Object Oriented Design

SOLID Principles

- **Single Responsibility Principle**
  - A class should have one and only one reason to change, meaning that a class should have only one job.
- **Open - Closed Principle**
  - Objects or entities should be open for extension, but closed for modification.
- **Liskov Substitution Principle**
  - Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program." See also design by contract.
- **Interface Segregation Principle**
  - A client should never be forced to implement an interface that it doesn't use or clients shouldn't be forced to depend on methods they do not use.
- **Dependency Inversion Principle**
  - Entities must depend on abstractions not on concretions. It states that the high level module must not depend on the low level module, but they should depend on abstractions.

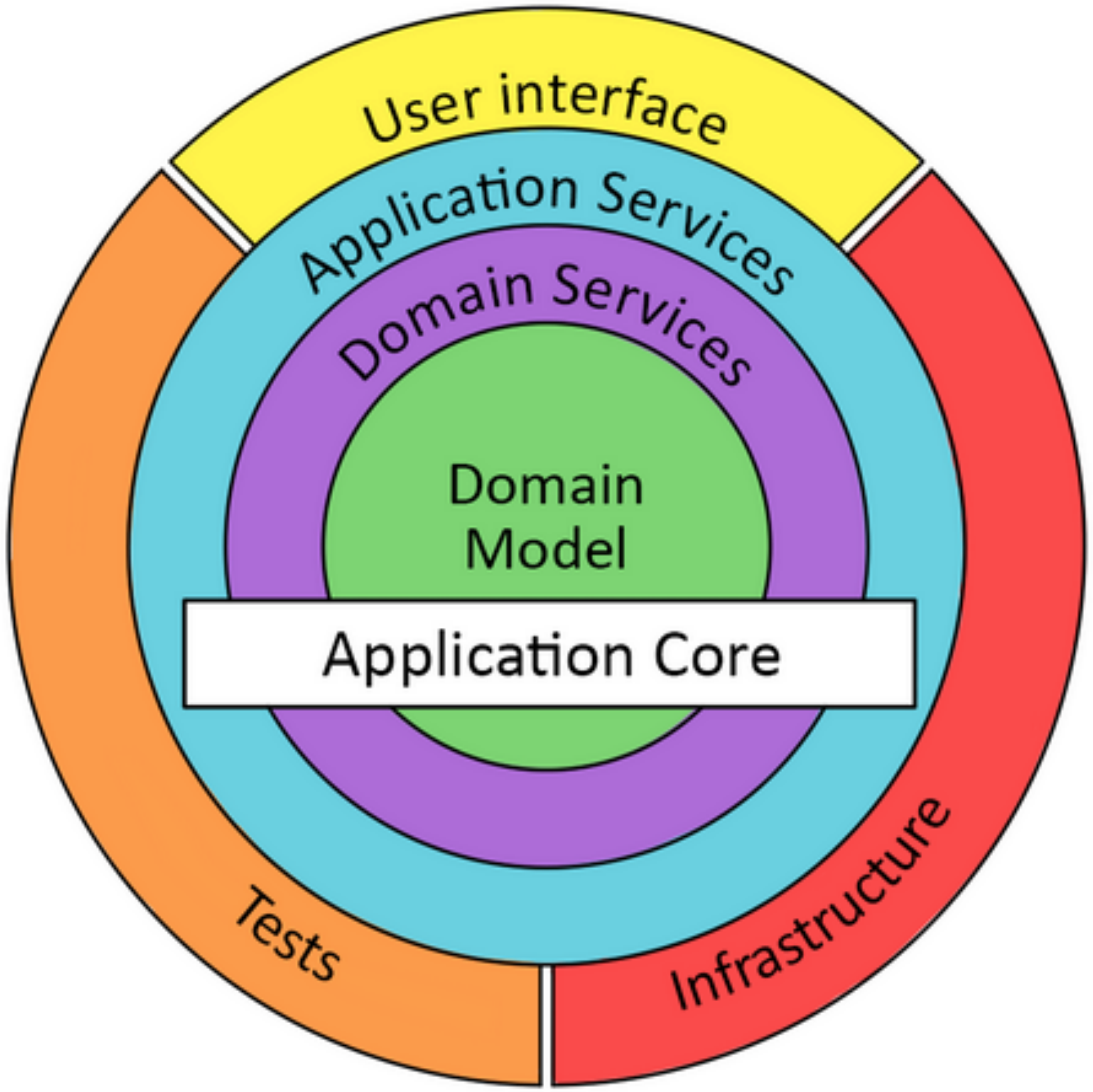
Design Patterns

- **Factory:** To separate the responsibility of object creation from it’s user
- **Adapter:** To provide known interface to unknown object
- **Proxy:** To do pre/post processing anonymously
- **Decorator:** To extend an object at runtime
- **Template:** To select varying part of algorithm at compile time
- **Chain Of Responsibility:** To escalate conditionally in an hierarchy
- **Builder**: To construct a complex object, step-wise
- **Facade**: To hide internal business interactions

Design Patterns

- **Abstract Factory**: To select factory for an interface
- **Singleton**: To limit the number of instances of a class to maximum of one
- **Composite:** To treat sum of the parts as a whole
- **Mediator:** To enable intra-group asynchronous communication
- **Observer:** To enable inter-group asynchronous communication
- **Command:** Object oriented callback
- **Visitor: To decorator an object graph**
- **Iterator:** To traverse object graph without knowing the internal structure
- **Strategy**: To select algorithm at runtime

Onion Architecture



Domain Driven Design

- **DDD:** A design approach that is driven by domain, not technology

- **Domain**
  - The sphere of knowledge, in problem space
  - The subject area to which the system belongs to
  - Core domain, supporting domain, generic domain
- **Model**
  - Abstractions to describe the domain in solution space
- **Context**
  - The scope in which the model is valid
- **Ubiquitous Language**
  - A language structure around the domain model
  - Used by the team

DDD Patterns

- **Domain Service**
  - Core for the given bounded context
  - Implements Business Logic
