Architectural Decomposition

Navigating toward Packaging in a highly customized environment

Bill Anderson

February 24, 2021

Agenda

1. High-Level Goals
2. Guidelines
3. Gradual Transition
4. Rethinking the “Code” Approach
5. Common Language
6. Decomposition
7. What is SF_DXCore (Core) ?
8. Business To Technology
9. Summary

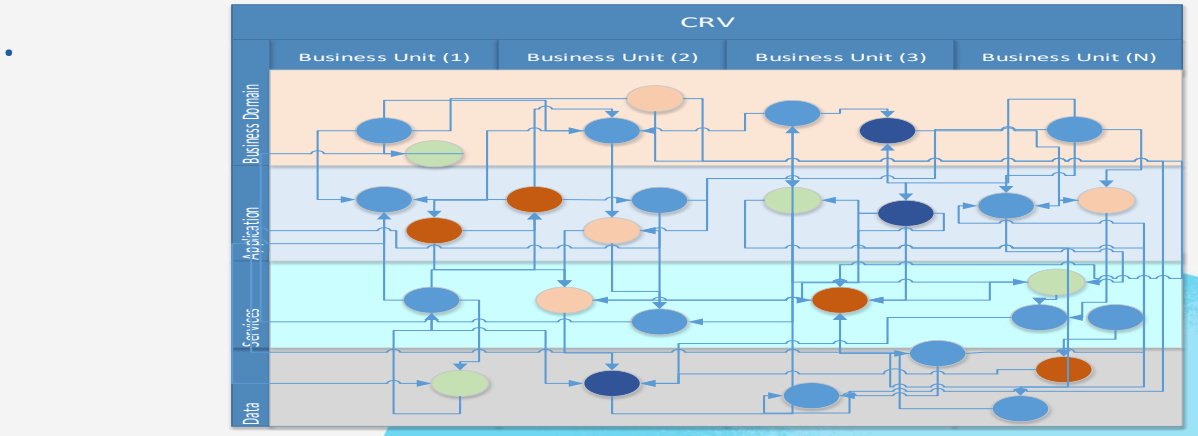
Overview

Concepts, Guidelines, Domain Driven, et. al.

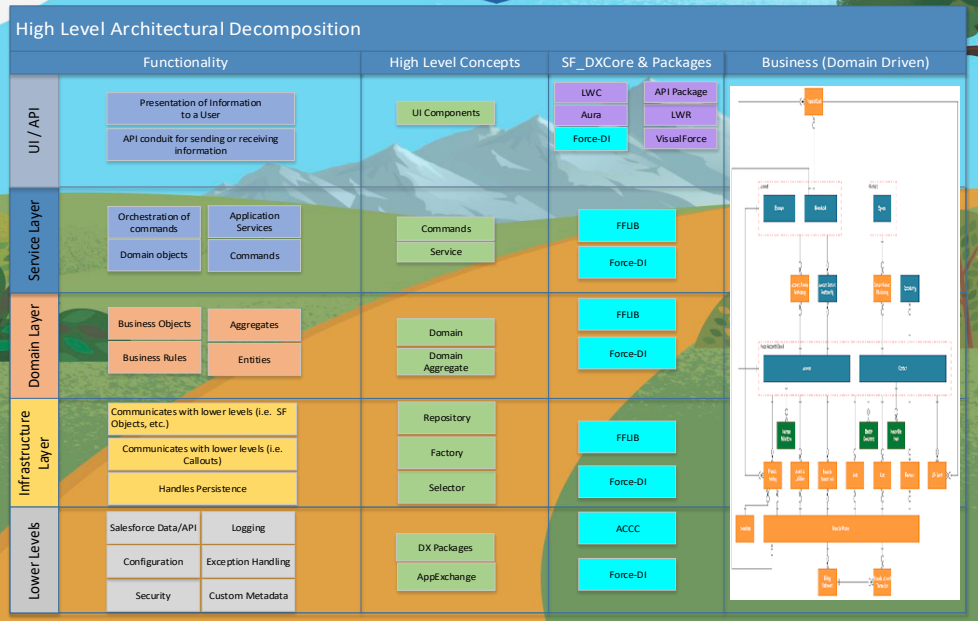
The slides that follow cover a broad range of topics to help walk through the high-level steps of transitioning from Business to Technology.

If you already are very familiar with the terms and concepts such as, *Domain-Driven*, *Layers*, *Design Patterns*, *Separation of Concerns*, *etc.*, you can fast-forward to the "Business To Technology" slide.

In order to reduce the size of the slides some concepts and insights are left out.



Moving toward



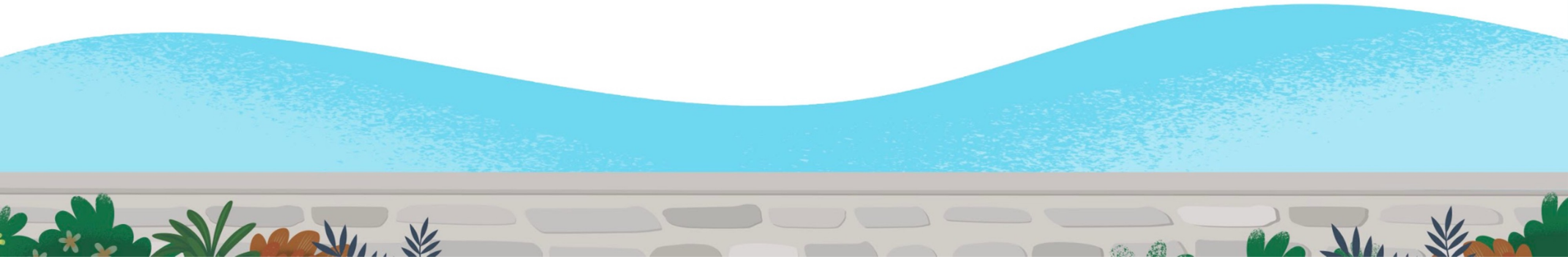
High-Level Goals

DX Journey

High-Level Goals

Product Centric focused

- ✓ Support federated development model
- ✓ Teams release independently from each other.
- ✓ Increase developer efficiency by standardizing on a toolset for development and CI/CD
- ✓ Team focus more on Business capabilities



Guidelines

"Clicks" before Code

Strategy - Salesforce Orgs

Guiding Principles

Realizing goals requires a Strategy with *Guiding Principles*:

- *Use Out-Of-The-Box **FIRST – "Clicks" before Code**,*
- *Define **WHAT** you are doing first; Not **HOW***
- *Use common functionality from packages*
- *Do not depend on concreteness, if possible*
- *Depend on abstraction (design contracts)*
- *Do not be rigid, as change is inevitable*
- *Design for reuse*
- *If coding, used SOLID Design Features.*

Gradual Transition

How to transition with ease

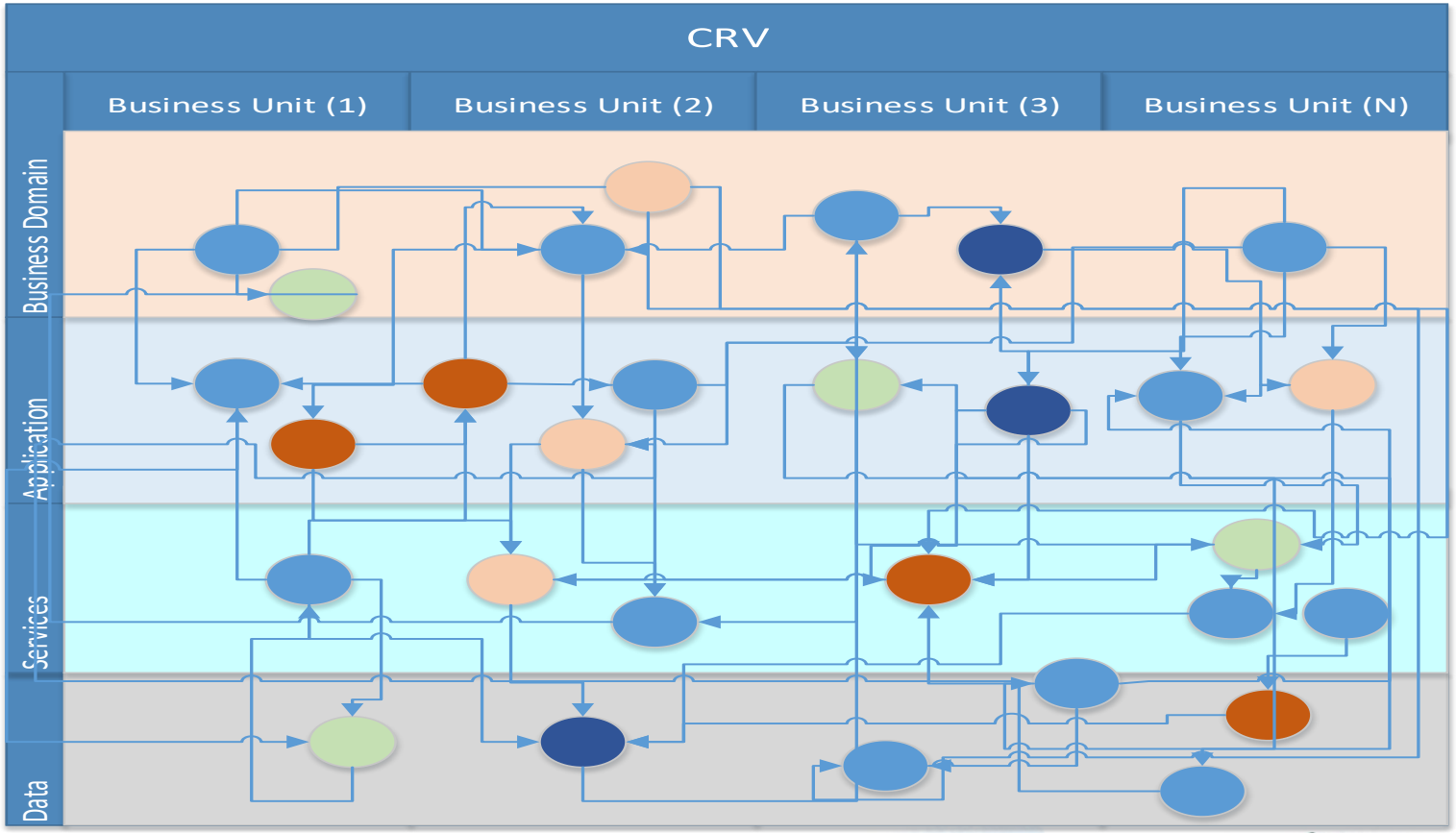
CRM – Trying to Navigate the Layers

Top to Bottom without Packaging

Large, heavily customized Salesforce Org, can become riddled with duplication, lack of reuse, negating alacrity of Business Capabilities such as *Customer Management*, *Sales Management, Marketing Interaction*, etc.

