Introduction to Domain Driven Design CSCI 5448 Presentation topic

Submitted by : Vivek Madhavan

The Participants

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

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

Note : The domain not necessarily means anything related to software and hardware , it can also mean a business as shown by an example towards the end of this presentation

What is Domain Driven Design?

- Focus on the domain and the domain logic

How to use Domain Driven Design?

- Complex domain designs should be based on a model.[*i.e. : A model that can be understood and interpreted seamlessly between domain expert(s) and developer(s)*)]
- Following suit there is also a need for a ubiquitous language for both the participants.

Modeling a domain

- 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 .

Ingredients of Effective Modeling

- ***Binding the Model and Implementation*** : Forming the link between the model and the code being implemented at an early state
- ***Cultivating a language based on the model :*** Understanding the terms used by the domain experts and developers and trying to have an organized set of terms which can be unambiguously understood by both participants

Ingredients of Effective Modeling ..cntd

- ***Developing a knowledge-rich model :*** The components of the model built should follow certain rules , have behaviors , constraints which will be important to solve a complex problem related to the domain or the software
- ***Distilling the model :*** During the course of future meetings the model will grow in code and features . Concepts may be added , removed, remodeled as deemed appropriate
- ***Brainstorming and Experimenting :*** The language ,model and brainstorming attitude will go on for testing and judging of variations suggested and lead towards a final satisfactory product .

Knowledge crunching & Knowledge-Rich Design

- Domain modelers (mostly developers) need to
  - obtain information from Domain experts
  - Create an organized model based on available information
  - Have domain experts over look the progress (else developers may end up with ideas and concepts that are too naïve or shallow)
- Models go beyond nouns , they also have behavior , constraints etc. Knowledge crunching can be difficult if inconsistencies in domain concepts exist . For example if certain policies in a domain are just “adjusted” for the situation, it is not applicable from the point of view of a developer.

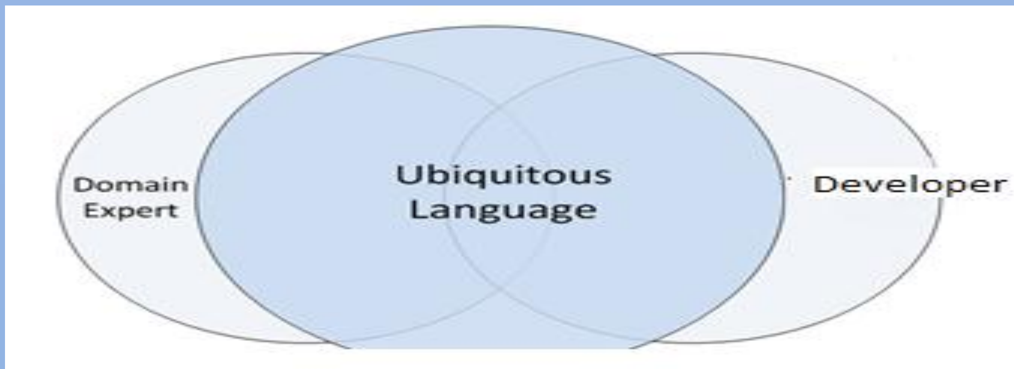
Fractured Language Problem

- Domain experts shave their own language while developers can have the own language for discussing domain. This implies fractured language and hence consequences on project.
- Same individual can imply a term of a jargon in different ways while documenting or while using it in speech . This makes knowledge crunching anemic.

Solution – Ubiquitous language

- Use the model discussed earlier as the backbone language
- Use it consistently in diagrams , documentation and speech .
- Iron out and resolve confusion over terms and expressions via conversation
- Again : Change in Ubiquitous language → Change in Model.

Confluence of Terminologies



ONE TEAM ,ONE LANGUAGE :

Developers will usually want to hide the models from the domain expert citing reasons like , it would be too profound for them to understand and that we will have to talk to them in their terminology .

But , in perspective , if a domain expert does not understand a model , it means that there has to be something wrong with the model itself . The inputs from a domain expert are essential to develop a good model

Considering the above point , a stance for Ubiquitous language is further justified .

Fine tune a model using U Language

- If we give the **Routing Service** an origin, destination, and arrival time, it can look up the stops the cargo will have to make and, well... Stick them in the database.(vague and technical)
- The origin, destination, and so on... it all feeds into the **Routing Service**, and we get back an **Itinerary** that has everything we need in it(better but verbose)
- A **Routing Service** finds an **Itinerary** that satisfies a **Route**(apt and concise)

NOTE : From the above 3 sentences , the third sentence implies a fine tuned model of a what a Routing Service object contains and the function it performs .

