Introducing Domain-Driven Design

Domain-Driven Design (DDD) is a great approach to building software that has been around for a long time. It is commonly referred to as DDD and has its own Twitter hashtag, dddesign. This course will focus on the developer perspective and the technical and coding aspects of DDD, such as modeling problems, technical components, and managing complex projects. Domain-Driven Design (DDD) is a fundamentals course that focuses on the problems of the business domain that you're attempting to solve. It is a critical shift from decades of focusing on how to store data and then letting that drive how the software is designed. Steve and I have both been designing and developing software for a long time, and it aligns very naturally with ideas that we've come to from our own experience. Eric Evans wrote his book to understand what was behind the successes he had achieved with large-scale, complex software projects and what the patterns were. Domain-driven design (DDD) is a principles and patterns approach to solving complex problems. It provides a clean representation of the problem in code that can be easily understood and verified through tests. DDD places emphasis on understanding the client's needs and working with them as full partners through a project. The ultimate goal of DDD is to solve problems, not build software. To gain a high-level understanding of DDD, it is helpful to look at it from a very high level and encourage better interaction with domain experts. DDD is a methodical approach to software development that involves dividing a problem into separate subdomains to tackle it independently. It requires communication with business experts and focusing on a single subdomain at a time to make the problem easier to solve. The principle of separation of concerns is critical in identifying and implementing each subdomain, and DDD helps maintain consistency in business logic by focusing on the domain.

Modeling Problems in Software

Bounded contexts are important to separate the core domain model from related subdomains in DDD. Ubnified language can have a large impact on model, design, and application. Steve Smith and Julie Lerman want to use DDD in their veterinary clinic management domain. Julie, a veterinarian and software development expert, has agreed to play the role of a domain expert. The system records information about the patient, observations, notes, and diagnoses. After an appointment, clients pay, and billing is done after the fact, as well as tracking purchases, prescriptions, toys, and lab work. A content management system (CMS) is also used to view information about their pets. Bounded contexts are used to identify the bounded context of a model, which is where it is valid. It is important to understand that appointments can be either an office visit or a surgery, and that the clinic does not schedule any of the patients themselves. Patients and clients are not the same thing to a veterinarian, so it is important to get on the same page with the customer early on in the process. Bounded contexts are used to identify the bounded context of a model, which is where it is valid. Bounded contexts are used to avoid inconsistent behavior in the system and to untangle shared concepts. DDD and bounded contexts are two important concepts in Debugging and Refactoring, where a subdomain is a view on the problem space, while a bounded context represents the solution space. Context maps should be used to visualize and demonstrate the boundaries between different teams, and it is important for separate teams responsible for different contexts to understand which aspects of the application they can change on their own and which are shared dependencies. Bounded contexts refer to two sides of an application, one being the notification and the other being the payment. To ensure integration, it is important to designate shared concepts or resources as a shared kernel, separate databases for each bounded context, and define context boundaries when creating an API. DDD is a project management technique that involves breaking down complex tasks into smaller, more manageable components, and it is often necessary to have a clear context boundary in order to effectively apply DDD. Bounded contexts are important for successful domain-driven software development (DDD). In our application, the main area of focus is the appointment scheduling bounded context. Code is isolated into its own package referred to as a shared kernel, and a bounded context does not always mean a separate application. Effective communication among stakeholders is essential, and a single, shared language is needed for this. Online translation tools can cause confusion when translating between domain experts and programmers' terms. A shared common language that incorporates and uses terms defined in the domain model is needed for effective communication.

