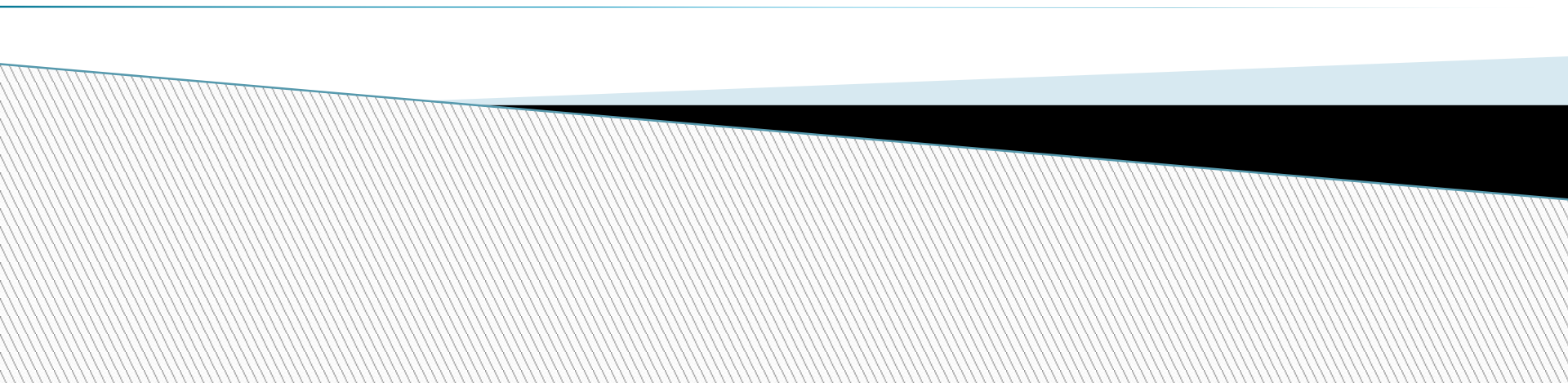


Object Oriented Design Principles



Symptoms of Rotting Design

- ▶ **Rigidity** - Difficulty to change
 - Even small changes take long time
- ▶ **Fragility** – Instability for change
 - Changes break software in many places
- ▶ **Immobility** – Inability to reuse
 - More difficulty in reusing a module than to rewrite
- ▶ **Viscosity** (stickiness) - Easy to do the wrong thing, but hard to do the right thing
 - Viscosity of design – Making a change preserving the design v/s using hacks
 - Viscosity of environment – Slow IDE, Compiler tools

Rigid Software

- ▶ “ Rich Motel”-one-way [proprietary :
Programmers check in but don't check out
[always in need for more efforts, add but not
read].
- ▶ " Management reluctant to allow changes
 - Official rigidity, "don't touch a working system“
 - Users forced to develop workarounds

Fragile Software

- ▶ Small changes have large side effects
 - New bugs appear regularly
 - In the limit of $P(\text{bug}|\text{change}) = 1$ system is impossible to maintain [bugs per change].
- ▶ It looks like control has been lost
 - Users become critical
 - Program loses credibility
 - Developers lose credibility

Immobility

- ▶ You have a problem and find some piece of code which might solve it
 - But it brings in a lot of other stuff
 - It needs changes here and there
- ▶ Eventually you have two choices
 - Take over maintenance of the branched code
 - Roll your own
- ▶ " You would like to link a library which is
 - maintained by somebody else

High Viscosity


- ▶ Viscosity of the design
 - Hard to make changes properly, i.e. without breaking the design (make hacks, improper fixes, instead).
- ▶ Viscosity of the environment
 - Slow and inefficient development environment
 - Large incentive to keep changes localized even if they break designs
 - Design changes are very difficult

Root cause to problems

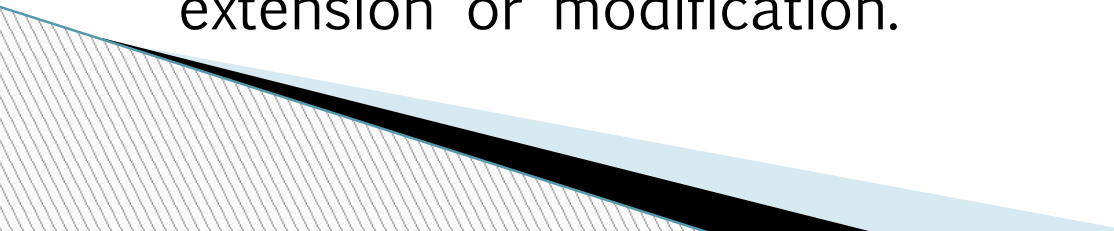
- ▶ Changes that introduce new or unplanned-for dependencies
 - Improper dependencies between modules of the software
- ▶ Need to manage dependencies and create dependency firewall

General OO Design Principles:

1. Encapsulate what varies:

- ▶ Identify the aspects of your application that vary and separate them from what stays the same.
 - ▶ Take the parts that vary and encapsulate them, so that later you can alter or extend the parts that vary without affecting the parts that don't.
 - ▶ Reduces the chances of breaking some other part of your app when you make a change.
- 

2. Program to an interface, not an implementation

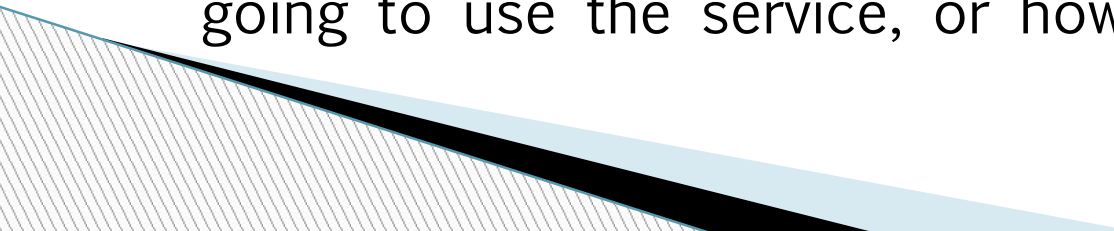
- ▶ Client app is decoupled from knowing the details of the concrete implementation.
 - ▶ If you see “As New” (or = New), you are creating a concrete implementation.
 - ▶ First higher level should be abstract (i.e. interfaces while lower level implementations should be concrete (i.e. classes).
 - ▶ Interfaces are more generic which allows for easy extension or modification.
- 

3. Implementation Inheritance vs Interface Inheritance

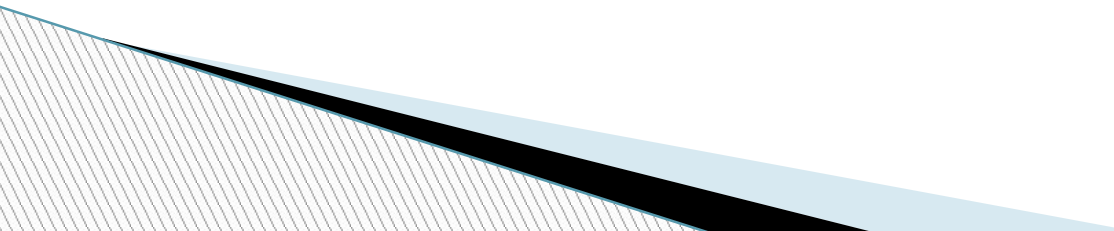
- ▶ *Implementation Inheritance (Class Inheritance)* - an object's implementation is defined in terms of another's objects implementation
- ▶ *Interface Inheritance (Subtyping)* - describes when one object can be used in place of another object
- ▶ The C++ inheritance mechanism means both class and interface inheritance
- ▶ C++ can perform interface inheritance by inheriting from a pure abstract class
- ▶ Java has a separate language construct for interface inheritance the Java interface
- ▶ Java's interface construct makes it easier to express and implement designs that focus on object interfaces

4. Favor Composition over Inheritance

- ▶ “Has A” vs. “Is A”
 - Generally done by keeping an instance of another class within your class, and using its properties and methods.
- ▶ You can only inherit from one class, you can compose a class of as many other classes as you want [Less complications].
- ▶ Lets you add functionality to a class without changing the class.
- ▶ Method of reuse in which new functionality is obtained by creating an object *composed of other objects*

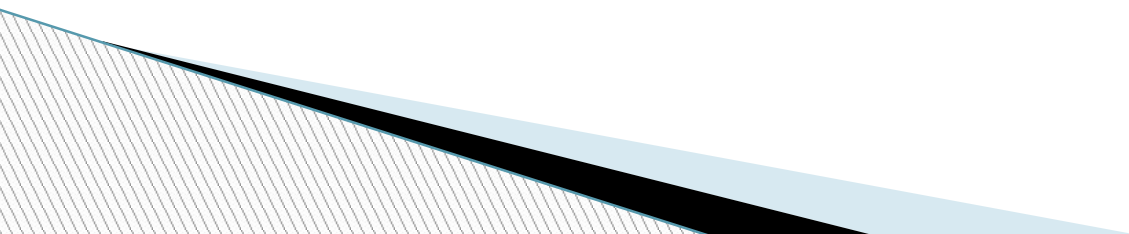
- ▶ The new functionality is obtained by delegating functionality to one of the objects being composed
 - ▶ Sometimes called *aggregation or containment*, *although some* authors give special meanings to these terms. For example:
 - Aggregation - when one object owns or is responsible for another object and both objects have identical lifetimes (GoF)
 - ▶ Strive for loosely coupled designs between objects that interact:
 - Lets you write code for a service without knowing what is going to use the service, or how it is going to use it.
- 

5. Principle of least knowledge – talk only to your immediate friends

- ▶ The more classes your classes interact with, the more fragile, and hard to understand it becomes.
 - ▶ Have your classes only talk to:
 - Their own methods
 - Objects passed in as parameters
 - Objects the object creates
 - Components of the object
- 

6. Hollywood Principle – Don't call us, we'll call you

- ▶ When a high level object instantiates a low level object, the low level object just does what it is asked to do. It does not ask the high level object to do anything.



Class Design Principles

- ▶ Open/Closed Principle (OCP)
- ▶ Single Responsibility Principle (SRP)
- ▶ Liskov Substitution Principle (LSP)
 - a.k.a. Design by Contract
- ▶ Dependency Inversion Principle (DIP)
- ▶ Interface Segregation Principle (ISP)

Open/Closed Principle (OCP)

Modules should be **open** for extension,
but **closed** for modification

Bertrand Meyer

Object Oriented Software Construction

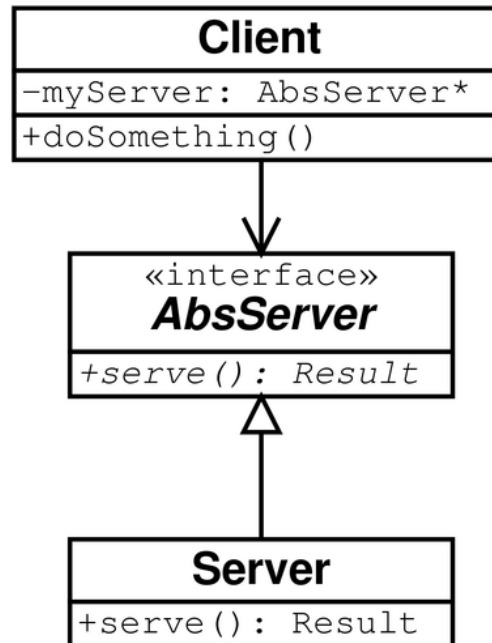
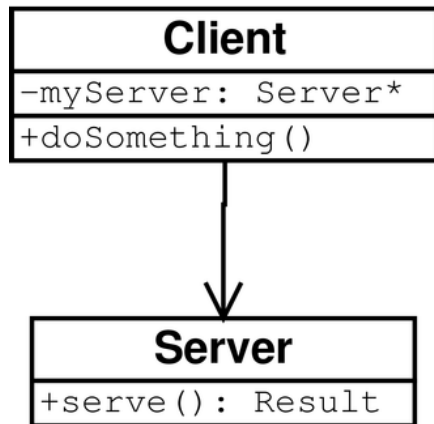
Change what the modules do, without changing the source code of the modules

Module: Class, Package, Function

New functionality → new code, existing code remains unchanged

"Abstraction is the key" → cast algorithms in abstract interfaces
develop concrete implementations as needed

Abstraction and OCP



Client is **closed** to changes
in implementation of **Server**

Client is **open** for extension
through new **Server**
implementations

Without **AbsServer** the **Client**
is **open** to changes in **Server**

The Shape Example - Procedural

Shape.h

```
enum ShapeType { isCircle, isSquare };  
typedef struct Shape {  
    enum ShapeType type  
} shape;
```

Circle.h

```
typedef struct Circle {  
    enum ShapeType type;  
    double radius;  
    Point center;  
} circle;  
void drawCircle( circle* );
```

Square.h

```
typedef struct Square {  
    enum ShapeType type;  
    double side;  
    Point topleft;  
} square;  
void drawSquare( square* );
```

drawShapes.c

```
#include "Shape.h"  
#include "Circle.h"  
#include "Square.h"
```

