FullStack.Cafe - Kill Your Tech Interview

Q1: What is Domain Driven Design? ☆☆

Topics: Software Architecture DDD

Answer:

Domain Driven Design is a methodology and process prescription for the development of complex systems whose focus is mapping activities, tasks, events, and data within a problem domain into the technology artifacts of a solution domain.

It is all about trying to make your software a model of a real-world system or process.

Q2: What is a *Domain Model* in DDD? ☆☆

Topics: DDD

Answer:

The **Domain Model** is a *software model* of the very specific business domain you are working in. Often it's implemented as an *object model*, where those objects have *both data and behavior* with literal and accurate business meaning.

Creating a *unique*, carefully crafted *domain model* at the heart of a core, strategic application or subsystems essential to practicing DDD. With *DDD your domain models* will tend to be smallish, very focused. Using DDD, you never try to model the whole business enterprise with a single, large domain model.

Q3: What is the distinction between *Value Types* and *Entities* in DDD? ☆☆

Topics: DDD

Answer:

- Classical implementation of DDD differentiates between **Value Types** and **Entity Types**, based on their *mutability and notion of identity*.
- **Value types** are *immutable* and do not convey enough information on their own

For example:

- `Color` could be a **value type** where *color type does not hold any meaning in itself* but when attached to an *entity* like a `shirt` or a `car` (for example a `red shirt` or a `black shirt`) which mean something in the domain.

On the contrary, an **entity** has a *lifecycle*. These are the types that are *mutable* and go through changes through different lifecycle events.

- `Order` could be an **entity** that goes through different lifecycle events such as *item added to order or item removed from order. Each lifecycle event mutates the entity*.

Q4: List some advantages of *Domain-Driven Design*. Why developers shall use it? ☆☆

Topics: DDD

Answer:

Simpler communication:

- Thanks to the *Ubiquitous Language*, *communication between developers and teams become much easier*.
- As the *Ubiquitous language* is likely to contain simpler terms developers refer to, *there's no need for complicated technical terms*.

More flexibility:

- As DDD is object-oriented, everything about the domain is *based on and object is modular and caged*.
- Thanks to this, the *entire system can be modified and improved regularly*

The domain is more important than `UI/UX`:

- As the domain is the central concept, developers will *bild applications suited for the particular domain*.
- This won't be another *interface-focused application*.
- Although you shouldn't leave out `UX`, using the DDD approach means that the *product targets exactly the users* that are directly connected to the domain.

Q5: What are the fundamental *components* of *Domain-Driven Design*? ☆☆

Topics: DDD

Answer:

Ubiquitous language and Unified Model Language (`UML`):

- *Ubiquitous language* is a common language to communicate within a project.
- It's because *designing a model* is a collaborative effort of *software designers*, *domain experts*, and *developers* that it requires a common language to communicate with.
- It removes **misunderstandings**, and **misinterpretations**.
- *Communication gaps so often lead to bad software - ubiquitous language* minimizes these gaps.

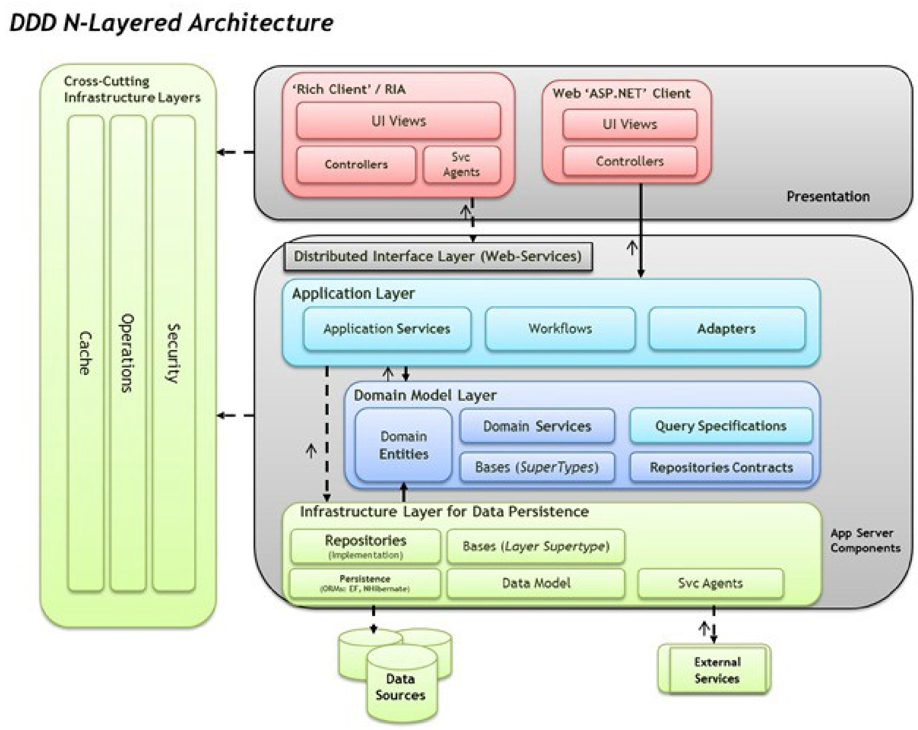
Multilayered architecture is a common solution for Domain-Driven Design and contains four layers:

- *Presentation* layer or (UI)
- *Application* layer
- *Domain* layer
- *Infrastructure* layer

Artifacts used in Domain-Driven Design to **express**, **create** and **retrieve** domain models mapping business domain concepts into software artefacts. It’s about organizing code artifacts in alignment with business problems, using the same common, ubiquitous language. Some of the **artefacts** are:

- DAO interface and implementation class
- Factories
- Repositories
- Domain Delegate (if needed)

- Facade
- DTO's
- Unit Tests for the above classes (including test class and test data)
- Configuration files



Q6: How would you describe what is *Domain Driven Design* from the developer point of view? ☆☆

Topics: DDD

Answer:

Domain Driven Design is a methodology and process prescription for the development of complex systems whose focus is mapping **activities**, **tasks**, **events**, and **data** within a problem domain into the technology **artifacts** of a solution domain.

