Granularity
The Principles of Package Cohesion

1. Reuse-Release Equivalence Principle (REP)
“The unit of re-use is the unit of release.”

• In general, sets of collaborating classes are reused
📫package as the unit of reuse
• Only packages that are tested and released through a tracking system
can be effectively reused
– introducing changes for re-users in a controlled way
• If a package contains classes that should be reused, then it should not contain classes that are not designed for reuse.
– either all classes in a package are reusable or none of them
– group classes in packages from the perspective of their reusers

Consequences
• only changes in classes interesting to the reuser will lead to a new
release of the package
• avoiding accidental reuse of classes not designed for re-use
• reduced effort for
– making releases
– upgrades at the reuser side

2.Common-Reuse Principle (CRP)

“The classes in a package are reused together. If you reuse one of the classes in a package, you reuse them all.”

• If the user is only interested in a part of a package
– its code still depends on the whole package
– own code has to be revalidated on any new release of the used package
(even the change affects a class that is actually not used)
• Classes that tend to be reused together belong in the same package
(similar to Single-Responsibility Principle (SRP) for packages)
• Classes that are not tightly bound to each other with class relationships should not be in the same package
• We want to make sure that the classes in a single package are inseparable, i.e., it is impossible to depend on some and not the others
📫high cohesion

3.Common-Closure Principle (CCP)
“The classes in a package should be closed together against the same kinds of changes. A change that affects a package affects all the classes in that package and no other packages.”

• The Single-Responsibility Principle says that a **class** should not contain multiple reasons to change

• Analogously, the Common-Closure principle says that a **package** should not contain multiple reasons to change

📫 All classes that are likely to change for the same reason should be packaged together

• Note: The Open-Closure Principle states that classes should be closed for modification but open for extension

• Full closure is not attainable; but, the common-closure principle makes the closure strategic by designing systems so that they are closed to the **most common kinds of changes**

Stability

The Principles of Package Coupling

1.Acyclic-Dependencies Principle (ADP)

• Goals

– Stabilize and release the project in pieces

– Avoid interference between developers ("The morning after"-Syndrome) by releasing packages as own units which do not immediately affect its users

– Allow incremental integration

• Acyclic-Dependencies Principle

– Packages are releasable units of work
– A working package is released and other developers can use it
– Development takes place on a private copy of the package, while other use the released one
– As a new version is available, developers can decide if they want to upgrade or keep the old version
To make this process work:
“Allow no cycles in the package-dependency graph.”

- Directed Acyclic Graph (DAG)
- Easy to find out who is affected by a change
- Easy to make isolated tests
- When it is time to release the whole system it is done bottom-up
- The typical process of developing a package structure is bottom-up

– As the software grows, we want to keep changes localized 📫 Single-Responsibility Principle and Common-Closure Principle
– When the software grows further, we are concerned with reusability and compose packages according to the Common-Reuse Principle
– Finally, cycles appear and the Acyclic-Dependencies Principle is applied

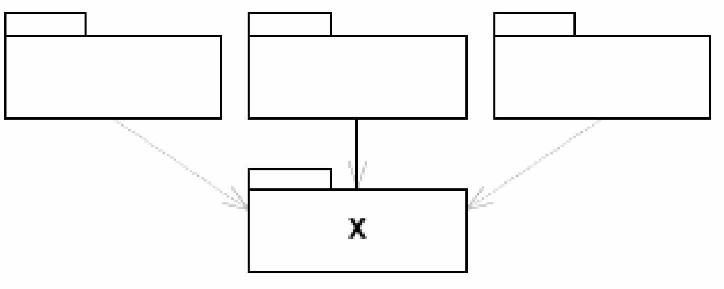
Developing the package structure top-down would fail: we don‘t know much about the common closure, we don‘t know the reusable elements and we would certainly create packages that produce cycles

2.Stable-Dependencies Principle (SDP)

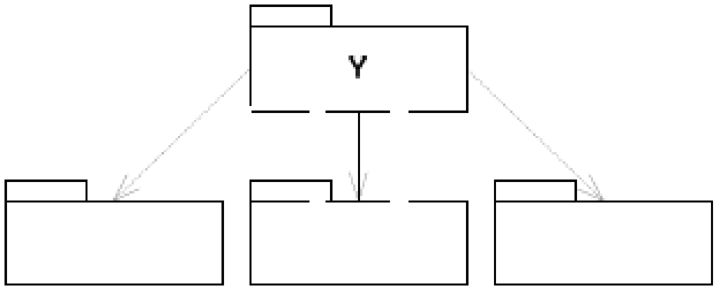
„Depend in the direction of stability.“

• Designs cannot be completely static; we expect some packages to change!
• Using the Common-Closure Principle, we create packages that are designed to be volatile
• Although, changing a package designed to be volatile can be hard if several other packages depend on it
📫Modules intended to be easy to change may not depend on modules that are harder to change than they are
- or in other words -
Depend always on something which is more stable than you are

