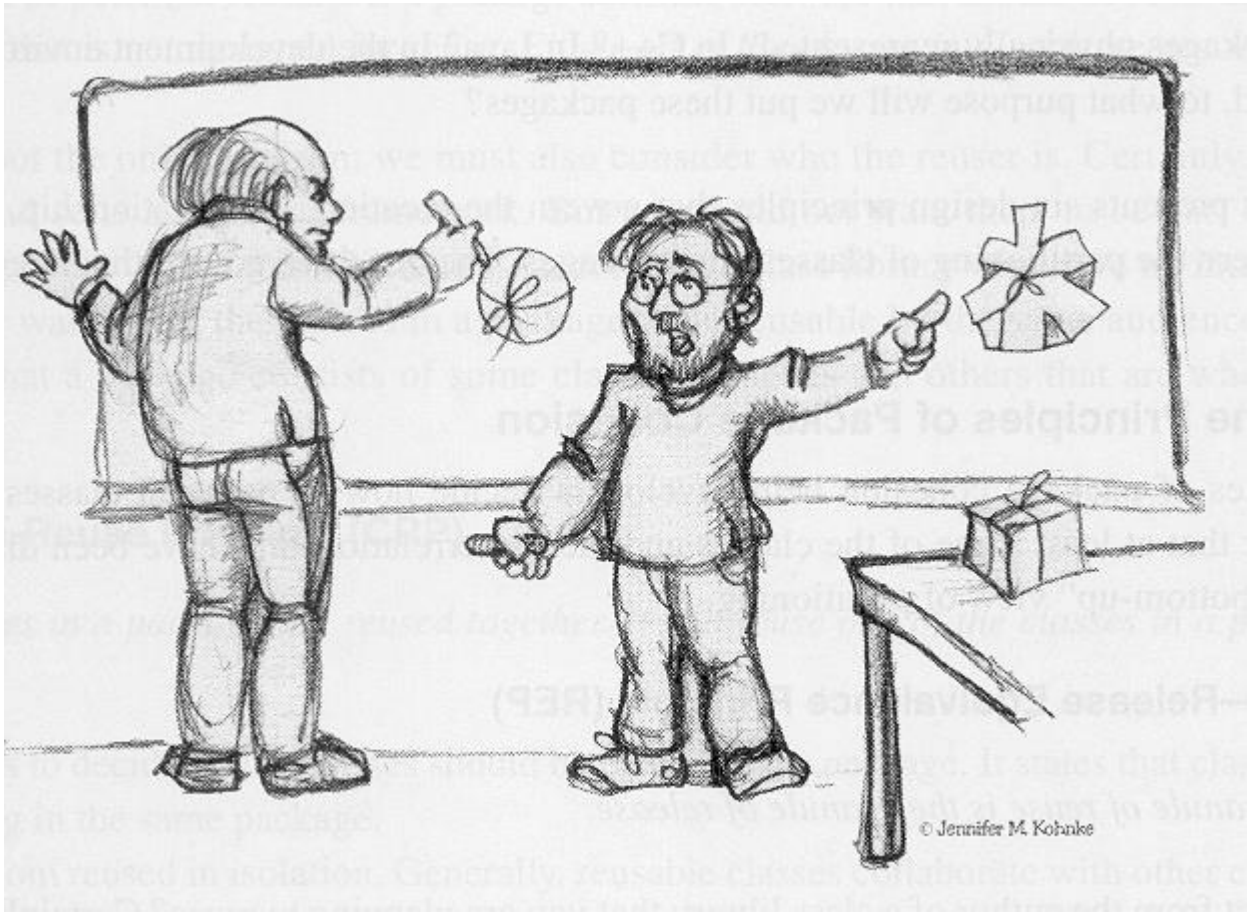
Design Principles 2

For

Software Components



Agenda

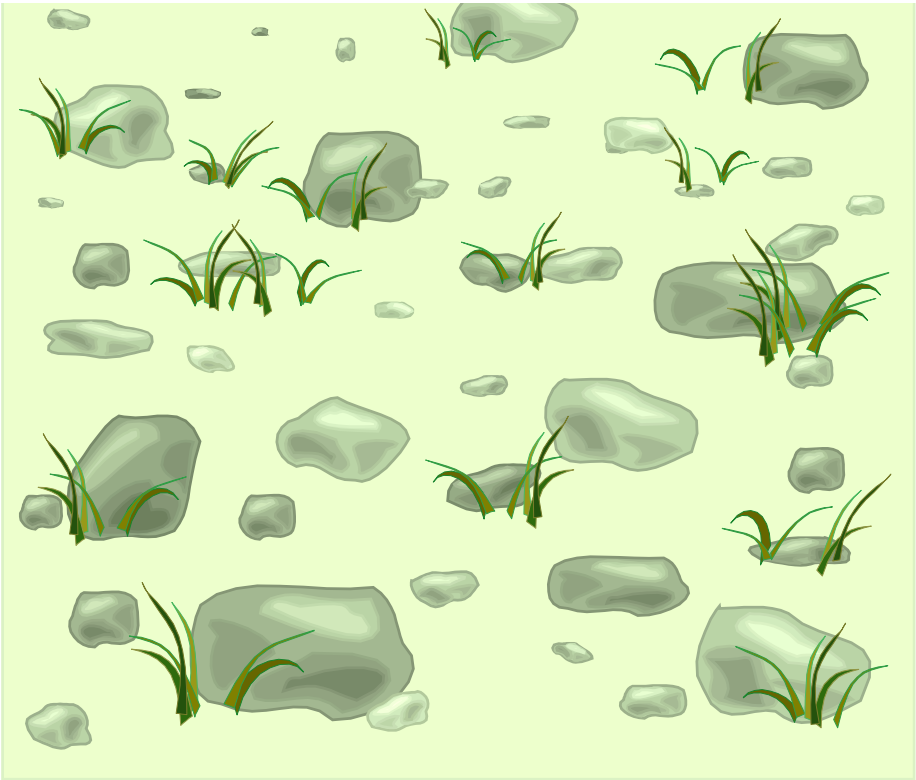
- Granularity:
  - REP
  - CRP
  - CCP
- Stability:
  - ADP
  - SDP
  - SAP
- Dependency Management Metrics

Granularity

The Principles of Package Cohesion

The three principles of package cohesion help developers decide how to partition classes into packages / components.

(*"bottom-up" view of partitioning*)



REP

Reuse/Release Equivalency Principle

REP: The Reuse/Release Equivalency Principle

The granule of reuse is the granule of release.

Only components that are released through a tracking system can be effectively reused

- To make it worth your while to reuse another person's code (component), you want the author to guarantee to maintain it for you.
  - After all, if *you* have to maintain it, you are properly better off if you design the component yourself.
- You will want the author too notify you in advance of any changes he plans to the interface and functionality of the component.
  - The author must also give you the option to refuse to use the new version, and still support the old version for some period of time

CRP

The Common Reuse Principle

Granularity: CRP The Common Reuse Principle

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