The emphasis of Domain Driven Design is to understand the problem domain in order to create an abstract model of the problem domain which can then be implemented in a particular set of technologies. Domain Driven Design as a methodology provides guidelines for how this model development and technology development can result in a system that meets the needs of the people using it while also being robust in the face of change in the problem domain.

The process side of Domain Driven Design involves the collaboration between domain experts, people who know the problem domain, and the design/architecture experts, people who know the solution domain. The idea is to have a shared model with shared language so that as people from these two different domains with their two different perspectives discuss the solution they are actually discussing a shared knowledge base with shared concepts.

Q7: What is *Domain* in DDD? ☆☆

Topics: DDD

Answer:

Domain is the field for which a system is built. Airport management, insurance sales, coffee shops, orbital flight, you name it.

It's not unusual for an application to span several different domains. For example, an online retail system might be working in the domains of shipping (picking appropriate ways to deliver, depending on items and destination), pricing (including promotions and user-specific pricing by, say, location), and recommendations (calculating related products by purchase history).

Q8: What is a *Specification* in DDD? ☆☆

Topics: DDD

Answer:

- A **Specification** represents a *business rule that needs to be satisfied* by at least part of the domain model,
- You can also use **Specification** for query *criteria*
  - For example, you can query for all objects that satisfy a given *specification*.

Q9: What is Domain in DDD? ☆☆☆

Topics: Software Architecture DDD

Answer:

In order to create good software, you have to know what that software is all about. You cannot create a banking software system unless you have a good understanding of what banking is all about, one must understand the domain of banking.

Domain is the field for which a system is built. Airport management, insurance sales, coffee shops, orbital flight, you name it.

It's not unusual for an application to span several different domains. For example, an online retail system might be working in the domains of shipping (picking appropriate ways to deliver, depending on items and destination), pricing (including promotions and user-specific pricing by, say, location), and recommendations (calculating related products by purchase history).

Q10: What is a Model in DDD? ☆☆☆

Topics: Software Architecture DDD

Answer:

> A **model** is a useful approximation to the problem at hand.

An `Employee` class is not a real employee. It models a real employee. We know that the model does not capture everything about real employees, and that's not the point of it. It's only meant to capture what we are interested in for the current context.

Different domains may be interested in different ways to model the same thing. For example, the salary department and the human resources department may model employees in different ways.

Q11: What is the difference between DTOs and ViewModels in DDD? ☆☆☆

Topics: Software Architecture DDD

Answer:

- The canonical definition of a **DTO** is the data shape of an object *without any behavior*. Generally DTOs are used to ship data from one layer to another layer across process boundries.

- **ViewModels** are the model of the view. ViewModels typically are full or *partial* data from one or *more* objects (or DTOs) plus any additional members specific to the view's behavior (methods that can be executed by the view, properties to indicate how toggle view elements etc...). In the MVVM pattern the ViewModel is used to isolate the Model from the View.

Q12: Explain the different *layers* in DDD ☆☆☆

Topics: DDD Layering & Middleware

Answer:

User Interface (or Presentation Layer):

- Responsible for *showing information* to the user and *interpreting the user's commands*.
- The *external actor* might sometimes be another *computer system* rather than a human user.

Application Layer:

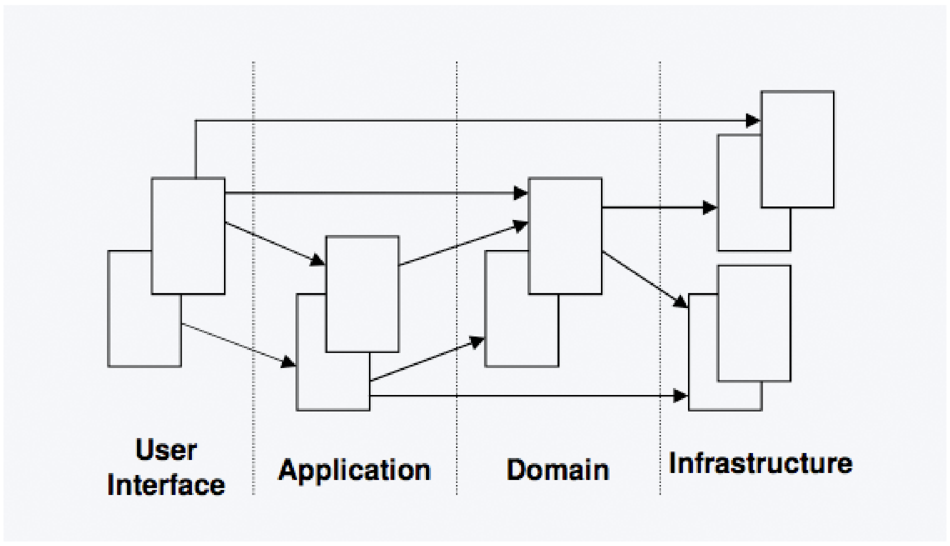
- Defines the jobs the *softwares supposed to do* and *directs the expressive domain objects* to work out problems.
- The tasks this layer is responsible for are meaningful to the *business* or *necessary for interaction with the application layers* of other systems.
- This layer is *kept thin*.
- It does not contain business rules or knowledge, but only *coordinates tasks and delegates work* to collaborations of domain object in the next layer down.
- It *does not have state reflecting the business situation*, but it can have *state that reflects the progress of a task* for the user or the program.