• The stability of a package refers to the amount of work required to make a change
• A stable, responsible package; three good reasons not to change



- An instable, irresponsible package; free to change



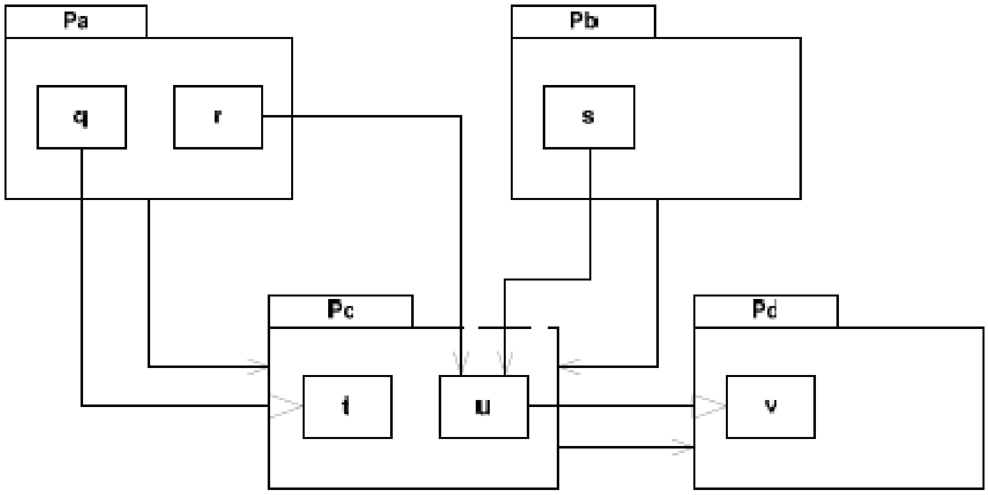
- Stability metrics

– Ca Afferent Couplings: Number of classes outside this package that
depend on classes within this package

– Ce Efferent Couplings: Number of classes inside this package that depend
on classes outside this package

– I Instability: I = Ce / (Ca + Ce); 0 ≤ I ≤ 1

- Example



Metrics for P

C_a = 3

C_e = 1

I = ¼

3.Stable-Abstractions Principle (SAP)

„A package should be as abstract as it is stable.“

- Stable packages should also be abstract so that its stability does not prevent it from being extended
- Instable packages should be concrete; its concrete code can be easily changed
- The Stable-Abstractions Principle and the Stable-Dependencies Principle correspond to the Dependency Inversion Principle for packages
  - Stable-Abstractions Principle: Dependencies should run in the direction of stability
  - Stable-Dependencies Principle: Stability implies abstraction

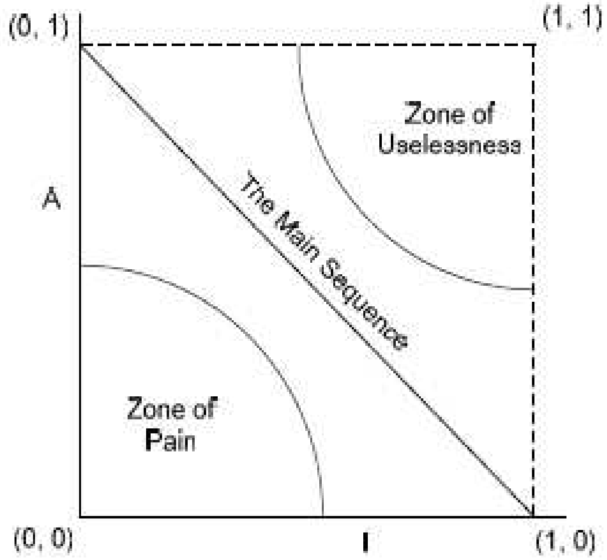
♉ Dependencies run in the direction of abstraction

- Abstraction metric
  - Nc Number of classes in the package
  - Na Number of abstract classes in the package
  - A Abstractness: A = NA / NC; $0 \leq A \leq 1$

Correlation of Stability and Abstractness

- Abstract packages should be responsible and independent (stable)
  - Easy to depend on
- Concrete packages should be irresponsible and

can be dependent (instable)
– Easy to change
• Zone of pain: highly stable and concrete package
– it is difficult to change because of its stability
– it cannot be extended because it is not abstract
– exceptions: typically utility packages, e.g. the string class
• Zone of uselessness: packages that are maximally abstract, but have
no dependents
• Main sequence: packages that are not too abstract, not too instable



Summary

• Package cohesion
– a cohesive package contains classes that implement one and only one
responsibility
– We extended the view of cohesion to packages
– The opposing forces involved in reusability and developability need to be
considered when packaging classes

– Three principles guide the decisions to partition the classes

- Package coupling

– The complexity of a system is significantly determined by the number of dependencies in this system

– Some dependencies are necessary, some others cause pain

– The principles help in guiding the decisions to package classes in order to avoid bad dependencies

– The dependency-management metrics measure the conformance of a design to a pattern of dependency and abstraction