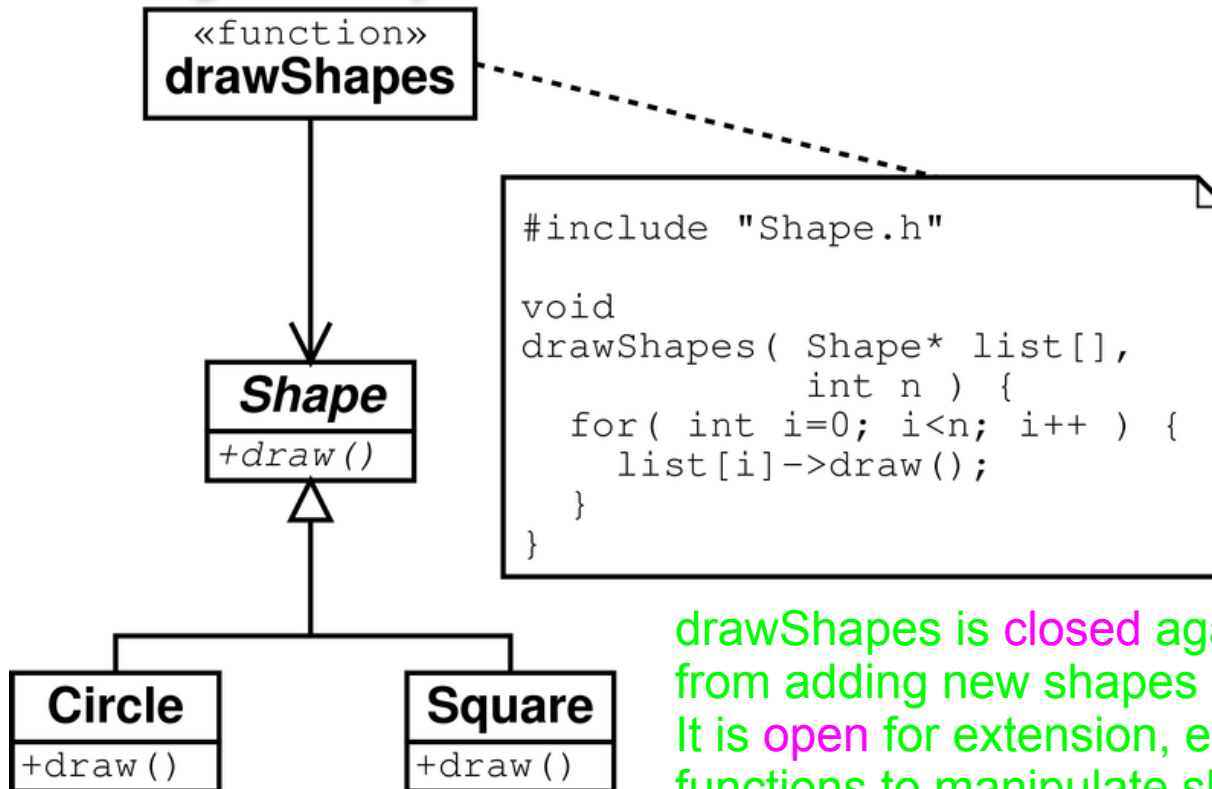
```
void drawShapes( shape* list[], int n ) {  
    int i;  
    for( int i=0; i<n; i++ ) {  
        shape* s= list[i];  
        switch( s->type ) {  
            case isSquare:  
                drawSquare( (square*)s );  
                break;  
            case isCircle:  
                drawCircle( (circle*)s );  
                break;  
        }  
    }  
}
```

C: Adding a new shape requires many changes

Problems with Procedural Implementation

- ▶ drawShapes is not closed
 - switch/case probably needed in several places
 - Adding a shape → modify switch/case
 - There may be many and the logic may be more complicated
 - Extending enum ShapeType → rebuild everything
- ▶ Rigid, fragile, highly viscous

The Shape Example OO – Dynamic Polymorphism



drawShapes is **closed** against changes from adding new shapes
It is **open** for extension, e.g. adding new functions to manipulate shapes

Just add new shapes or functions and relink

The Shape Example OO – Static Polymorphism

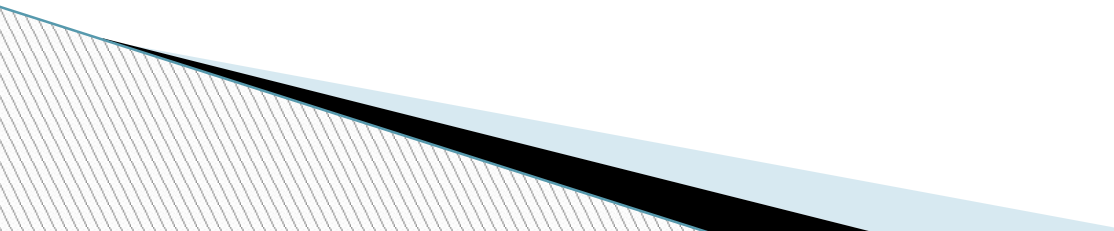
- ▶ Using Generics / template

```
template <typename SHAPE>
void draw(SHAPE& s)
{
    s.draw();
}
```

OCP Summary

- ▶ Open for extension
 - Add new code for new functionality, don't modify existing working code
 - Concrete implementations of interfaces somewhere
- ▶ Closed for modification
 - Need to anticipate likely modifications to be able to plan ahead in the design
 - e.g. ordering shapes? No closure against this requirement ... but could be added in a design-preserving way (low viscosity)

OCP How-To

- ▶ How is the system going to evolve?
 - ▶ How will its environment change?
 - ▶ Isolate against *kinds of changes*, e.g.
 - *database schema (data model)*
 - *hardware changes (sensors, ADCs, TDCs, etc)*
 - *data store technology (e.g. Objectivity vs ROOT)*
 - ▶ Plan ahead, but don't implement what is not already needed
- 

Liskov Substitution Principle (LSP)

All derived classes must be substitutable
for their base class

Barbara Liskov, 1988

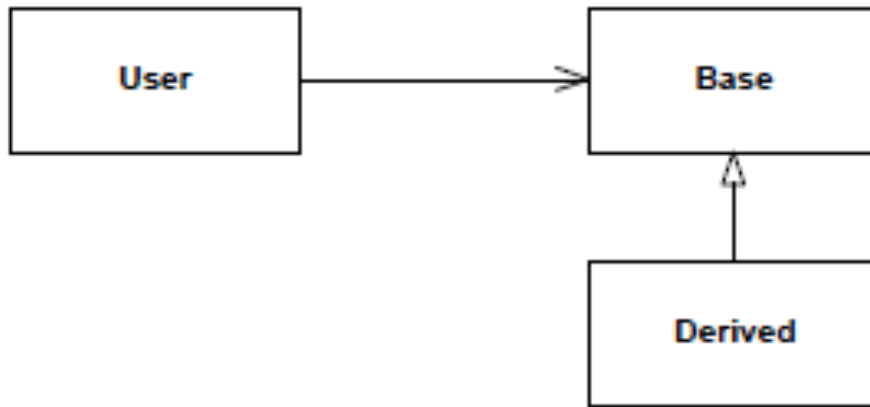
The "Design-by-Contract" formulation:

All derived classes must honor the contracts
of their base classes

Bertrand Meyer

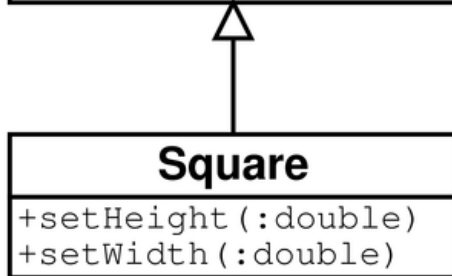
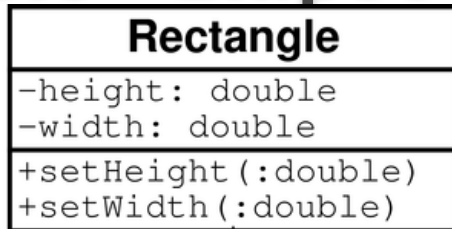


LSP



User of a base class should continue to function properly if a derivative of that base class is passed to it

The Square-Rectangle Problem or Circle- Ellipse Problem



Clients (users) of Rectangle expect that setting height leaves width unchanged (and vice versa)

Square does not fulfill this expectation
Client algorithms can get confused

Hack: attempt to identify subclasses and use if/switch

```
void Square::setHeight( double h ) {  
    Rectangle::setHeight( h );  
    Rectangle::setWidth( h );  
}  
void Square::setWidth( double w ) {  
    setHeight( w );  
}
```

↑
This is evil!

Pre-condition & Post-condition

▶ Pre-condition

- what must be true before the method is called
- If the precondition fails, the results of the method are undefined, and the method ought not be called

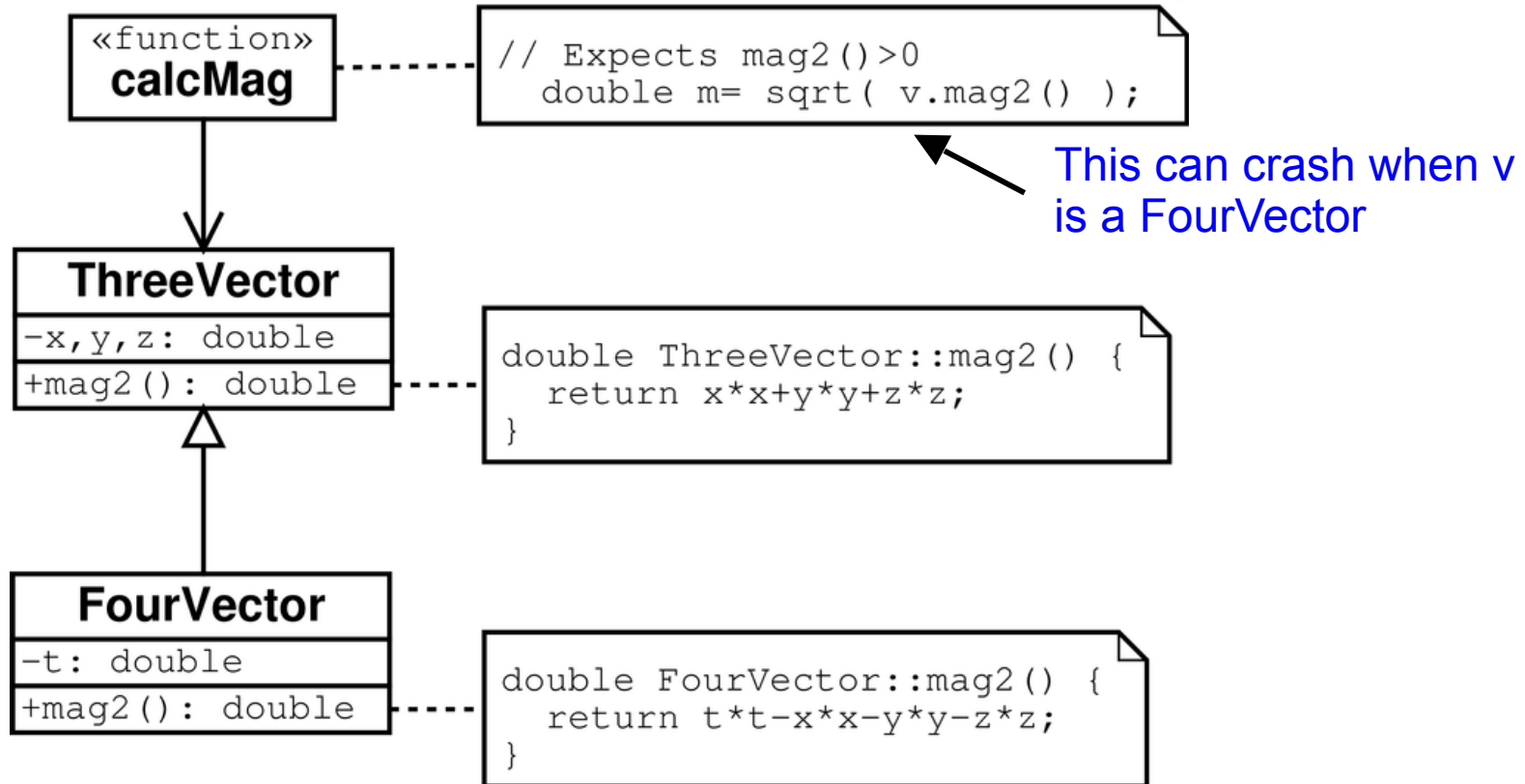
▶ Post-condition

- what the method guarantees will be true once it has completed
- method that fails its post-condition should not return

Contract Violation

- ▶ The contract of Rectangle
 - height and width are independent, can set one while the other remains unchanged, $\text{area} = \text{height} * \text{width}$
- ▶ Square breaks this contract
- ▶ Derived methods should not expect more and provide no less than the base class methods
 - Pre-conditions are not stronger than base
 - Post-conditions are not weaker than base

The FourVector Example



A 4-vector IS-A 3-vector with a time-component? Not in OO,
4-vector has different algebra → can't fulfill 3-vector contracts

LSP Summary

- ▶ Derived classes must fully substitute base class
 - Guides design and choice of abstractions
- ▶ Good abstractions are not always intuitive
- ▶ Violating LSP may break OCP
 - Need RTTI (Run Time Type Information) and if/switch
-> lost closure
- ▶ Inheritance/polymorphism powerful tools
 - Use with care
- ▶ “is a” relation really means behaviour

Dependency Inversion Principle (DIP)

Depend upon Abstractions. Do not depend upon concretions.

(or)

Details should depend on abstractions.
Abstractions should not depend on details.

Robert Martin

Why *dependency inversion*? In OO we have ways to invert the direction of dependencies, i.e. class inheritance and object polymorphism

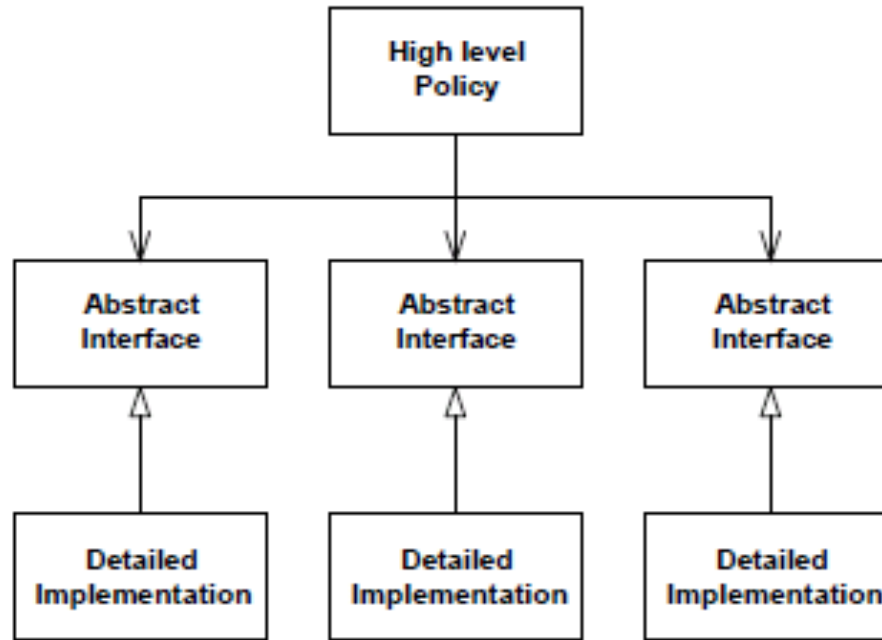
If the OCP states the goal of OO architecture, the DIP states the primary mechanism. Behind COM, CORBA, EJB, etc



DIP

- ▶ The modules that contain detailed implementation are no longer depended upon, rather they depend themselves upon abstractions. Thus the dependency upon them has been inverted
- ▶ The DIP makes the assumption that anything concrete is volatile. Exceptions – ex. C String library
 - Depending upon it is not harmful. What if you want to change to unicode
- ▶ Non-volatility is not a replacement for the substitutability of an abstract interface.