Domain Layer (or Model Layer):

- Responsible for representing *concepts* of the business, *information* about the business situation, and *business rules*.
- *State* that reflects the business situation is *controlled and used here*, even though the technical details of *storing it are deegated to the infrasturcture*.
- This layer is the *heart of business software*.

Infrastructure Layer:

- Provides *generic technical capabilities* that support the higher layers: *message sending* for the application, *persistence* for the domain, *drawing widgets* for the UI, and so on.
- The *infrastructure layer* may also support the pattern of *interactions between the four layers* through an architectural framework.



Q13: What types of issues does an *Aggregate* solve in DDD? ☆☆☆

Topics: DDD

Answer:

Basically *Aggregates* help:

- *Simplify* the models when they start getting out of hand
- *Isolate* complex business rules
- *Deal with performance issues* when loading large object graphs into memory
- *Allow flexibility* to more easily deal with future unexpected business requirements

Q14: What does *Bounded Context* mean in DDD? ☆☆☆

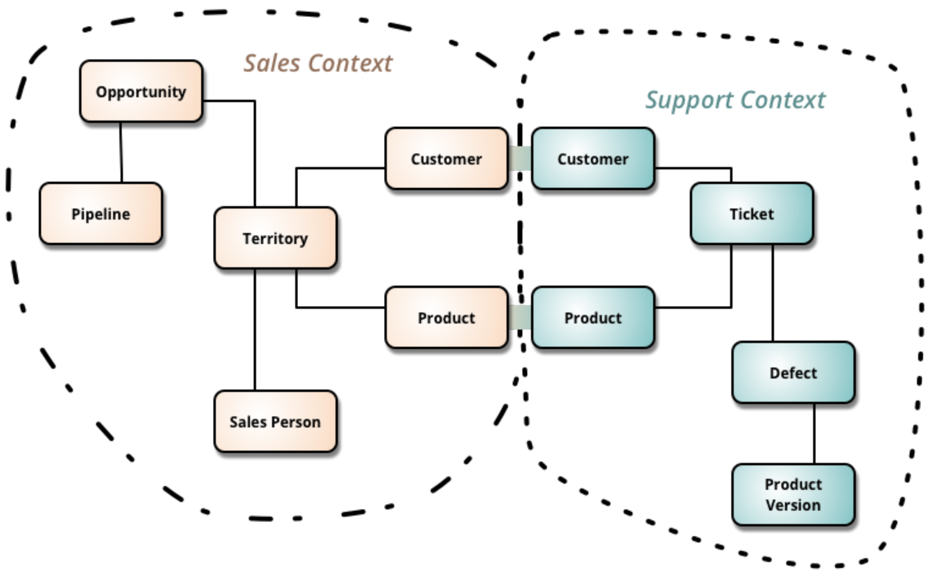
Topics: DDD

Answer:

- A *Bounded Context* is a conceptual boundary around *parts of the application and/or the project* in terms of business *domain*, *teams*, and *code*.
- It *groups* related components and concepts and avoids ambiguity as some of these could have similar meanings without a clear context.

Bounded Contexts have both unrelated concepts (such as a support ticket only existing in a customer support context) but also share concepts (such as products and customers). Different contexts may have completely different models of common concepts with mechanisms to map between these polysemic concepts for integration. Several DDD patterns explore alternative relationships between contexts.

Consider:



Q15: What is the differences between *Strategic Patterns* and *Tactical Patterns*? ☆☆☆

Topics: DDD

Answer:

Domain-Driven Design (DDD) is divided into *strategic patterns* and *tactical patterns*:

- The **Strategic Pattern** consists of things like *Bounded context*, *Ubiquitous language*, and *Context Map*
- The **Tactical Pattern** consists of concepts like *Value Types*, *Entities*, and *Aggregate*.

Consider:

- *Strategic patterns* are *easy to map* to any language.
- *Strategic patterns* mostly *cover a higher-level design of software* like *bounded context*, *context map*, *anti-corruption layer* and *patterns for bounded context* integration.
- *Strategic patterns do not depend on the programming language or framework* used.
  - However, the *tactical pattern* relies on *programming language* constructs and paradigms.

Q16: Mention how can you give objects an *Unique Identity* in DDD? ☆☆☆

Topics: DDD

Answer:

There are different ways to create a **unique identity** for objects:

- Using the **primary key** in a table.
- Using an **automated generated ID** by a domain module. A domain program *generates the identifier and assigns it to objects* that is being persisted among different layers
- A few real-life objects carry **user-defined** identifiers themselves. For example, each country has its own country codes for dialling ISD calls.
- **Composite key**. This is a *combination of attributes* that can also be used for creating an identifier.

