Classification of a Package

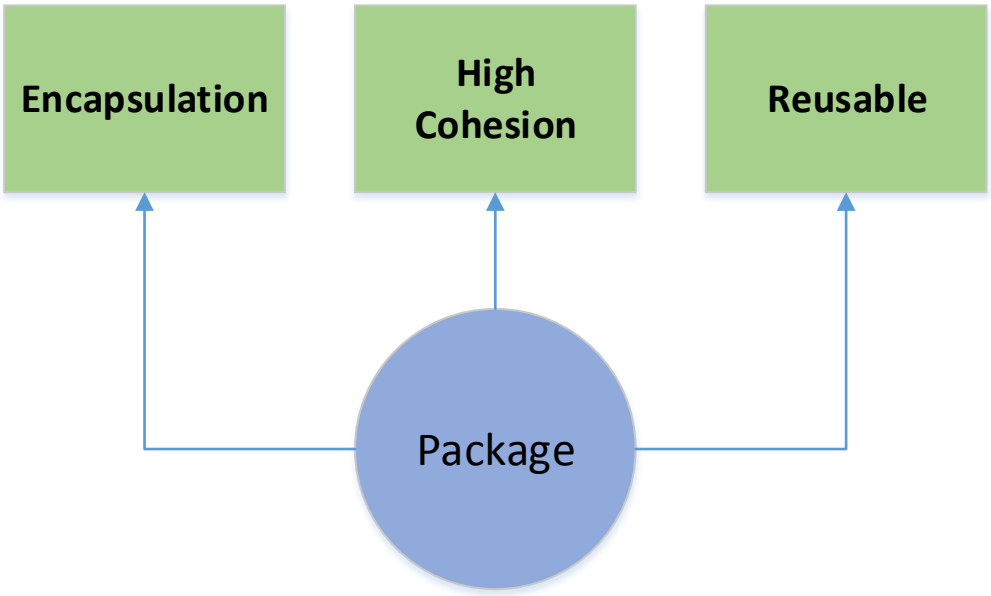
Traversing the concept of DX Packaging can be a daunting task. In our exuberance of adoption, we cannot lose sight of purpose and value. Packaging for sake of packaging is a path toward disaster; just a road trip to nowhere. You wind up with a lot of cost and lost time.

Before you go down the path of DX Packaging first understand the purpose and value proposition. There already exists a document, ***Salesforce Packages: Pattern Strategy***, which delves further into the value of slicing (*Vertical* vs *Horizontal*) of a Salesforce Org.

Instead, this document attempts to provide classification of a Package, the Categories of a Package and the principles. This understanding will allow you to identify the package, categorize so that you can create a roadmap to cost effectiveness with guiding principles.

Pillars of a Package

There exist three pillars of a Package which include *Encapsulation*, *High Cohesion* and *Reusability*. These pillars define a package as an entity not reliant on the environment/Org it exists in; except for standard Salesforce components.