Dependency Structure of a OO Architecture

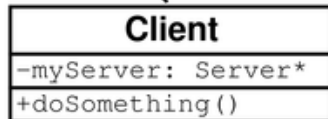


- ▶ Concrete things change a lot, abstract things change much less frequently.
- ▶ Abstractions are “hinge points”, they represent the places where the design can bend or be extended, without themselves being modified (OCP).

DIP Example

```
#include "Server.hh"

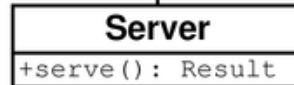
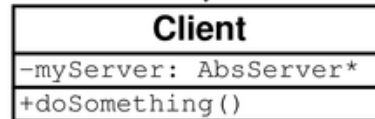
void Client::doSomething() {
    ...
    Result r= myServer->serve();
    ...
}
```



```
// no #include
Result Server::serve() {
    ...
    return Result;
}
```

```
#include "AbsServer.hh"

void Client::doSomething() {
    ...
    Result r= myServer->serve();
    ...
}
```



```
#include "AbsServer.hh"

Result Server::serve() {
    ...
    return Result;
}
```

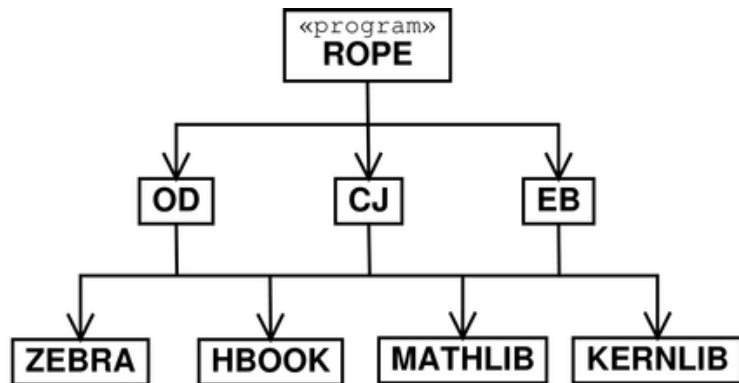
Dependency changed from concrete to abstract ...

The abstract class is unlikely to change

```
class AbsServer {
public:
    virtual
    Result serve() = 0;
}
```

... at the price of a dependency here, but it is on an abstraction. Somewhere a dependency on concrete Server must exist, but we get to choose where.
-- ABSTRACT FACTORY

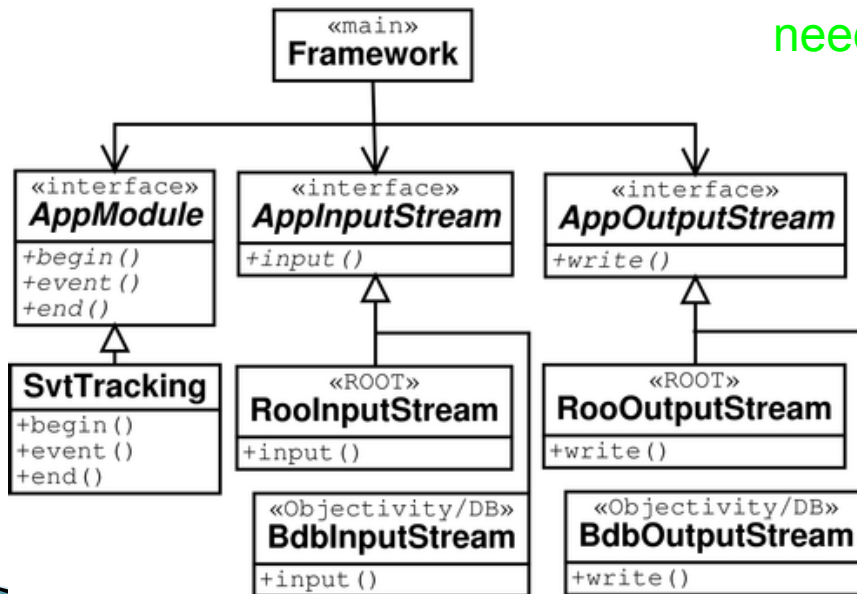
DIP and Procedural Design



Procedural:

Call more concrete routines
Dependence on (reusable)
concrete modules


In reality the dependencies are cyclic →
need multi-pass link and a "dummy library"



Framework classes
depend on interfaces

Can e.g. change data store
technology without disturbing
the Framework classes

DIP Summary

- ▶ Use DI to avoid
 - deriving from concrete classes
 - associating to or aggregating concrete classes
 - dependency on concrete components
 - ▶ Encapsulate invariants (generic algorithms)
 - Abstract interfaces don't change
 - Concrete classes implement interfaces
 - Concrete classes easy to throw away and replace
 - ▶ Foundation classes (STL, CLHEP, MFC, etc)?
- 

Interface Segregation Principle (ISP)

Many client specific interfaces are better than one general purpose interface

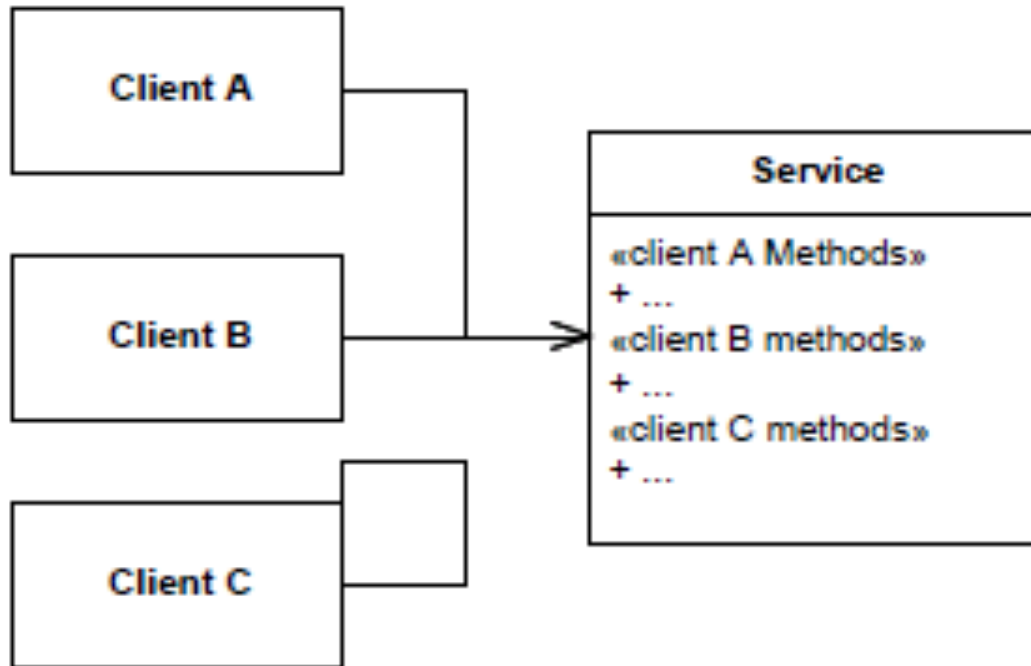
Clients should not be forced to depend upon interfaces they don't use

- 1) High level modules should not depend on low level modules. Both should depend upon abstractions (interfaces)
- 2) Abstractions should not depend upon details. Details should depend on abstractions.