Q17: What does it mean to focus on *Problem Space* rather than the *Solution Space*? ☆☆☆

Topics: DDD

Answer:

By applying DDD we first try to study and understand the domain (or field) we work on. One of the practices (taken from XP) would be the writing of stories that occur in the problem domain. From these you can identify your **use cases** and **objects** for your design. They *emerge* and tell you what needs to be in the **solution**, and how they will need to interact with each other.

The concepts described by the UL will form the basis of your object-oriented design and **system artifacts**. DDD provides some clear guidance on how your objects should interact, and helps you divide your objects into the following categories:

- **Value Objects**, which represent a value that might have sub-parts (for example, a date may have a day, month and year)
- **Entities**, which are objects with identity. For example, each Customer object has its own identity, so we know that two customers with the same name are not the same customer
- **Aggregate Roots** are objects that own other objects. This is a complex concept and works on the basis that there are some objects that don't make sense unless they have an owner. For example, an 'Order Line' object doesn't make sense without an 'Order' to belong to, so we say that the Order is the aggregate root, and Order Line objects can only be manipulated via methods in the Order object

By practicing DDD and exploring business domain, you build a ubiquitous language (UL), which is basically a conceptual description of the **system** that implements the **solution**.

Q18: What is the purpose of *Service* in Domain-Driven Design (DDD) ☆☆☆

Topics: DDD

Answer:

According **Eric Evans** Domain-Driven Design:

> When a significant process or transformation in the domain is not a natural responsibility of an ENTITY or VALUE OBJECT, add an operation to the model as 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.

These types of services can be identified more specifically as domain services and are part of the domain layer. Domain services are different from infrastructural services because they embed and operate upon domain concepts and are part of the ubiquitous language.

Q19: Explain the concept of *Repository* in the context of DDD ☆☆☆

Topics: DDD

Answer:

A **Repository** represents all objects of a type as a *conceptual set* (usually emulated). It acts like a collection, except with more elaborate *querying ability.* Objects of the appropriate type are added and removed, and the machinery behind the repository inserts them or deletes them from the database.

For each type of object that requires global access, create an object that can provide the illusion of an in-memory collection of all objects of that type. The primary use case of a repository: given a key, return the correct root entity. The repository implementation acts as a module, which hides your choice of persistence strategy.

For each aggregate root (AR) you should have a repository. As a minimum the repository would probably have a `void Save(Aggregate aggregate)` and a `Aggregate Get(Guid id)` method. The returned aggregate would always be fully constituted.

Q20: What is the difference between *Value* vs *Entity Objects* in DDD? ☆☆☆

Topics: DDD

Answer:

Think of:

- **Value Objects** as static data that will never change
  - *immutable*, has no *lifecycle*
  - has _no identity _
  - example: `Color`
- **Entities** as data that evolves in your application
  - *mutable*, has *lifecycle*
  - has *identity*
  - example: `Customer`

FullStack.Cafe - Kill Your Tech Interview

Q1: What is the difference between *Domain Objects*, *POCO*, *Services*, *Repositories* and *Entities*? ☆☆☆

Topics: DDD Layering & Middleware

Answer:

- `POCO` - Plain Old %Insert_Your_Language% Object. A type with no logic in it. It just stores data in memory. You'd usually see just auto properties in it, sometimes fields and constructors.
- `Domain Object` an instance of a class that is related to your domain. I would probably exclude any satellite or utility objects from domain object, e.g. in most cases, domain objects do not include things like logging, formatting, serialisation, encryption etc - unless you are specifically building a product to log, serialise, format or encrypt respectively.
- `Model Object` I think is the same as `Domain object`. Folks tend to use this interchangeably.
- `Entity` a class that has `id`
- `Repository` a class that speaks to a data storage from one side (e.g. a database, a data service or ORM) and to the service, UI, business layer or any other requesting body. It usually hides away all the data-related stuff (like replication, connection pooling, key constraints, transactions etc) and makes it simple to just work with data
- `Service` a class (or microservice) that provides some functionality usually via public API. Depending on the layer, it can be for example a RESTful self-contained container, or class that allows you to find a particular instance of needed type.

Q2: What is the difference between *Behavior-Driven Development (BDD)* vs *Domain-Driven Design (DDD)*? ☆☆☆

Topics: Software Architecture DDD

Answer:

- **DDD** focuses on defining the vocabulary in that language: actors, entities, operations, ... An important part of DDD is also that the *ubiquitous language* can be clearly seen in the code, too, not only in communication between the implementor and the domain expert. So an extreme view of DDD is quite static: it describes the finished system as a whole.

- **BDD** focuses on *defining user* stories or *scenarios*. It is closely related to an incremental process, but it can also be viewed as static: it describes all the *interactions* between users and the finished system.

DDD and BDD *can* be applied with no overlapping: BDD user stories can be written using only UI vocabulary (buttons, labels, ...), and DDD can avoid mentioning interactions.

But ideally these two overlap and work together: the BDD stories are rich in the ubiquitous language, describing the user experience with domain concepts. And DDD makes sure the stories can be found in the code.