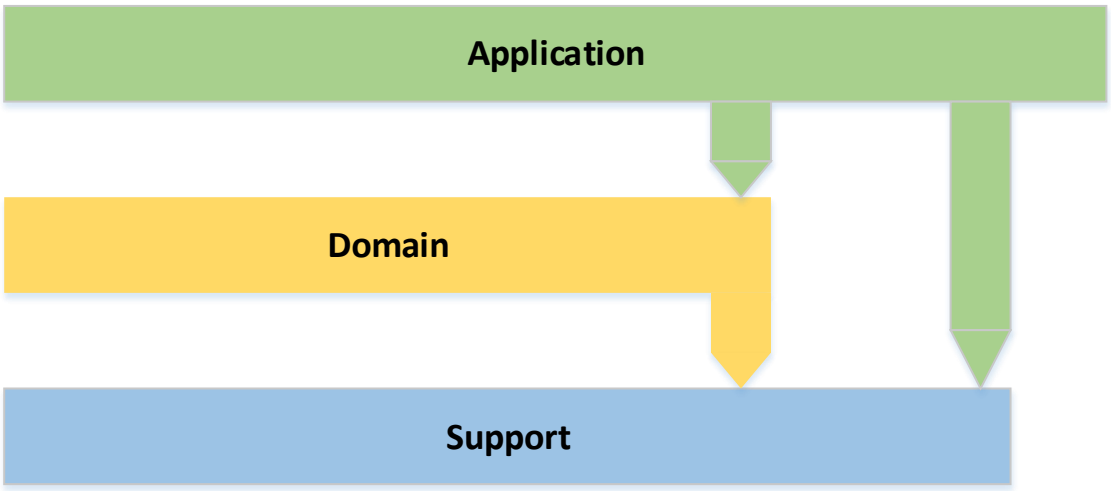


Pillars of a Package	
Encapsulation	Defines the bundling of data with the methods that operate on said data. Encapsulation is used to hide the values or state of a structured data object inside a package
High Cohesion	Defines how closely all the components in a package support a common purpose.
Reusable	The package can be reused without consequences of the environment; morphing to each environment. Please note, if a package has a dependency, then one need to adhere to the guiding principles.

Categories of a Package

Understanding the core pillars of a package we can now begin to classify packages. Packages fall into three categories, *Application*, *Domain* and *Support*. These base concepts follow the same paradigm around 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
Domain Package	Encapsulate expertise to a domain. These packages can provide a horizontal slice, such as, trigger, data management, etc.
Support Package	Encapsulate expertise to functionality such as communication, data manipulation, user management, etc. These support packages help facilitate Domain and Application packages.



Common Principles for Packaging

For packaging we can adopt, but vary slightly, [packaging as defined by Bob Martin] in a non-Salesforce environment. The core concepts are very familiar to many that have developed in a non-Salesforce Object Oriented environment. These concepts transcend those environments and can be applied to Salesforce DX as well; especially, with Apex. What follows are excerpts from Bob Martin (slightly modified). Please note, the document does not bring in all the information from the various reference. For more in depth information, please see the link provided above.

The first three package principles are about package ***cohesion***, they tell us what to put inside packages:

REP	The Release Reuse Equivalency Principle	*The granule of **reuse** is the granule of **release**.*
CCP	The Common Closure Principle	*Components that change together are packaged together.*
CRP	The Common Reuse Principle	*Components that are used together are packaged together.*

The last three principles are about the ***couplings*** between packages and talk about metrics that evaluate the package structure of a system.

ADP	The Acyclic Dependencies Principle	*The dependency graph of packages must have no cycles.*
SDP	The Stable Dependencies Principle	*Depend in the direction of stability.*
SAP	The Stable Abstractions Principle	*Abstractness increases with stability. (code-related)*

Summary

Principles in other discipline are just as applicable to Salesforce as they are to non-Salesforce Platform. The problem domain was well established before Salesforce, and we more recognize they apply now more than ever.

The Anatomy of a Package is important to understand; however, it is also important to apply principles to packages to obviate issues we would encounter later.

Cohesion

The Release Reuse Equivalency Principle (REP)

THE GRANULE OF REUSE IS THE GRANULE OF RELEASE. ONLY COMPONENTS THAT ARE RELEASED THROUGH A TRACKING SYSTEM CAN BE EFFECTIVELY REUSED. THIS GRANULE IS THE PACKAGE.

Reusability is one of the most oft claimed goals of OOD. But what is reuse? Is it reuse if I snatch a bunch of code/components[1] from one source and textually insert it into another? It is reuse if I steal a module from someone else and bring it into my own source? I don't think so. The above are examples of code/component copying; and it comes with a serious disadvantage: you own the code/component you copy! If it doesn't work in your environment, you must change it. If there are bugs in it, you must fix them. If the original author finds some bugs in the code/component and fixes them, you must find this out, and you must figure out how to make the changes in your own copy. Eventually the

[1] Code and Component represents elements in a Salesforce Org (i.e. Apex, Aura, LWC, etc.)

code/component you copied diverges so much from the original that it can hardly be recognized. The code/component is yours. While code/component copying can make it easier to do some initial development; it does not help very much with the most expensive phase of the software lifecycle, ***maintenance***.