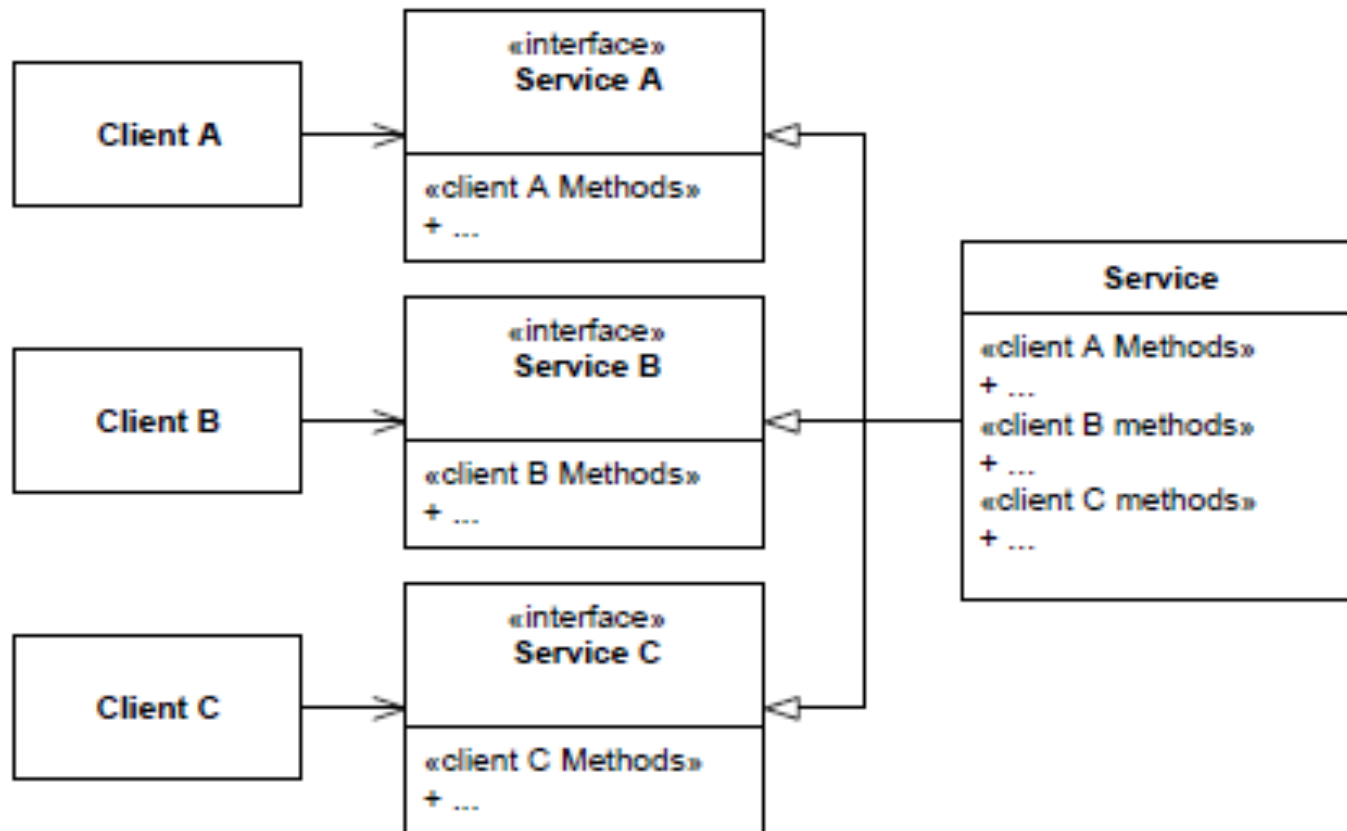
Robert Martin



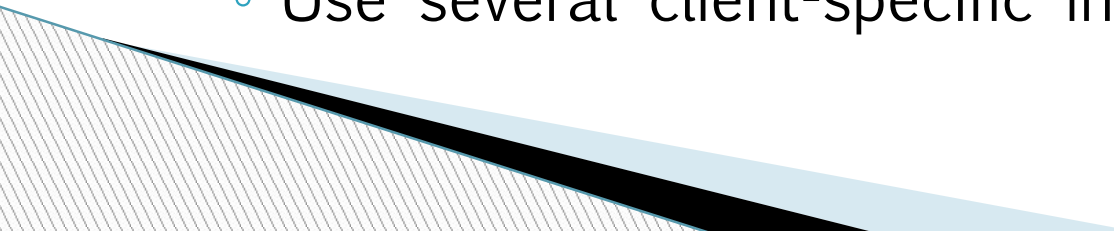
Fat Service with Integrated Services



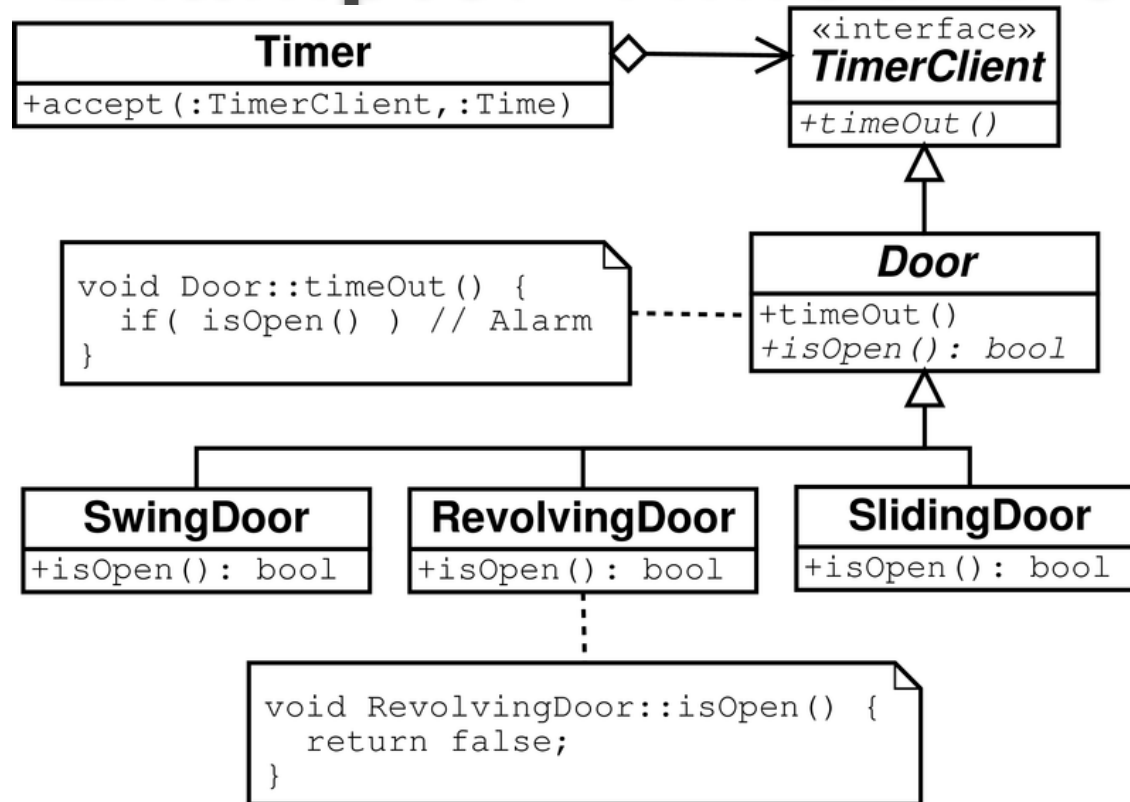
Segregated Interfaces



ISP Explained

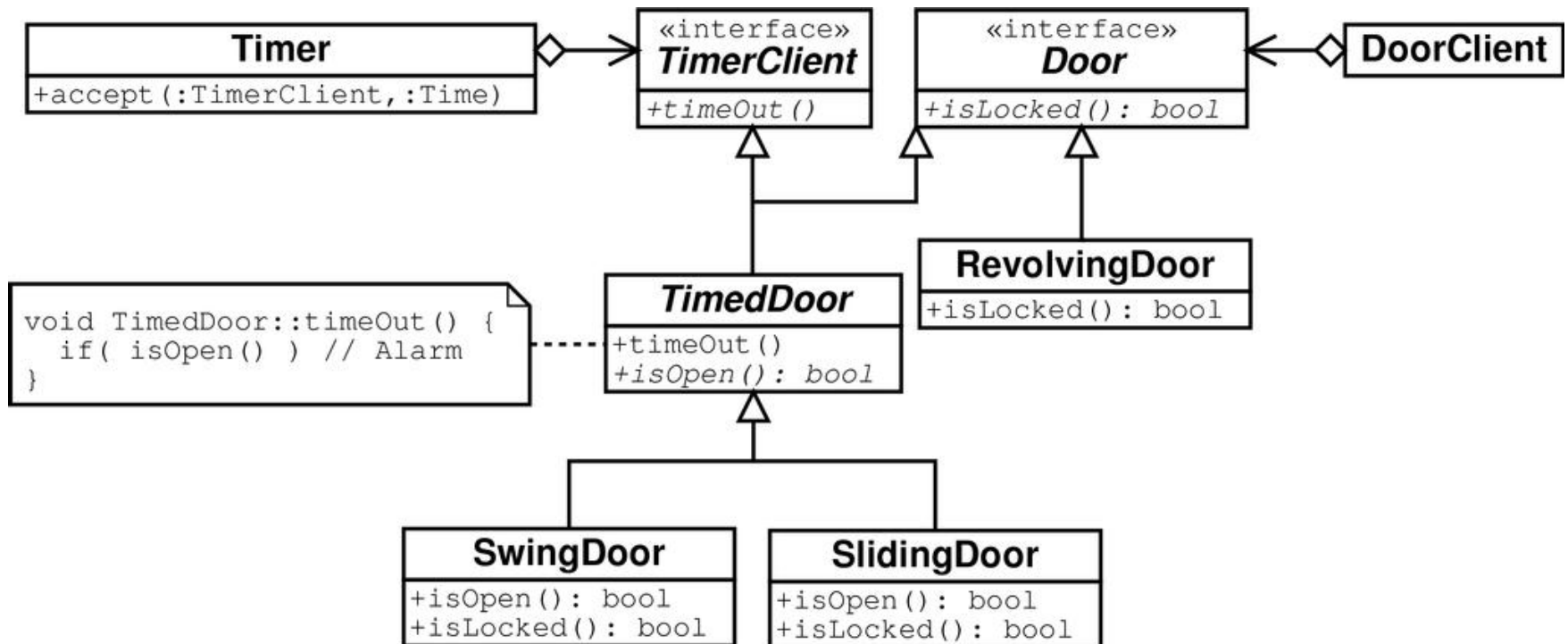
- ▶ Multipurpose classes
 - Methods fall in different groups
 - Not all users use all methods
 - ▶ Can lead to unwanted dependencies
 - Clients using one aspect of a class also depend indirectly on the dependencies of the other aspects
 - ▶ ISP helps to solve the problem
 - Use several client-specific interfaces
- 

ISP Example: Timed Door



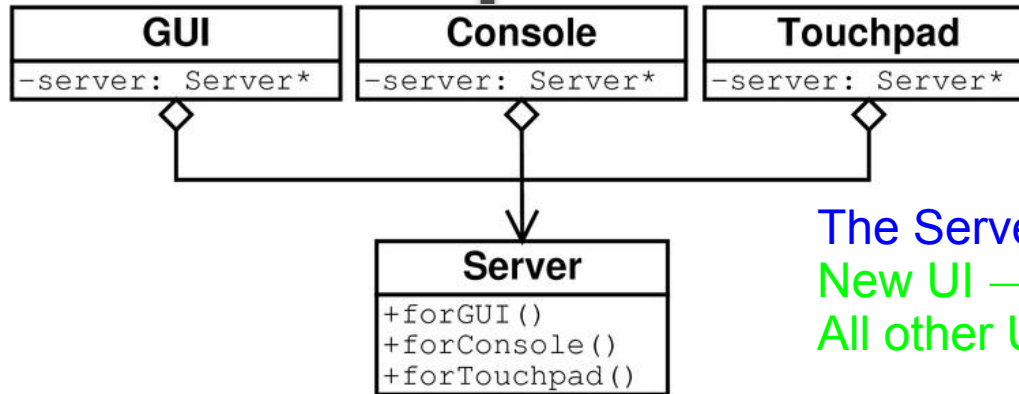
There may be derived classes of Door which don't need the TimerClient interface. They suffer from depending on it anyway.

Timed Door ISP

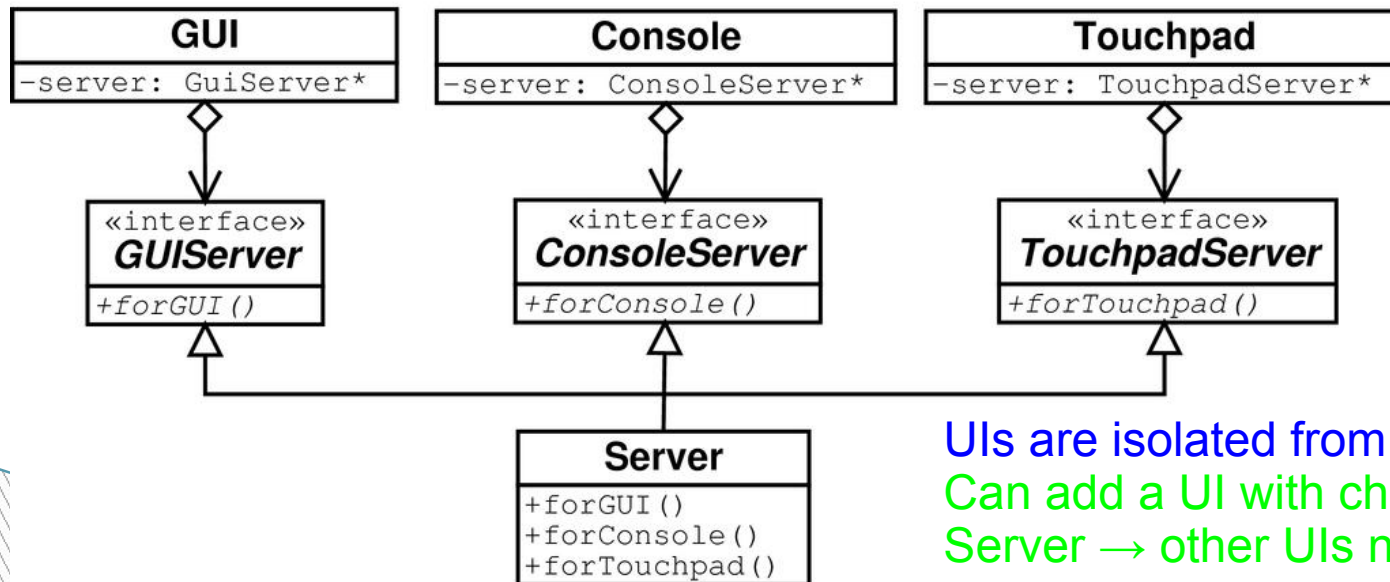


RevolvingDoor does not depend needlessly on TimerClient
SwingDoor and SlidingDoor really are timed doors

ISP Example: UIs



The Server "collects" interfaces
New UI → Server interface changes
All other UIs recompile



UIs are isolated from each other
Can add a UI with changes in
Server → other UIs not affected

ISP Summary

- ▶ When a class (Server) collects interfaces for various purposes (Clients) → fat interface
 - Use separate interfaces to hide parts of the Server interface for Clients
 - Similar to data hiding
 - Or split the Server in several parts
- ▶ Be careful with vertical multiple inheritance
 - You might drag in dependencies you don't want/need/like

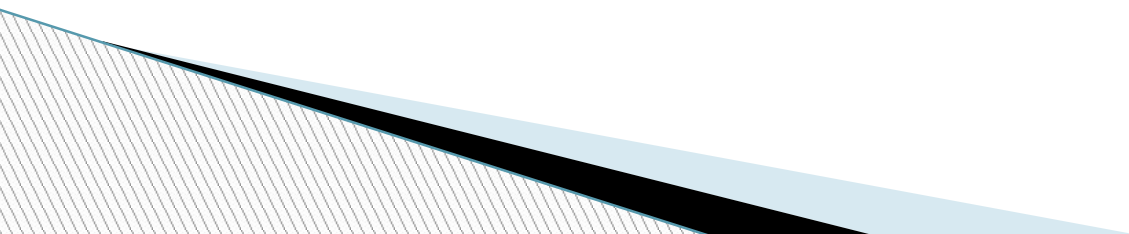
Single Responsibility Principle (SRP)

A class should have only one reason to change

Robert Martin

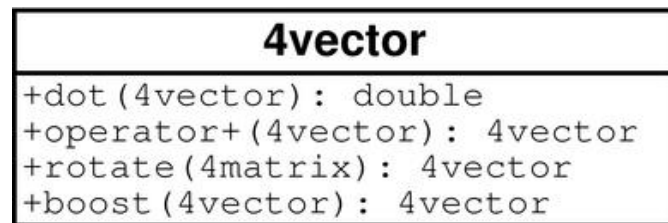
Related to and derived from *cohesion*, i.e. that elements in a module should be closely related in their function

Responsibility of a class to perform a certain function is also a reason for the class to change



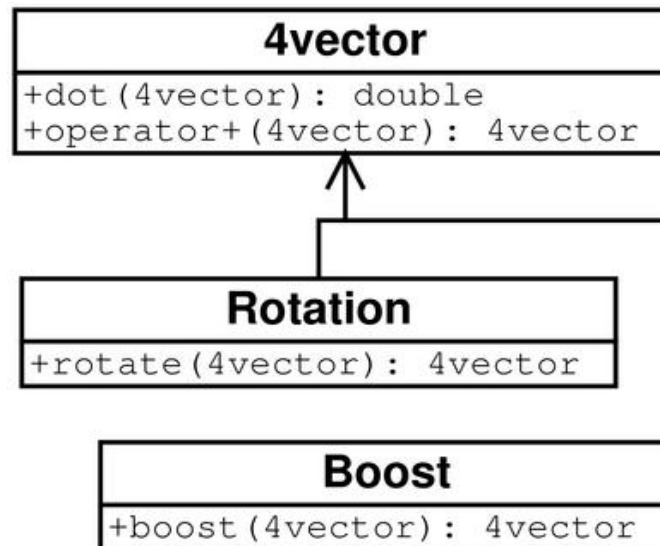
SRP Example

All-in-one wonder



Always changes to 4vector

Separated responsibilities



Changes to rotations or boosts
don't impact on 4vector

SRP Example

// single responsibility principle - bad example

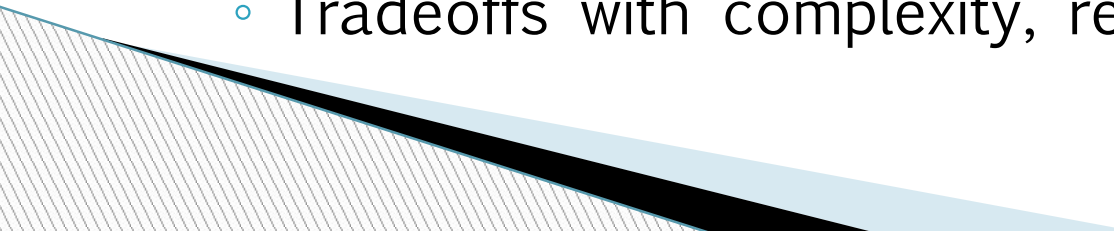
```
interface IEmail {  
    public void setSender(String sender);  
    public void setReceiver(String receiver);  
    public void setContent(String content);  
}  
  
class Email implements IEmail {  
    public void setSender(String sender) { // set sender; }  
    public void setReceiver(String receiver) { // set receiver; }  
    public void setContent(String content) { // set content; }  
}
```

