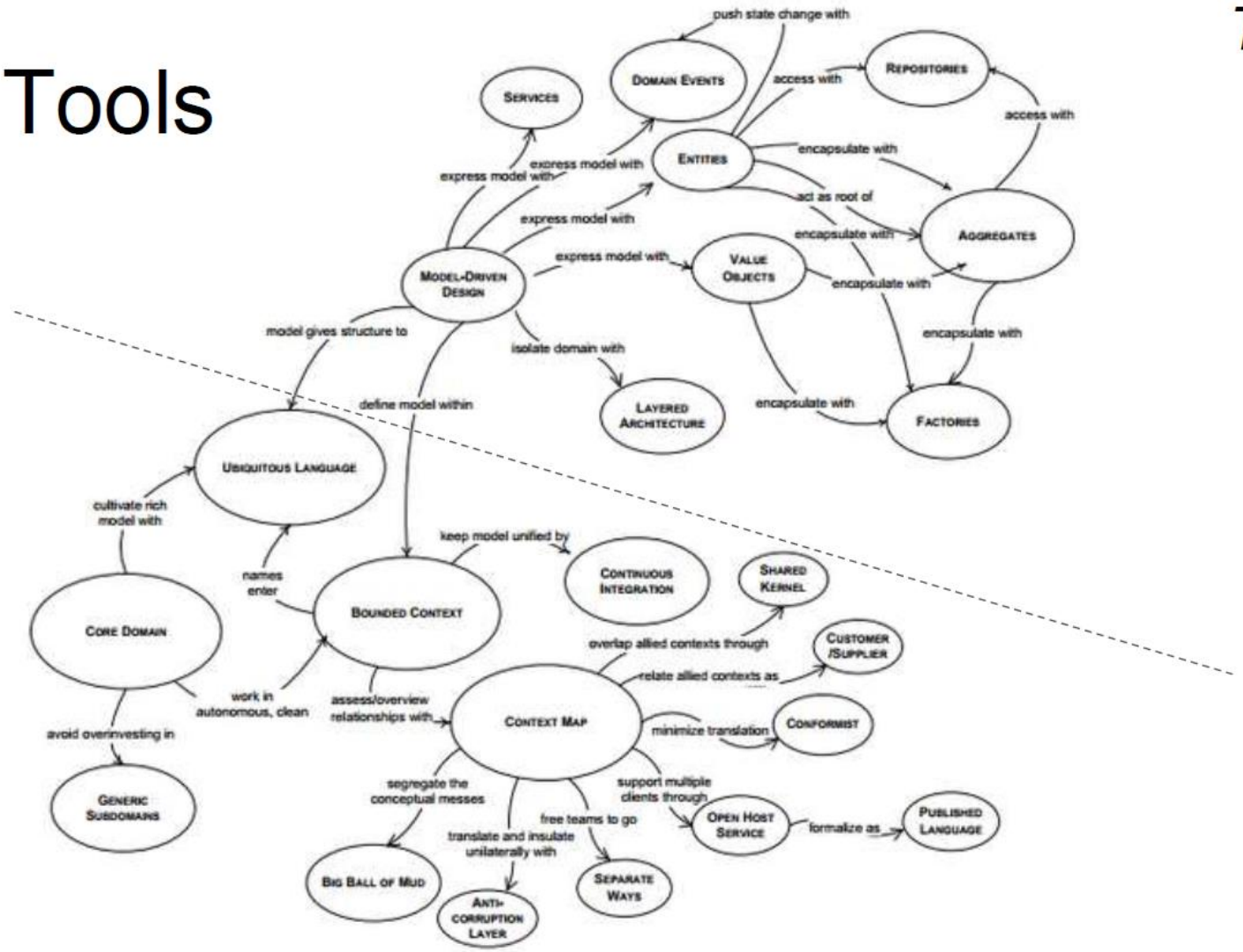
Tactical

vs

Strategic

DDD Tools

Tactical



Strategic

The Strategic Patterns of DDD

- The strategic patterns of DDD distil the problem domain and shape the architecture of an application.

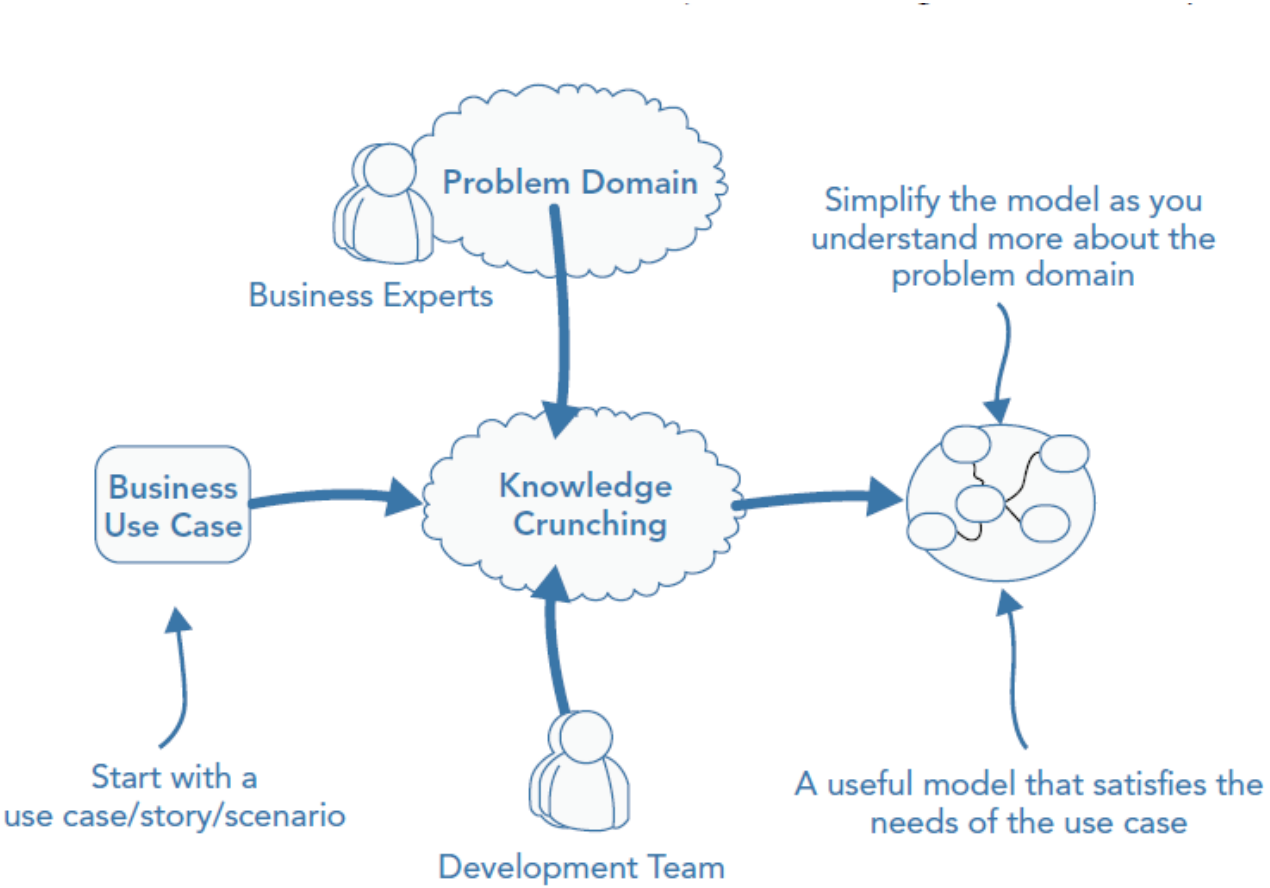
Bounded Context – Strategic Design

- Bounded Context is a Specific **Business Process / Concern**.
- Components / Modules inside the Bounded Context are context specific.
- Multiple Bounded Contexts are linked using **Context Mapping**.
- **One Team** assigned to a Bounded Context.
- Each Bounded Context will have it's own **Source Code Repository.**
- When the Bounded Context is being developed as a key strategic initiative of your organization, it's called the **Core Domain**.
- Within a Bounded Context the team must have same language called **Ubiquitous language** for Spoken and for Design / Code Implementation.

Distilling the Problem Domain to Reveal What Is Important

- Development teams and domain experts use analysis patterns and knowledge crunching to distill large problem domains into more manageable subdomains.
- This distillation reveals the core sub domain—the reason the software is being written.
- The core domain is the driving force behind the product under development; it is the fundamental reason it is being built.
- DDD emphasizes the need to focus effort and talent on the core subdomain(s) as this is the area that holds the most value and is key to the success of the application.

Distilling the Problem Domain to Reveal What Is Important



Distilling the Problem Domain to Reveal What Is Important

- Discovering the core domain helps teams understand why they're producing the software and what it means for the software to be successful to the business.
- It is the appreciation for the business intent that will enable the development team to identify and invest its time in the most important parts of the system.
- As the business evolves, so in turn must the software; it needs to be adaptable.
- Investment in code quality for the key areas of an application will help it change with the business.
- If key areas of the software are not in synergy with the business domain then, over time, it is likely that the design will rot and turn into a big ball of mud, resulting in hard-to-maintain software.

