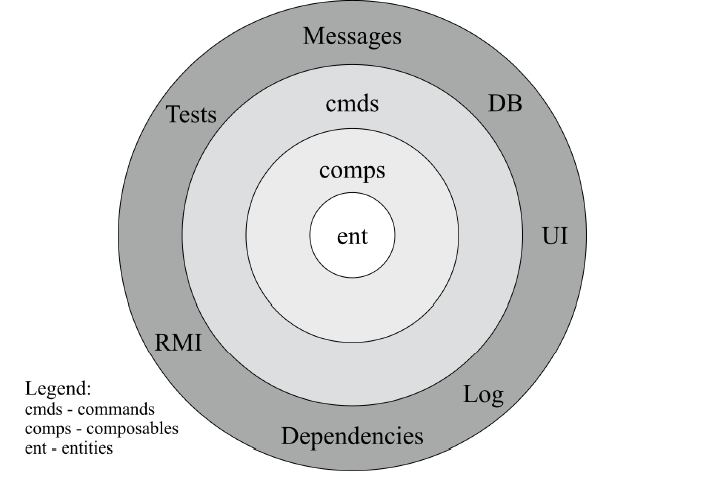
Software Architecture

A layered approach solution is followed to describe the Components internal structure as:



This design follows a separation of concern principle, stating testability without UI, Dababase, Application Server or Web Server Layer, and independence from frameworks as the business rules should not rely on anything from outside. In other words, this approach enforce containments dependencies, technologically and to other components.

Four main layers are found: Connectors, Commands, composable services and entities, considering in general that the further you go, the higher the software level is, where the outermost layers represent mechanism and the innermost represent policies.

Source Code Dependency

The main rule for this structure depicts that source code dependencies must point only inwards, meaning that the name of the code artifacts existing in the outer layer, must not be mentioned in the inner. The physical artifacts produced by a given layer can only import the artifacts produced by any layer contained inside its circle. When crossing a boundary, all the data is in the form that is most convenient for inner layer.

*Connectors*: components in this layer may be driven by any input source, e.g users, other components, automated tests, while the innermost layers remain completely independent from that input. Base rule follows that when a component need to send something outside, it does it through the connectors. Connectors are responsible for converting any input from the outside world to the commands or entities format and the other way around (for example REST1 is a Web Service specification; ORM2 is a Database Connector). They can be seen as *inbound* when used by other components to reach their providers, or as *outbound* when used by the infrastructure through *dependency inversion*.

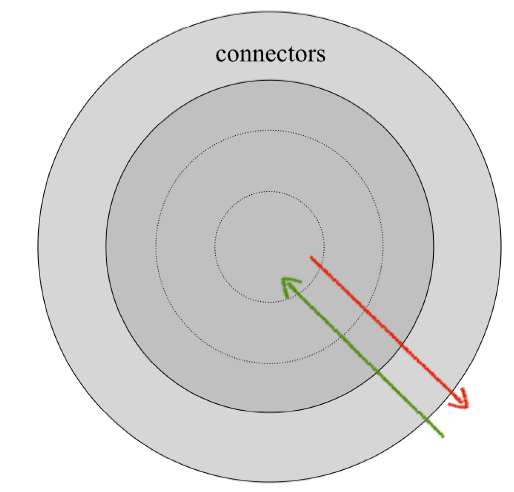
*Commands*: This layer can be referred like a business logic implementation of a given component, orchestrating the flow of data from and to the entities, representing all the use cases. Each command is in a context of a Transactional Boundary and cannot depend on any other command. If a command is to be executed as a result of execution of another command, then this command should not be directly called, instead it should be encapsulated in an event or message queue to decouple the call.

*Composables*: layer containing code not belonging to the business logic domain and that is subject to be reused across different commands, meaning that it contains the abstraction allowing access to connectors and infrastructure, such as Database, messaging, web services, etc.