I prefer to define reuse as follows. I reuse code if, and only if, I never need to look at the source code (other than the public portions of header files). I need only link with static libraries or include dynamic libraries. Whenever these libraries are fixed or enhanced, I receive a new version which I can then integrate into my system when opportunity allows. That is, I expect the code I am reusing to be treated like a product. It is not maintained by me. It is not distributed by me. I am the customer, and the author, or some other entity, is responsible for maintaining it. When the libraries that I am reusing are changed by the author, I need to be notified. Moreover, I may decide to use the old version of the library for a time. Such a decision will be based upon whether the changes made are important to me, and when I can fit the integration into my schedule. Therefore, I will need the author to make regular releases of the library. I will also need the author to be able to identify these releases with release numbers or names of some sort.

Thus, I can reuse nothing that is not also released. Moreover, when I reuse something in a released library, I am in effect a client of the entire library. Whether the changes affect me or not, I will have to integrate with each new version of the library when it comes out, so that I can take advantage of later enhancements and fixes. And so, the REP states that the granule of reuse can be no smaller than the granule of release. Anything that we reuse must also be released. Clearly, packages are a candidate for a releasable entity. It might be possible to release and track classes, but there are so many classes in a typical application that this would almost certainly overwhelm the release tracking system. We need some larger scale entity to act as the granule of release; and the package seems to fit this need rather well.

The Common Closure Principle (CCP)

> ***THE COMPONENTS IN A PACKAGE ARE REUSED TOGETHER. IF YOU REUSE ONE OF THE COMPONENTS IN A PACKAGE, YOU REUSE THEM ALL.***

This principle helps us to decide which components should be placed into a package. It states that components [2]that tend to be reused together belong in the same package.

Components are seldom reused in isolation. Generally reusable classes/components collaborate with other classes/components that are part of the reusable abstraction. The **CRP** states that these classes/components belong together in the same package.

A simple example might be a trigger handler interface and its associated base classes. These classes are reused together because they are tightly coupled to each other. Thus, they ought to be in the same package.

[2] Component, or Class represents elements in a Salesforce Org (i.e. Apex Class, Aura, LWC, etc.)

The reason that they belong together is that when an engineer decides to use a package a *dependency is created upon the whole package*. From then on, whether the engineer is using all the classes in the package or not, every time that package is released, the *modules/application that use it must be revalidated and rereleased*. If a package is being released because of changes to a class that I don't care about, then I will not be very happy about having to revalidate my application. Moreover, it is common for packages to have physical representations as 2GPs. If a 2GP is released because of a change to a class that I don't care about, I still must redistribute that new 2GP and revalidate that the application works with it.

Thus, I want to make sure that when I depend upon a package, I depend upon every component in that package. Otherwise I will be revalidating and redistributing more than is necessary and wasting lots of effort.

The Common Reuse Principle

> **THE COMPONENTS IN A PACKAGE SHOULD BE CLOSED TOGETHER AGAINST THE SAME KINDS OF CHANGES. A CHANGE THAT AFFECTS A PACKAGE AFFECTS ALL THE COMPONENTS IN THAT PACKAGE.**

More important than reusability, is maintainability. If the components in an application must change, where would you like those changes to occur: all in one package, or distributed through many packages? It seems clear that we would rather see the changes focused into a single package rather than have to dig through a whole bunch of packages and change them all. That way we need only release the one changed package. Other packages that don't depend upon the changed package do not need to be revalidated or rereleased.

The **CCP** is an attempt to gather in one place all the components that are likely to change for the same reasons. For example, if two classes, are so tightly bound, either physically or conceptually, such that they almost always change together; then they belong in the same package. This minimizes the workload related to releasing, revalidating, and redistributing the software.

This principle is closely associated with the *Open Closed Principle* (**OCP**). For it is "*closure*" in the **OCP** sense of the word that this principle is dealing with. The **OCP** states that classes should be closed for modification but open for extension. As we learned in the article that described the **OCP**, 100% closure is not attainable. Closure must be strategic. We design our systems such that they are closed to the most likely kinds of changes that we foresee.

CCP amplifies this by grouping together classes components cannot be closed against certain types of changes into the same packages. Thus, when a change in requirements comes along; that change has a good chance of being restricted to a minimal number of packages.