Elements of a Domain Model

DDD is a term used to describe domain models, which are driven by behaviors, not classes and properties. It is important to stay focused on the domain and understand why it's important to stay there. This module will focus on the technical aspects involved when modeling a bounded context, such as entities and aggregates. Entities are the key types in a system, but not every type is an entity. Understanding DDD terms is essential to make it easier to talk about the process and to differentiate them from simpler entities. DDD stands for Domain-Driven Design and involves entities, context, and value objects. It is important to understand these elements to effectively model domains and implement the model. Focusing on the domain is key, as the user interaction with the app should be considered in this regard. Focus on the domain, not just the technical details, to avoid complications and distractions when designing business software. The domain layer represents concepts, information, and rules, while the infrastructure delegates technical details. When modeling a domain, behaviors should be considered instead of just changing object states. Identifying events through event storming can lead to understanding behaviors, as seen in Alberto Brandolini's method for past-tense brainstorming with clients. DDD is a methodology used to model a system based on events. Event Modeling is another interesting methodology for modeling a system based on events. Anemic and Rich Domain Models are two commonly-used terms in domain-driven design. Anemic domain models are focused on the state of its objects, which is the antithesis of DDD. Martin Fowler writes about anemic domain models, which look like the real thing at first, but have very little behavior. DDD is a type of database modeling that uses service objects to capture all the domain logic. It is important to strive for rich domain models rather than anemic domain models, as they can be anti-patterns and contrary to object-oriented design. Entities are objects defined by an identity and their values, which can be tracked, located, retrieved, and stored with an identity key. Domain-Driven Design (DDD) is a pattern used to ensure that objects have the technical attributes of Domain-Driven Design entities. The most important entity in our model is Appointment, which inherits from a base class called Entity. However, we decided to have a separate utility for managing client and patient information and to manage information about staff and staff scheduling. We also differentiated CRUD from complex problems that benefit from DDD, as the client, patient, doctor, and room classes are completely different from CRUD classes. All we need to know about these objects when we're scheduling is their IDs, names, and some other details. The user interface of the software can be designed to make moving from one context to another seamless, even if it is in separate bounded contexts. This is demonstrated when maintaining client and patient data, which is a separate task, but Michelle wanted to ensure that anyone working at the front desk can easily move between these tasks in the software without disrupting their workflow. The design of the application ensures that everything is bound within its own individual context, making it easy for end-users to access and edit appointments without having to open a different application. DDD stands for Decentralized Decision Making and is a concept used in Computer Science. It involves using a unique identity to persist and retrieve an appointment even if some of its values change. This is done by creating a stripped down read-only type of entity, such as client, patient, doctor, and room, that only has minimal amount of detail for each. We also discussed how to name the types that are reference types in this context, and Vaughn Vernon, a DDD expert, triple checked our decision. All these types inherit from our base entity class, but they use ints for their base entity's ID instead of the GUID used by appointment. DDD is a style of application that uses database-generated ints for entity management. It is easier to use GUIDs when building DDD entities and their related logic than relying on the database. Integer IDs can also be used in DDD applications, but are not always necessary. To bridge the conflict between using GUIDs and database-generated ints, both GUIDs and database-generated ints can be used in an entity. Eric Evans discussed the single responsibility principle for entities and how it applies to their lifecycle. Entities are central in a system, but as they become more complex, there are more conflicting demands for them. The main responsibility of an entity is its identity and lifecycle. Implementing Entities in Code involves using the Appointment class in FrontDesk's veterinary appointment scheduling application. The Appointment class inherits from BaseEntityT, which is a generic base class, and BaseEntityGuid, which defines the type of identity property used. The BaseEntity class uses a GUID as its identity to allow for easy creation of new appointments without waiting for the database to generate the ID. The class also has a property to hold a list of domain events that define explicitly for each type that inherits from the base entity. To create an appointment in a valid state, the minimum necessary elements are passed in through methods rather than setters. Modifications to appointments can be done through methods and guards to ensure a valid value is passed. An appointment UpdatedEvent is raised to handle and send a notification or perform other actions, providing flexibility in the future to change logic. This document explains how to maintain consistency in the domain model by using a method to explicitly update a room and using guard clauses to prevent invalidation by using non-existent foreign key relationships between appointments and rooms. The appointment system then records each appointment as part of the clinic schedule and uses a method called Schedule to query the database for other available slots and save the appointment accordingly. The next module will explore this design further and revise it slightly. Finally, the Doctor class is also considered a simple entity necessary for this bounded context design. The Doctor type is a minimal implementation of the Doctor type that satisfies the scheduling bounded context. It inherits from BaseEntity but uses an int for its key and only has a string Name property. The other reference types, such as Patient, Patient, and Room, are organized into a folder called SyncedAggregates. The Clinic Management app is responsible for updating the Front Desk bounded context when changes are made. Eventually, the different bounded contexts are updated to the new state when a change is made through message queues. This module covers some common terms related to anemic domain models and rich domain models. DDD is a type of database-driven design that focuses on behavior and entities. Entities are the classes in the domain model that are tracked by an identifier and can be used to build graphs and persist data. Anemic models are used for CRUD, while rich models are used for complex problems. In this module, we learned how to identify the differences between entities and reference data. We also looked at the appointment class in our scheduling app and reference entities. In the next module, we'll focus on domain models, value objects, and domain services. Resources for further reading are provided.