SRP Example

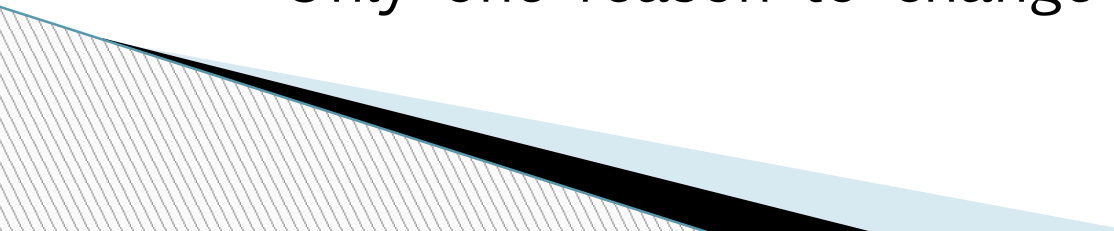
// single responsibility principle - good example

```
interface IEmail {  
    public void setSender(String sender);  
    public void setReceiver(String receiver);  
    public void setContent(IContent content);  
}  
  
interface Content {  
    public String getAsString(); // used for serialization  
}  
  
class Email implements IEmail {  
    public void setSender(String sender) { // set sender; }  
    public void setReceiver(String receiver) { // set receiver; }  
    public void setContent(IContent content) { // set content; }  
}
```


SRP Summary

- ▶ Class should have only one reason to change
 - Cohesion of its functions/responsibilities
 - ▶ Several responsibilities
 - mean several reasons for changes → more frequent changes
 - ▶ Sounds simple enough
 - Not so easy in real life
 - Tradeoffs with complexity, repetition, opacity
- 

Class Design Principles Summary

- ▶ Open-Closed Principle (OCP)
 - Extend functionality with new code
 - ▶ Liskov Substitution Principle (LSP)
 - Derived classes fully substitute their base classes
 - ▶ Dependency Inversion Principle (DIP)
 - Depend on abstractions, not details
 - ▶ Interface Segregation Principle (ISP)
 - Split interfaces to control dependencies
 - ▶ Single Responsibility Principle
 - Only one reason to change
- 

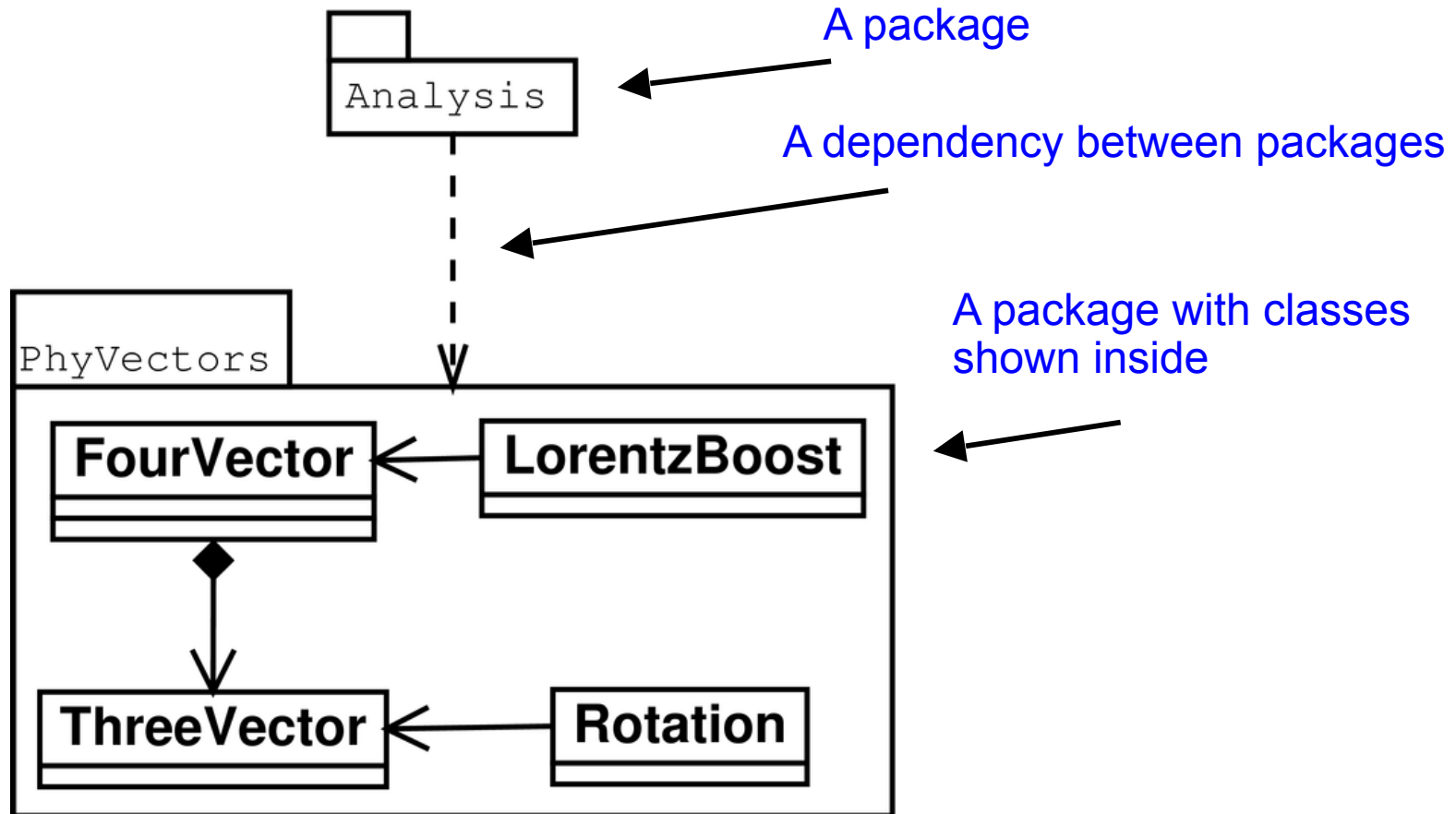
Packages Introduction

- ▶ What is a package?
 - Classes are not sufficient to group code
 - Some classes collaborate
 - Some don't know each other
- ▶ Grouping related classes together seems natural
 - But how?
 - Dependencies between packages

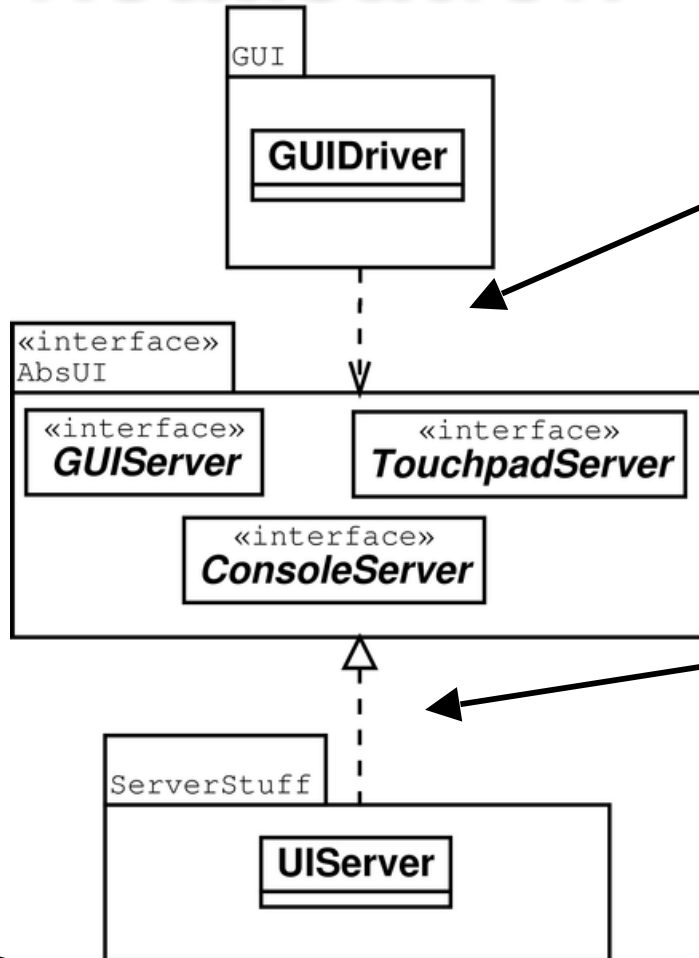
Package

- ▶ A package is a group of classes
- ▶ Classes in a package are often compiled together into a library
 - but unit of compilation is mostly individual class
- ▶ A package is a unit for testing
- ▶ A package can be a releasable component
 - a CVS module

Packages in UML



Realisation



GUI depends on AbsUI

Associations exist between classes in GUI and AbsUI

AbsUI is an abstract package

ServerStuff realises AbsUI, it is a concrete package

An inheritance relationship exists between classes in AbsUI and ServerStuff

Package Principles

▶ Granularity

- Growing software size necessitates something “larger” than a class to help organize applications
- “Class category”, “clusters”, “subject areas”, “domains” – “packages”

Package Principles

▶ Stability

- Stability is not a measure of the likelihood that a module will change; rather it is a measure of the difficulty in changing a module
- The most stable classes of all are classes that are both Independent and Responsible. Such classes have no reason to change, and lots of reasons not to change.

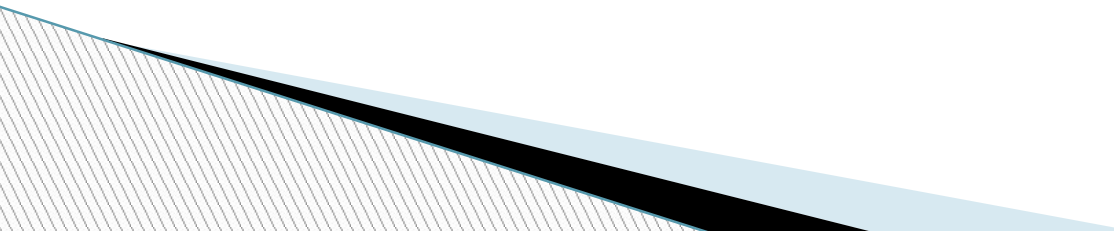
Designing with Packages

- ▶ What are the best partitioning criteria?
- ▶ What are the relationships that exist between packages, and what design principles govern their use?
- ▶ Should packages be designed before classes (Top down)? Or should classes be designed before packages (Bottom up)?
- ▶ How packages are physically represented?
- ▶ Once created, to what purpose will we put these packages?

List of Package Principles

- ▶ Based on package cohesion
 - tell us what to put inside packages
 - for robust, maintainable, and reusable software
 - Packages are the units that focus change, that enable reuse, and that provide the unit of release that prevents developers from interfering with each other
- ▶ Based on couplings between packages
 - talk about metrics that evaluate the package structure of a system

Three Package Design Principles On Package Cohesion

- ▶ Release Reuse Equivalency Principle (REP)
 - ▶ Common Closure Principle (CCP)
 - ▶ Common Reuse Principle (CRP)
- 

Release Reuse Equivalency Principle (REP)

The unit (or granule) of reuse is the unit (granule) of release

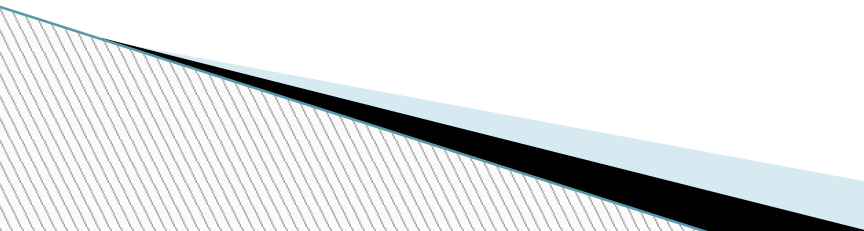
- *Bob Martin*

It is about reusing software

Reusable software / library is like an external software

- You use it but somebody else maintains it.
- You continue to use it even when the software continue to evolve.
- You can't reuse that is not released.

Release-Reuse Equivalency

- ▶ Expectations on external software
 - Documentation
 - complete, accurate, up-to-date
 - Maintenance
 - bugs will be fixed, enhancements will be considered
 - Reliability
 - no major bugs
 - no sudden changes
 - can stay with proven versions (for a while)
- 

Release Control

- ▶ Requirements for reusable software
 - Put reusable components into a package
 - Track versions of the package
 - Assign release numbers to stable releases
 - Stable releases need release notes
 - Allow users to use older releases for a while
- ▶ The unit of reuse is the unit of release

REP Summary

- ▶ Group components (classes) for users
- ▶ Single classes are usually not reusable
 - Several collaborating classes make up a package
- ▶ Classes in a package should form a reusable and releasable module
 - Module provides coherent functionality
 - Dependencies on other packages controlled
 - Requirements on other packages specified
- ▶ Reduces work for the clients using reusable module

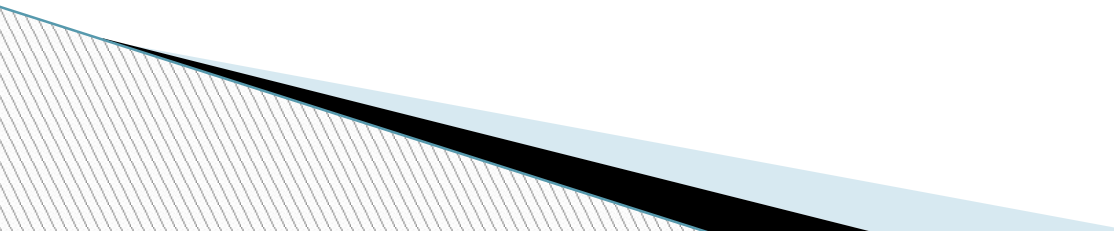
Common Closure Principle (CCP)

Classes that change together, belong together.

Bob Martin

Minimise the impact of change for the programmer.

When a change is needed, it is good for the programmer if the change effects as few packages as possible, because of compile and link time and revalidation



CCP

- ▶ If two classes are so tightly bound, either physically or conceptually, such that they almost always change together; then they belong in the same package.