- The common re-use principle, by which classes that are reused together are packaged together in a single component.
- **An important corollary of the CRP is that classes that aren't *always* re-used together *shouldn't* be packaged together as a component**.
- **Of particular importance to CBD is separating the re-usable from the non re-usable.**

CCP

The Common Closure Principle

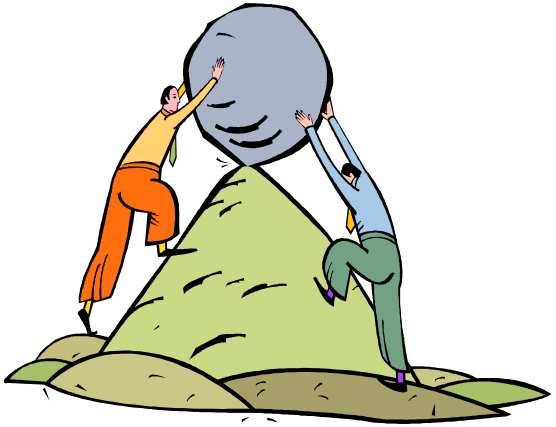
CCP The Common Closure Principle

The classes in a component should be closed together against the same kinds of changes.
A change that affects a component affects all the classes in that component and no other components .

- This is the Single-Responsibility Principle restated for packages.
- In many applications, maintainability is more important than reusability.
- CCP will minimize the workload related to releasing, revalidating, and redistributing the software

The common closure principle (CCP) says we should package things together (into single components) that are likely to change together.

Stability



The Principles of Package/Component Coupling

Component Dependency management

- Good dependency management is key to component based design.
- You must have a clear idea exactly what (in terms of other components) your component *is* dependent on, and what it *isn't* dependent on.
- Good dependency management is supported by the following design principles:
  - ADP
  - SDP
  - SAP

