# Principles of Object-Oriented Design

## Part II

# The Law of Demeter

*Any object receiving a message in a given method must be one of a restricted set of objects.*

1. **Strict Form:** *Every supplier class or object to a method must be a preferred supplier*
2. **Minimization Form***: Minimize the number of acquaintance classes / objects of each method*

**Lieberherr and Holland**

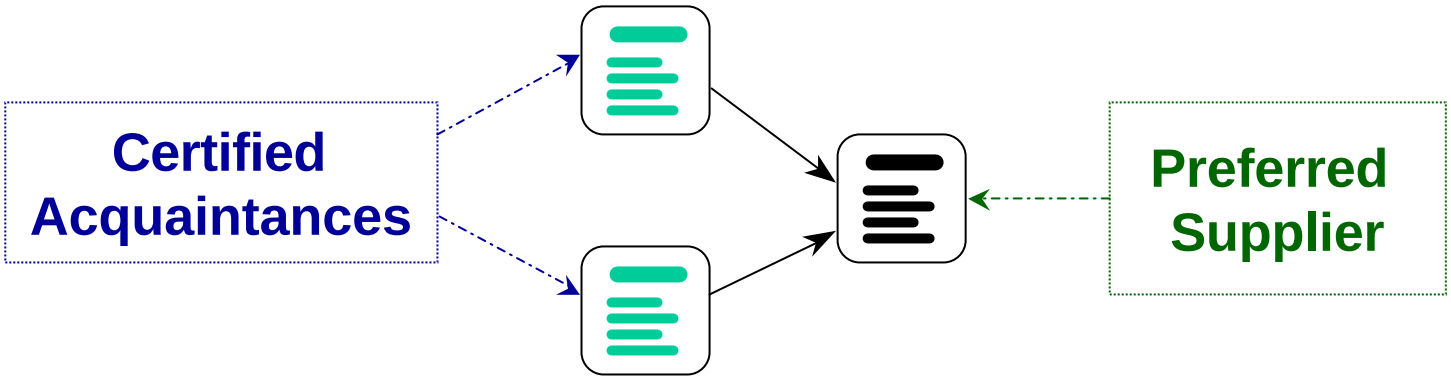
# Shy Code

- Don't reveal yourself to others
- Don't interact with to many people

- Spy, dissidents and revolutionaries
  - eliminating interactions protects anyone

- The General contractor example
  - he must manage subcontractors

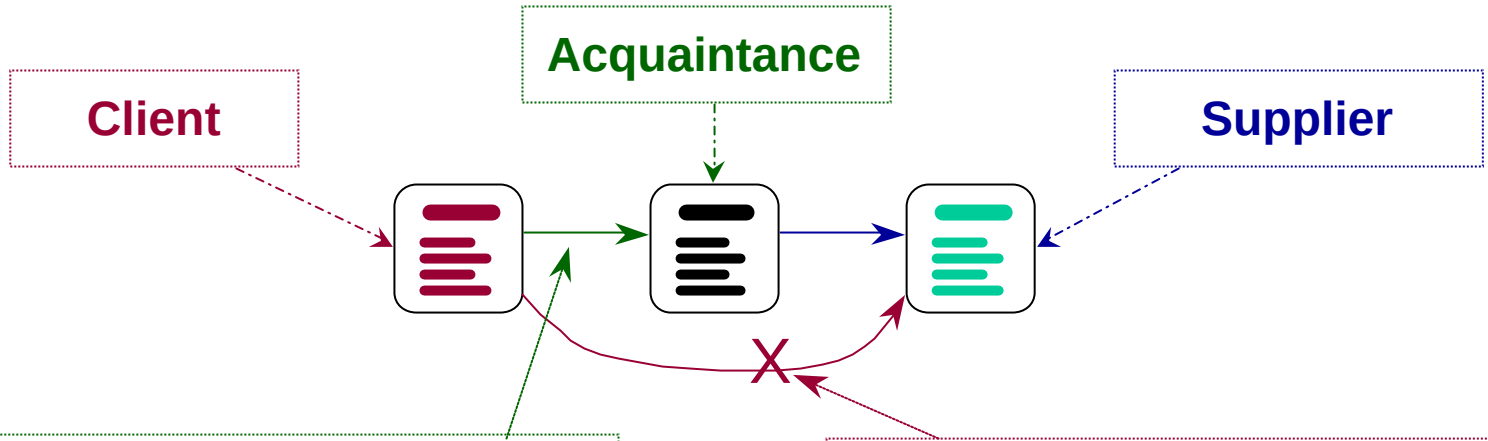
# Law of Demeter




- Minimize the number of acquaintances which directly communicate with a supplier
  - easier to contain level of difficulty per method
  - reduces amount of rework needed if supplier's interface is modified
  - So only allow "certified" classes to call supplier

# Demeter's "Good Style"

- Access supplier methods only through methods of "preferred acquaintances"
  - bypass preferred supplier only if later optimization demands direct access



# Demeter's Law for Functions

```
class Demeter {
private:
  A *a;
public:
  // …
  void example(B& b);

void Demeter::example(B& b) {
  C c;
  int f = func();
  b.invert();
  a = new A();
  a->setActive();
  c.print();
}
```

*Any methods of an object should call only methods belonging to:*

*itself*

*passed parameters*

*created objects*

*directly held component objects*

# Acceptable LoD Violations

- If optimization requires violation
  - Speed or memory restrictions
- If module accessed is a fully stabilized “Black Box”
  - No changes to interface can reasonably be expected due to extensive testing, usage, etc.
- Otherwise, do not violate this law!!
  - Long-term costs will be very prohibitive