Q3: What is *Event Storming* in DDD? ☆☆☆

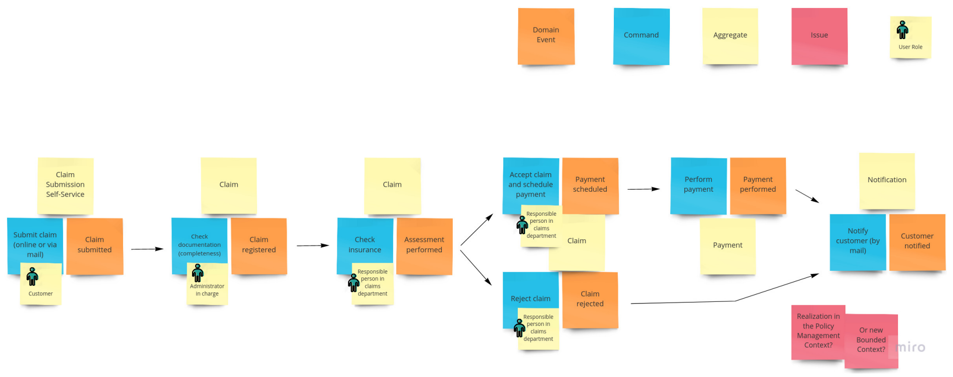
Topics: DDD

Answer:

Event storming is not a notation. It's a very interactive team **practice** based on sticky notes put on a board and rearranged as the modelling progresses. It starts with the identification of the events, seeks the commands that cause them, and then looks at the involved aggregates/entities. It engages both *Domain Experts* and *developers* in a fast-paced learning process.

The output of an event storming describes the domain using the following DDD concepts:

- Domain Events
- Commands (that cause the Domain Events)
- Aggregates (or Aggregate root Entities)
- Issues (optional)
- User Roles (optional)
- Views / Read Models (optional)
- Bounded Contexts
- Subdomains
- Event Flows (optional)


Domain Event
Command
Aggregate
Issue
User Role
Claim Submission Self-Service
Submit claim (online or via mail)
Customer
Claim submitted
Claim
Check documentation (completeness)
Administrator in charge
Claim registered
Claim
Check insurance
Responsible person in claims department
Assessment performed
Accept claim and schedule payment
Responsible person in claims department
Payment scheduled
Claim
Reject claim
Responsible person in claims department
Claim rejected
Perform payment
Payment performed
Payment
Notification
Notify customer (by mail)
Customer notified
Realization in the Policy Management Context?
Or new Bounded Context?
miro

Q4: What is the *Command and Query Responsibility Segregation (CQRS)* Pattern? ☆☆☆

Topics: DDD Software Architecture Design Patterns

Answer:

In traditional architectures, the same data model is used to query and update a database. That's simple and works well for basic CRUD operations.

CQRS stands for **Command and Query Responsibility Segregation**, a pattern that separates read and update operations for a data store. CQRS separates *reads* and *writes* into different models, using **commands** to update data, and **queries** to read data.

- Commands should be task-based, rather than data-centric. ("Book hotel room", not "set ReservationStatus to Reserved").
- Commands may be placed on a queue for asynchronous processing, rather than being processed synchronously.
- Queries never modify the database. A query returns a DTO that does not encapsulate any domain knowledge.

Q5: Name some benefits of *CQRS Pattern* ☆☆☆

Topics: DDD Software Architecture Design Patterns

Answer:

Benefits of CQRS include:

- **Independent scaling**. CQRS allows the read and write workloads to scale independently, and may result in fewer lock contentions.
- **Optimized data schemas**. The read side can use a schema that is optimized for queries, while the write side uses a schema that is optimized for updates.
- **Security**. It's easier to ensure that only the right domain entities are performing writes on the data.
- **Separation of concerns**. Segregating the read and write sides can result in models that are more maintainable and flexible. Most of the complex business logic goes into the write model. The read model can be relatively simple.
- **Simpler queries**. By storing a materialized view in the read database, the application can avoid complex joins when querying.

Q6: Describe what is the *Event Sourcing Pattern* ☆☆☆

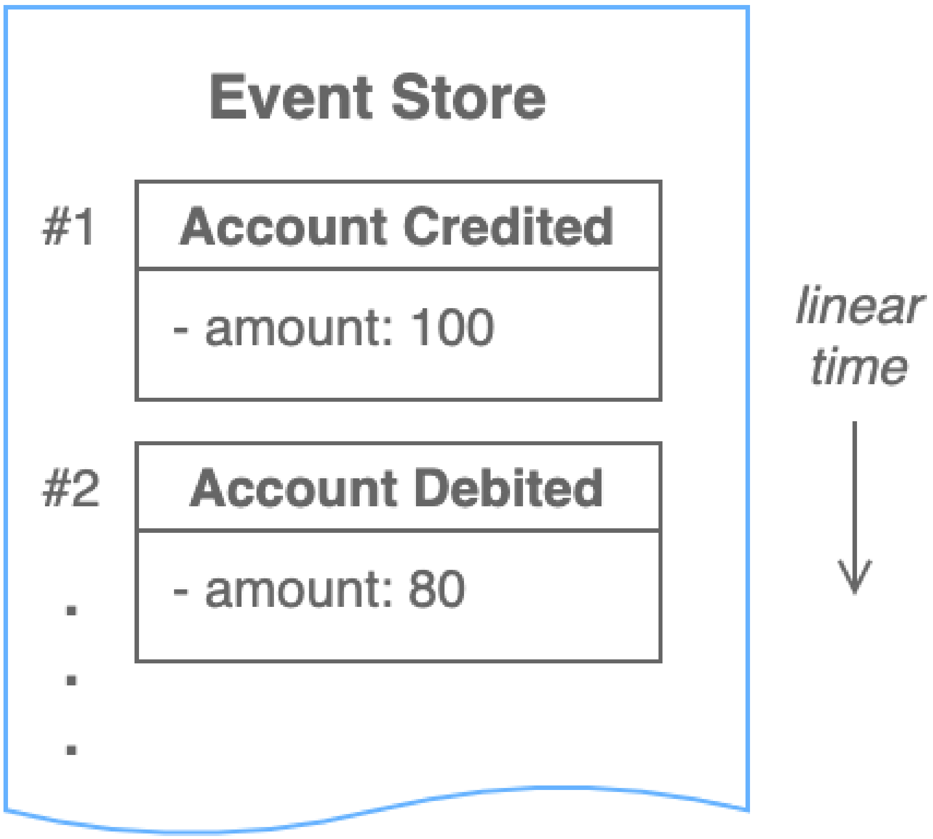
Topics: DDD Software Architecture Design Patterns