Coupling

The Acyclic Dependencies Principle (ADP)

> **THE DEPENDENCY STRUCTURE BETWEEN PACKAGES MUST BE A DIRECTED ACYCLIC GRAPH (DAG). THAT IS, THERE MUST BE NO CYCLES IN THE DEPENDENCY STRUCTURE.**

Have you ever worked all day, gotten some stuff working and then gone home; only to arrive the next morning at to find that your stuff no longer works? Why doesn't it work? Because somebody stayed later than you! I call this: "*the morning after syndrome*".

The "*morning after syndrome*" occurs in development environments where many developers are modifying the same source files. In relatively small projects with just a few developers, it isn't too big a problem. But as the size of the project and the development team grows, the mornings after can get pretty nightmarish. It is not uncommon for weeks to go by without being able to build a stable version of the project. Instead, everyone keeps on changing and changing their code trying to make it work with the last changes that someone else made.

The solution to this problem is to partition the development environment into releasable packages. The packages become units of work which are the responsibility of an engineer, or a team of engineers. When the responsible engineers get a package working, they release it for use by the other teams. They give it a release number and move it into a directory for other teams to use. They then continue to modify their package in their own private areas. Everyone else uses the released version.

As new releases of a package are made, other teams can decide whether or not to immediately adopt the new release. If they decide not to, they simply continue using the old release. Once they decide that they are ready, they begin to use the new release.

Thus, none of the teams are at the mercy of the others. Changes made to one package do not need to have an immediate effect on other teams. Each team can decide for itself when to adapt its packages to new releases of the packages they use.

This is a very simple and rational process. And it is widely used. However, to make it work you must manage the dependency structure of the packages. There can be no cycles. If there are cycles in the dependency structure, then the "morning after syndrome" cannot be avoided. I'll explain this further, but first I need to present the graphical tools that the UML 0.9 uses to depict the dependency structures of packages.

Packages depend upon one another. Specifically, a class in one package may #include the header file of a class in a different package. This can be depicted on a class diagram as a dependency relationship between packages (See *Figure 1 Dependency Relationship*).

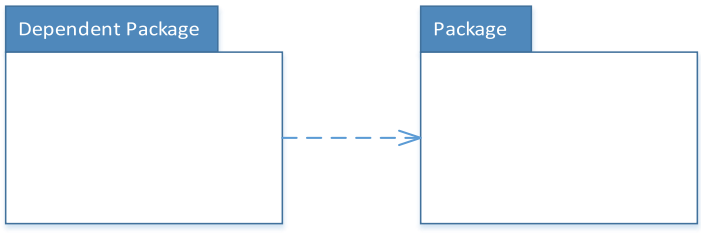


Figure 1 Dependency Relationship

For DX Packages we will use UML, which shows packages as "*tabbed folders*". Dependency relationships are dashed arrows. The arrows point in the direction of the dependency. That is, the arrowhead is placed next to the package that is being depended upon. In Packaging terms, there is a package dependency specified in the *sfdx-project.json*. And, for example, a class may reference another class within the dependent package being depended upon.

Consider the package diagram in Figure 2. Here we see a structure of packages assembled used in an (contrived) application. The function of this application is unimportant for the purpose of this example. What is important is the dependency structure of the packages. Notice how this structure is a graph. The packages are the nodes, and the dependency relationships are the edges. Notice also that the dependency relationships have direction. Thus, this structure is a *directed graph*.