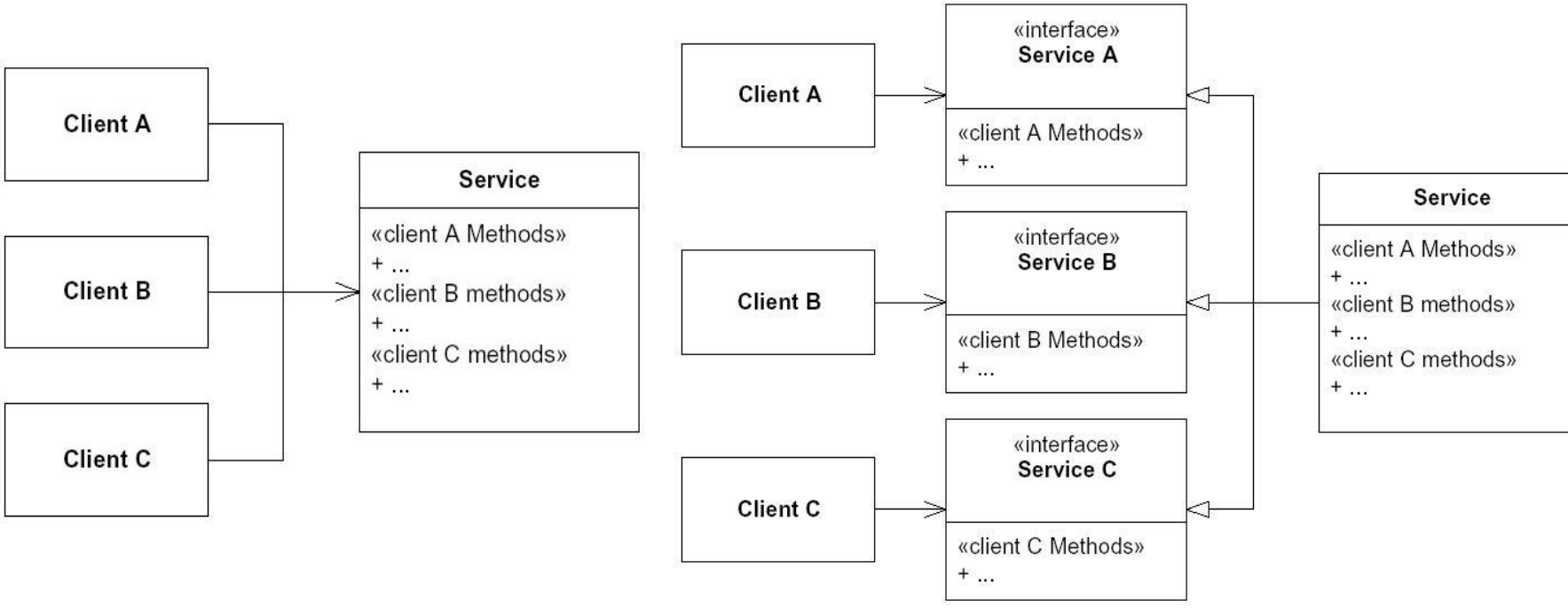
# Interface Segregation Principle

> *Clients should not be forced to depend upon interfaces that they do not use.*
>
> **R. Martin**, 1996

- *Many client-specific interfaces are better than one general purpose interface*

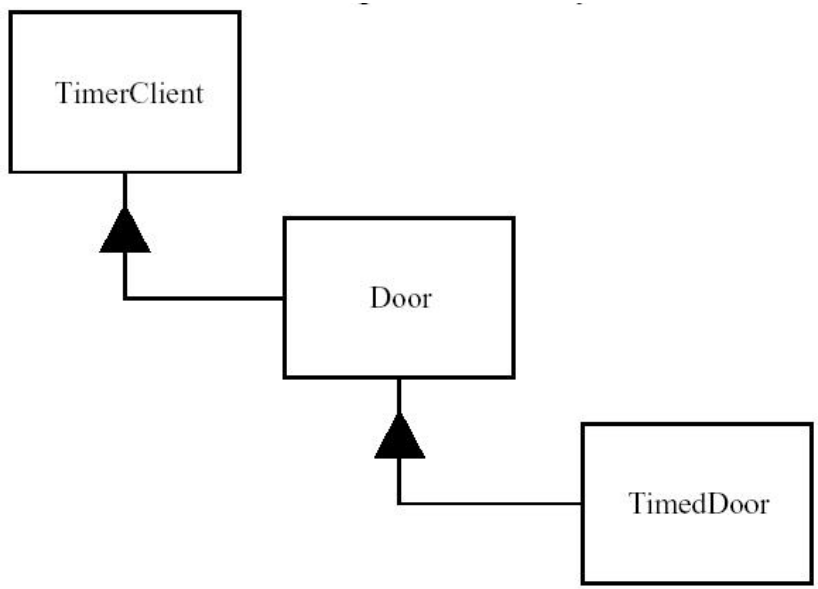
- Consequence:
  - impact of changes to one interface aren't as big if interface is smaller
  - interface pollution

# Interface Segregation Principle (2)



# ISP Example

- Door and Timed Door
- Timer Client



# High-Level Design

- Dealing with *large-scale systems*
  - > 50 KLOC
  - team of developers, rather than an individual

- Classes are a valuable but not sufficient mechanism
  - too *fine-grained* for organizing a large scale design
  - need mechanism that impose a higher level of order
    - *clusters* (Meyer); *class-category* (Booch); *subject-areas* (Coad)

## Packages

- a logical grouping of declarations that can be imported in other programs (in Java and Ada)
- containers for a group of classes (UML)
  - reason at a higher-level of abstraction

# Issues of High-Level Design

## Goal

- *partition* the classes in an application according to some *criteria* and then *allocate* those partitions to packages

## Issues

- What are the best partitioning criteria?
- What principles govern the design of packages?
  - *creation* and *dependencies* between packages
- Design packages first? Or classes first?
  - i.e. *top-down* vs. *bottom-up approach*

## Approach

- Define principles that govern package design
  - the creation and interrelationship and use of packages