As time passed, the design falls into disrepair and is difficult to break apart let alone package.

How can we **Transition?**



Transitioning without Impeding

Transition

- In order to meet Wells Fargo Goals, it was decided to adapt a **Core Framework** to gradually transition from the current software without impeding progress. This is facilitated as follows:
  - Bounded Context (FSC)
  - Provides a Reference Architecture
  - Architectural Roadmap
  - Governance and Oversight
  - Samples and Code Generators

Features

- Plug and Play based on Capability and Environment
- Runtime control of behavior
- Package to support reuse

Results

- Incremental Adjustment of CRM
- Allows gradual adoption
- Incremental Improvement
- Less Code by Business (IT) Units
- Quicker to Market
- Reuse
- Less Ambiguity

Rethinking the “Code” Approach

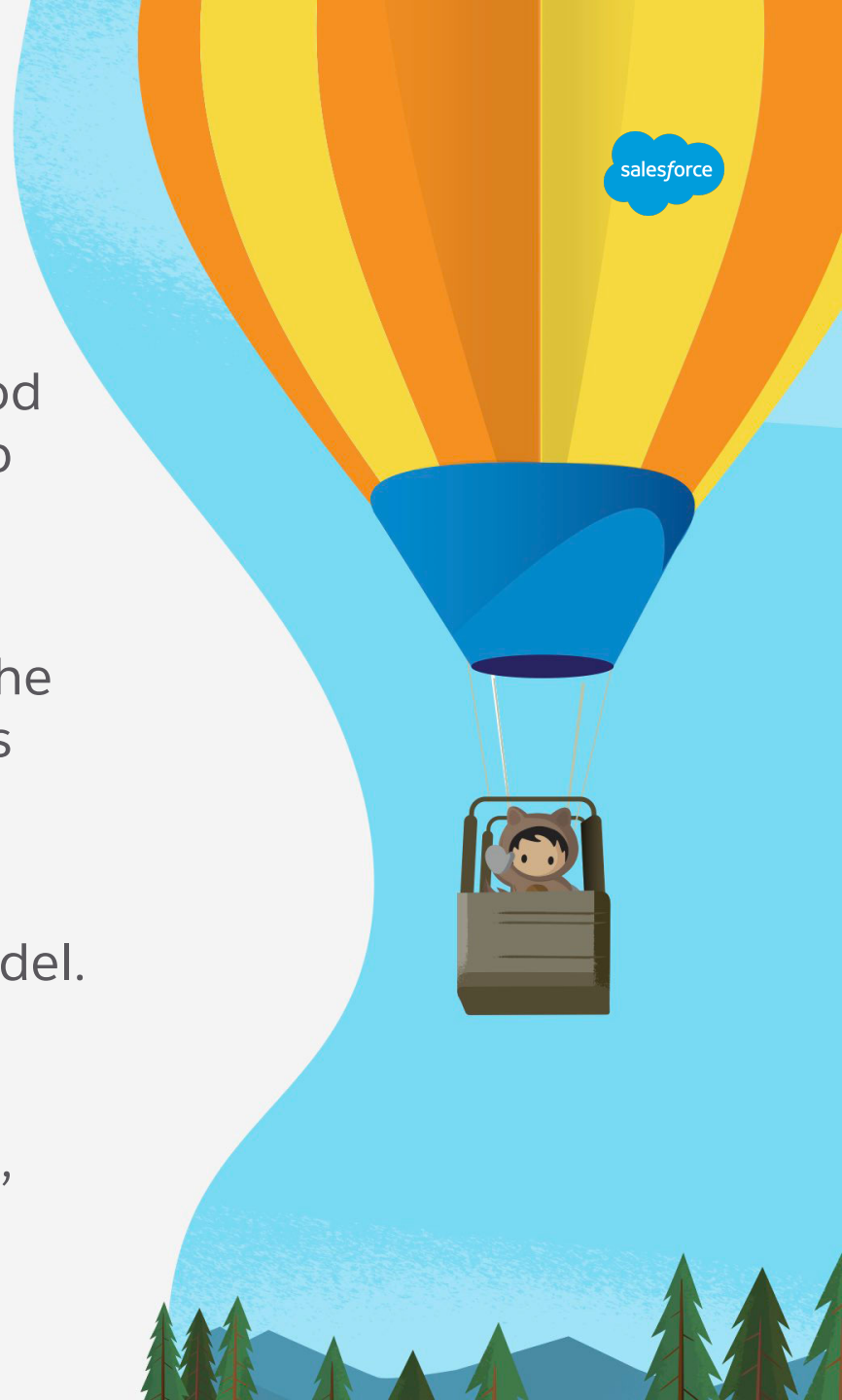
Decomposition to Ease Adoption

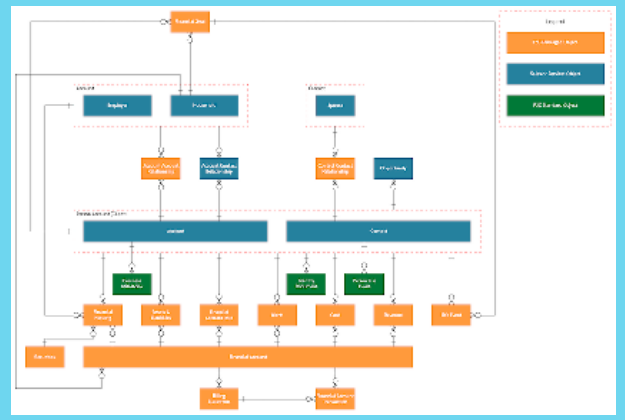
Top Down

Transition to using Core (and Packaging) may best be understood moving from the Top to Bottom. However, moving from the Top (Business) to the Bottom (Technical) needs a common Domain (*Ubiquitous*) language.

- **First**, we will short circuit the *Ubiquitous* Language by using the language (from Financial Service Cloud [*FSC*]) which provides the high-level domain language; allowing us to move from Business to Technology.
- **Second**, we will decompose Business Model to Technical Model.
- **Finally**, we will connect the dots (from top to bottom)

Not covered in this presentation are Bounded Contexts, such as, Sales, Service, Marketing, etc.





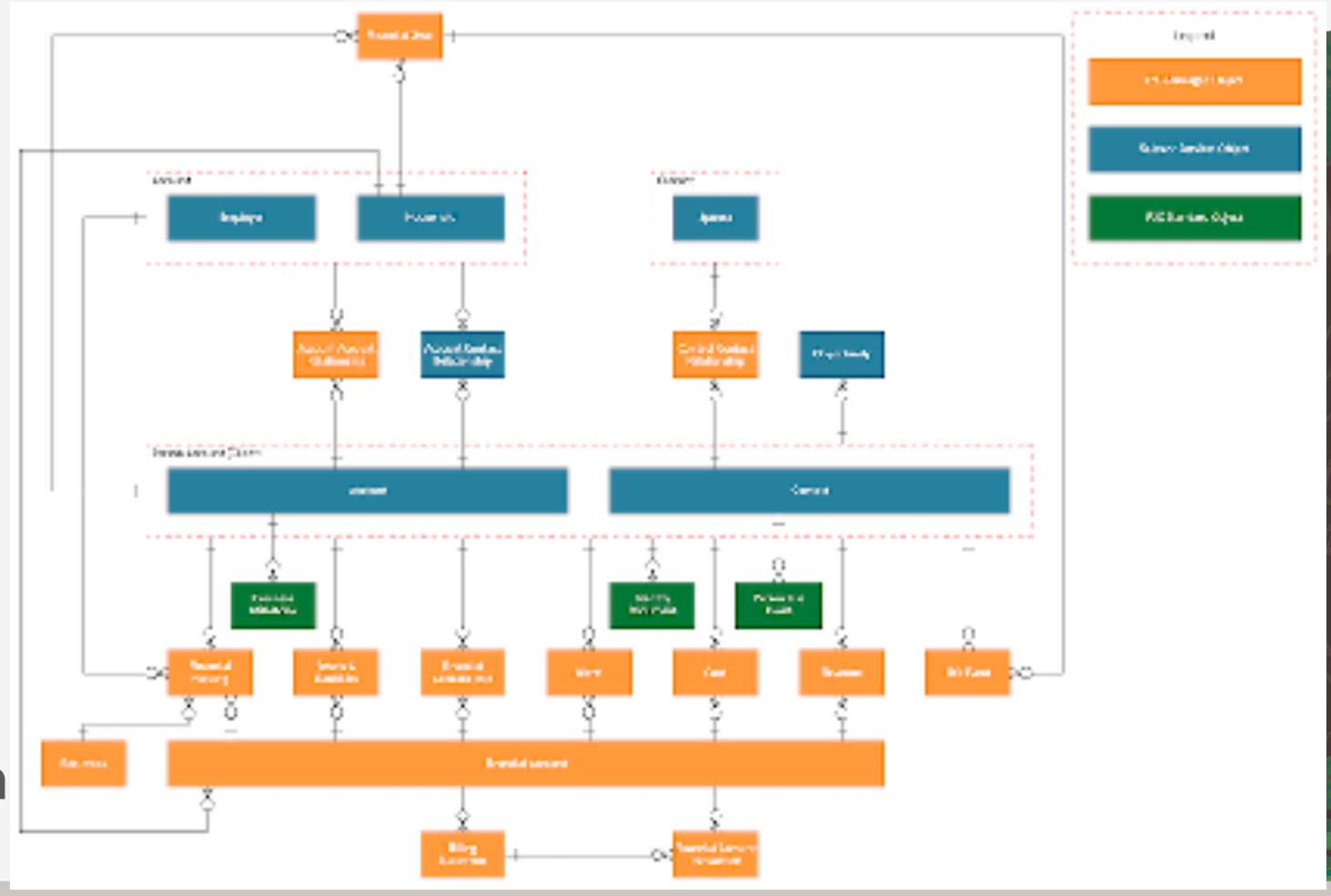
“Common Language”

Financial Service Cloud (FSC)

Transition Guide / Top Down

Transition requires a Common Language understood by both a Subject Matter Experts (SMEs) and a Software Architect. We need common Business Language to transition from the Top (Business) to the lower levels (Technical) for guidance.