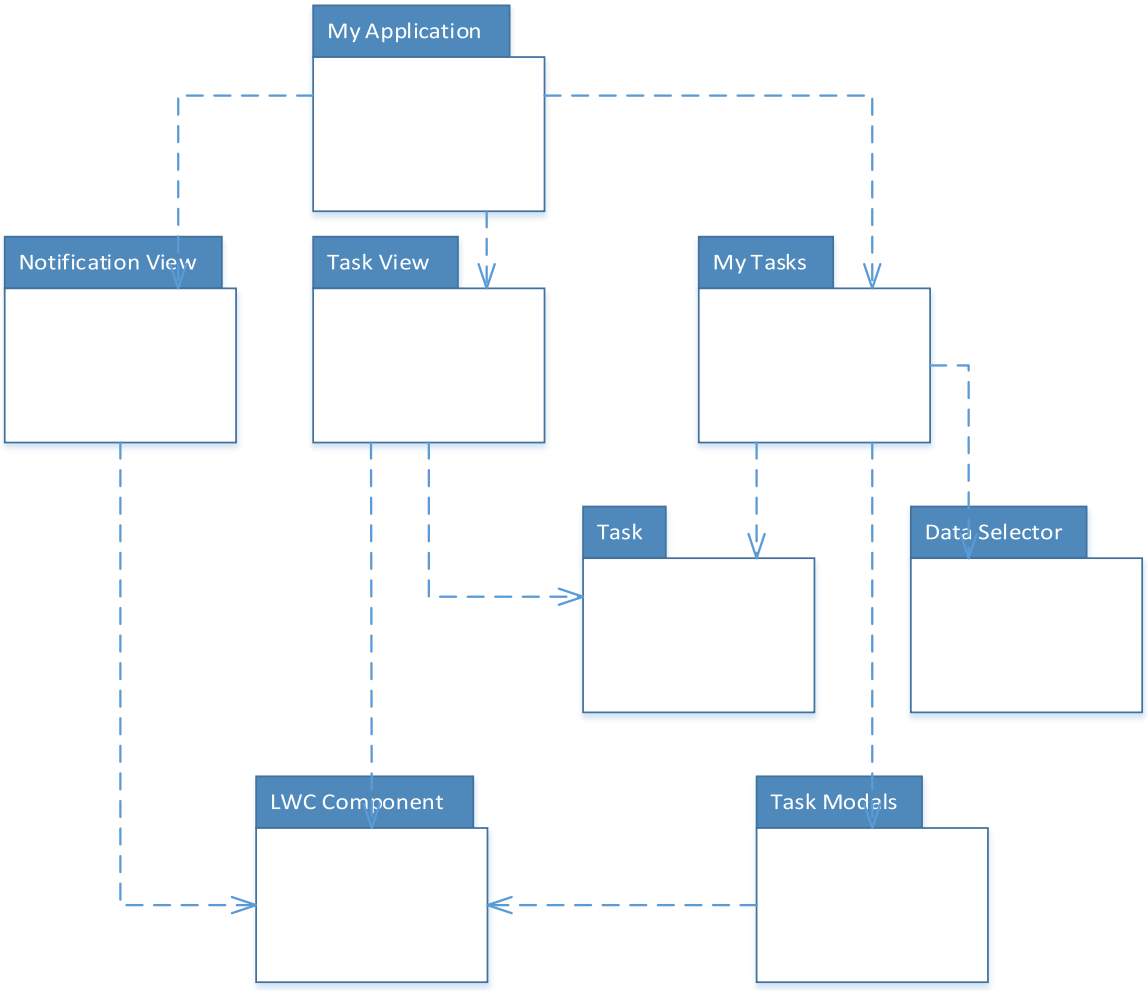


Figure 2 Package Diagram without cycles

Now notice one more thing. Regardless of which package you start with; it is impossible to follow the dependency relationships and wind up back at that package. This structure has no cycles. It is a **directed acyclic graph** (DAG).

Now, notice what happens when the team responsible for *TaskModals* makes a new release. It is easy to find out who is affected by this release; you just follow the dependency arrows backwards. Thus, *MyTasks* and *MyApplication* are both going to be affected. The teams responsible for those packages will have to decide when they should integrate with the new release of *TaskModals*.

Notice also that when *TaskModals* is released it has utterly no affect upon many of the other packages in the system. They don't know about *TaskModals*; and they don't care when it changes. This is nice. It means that the impact of releasing *TaskModals* is relatively small.

When the engineers responsible for the *TaskModals* package would like to run a unit test of their package, all they need do is compile and depend their version of *TaskModals* with the version of the *LWCComponent* package that they are currently using. None of the other packages in the system need be involved. This is nice, it means that the engineers responsible for *TaskModals* have relatively little work to do to set up a unit test; and that there are relatively few variables for them to consider.

When it is time to release the whole system; it is done from the bottom up. First the *LWCComponent* package is deployed/compiled, tested, and released. Then *NotificaitonView* and *TaskModals*. These are followed by *Task*, and then *TaskView* and *DataSelector*. *MyTasks* is next; and finally, *MyApplication*. This process is very clear and easy to deal with. We know how to build the system because we understand the dependencies between its parts.