# Principles of OO High-Level Design

- Cohesion Principles
  - Reuse/Release Equivalency Principle (REP)
  - Common Reuse Principle (CRP)
  - Common Closure Principle (CCP)
- Coupling Principles
  - Acyclic Dependencies Principle (ADP)
  - Stable Dependencies Principle (SDP)
  - Stable Abstractions Principle (SAP)

# What is really Reusability ?

- Does copy-paste mean reusability?
  - Disadvantage: **You** own that copy!
    - you must change it, fix bugs.
    - eventually the code diverges
  - Maintenance is a nightmare

- Martin's Definition:
  - *I reuse code if, and only if, I never need to look at the source-code*
  - treat reused code like a *product* $\Rightarrow$ don't have to maintain it

- Clients (re-users) may decide on an appropriate time to use a newer version of a component release

# Reuse/Release Equivalency Principle (REP)

> *The granule of reuse is the granule of release. Only components that are released through a tracking system can be efficiently reused.*
>
> **R. Martin**, 1996

- **What means this ?**
  - A reusable software element cannot really be reused in practice unless it is managed by a *release system* of some kind
    - e.g. release numbers or names
  - All related classes must be released together
  - Release granule ≥ Reuse granule
    - no reuse without release
    - must integrate the entire module (can't reuse less)
  - Classes are to small
    - we need larger scale entities, i.e. ***package***

# The Common Reuse Principle

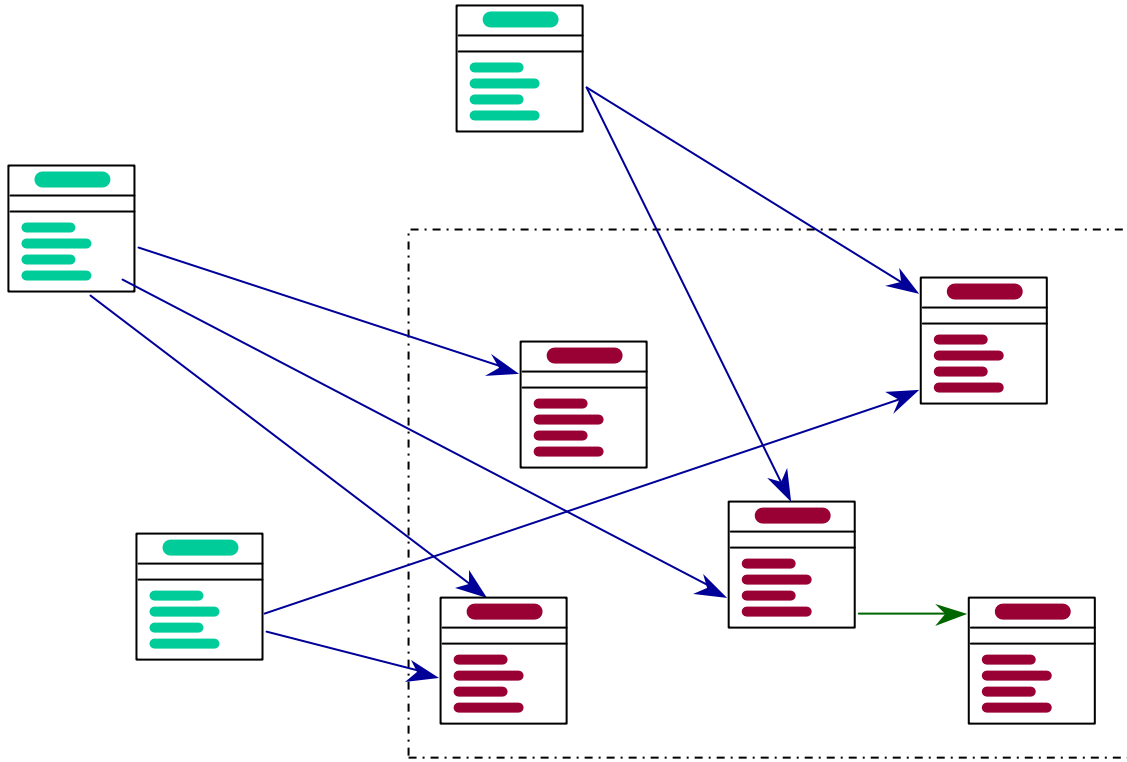
> *All classes in a package [library] should be reused together.*
> *If you reuse one of the classes in the package,*
> *you reuse them all.*
>
> **R. Martin**, *Granularity* 1996

- Packages of reusable components should be grouped by expected usage, Not:
  - common functionality, nor
  - another arbitrary categorization.
- Classes are usually reused in groups based on collaborations between library classes