This initial decomposition uses Salesforce Financial Service Cloud (FSC) as the Core Domain and is our guide for transitioning to Core Package.



Domain - Common Language

Financial Service Cloud (FSC)

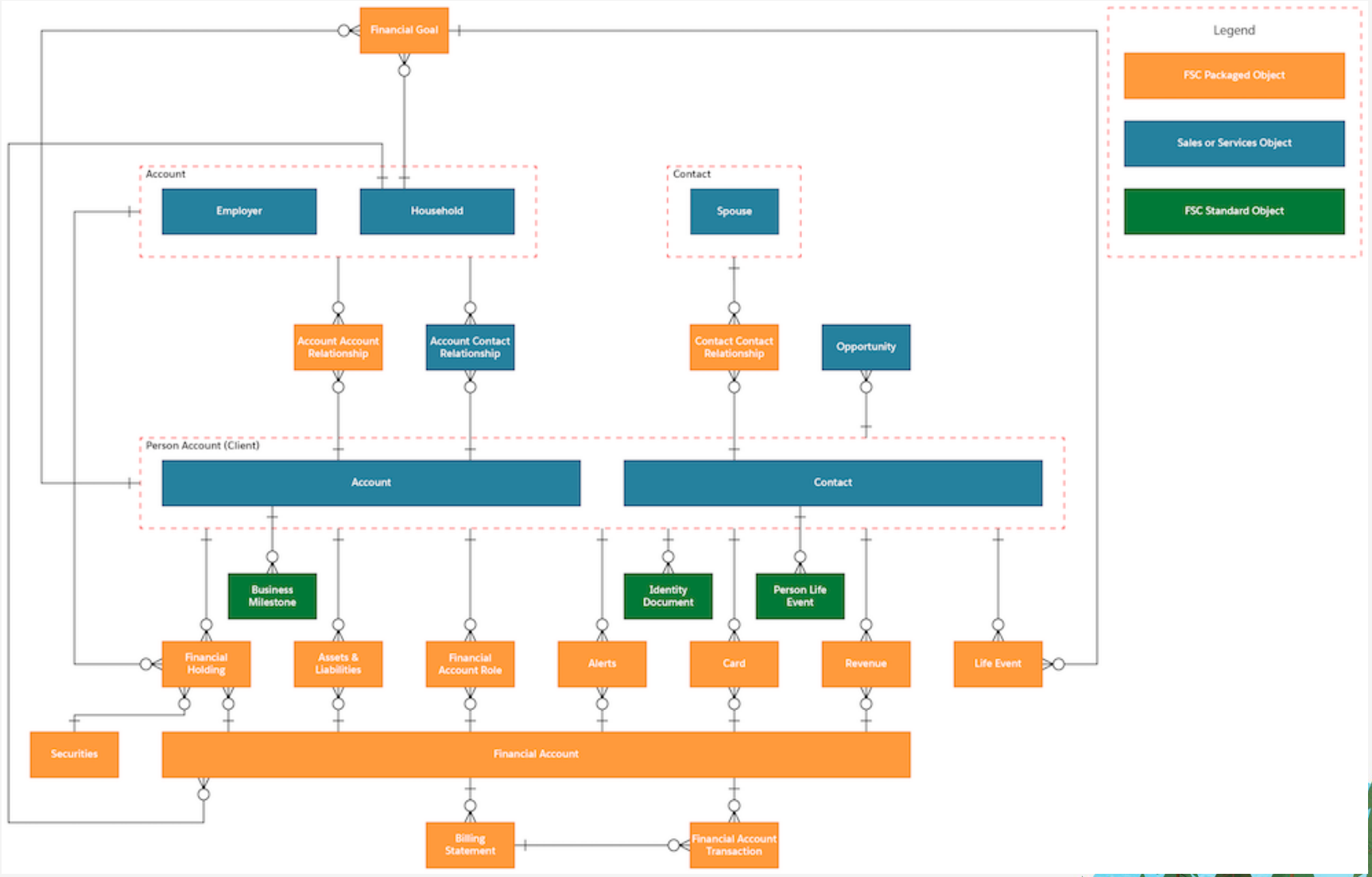
Common Language Definition

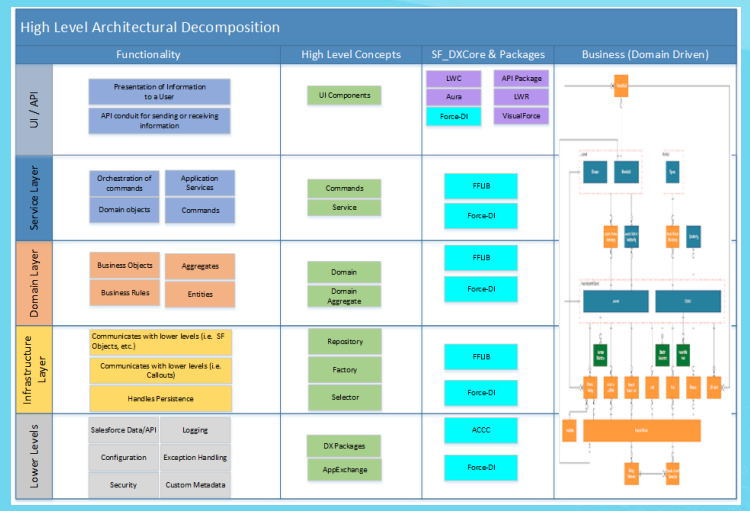
Customer – Person with Financial ties to Bank

Opportunity – Ability to sell customers new products or services

Financial Account – A brokerage account or bank account.

Products – Financial products (i.e., bank account, credit card, mortgage, ATM card, etc.)





Decomposition

“SF_DXCore” and Components

Decomposition Guide

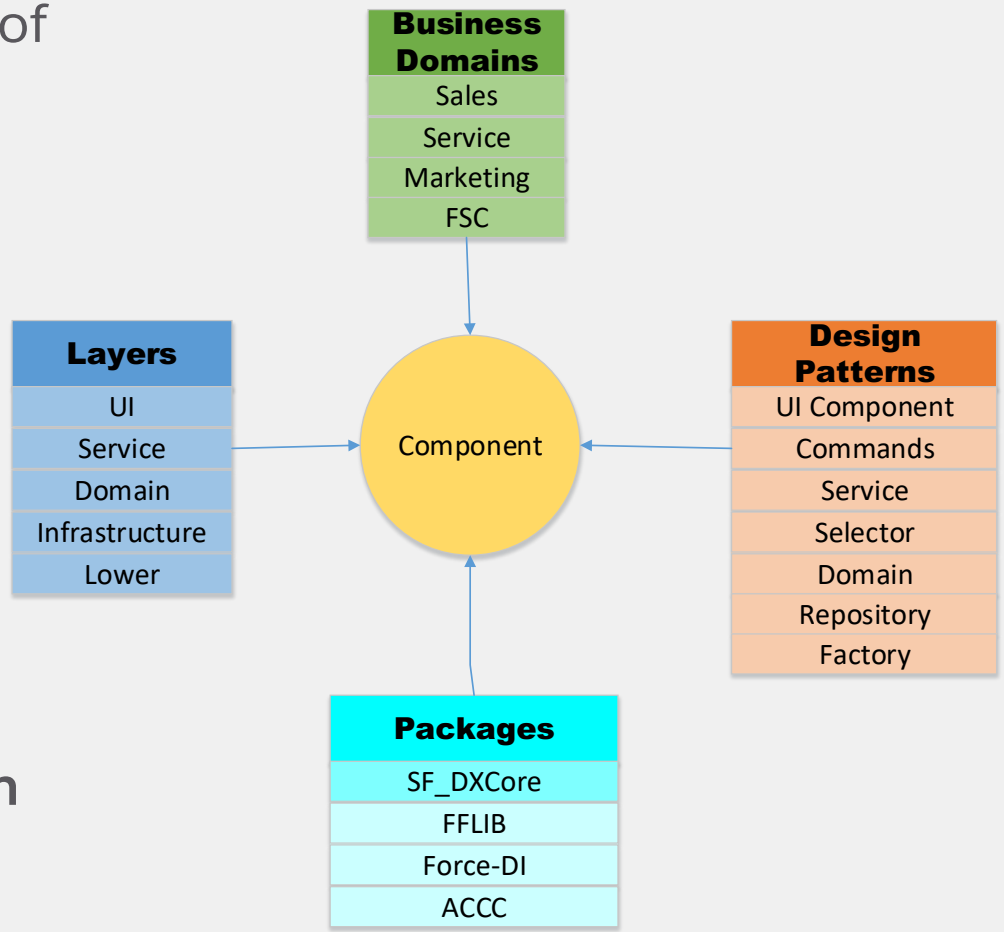
The slides that follow provides a high-level view of decomposing the Business Model. The Business Domains will be decomposed into Layers, Packages, and the use of Design Patterns. Why?

Due to the complexity, we

- Use layers to segment and manage,
- Use Design Patterns for Guidance/Best Practice
- Use Packages for Reuse and Support

We do not go into low-level details about **Design Patterns**, except:

They isolate the variability and make systems easier to understand, maintain and communicate



Layers

Decomposing to manage complexity

Most large enterprise functionality with significant business and technical complexity are best decomposed into multiple layers.

The *layers are a logical artifact to help us manage the complexity in the model (or design)*.