Isolating the domain

- In OO products usually , the UI , Database and other supports are written down directly in single objects .
- Additional work if needed is further inserted in to the same (business) object.
- ***Problem*** ? : It will work only in the short run. When complications arise later on , it will be difficult to reason and work around it .
- ***Solution*** : Isolate the Domain !!!! Rather than implementing domain concepts in the first layer have a separate layer which is responsible for the domain logic

(Layered Architecture)

- There have been many ways the software product can be divided to bring about an isolation of the domain
- However , Through experience and convention , the industry has converged on a layered architecture for division of a software system
- Basically broken as
  - User Interface layer
  - Application layer(sometimes merged with UI layer)
  - Domain layer
  - Infrastructure layer (sometimes multilayered)

User Interface

- Show information to the user
- Interpret user commands
- User maybe man or machine

Application Layer

- Defines the software
- Does not contain any domain knowledge, instead delegates the work to the domain layer
- Reflects the progress of a task to the user.

Domain (Model Layer)

- Known as the heart of business software
- Responsible for representing the concepts of the domain (business)
- Reflects the state of the business (domain).
- Storing details delegated to the Infrastructure layer
- May be the smallest yet most important part of the entire software system .

Infrastructure layer

- Provides generic technical capabilities to support higher layers .
- Supports pattern of interaction between the higher layers .

Layer Policies

- They should be strongly cohesive
- Should depend on only the layers below (loosely coupled).
- An object of any layer can depend on another object in the layers below
- Model view control is the suggested approach
- Domain layer should only focus on expressing the domain model

Communication with higher layers

In certain cases where a lower layer needs to communicate with the upper layer it can use

Callbacks

- Passing an abstract class or interface, of which the receiver will call one or more methods, while the calling end provides a concrete implementation

Observer Pattern

- An object maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods.

Does an Architecture support Domain Driven Design?

Certain architecture vary from the Layered approach If an architecture isolates domain related code and logic in a single layer such that …

- It is strongly cohesive
- Loosely coupled with the rest of the system
- Can perform tasks related to the domain concepts or policies delegated from other layers/interfaces.

Then such an architecture is capable of support domain driven designs .

Model expressed in software

Now that a model schema has been established it needs to be expressed in software . The basic components(object) that Eric Evans explains are

- *Entities*
- *Value Objects*
- *Services and*
- the *Association* between these objects
- Modules (two or more elements grouped)

Associations

- Every association in the model has mechanism in the software with the same properties
- Can be implemented as a collection in an instance variable or as an accessor that queries the database
- **PROBLEM ?**:General associations complicate implementations
- **SOLUTION**(make associations more tractable by)
    - Imposing a traversal direction
    - Adding a qualifier , reducing multiplicity
    - Eliminating non essential associations.

Entities

- Some objects are not primarily defined by their attributes (**called Entities**)
- Such objects must be distinguished from other similar objects having the same attributes
- **Reason ?:** Mistaken identity of similar objects can lead to data corruption.

Example : two square shaped objects having the same attribute values must need an identity to distinguish between each other

Designing Entities

- Unique identity(string ,integer or any other single attribute) for an Entity (object)
- Define an operation that is guaranteed to produce a unique identity result for each object.
- Unique identification can be irrespective of form , state , history and other attributes .
- Or it may be a combination of certain attributes and still guaranteed to be unique .
- The identification element can come form outside or may be created within the object .

Designing Entity … contd

- Once identity is symbol is created it is designated immutable.
- Sometimes an Entity can be generated automatically by the system . Most of the 3GL and 4GL’s usually provide this mechanism
- Sometimes certain objects may need the same identity and hence be the same . In such cases it has to be done manually in code and not the generated way .

Value Object

- When only the attributes of an object are important considerations then such an object is classified as a Value Object.
- Value objects are instantiated to represent elements of a design that we care about only for what they are and not who they are.
- Value objects are passed as parameters in messages between other objects (usually Entities)
- Value objects can be used as attributes within an entity too .

