Package Cohesion Principles

The Release Reuse Equivalency Principle (REP)
The Common Closure Principle (CCP)
The Common Reuse Principle (CRP)
The Release Reuse Equivalency Principle (REP)

- The granule of reuse is the granule of release.
  - A reusable element, be it a component, a class, or a cluster of classes, cannot be reused unless it is managed by a release system of some kind. Users will be unwilling to use the element if they are forced to upgrade every time the author changes it. Thus, even though the author has released a new version of his reusable element, he must be willing to support and maintain older versions while his customers go about the slow business of getting ready to upgrade. Thus, clients will refuse to reuse an element unless the author promises to keep track of version numbers, and maintain old versions for awhile. Therefore, one criterion for grouping classes into packages is reuse. Since packages are the unit of release, they are also the unit of reuse. Therefore architects would do well to group reusable classes together into packages.
The Common Closure Principle (CCP)

- **Classes that change together, belong together.**
  - A large development project is subdivided into a large network of interrelated packages. The work to manage, test, and release those packages is non-trivial. The more packages that change in any given release, the greater the work to rebuild, test, and deploy the release. Therefore we would like to minimize the number of packages that are changed in any given release cycle of the product. To achieve this, we group together classes that we think will change together. This requires a certain amount of precience since we must anticipate the kinds of changes that are likely. Still, when we group classes that change together into the same packages, then the package impact from release to release will be minimized.
The Common Reuse Principle (CRP)

- Classes that aren't reused together should not be grouped together.
  - A dependency upon a package is a dependency upon everything within the package. When a package changes, and its release number is bumped, all clients of that package must verify that they work with the new package -- even if nothing they used within the package actually changed. We frequently experience this when our OS vendor releases a new operating system. We have to upgrade sooner or later, because the vendor will not support the old version forever. So even though nothing of interest to us changed in the new release, we must go through the effort of upgrading and revalidating. The same can happen with packages if classes that are not used together are grouped together. Changes to a class that I don't care about will still force a new release of the package, and still cause me to go through the effort of upgrading and revalidating.
Tension between the Package Cohesion Principles

- These three principles are mutually exclusive. They cannot simultaneously be satisfied. That is because each principle benefits a different group of people. The REP and CRP makes life easy for reusers, whereas the CCP makes life easier for maintainers.

- The CCP strives to make packages as large as possible (after all, if all the classes live in just one package, then only one package will ever change).

- The CRP, however, tries to make packages very small.

- Fortunately, packages are not fixed in stone. Indeed, it is the nature of packages to shift and jitter during the course of development. Early in a project, architects may set up the package structure such that CCP dominates and development and maintenance is aided. Later, as the architecture stabilizes, the architects may refactor the package structure to maximize REP and CRP for the external reusers.
The Package Coupling Principles

The next three packages govern the interrelationships between packages. Applications tend to be large networks of interrelated packages. The rules that govern these interrelationship are some of the most important rules in object oriented architecture.
The Acyclic Dependencies Principle (ADP)

- *The dependencies between packages must not form cycles.*
  - Since packages are the granule of release, they also tend to focus manpower. Engineers will typically work inside a single package rather than working on dozens. This tendency is amplified by the package cohesion principles, since they tend to group together those classes that are related. Thus, engineers will find that their changes are directed into just a few package. Once those changes are made, they can release those packages to the rest of the project. Before they can do this release, however, they must test that the package works. To do that, they must compile and build it with all the packages that it depends upon. Hopefully this number is small.
The Acyclic Dependencies Principle (ADP)

- Consider:
The Acyclic Dependencies Principle (ADP)

- Astute readers will recognize that there are a number of flaws in the architecture. The DIP seems to have been abandoned, and along with it the OCP. The GUI depends directly upon the communications package, and apparently is responsible for transporting data to the analysis package.

- Still, let's use this rather ugly structure for some examples. Consider what would be required to release the Protocol package. The engineers would have to build it with the latest release of the CommError package, and run their tests. Protocol has no other dependencies, so no other package is needed. This is nice. We can test and release with a minimal amount of work.
The Acyclic Dependencies Principle (ADP)

- A Cycle Creeps In.
  - But now lets say that I am an engineer working on the CommError package. I have decided that I need to display a message on the screen. Since the screen is controlled by the GUI, I send a message to one of the GUI objects to get my message up on the screen. This means that I have made CommError dependent upon GUI.
The Acyclic Dependencies Principle (ADP)
The Acyclic Dependencies Principle (ADP)

- Now what happens when the guys who are working on Protocol want to release their package. They have to build their test suite with CommError, GUI, Comm, ModemControl, Analysis, and Database! This is clearly disastrous. The workload of the engineers has been increased by an abhorent amount, due to one single little dependency that got out of control. This means that someone needs to be watching the package dependency structure with regularity, and breaking cycles wherever they appear. Otherwise the transitive dependencies between modules will cause every module to depend upon every other module.
The Acyclic Dependencies Principle (ADP)

- Breaking a Cycle.
  - Cycles can be broken in two ways. The first involves creating a new package, and the second makes use of the Dependency Inversion Principle (DIP) and Interface Segregation Principle (ISP).
The Acyclic Dependencies Principle (ADP)

- This shows how to break the cycle by adding a new package. The classes that CommError needed are pulled out of GUI and placed in a new package named MessageManager. Both GUI and CommError are made to depend upon this new package.
**The Acyclic Dependencies Principle (ADP)**

- This shows a before and after picture of the other technique for breaking cycles. Here we see two packages that are bound by a cycle. Class A depends upon class X, and class Y depends upon class B. We break the cycle by inverting the dependency between Y and B. This is done by adding a new interface, BY, to B. This interface has all the methods that Y needs. Y uses this interface and B implements it. Notice the placement of BY. It is placed in the package with the class that uses it.
The Stable Dependencies Principle (SDP)

- *Depend in the direction of stability.*
  - Though this seems to be an obvious principle, there is quite a bit we can say about it. Stability is not always well understood.
The Stable Dependencies Principle (SDP)

- This shows a stable package. This package has three packages depending upon it, and therefore it has three good reasons not to change. We say that it is responsible to those three packages. On the other hand, X depends upon nothing, so it has no external influence to make it change. We say it is independent.
The Stable Dependencies Principle (SDP)

- This shows a very instable package. Y has no other packages depending upon it; we say that it is irresponsible. Y also has three packages that it depends upon, so changes may come from three external sources. We say that Y is dependent.
The Stable Dependencies Principle (SDP)

- We can envision the packages structure of our application as a set of interconnected packages with instable packages at the top, and stable packages on the bottom. In this view, all dependencies point downwards. Those packages at the top are instable and flexible. But those at the bottom are very difficult to change. And this leads us to a dilemma: Do we want packages in our design that are hard to change?

![Diagram of SDP](image-url)
The Stable Dependencies Principle (SDP)

- Clearly, the more packages that are hard to change, the less flexible our overall design will be. However, there is a loophole we can crawl through. The highly stable packages at the bottom of the dependency network may be very difficult to change, but according to the OCP they do not have to be difficult to extend! If the stable packages at the bottom are also highly abstract, then they can be easily extended. This means that it is possible to compose our application from instable packages that are easy to change, and stable packages that are easy to extend. This is a good thing. Thus, the SAP is just a restatement of the DIP. It states the packages that are the most depended upon (i.e. stable) should also be the most abstract.