Creating a Model to Solve Domain Problems

- In the solution space a software model is built for each subdomain to handle domain problems and to align the software with the business contours.
- This model is not a model of real life but more an abstraction built to satisfy the requirements of business use cases while still retaining the rules and logic of the business domain.
- The development team should focus as much energy and effort on the model and domain logic as it does on the pure technical aspects of the application.
- To avoid accidental technical complexity the model is kept isolated from infrastructure code.

Creating a Model to Solve Domain Problems

- In the solution space a software model is built for each subdomain to handle domain problems and to align the software with the business contours.
- This model is not a model of real life but more an abstraction built to satisfy the requirements of business use cases while still retaining the rules and logic of the business domain.
- The development team should focus as much energy and effort on the model and domain logic as it does on the pure technical aspects of the application.
- To avoid accidental technical complexity the model is kept isolated from infrastructure code.

Creating a Model to Solve Domain Problems

- All models are not created equal; the most appropriate design patterns are used based on the complexity needs of each subdomain rather than applying a blanket design to the whole system.
- Models for subdomains that are not core to the success of the product or that are not as complex need not be based on rich object-oriented designs and can instead utilize more procedural or data-driven architectures..

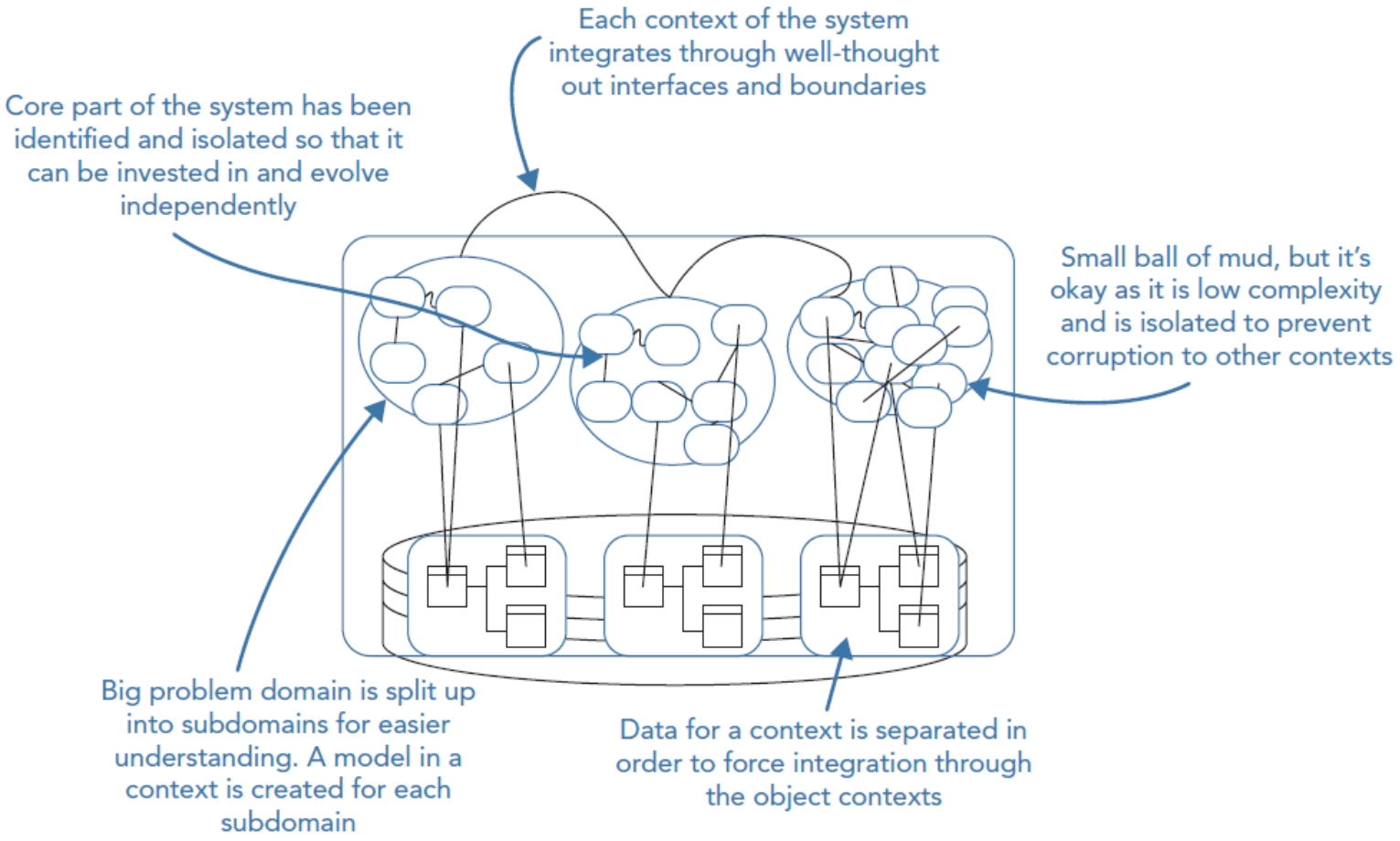
Using a Shared Language to Enable Modeling Collaboration

- Models are built through the collaboration of domain experts and the development team.
- Communication is achieved using an ever-evolving shared language known as the ubiquitous language (UL) to efficiently and effectively connect a software model to a conceptual analysis model.
- The software model is bound to the analysis model by using the same terms of the UL for its structure and class design.

Isolate Models from Ambiguity and Corruption

- Models sit within a bounded context, which defines the applicability of the model and ensures that its integrity is retained.
- Larger models can be split into smaller models and defined within separate bounded contexts where ambiguity in terminology exists or where multiple teams are a working in order to further reduce complexity.

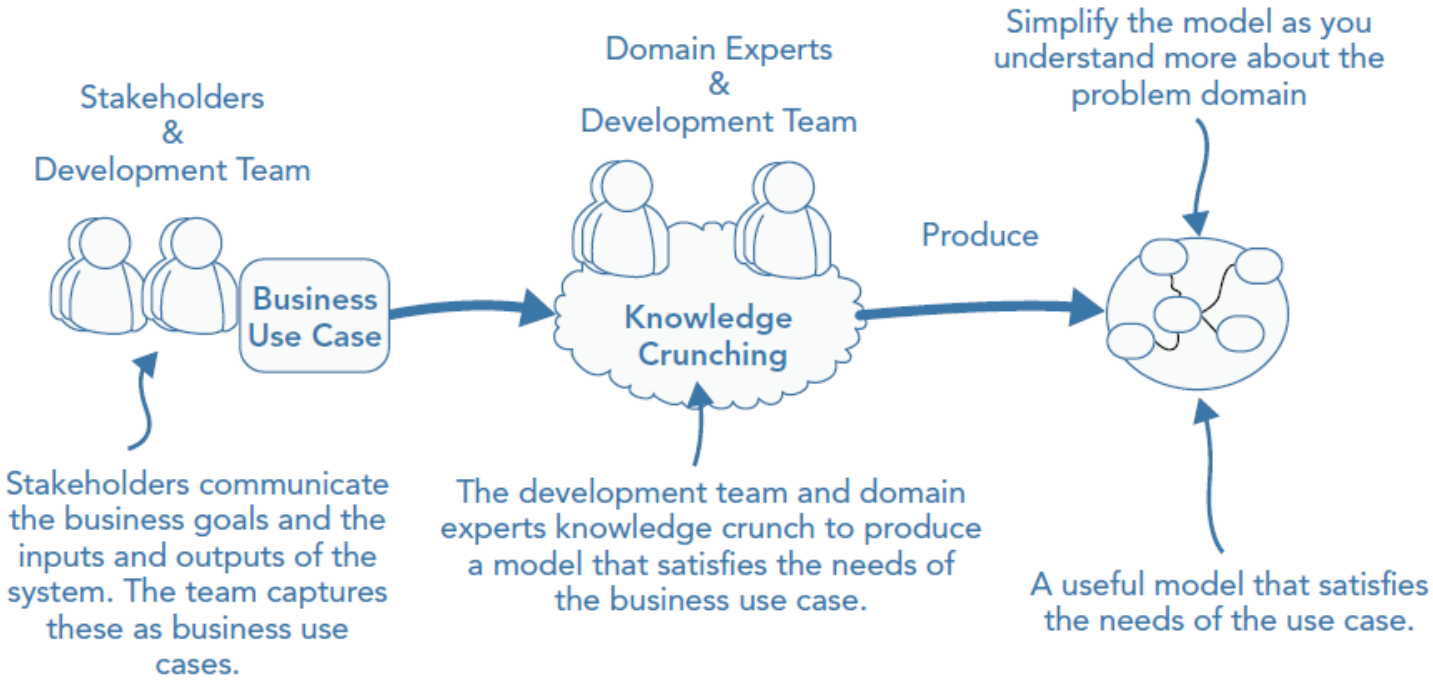
Applying the strategic patterns of Domain-Driven Design.