Understanding Value Objects & Services in the Model

This module introduces value objects and domain services. Value objects are immutable objects that measure, quantify, or describe something in a domain. They have specific characteristics and cannot be changed once created. Methods on value objects should only compute things and not change the state of the object. In DDD, both entities and value objects are typically defined as classes. Value objects, such as strings and monetary values, have advantages over structs in terms of encapsulation and inheritance-based extension and reuse. Strings are immutable and the order of characters is critical for their meaning..NET allows for modifications to strings, such as changing the length or upper case. Monetary values in financial systems are a great example of value objects, as they provide a specific measurement for a company's worth. The point in time of a financial system like 50 million dollars needs to be taken into account when creating a value object. While a decimal property representing the worth amount and a string property representing the worth unit are related, they are not tied together in any way. To prevent this, a separate value object can be introduced to represent the entire worth concept. Value objects can also be used for other types of descriptions like dates, which have private setters and are immutable. The full logic of the class will be demonstrated in the application. ValueObjects should be used instead of entities in domain-driven design. Vaughn Vernon recommends that entity designs should be biased towards serving as value containers, with properties being themselves value objects. Identity values can also be treated as value objects, as they can be stored as ints or GUIDs. An example of this is a Client class inheriting from a base entity, specifying the type as ClientIdValueObject and then using a CreateAppointmentFor method to take a clientId and a patientId. Value Objects are specialized objects that can be used to tightly constrain parameters in order to avoid mistakes when passing them in the wrong order. Eric Evans believes that value objects are a great place to put methods and logic, as they can do reasoning without side effects and complexity. Date libraries are also a good example of a value object, as they perform common functions on dates without having to code them ourselves. Implementing Value Objects in Code involves scheduling appointments, and DateTimeRange and DateTimeOffsetRange are implemented in the shared kernel package. ValueObject is a type of property that is read only and should not be changed once it's been created. It should get all of its state through its constructor, and any invariants that need to be checked should happen in a constructor. The DateTimeRange type does have some additional methods that let us create new DateTimeRange instances from existing ones. Finally, the base ValueObject class requires overriding a GetEqualityComponents method. Custom logic needed to determine whether one appointment overlaps with another is another area where the ValueObject can help. ValueObjects are a way to organize and store data in a structured way. They are more reusable and reduce the complexity of other domain types. Eric suggests moving logic from entities into value objects, which allows for more scalable and easily testable data. AnimalType is an example of a value object that encapsulates related properties in a single value object. When using services, it is important to find a natural home for them in an existing entity or value object to avoid an anemic model. Good domain services should not be a natural part of an existing entity or value object, have a defined interface made up of domain model elements, and be stateless. These rules apply specifically to domain services that belong in the core of an application. The UI layer represents the front end of the system, while the application layer is concerned with behavior necessary for the application, but unrelated to the customer's problem domain. In the core of the application, domain services define operations on multiple domain elements or may orchestrate workflows. Infrastructure-level services implement interfaces defined in the core of the domain. This module focused on value objects, which are used in a domain model to measure quantify or describe something in the domain. It was important to review the terms immutability, value object, domain services, and side effects. Value objects are immutable classes defined by the sum of their properties, and they don't need an identity outside of their individual properties. Domain services provide a place for logic and behavior that can't be found in entities and value objects in a domain. Side effects are changes that occur in a system or interaction with the outside world, and it's important to keep operations that query information separate from those that change state. Value objects should be compared using only their values and should be immutable. Examples of value objects include the.NET Framework string type and custom value objects. Domain services are used to orchestrate operations between different parts of the domain model, but should not be used if an entity or value object does not exist. The next module will focus on building aggregates from entities and value objects while respecting their relationships.