# Common Reuse in Practice

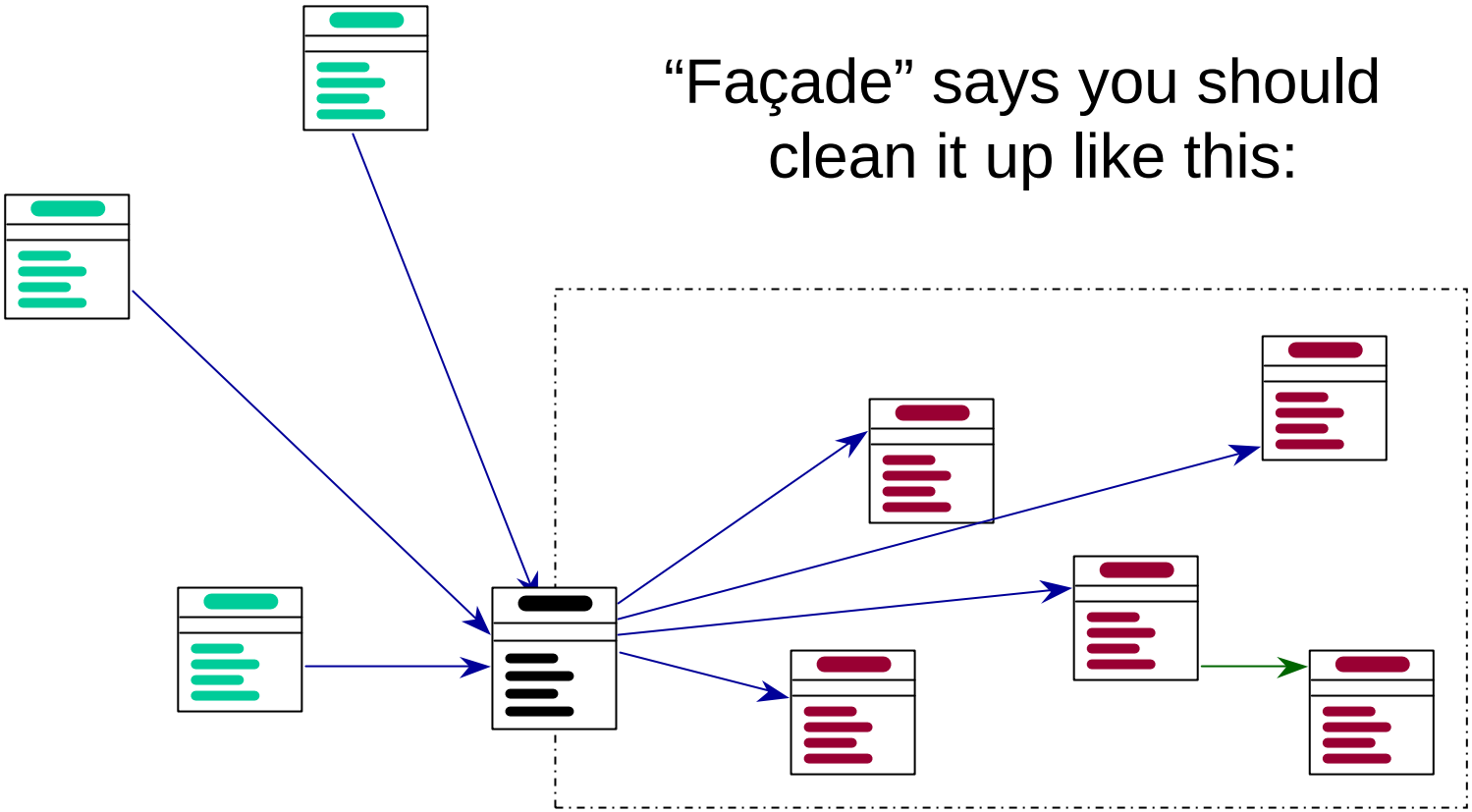
- The Façade Pattern becomes critical when the level of reuse becomes a targeted goal:
- If you started with…

*(e.g. a mess!)*



# The Façade Solution

"Façade" says you should clean it up like this:



# Common Closure Principle (CCP)

*The classes in a package should be closed against the same kinds of changes.*
*A change that affects a package affects all the classes in that package*