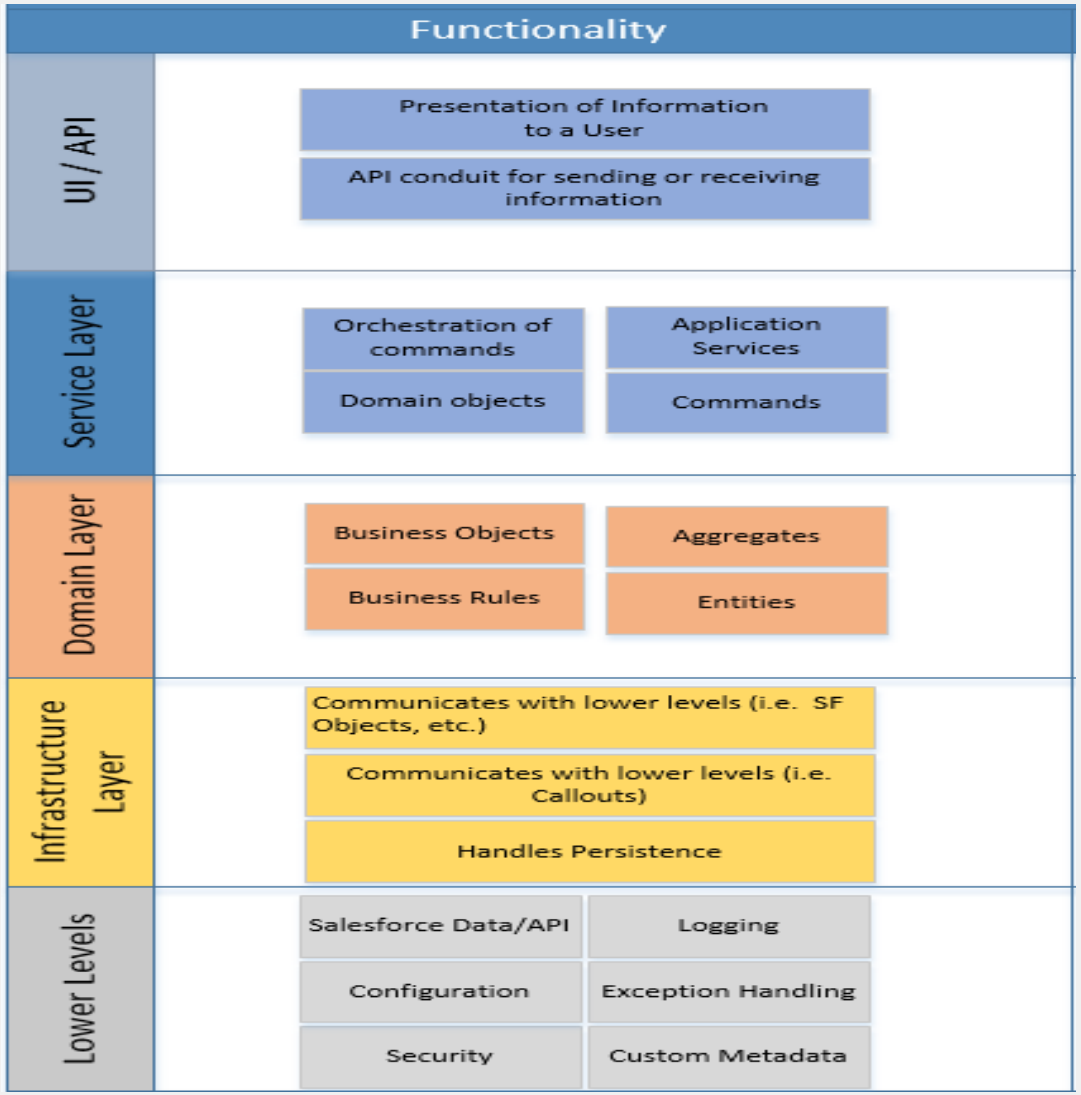
Different layers (like the domain layer versus the UI layer, etc.) might have different types, which mandate translations between those types (i.e., via a *Service*)

Layers	Description
Presentation	Provides content to the end user through GUI
Service	Bridge between the higher and lower layers
Domain	Collection of entity objects and related business logic that is designed to represent the enterprise business model
Infrastructure	Interact with the low-level data access layer
Common	Provides cross-cutting-concerns, support packages, external access

Functionality/Responsibility found in each Layers

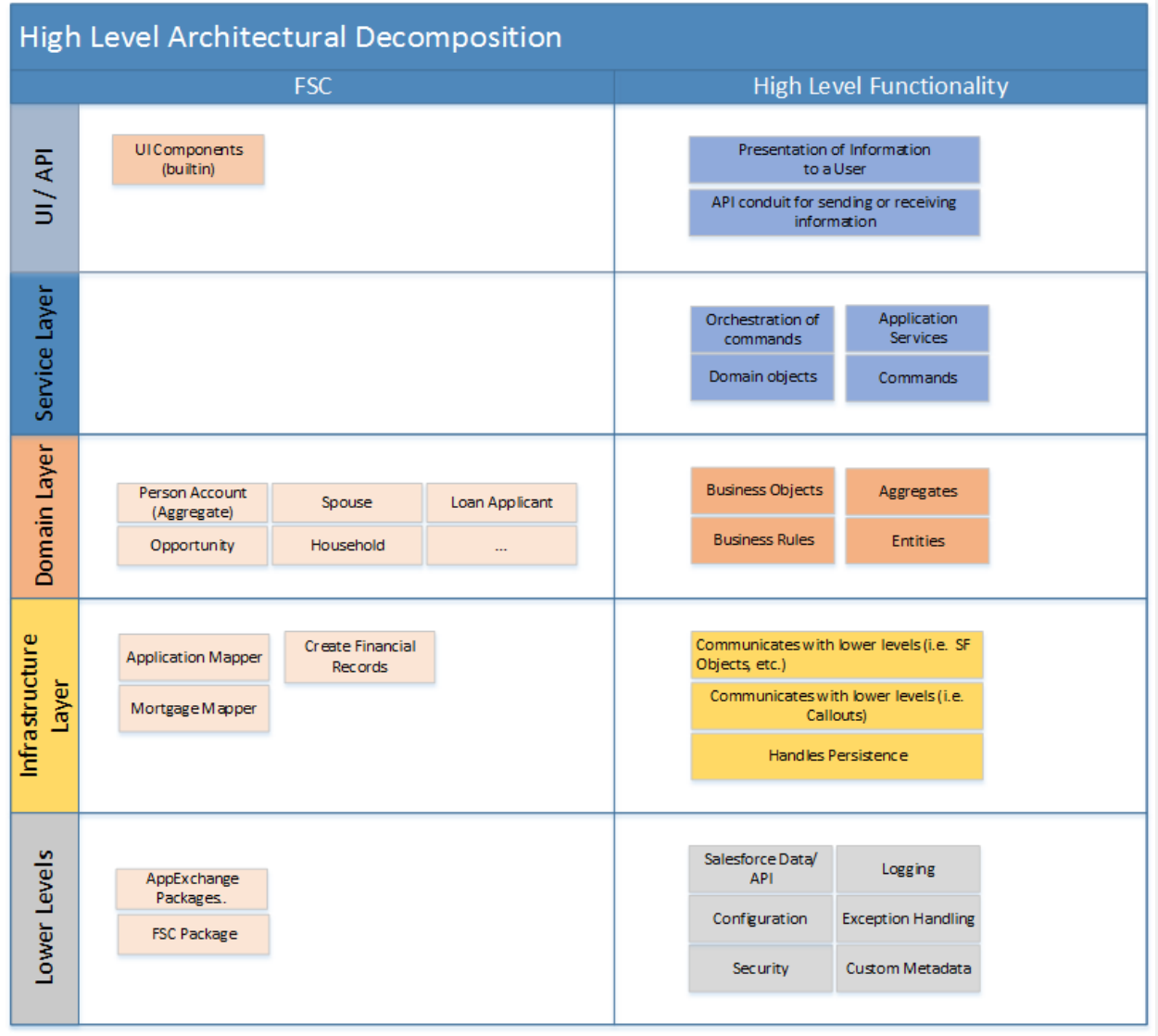
Decomposing to manage complexity

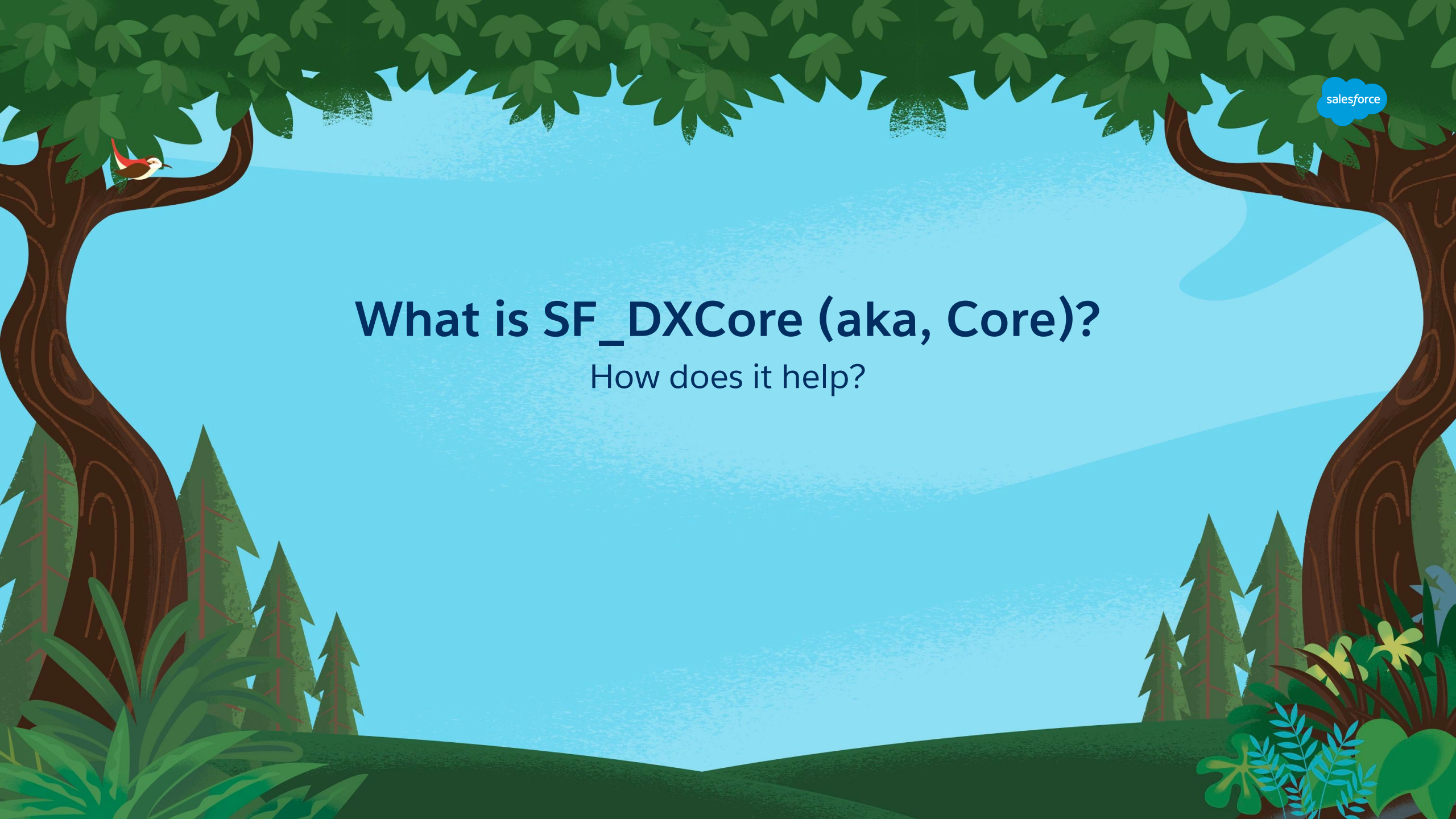
On the right, the high-level functionality/responsibility is enclosed in the layers (separation of concerns).