Let us say that there is a new requirement forces us to change one of the classes in *TaskModals* such that it creates a dependency on *MyApplication*. This creates a dependency cycle as shown in Figure 3

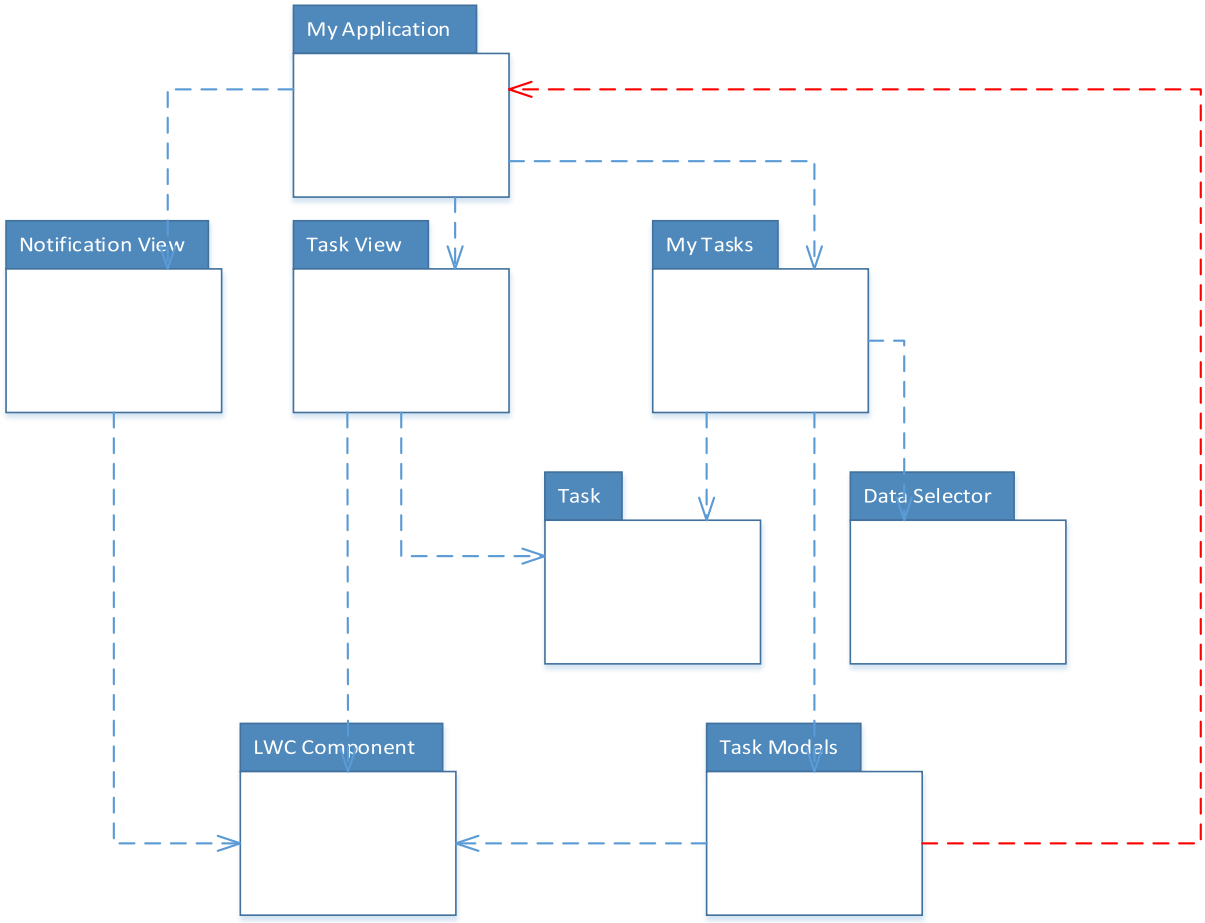


Figure 3 Package Diagram with cycles

This cycle creates some immediate problems. For example, the engineers responsible for the *MyTasks* package know that in order to release, they must be compatible with *Task*, *TaskModal*, *DataSelector* and *LWCComponent*. However, with the cycle in place, they must now also be compatible with *MyApplication*, *TaskView* and *NotifictionView;* that is, *MyTasks* now depends upon every other package in the system. This makes *MyTasks* very difficult to release. *TaskModals* suffers the same fate. In fact,

the cycle has had the effect that *MyApplication*, *MyTasks*, and *TaskModals* must always be released at the same time. They have, in effect, become one large package. And all the engineers who are working in any of those packages will experience *"the morning after syndrome"* once again. They will be stepping all over one another since they must all be using exactly the same release of each other.