ADP

The Acyclic Dependencies Principle

ADP The Acyclic Dependencies Principle

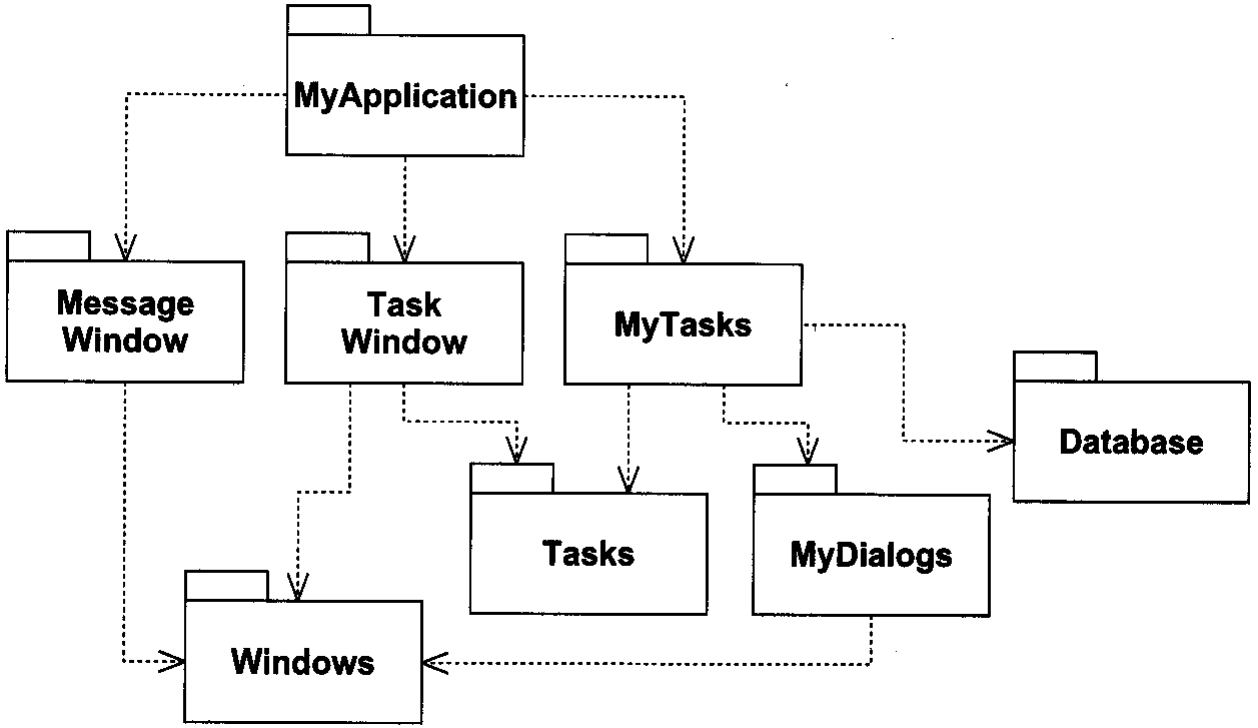
Allow no cycles in the package-dependency graph

ADP The Simple Example

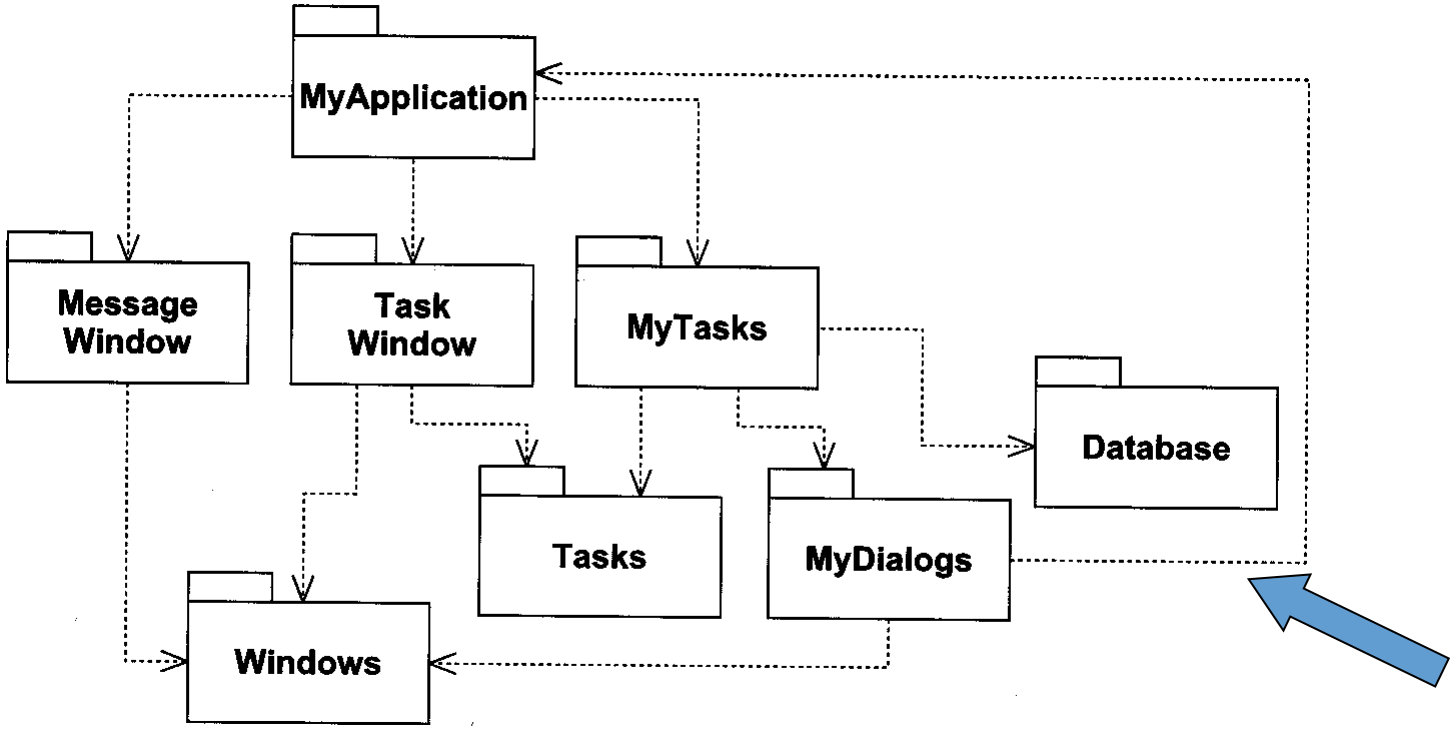
- Consider two components which are mutually (and hence cyclically) dependent on each other – each requiring the other when deployed.