Understanding the Relationships between Contexts

- DDD understands the need to ensure that teams and the business are clear on how separate models and contexts work together in order to solve domain problems that span across subdomains.
- Context maps help you to understand the bigger picture; they enable teams to understand what models exist, what they are responsible for, and where their applicability boundaries are.
- These maps reveal how different models interact and what data they exchange to fulfill business processes.
- The relationships between the connections and more importantly the grey area of process that sits between them is often not captured or well understood by the business.

Stake Holders vs Domain Experts



THE PRACTICES AND PRINCIPLES OF DOMAIN-DRIVEN DESIGN

Focusing on the Core Domain

- DDD stresses the need to focus the most effort on the core subdomain.
- The core subdomain is the area of your product that will be the difference between it being a success and it being a failure.
- It's the product's unique selling point, the reason it is being built rather than bought.
- The core domain is the area of the product that will give you a competitive advantage and generate real value for your business.
- It is vital that all of the team understand what the core domain is.

Learning through Collaboration

- DDD stresses the importance of collaboration between the development teams and business experts to produce useful models to solve problems.
- Without this collaboration and commitment from the business experts, much of the knowledge sharing will not be able to take place, and development teams will not gain deeper insights into the problem domain.
- It is also true that through collaboration and knowledge crunching, the business has the opportunity to learn much more about its domain.

Creating Models through Exploration and Experimentation

- DDD treats the analysis and code models as one.
- This means that the technical code model is bound to the analysis model through the shared UL.
- A breakthrough in the analysis model results in a change to the code model.
- A refactoring in the code model that reveals deeper insight is again reflected in the analysis model and mental models of the business.
- Breakthroughs only occur when teams are given time to explore a model and experiment with its design.
- Spending time prototyping and experimenting can go a long way in helping you shape a better design.
- It can also reveal what a poor design looks like.
- Eric Evans suggests that for every good design there must be at least three bad ones, this will prevent teams stopping at the first useful model.

Communication

- The ability to effectively describe a model built to represent a problem domain is the foundation of DDD.
- This is why, without a doubt, the single most important facet of DDD is the creation of the UL.
- Without a shared language, collaboration between the business and development teams to solve problems would not be effective.
- Analysis and mental models produced in knowledge-crunching sessions between the teams need a shared language to bind them to a technical implementation.
- Without an effective way to communicate ideas and solutions within a problem domain, design breakthroughs cannot occur.

Understanding the Applicability of a Model

- Each model that is built is understood within the context of its subdomain and described using the UL.
- However, in many large models, there can be ambiguity within the UL, with different parts of an organization having different understandings of a common term or concept.
- DDD addresses this by ensuring that each model has its own UL that is valid only in a certain context.
- Each context defines a linguistic boundary; ensuring models are understood in a specific context to avoid ambiguity in language.
- Therefore a model with overlapping terms is divided into two models, each clearly defined within its own context.
- On the implementation side, strategic patterns can enforce these linguistic boundaries to enable models to evolve in isolation.
- These strategic patterns result in organized code that is able to support change and rewriting.

Constantly Evolving the Model

- Any developer working on a complex system can write good code and maintain it for a short while.
- However, without synergy between the source code and the problem domain, continued development will likely end up in a codebase that is hard to modify, resulting in a BBoM.
- DDD helps with this issue by placing emphasis on the team to continually look at how useful the model is for the current problem.
- It challenges the team to evolve and simplify complex models of domains as and when it gains domain insights.
- DDD is still no silver bullet and requires dedication and constant knowledge crunching to produce software that is maintainable for years and not just months.
- New business cases may break a previously useful model, or may necessitate changes to make new or existing concepts more explicit.

POPULAR MISCONCEPTIONS OF DOMAIN-DRIVEN DESIGN

- Tactical Patterns Are Key to DDD
- DDD Is a Framework
- DDD Is a Silver Bullet

THE SALIENT POINTS

- Domain-Driven Design (DDD) is a development philosophy that is designed to manage the creation and maintenance of software written for complex problem domains.
- DDD is a collection of patterns, principles, and practices, which can be applied to software design to manage complexity.
- DDD has two types of patterns.
- Strategic patterns shape the solution, while tactical patterns are used to implement a rich domain model.
- Strategic patterns can be useful for any application but tactical patterns are only useful if your model is sufficiently rich in domain logic.

THE SALIENT POINTS

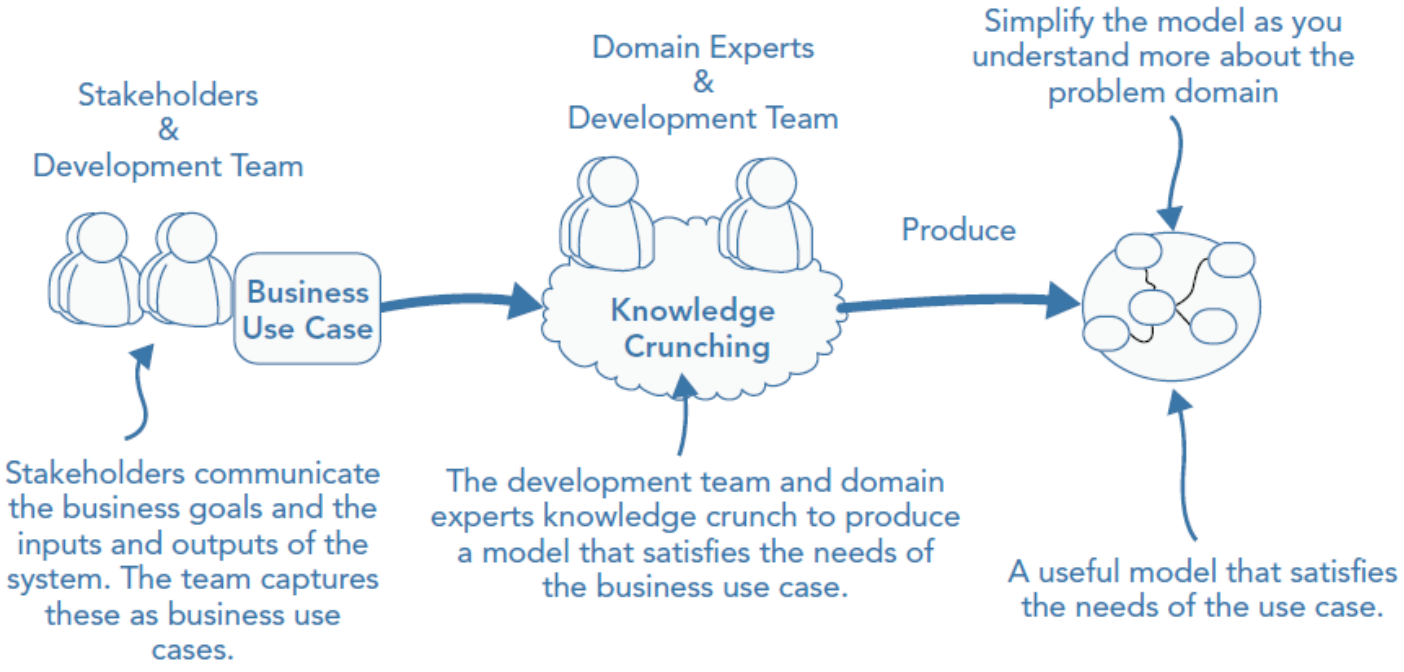
- Distillation of large problem domains into subdomains can reveal the core domain—the area of most value.
- Not all parts of a system will be well designed; teams must invest more time in the core subdomain(s) of a product.
- An abstract model is created for each subdomain to manage domain problems.
- A ubiquitous language is used to bind the analysis model to the code model in order for the development team and domain experts to collaborate on the design of a model.
- Learning and creating a language to communicate about the problem domain is the process of DDD.

THE SALIENT POINTS

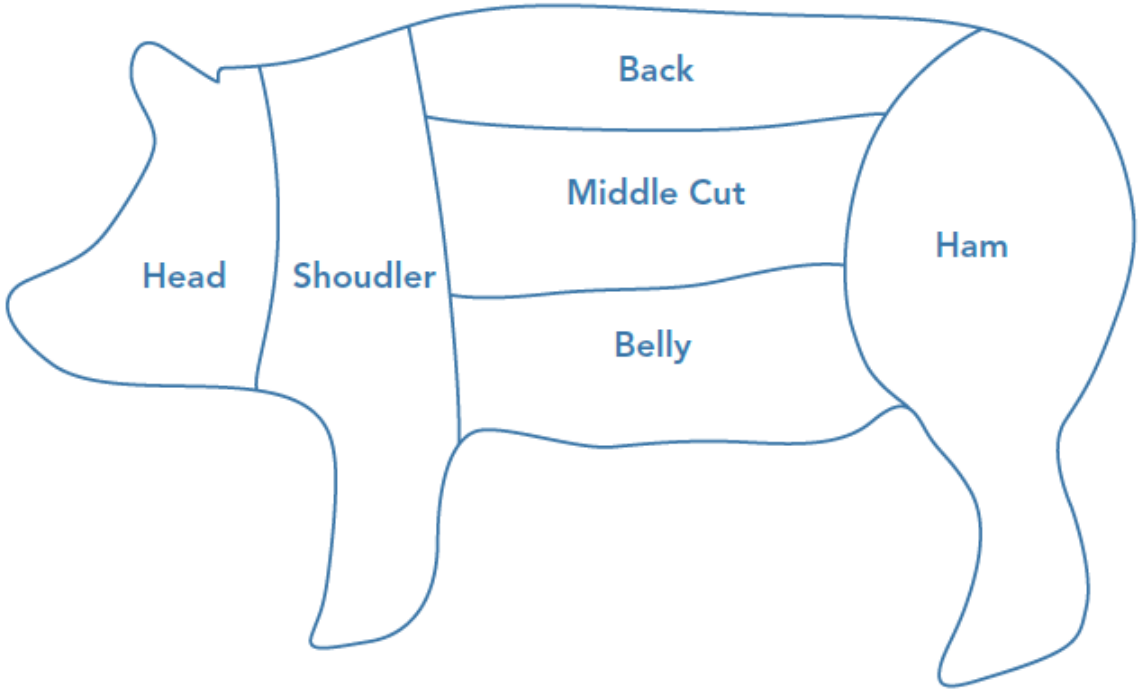
- Code is the artifact.
- In order to retain the integrity of a model, it is defined within a bounded context.
- The model is isolated from infrastructure concerns of the application to separate technical complexities from business complexities.
- Where there is ambiguity in terminology for a model or multiple teams at work the model can be split and defined in smaller bounded contexts.
- DDD does not dictate any specific architectural style for development, it only ensures that the model is kept isolated from technical complexities so that it can focus on domain logic concerns.