But this is just the tip of the trouble. Consider what happens when we want to unit test the *TaskModals* package. We find that we must link in every other package in the system; including the *DataSelector* package. This means that we bring in all packages into an Org (Scratch Org) just to unit test *TaskModals*. This is intolerable.

If you have ever wondered why you have to depend on so many different packages, and so much of everybody else's stuff, just to run a simple unit test of one of your classes, it is probably because there are cycles in the dependency graph. Such cycles make it very difficult to isolate modules. Unit testing and releasing become very difficult and error prone. And load and unit test times grow geometrically with the number of modules.

The Stable Dependencies Principle (SDP)

> **THE DEPENDENCIES BETWEEN PACKAGES IN A DESIGN SHOULD BE IN THE DIRECTION OF THE STABILITY OF THE PACKAGES. A PACKAGE SHOULD ONLY DEPEND UPON PACKAGES THAT ARE MORE STABLE THAT IT IS.**

Designs cannot be completely static. Some volatility is necessary if the design is to be maintained. We accomplish this by conforming to the *Common Closure Principle* (**CCP**). By using this principle, we create packages that are sensitive to certain kinds of changes. These packages are designed to be volatile. We expect them to change.

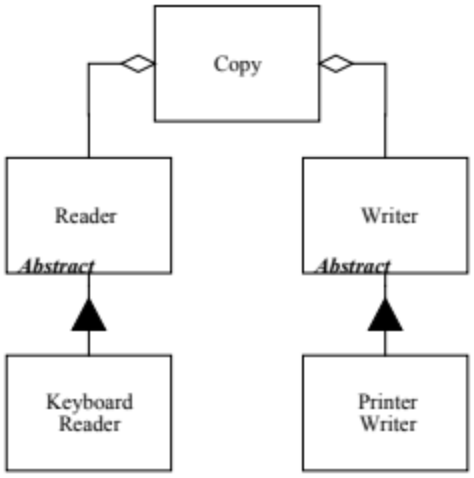
Any package that we expect to be volatile should not be depended upon by a package that is difficult to change! Otherwise the volatile package will also be difficult to change.

By conforming to the **SDP**, we ensure that modules that are designed to be instable (i.e. easy to change) are not depended upon by modules that are more stable (i.e. harder to change) than they are.

The Stable Abstractions Principle (SAP) [Code-related]

> **PACKAGES THAT ARE MAXIMALLY STABLE SHOULD BE MAXIMALLY ABSTRACT. INSTABLE PACKAGES SHOULD BE CONCRETE. THE ABSTRACTION OF A PACKAGE SHOULD BE IN PROPORTION TO ITS STABILITY.**

This principle sets up a relationship between stability and abstractness. It says that a stable package should also be abstract so that its stability does not prevent it from being extended. On the other hand, it says that an instable package should be concrete since its instability allows the concrete code within it to be easily changed.



Consider the "*Copy*" program above. The "*Reader*" and "*Writer*" classes are abstract classes. They are highly stable since they depend upon nothing and are depended upon by "*Copy*" and all their derivatives. Yet, "*Reader*" and "*Writer*" can be extended, without modification, to deal with many kinds of I/O devices.

Thus, if a package is to be stable, it should also consist of abstract classes so that it can be extended. Stable packages that are extensible are flexible and do not constrain the design.

The **SAP** and the **SDP** combined amount to the Dependency Inversion Principle for Packages. This is true because the **SDP** says that dependencies should run in the direction of stability, and the **SAP** says that stability implies abstraction. Thus, dependencies run in the direction of abstraction.

However, the **DIP** is a principle that deals with *classes*. And with classes there are no shades of grey. Either a class is abstract, or it is not. The combination of the **SDP** and **SAP** deal with packages and allow that a package can be partially abstract and partially stable.

Appendix: Related Document

Salesforce Packages: Pattern Strategy

Salesforce Packages
- Pattern Strategy.pc