- In such circumstance we should surely ask why do we need *two* components?
  - If the components can't be deployed separately, then the reality of the situation is that we're actually dealing with a single component, and that releasing the two components separately can only make the client developers life more difficult!

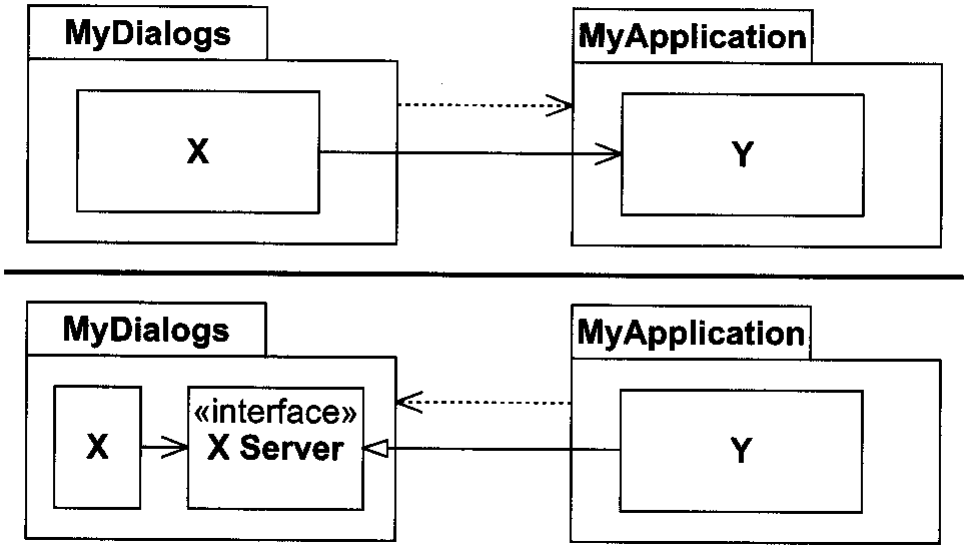
ADP – Adv. Example 1



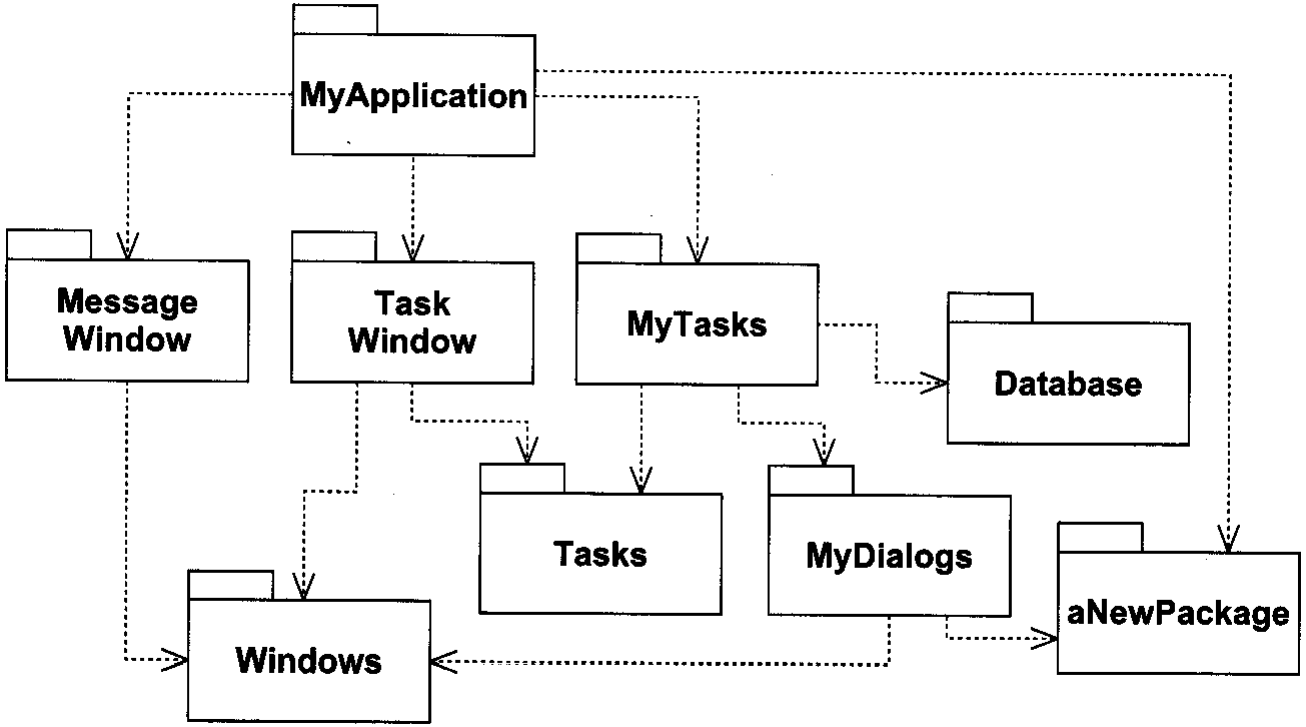
ADP – Adv. Example 2



Braking the cycle with DIP



Braking the cycle with a new component



Move the class(es) that they both depend on into a new package/component