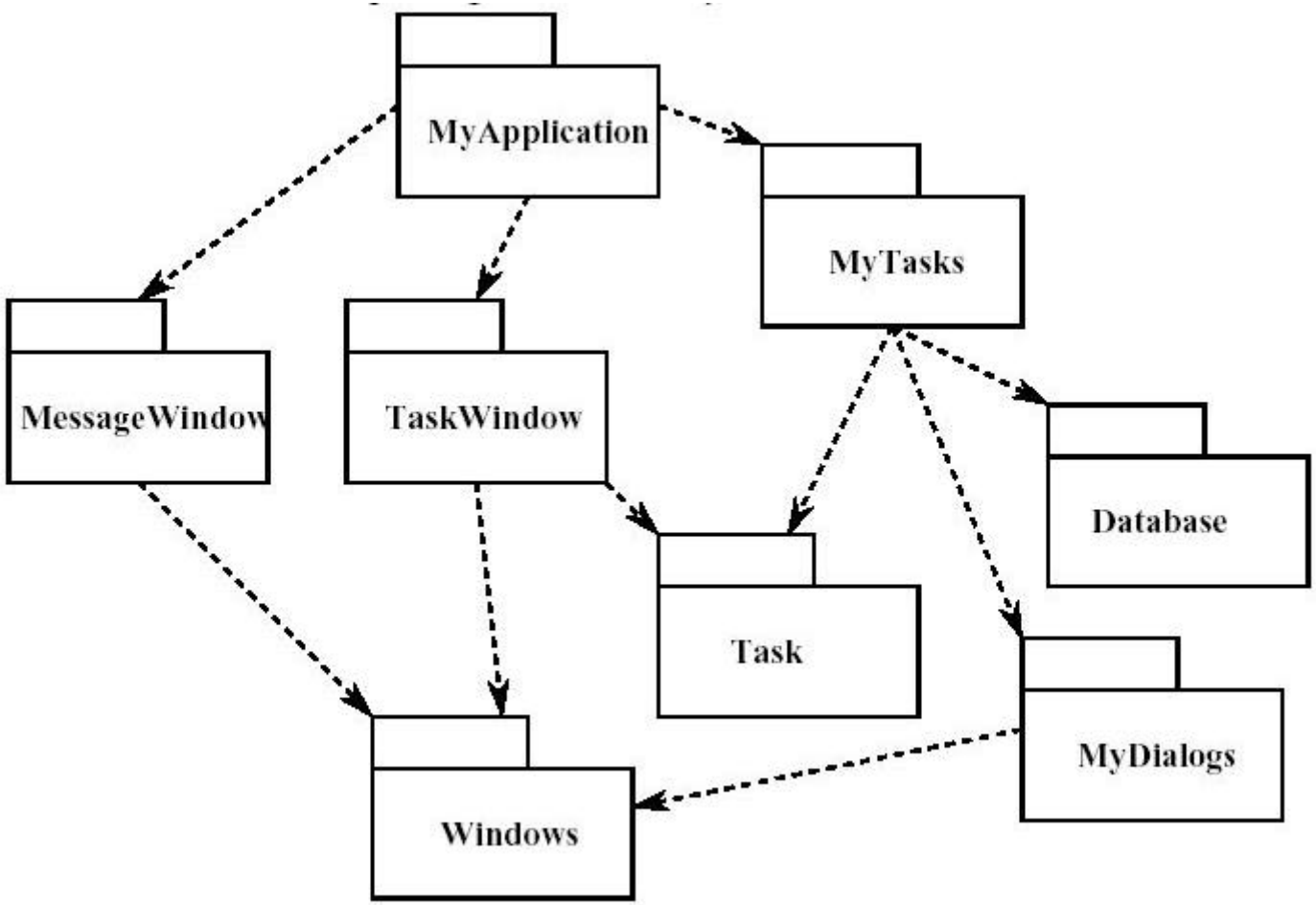
**R. Martin**, 1996

- **What means this ?**
  - *Classes that change together belong together*
  - Goal: limit the dispersion of changes among released packages
    - changes must affect the smallest number of released packages
  - Classes within a package must be cohesive
  - Given a particular kind of change, either all classes or no class in a component needs to be modified

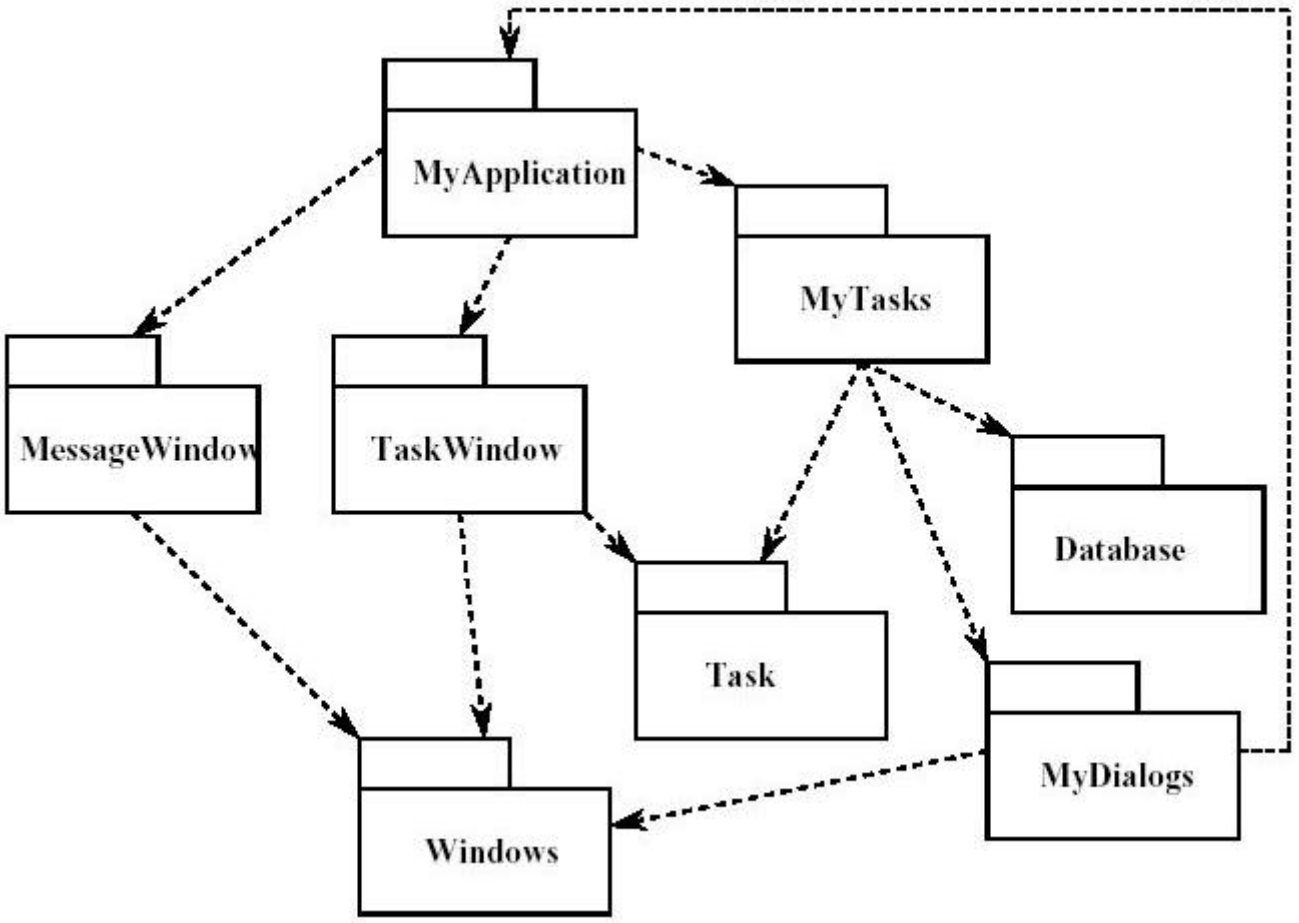
# Reuse vs. Maintenance

- REP and CRP makes life easier for reuser
  - packages very small
- CCP makes life easier for maintainer
  - large packages
- Packages are not fixed in stone
  - early lifetime dominated by CCP
  - later you want to reuse: focus on REP CRP