Designing Value Objects

- Value objects have to be immutable .If a value has to change within an entity it has to be fully replaced (should not have a pointer to another instance)

Services

- Some concepts from domain are not capable to be modeled as entity or Value objects and hence called Services.
- Services tend to be named for what they can do (for a client)
- A service has to be offered as an interface that is defined as a part of the model .
- Its parameters and results should be domain objects

Services ..Contd

The 3 characteristics for good service are

- The operation relates to a domain concept that is not a natural part of an entity or value object
- The interface is defined in terms of other elements of the domain model
- The operation is stateless (client can use any instance of a service irrespective of its previous history (state))

Modules

- Two or more elements of a Model constitute a module
- This classification is usually done to reduce cognitive overloading from an individuals perspective
- Modules provide 2 views on a model ,namely
  - An Individual Module characteristic
  - Idea about relations between two modules ,neglecting the interior details.
- High cohesion and low coupling are the unsung factors of modules .
- Nomenclature of modules should find itself in the ubiquitous language being used by the team .

Modules ….Contd

- **Problem** : Change in modules may imply extensive change of code .

Solution : Reorganize modules when a potential issue is spotted rather than waiting and increasing inertia.(AGILE MODULE)

Non-Objects in an Object World

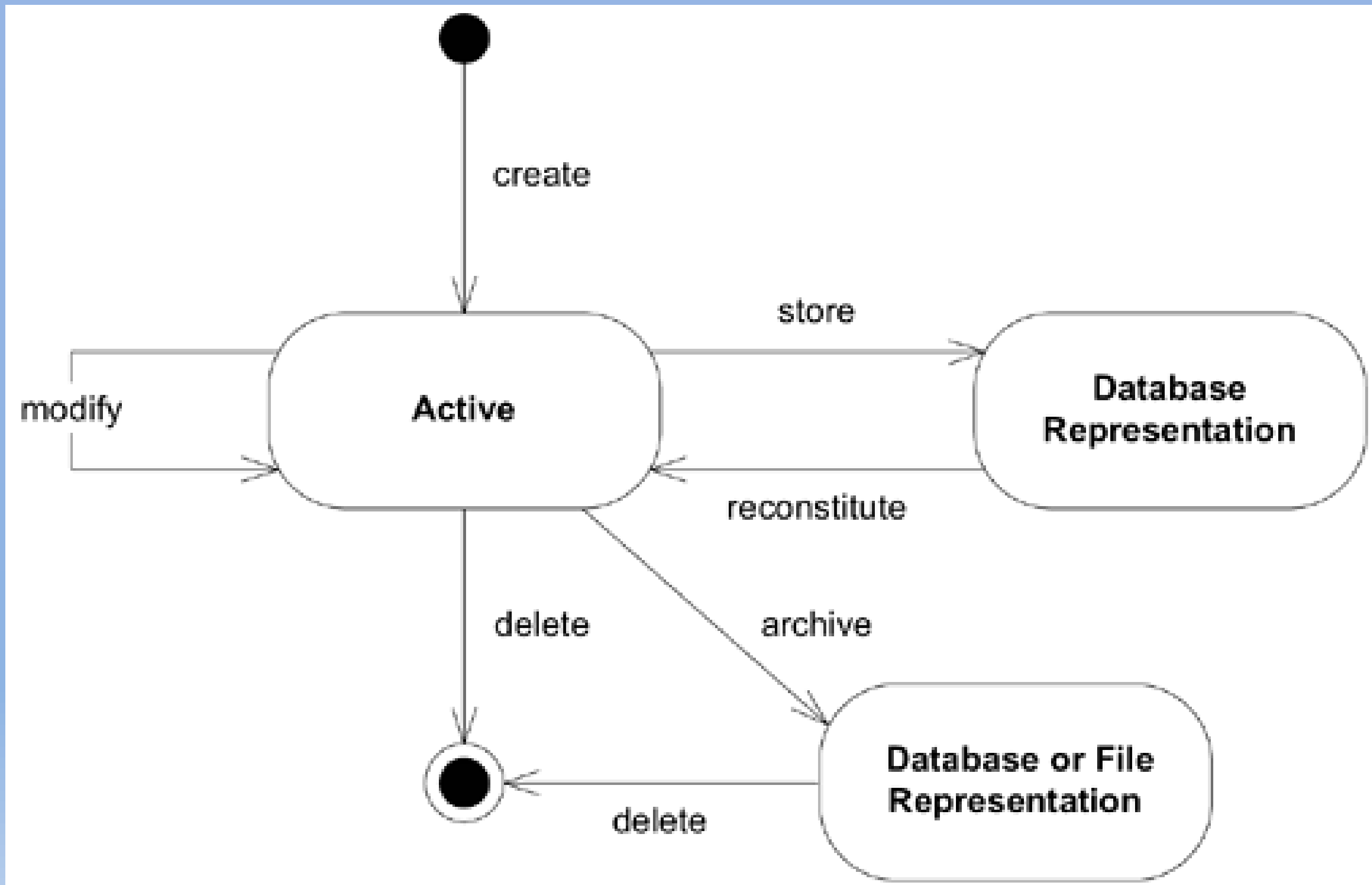
- A domain may have some parts that can be more effectively expressed and used in some other paradigm .
- Sometimes these parts are so overwhelming that there would suddenly be a need to model each part in a paradigm that fits best .
- The most effective tool again for holding such a heterogeneous model is a robust UBIQUITOUS language.

