Discovering the Domain Architecture through DDD

DDD is a tool used to organize business logic in software. It involves the Ubiquitous Language, bounded contexts, and context mapping. The Ubiquitous Language defines a common vocabulary to express models and processes, and is used regularly in all forms of spoken and written communication. Bounded contexts are segments of the domain that are preferable to treat separately, and all identified contexts and their explicit relationships form a map. This helps make better sense of user requirements and reduces the risks of misunderstandings.

The ubiquitous language is the language of the domain model being built, which is very close to the natural language of the business domain. It is iteratively composed along the way and changed over time to better reflect the understanding of the mechanics of the domain. It is rigorous, fluent, and unambiguous so that it meets expectations of both domain experts and technical people. To be effective, the language must be used in all forms of communication and interaction, such as user stories, request-for-changes, meetings, and emails. Changes to the ubiquitous language often implies changes to the source code as well.

Ubiquitous language is a combination of business and technical jargon that should be used in meetings to share ideas and purposes. It should consist of words and verbs that reflect the semantics of the business domain, without using terms like coupons or gift cards. Ambiguity and synonyms should be removed as discovered along the way, and matching concepts should be named equally. It is important to avoid acronyms in the ubiquitous language and instead introduce new words that retain the original meaning of the acronym. The most important details of the phrases ubiquitous language, DDD, and code are that the ubiquitous language is a glossary, it is written in the official, natural language of the project, hiring programmers with expertise in a given business domain reduces the troubles with building and maintenance of the ubiquitous language, and when it comes to turning the ubiquitous language in code, naming convention is absolutely critical.

The ubiquitous language is a language that is constantly evolving to represent both the evolving business and the growing architect's understanding of the domain. To combat ambiguity and duplication of concepts, domain-driven design introduces the bounded context, which is the delimited space within the domain that gives any elements in the ubiquitous language a unique and unambiguous, well-defined meaning. Bounded contexts serve three main purposes: to fight ambiguity and duplication of concepts, simplify software module design, and integrate legacy code and external components.

Bounded context is an area of the domain model that has its own ubiquitous language, its own independent implementation based on a supporting architecture, such as CQRS, and a public documented interface to interact with other bounded contexts. It is used when the same term means slightly different things to different people, when the same term is used to refer to completely different things, or when there is dependency on external subsystems. When developing a software system, it is important to consider the different functional areas such as the club website, log in subsystem, backoffice, and reporting module. Additionally, external systems like PayPal can be used for various purposes. It is possible to manage complexity by introducing bounded contexts, but this may not be easy. Another solution is to isolate shared elements in a separate kernel and load the right one on demand. Another solution is to give each complex the implementation and names it needs.

Bounded contexts often reflect the physical structure of the organization that commissions the project. Bounded contexts are independent implementations of a system, with each bounded context having its own supporting architecture and technologies. The role of U and D in the context map indicates the direction of the relationship between upstream and downstream contexts. Relationships can be conformist or customer/supplier, with the latter allowing for negotiation between the teams managing the involved context. Upstream context influences downstream, while downstream context influences vice versa.

Event storming is a technique used to explore the business domain and identify key events and commands. It involves getting developers and experts in a room to ask questions and find answers, with the goal of identifying relevant domain events. The difference between conformist and customer/supplier is in the margin for negotiation and talk, while the partner relationship refers to a mutual dependency set between two involved contexts. The shared kernel refers to a piece of the model shared by multiple contexts, and the anti-corruption layer provides an additional interface to deal with no matter what happens in the upstream context. In an ecommerce scenario, sticky notes can be used to identify and explain events in a system. These events can be caused by a user command, asynchronous happenings, or following events. The modeling surface should contain sticky notes for all identified events. An external figure, called the facilitator, should lead the session and ensure focus is not lost, regardless of domain expertise. The facilitator guides others to understand and formalize the domain through event storming, which provides a quick way to gain a comprehensive vision of the business domain. It also helps form an idea about the types of users in the system and where the user experience is important. Although it may not have detailed instructions, one can find them using a search engine.