SDP

The Stable Dependencies Principle

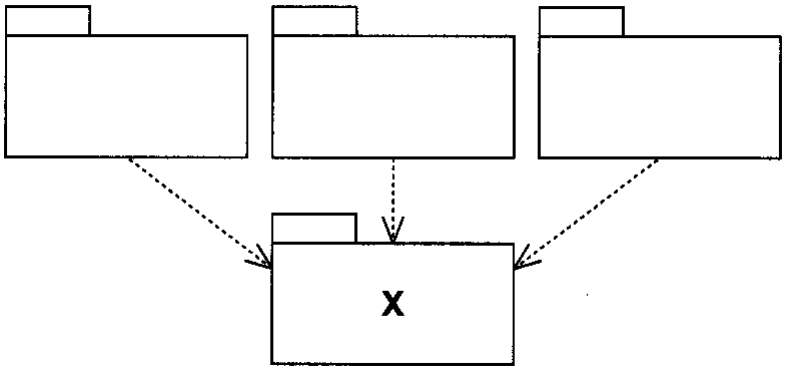
SDP The Stable Dependencies Principle

Depend in the direction of stability

- The stable dependencies principles states that a component should only depend on components that are more stable (less likely to change) than itself.

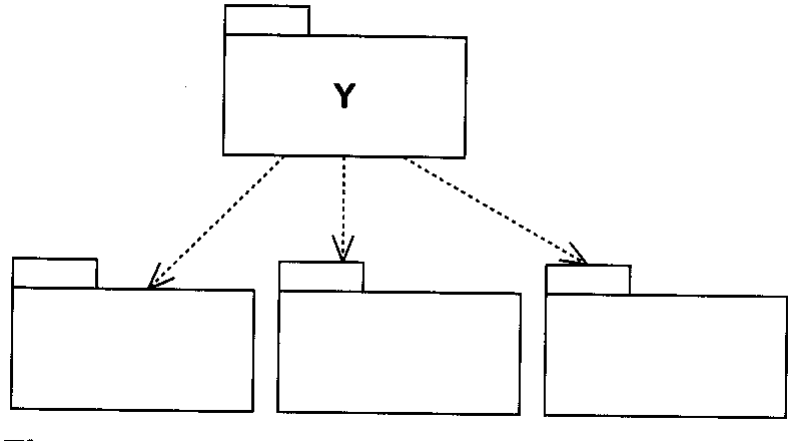
- Violating this rule means that the dependent component is likely to be in a continuing state of flux as its author strives to keep up with the changes being made to the components it is dependent upon.

