

# Principles of Package Design

## Principles of coupling and cohesion

- A class is too fine grained to be used as the organizational unit for larger applications
- Also, classes may have roles that have nothing to do with the 'components' of the application (e.g. think of the classes in many design patterns)
- Packages are used in design to organize software on a coarser level than classes
- By grouping classes into packages, we can
  - reason about the design at a higher level of abstraction
  - manage the development and distribution of the software
- But how to decide what classes to put into a same package?

# Principles of coupling and cohesion

- Questions
  - What are the principles for allocating classes to packages?
  - What design principles govern the relationships between packages?
  - Should packages be designed before or after the classes?
  - How are packages physically represented? In different languages, different environments?
  - Once created, to what purpose we put these packages? Units of architecture, reuse, deployment, compilation...?
- Six principles for organizing software into packages
  - three concerning cohesion within a package
    - classes in a package must have a good reason to be there, they belong together according to some criteria
  - three concerning coupling between packages
    - dependencies cross package boundaries

# Principles of package cohesion

- Granularity: the principles of package cohesion
  - REP, CRP and CCP
  - help to decide how to partition classes to packages
  - assume that most of the classes and their interrelations have been discovered, take thus a "bottom-up" view to partitioning
- Cohesion of a package is more complicated than having modules provide only a single responsibility. Cohesion is a richer concept and involves issues like
  - reusability
  - developability
- Achieving cohesion is balancing different forces, often dynamic forces that evolve during the development as the focus of the project changes.

## Reuse-Release Equivalence Principle (REP)

*The granule of reuse is the granule of release*

- Anything that we reuse, must be released and tracked. A plain class simply is not reusable.
  - reusability requires a tracking/release system for reusable assets to offer guarantee of notification, safety and support that the reusers will need.
- This makes a political force a criterion for partitioning the software
  - reusable classes must be put into a released package
  - reusability management, conventions, policy create high level software structure

## 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.*

- (where it says 'reuse' think just 'use')
- The internal content of a reusable package must be considered from the point of view of potential reusers
  - either all the classes in a reusable package are reusable, or none of them are
  - all the reusable classes in a reusable package must be targeted to same audience
    - matrix arithmetic -classes and a financial framework are both reusable, but also reusable separately. Do not put them in the same package.
    - compare this to ISP
- Classes that tend to be reused together, belong together
  - these classes typically have tight coupling
  - e.g. container class and its associated iterators

## Common-Reuse Principle (CRP)

- CRP tells more about what classes should not be put to same package
  - When one package uses another, a dependency is created (the dependency may be to only one class)
  - Every time the used package is released (e.g. DLL, JAR), the using package must be revalidated and re-released.
  - Thus if a package depends on another package, it should depend on all classes of the package. Otherwise a change in a non-dependend class forces the depending package to be revalidated.
- Thus, classes which are not tightly bound with class relationships should not be in the same package
- The classes in the same package should be inseparable – impossible to reuse one without another

## 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.*

- This is the SRP for packages: the package should not have multiple reasons to change
  - if two classes are so tightly bound, either physically or conceptually, that they change together, then they belong to the same package
- Often maintainability is more important than reusability. When a change is needed, it should affect only one package.
  - Minimizing the workload of releasing, revalidating, redistributing the software.
- Closely related to OCP.
  - 100% closure is not attainable, thus closure must be strategic: close against certain types of changes that are probable
  - CCP encourages to group together classes that are open to same type of change.

## Principles of package coupling

- Stability: the principles of package coupling
  - ADP, SDP and SAP
  - these deal with the relationships between packages
  - address the tension between logical design and developability
- Forces acting on the package architecture are technical, political and ... volatile
- We define also two measures for evaluating the package architecture

## Acyclic Dependencies Principle (ADP)

- "The morning after –syndrome"
  - Everybody is changing and changing the code trying to make it work with the latest changes somebody else did. No stable version can be built.
- Two solutions – the weekly build or partitioning the development environment into releasable packages and ensuring ADP
- The weekly build
  - Have developers work alone most of the week and integrate on Friday
  - Works on medium-sized projects, forces to make the iteration longer (monthly build?) as software size increases, thus losing rapid feedback

## Acyclic Dependencies Principle (ADP)

*Allow no cycles in the package dependency graph*

- Use release-control for packages
- Each team works independently on its own packages and can decide independently when to adapt the packages to new releases of the packages they use.
- No team is at the mercy of the others.
- There is no “big bang” integration, it happens in small increments.
- To make this simple and efficient scheme work, the dependency tree must be managed, *there must not be any cycles* or the “morning after –syndrome” is not avoided.

## Acyclic Dependencies Principle (ADP)

- Without a cycle it is easy to compile, test and release “bottom-up” when building the whole software
- With a cycle in the dependency graph, we may end up with a package that depends on every other package in the system
  - Problems with building the software: all teams will be stepping all over one another since they must be using exactly the same release of each others packages
- The packages in the cycle will become *de facto* a single package
  - Compile-times increase
  - Testing becomes difficult, complete build is needed to test a single package

## Acyclic Dependencies Principle (ADP)

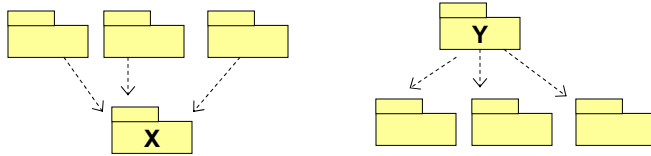
- Breaking a cycle: always possible to break a cycle in a dependency graph. Two techniques:
  - Apply the DIP: create an abstract base class that the depending client owns and make the used depended class in the used package implement it.
  - Create a new package on which both packages depend. Move the classes that they both depend on into that new package.
- Corollary: the package structure cannot be designed top-down. It evolves as the system grows and changes!
  - This has implications on how you should organize and plan the work in a multi team project!