Answer:

Event Sourcing is a pattern for the recording of state in a *non-destructive* way. **Each change of state is appended to a log**. Because the changes are non-destructive, we preserve the ability to answer queries about the state of the object at any point in its life cycle.

- Event sourcing enables *traceability* of changes.
- Event sourcing enables *audit logs* without any additional effort.
- Event sourcing makes it possible to *reinterpret* the past.
- Event sourcing *reduces the conflict* potential of simultaneously occurring changes.
- Event sourcing enables *easy versioning* of business logic.

Event Sourcing is not necessary for **CQRS**. You can combine Event Sourcing and CQRS. This kind of combination can lead us to a new type of CQRS. It involves modelling the state changes made by applications as an immutable sequence or log of events.



Q7: What are *Aggregates* in Domain-Driven Design? ☆☆☆☆

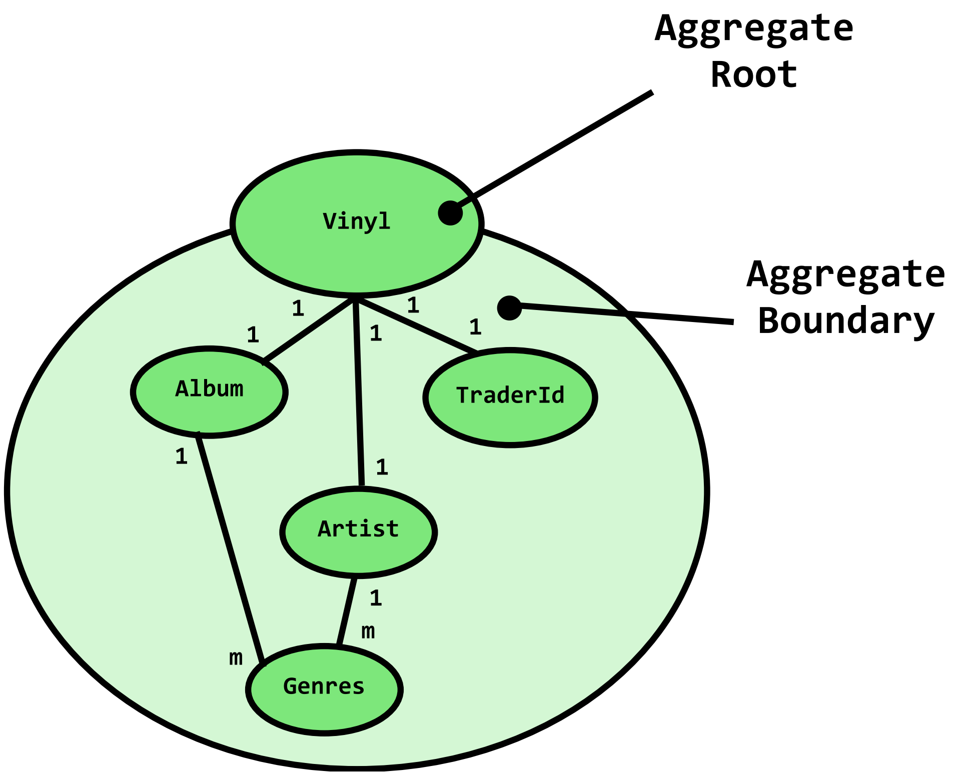
Topics: DDD

Answer:

- *Aggregate* is a **pattern** in Domain-Driven Design.
- A DDD *aggregate* is a *cluster of domain objects* that can be *treated as a single unit* for the purpose of data changes.
- An *aggregate* will have one of its componet objects be the **aggregate root**.
- Any references from outside the *aggregate* should only go to the **aggregate root**.
- The **root** can thus *ensure the integrity of the aggregate* as a whole.

Note:

- *Aggregates* are the *basic element of transfer of data storage* - you request to load or save whole *aggregates*.
- **Transactions** should not cross *aggregate boundaries*.