The DDD Layered Architecture

Module three of the Pluralsight course on Modern Software Architecture discusses the layered architecture, which is a multilayer segmentation of a software system made up of presentation, business, and data access. The module covers the different layers of the architecture, including the presentational layer, application layer, domain layer, and infrastructure. The module also explores common patterns for organizing business logic and discusses the four layers of the architecture: presentational, application, domain, and infrastructure. The term "layer" should be used instead of "tier" to indicate a physical container for code, such as a process space or machine.

DDD is a separation of concerns principle that involves building a model for the domain and then creating a layered architecture with standard segments, presentation, application, domain, and infrastructure. The presentation layer is responsible for providing a user interface to accomplish any task, and is evolving from a bottom-up approach towards a more effective top-down approach. It provides the user interface and an effective, smooth, and pleasant user experience.

The business logic concern is resolved in an appropriate design pattern, while the persistence concern is resolved by choosing an appropriate medium and format for storing data. The presentation layer is important for displaying real-world processes and must be device-sensitive, user-friendly, and faithful to the application's implementation. The application layer separates the presentation and domain layers, enabling clear design and orchestration of use cases. The application layer must report to the presentation and serve ready-to-use data in the required format. It is doubly linked with the presentation and may need to be extended or rewritten when a new front-end is added. In ASP. NET MVC, the application layer is made of classes invoked from controllers, who are part of the presentation layer but running server side.

Domain logic is the part of a software system that is invariant to use-cases and made up of entities to hold data and workflows to orchestrate behavior. Domain logic is the orchestration of all tasks required by implemented use-cases, and it is used to bake business rules into the actual code. There are three common patterns for organizing the business logic: transaction script, table module, and domain model. DDD is a decision between object-oriented, functional, or procedural design. The transaction script pattern is the simplest and most procedural approach to business logic. It leads to a design in which actionable UI elements invoke application layers end points and run a transaction script for each task. The table module pattern heralds a more database-centric way of organizing the business logic. The term domain model is often used in DDD to refer to having a software model for the domain. DDD is a pattern used in domain-driven design to focus on the expected behavior of the system and data flows. It involves creating an object model that fully represents the behavior and processes of the business domain. An aggregate model is used to refer to the core object of a domain model, and classes in the domain model should be agnostic to persistence. In the domain layer, an architect places all the logic that is invariant to use-cases, and there are two possible flavors of the domain model: an object-oriented model and a functional model. DDD conventions require classes to follow, such as factories with value types, and anemic domain models with only data and no behavior. Domain services complement the domain model and contain pieces of domain logic that don't fit into other entities. Infrastructure layer connects buildings to external sources of power and water. DDD is a type of database-defined deep web application. It is composed of several fundamental facilities, such as databases, persistence, security, logging, tracing, caching, and networking. To keep the application decoupled from specific products, facades can be introduced to hide technology details. In the next module, we'll explore an architecture using the domain model pattern and domain event support.