THE SALIENT POINTS

- DDD values a focus on the core domain, collaboration, and exploration with domain experts, experimentation to produce a more useful model, and an understanding of the various contexts in play in a complex problem domain.
- DDD is not a patterns language, it is a collaboration philosophy focused on delivery, with communication playing a central role.
- DDD is a language- and domain-centric approach to software development.

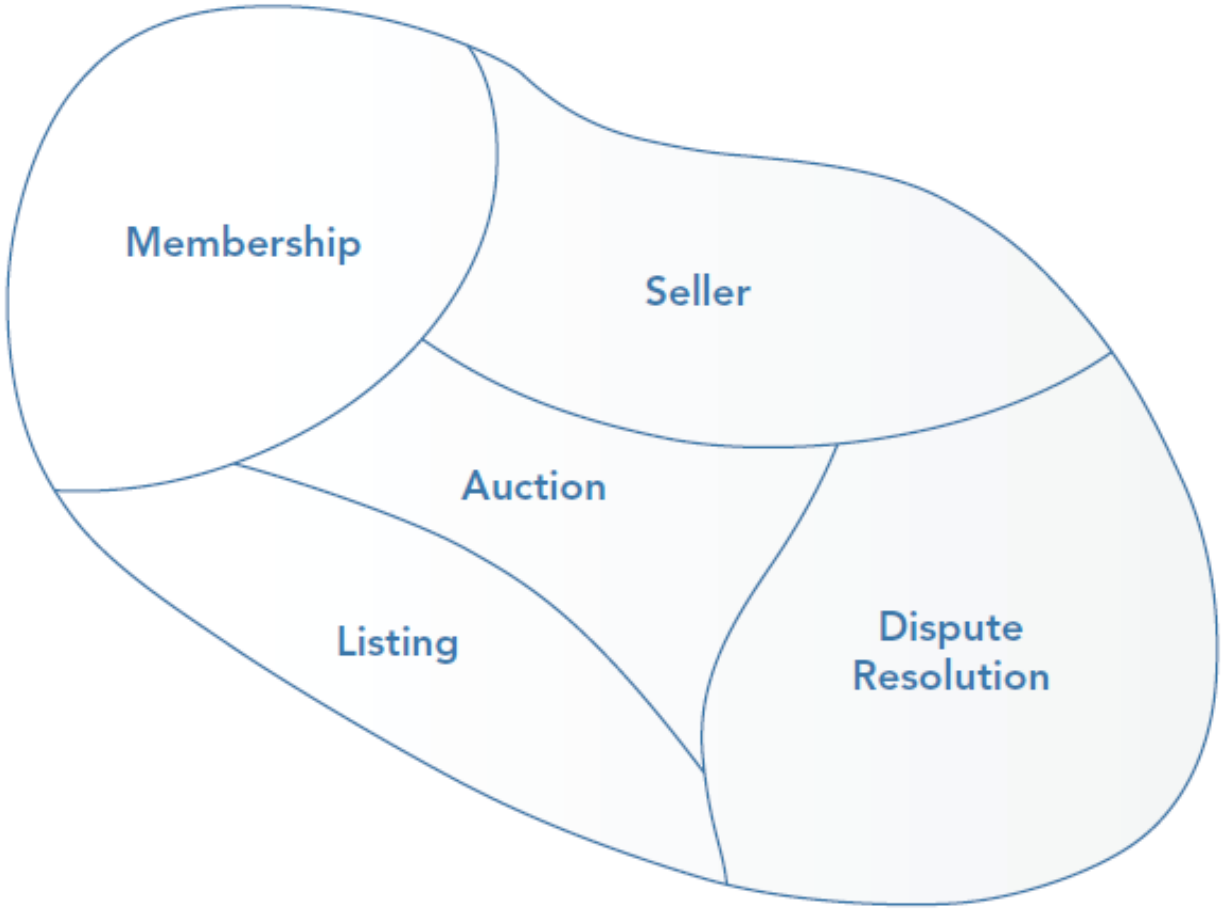

Stakeholders
&
Development Team
Business
Use Case
Stakeholders communicate the business goals and the inputs and outputs of the system. The team captures these as business use cases.
Domain Experts
&
Development Team
Knowledge
Crunching
The development team and domain experts knowledge crunch to produce a model that satisfies the needs of the business use case.
Produce
Simplify the model as you understand more about the problem domain
A useful model that satisfies the needs of the use case.

