# Packaging – To Be or Not To Be?

When Salesforce introduced the concept of packaging many companies were excited by the concept but when it came to adoption there was confusion. In a non-Salesforce environment, the concept of packaging is not new. It's origins can go back to Parnas's seminal 1972 paper *On the Criteria to Be Used in Decomposing Systems into Modules*, where *high cohesion* within modules and *loose coupling* between modules is fundamental to modular design in software; the premise for packages. The way to the future is paved by work from the past.

Thus, in order to avoid pitfalls with packaging we look back to foundational work gleaned from the work of Robert Martin, Martin Fowler, David Parnas, et. al. to define the principles by which we package. When it comes to Salesforce, the metadata is a double edge sword and a proper package, if needed, can provide the right handle.

This paper outlines fundamental concepts of packaging so that we can better define our goals, categorization, design principles and use. *We do not want to package, just for the sake of packaging*!

Please note, this paper does not include Naming convention, shared services, on/off platform services, DevOps, etc. that can have an impact on packaging.

## Foundational Concepts

Package concept from Salesforce is new. However, we should have already defined your goals, roadmap and guiding principles for our Salesforce Org(s). These aspects will help drive how to adopt, *or not*, packages.

Let's work backwards and define what is a package and map our goals and guidelines to packaging.

### What is a Package?

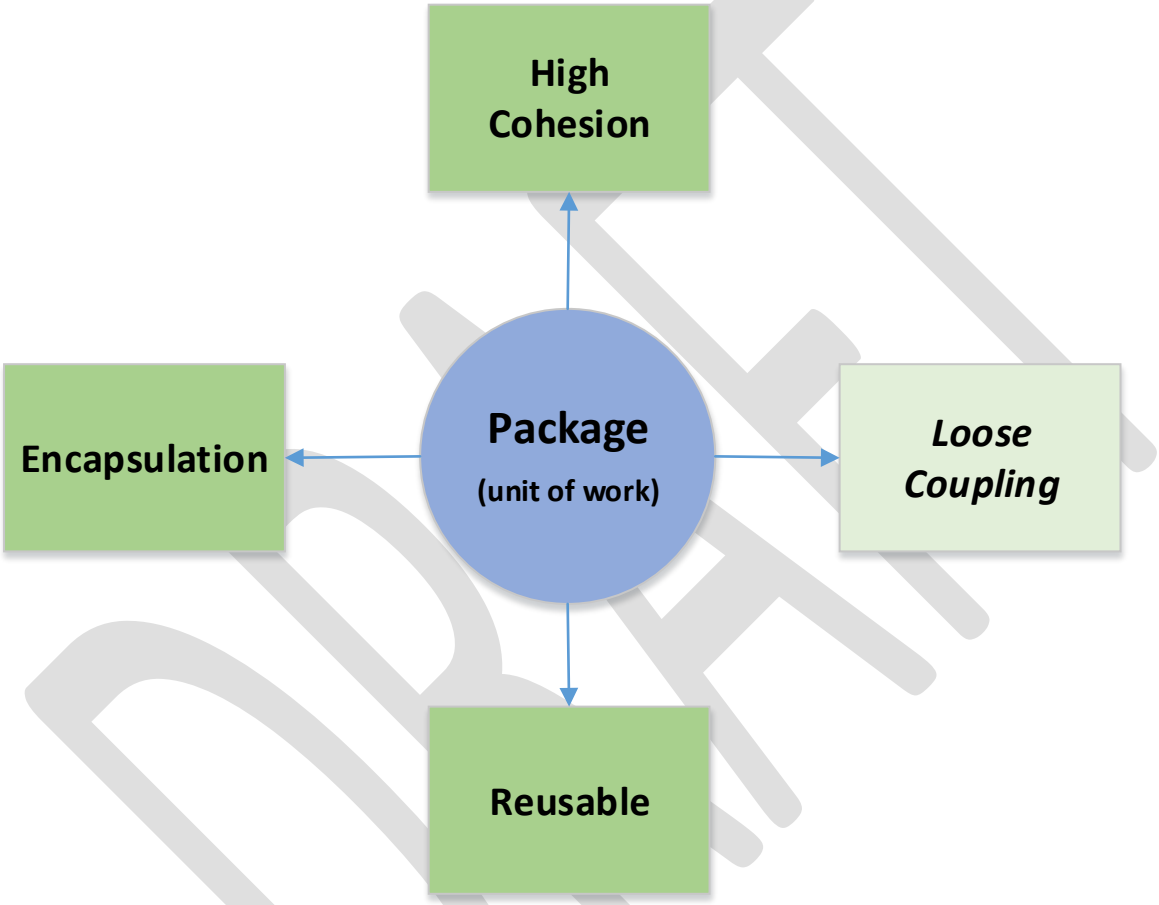
Package, much like libraries found in other languages (i.e. Java, C#, C++, etc.) maintained in a (GIT) repository have the following attributes:

