Key Points Domain Driven Design

1. Domain experts interact with developers to help draw up a model for the domain in question. In place of domain expert we can have chatgpt to interact and refine the domain model.
2. We should draw model diagrams so that they capture the details and rules more clearly and its not hidden in code.
3. Domain Driven Design calls for designing a model that not only captures information but also lays the foundation of design i.e helps in implementation (there is some sort of binding between domain model and implementation)
4. An analysis must capture fundamental concepts from the domain in a comprehensible, expressive way. The design has to specify a set of components that can be constructed with the programming tools in use on the project.
5. MODEL-DRIVEN DESIGN discards the dichotomy of analysis model and design to search out a single model that serves both purposes. Setting aside purely technical issues, each object in the design plays a conceptual role described in the model.
6. There are always many ways of abstracting a domain, and there are always many designs that can solve an application problem. This is what makes it practical to bind the model and design. This binding mustn't come at the cost of a weakened analysis, fatally compromised by technical considerations. Nor can we accept clumsy designs, reflecting domain ideas but eschewing software design principles.
7. Sometimes there will be different models for different subsystems but only one model should apply to a particular part of the system, throughout all aspects of the development effort, from the code to requirements analysis.
8. Isolating the domain: Software programs involve design and code to carry out many different kinds of tasks. They accept user input, carry out business logic, access databases, communicate over networks, display information to users, and so on. So, the code involved in each program function can be substantial. When the domain-related code is diffused through such a large amount of other code, it becomes extremely difficult to see and to reason about. We need to decouple the domain objects from other functions of the system, so we can avoid confusing the domain concepts with other concepts related only to software technology or losing sight of the domain altogether in the mass of the system.
9. There are all sorts of ways a software system might be divided, but through experience and convention, the industry has converged on LAYERED ARCHITECTURES, and specifically a few fairly standard layers. Although there are many variations, most successful architectures use some version of these four conceptual layers:
    1. 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.

    2. Application Layer: Defines the jobs the software is 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.
    3. 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 delegated to the infrastructure. This layer is the heart of business software.
    4. 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.

10. Partition a complex program into layers. Develop a design within each layer that is cohesive and that depends only on the layers below. Follow standard architectural patterns to provide loose coupling to the layers above. Concentrate all the code related to the domain model in one layer and isolate it from the user interface, application, and infrastructure code. The domain objects, free of the responsibility of displaying themselves, storing themselves, managing application tasks, and so forth, can be focused on expressing the domain model. This allows a model to evolve to be rich enough and clear enough to capture essential business knowledge and put it to work.

Idea for our work: Step1 was to generate the domain model and use cases, we can get the domain layer first like this, within that domain model we get how patterns can be used in this model, Step2 was to generate multiple architectural candidates, here we can ask gpt to create the structure including the other layers and the domain model it generated, suggest architectural styles that can be used to combine them all. OR in step 2 we can just focus on architecture style for the domain specifically asking gpt to not consider other layers.