HOW TO FOCUS ON THE CORE PROBLEM

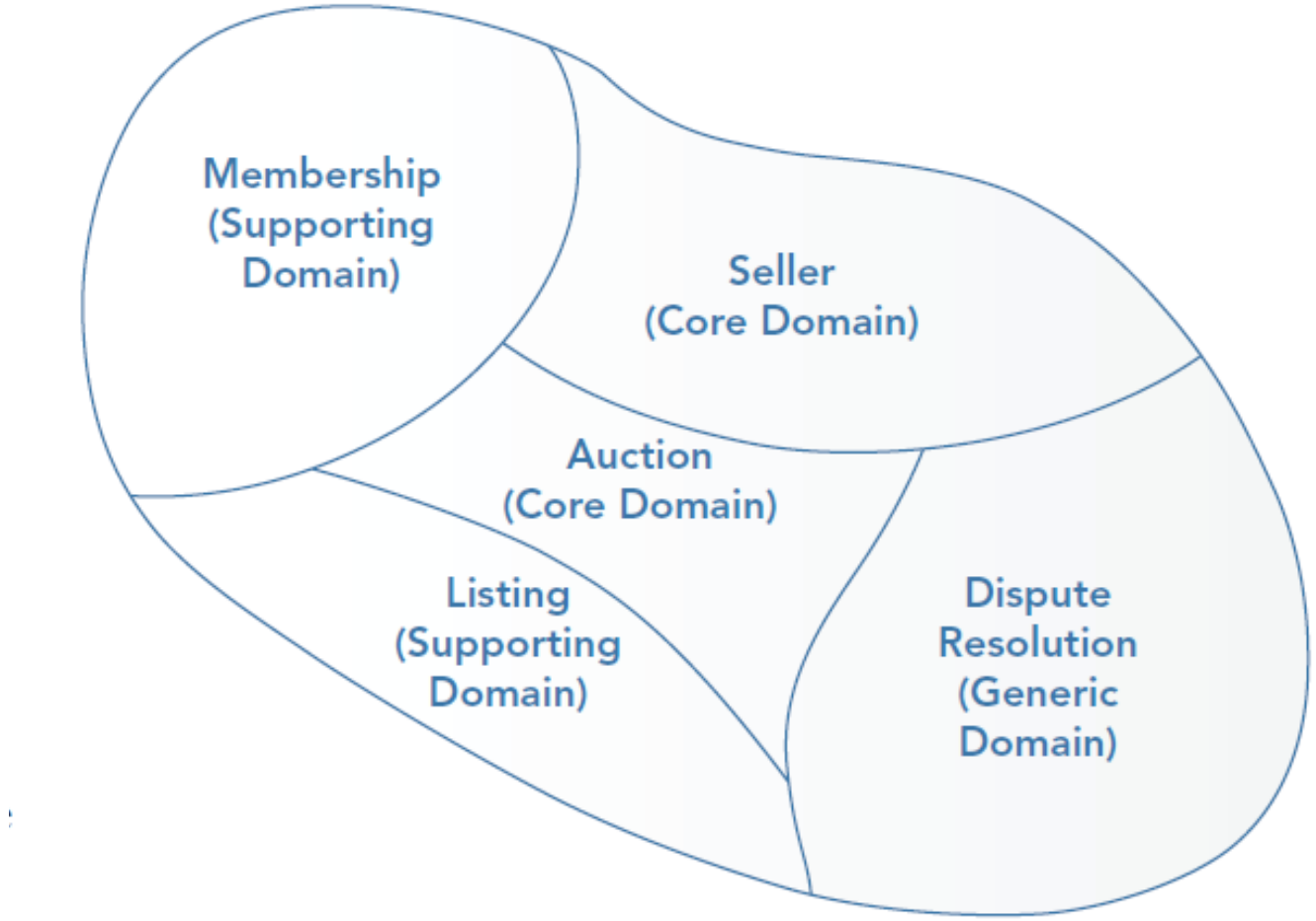


Distilling a Problem Domain

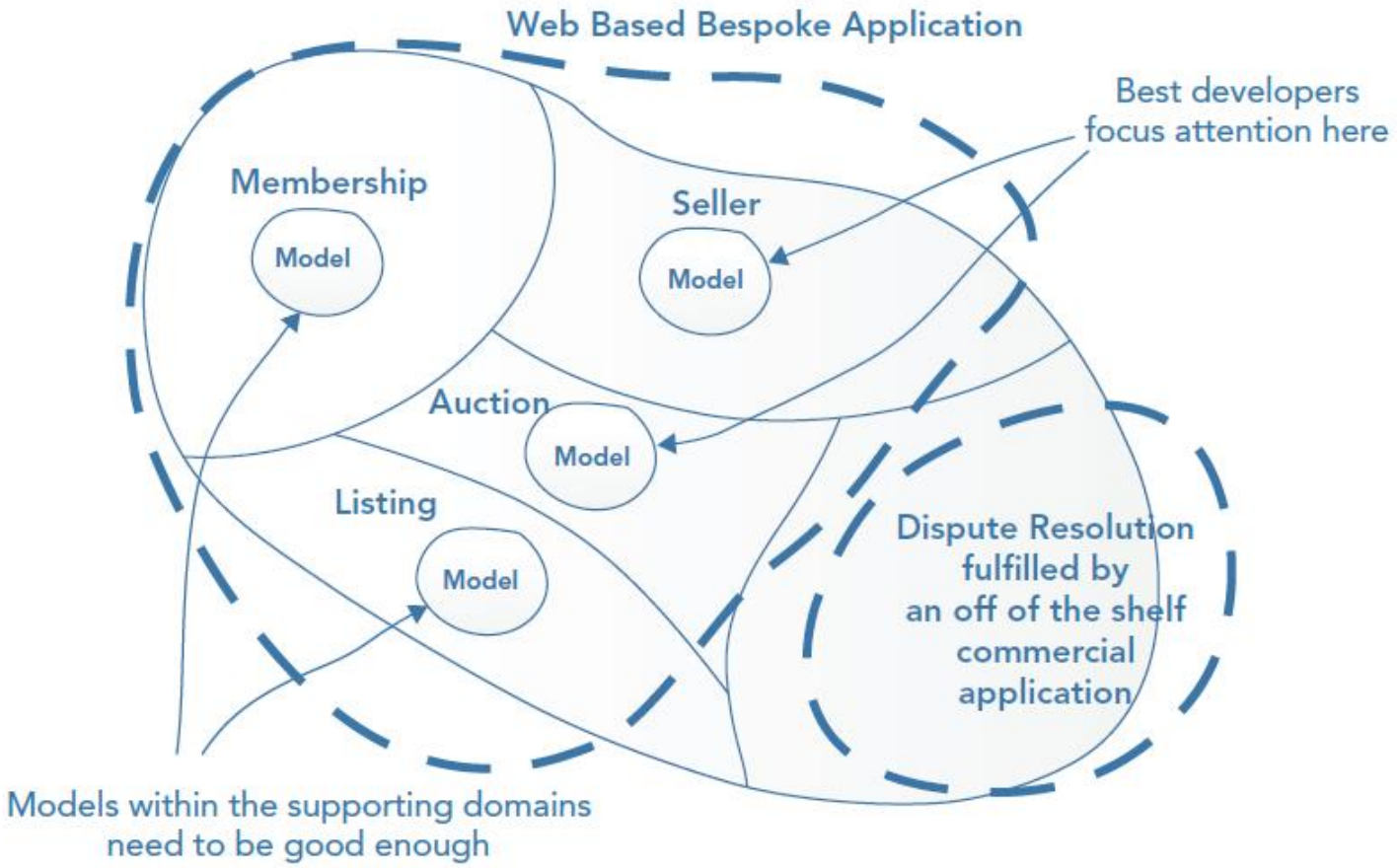
The domain of an online auction site distilled into subdomains



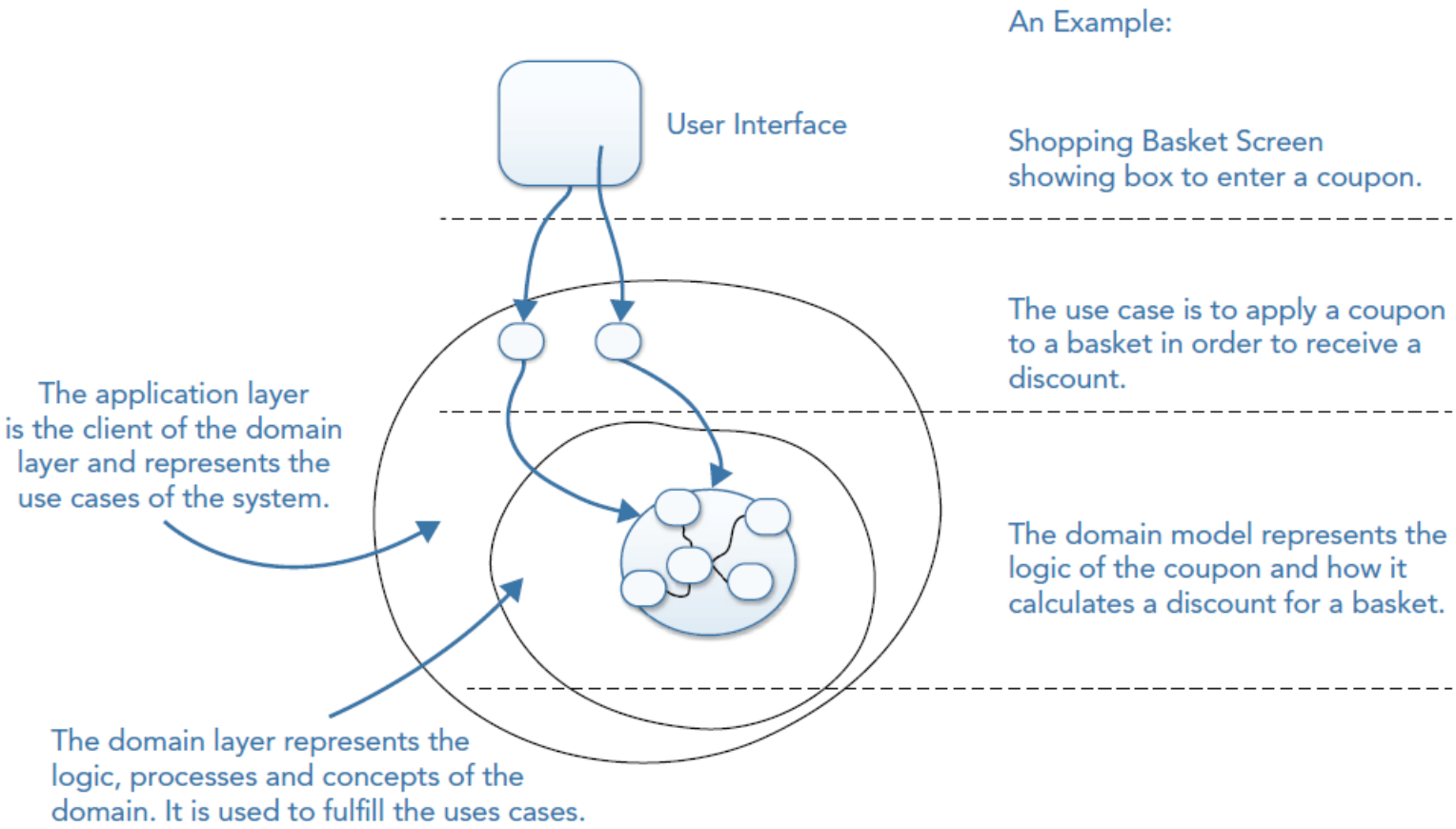
The distilled domain of an online auction site partitioned into core, generic, and supporting domains



How a solution maps to the subdomains of the auction system.