Tackling Complexity with Aggregates

Aggregates and aggregate roots are important concepts in Data driven Design (DDD). In DDD, relationships between entities are referred to as associations. To manage data complexity, it is important to use techniques such as aggregates and aggregate roots. Additionally, limiting bidirectional relationships and separating dependencies between entities can help reduce the complexity of the system. Aggregates are units of data changes that need to be treated as a unit for data changes. They must have an aggregate root, which is the parent object of all members of the aggregate. Aggregates can have rules that enforce data consistency across multiple objects, such as a collection of components needing a specific set of components to be in a valid state. Data changes to the aggregate should follow ACID, and the aggregate root should maintain its invariants. Aggregates are a cluster of associated objects that are treated as a unit for the purpose of data changes. When considering whether an object should be considered an aggregate root, it should be considered if deleting it should cascade or if it makes sense to have just this object detached from its parent. DDD brings important concepts when considering relationships among entities, such as bidirectional relationships which can make things overly complex. DDD is a domain-driven design guide that defaults to one way, unidirectional relationships. Relationships should be part of a type's definition, and should be defined using properties that allow us to traverse from one end of the relationship to the other. Bidirectional relationships should only be introduced when neither object can be defined without the other, and should be specific about the direction of the relationship to keep the model design simple. With a DDDI, we can look at our model and ask, Can we define a client without identifying their pets? Aggregates serve as boundaries between logical groupings within an application, and are enforced by prohibiting direct references to objects within an aggregate that aren't the root of the aggregate. Relationships that traverse from appointment to doctor, patient, and client, and to client to patients or their pets, but not the other way around. Bidirectional relationships can cause serialization to fail, so it's best to define single-direction relationships instead. Aggregates and aggregate roots only apply to objects, not data. References to customers should be made through object references, not direct navigation properties. When dealing with appointment scheduling, a reference to the client from the patient is important. The most important details of the phrase "aggregate" are that it is a list of resources tied to a particular timespan, it eliminates most of the object relationships from the appointment classes designed, and it includes the IDs of the related concepts rather than object references. Invariants are used to better understand the aggregate, such as not being double booked, and invariants are used to verify any invariance the aggregate may have. Invariants are constants or requirements that must be true for a system to be consistent or valid. Examples of invariants include the speed of light, total items on a purchase order, and appointment overlap. In the example above, the purchase order would be the aggregate root, while the individual line items would be modeled as separate objects. However, it was found that appointments did not make sense as an aggregate root, as the schedule knew about such things. This shift in thinking could lead to a more efficient model design. Refactoring is an important part of domain-driven design, and Eric Evans talks about it in his book. Steve and I recognized signs of a misidentified aggregate when we initially had the appointment as the central focus of the design. The original structure had appointment in its own folder and marked with the IAggregateRoot interface, but this did not work as well as the later version. To enforce the invariant that appointments whose times overlap for the same pet should be marked as potentially conflicting, we used a repository to get those other appointments for the same date as this one. DDD is a method that enforces cross-aggregate invariants between entities. In this case, the schedule aggregate was introduced as an explicit object in the model to ensure that appointments do not overlap one another and that the responsibility for ensuring that appointments are not double booked and similar invariants can be performed by the schedule. This makes the design much simpler and ensures that appointments don't overlap one another when saving changes to a schedule or deleting an entire schedule. The Schedule Aggregate is a class that inherits from the BaseEntity type and uses a GUID for its id key. It is responsible for ensuring that the incoming values are valid and that the schedule has only a read-only IEnumerable of appointments. The ScheduleAggregate is a collection of data used to store and retrieve appointments from the database. It has a date range that can vary with any given instantiation, and its persistent details are configured in the infrastructure project's Data Config folder. The first method for adding new appointments forces all new appointments through this method, and the schedule is responsible for marking any conflicts that might be marked. After deleting an appointment, the schedule needs to once more mark any appointments that might be conflicting. There are also two TODO exercises included in the sample. The MarkConflictingAppointments method is responsible for detecting and marking appointments that might conflict. It is encapsulated in the schedule aggregate and checks whether the patient has two appointments that overlap. The last method on schedule provides a hook for its appointments to use to notify it when changes are made to one of them. This handler simply calls MarkConflictingAppointments, but is exposed as its own separate method. To see how it's used, the appointment class's UpdateStartTime method is called when the application needs to update the start time for an appointment. Aggregates are used to reduce complexity and can be used with foreign key values as a reference. An aggregate is a group of related objects that work together in a transaction, with the root becoming the entry point and responsible for ensuring all rules are met. Aggregates are a set of related objects that live in a single transaction to make sure the system is consistent. One-way relationships should be used as a default, and bidirectional navigation should only be introduced when necessary. It is important to update your model regularly as you learn more about the domain. Repitories are a critical pattern in domain-driven design.