From OCP to CCP

- ▶ OCP: Classes should be open for extension, but closed for modification
 - This is an ideal
 - Classes will be designed for likely *kinds of changes*
- ▶ *Cohesion of closure for packages*
 - *Classes in a package should be closed to the same kinds of changes*
 - *Changes will be confined within few packages*

CCP Summary

- ▶ Group classes with similar closure together
 - package closed for anticipated changes
- ▶ Confines changes to a few packages
- ▶ Reduces workload related to releasing, revalidating, and redistributing the software

Common Reuse Principle (CRP)

Classes in packages should be reused together
Or

Classes that aren't reused together should not
be grouped together.

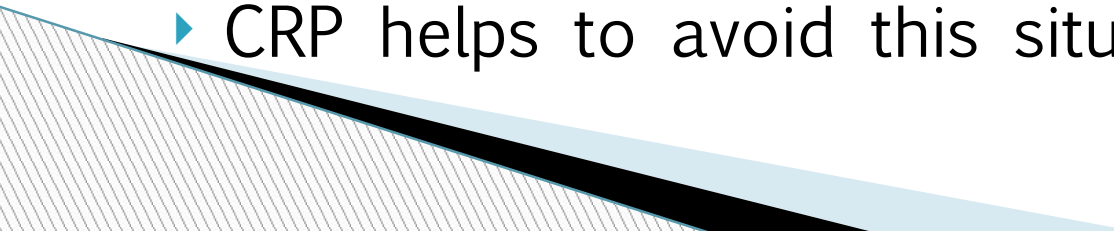
- Bob Martin

Packages should be focused, users should use all classes
from a package

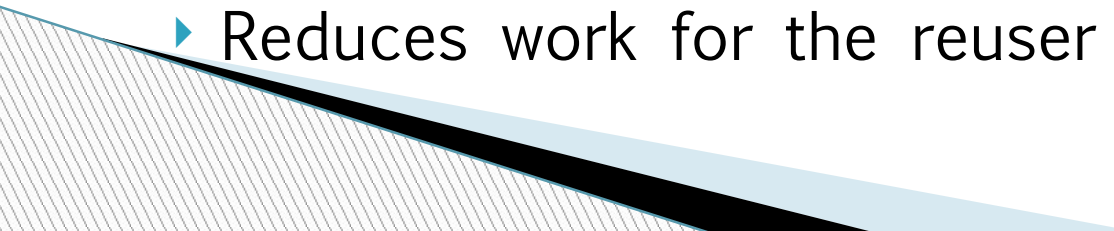
CRP for packages is analogous to SRP for classes



Common Reuse

- ▶ Use of a package brings in all its dependencies
 - ▶ When a user is only interested in a few classes of a package
 - the user code still depends on all dependencies of the package
 - the user code must be recompiled/relinked and retested after a new release of the package, even if the actually used classes didn't change
 - ▶ CRP helps to avoid this situation
- 

CRP Summary

- ▶ Group classes according to common reuse
 - avoid unnecessary dependencies for users
 - ▶ Following the CRP often leads to splitting packages
 - Get more, smaller and more focused packages
 - ▶ CRP analogous to SRP for classes
 - ▶ Reduces work for the reuser
- 

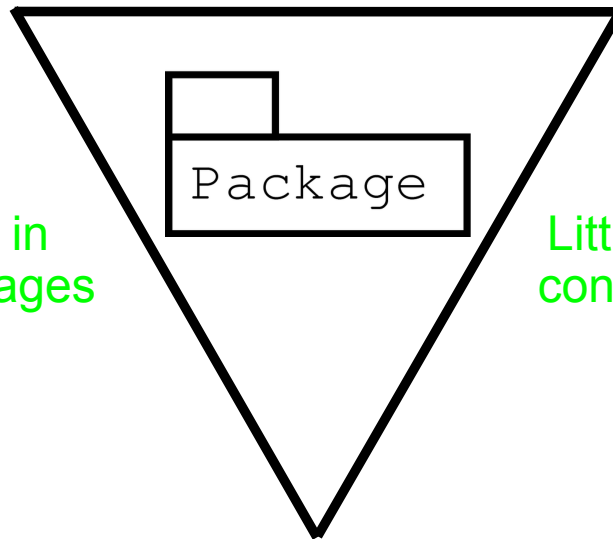
The Triad Triangle

REP: Group
for reusers

Unneeded
releases

CCP: Group for
maintainer

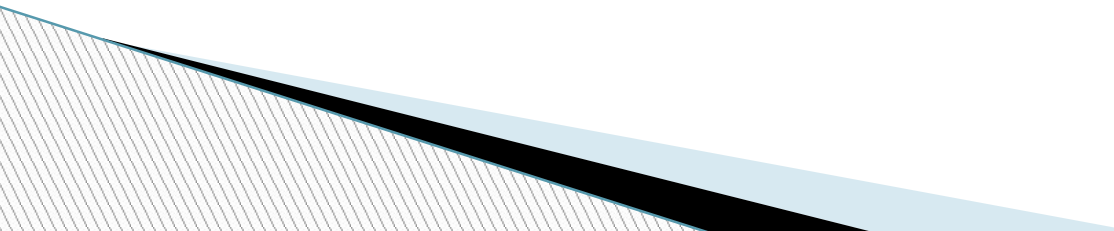
Changes in
many packages



Little reuser
convenience

CRP: Split to get
common reuse

Three Package Principles on Package Couplings

- ▶ Acyclic Dependencies Principle (ADP)
 - ▶ Stable Dependencies Principle (SDP)
 - ▶ Stable Abstractions Principle (SAP)
- 

The Acyclic Dependencies Principle (ADP)

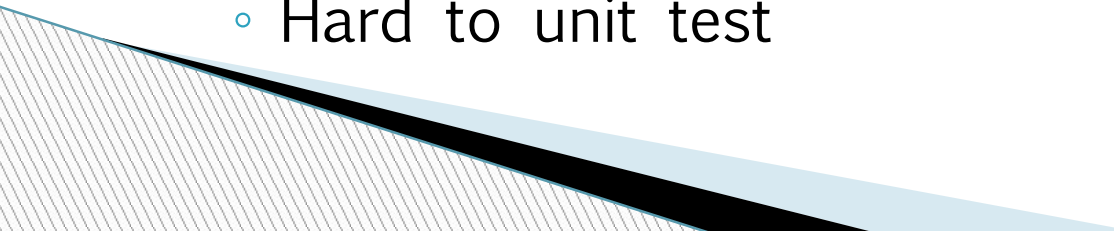
The dependency structure for packages must be a Directed Acyclic Graph (DAG)

Stabilise and release a project in pieces

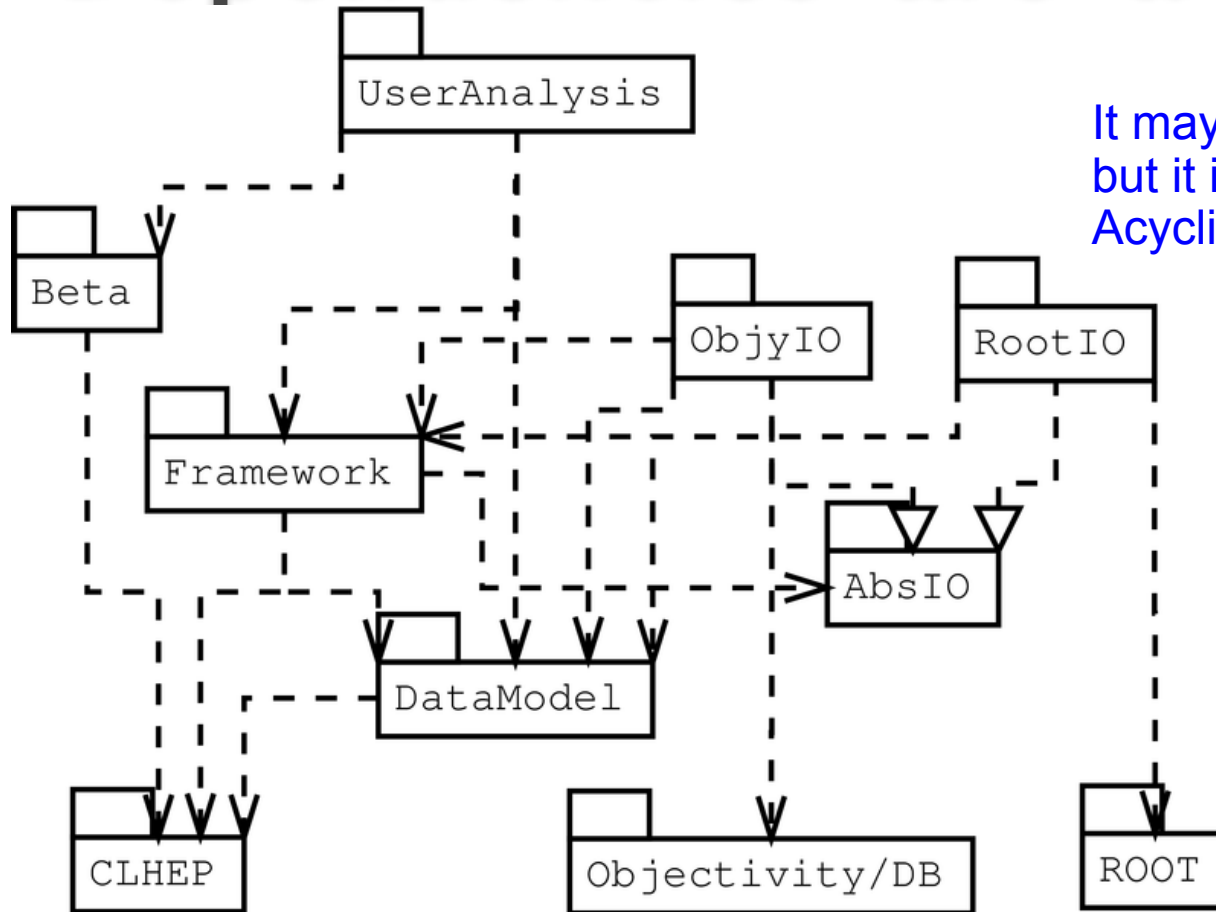
Avoid interfering developers [?] *Morning after syndrome*

Organise package dependencies in a top-down hierarchy

Morning-After-Syndrome

- ▶ You work on a package and eventually it works → you go home happy
 - ▶ The next day your package stopped working!
 - A package you depend upon changed
 - Somebody stayed later or came in earlier
 - ▶ When this happens frequently
 - Developers interfere with each other
 - Hard to stabilise and release
 - Hard to unit test
- 

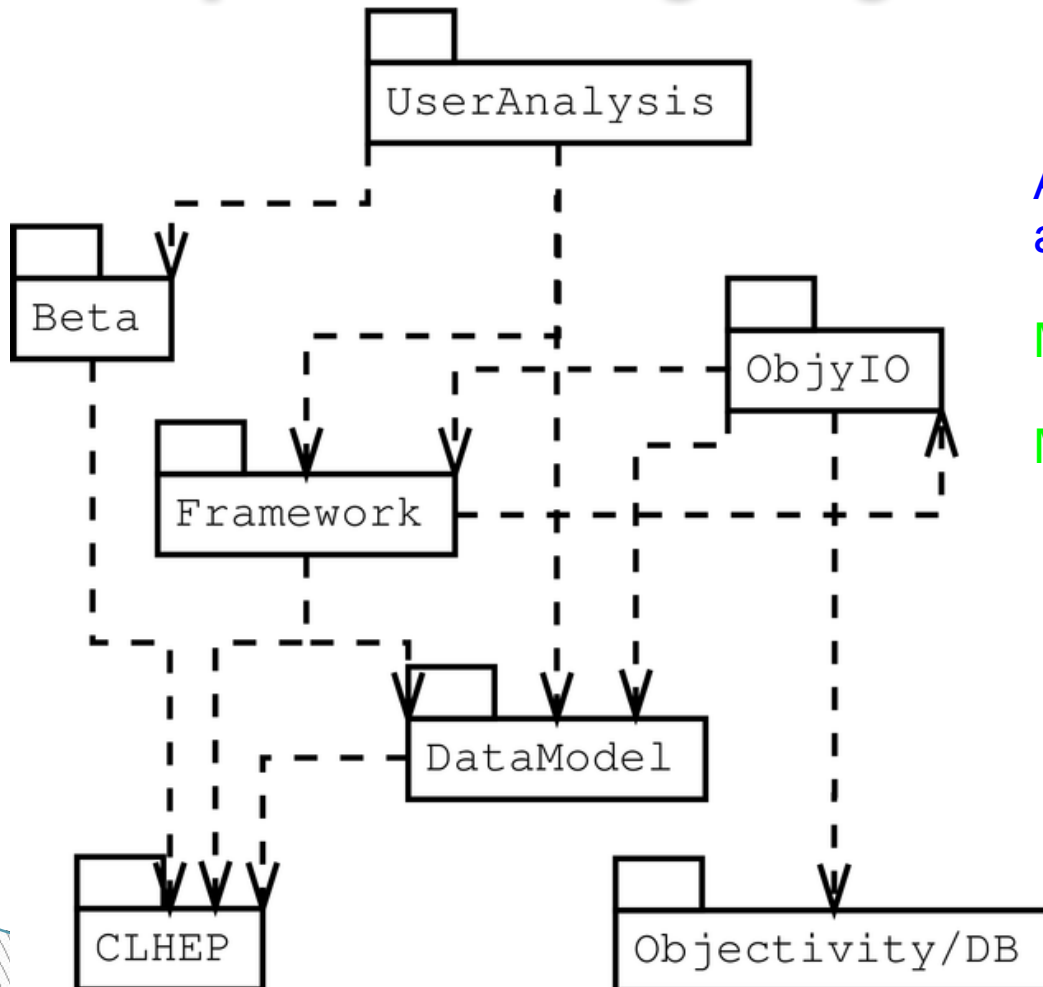
Dependencies are a DAG



It may look complicated,
but it is a DAG (Directed
Acyclic Graph)