The "Domain Model" Supporting Architecture

Domain services are an important component of a domain model architecture, which is composed of two main components: the domain model and domain services. A domain model is an object-oriented model that fully represents the business domain, while domain services are responsible for things like cross-object communication, data access via repositories, and the use of external services. Misconceptions about domain-driven design include the object model being agnostic of data storage and the ubiquitous language being a guide to naming all the classes in the object model. Domain services are a type of layered architecture used to organize and express the business logic. They consist of a model and domain services, which are made up of classes that represent entities and values, as well as special components such as repositories that provide access to storage and proxies towards external web services. To make domain-driven design a success, it is important to understand how the business domain works and render it with software, and to ensure that no wrong call to the API is possible that can break the integrity of the domain. Value objects and entities are similar in NET classes, but represent different concepts and implementation details. In domain-driven design, attributes define a value object, which is fully identified by the collection of attributes. Entities may be grouped together to form aggregates. To indicate a value object in NET, a value type is used, which is a collection of individual values that is immutable once the instance has been created. Value types are used instead of primitive types to more precisely and accurately render values and quantities in the business domain. Domain services manage the infrastructure layer, while domain logic goes in the domain layer. Entities are classes with properties and methods, and domain logic is responsible for behavior. An aggregate is a collection of logically related entities and value types, with a root object as the public endpoint. Consistency is important for object behavior in a domain model. Depending on the business processes, boundaries may be drawn between aggregates with direct communication only allowed between roots. The connection between order detail and product is not allowed directly, but communication is allowed through the order proxy. Database-centric domain models can have different features depending on their focus, such as persistence or business domain. A persistence model maintains data and uses objects to persist it, while a side-by-side domain model prioritizes behavior and business processes. An example of a persistence model is shown below. This document explains how to use a domain to represent a sport game. The Match class is used, with an Id, string properties for teams, an enum MatchState, and object Score for possible scores. MatchState is an enum with three or more values, while Score is a distinct class with individual values for the score. While the Match class is effective for persistence, it does not describe the behavior or business context in which it can be used. The Score class is useful for creating a live scoring application, but it doesn't consider business rules or processes. TotalGoals1 is an integer property with a public getter and setter method, allowing any user or consumer to assign any value to it. However, assigning a negative value to TotalGoals1 may not be consistent with real business needs. Behavior refers to methods that validate the state of an object and invoke business actions. In a domain model with public classes, additional components are needed to access classes in a way that is consistent with business rules. The best way to model the real world is through an API that is part of the object, rather than an external layer of code. The Match class can be rewritten to include properties such as ID, Team1, State, and CurrentScore, as well as features like IsBallInPlay and CurrentPeriod. All of these properties are either private or internal, meaning there is no way for external users to directly set or change any aspect of the internal state of the Match class. However, the most important part of this is how to enable consumers of the class to alter the state. The Match class is used in a scoring application to dynamically create a score card as the match progresses. The behavior section of the Match class provides various public methods for describing the events during the match. The Start method sets the match state to MatchState and the Finish method changes it to finished. The Goal method creates a new instance of the Score value type and assigns it to the CurrentScore property. The Score class is a real value type that only allows for the number of goals for both teams and has no way for external callers to change it directly. The score is a constant type and only the match can set the CurrentScore property with this new instance. The Equals method has an override for the GetHashCode methods to recognize two score objects as the same if they have the same number and value. Unit tests can be done using a Match class and a sequence of actions close to the flowchart from domain experts. Aggregates are useful in simplifying the implementation of the domain model and ensuring that contained objects are always valid according to business rules consistency. An aggregate is a collection of objects that work together to form a larger entity, but there are no mathematical rules for identifying which objects form an aggregate. The boundaries of aggregates are determined by business rules and can vary between different domains. An aggregate root object is the root of the cluster of associated objects and has global visibility throughout the domain model. It has responsibilities such as ensuring encapsulated objects are always in a consistent state, taking care of persistence, and cascading updates and deletions through the graph. An aggregate root has its own dedicated repository service that implements consistency. To create an aggregate class, start with a plain class and upgrade some of the entity classes to become an aggregate. However, in a large domain model, it's useful to identify some classes as aggregates with a marker interface like IAggregate. For individuals and organizations, consider using the party pattern to define properties common to both individuals and organizations. Initialize classes with constructors or factories for more expressiveness. Domain services are classes whose methods implement the domain logic that doesn't belong to a particular aggregate and most likely span over multiple aggregates. They coordinate the activity of the various aggregates and repositories with the purpose of implementing all business actions, and they are not arbitrarily pieces of code created because you think you need them. For example, if you want to determine whether a given customer has reached the status of a gold customer after exceeding a given threshold of orders on a selected range of products, you need to query orders and products to collect data necessary to evaluate the status. The most important details in this text are the two options for booking a meeting room: using a booking domain service or having a booking aggregate. In domain-driven design, a repository is just the class that handles persistence on behalf of entities and ideally aggregate roots, and it is the place where connection strings and SQL commands are dealt with. Events in the context of the domain layer are becoming increasingly popular. They are optional but can be used to express the intricacy of some real-world business domains. An example is an online store application where an order is successfully processed and a special task needs to be performed. The first option is concatenating code to implement additional tasks to the domain service method, but this is not expressive and can make the service code convoluted. Additionally, events may violate the ubiquitous language when observed. traction as they provide benefits such as reducing the need to handle code in one place, allowing for multiple handlers to deal with the same event independently, and utilizing a plain class with a marker interface. Events can be dispatched using a bus, which is an external component typically part of the infrastructure. Events offer greater flexibility in code as they can record and log what happened and add/remove handlers easily.