Evolving the Application Easily Thanks to DDD

DDD stands for Domain-Driven Design and is a design practice that involves using logic to control and maintain a system. In this module, we'll review our current system design and how it incorporates DDD patterns and practices. We'll also learn about a new feature that leverages message queues. The main benefit of our design choices is the ease with which the system can be extended and maintained in the future. The system is currently two different web applications, with the main focus being the appointment management application. It uses value objects and domain events to model concepts in the domain. The design of the system is very clean and reflects the customer's domain. The customer has a request for the new scheduling application to address the issue of customers forgetting their appointments. The current system does not allow for automatic reminders, but the customer could send an email to remind them of their appointment when it is scheduled or on the day before it is scheduled. The proposed solution would be to send an email to the client when they schedule the appointment, but the existing model already handles certain events such as when appointments are scheduled and marked as confirmed. DDD is a distributed dynamic scheduling (DDS) architecture that allows for easy communication between clients and appointments. The implementation steps involve sending confirmation emails to clients, allowing them to click a link to confirm their appointment, and marking the appointment as confirmed. The system also uses existing and new domain events, application events, event handlers, services, and messaging queues to communicate between separate applications. Message queues are used between applications to decouple them and allow one application to drop off something into a queue while the other continues on with its work. There are various implementations of message queues available online, including free and built-in services in cloud services. In some cases, a service bus is needed to ensure that messages are delivered to different applications that care about the message, even if they didn't exist at the time. The service bus allows for decoupling and hooking up other applications to listen to the queue. The most important details in this text are the use of RabbitMQ as a message queue, the AddNewAppointment method inside the schedule aggregate root, and the integration events that are published by MediatR. These events are formatted as JSON data before being inserted into the queue. This is done by sending a message to the queue, which is a separate bounded context used to publish and consume events. RabbitMQ has a user interface to inspect queues, which can be accessed from the Front Desk app. The VetClinicPublic app is tracking a message in the vetclinicpublic queue, with the payload being a JSON expression of event data. The Front Desk app publishes the message to the queue, and the VetClinicPublic app reads from it to confirm the appointment. The public website hosts the email and retrieves the information to create a confirmation email. The email contains a link to the public website, which includes the appointment ID. The website then sends an email containing the confirmation link. The demo solution involves using FrontDeskRabbitMqService to periodically check for messages in the message queue. A tool called Papercut is used to emulate a local email server for testing. When the user clicks on the CONFIRM button, they are directed to the website's GUID, and the website uses its own method called confirm to push the relevant appointment ID into another queue. This document explains how to use multiple message queues in a system to handle various communications, including relaying messages from the public website to the FrontDesk app when a client clicks on the CONFIRM link. The two applications, Fdvcp and VetClinicPublicRabbitMQService, communicate back and forth using their respective queues to retrieve and display confirmation information. A hosted service called VetClinicPublicRabbitMQService is also implemented to respond to incoming messages and confirm appointments. Appointment