Can exchange
ObjyIO and RootIO

Dependency Cycles



A cycle between Framework and ObjyIO

Must develop together

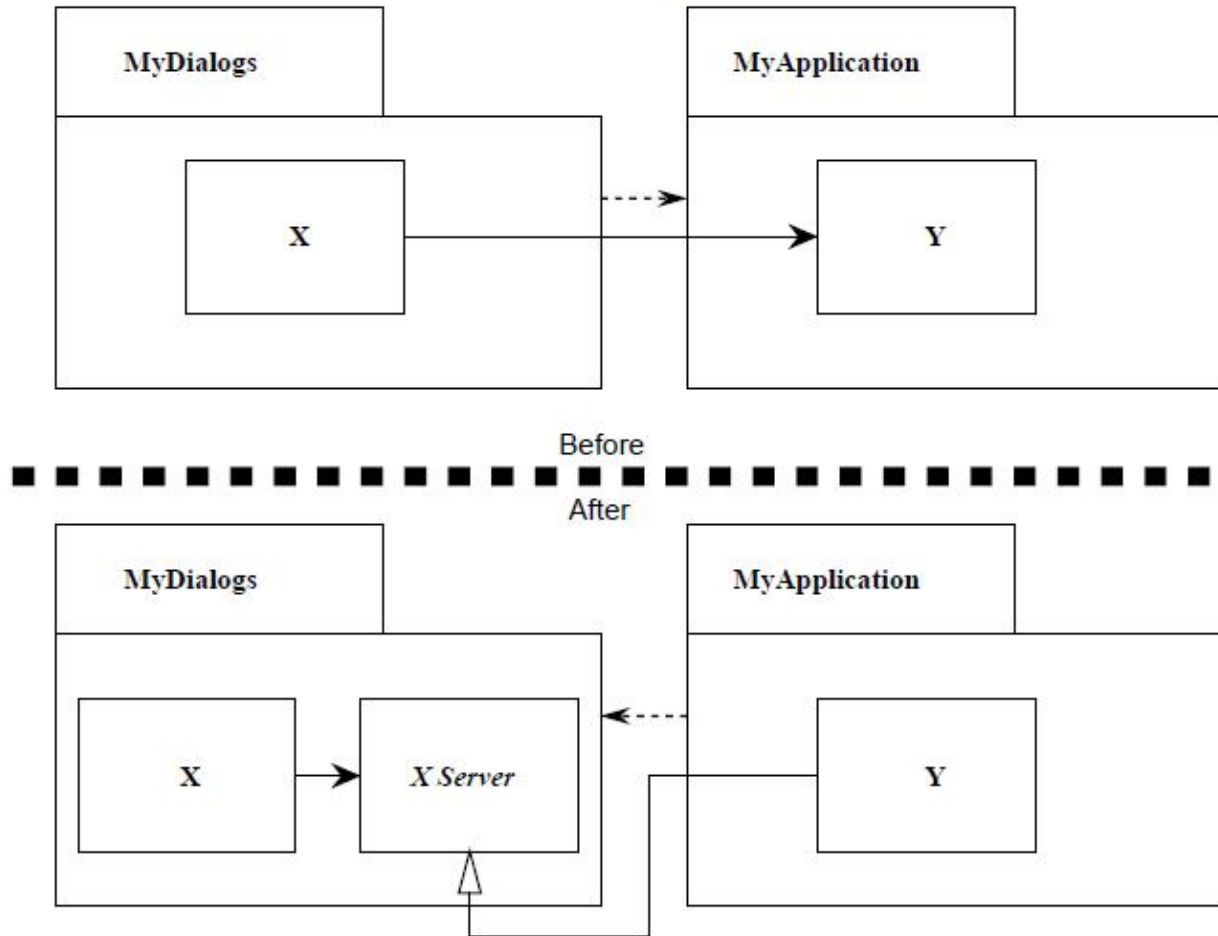
May need multipass link

ADP Summary

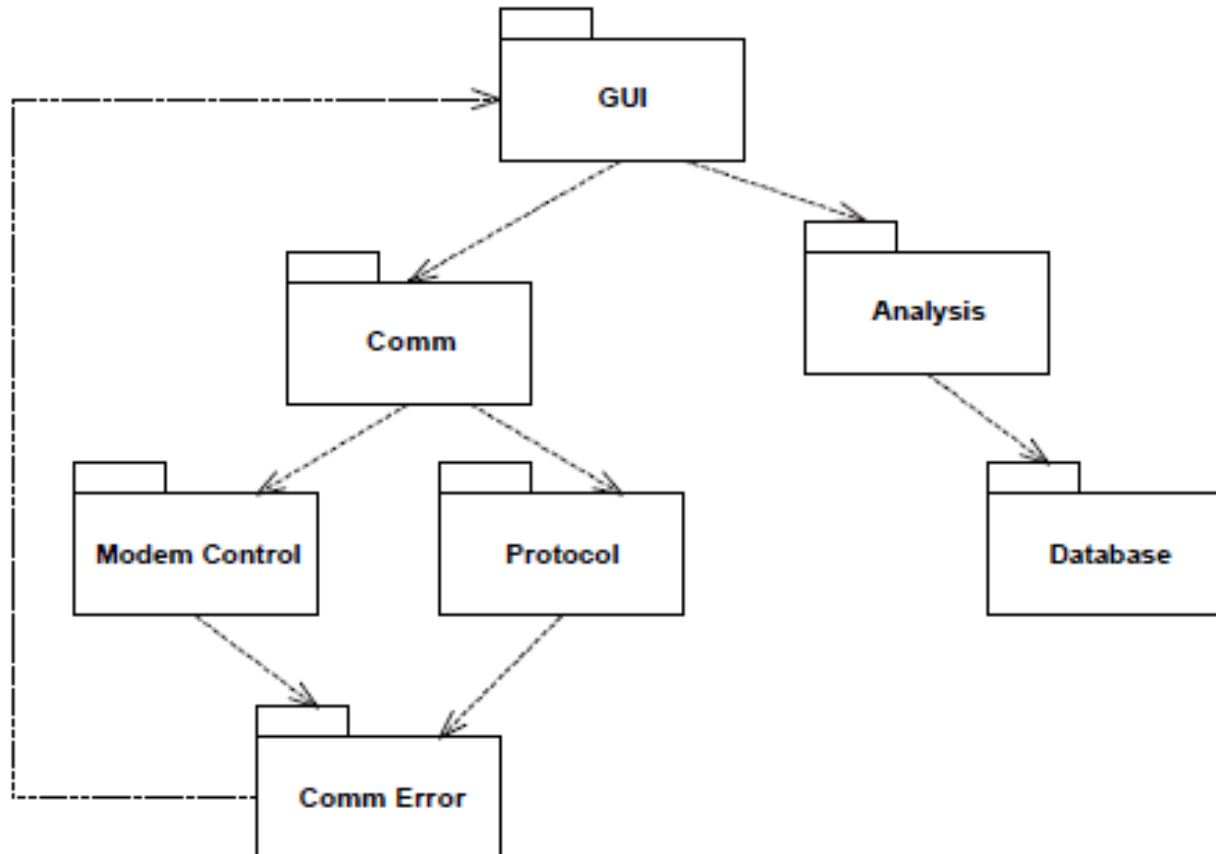
- ▶ Dependency structure of packages is a DAG
- ▶ Dependency cycles result in Morning-After-Syndrome
- ▶ Dependency hierarchy should be shallow
- ▶ Break cycles with
 - Abstract interfaces (DIP)
 - Interface Segregation Principle (ISP)
 - Splitting packages (CRP)
 - Reorganising packages
- ▶ Package dependency structure grows and evolves with the logical design of the system

Breaking Cycle with DI

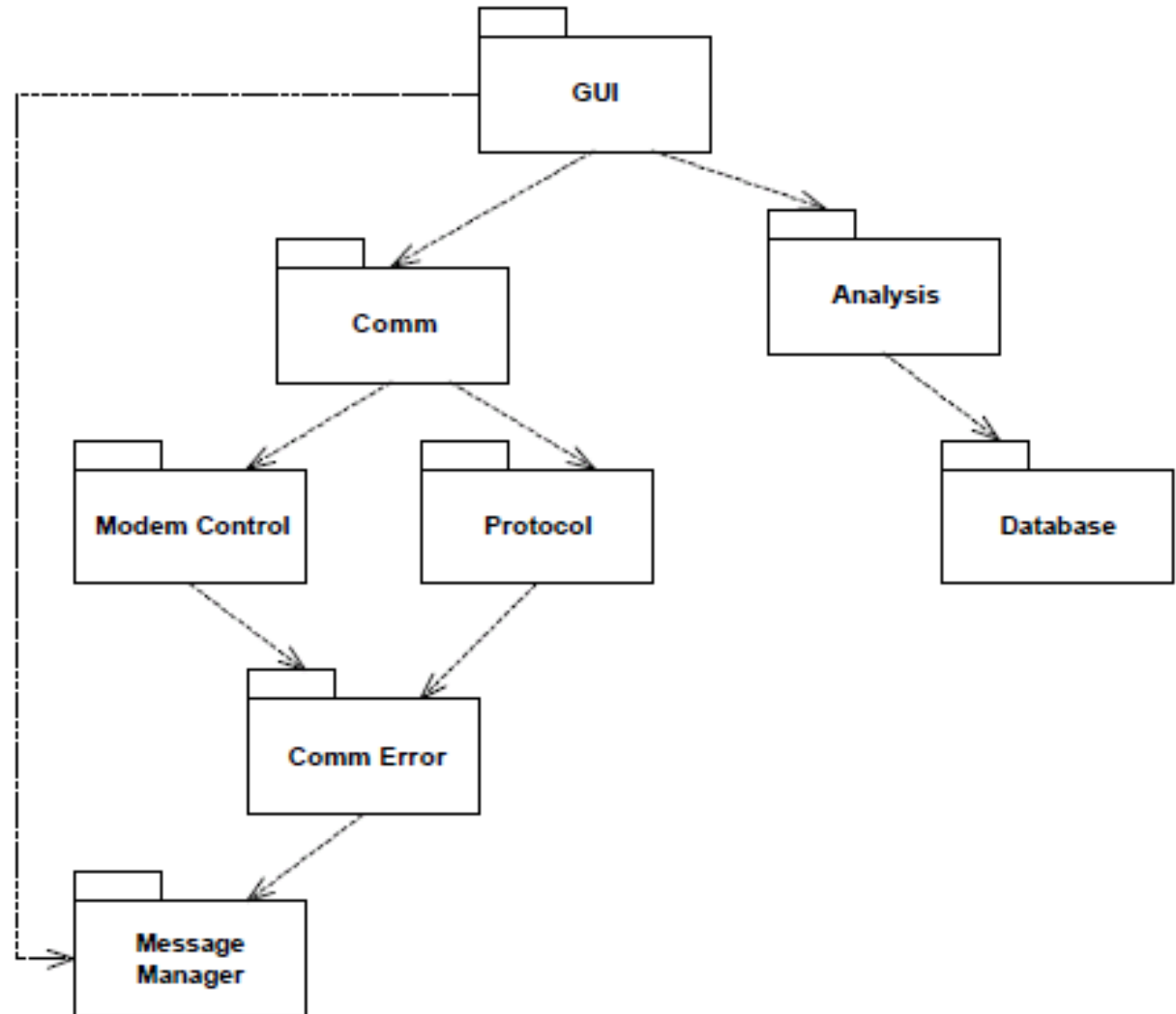
Breaking the Cycle with Dependency Inversion



Cyclic dependency

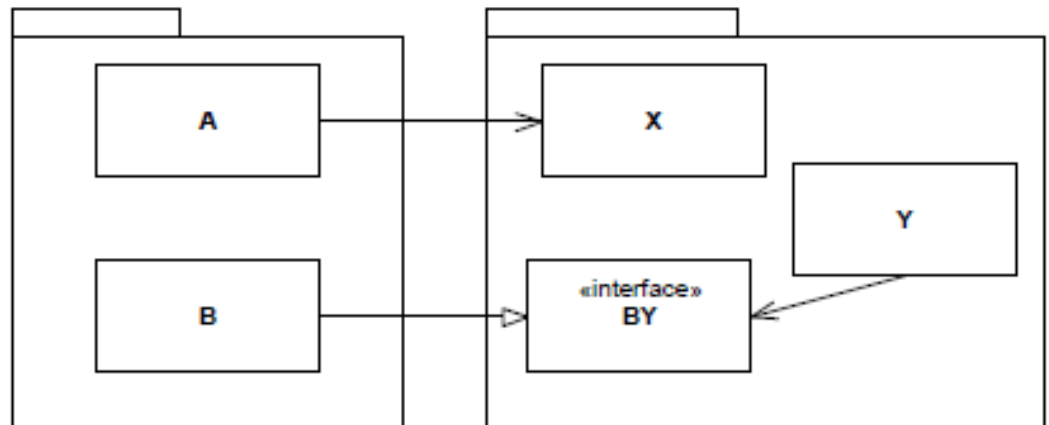
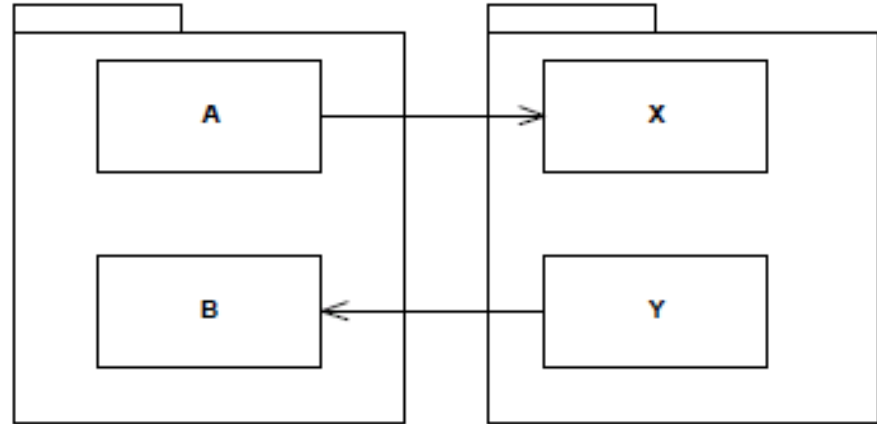


Removing cycle by creating new package



Remove cycle through Interface Segregation Principle

Interfaces are very often included in the package that uses them, rather than in the package that implements them.