- Modular,
- Deployable,
- Shared,
- Low volatility / Stable [*Cost efficient*]
- Reusable

A package supports these attributes with, at least, three pillars:

| Pillar | Description |
|---|---|
| **Encapsulation** | Defines the bundling of data with the methods that operate on said data. Encapsulation is used to hide the values or state of a structured data object inside a package |
| **High Cohesion** | Defines how closely all the components in a package support a common purpose. |

| Pillar | Description |
|---|---|
| **Reusable** | The package can be reused without consequences of the environment; morphing to each environment. If adopting to multiple Orgs, may need Loose coupling |
| ***Loose Coupling*** | Makes code extensible, and extensibility makes it maintainable. DI is nothing more than a technique that enables loose coupling. *[Note, without code, this pillar is optional]* |

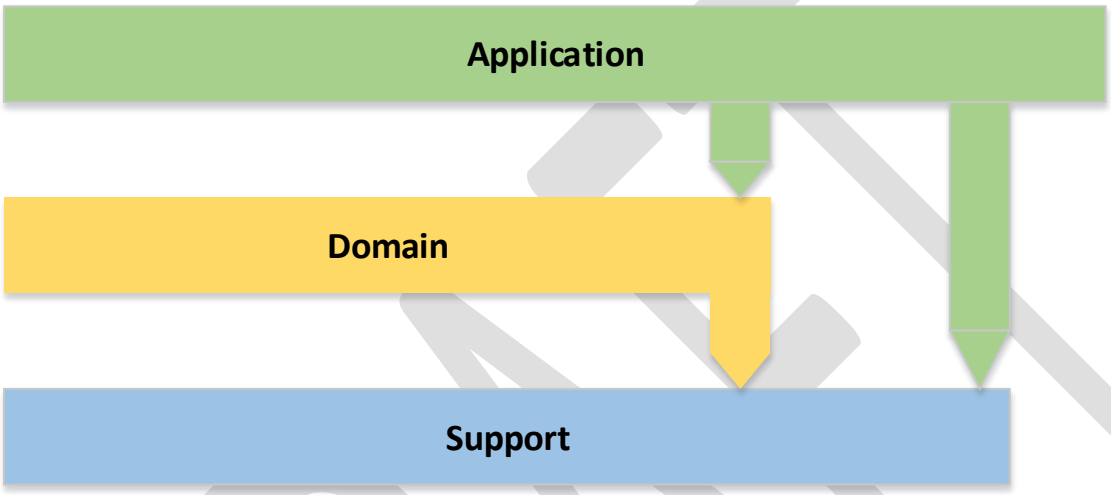



## Package Categorization

Understanding the core pillars of a DX package, we can now begin to classify. Packages fall into three categories, *Application*, *Domain* and *Support*. These base concepts follow the same paradigm found in other (non-Salesforce) Frameworks:

| Package Categories | |
|---|---|
| **Application Package** | Encapsulate expertise in a program such as Leads and Referrals, Customer Management, etc. These high-level packages may utilize both ***Domain*** and ***Support*** packages. (see easy-lwc) |

| | |
|---|---|
| **Domain Package** | Encapsulate expertise to a domain. These packages can provide a vertical or horizontal slice, such as, lead management, opportunity management, etc. Domain packages may consist of just the schema (layout, validation, etc.) |
| **Support Package** | Encapsulate expertise to specific functionality, such as communication, data manipulation, data access, etc. These support packages help support **Domain** and **Application** packages. (see sf-trigger-handler, force-di). |




The above are the high-level categorization. You can have sub-categories, such as,

- Domain -- Schema
- Support – Mapping/Translation,
- Support – String Manipulation
- Support – SObject (i.e. describe, Accessibility, etc.)
- Application – Configuration/Environment

The important point is to create a catalog of functionality. Over time, this will pay dividends so as to avoid DRY principle.

## What are our Goals?

Now that we know what to expect from a package. We need to decide, based on our goals, is there an alignment. If so, can we measure our success. For example, if business goals state,

- Faster time to Market,
- Developer Efficiency,
- Federated Development Model,
- Reusable Artifacts,
- Extensible Business Capabilities

Then do the above goals align to a packaging. How? We can map those goals to our definition of a Package.

| Goals | Package Pillar(s) |
|---|---|
| **Developer Efficiency** | Encapsulation, Reusable, Loose Coupling |
| **Federated Model** | Encapsulation, Reusable, High Cohesion |
| **Reusable Artifacts** | Encapsulation, Reusable, High Cohesion |
| **Extensible Business Capabilities** | Encapsulation, Reusable, High Cohesion, Loose Coupling |

Note, ***Faster time to Market*** goal, was not mapped to a Package Definition. *Why*? Let's first address our guidelines before we tackle that question.

## What are our Guidelines?

All Salesforce Orgs should have a Customization Strategy with Guiding Principles to realize your goals. The Guiding Principles will help make decisions at a higher level, i.e. Business stakeholders.