## Stable Dependencies Principle (SDP)

*Depend in the direction of intended stability*

- Designs cannot be completely static, some volatility is required in order the design to be maintained.
  - this is achieved by CCP - some packages are sensitive to certain types of changes. These are *designed* to be volatile.
- Any package that we expect to be volatile should not be depended on by a package that is difficult to change. Otherwise the volatile package will also become difficult to change!
  - A module designed to be easy to change can (accidentally) be made hard to change by someone else hanging a dependency on it.
- Definition: 'Stability' = not easy to change
  - factors affecting how much effort it takes to change a package: size, complexity, clarity, incoming dependencies
  - sure way to make a package hard to change: make many other packages depend on it.
- By conforming to SDP we ensure that modules that are intended to be easy to change are not depended on by modules that are harder to change.

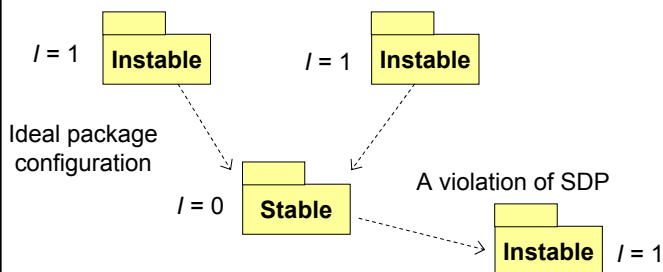
## Stable Dependencies Principle



- **X** is a stable package, it has three packages depending on it, therefore three good reasons *not to change*.
- **Y** is independent package, no other package depends on it. A change in Y affects no other package.
- A *Stability Metric I* is defined to measure the stability of a package (w.r.t dependencies)
  - $C_a$  is afferent couplings: the number of classes outside this package that depend on classes within this package
  - $C_e$  is efferent couplings: the number of classes inside this package that depend on classes outside this package

$$I = \frac{C_e}{C_a + C_e}$$

## Stable Dependencies Principle



- Not all packages should be stable, we want some to be stable and some not.
- Fixing the violation of SDP
  - employ DIP: create an interface that contains all the methods the stable class needs from the depended instable class, and have the depended instable class implement the interface. The interface is put in a new stable package that both other packages depend on.

## High level design ?

- Where do we put the high-level design?
- The high-level architecture should not change very often, architectural decisions must not be very volatile.
  - So put the high-level design into stable packages ( $I = 0$ ), that every other module would depend on?
  - If architecture should be flexible to some extent, you cannot do this.
- We need a package that is maximally stable but at the same time flexible enough to withstand change.
- OCP gives the answer: flexible enough to be extended without requiring modification.
  - these are *abstract classes*

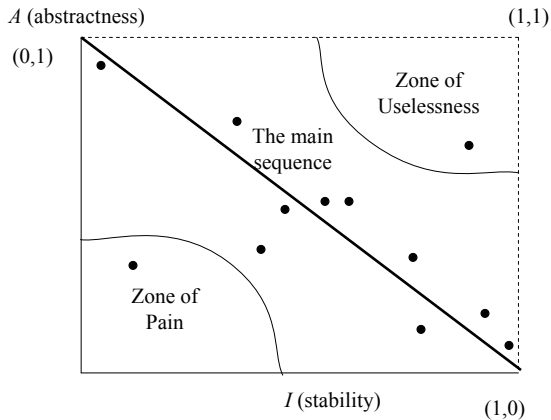
## Stable Abstractions Principle (SAP)

*A package should be as abstract as it is stable*

- The principle sets up a relationship between stability (as defined before) and abstractness: a stable package should contain abstract classes
  - stable packages should also be abstract so that stability does not prevent them from being extended.
  - instable packages should also be concrete since the instability allows the concrete code be easily changed.
- SDP and SAP together form the DIP for packages: *dependencies run in the direction of abstractions.*
  - however, with packages we have varying degrees of abstractness, the package can contain both abstract and concrete classes (with classes DIP was clear-cut, a class is either abstract or concrete)
  - we need a metric  $A$  to measure the abstractness of a package
    - A simple metric is the ratio of number of abstract and concrete classes in the package.

$$A = \frac{N_a}{N_c}$$

# Relationship between Stability and Abstractness



Focus your effort on improving the architecture on the packages that are far from the main sequence.

The cumulative distance from the main sequence of all packages can be used as a metric of the quality of the whole design !

- Packages in zone of pain are *rigid*: cannot be changed, cannot be extended
- Packages in zone of uselessness are abstract, yet have no dependents. Therefore they are useless.
- The most ideal packages are on the two endpoints of the main sequence, however in practice the distance from the main sequence is what should be minimized.

What is software architecture?

Everything we do NOT want to change !