Domain-Driven Design (DDD) Cheatsheet

A concise reference for Domain-Driven Design principles, patterns, and practices to help build software that closely reflects the business domain.

Core Concepts

Domain

The **domain** is the specific subject area to which the user applies a program. DDD focuses on understanding and modeling this domain.

Key aspect: Shared understanding between developers and domain experts.

Ubiquitous Language: A common language used by all team members (developers, domain experts, etc.) to avoid misunderstandings.

Key aspect: Improves communication and reduces ambiguity in code and documentation.

Bounded Context

A **bounded context** defines the scope in which a particular domain model applies. It represents a semantic boundary.

Key aspect: Isolates domain models, preventing them from becoming overly complex.

Each bounded context should have its own Ubiquitous Language.

Key aspect: Ensures clarity and consistency within the context.

Strategic vs. Tactical DDD

Strategic DDD	Focuses on the big picture: understanding the overall domain, identifying bounded contexts, and defining relationships between them.
Tactical DDD	Focuses on the implementation details within a single bounded context: designing aggregates, entities, value objects, and domain services.

Tactical Patterns

Entities

An **entity** is an object with a distinct identity that persists over time. The identity, rather than the attributes, distinguishes one entity from another.

Example: A `Customer` identified by their ID, even if their address changes.

Entities have a lifecycle and can change state.

Key aspect: Focus on identity, state, and behavior.

Value Objects

A **value object** is an immutable object defined by its attributes. Two value objects are considered equal if their attributes are equal.

Example: An `Address` consisting of street, city, and zip code. Changing any part of the address creates a new `Address` object.

Value objects are often used to represent concepts that don't have a unique identity.

Key aspect: Immutability, equality based on attributes, and conceptual wholeness.

Aggregates

An **aggregate** is a cluster of associated objects that are treated as a single unit for data changes. One entity within the aggregate is designated as the **aggregate root**.

Example: An `Order` aggregate with the `Order` as the root, containing `OrderItem` value objects.

All external access to the aggregate is controlled through the aggregate root.

Key aspect: Enforces consistency and encapsulates complexity.

Domain Services

A **domain service** is a stateless operation that performs a significant process in the domain that doesn't naturally fit within an entity or value object.

Example: A `TransferService` that transfers money between two accounts.

Services often involve multiple entities or external systems.

Key aspect: Represents domain logic that transcends single objects.

Repositories

A **repository** provides an abstraction for accessing data persistence. It acts as a collection-like interface for domain objects.

Example: A `CustomerRepository` that provides methods for finding, adding, and removing `Customer` entities.

Repositories decouple the domain model from the data access layer.

Key aspect: Enables easier testing and switching between persistence mechanisms.

Strategic Patterns

Context Mapping

Context Mapping is the process of defining the relationships between bounded contexts.

Key aspect: Ensures clear understanding of dependencies and interactions between different parts of the system.

Common context map patterns include:

- **Partnership**: Two contexts collaborate closely and succeed or fail together.
- **Shared Kernel**: Two contexts share a subset of the domain model.
- **Customer-Supplier**: One context provides services to another.
- **Conformist**: One context aligns its model to the upstream context.
- **Anticorruption Layer**: A layer that translates between different models to prevent corruption of the downstream context.

Subdomains

A **subdomain** is a specific area within the overall domain. Identifying subdomains helps to break down the complexity of the problem.

Key aspect: Focus on different areas of expertise and responsibility.

Subdomains can be classified as:

- **Core Domain**: The most important and differentiating part of the business.
- **Supporting Subdomain**: Important but not differentiating.
- **Generic Subdomain**: Not specific to the business and can be purchased off-the-shelf.

Implementation Considerations

Event Storming

Event Storming is a workshop-based method for collaboratively exploring a domain and identifying key events, commands, and aggregates.

Key aspect: Facilitates communication and shared understanding between developers and domain experts.

Involves domain experts, developers, and testers working together to model the domain on a large surface using sticky notes.

Benefits: Quick way to visualize the domain and identify potential problems.

CQRS (Command Query Responsibility Segregation)

CQRS is a pattern that separates read and write operations for a data store.

Key aspect: Allows for optimization of read and write models independently.

Commands are used to update data, while queries are used to retrieve data. This separation can improve performance and scalability.

Considerations: Increases complexity and requires eventual consistency for read models.

Eventual Consistency

Eventual Consistency is a consistency model where updates to data may not be immediately reflected in all replicas or read models.

Key aspect: Data will eventually become consistent, but there may be a delay.

Often used in distributed systems and CQRS architectures.

Considerations: Requires careful handling of potential data inconsistencies.