Decompose Financial Service Cloud to High-Level Functionality

On the right, we map the Financial Service Cloud (FSC) components to our High-Level Functionality.



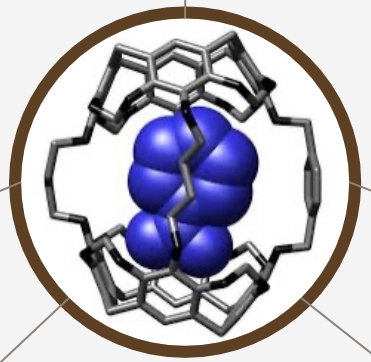


What is SF_DXCore (aka, Core)?

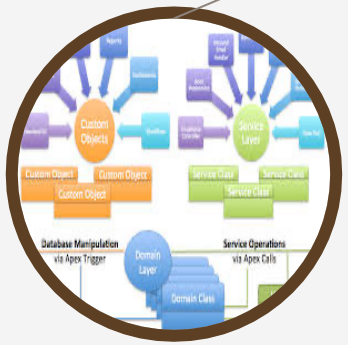
How does it help?

What is in Core Package "*SF_DXCore*" ?

Packages the other components and provides a wrapper around the other frameworks



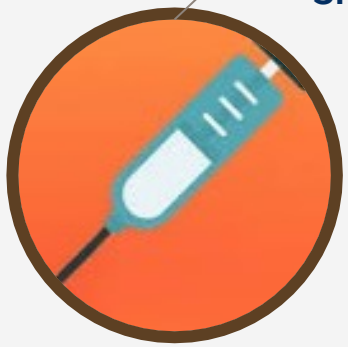
SF_DXCore



FFLIB

Provides Separation of Concerns

(Domain, Selector, Service)



Force-DI

Allows Dependency Injection; so, runtime behavior is not static

Apex Cross Cutting Concerns

Ability to change runtime behavior of cross cutting concerns; such as, logging, exception handling, etc.

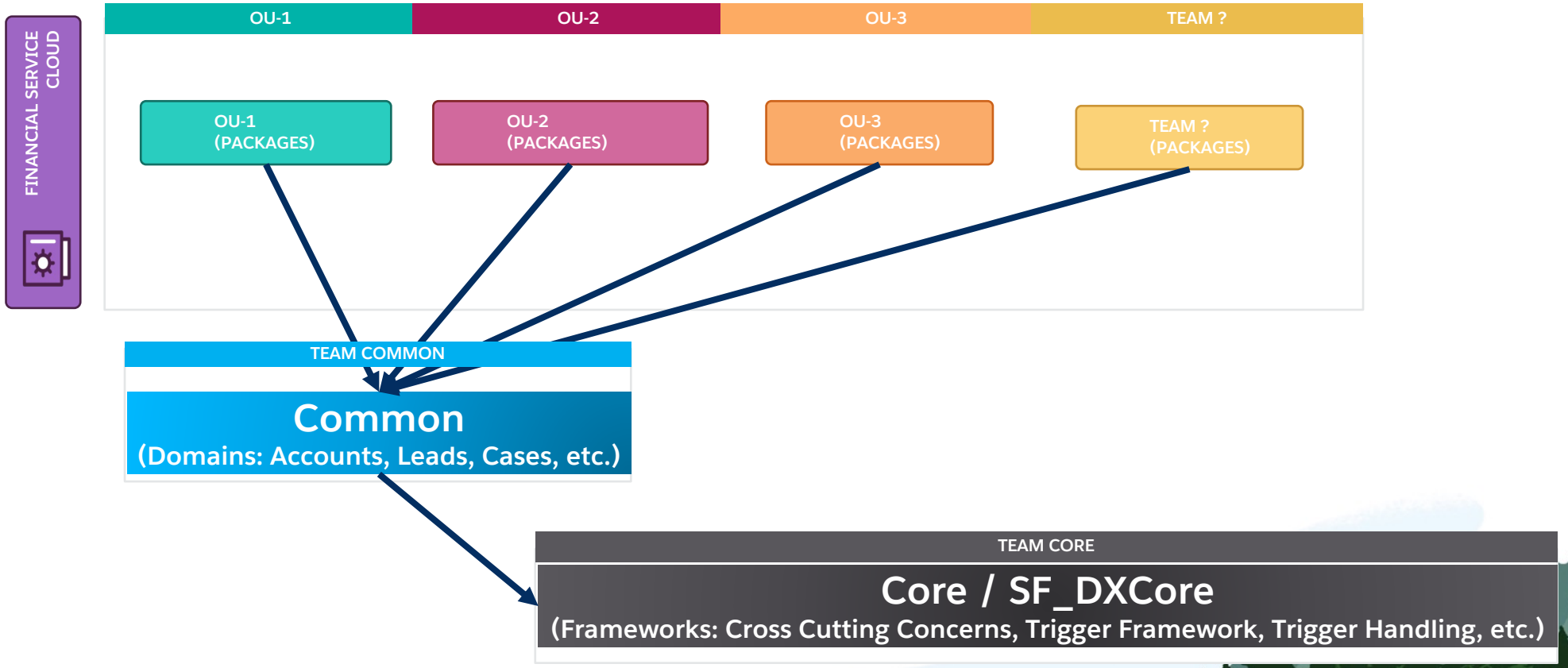
DX Packaging

Ability to create packages for the purpose of reuse

How can Teams benefit from ***SF_DXCore*** when moving into Packages?

Iterative Journey with *"SF_DXCore"*

*Decomposing specific Functionality, with the help of **SF_DXCore**, allows for an easier transition into Packaging. In addition, using an architectural decomposition process, packaging becomes easier and more adaptable with multiple teams.*



High Level Decomposition of Components into Layers

Segmenting Business Functionality into layers.

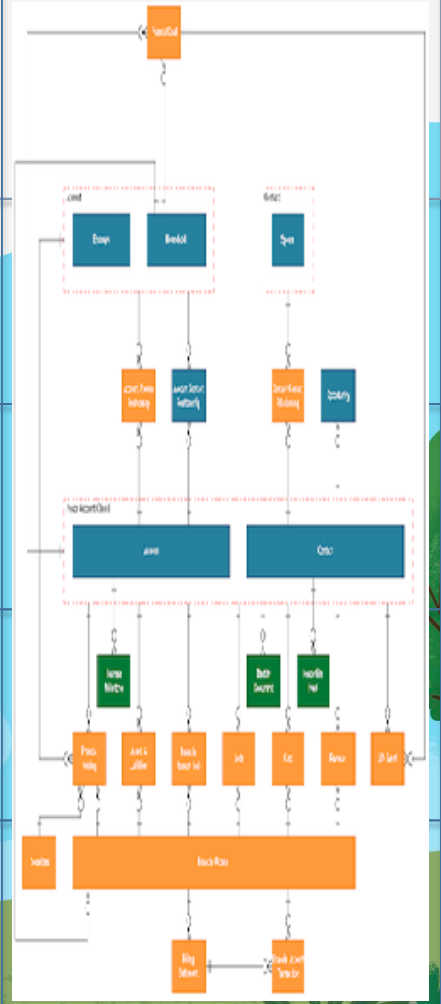
The map on the right shows salient components of ***SF_DXCore*** and where they can be used. This helps on packaging.

NOTE: Even if slow to packaging, you get the added benefit of:

- Better Maintenance
- Reuse
- Faster to Release

High Level Architectural Decomposition

	Functionality	High Level Concepts	SF_DXCore & Packages	Business (Domain Driven)
UI / API	Presentation of Information to a User API conduit for sending or receiving information	UI Components	LWC, Aura, Force-DI, API Package, LWR, VisualForce	
Service Layer	Orchestration of commands, Application Services, Domain objects, Commands	Commands, Service	FFLIB, Force-DI	
Domain Layer	Business Objects, Aggregates, Business Rules, Entities	Domain, Domain Aggregate	FFLIB, Force-DI	
Infrastructure Layer	Communicates with lower levels (i.e. SF Objects, etc.) Communicates with lower levels (i.e. Callouts) Handles Persistence	Repository, Factory, Selector	FFLIB, Force-DI	
Lower Levels	Salesforce Data/API, Logging, Configuration, Exception Handling, Security, Custom Metadata	DX Packages, AppExchange	ACCC, Force-DI	



Business to Technology

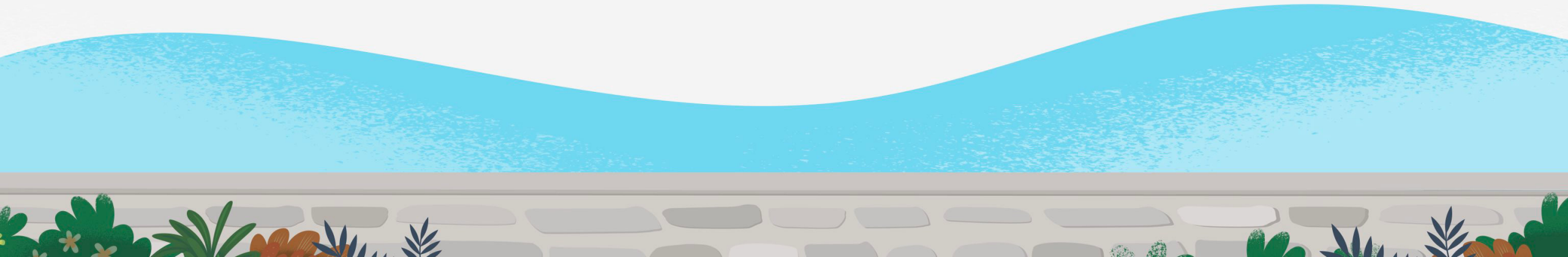
Business Capability with Core

Overview : Final Steps

Finally, we move from the Business (Top) to lower levels (Technology). The previous slides were a prelude to this section. If you were already familiar with the terms and concepts you could have skipped to this section.