Stability



X is a stable component

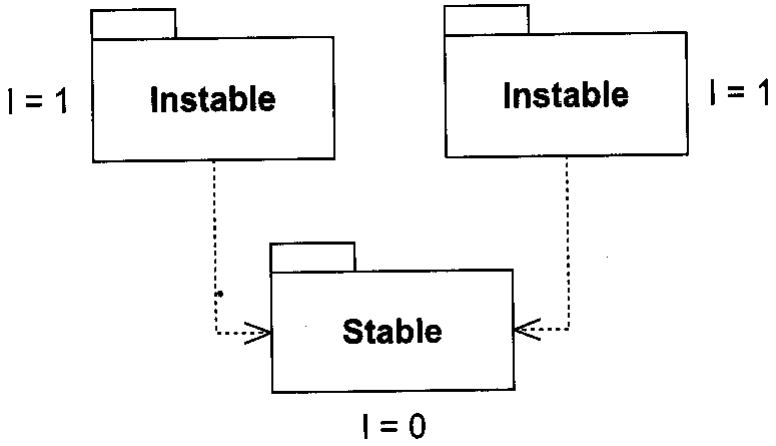
And is independent



Y an unstable component

And is dependent

Not all components should be stable



SAP

the Stable-Abstraction Principle

SAP the Stable-Abstraction Principle

A package should be as abstract as it is stable

- 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.

DEPENDENCY MANAGEMENT METRICS

Or how to measure stability?

Positional stability of a component

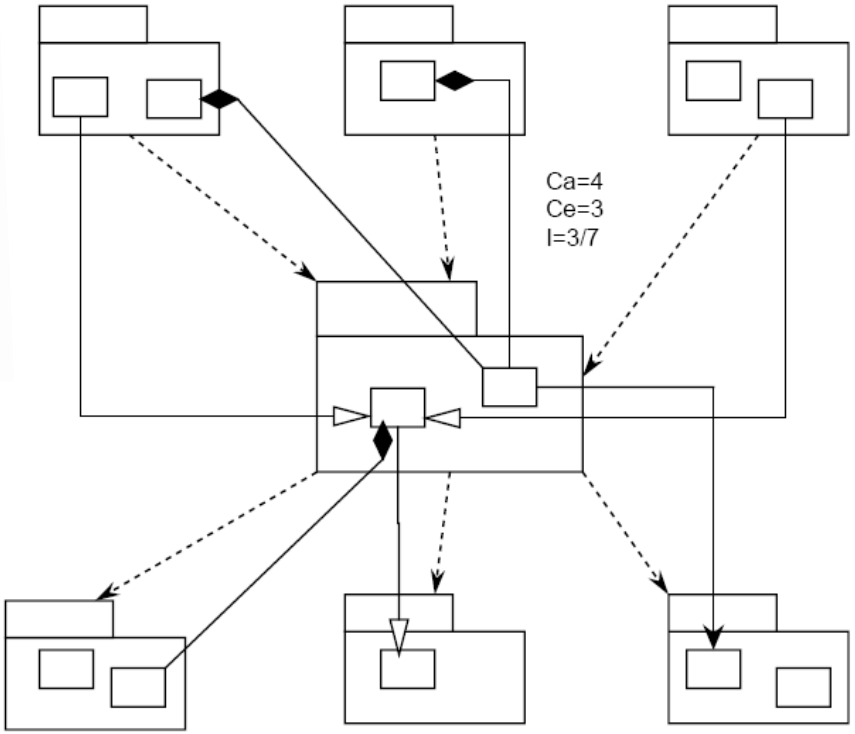
- **Ca**: *Afferent Couplings*: The number of classes outside this package that depend upon classes within this package.
- **Ce** : *Efferent Couplings* : The number of classes inside this package that depend upon classes outside this package.
- **I** : *Instability* :

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

  - This metric has the range [0,1].
  - I=0 indicates a maximally stable package.
  - I=1 indicates a maximally instable package.