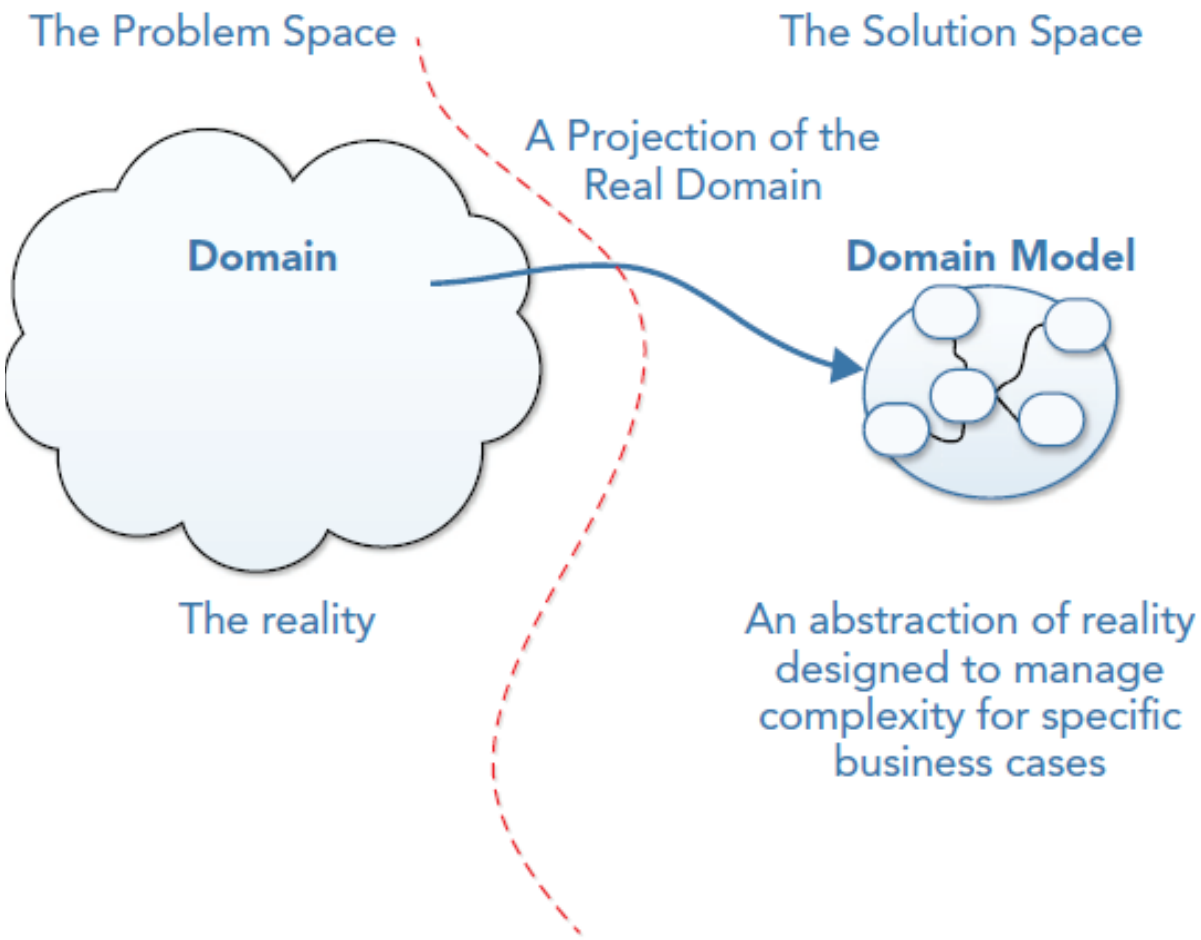
The role of a domain model



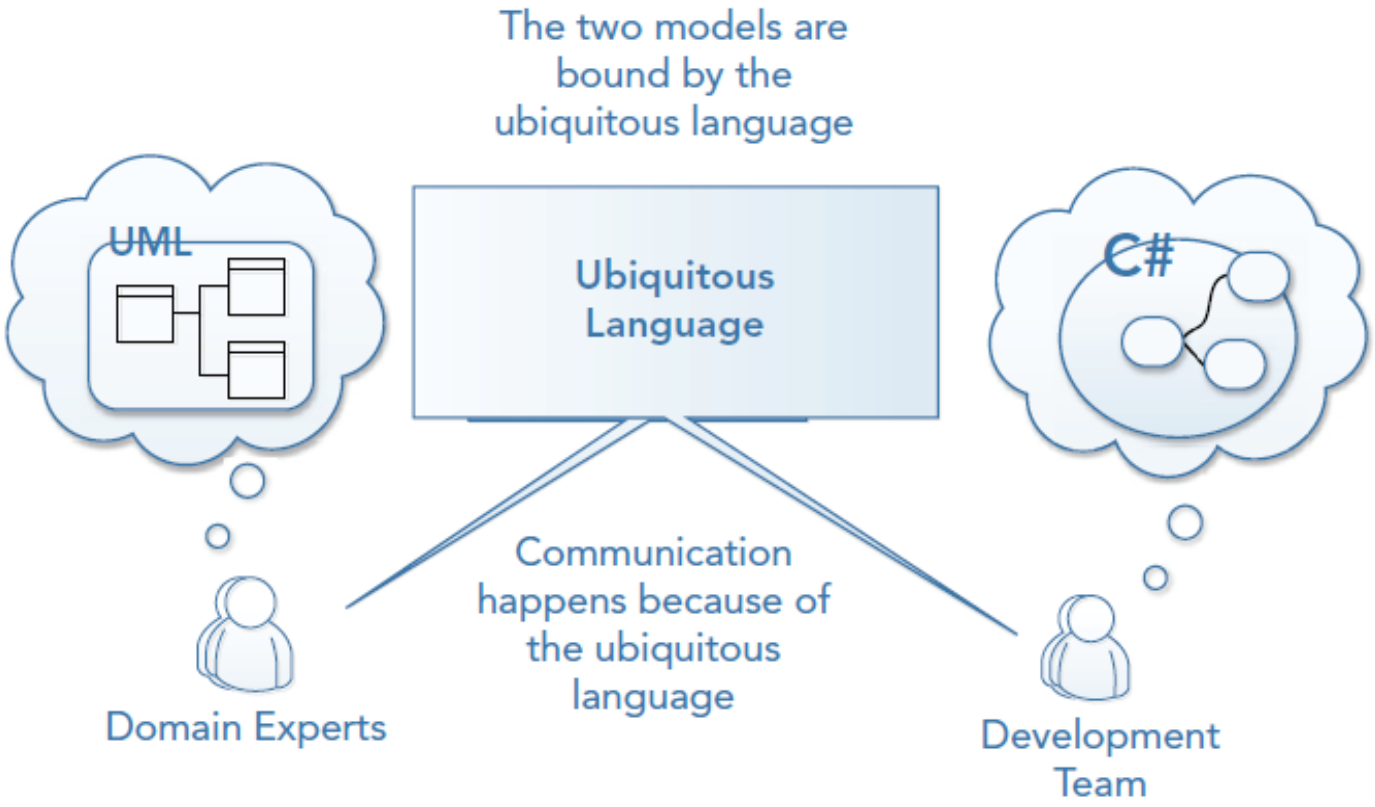
Model Driven Design

- A model driven design is a code or software built around a set of domain concepts extracted from the domain experts.
- Hence right from the start the implementation and domain are dependent on each other thus creating a feedback loop which unlike the waterfall model will be very useful .
- Model Change ➔Code Change and likewise
- Code Change ➔ Model Change