Every company which uses Salesforce will have their own *Guiding Principles* based on:

- Amount of customization via code or click,
- Technical, or non-technical staff
- Etc.

We will define two sets of guiding principles; one for code customization and one for non-code customization. But, all guiding principles will have, at the number one goal, ***Clicks, Not Code*** principle.

### Code Customization Guidelines

- ***[Top] Clicks, Not Code,***
- *Define **WHAT** you are doing first; Not **HOW***
- *Use common functionality from packages, including AppExchange*

*If we must customize …*

- *Do not depend on concreteness, if possible*
- *Depend on abstraction (design contracts)*
- *Do not be rigid, as change (volatility) is inevitable*
- *Design for reuse*
- *If coding, used SOLID Design Features.*
- *When packaging, ensure, at least, the three Pillars are present*

### Non-Code Customization Guidelines

- ***[Top] Clicks, Not Code,***
- *Define **WHAT** you are doing first; Not **HOW***
- *Buy before Build, i.e. Use common functionality from packages, including AppExchange*

## Should We Package?

Remember we said we would address the ***Faster time to Market*** question. Now that we have defined all aspects, let's tackle that question.

Our top guiding principle is ***Clicks, Not Code***. If we are not heavily customizing our Salesforce Org with (Apex) Code, do we need to package? Let's look at some other considerations:

- Do we have developers that can handle the complexity within our Salesforce Org?
- Are we comfortable with VCS, i.e. Git, Git-Flow or Trunk-base?
- Do ***Changesets*** and Salesforce CLI (or *Copado*, *AutoRabit*, etc.) work without issues? Is that enough?
- Out-Of-The-Box functionality supports all my needs, Do I need heavy Customization?
- Do we have the technical ability, currently, to take on packaging complexity?

Using our goals and guidelines we can take into consideration our technical and people capabilities to drive the toward packaging or not without assuming addition risks.

If you can you answer the question, based on goals, guidelines, and people, ***does packaging make us Better, Faster and Economical?*** Then your path to *adopt*, or *not to adopt,* can be realized.

Of course, with the goals we should be able to measure success (KPIs):

- Time to Market (# days),
- Sprint Velocity Increase (over time [i.e. 6 months]),
- Number of ***net*** new components vs Past Sprints net new Components,
- Number of Packages utilized / Reuse / Sprint
- Number of Packages modified (not extended) / Sprint

## Level of Packaging?

The level of Packaging can be dictated by the Pillars of Packaging. Recall, the minimum pillars were *encapsulation*, *high cohesion* and *reusable*. These pillars are used to define **local** packages for a Salesforce Org. However, if you have more than one Salesforce Org and have done code customization, you may want to package for a **global** concern; thus, one needs to add the *fourth* pillar, *loose coupling*.

### Local Package

***Local** package* may not take extensibility into consideration nor running environment. There may be code customization in the local package, but we cannot extend the behavior. We are force to utilize the behavior defined.

This does not mean it is a bad design. Many factors may have led to this design, such as, time, money, expertise, need, etc. For example, one could package only new fields and layouts for Account, Contact, etc. Here we may have more of a tailored view model aligned to a business need (of Account, Contact, etc.). This I would categorize more of a **Global**.**Domain**.**Schema** Package.

### Global Package

***Global** package* is agnostic to its running environment. Global packages limit their dependencies on core/known functionality in a Salesforce Org. If there is code customization, and we can extend the behavior via DI (Dependency Injection), our package is *loosely coupled*, the fourth pillar. The fourth pillar requires more thought around the architecture and dependencies.

## Local and Global Package Attributes

Local and Global Packages can have the following attributes,

| Attribute | Type | Description |
|---|---|---|
| **Extensible** | Global | Open for extension, closed for modification. Supports Dependency Injection (DI) |
| **Stability** | Local / Global | The amount of work required to change is low and often can have many other packages which depend upon it. |
| **Low Volatility** | Local / Global | The package changes rarely. It does not mean there are no additions, it means that which was encapsulated can be extended; or, there is abstracted areas which allow consumers (those that depend) to extend. |
| **Environment Agnostic** | Global | Does not Depend upon the environment (Health, Financial, IOT, etc. Cloud). Encapsulates the volatility via abstraction or augmentable behavior. |

## Package Principles

Here are some packaging principles packaging to keep in mind as your progress/

- ***Package Closure*** – Items that change together, live together in the same package**.**
- ***Package Release Reuse*** – If a package is commonly adopted/reused; then the unit of release is also the unit of reuse.
- ***Package Cohesion* –** A package should not cross layers but remain true to a cohesive ([**SRP**] Single Responsibility Principle) functionality. For example, if a package contains both UI (LWC/VFP) and CRUD capabilities, then those depending upon only the CRUD functionality will be affected by UI changes, unnecessarily.
- ***Package Stability* –** Package does not change much, and if there are changes, it is via extensions.

## Apex Code Customization

If there is a need for Apex Code Customization, please remember Apex is an Object-Oriented Language that one should avail oneself of good Design Principles, such as SOLID.