Instability Example

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

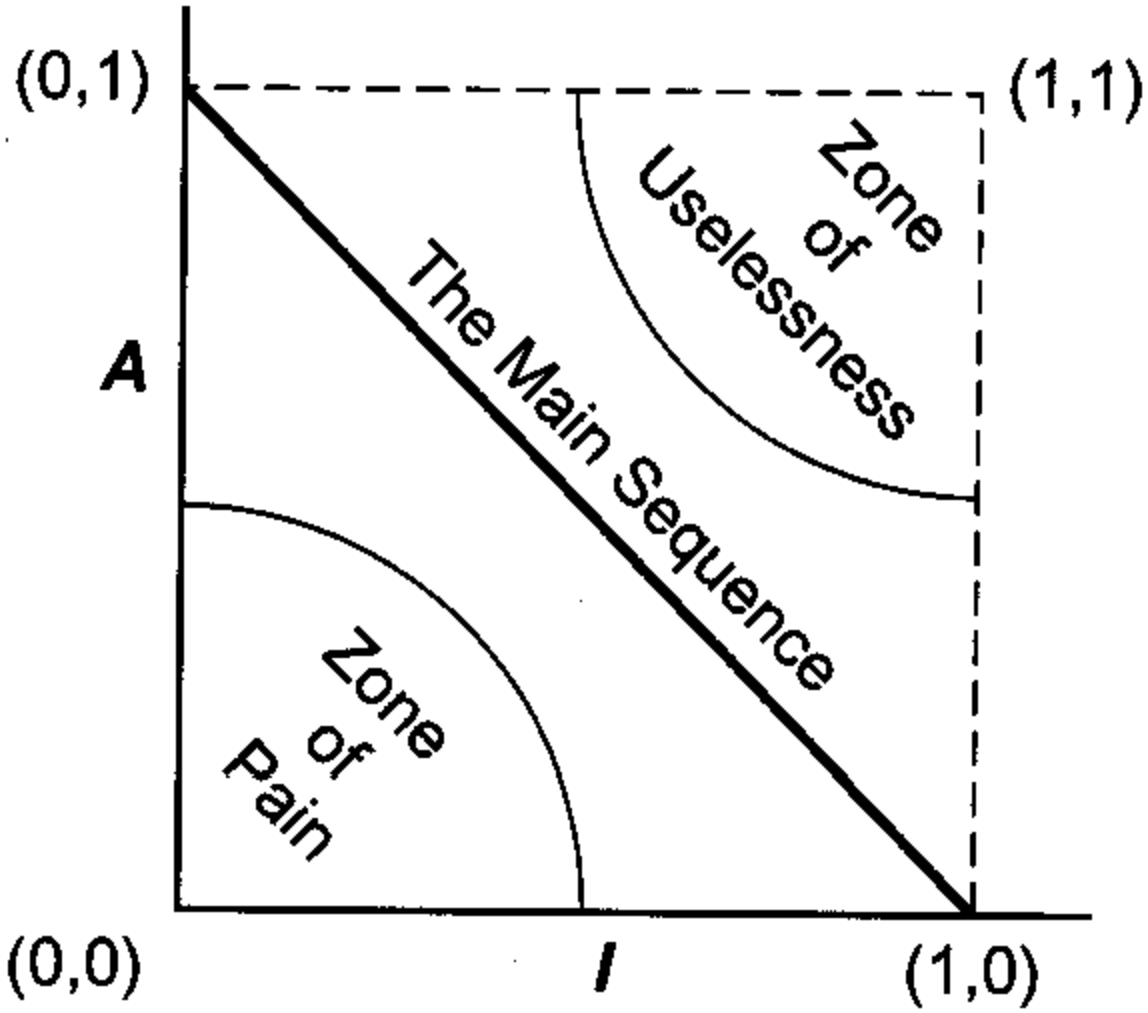


How to Measure Abstraction

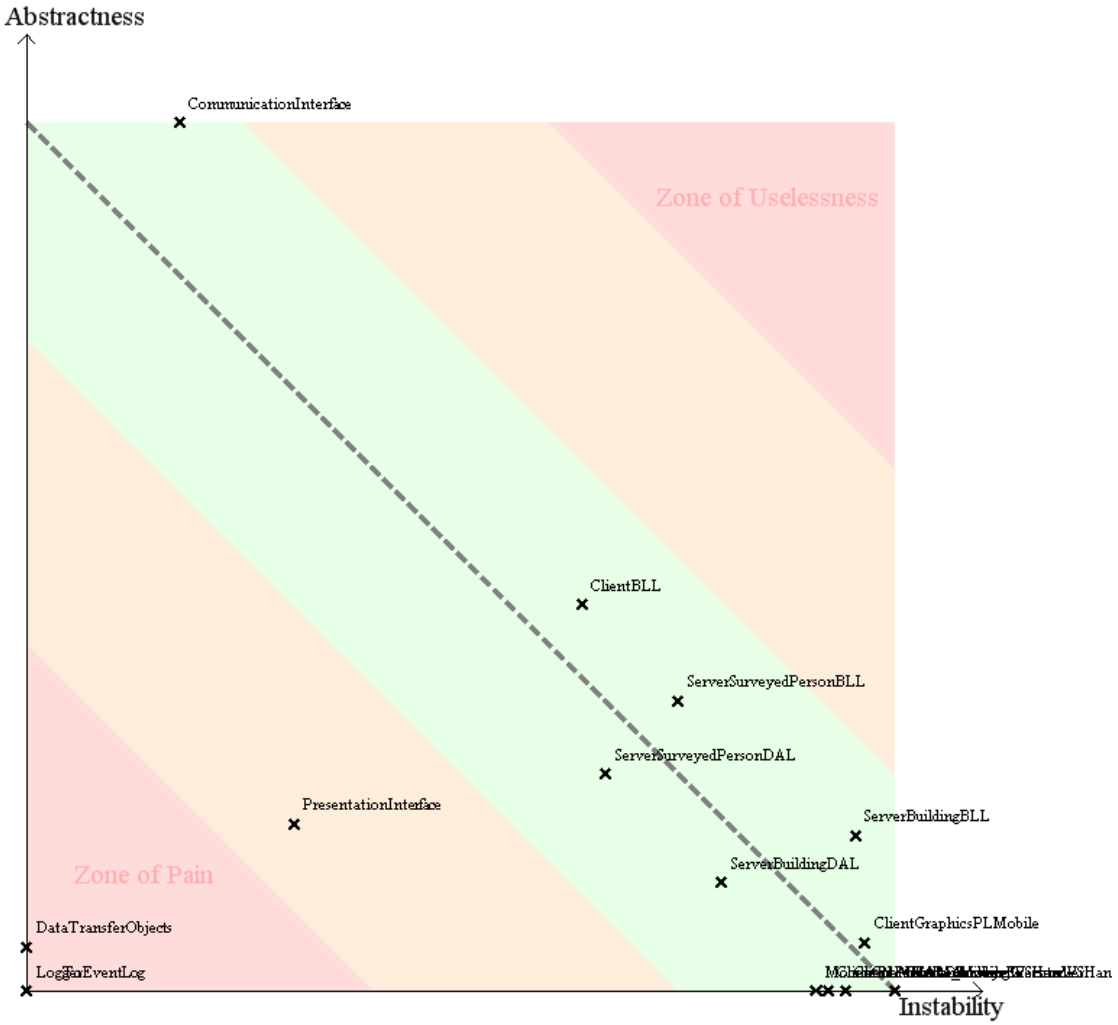
$$A = \frac{N_a}{N_c}$$

- **N**a: the number of abstract classes in the component.
- **N**c: the total number of classes in the component.

The A-I Graph


(0,1)
(1,1)
Zone of Uselessness
The Main Sequence
A
Zone of Pain
(0,0)
I
(1,0)

The A-I From a Bachelor Project