# Acyclic Graph of Dependencies



# Cyclic Dependencies

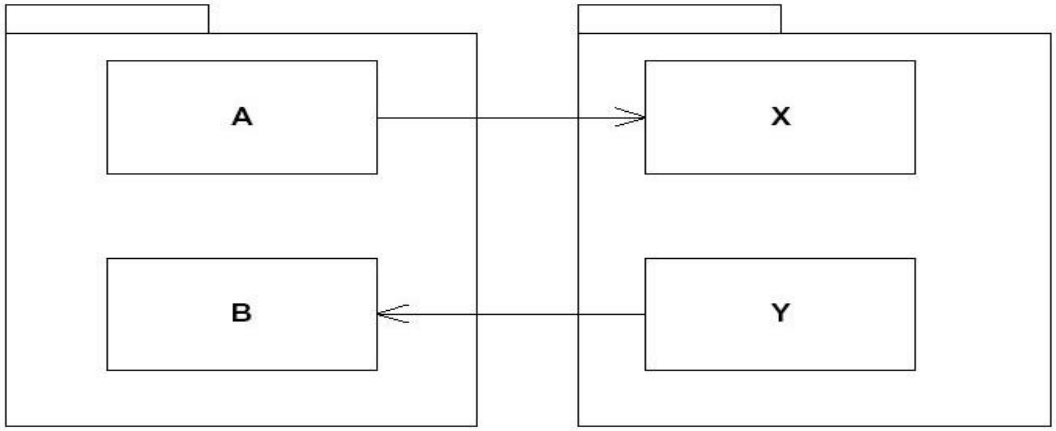


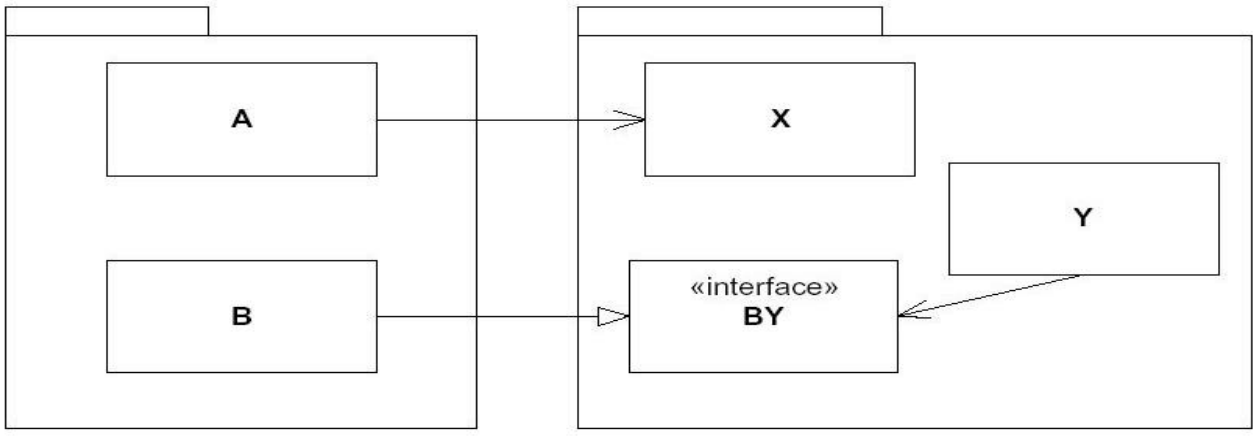
# Acyclic Dependencies Principles (ADP)

*The dependency structure for released component must be a Directed Acyclic Graph (DAG)*
*There can be no cycles.*

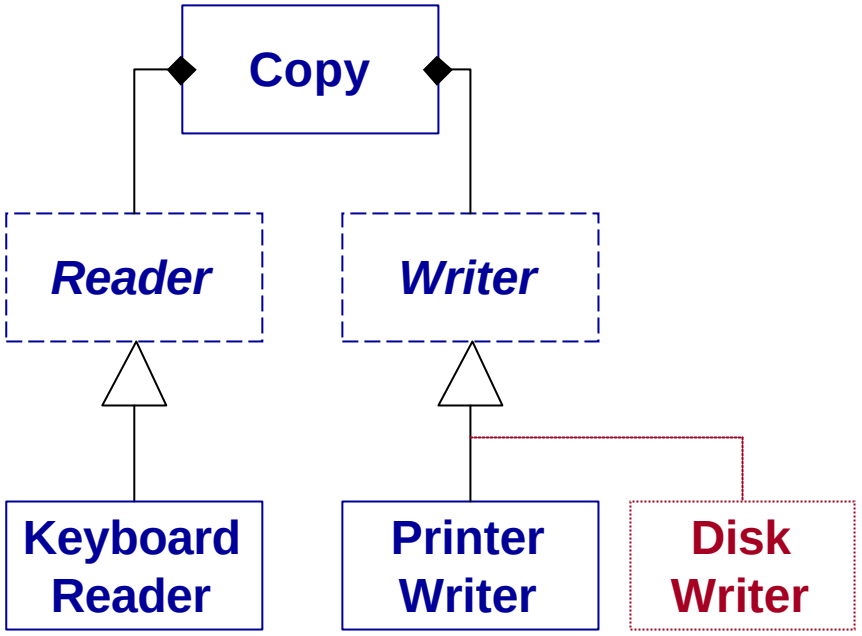
**R. Martin**, 1996

# Breaking the Cycles



# "Copy" Program Revisited




```
class Reader {
  public:
    virtual int read()=0;
};

class Writer {
  public:
    virtual void write(int)=0;
};

void Copy(Reader& r, Writer& w){
   int c;
   while((c = r.read()) != EOF)
     w.write(c);
}
```

# Good Dependencies

- Lack of interdependencies
  - makes the “Copy” program robust, maintainable, reusable
- Targets of unavoidable dependencies are non-volatile
  - unlikely to change
  - e.g. **`Reader`** and **`Writer`** classes have a low volatility $\Rightarrow$ the **`Copy`** class is “immune” to changes

A “Good Dependency” is a dependency upon something with low volatility.