Final Steps:

1. Map Business Capability (High-Level to Concepts)
2. Map the Capability to the Layers (for reference)
3. Validate Flow with other (known) functionality

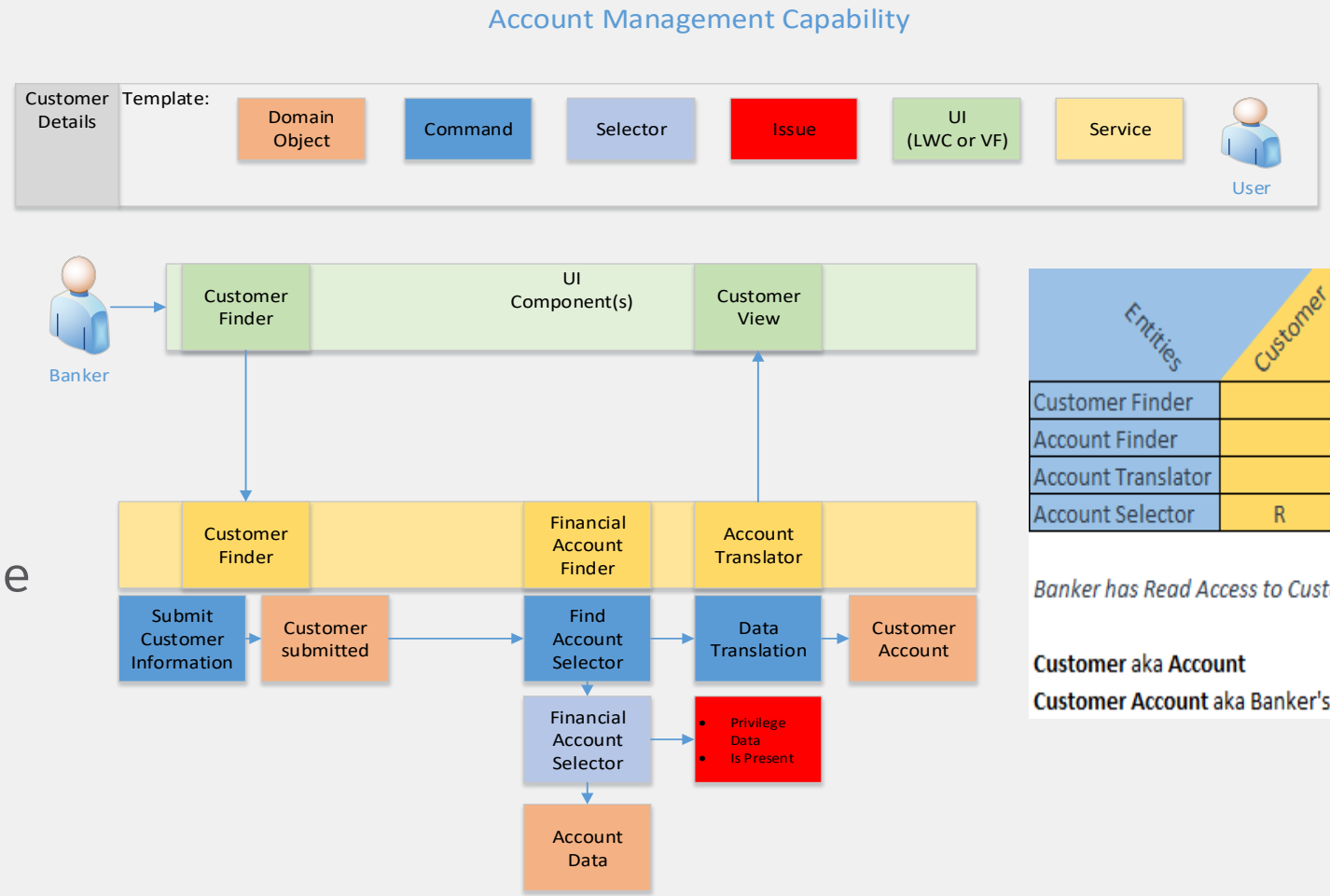


Business to Technology

Simple Account Management Capability : Customer Details

Using high-level components, we map the functionality to the high-level concepts (found in each layer) while also addressing CRUD access (i.e., Banker Persona)

The next slide maps these components to the appropriate layers.



Entities	Customer	Customer Account
Customer Finder		
Account Finder		
Account Translator		
Account Selector	R	

C	reate
R	ead
U	pdate
D	elete

Banker has Read Access to Customer (aka Account)

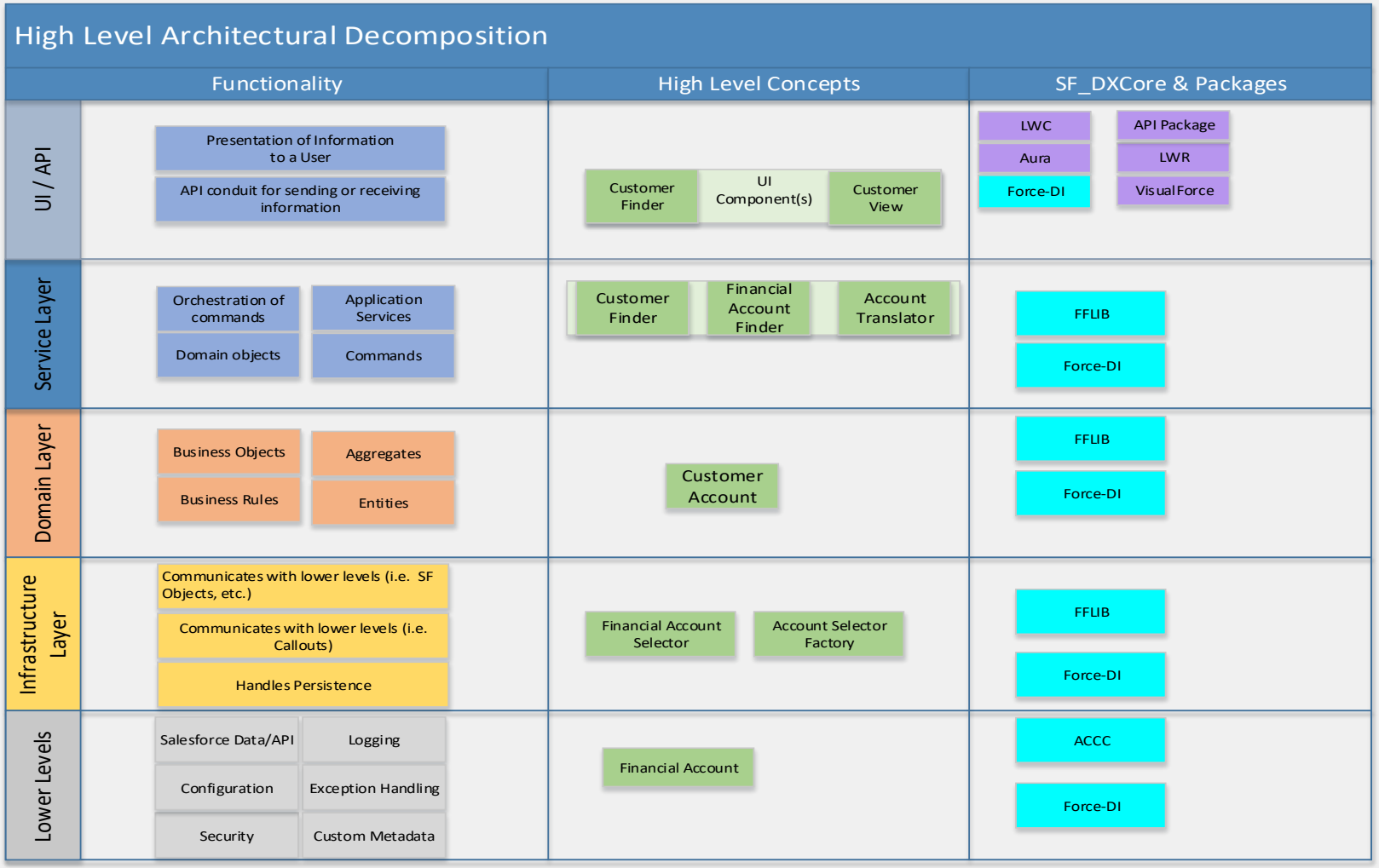
Customer aka **Account**

Customer Account aka Banker's View Model of Customer

Concepts Utilized in Layers

Simple Account Management Capability : Customer Details

This slide shows where the components used in the previous slide are mapped to their layers.