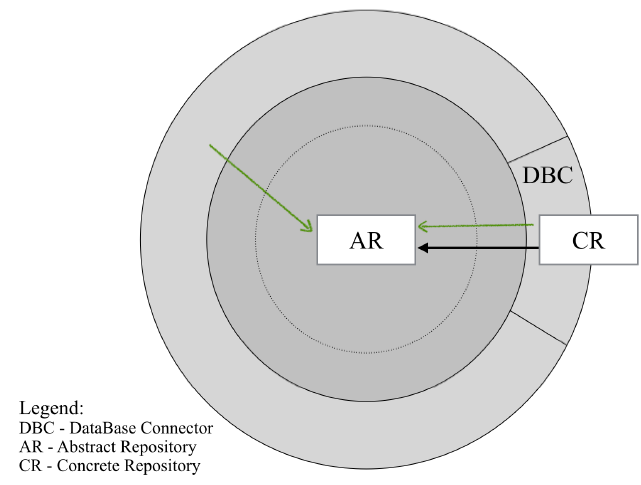
*Entities*: contains business rules and the state of the component, where each entity represents a business object that contains the most general rules shared by the use cases, materialized in the outer layers.

Data Flow

As can be seen on the following picture, the green arrow pointing inward represents a legal source code dependency and the red (pointing outward), an illegal one.



An output will always be produced where general flows go from a connector through inner layers, and then back again to a connector. The same principle applies when an inner layer must use infrastructure to interact with another component or to store data in a Database, for example. This case can be solved using the principle of *dependency inversion* through interfaces and inheritance relationship to take advantage of Polymorphism principle opposing the flow of control at the necessary points. As an example, a command can inject an abstraction (e.g an interface or an Abstract class) living in an inner layer, but implemented by and resolved to a concrete type belonging to a given connector. This can be illustrated in the following picture:



Testability and Modularity

With this separation of concern, we create modular components that are intrinsically testable, carrying with it all the benefits that this implies.

Technological dependencies are confined to components, the code base is more resilient to technological shift, meaning that the choice of frameworks will no longer affect the majority of the component's code base, increasing deployment and flexibility, as a consequence.

External representations must be specified, constraining that internal representation (entities) cannot be exposed by the public interfaces. So, taking advantage of the Data Transfer Object patter is necessary, avoiding exposure of Internal representations.

The main idea to remain is that dependencies to frameworks and other dependencies are circumstantial and not relevant to the business domain, therefore they must be contained in peripheral to the software design.

Component deployment artifacts

As was explained before, integrity is an important part of this architecture design, so illegal dependencies must be identifiable at running time. This constraint influences the application *physical design*, that is materialized in the component modules. These modules should the materialize the dependency constraints explained above and also ensure segregation between public (accessed from the outside) and private (accessed exclusively from component implementation) artifacts. For the sake of modularity, public artifacts cannot depend on the private artifacts, otherwise encapsulation is lost due to transitive dependency to other module implementation details.

Additionally, each connector should be encapsulated in its own module, since they deal with different technological aspects.

Behavioral Constraints and Life-cycle

External stimulations should be classified either as actions or events. Actions are described as requests from other components or UI whereas events represents notifications issuing state changes. Their implementations are materialized in the command layer, trough commands (allusive to the Command Design Pattern) and later through event handlers.

When a request arrives at the *connector layer,* an instance of the corresponding command is obtained or created, and it is scheduled to be executed in an available thread. Once the command is executions is finished, both the command instance and the selected thread are released for subsequence use. Considering latency requirements, I/O operations cannot be performed in events handling execution paths, and so the commands should specify the transaction boundary, that is the command's execution delimits the transactions.

Specific class instance cannot influences component interaction (no meaning for global object's identities), and all data structure at each layer's public interface must be immutable, using the Value Object Design Pattern.

Component Design

*Core:*Contains implementation of the domain entities and domain policies.

*Composable:* does not represent domain concerns, but rather the utilities aimed and preventing code duplication, applying the DRY principle.

*Commands*: definitions of the component use cases, that materializes on implementation of actions and event handlers.

*Connector:* contains the implementation of the requirements for the supported deployments (i.e REST, SOAP WS).For supporting other deployments, it only requires creation of the corresponding connectors.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Tasks to be performed

1- Create Command module

2- Create REST-connector module

T