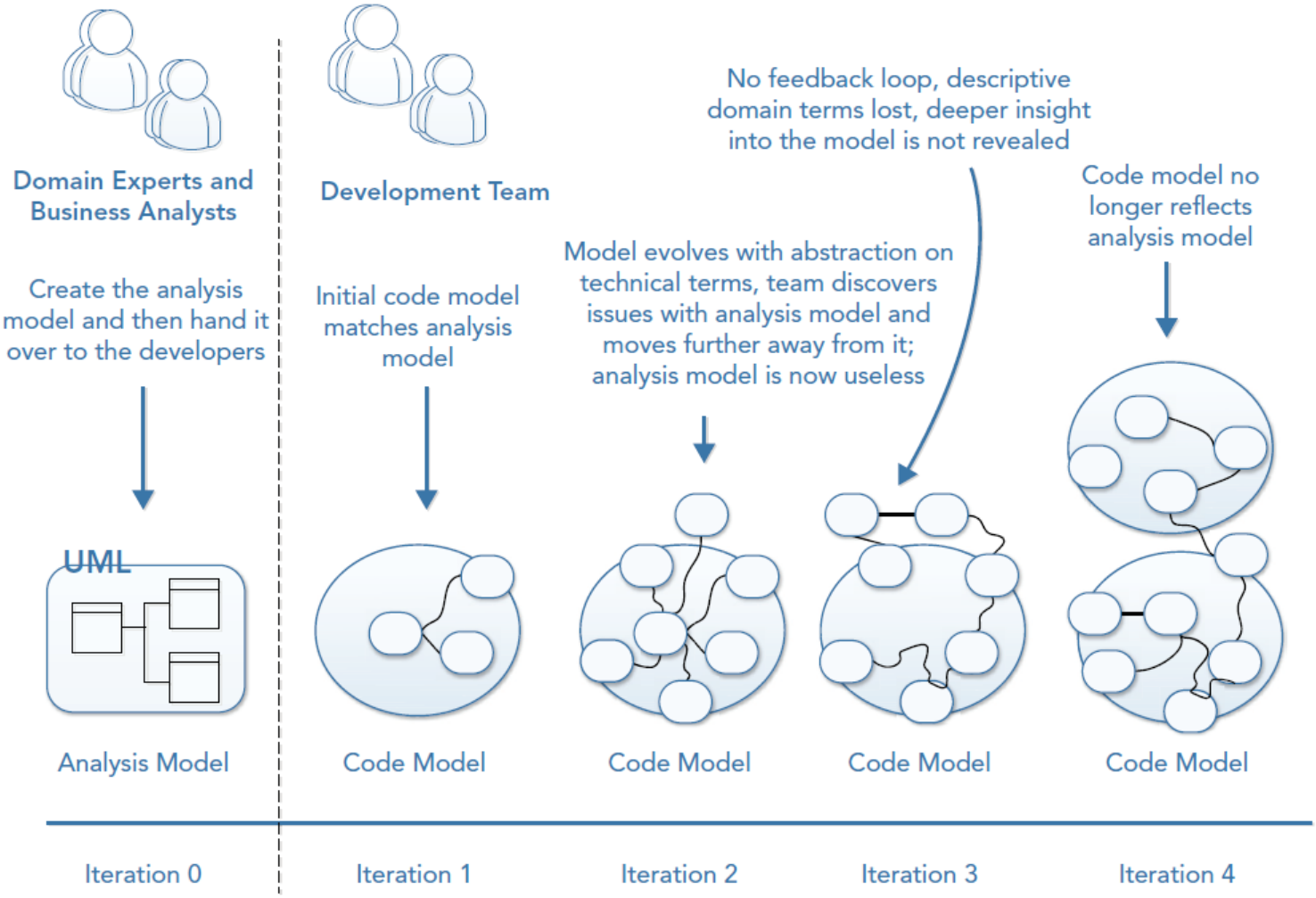
The domain versus the domain model



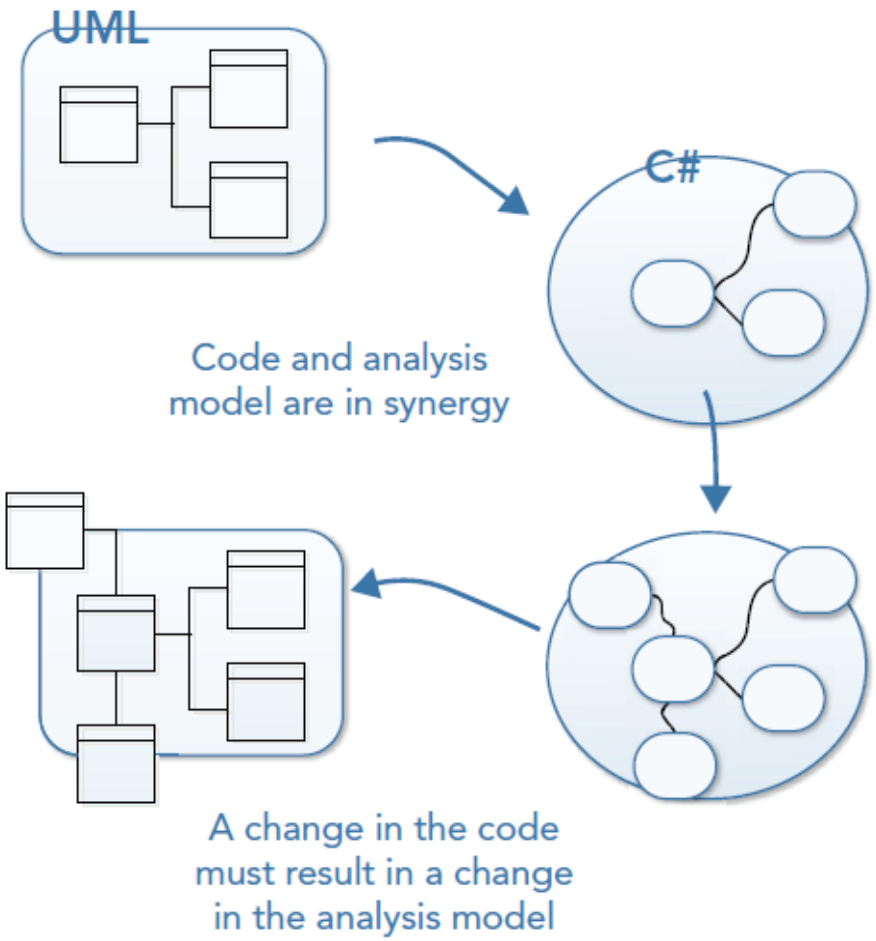
The binding between the code and analysis model



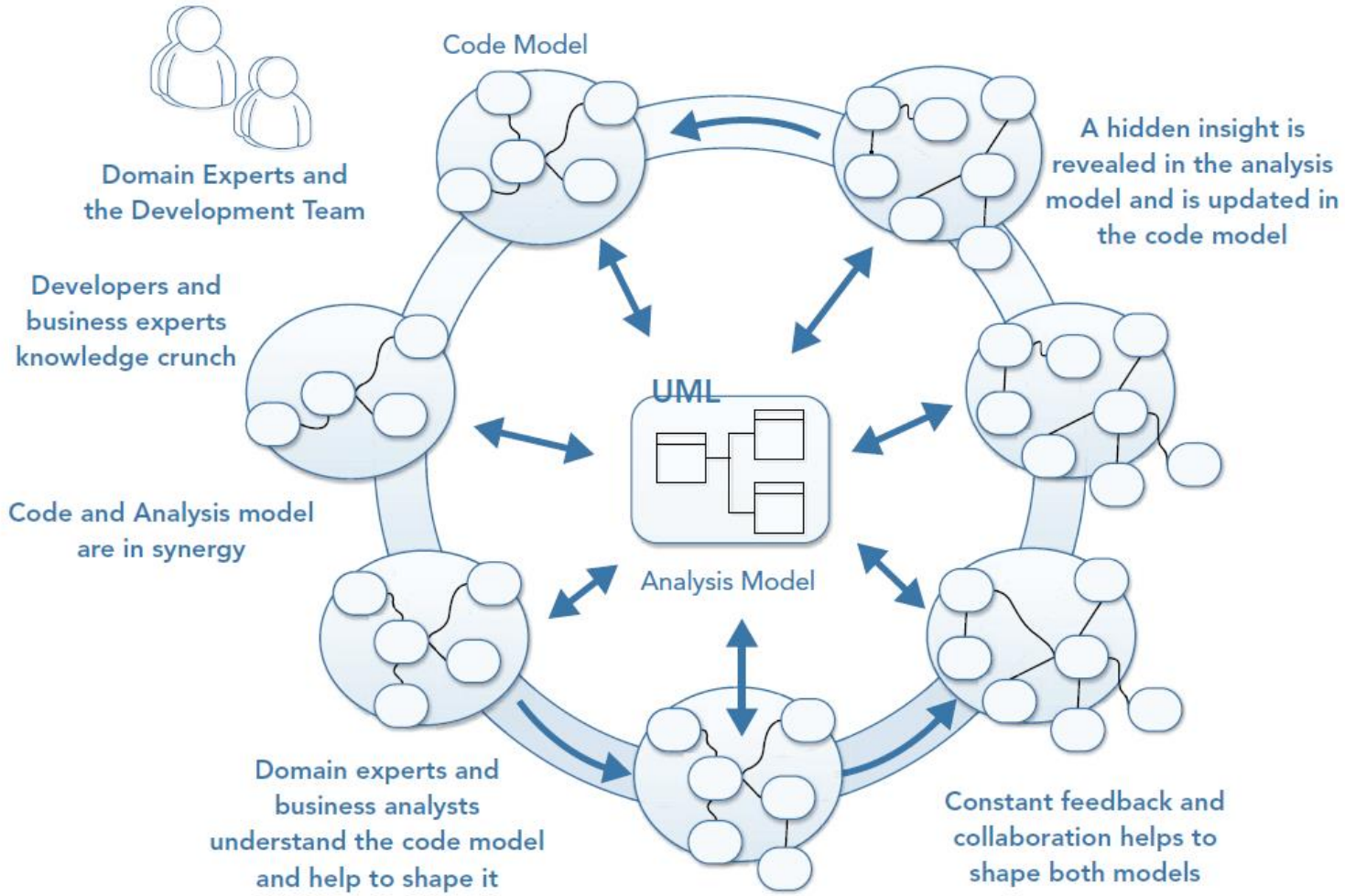
how the analysis and code models can diverge from each other if the development team is not involved in domain knowledge crunching