The CQRS Supporting Architecture

CQRS is an acronym for command query responsibility segregation, which is a new way of expressing the business domain through an object-oriented model. It is based on separating commands from queries using distinct application stacks. The module discusses the foundation of CQRS and identifies three flavors of it: regular CQRS, premium CQRS, and CQRS deluxe. The module also presents two ways to design a class for a sport match, one that incorporates business rules for the internal state and exposed behavior, and one that is a mere data transfer object trivial to persist, but devoid of business logic and with a public read write interface.

CQRS is a method-driven approach to system architecture that separates commands and queries. It involves defining a behavior rich class for use cases where the state of the system is being altered and a query for use cases where the state of the system is being reported. The behavior rich class requires fixes to fully support persistence via an O/RM, while the other class has no business rules implemented inside, leading to inconsistent instances.

In terms of layers and system architecture, CQRS moves from a classic multilayer architecture with presentation, application, domain, and infrastructure, to a slightly different layout with presentation and infrastructure unchanged.

CQRS is a concrete implementation pattern that allows for parallel and independent development, distinct optimization of the same stacks, and scalability potential. It works well with plain old CRUD applications, and to make an existing CRUD system CQRS aware, the application architecture changes slightly.

CQRS is a command and data modeling approach that allows for simple and scalable architecture with no need to sacrifice existing code. It can be used in conjunction with other patterns, such as table modules or fully procedural transaction script patterns. In the read stack, O/RM of choice, ADO. NET, ODBC, Micro Frameworks, stored procedures, and LINQ can be used to bring data back. A read-only wrapper for the Entity Framework DbContext can be used in the read stack. CQRS is a JavaScript-based method to handle requests in a plain simple CRUD solution. It involves using a database class instead of the native Entity Framework DbContext and returning generic DbSet objects as high queryable. The key thing in ASP. NET MVC is that when a request arrives, it is captured by a controller and then processed. The dependencies diagram for a website designed with CQRS is shown in Visual Studio, and everything is orchestrated from the application layer which in turn responds to requests intercepted by controllers at the presentation level. CQRS CRUD ASP. NET MVC website implements the Post-Redirect-Get pattern to ensure that the last operation tracked by the browser is always a get. CQRS is a command-oriented approach that uses ad-hoc storage to optimize the workflow and use and persist data as it comes natural. CQRS is a relational database system that uses O/RM and LINQ for data access. It requires distinct data stores for commands and queries, which can be synced up when a command executes. Synchronization can be done synchronously or asynchronously, depending on what's best for the application. If stale data is acceptable, synchronization can be scheduled as a periodical job. CQRS is a collection of principles that include synchronization, async, scheduled, and on-demand synchronization. The async approach keeps data up-to-date automatically, while the scheduled approach assumes the application can work with stale data for a few seconds or hours. CQRS Premium is a more sophisticated use of the principles, with a server central model, command and read stacks, and a shared helper library. It is implemented in the context of a layered system, with the user interface placing a request and the application layer sending the request to the command stack. CQRS is a command-oriented database system that uses a data store to store and retrieve actions from a given entity. The data store can be either relational or NoSQL, and the sync function involves reading the list of recorded actions for a given aggregate entity to extract information for the UI listeners. The premium implementation of CQRS involves a Controller class that can handle post-requests and a MatchService class that represents the application layer. The ProcessAction method on the matchService class logs events using the helper class EventSourceManager. The difference between events is the number and values of parameters passed. The EventSourceManager class has overloads for the Log method. Data is assembled into a matchEvent that can persist in the EventRepository. To sync up with the read model, a domain model class is used to represent a match instance. A helper class, MatchSynchronizer, extracts information from a snapshot of the state to create an ad-hoc snapshot for the presentation layer. Tasks are workflows that are concatenated sets of commands and events.

CQRS is a command-based architecture that simplifies the management of complex business workflows. It involves defining classes for commands, queries and events, using standard naming conventions. The application layer sends messages and the command layer processes them, generating tasks which can be long-running states or stateless processes. Commands usually don't return data back to the application layer, but can trigger commands following user actions, incoming data from asynchronous streams, or other events generated by previous commands.