# Volatility and Stability

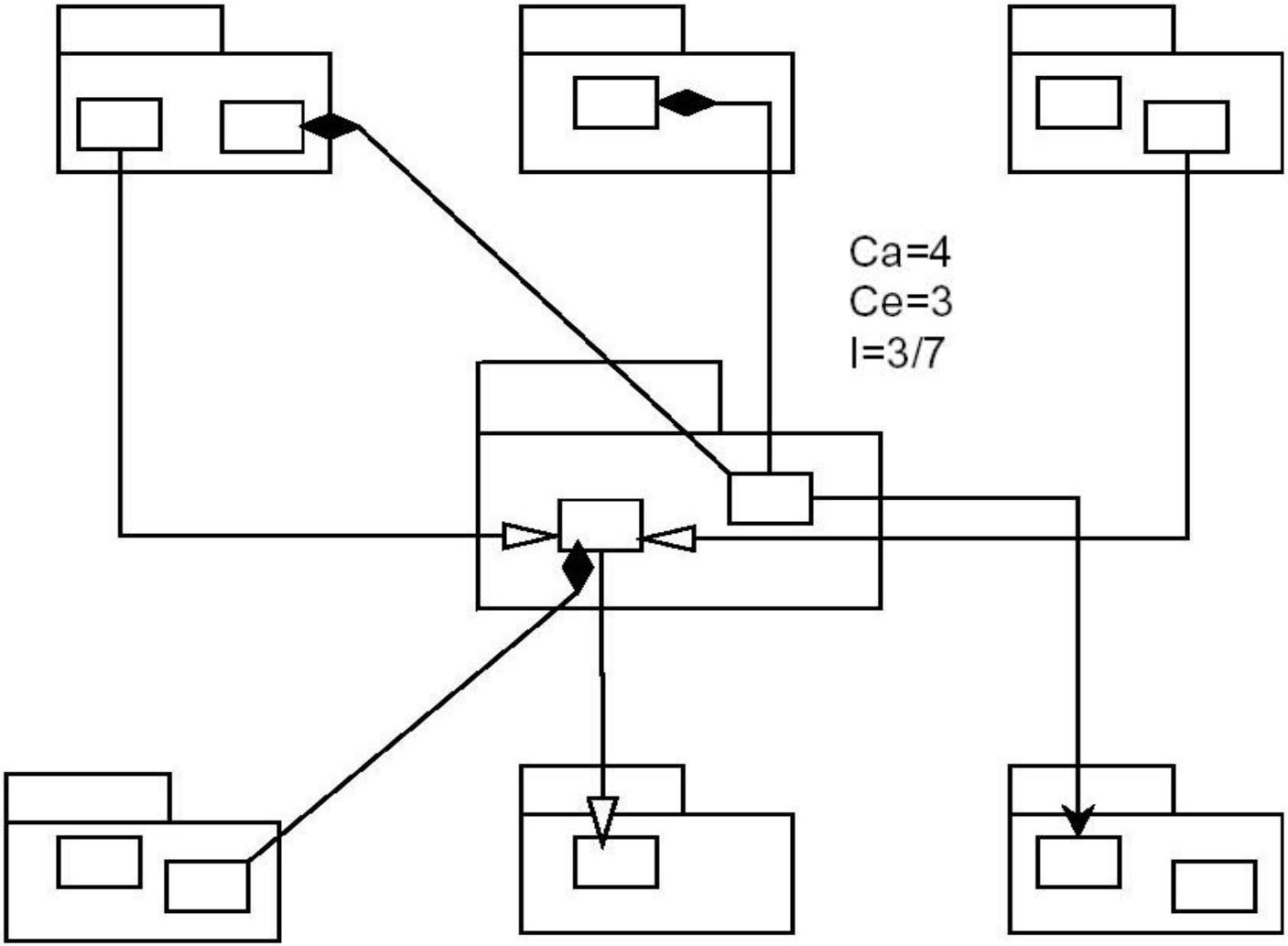
- Volatility Factors
  - hard-coding of improper information
    - e.g. print version number
  - market pressure
  - whims of customers
- Volatility is difficult to understand and predict

- Stability
  - defined as “not easily moved”
  - a measure of the difficulty in changing a module
    - **not** a measure of the likelihood of a change
  - *Stabile = Independent + Responsible*
  - modules that are stable are going to be less volatile
    - e.g. **`Reader`** and **`Writer`** classes are stable

# Stability Metrics

- Afferent Coupling (**$C_a$**)
  - number of classes outside the package that depend upon the measured package
    - "*how responsible am I?*" (FAN-IN)
- Efferent Coupling (**$C_e$**)
  - number of classes outside the package that classes within the measured package depend on
    - "*how dependent am I ?*" (FAN-OUT)
- Instability Factor (**I**) $$I = \frac{C_e}{C_a + C_e}$$
  - $I \in [0, 1]$
  - 0 = totally stable; 1 = totally unstable

# Computing Stability Metrics



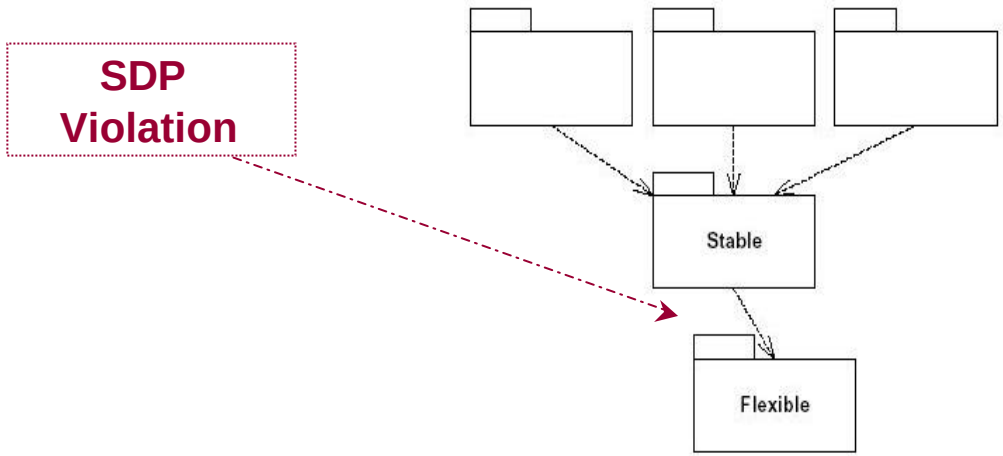
# Stable Dependencies Principle (SDP)