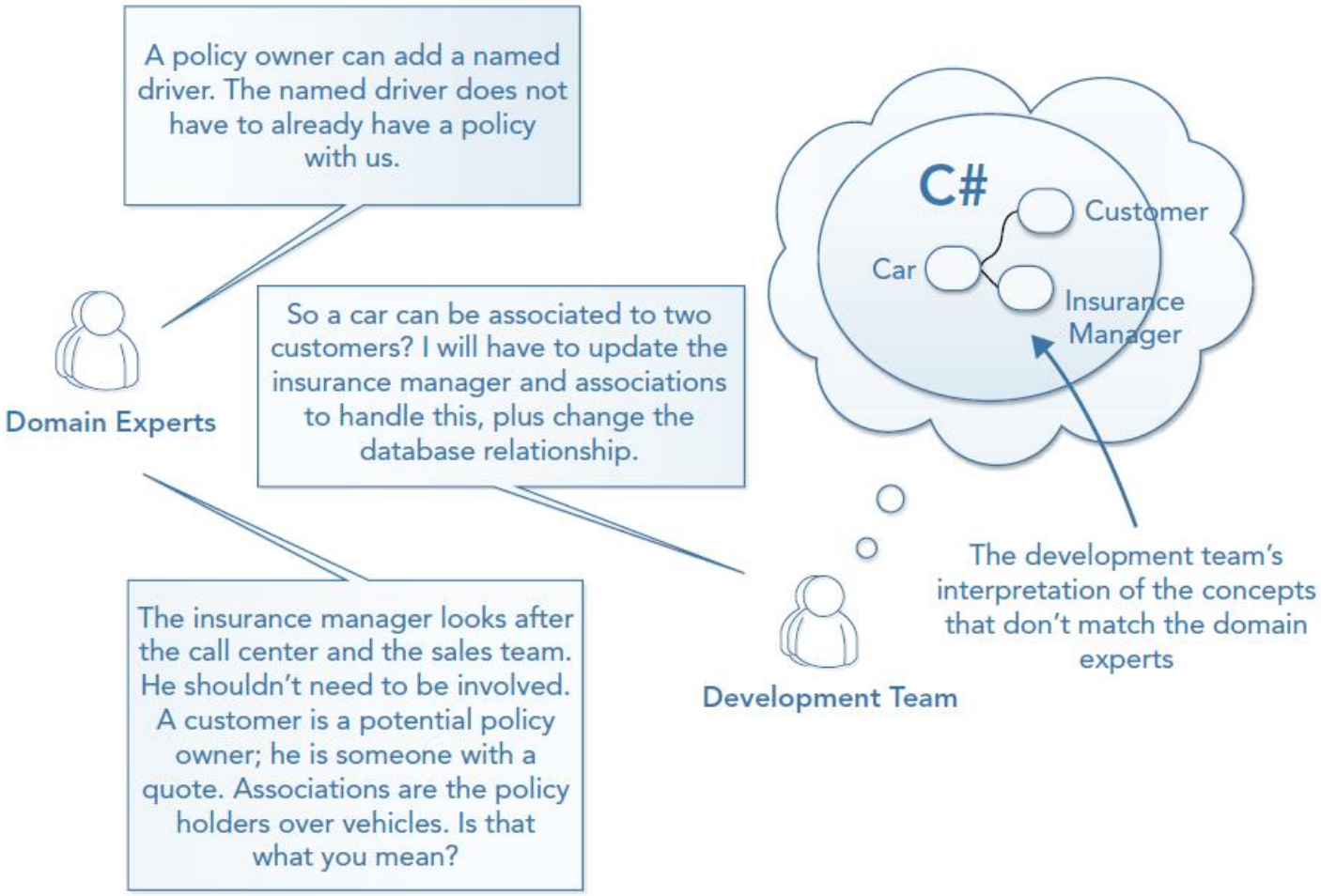
The code model and the analysis model are kept in synergy



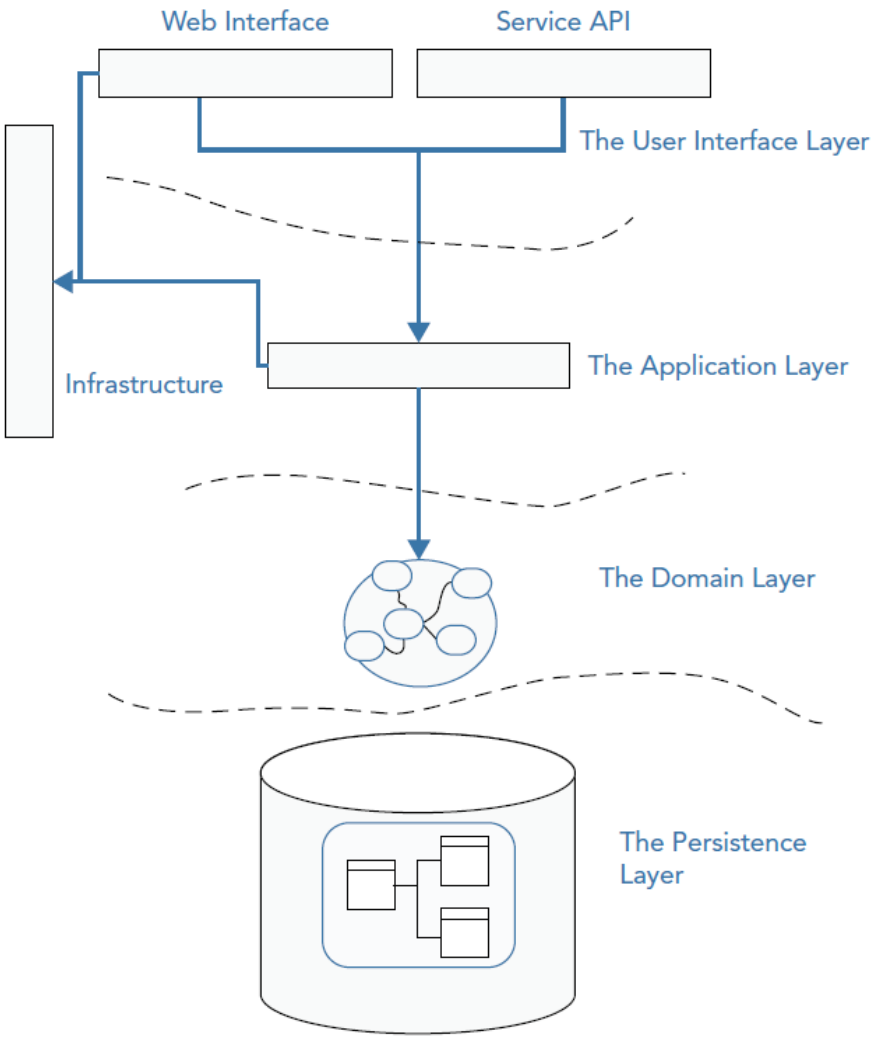
Team modeling



Translation costs of the project



THE DOMAIN LAYER



Domain Model Pattern

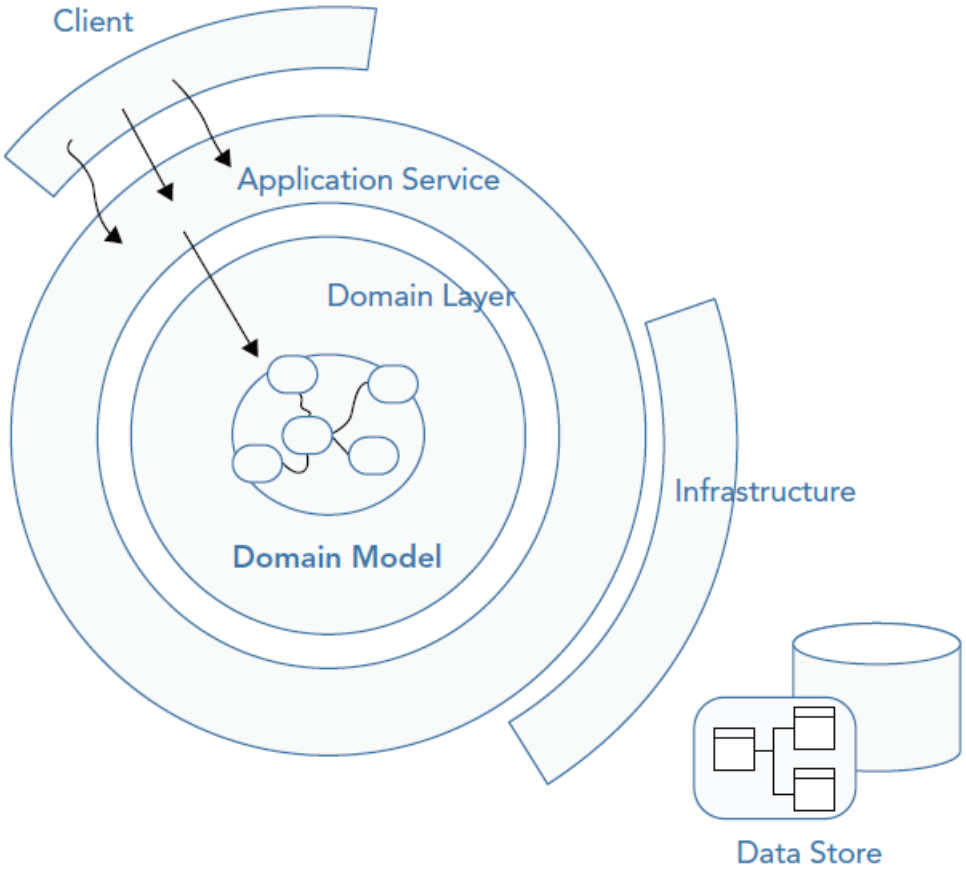


FIGURE 5-3: The domain model pattern.

Questions

Module Summary

- Spring Integration Framework.
- Message, Channel and Adapter
- Understood the different Component Integration
- Understood the Event-Driven Architecture