triggers domain events that update the user interface when an appointment is confirmed, resulting in a green bar on the top of the appointment. This is easy to implement because the website already listens for events. The original sample for this course used confirmation links from SignalR and WebSocket, but now people are receiving them every time they make an appointment. The code for the solution is available on GitHub, and it can be run using Docker or Visual Studio. To debug the app, run RabbitMQ and PaperCut using Docker commands. Add a new appointment and save it, triggering the ScheduleAggregate root's AddNewAppointment method. Use the RelayAppointmentScheduled service to create an event that gets pushed onto the public website's message queue. The event will be used to send an email to the client with all necessary data. The Publish method lives inside of RabbitMessagePublisher and is responsible for getting the message into a structure that RabbitMQ can use. Finally, inspect the message queue in RabbitMQ to verify that the message has been queued up for the VetClinicPublic input queue. The code for the VetClinicPublic application completes the actual thread of the UI. The next step involves watching the flow of the code after the hosted service starts up. The HandleMessage method is used to parse the message using JSON and deserialize it into an appropriate type. A mediator is used to send the command, and a separate handler is used to send the email. The reason for this is that the end user needs to be able to click a link that goes to a public location on the internet. Finally, clicking the link brings us back to the VetClinicPublic application's, AppointmentController class, which creates a new event called AppointmentConfirmLinkClickedIntegrationEvent. DDD is a method used to dynamically add new items to a queue in a web application. It involves sending an event using a RabbitMQ message publisher to a front desk input queue. The event, AppointmentConfirmLinkClickedIntegrationEvent, is then parsed and published by the HandleMessage method in the service. This event triggers a call to the EmailConfirmationHandler, which loads the schedule aggregate and locates the appropriate appointment. The appointment.confirm method makes an appointmentConfirmed domain event, which is fired once the aggregate is saved, and this event triggers a handler in the FrontDesk UI. Microservices have become popular and have some benefits, such as being self-contained and independently deployable. DDD is a concept that refers to domain-driven design (DDD). It involves creating a boundary around a microservice and focusing on specific behaviors within it. Microservices can be treated as bounded context with their own terminology and language. However, it is important to not assume that microservices and bounded context always align. DDD can be useful for microservices, as many of the problems they solve are also solved by DDD. In a sample application, the confirmation email sending logic could be moved into its own process and treated as a separate microservice. The benefit of carving out a microservice is that if it is stable and working, it can be left alone. This document explains how to extend and run a microservice using the provided resources. It emphasizes the importance of keeping the application running and up-to-date with changes to other parts of the app or system. The author provides tips for extending the sample application, including running it in Visual Studio, having a local SQL Server, and setting up RabbitMQ and Papercut test email servers. Additionally, the author suggests running the app with just two commands, dockercompose build parallel and dockercompose up, to see everything working. The README file shows the ports for various applications and utilities. The control used solved problems, but the UI impacted the domain design. It's important to consider the UI early in planning, as even sketching can affect the entire system. This approach can benefit the scheduler sample. DDD is a domain-driven design technique that considers the user experience, and can help visualize how processes communicate within a bounded structure.