Non-Objects in an Object World ..cntd

Non-object elements into a predominantly Model Driven , object-oriented system

- Don’t find the implementation paradigm .
  - Use the model concepts that fit it instead !!
- Lean on Ubiquitous language.
  - Keep the design from diverging using connection terms even if there does not exist a rigorous connection between different parts.
- Don’t get hung up on UML diagrams.
  - Simple words can be good enough a description than horrendous implementations !!
  - Certain constraints cannot be explained in the limited structure of a UML diagram
- Be Skeptical .
  - Don’t stick to rules , constantly question them and reason instead!
  - Rules as objects will cause complications in a multiple paradigm environment

Life Cycle of a Domain object



Life cycle .. Contd

- **Problems ?**
  - Maintaining integrity throughout the lifecycle.(*i.e. ensuring that all the rules and restrictions are sustained*)
  - Preventing the model from getting swamped by the complexity of maintaining the life cycle (*i.e. Maintain it as a domain object and nothing else)*

- **Solution ?**
  - Aggregates
  - Factories
  - Repositories

Aggregates

- **Problem?**

Most domain layer objects need to engage and traverse a long way through object references & tracing becomes difficult

  – Furthermore , real time domains seldom provide sharp boundaries in this regard
  – These objects can be manipulated and used by other objects or if in UI then by human /machine users itself

- **Solution :**

An Aggregate can define the right boundaries which can help monitoring and tracing the traversal of these objects

Aggregates …contd

- An Aggregate is a cluster of Entities and Value objects.
- Each aggregate is treated as one single unit.
- It has boundary defined and maintains a strict idea of what belongs to the aggregate and what does not .
- One entity within each aggregate is chosen as the **root** , which helps control all access to the objects inside the predefined boundary.

Aggregates …Contd

- Example :

Entity : car , tire

We can have an Aggregate Unit (car → root

Tire → an object within the car root)

If tire attributes or objects need to be manipulated , the client will have to reference to it only through the root (the car in this case)

Aggregates …contd

- Root : has a global identity (car)
- Entities inside root have local identity , no global identity (tire)
- Nothing outside aggregate boundary can reference to anything inside.
- Aggregate can be deleted entirely . (i.e. , if car is deleted , the tire should go away too !)
- Invariants if present within the aggregate are also maintained by the root entity .

ROOT HELPS !! ☺

Factories

Creation of a complex object may require assembly of behaviors , attributes and responsibilities from different classes (or objects) .

Problem : This would breach encapsulation of the aggregate formed via assembly and over couples the client to the implementation of the created object.

Solution: Create Factories!!

Shift responsibility to a separate object which is a part of the domain design and provide an interface that encapsulates all the assembly operation so that the client class does not need to reference the concrete implementation directly.

Factories …. Contd

Although , constructors and destructors are available to use in object oriented programs , there is a need for more abstract construction mechanisms that are decoupled from other objects in the domain and hence a program element named factory is required .

The above diagram implies

- The client invoking a factory method by passing parameter(s)
- The factory then creates the new customized object required using the different domain objects .
- The Factory object then returns the custom object to the client .

Factories ..contd

- The basic requirements for a good factory ..
  - Each creation method is atomic .It should not be affected by other threads using similar resources until all the invariants are satisfied
  - In case of interruption or the request for an object is not possible there should be an exception thrown using the try catch or similar method.
  - Factory method should return abstract and not concreted objects . (I.e. : the abstract superclass)
  - The factory will be coupled to its arguments

Where should the Factory be positioned ?