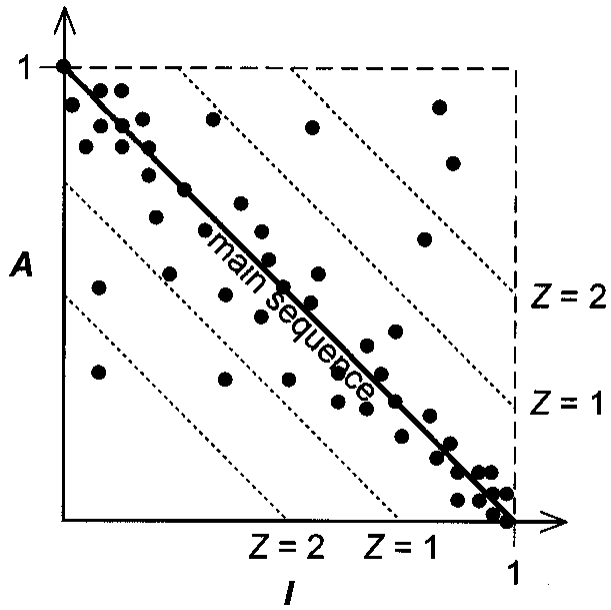
Distance from Main Sequence

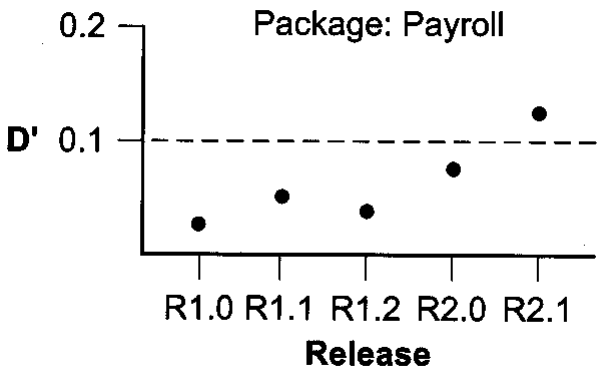
D—Distance.

$$D = \frac{|A + I - 1|}{\sqrt{2}}.$$

This metric ranges from [0,~0.707].

$$D' = |A + I - 1|.$$





Links

- **NDepend**: code metrics at your service for .Net
  http://ndepend.com/Default.aspx
  Se also paper in Campusnet file storage

- **XDepend:** code metrics at your service for Java
  http://www.xdepend.com/