CQRS is a message-based system that uses a bus and listeners as the main building blocks. The workflow is the direct descendent of user defined flowcharts and advances through messages, commands, and events. CQRS Deluxe is a flavor of command query separation that relies on a message-based implementation of the business tasks.

The application layer turns any input into a command and pushes it to the bus, which is generically referring to a shared communication channel.

There are two types of message handlers called sagas and handlers, and the flowchart behind the business task is never laid out entirely. CQRS Deluxe is a command-line system that uses a bus to orchestrate the steps of a business task. It has an innovative architecture of the command stack, which allows for event sourcing and stale data synchronization. The implementation of CQRS Deluxe requires a reacher and more sophisticated infrastructure to support listeners and handlers of business logic commands and events. The bus is a class that maintains a list of known saga types, running saga instances, and known handlers. It dispatches messages to sagas and handlers based on their declared interests. A saga is characterized by a command or event that starts the process and a list of commands and events that saga can handle. Each saga must be uniquely identified by an ID, which can be a GUID or the ID of the aggregate saga. A saga may be stateful or stateless. In the end, a saga is what is needed. CQRS Deluxe is a message-based approach to extending existing solutions when new business needs arise. It involves implementing a bus and more sophisticated infrastructure for the business logic, allowing developers to easily add new sagas or handlers without touching existing workflows or code.