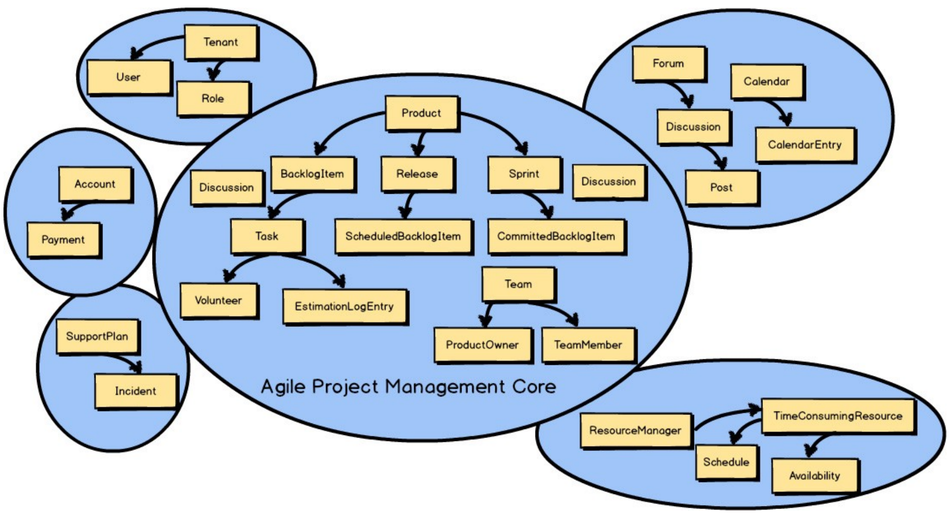
Q8: What is *Context Mapping's* purpose in DDD? ☆☆☆☆

Topics: DDD

Answer:

Context mapping is a tool that allows you to identify the relationship between bounded contexts and the relationship between the teams that are responsible for them. Context mapping helps:

- *Identifying and graphically documenting* each **Bounded Context** in the project is called *Context Mapping*
- *Context Maps* help better understand *how **Bounded Contexts** and teams relate and communicate* with each other
- They *give a clear idea of the actual boundaries* and help teams visually describe the conceptual subdivisions of the system's design



Q9: What exactly are the *Anti-Corruption* layers in DDD? Provide an example. ☆☆☆☆

Topics: DDD

Answer:

An Anti-Corruption Layer (ACL) is a set of patterns placed between the domain model and other bounded contexts or third-party dependencies. The intent of this layer is to prevent the intrusion of foreign concepts and models into the domain model. This layer is typically made up of several well-known design patterns such as **Facade** and **Adapter**. The patterns are used to map between foreign domain models and APIs to types and interfaces that are part of the domain model.

An Anti-Corruption Layer (`ACL`) encourages you to create gatekeepers that work to prevent non-domain concepts from leaking into your model. ACLs are about dealing with conceptual mismatches, not poor quality. An ACL is not just to be used with Messy code, but as a means to communicate between bounded contexts. It translates from one context to the other, so that data in each context reflects the language and the way that that context thinks and talks about the data.

Use case to look at in your system to apply ACL:

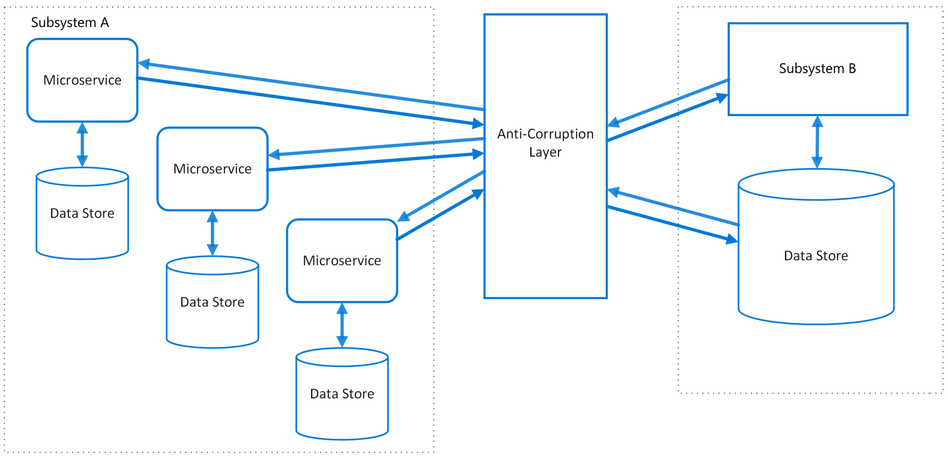
When you have a system that's out of your domain, yet your business needs require you to work with that other domain. You do not want to introduce this other domain into your own, therefore corrupting it, so you will translate the concept of your domain, into this other domain, and vice-versa.

Example:

One system views the customer as having a name and a list of strings, one for each transaction. You view Profiles as stand-alone classes having a name, Transactions as stand-alone classes having a string, and Customer as having a Profile and a collection of Transactions.

So you create an ACL layer that will let translate between your Customer and the other system's Customer. This way, you never have to use the other system's Customer, you simply need to tell the ACL:

> "give me the Customer with Profile X, and the ACL tells the other system to give it a Customer of name X.name and returns you a Customer with Profile X."



Q10: What is the difference between *Factory* and *Repository* in DDD? ☆☆☆☆

Topics: DDD

Answer:

The difference between a repository and a factory is that

- a **repository** represents an abstract *persistent storage*, while
- a **factory** is responsible for *building* an object.

So, for example, let's say I'm registering a user. I'll get my user object from a factory

```
IUser user = userFactory.Create(name, email);
```

Then pass it to the repository, which will be responsible for dealing with it.

```
userRepository.Insert(user);
```

Factories in DDD can be thought of as a way to hide new, an abstraction of the details of instantiation. It allows you to very effectively program to an interface rather than a concrete.