  - Involves more than one domain object
  - Not CRUD operations
- **Application Services**
  - Entry point to access the domain layer
  - Interface to external consumers
  - REST layer
- **Infrastructure Services**
  - Generic technical services
  - Email, Queues and etc
  - Typically ready-made services

DDD Patterns

- **Entities**
  - Identifiable and Mutable
  - Can be created with minimum requirements
- **VO/Value Objects**
  - No identity, Immutable and interchangeable
  - Alternative to primitive obsession
- **DTO/Data Transfer Object**
  - Data containers for transport across layers
  - Hides internal data structures
- **Aggregate**
  - Cluster of objects with an aggregate root
- **Repository**
  - Handles CRUD on Aggregates
  - Gives an impression of in-memory storage

Strategic DDD

- **Context/Bounded Context**
  - Packages, Namespaces, Modules
  - Subsystems or Microservices
  - Specific Teams, Databases, Language
- **Continous Integration**
  - Common code repository
  - Frequent Builds
  - Automated Tests
- **Context Map**
  - Maps different bounded contexts
- **Bounded Context Vs Subdomain**
  - Subdomain belongs to problem
  - Bounded Context belongs to solution
  - Multiple bounded-contexts within a given subdomain

Integration

- Multiple Bounded Contexts
- Legacy Systems
- Third-party Services
- **Integration Strategies**
  - **Shared Kernel:** Shared minimal domain model
  - **Customer/Supplier:** Aligned priorities, tasks
  - **Conformist:** At the mercy of third-party
  - **Anti-Corruption Layer:** Translation layer
  - **Open Host:** Services for others to integrate
  - **Published Language:** JSON/XML Messages
  - **Separate Ways:** Let’s not integrate
- Final solution may involve multiple strategies

Case Study

- **Product Search and Catalog**
  - Users should be able to see the product listings
  - Users should be able to search for products
- **Orders, Payments and Delivery**
  - Users should be able to place orders
  - Users should be able to make payments
  - Users should be able to receive the products
- **Security**
  - Authentication and authorisation
- **Customer Management**
  - Maintain customer details, preferences and etc.,

Case Study

- **Domain**
  - E-Commerce
- **Without Domain Driven Design**
  - A monolith
- **With Domain Driven Design**
  - Core Subdomain
    - Catalog
  - Support Subdomains
    - Order Management
    - Shipping
    - Customer Management
  - Generic Subdomains
    - Security
    - Payment

Final Words

Do not base the design on databases
Base the design on domain