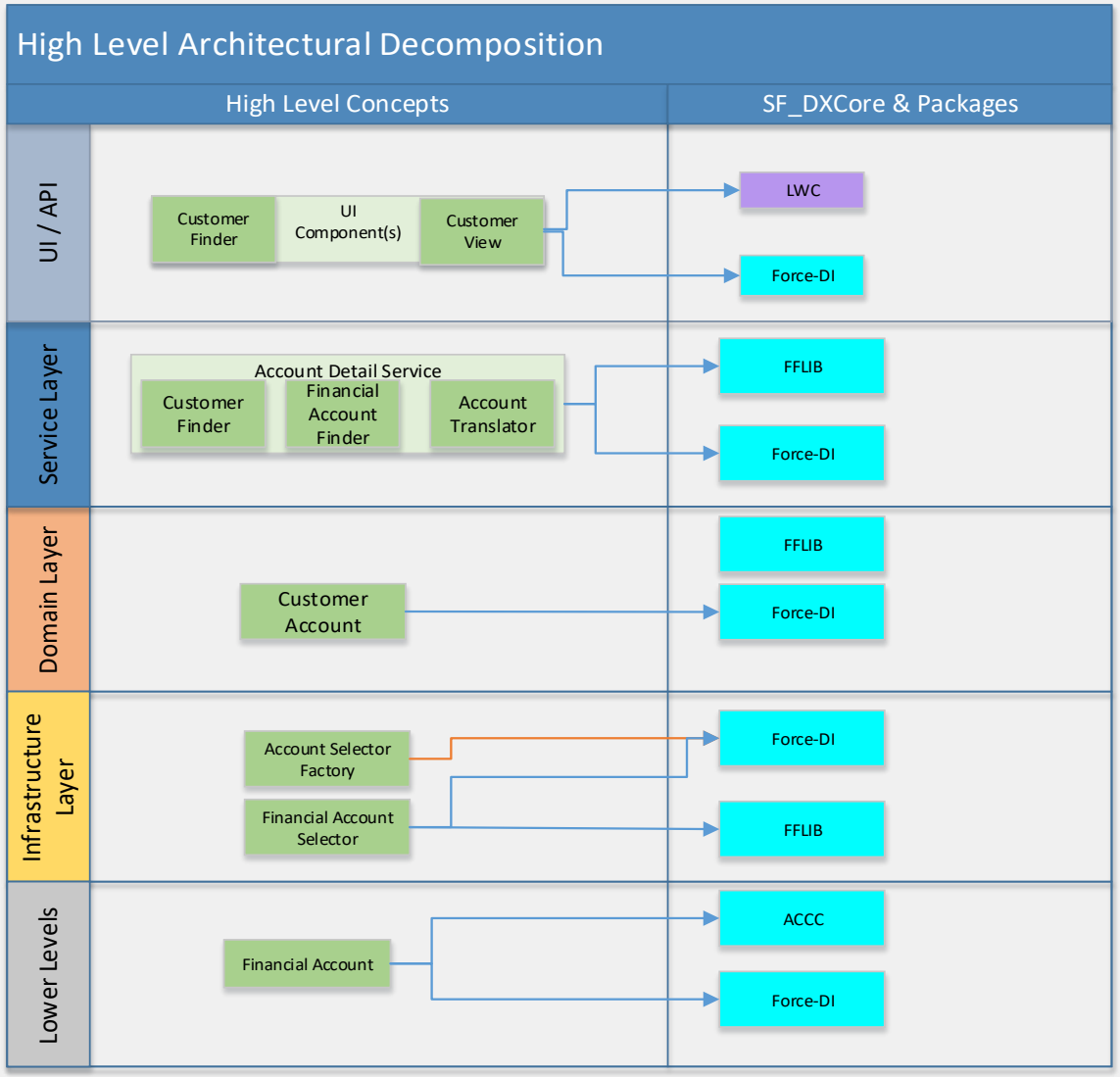


Concepts mapped to SF_DXCore

Simple Account Management Capability : Customer Details

This slide shows the Classes/Objects (Concepts) utilizing the different Frameworks within ***SF_DXCore.***

Please Note, there are currently no **LWC** components in the Package.



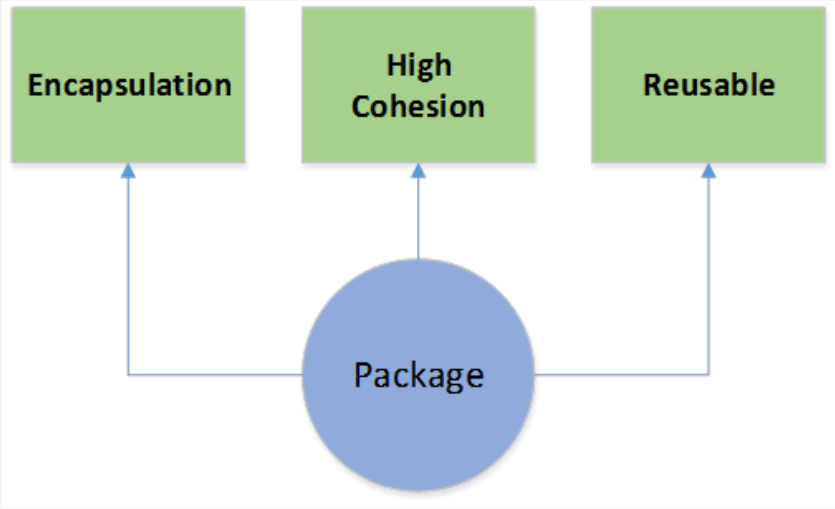
Summary

Using a common language with a bounded context (FSC) we can better understand our components relationships. Thus, by removing the ambiguity between a SME and Software Architect , they can have meaningful conversations that can lead to better packages (now or in the future).

We used layers to reduce the complexity and facilitated those layers with common (Core) components which reduce the clutter and increase reuse.

We showed how we can trace from Business Capabilities directly to the Technology.

Finally, by encapsulating components into *highly cohesive re-usable* parts *supporting a common language*, what emerges is a **Package**. In this case, ***SF_DXCore***.




Thank
You

Architecture Consideration

Concepts to help move toward packaging

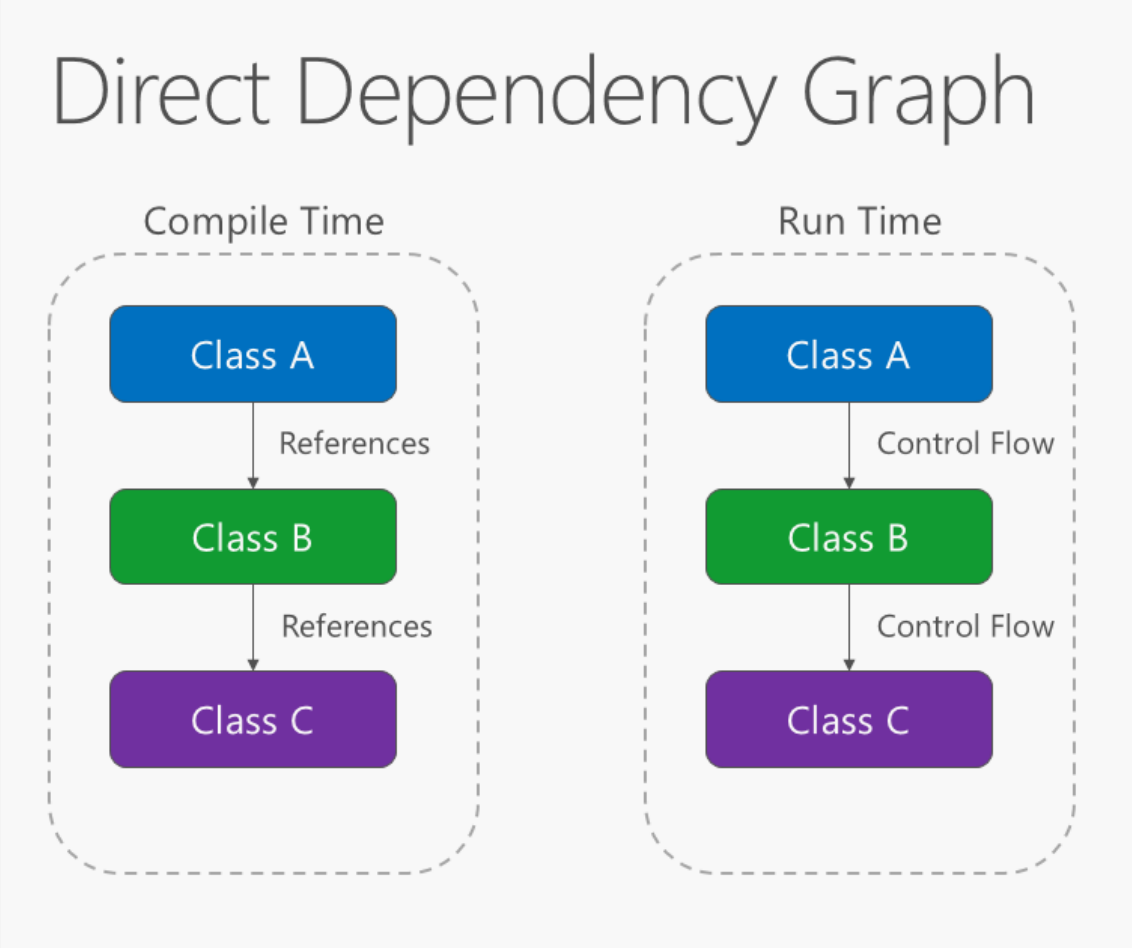
Dependency Inversion?

If coding, Why dependency inversion (excerpts from Microsoft Architectural Decomposition)?

The direction of dependency within your code should be in the direction of abstraction, not implementation details.

Most code are written such that compile-time dependency flows in the direction of runtime execution, producing a direct dependency graph.