Factory methods should be positioned according to the situation .

- In case of adding elements in a pre-existing aggregate , the root is the best place
- It can also be placed in an object that is closely responsible for spawning another object
- **STANDALONE FACTORY** : It can produce an entire object handing out reference to the root . (Used generally when an interior object of an aggregate requires a factory and the root is not responsible for it)

Factories Vs. Constructors

There are times that a simple constructor can be the best solution instead

- The class is the Type
- Client cares abt Implementation (ex: strategy)
- All attributes of the required objects are already available to the client
- The construction is very simple
- For construction of Value objects

However , Atomicity is still essential even in the case of a public constructor .

Repositories

To use an object , we need to have a reference to it

Ways to get the reference are –

- Create the object ➔ operation returns an object
- Traverse an association ➔ ask for an associated object from an object we already know .
- Searching an Object Database(Can muddle the code)
  - Ie : Domain logic moves into queries and client code and the entities, value objects become mere containers .
  - Causes swamping of the client code because of complexity
  - Developer is forced to dumb down the Domain Driven Design Itself and the whole purpose of a model driven design is lost.

Repositories ….cntd

We need to reduce the scope of the object access problem discussed above. (How?)

- Dot not concern about transient objects ➔ Use only objects which are created
- Need no query access for objects easily found by traversal ➔ e.g. : the address of a person
- Any object that's internal to aggregate has to be prohibited from access except via the root.

Repositories ..cntd

What is a Repository ?

- This will represent all the objects of a certain type as a conceptual set (Usually emulated).
- Acts as a collection except with more elaborate querying capability .
- Addition ,removal of objects are possible with the help of the machinery behind the REPOSITORY which inserts or deletes them from the database.
- Access to the root of the aggregates is possible which obviously is not possible easily by plain traversal.

Advantages of Repository

- Present client with a simple model for obtaining persistent objects and managing their life cycle.
- Decouple application and Domain design form the persistence technology
- Communicate design decisions about object access
- Allow dummy implementation that can be used in testing

