PartsTrack Overview

In Domain-Driven Design (DDD), a layered architecture is used to separate parts of the application so each layer has a clear, focused responsibility. This separation keeps the business logic isolated from other concerns, making it more manageable and closely aligned with the domain.

**Domain Layer:**

* **Entities (e.g., Part, Warehouse, Inventory)**: These represent core business objects, capturing essential properties and behaviors in the domain.
* **Catalogs (e.g., IPartCatalog, IInventoryCatalog)**: These are interfaces for accessing domain data without binding the domain to any specific database or technology.
* **Services (e.g., InventoryService)**: Domain services encapsulate business logic that spans multiple entities or doesn’t naturally fit within a single entity.

**Application Layer:**

* **Use Cases (e.g., InventoryUseCases)**: Use cases coordinate domain services to accomplish tasks, acting as a bridge between the domain and the console interface without containing the business logic itself.
* **DTOs (Data Transfer Objects)**: DTOs structure data for communication between layers, exposing only necessary details and preserving the encapsulation of the domain model.

**Infrastructure Layer:**

* **Catalog Implementations (e.g., PartCatalog, InventoryCatalog)**: These implement repository patterns to manage data persistence with SQL, abstracting storage details from the domain.
* **DbContext**: This manages database access, enforcing a clear separation between data handling and business logic.

This layered setup aligns with DDD, allowing each layer to be tested, modified, or replaced independently.

**2. Focus on the Domain Model**

DDD emphasizes a rich domain model that reflects the business’s core processes. In PartsTrack:

* **Entities (e.g., Part, Warehouse, Inventory)**: Each entity encapsulates data and behaviors that are directly tied to the business. For instance, Inventory represents the relationship between a Part and a Warehouse and might include operations like updating quantities.
* **Repositories (Catalogs)**: Instead of direct database access, catalogs (e.g., IPartCatalog, IInventoryCatalog) allow the domain to specify "what it needs" without defining "how" it should be provided. This flexibility improves testability.
* **Domain Services (e.g., InventoryService)**: These services manage business rules that span multiple entities, like updating or retrieving inventory details.

**3. Encapsulation and Abstraction**

DDD enforces encapsulation by containing domain logic within entities, services, and repositories rather than spreading it across the application:

* **Encapsulation**: For example, InventoryService contains inventory management rules, so other application parts don’t need to understand or replicate these details.
* **DTOs**: These communicate data between the application and domain layers, concealing domain complexity from external consumers.
* **Abstraction**: Catalog interfaces (e.g., IPartCatalog and IInventoryCatalog) abstract data access details, letting the domain layer focus on business logic while the infrastructure handles SQL logic.

**4. Separation of Concerns**

Each part of the system has a specific role, adhering to the separation of concerns principle:

* **Domain Layer**: Dedicated to business logic and rules.
* **Application Layer**: Manages use cases and orchestrates domain operations without containing domain-specific rules.
* **Infrastructure Layer**: Handles external concerns like data persistence, logging, and configuration.

**5. Dependency Injection and Inversion of Control**

DDD uses Inversion of Control (IoC) to keep the domain layer independent of infrastructure concerns. With Dependency Injection (DI) in Program.cs, IoC is achieved:

* Services like InventoryUseCases rely on abstractions (e.g., IPartCatalog, IInventoryCatalog) rather than concrete classes (e.g., PartCatalog, InventoryCatalog). This means the domain and application layers can remain unchanged even if implementations change.
* DI also simplifies testing by allowing mock or in-memory versions of catalogs to be injected instead of the actual database implementations.

**6. Explicitly Defined Boundaries**

In DDD, bounded contexts define boundaries where the domain model applies. Here, each catalog interface (e.g., IPartCatalog, IInventoryCatalog) establishes a boundary for managing specific parts of the system, ensuring that each module works within its context and preventing unintentional dependencies.

**Summary**

Organizing the PartsTrack project around DDD principles achieves:

* A domain-focused design that reflects the business context rather than technical details.
* Loose coupling between layers, making them easier to test, maintain, and modify independently.
* Separation of concerns, with each layer responsible for specific application aspects.
* A flexible and scalable architecture that can adapt to future changes in business requirements without major restructuring.

This DDD approach is ideal for projects with complex business rules, ensuring that the software can evolve alongside the business domain’s needs.