Event Sourcing

CQRS stands for Control Queue Routing System and is a method used to efficiently route messages between the presentation and business layers of a system. In module five of Pluralsight's course on Modern Software Architecture, the focus was on using events to implement business workflows triggered by commands. Events were just cross handler notifications of things that just happened, and this was the first step of a longer evolution aimed at transitioning software architecture from the idea of models to persist to the idea of events to log. The use of events in software architecture is becoming more prevalent, with customers demanding the ability to know the state of a system at a given date for statistical analysis and business intelligence. Events are not only for complex applications, but can also be used for common applications. An example of this is invoices and their sales taxes, where extra VAT tax must be injected into the invoice itself, but what happens when the VAT varies? This raises the question of how to organize tables in a database that needs to update as taxes laws change. Events provide a more powerful and efficient approach to data warehousing. Event sourcing is a design approach based on the assumption that all changes made to the application state during the entire lifetime of the application are stored as a sequence of events. This means that serialized events are the data building blocks of the application, and the current state of domain entities is used as the starting point to process business transactions. An example of this is a shopping cart, where the model is a faithful representation of the internal logic associated with it. Event sourcing is the process of describing the state of a shopping cart through the sequence of events that brought it to a given list of items. This data is then used to rebuild at any time the state of the shopping cart. This event-based representation of an entity is functionally equivalent to structural and event-based representations of entities, but for most applications, events are ignored. To store events, one can use an ad-hoc relational table, a NoSQL store, or even with a specific event store product. An event store is append-only and does not support deletions. CQRS is a data source that looks beyond state and looks in the direction of events. Events express the state of a domain entity and can be replicated and replicated, but they are immutable once stored. CQRS is not just for a few types of apps, but is the dawn of a new software design world. Events are not a revolutionary new approach to software design, as relational databases do not manage current state internally. Instead, events are tokens stored in application data sources and provide a more general storage solution for many systems. There are two options for storing events: primarily saving the current state or storing relevant application facts through a separate log. When users add products to a shopping cart, events are recorded and used to represent the current state of the shopping cart or order. Businesses should consider tracking when items are removed from the cart to determine if events are necessary. CQRS is a programming language used to store and retrieve events in a stream. It involves command and query responsibility segregation, and uses event streams and structured data to work together. Persistence in software is made up of four operations: CREATE, UPDATE, and DELETE, and QUERIES to read the state without altering. The key principle of CQRS is that asking a question should not change the answer, and events are immutable and self-contained. To track events in a database, each event should be identified with a unique code and associated timestamp. Full details of the record should also be saved, including order details. Event storage can be transparent using relational, document-based or graph-based technology. The CREATE and UPDATE operations are similar but differ in that they require unique event IDs, timestamps, and changes applied. DELETE operations are similar but require storing the full state of the entity along with the specific event information. Event sourcing is a data storage technique that allows users to store and retrieve events from stored events. It involves deleting specific information, which can be done logically or through UNDO functionality. Data projections from stored events are also important for understanding the state of the system. Replay of events is a two-step operation that involves retrieving all events stored for a given aggregate and copying the information to a fresh instance. Queries can be used to return the full or partial stream of events. Event sourcing is a process of collecting data from a stream of events, such as a RavenDB or relational database. Key pieces of information include EventId, Timestamp, AggregateId, and event-specific data. The actual rebuilding of the state involves going through all events, grabping information, and then altering the state of a fresh new instance of the aggregate. Replay is not about repeating commands for generated events, but instead, looking into this data and performing logic to extract information. Events are data rendered at a lower abstraction level than plain state, which can be used for business intelligence, statistical analysis, and simulation. Event sourcing is the process of collecting and reusing continuous data from events to enhance the scalability of an application. However, there are performance concerns when processing too many events for rebuilding the desired projection of data. An effective workaround is to set a snapshot of the aggregate state at a given point in time and save that as a value. This is done by keeping track of the snapshot point and replaying events for an aggregate from the latest snapshot to the point of interest. An example of this is a web application representing a booking system, where a booking is made for 1 hour and the length of the booking is for just 1 hour. Event sourcing and CQRS are two important concepts that can be used to track the history of a resource, such as a booking. CQRS allows for the separation of command and read stacks with event sourcing, which allows users to track the whole history of a given resource. In Visual Studio, the code for editing the booking is shown in the screenshot below. An Ajax call is made to an entry point behind an API controller in the MVC application, which calls into a History method on the BookingService class. The method, BookingHistory, takes the ID of the booking and returns the history of the booking for the specified AggregateId. This is done by querying all events in store for the particular AggregateId and deserializing all the information about the particular event. Once the history has been obtained, it is turned into a JavaScriptSlotHistory for the purpose of making the UI nicer for uses. This involves normalizing data for JavaScript rendering and storing empty fields and values for elements of the booking record that have not been changed with the update. Event sourcing is a powerful tool that allows users to quickly and easily manage and store large amounts of data from various sources. Event-based Data Stores are becoming increasingly popular as they provide a more structured way to store and manage event data. Examples of event-based data stores include geteventstore.com, which offers an API for plain HTTP and.NET, as well as streamed events. CQRS is a data warehouse that allows users to write, read and subscribe to events. It has three types of subscriptions: volatile, catch-up and persistent. Catch-up subscriptions are good for denormalizers, which play a key role in the query stack.

Designing Software Driven by the Domain

Domain-driven design (DDD) is an attempt to systematize software design and development. It involves building software that mirrors the business domain and focusing on tasks to model the system. Legacy code can be complex and undocumented, making it difficult to refactor or rewrite. Legacy code can be incorporated into new applications by exposing it as a service or rewriting it. Evaluate the costs of not rewriting from scratch and integrating legacy code as a service. Revisiting CRUD Systems focuses on four fundamental operations. CRUD systems are database-centric, but must be devised in a different way to be realistic and effective. They have basic business logic and are used by one or few users and deal with elemental collections of data. The CQRS approach is essential for modern software and systems, as it takes into account business logic, concurrency issues, and interconnected entities. UX-driven design is the experience that users go through when they interact with a given application. The bottom-up approach to software architecture is still the most effective, but now users are actively dictating user interface and user experience preferences. UX-driven design involves building UI forms and screens as users approve, then implementing workflows and binding workflows to presentation endpoints. UX-driven design involves iterating on the UI forms approval process until users agree. Sketches, wireframes, mockups, and prototypes are used to show live how the application will work. A UX Architect works side-by-side with the solution architect role. The UX Architect is responsible for defining information architecture, interaction, visual design, usability reviews, and using tools such as Balsamiq, UXPin, Infragistics Indigo, and JustInMind. Layers over tiers, top-down design, CQRS command and query responsibilities, segregation, and event sourcing are key to modern software architecture, making it easier to code and optimize.