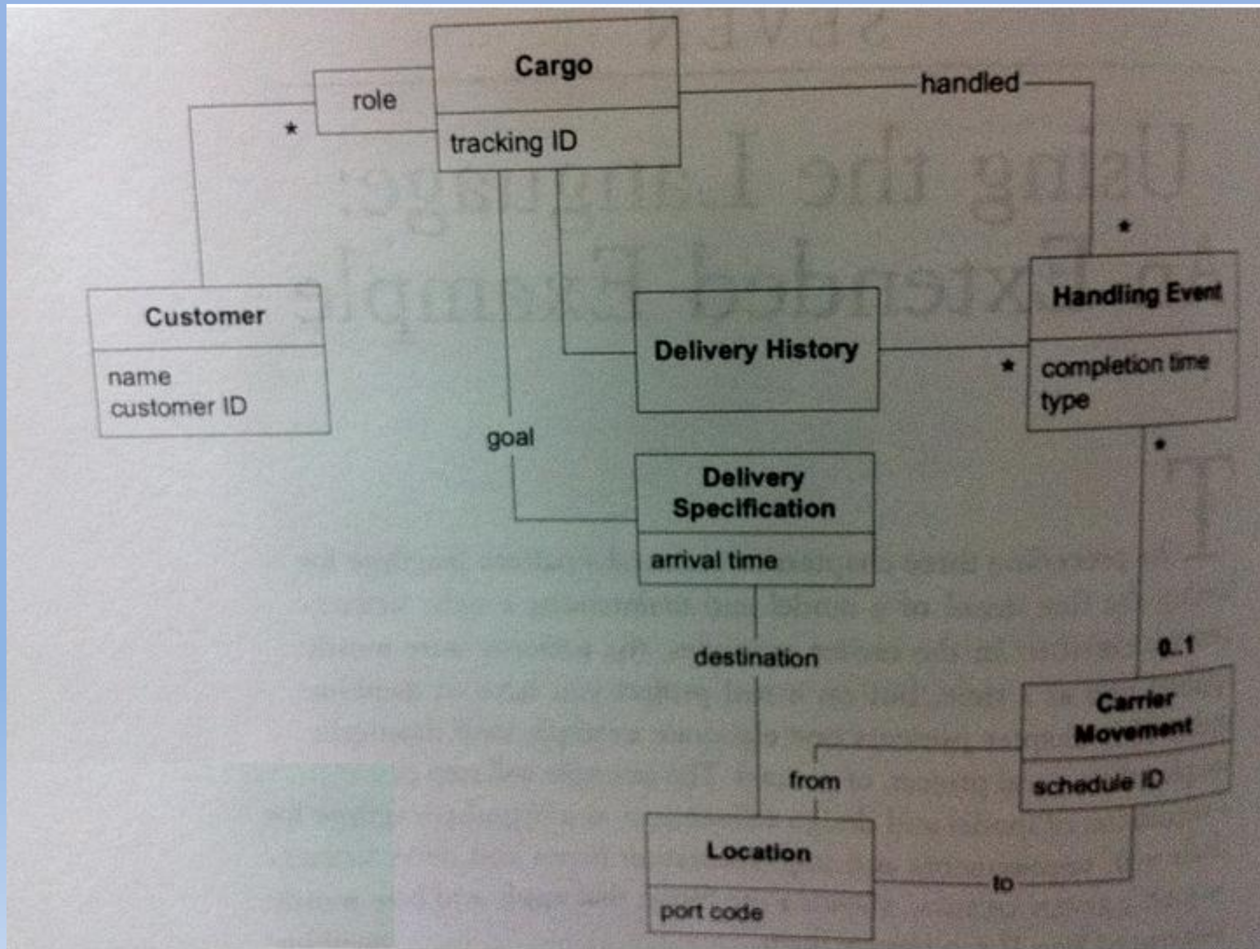
Repositories ..cntd

- **Querying a Repository**
  - Hard coded queries with specific parameters
  - Specification based query ,which allows a client to specify what it wants without concern for how it will be obtained.
- **Client Ignores Repository Implementation , not Developers**
  - Developers need to understand the implications of using encapsulated behavior ,not necessarily familiarity of the implementation.
  - Implementing depends on the technology being used for persistence
  - Hide details of inner working from client and not the developer.
- **Implementation rules**
  - Abstract the type
  - Take advantage of decoupling from the client .
  - Leave transaction control to the client .

Putting it all together :The Cargo tracking example

- **Domain ? :** Cargo shipping.
- **Requirements ? :**
  - Track handling of customer cargo.
  - Book cargo in advance.
  - Send invoices and updates to customer.
- Model of shipping domain classes
  - Customer , cargo, delivery history, delivery specification , handling event, location, carrier movement

Model diagram



- Advantages of the model
  - Delivery specification eases pressure on the cargo object
  - That there is a delivery specification class makes it easier for abstraction as other objects do not need to worry about the details of delivery.
  - A more expressive model .
  - Multiple customers are involved with the cargo, each playing a different role
  - A series of Carrier Movements satisfying the specification will fulfill the delivery goal .
  - Basic need , i.e. … tight cohesion and loose coupling are obvious

Implementation :

- Isolating the design

Coordinator classes(Application Layer)

  - Tracking query → access past and present handling of cargo
  - Booking application→ new cargo registration
  - Incident logging application → Record each event of the cargo , providing tracking query information.

Entities

These may have automatically generated identities or , manually allotted ones

  - Customer
  - Cargo
  - Handling event
  - Carrier Movement
  - Location
  - Delivery History

- **Value Objects**

These cannot be shared by 2 cargos even if they are going to the same destination.

  - Delivery Specification

- **Identifying and designing Associations.**
  - Making association traversable only from **Handling event** to **Carrier Movement .**
  - Circular reference : Cargo → Delivery History → handling event→ Cargo

- **Aggregate Boundaries**
  - Customer location ,Carrier Movement have own identity ,shared by many cargos. Hence they must be routes of their aggregates.
  - Delivery History /Specification follow suit

- **Selecting Repositories**
  - 5 aggregate roots
  - Decision of which of these elements should have a repository depends on application requirements
  - Repositories needed are
    - Customer Repository
    - Location Repository
    - Carrier Movement Repository
    - Cargo Repository
- **Walking through the Scenarios**

Check for a use case for each Aggregate and possible Scenarios

  - Delivery History → review , add , delete
  - Customer roles→ copy keyed referenced Maps
  - Tracking id → Creating a new tracking id for a cargo delivery .

Further Discussion/Topics

- The Book “ Domain Driven Design” ,by Eric Evans provides more detailed information and advanced topics henceforth .