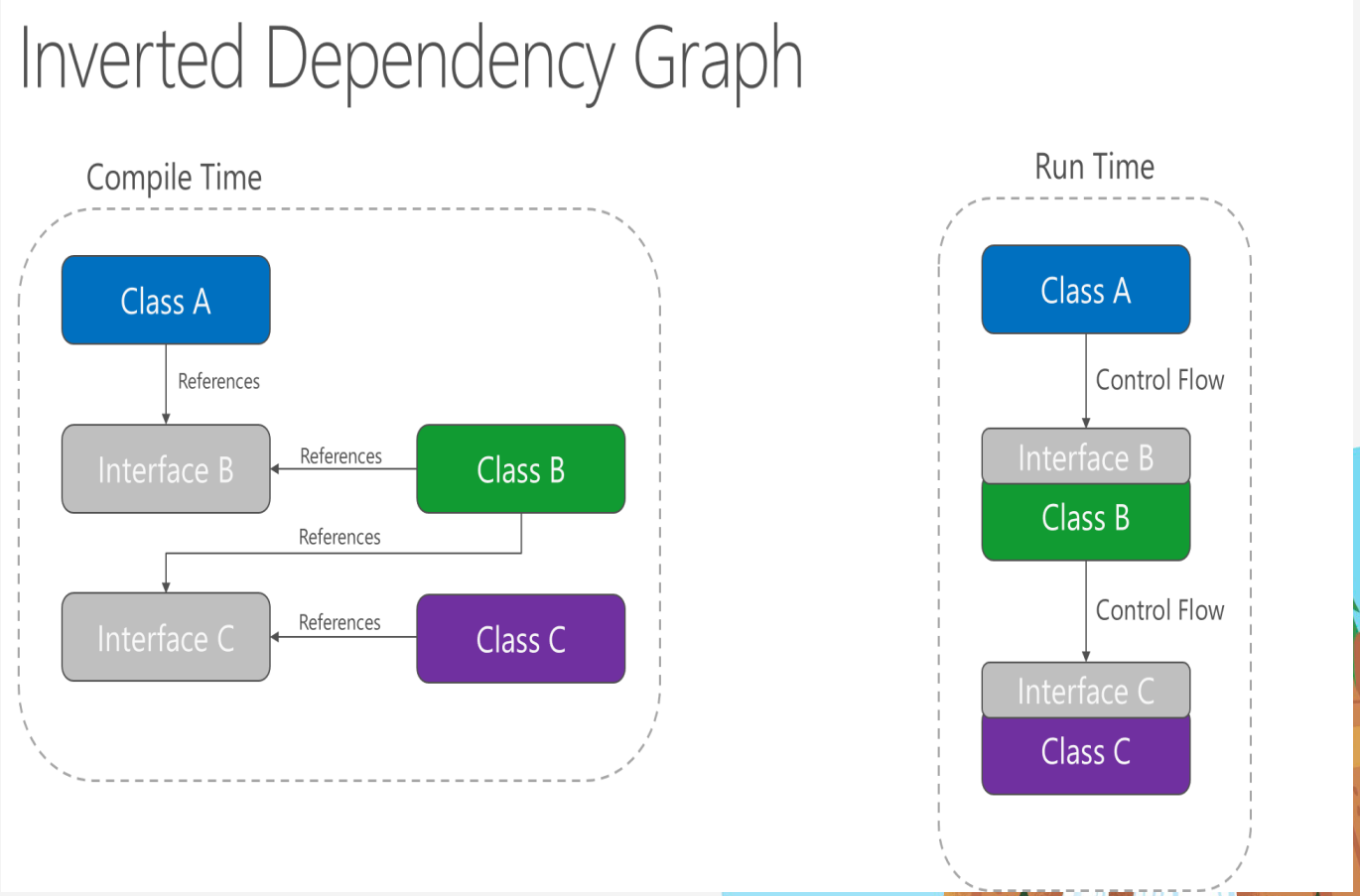
That is, if module A calls a function in module B, which calls a function in module C, then at compile time A will depend on B, which will depend on C



Why Dependency Inversion?

When packaging

Applying the dependency inversion principle allows A to call methods on an abstraction that B implements, making it possible for A to call B at runtime, but for B to depend on an interface controlled by A at compile time (thus, inverting the typical compile-time dependency). At run time, the flow of program execution remains unchanged, but the introduction of interfaces means that ***different implementations of these interfaces can easily be plugged in****.*



Why Dependency Inversion important for Package?

Dependency inversion *is a key part of building loosely coupled components, since implementation details can be written to depend on and implement higher-level abstractions, rather than the other way around.*

The resulting code components are ***more testable****,* ***modular (to package)****,* ***and maintainable*** *as a result. The practice of dependency injection is made possible by following the dependency inversion principle.*

Concepts used to move toward Packaging

Architectural Guidelines

- **Single Responsibility** - *Presentation responsibility should remain in the UI project, while data access responsibility should be kept within an infrastructure project. Business logic should be kept in the application core project, where it can be easily tested and can evolve independently from other responsibilities.*
- **Don’t Repeat Yourself (DRY)** - *The code should avoid specifying behavior related to a particular concept in **multiple places as this practice is a frequent source of errors**. At some point, **a change in requirements will require changing this behavior.** It's likely that at least one instance of the behavior will fail to be updated, and the system will behave inconsistently.*
- **Persistence Ignorance(PI) – PI** refers to types that need to be persisted, but whose code is unaffected by the choice of persistence technology. Such types in APEX are sometimes referred to as Data Access Objects (DAOs), because they do not need to inherit from a particular base class or implement a particular interface. Persistence ignorance is valuable because it allows the same business model to be persisted in multiple ways, offering additional flexibility to the application

Concepts used to move toward Packaging (continued)

Architectural Guidelines

Bounded contexts -- *are a central pattern in Domain-Driven Design. They provide a way of tackling complexity in large code-base or organizations by breaking it up into* ***separate conceptual modules****. Each conceptual module then represents a context that is separated from other contexts (hence, bounded), and can evolve independently. Each bounded context should ideally be free to choose its own names for concepts within it and should have exclusive access to its own persistence store.*

At a minimum, individual web applications should strive to be their own bounded context, with their own persistence store for their business model, rather than sharing a database with other applications. ***Communication between bounded contexts occurs through programmatic interfaces, rather than through a shared database****, which allows for business logic and events to take place in response to changes that take place. Bounded contexts map closely to packages, which also are ideally implemented as their own individual bounded contexts.*

High Level Concepts

Design Patterns

Commands– Encapsulate all information needed to perform an action or trigger an event. Design Pattern.

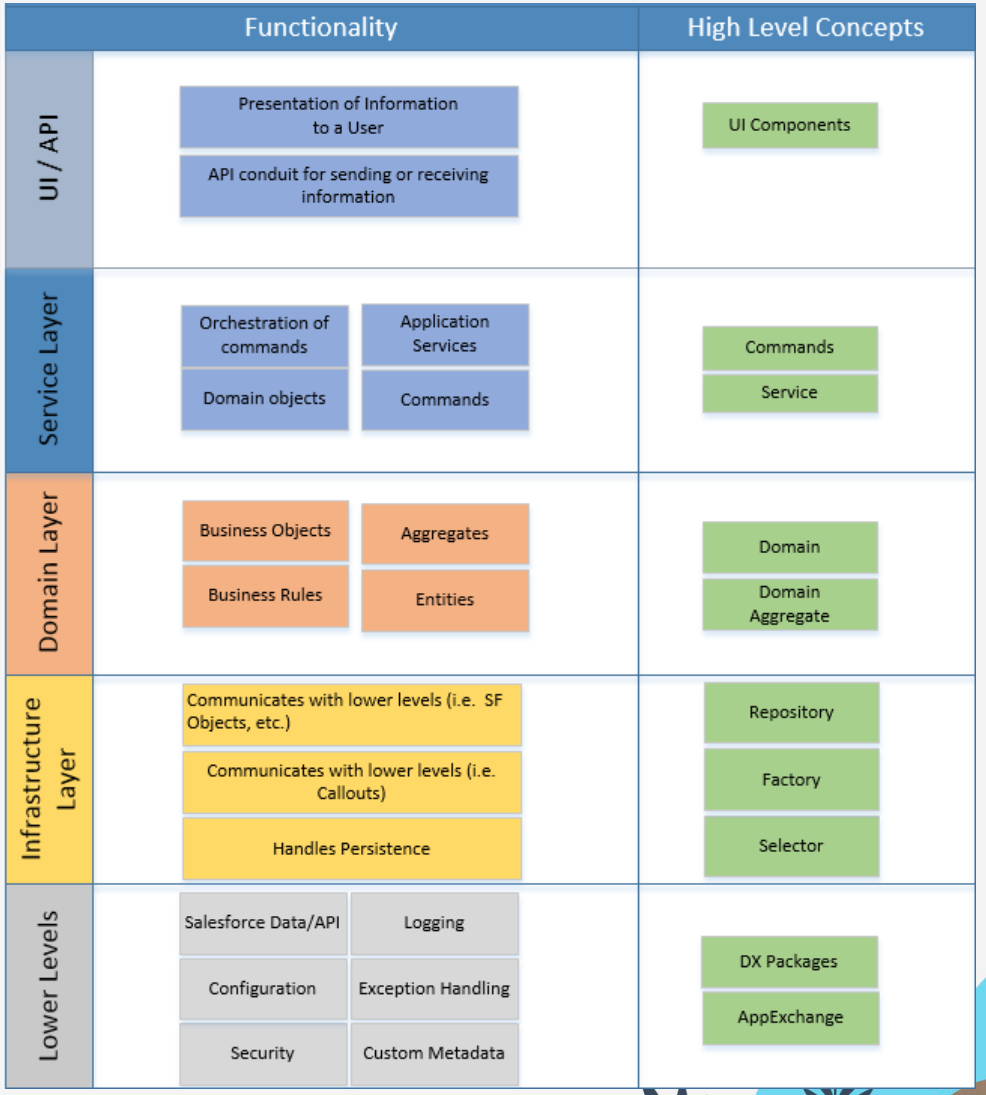
Domain – Domain Objects (Account, Contact, etc.)

Domain Aggregate – Combines multiple Domain Objects (i.e., Person Account)

Factory – Allows creating objects without having to specify the exact class of the object that will be created.

Repository - Abstracts the data layer and centralizes the handling of the domain objects.

Selector - Encapsulates logic responsible for querying information



Definitions

Domain Drive Design and Object Orient concepts

Domain – A sphere of knowledge or activity

Model – A system of abstractions representing selected aspects of a domain

Ubiquitous Language – A language structure around the domain model and used by all teams around a bounded context

Bounded Context -- A description of a boundary (typically a subsystem, or the work of a particular team) within which a particular model is defined and applicable.

Context -- The setting in which a word or statement appears that determines its meaning. Statements about a model can only be understood in a context

Design Patterns -- a general repeatable solution to a commonly occurring problem in **software** design

Domain Driven Design – follow this reference.