*The dependencies between components in a design should be in the direction of stability.*
*A component should only depend upon components that are more stable than it is.*

**R. Martin**, 1996

Depend only upon components whose ***I*** metric is lower than yours

**R. Martin**, 1996



# Where to Put High-Level Design?

- High-level architecture and design decisions don't change often
  - shouldn't be volatile ⇒ place them in stable packages
  - design becomes hard to change ⇒ *inflexible design*

- How can a totally stable package (I = 0) be flexible enough to withstand change?
  - improve it without modifying it...

- Answer: ***The Open-Closed Principle***
  - classes that can be extended without modifying them

    ⇒ Abstract Classes

# Stable Abstractions Principle (SAP)

*The abstraction of a package should be proportional to its stability!*
*Packages that are maximally stable should be maximally abstract.*
*Instable packages should be concrete.*

**R. Martin**, 1996

- **Ideal Architecture**
  - Instable (changeable) packages on the top
    - must be *concrete*
  - Stable (hard to change) package on the bottom
    - hard to change, but easy to extend
    - highly *abstract* (easily extended)
    - Interfaces have more intrinsic stability than executable code
- SAP is a restatement of DIP

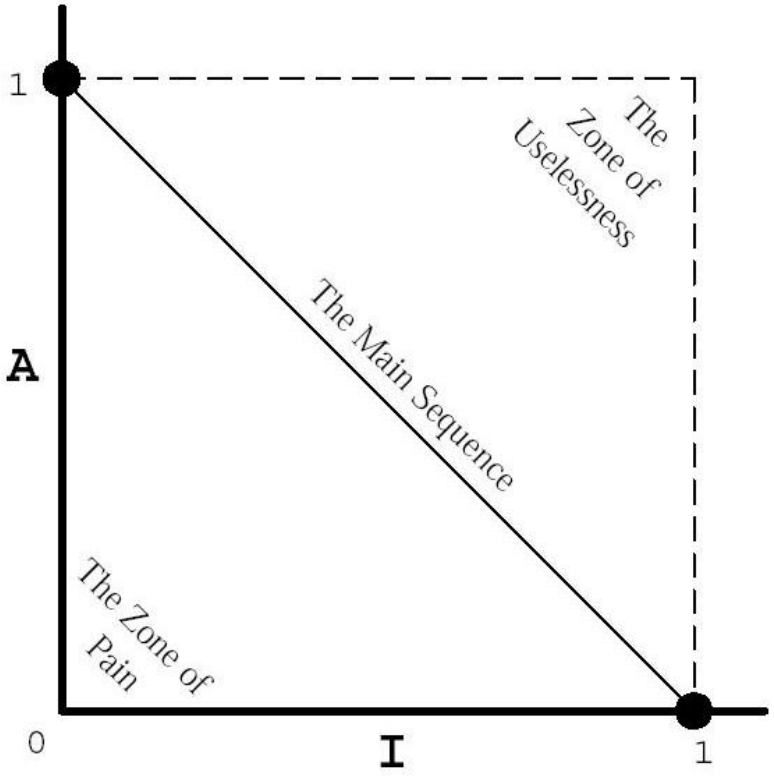
# Measuring Abstraction

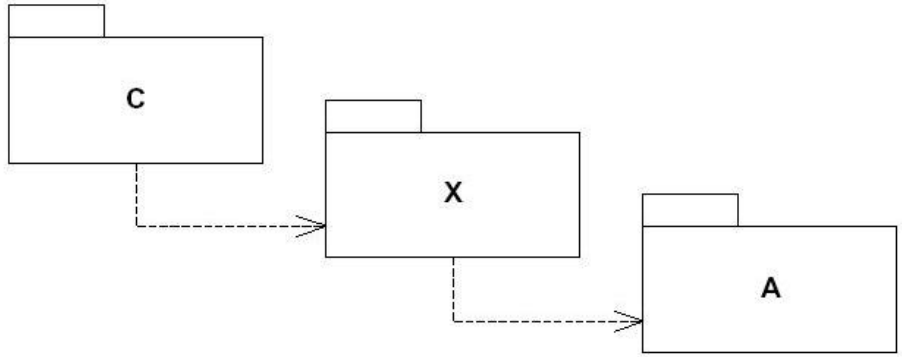
- Abstraction of a Package
  - A ∈ [0, 1]
  - 0 = no abstract classes
  - 1 = all classes abstract

$$A = \frac{NoAbstractClasse}{NoClasses}$$

- Metric ***A*** is imperfect, but usable
  - hybrid classes are also counted as abstract
    - including pure virtual functions
- Alternative
  - find a metric based on the ratio of virtual functions

# The Main Sequence




*Instability* (***I***) should increase as *abstraction* (***A***) decreases.

**R. Martin**, 2000

# Main Sequence (contd.)

- Zone of Pain
  - highly stable and concrete ⇒ rigid
  - famous examples:
    - database-schemas (volatile and highly depended-upon)
    - concrete utility libraries (instable but non-volatile)
- Zone of Uselessness
  - instable and abstract ⇒ useless
    - no one depends on those classes
- Main Sequence
  - maximizes the distance between the zones we want to avoid
  - depicts the balance between abstractness and stability.

# Measurement of OO Architectures

- Distance

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

  - ranges from [0, ~0.707]

- Normalized Distance

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

  - ranges from [0, 1]
  - 0 = package on the main sequence
  - 1 = package far away from main seq.