In addition, this allows repositories to be focused on their entity type, and thus the use of generics becomes very powerful.

Q11: What is main difference between *Domain* vs *Application* vs *Infrastructure Services*? ☆☆☆☆

Topics: DDD Layering & Middleware

Answer:

Services come in 3 flavours: **Domain Services**, **Application Services**, and **Infrastructure Services**.

- **Domain Services** : Encapsulates *business logic* that doesn't naturally fit within a domain object, and are **NOT** typical CRUD operations – those would belong to a *Repository*.
- **Application Services** : Used by external consumers to talk to your system (think *Web Services*). If consumers need access to CRUD operations, they would be exposed here.
- **Infrastructure Services** : Used to abstract technical concerns (e.g. MSMQ, email provider, etc).

Keeping Domain Services along with your Domain Objects is sensible – they are all focused on domain logic. And yes, you can inject Repositories into your Services. Application Services will typically use both Domain Services *and* Repositories to deal with external requests.

Q12: Provide some examples of *Infrastructural Services* in DDD ☆☆☆☆

Topics: DDD Layering & Middleware

Answer:

- An **email infrastructure service** can handle a domain event by generating and transmitting an appropriate email message.
- Another infrastructural service can handle the same event and send a **notification via SMS** or other channel.

- A **repository implementation** is also an example of an infrastructural service. The interface is declared in the domain layer and is an important aspect of the domain. However, the specifics of the communication with durable storage mechanisms are handled in the infrastructure layer.

Q13: What is the difference between *Infrastructure Service* and *Repository* in DDD? ☆☆☆☆

Topics: DDD Layering & Middleware

Answer:

- A **Repository** directly correlates to an *Entity*, often an *Aggregate Root*.
- When working at the conceptual level, a DDD **Repository** differs from a DDD service in that it is specifically tied to Entity *persistence*.
- A **Service** can address any Domain, Application, or Infrastructure problem you may have.
- A **Service** defines behaviors that don't really belong to a single Entity in your domain.

Q14: What is the difference between *DAO* and *Repository* in DDD? ☆☆☆☆

Topics: DDD

Answer:

- `DAO` is an abstraction of **data persistence**.
- `DAO` would be considered closer to the database, often table-centric.
- `Repository` is an abstraction of **a collection of objects**.
- `Repository` would be considered closer to the Domain, dealing only in Aggregate Roots.

`Repository` could be implemented using `DAO`'s, but you wouldn't do the opposite. Also, a `Repository` is generally a narrower interface. It should be simply a collection of objects, with a `Get(id)`, `Find(ISpecification)`, `Add(Entity)`.

A method like `Update` is appropriate on a `DAO`, but not a `Repository` - when using a `Repository`, changes to entities would usually be tracked by separate *UnitOfWork*.

It does seem common to see implementations called a `Repository` that is really more of a `DAO`, and hence I think there is some confusion about the difference between them.

Q15: What is *Core Domain*, *Supporting Subdomain*, and *Generic Subdomain* in DDD? ☆☆☆☆

Topics: DDD

Answer:

- **Core Domain** - the most important subdomain, which is essential for the business. Without it, the business would fail. Start with the core domain if you ever need to pick the first solution to implement. The Core Domain essentially is the set of Bounded Contexts that worth the application's development from the customer point of view.
- **Supporting Subdomain** - subdomain, which is less valuable for business than the Core Domain. Without it, business maybe can even survive for some time. But it still is quite important (supports Core Domain), it also

is specific for the domain and has to be developed. In this case, for some reason, we can't buy an existing software or component to solve the problem.
- **Generic Subdomain** - subdomain which is less valuable for business than the Core Domain. It also is generic enough to allow buying it off the shelf (unlike supporting domain).

Q16: Where DTO should be implemented, in a *Domain Layer* or in an *Application Service Layer*? Explain. ☆☆☆☆☆

Topics: DDD Software Architecture Layering & Middleware

Answer:

DTOs that are exposed to the outside world become part of a contract. Depending on their form, a good place for them is either the Application Layer or the Presentation Layer.

- If the DTOs are only for presentation purposes, then the Presentation Layer is a good choice.
- If they are part of an API, be it for input or output, that is an Application Layer concern. The Application Layer is what connects your domain model to the outside world.

Don't put your DTO in the Domain Layer. The Domain Layer does not care about mapping things to serve external layers (the domain does not know there is a world outside of its own). The Application Layer is what connects your domain model to the outside world. Presentation Layer should access the domain model only through the Application Layer.

Q17: Can we use the *CQRS* without the *Event Sourcing*? ☆☆☆☆☆

Topics: DDD Software Architecture Design Patterns

Answer:

Event Sourcing (ES) is optional and in most cases complicates things more than it helps if introduced too early. Especially when transitioning from legacy architecture and even more when the team has no experience with **CQRS**.

Most of the advantages being attributed to **ES** can be obtained by storing your events in a simple Event Log. You don't have to drop your state-based persistence, (but in the long run you probably will, because at some point it will become the logical next step).

My recommendation: Simplicity is the key. Do one step at a time, especially when introducing such a dramatic paradigm shift. Start with simple CQRS, then introduce an Event Log when you (and your team) have become used to the new concepts. Then, if at all required, change your persistence to Event Sourcing.