Stable Dependencies Principle (SDP)

Depend in the direction of stability

- Robert Martin

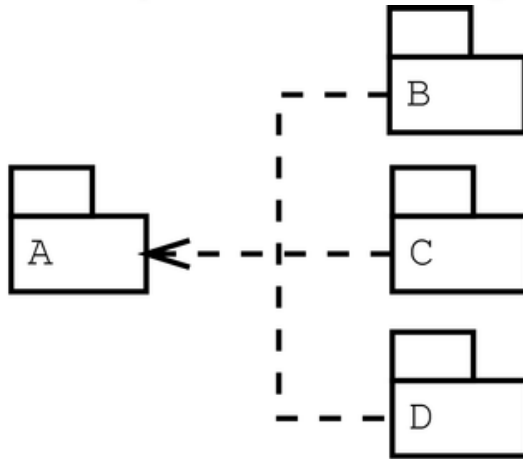
Stability: Corresponds to effort required to change a package

Stable package: Hard to change within the project

Stability can be quantified

Any package that we expect to be volatile should not be depended upon by a package that is difficult to change! Otherwise the volatile package will also be difficult to change.

Quantifying Stability



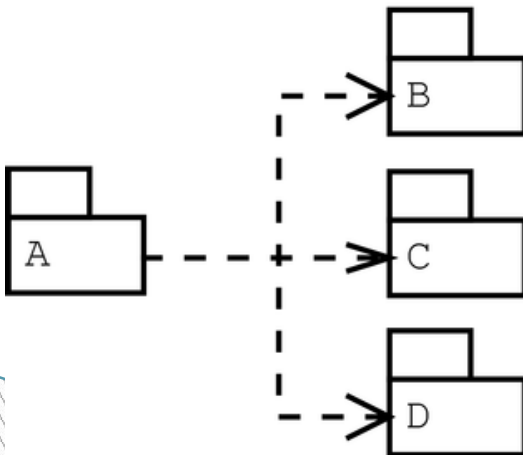
A is a stable package,
many other packages
depend on it



Responsible
 $I = 0$

$C_a = \#$ classes outside the package
which depend on classes
inside the package
(Afferent Couplings)

$C_e = \#$ classes outside the package
which classes inside the
package depend upon
(Efferent Couplings)



A is unstable, it
depends on many
other packages



Irresponsible
 $I = 1$

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

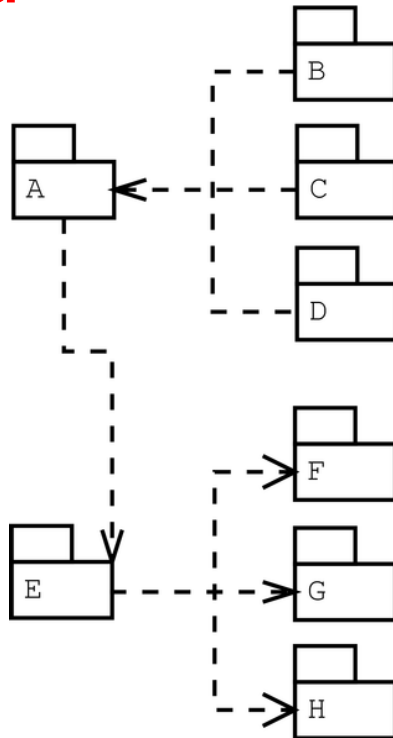
I – Instability Metric

SDP Example

Bad

A is responsible for B, C and D.
It depends on E,
→ irresponsible

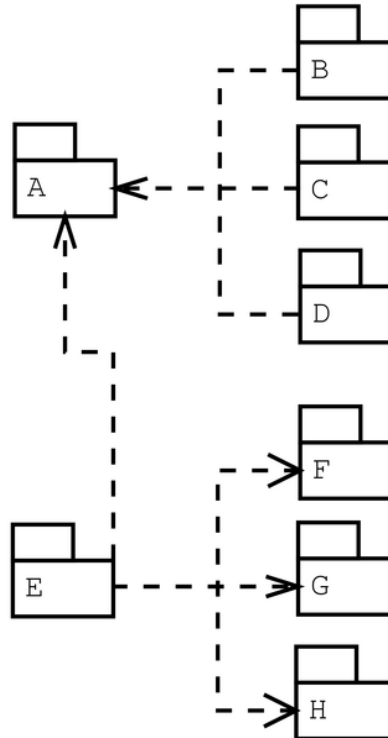
E depends on F, G and H. A depends on it. E is responsible and irresponsible.



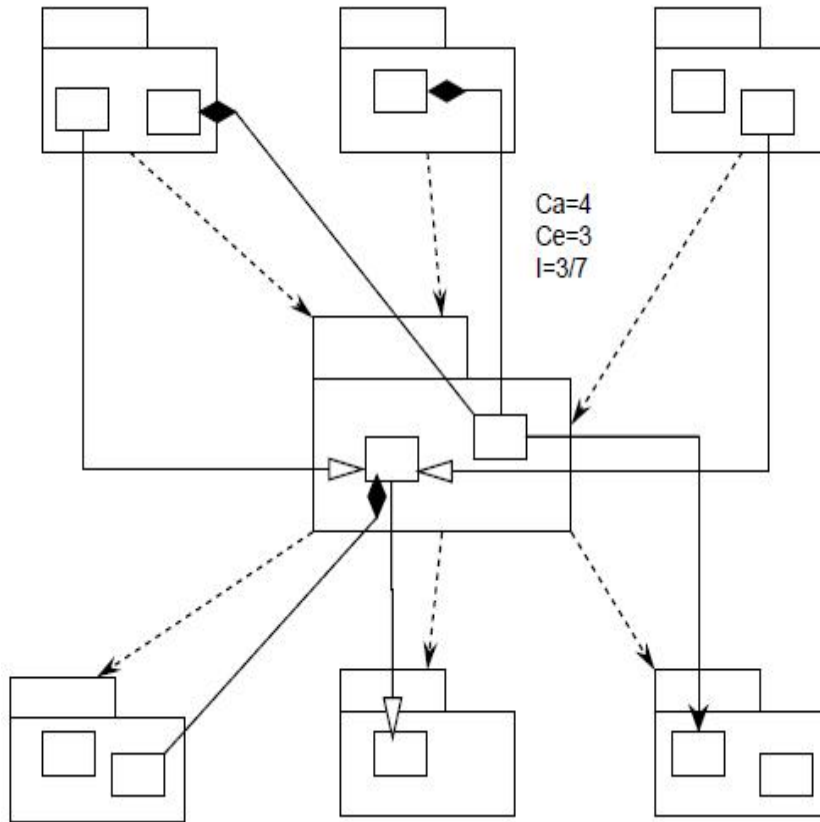
Good

A is responsible for B, C, D and E. It will be hard to change

E depends on A, F, G and H. It is irresponsible and will be easy to modify.



Stability Metrics



The SDP says that the I metric of a package should be larger than the I metrics of the packages that it depends upon.

i.e. I metrics should decrease in the direction of dependency.

SDP Summary

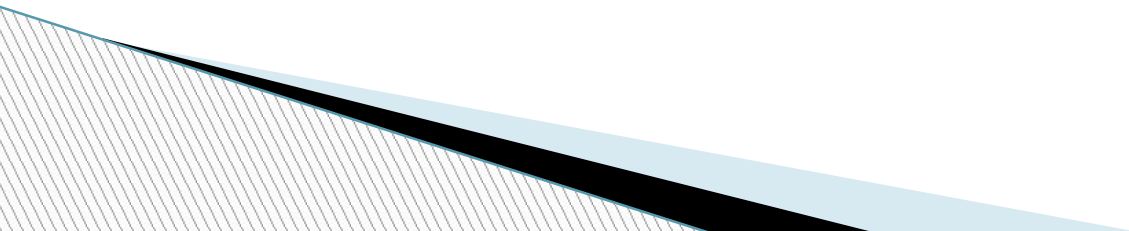
- ▶ Organise package dependencies in the direction of stability
- ▶ (In-) Stability can be quantified → I-Metric
- ▶ More the I, less the stability
- ▶ Dependence on stable packages corresponds to DIP for classes
 - Classes should depend upon (stable) abstractions or interfaces
 - These can be stable (hard to change)

Stable Abstractions Principle (SAP)

Stable packages should be abstract packages.
Unstable packages should be concrete packages.

Robert Martin

Stable packages contain high level design. Making them abstract opens them for extension but closes them for modifications (OCP). Some flexibility is left in the stable hard-to-change packages.



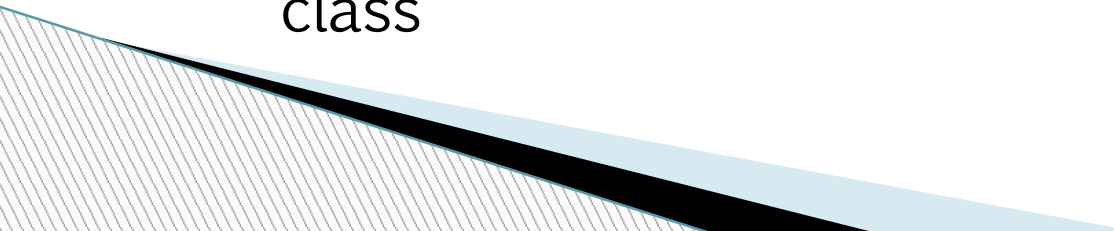
Correlation of Stability and Abstractness

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

Dependency Inversion Principle for packages

- ▶ SAP and the SDP combined amount to the Dependency Inversion Principle for Packages.
- ▶ SDP - dependencies should run in the direction of stability
- ▶ SAP - stability implies abstraction
- ▶ Thus, dependencies run in the direction of abstraction.

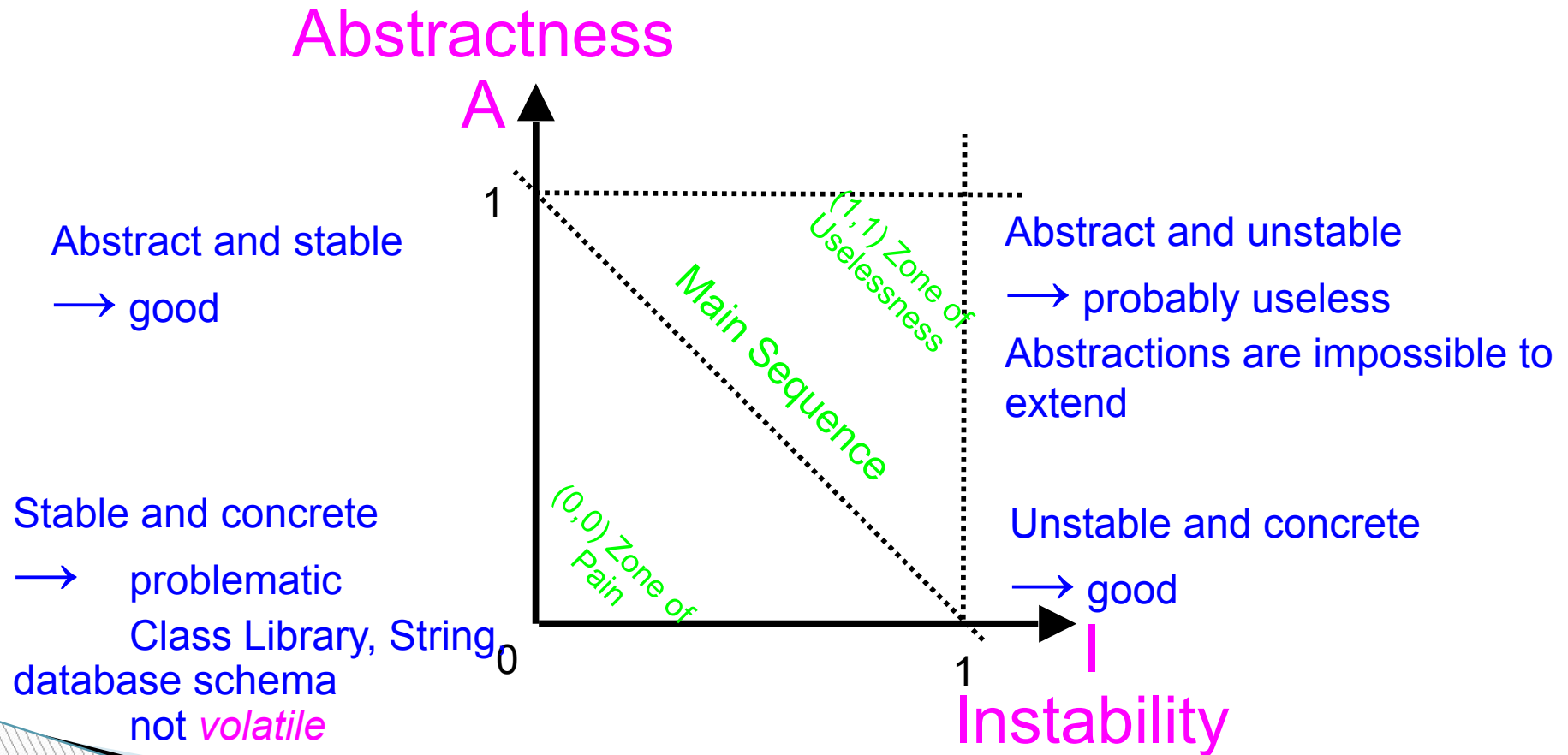
Quantifying Abstractness

- ▶ The Abstractness of a package can be quantified
 - ▶ Abstractness A is defined as the fraction of abstract classes in a package.
 - ▶ Corresponds to abstract classes
 - Abstract classes have at least one pure virtual member function (for C++)
 - Abstract packages have at least one abstract class
- 

A Metric

- ▶ Measure of the abstractness of a package
- ▶ $A = (\text{abstract classes in a package}) / (\text{total number of classes in the package})$
- ▶ Value from 0 to 1

The A vs I Plot (A/I graph)



Packages have the best characteristics if they are on or close to the main sequence

Distance from Main Sequence D-Metric

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

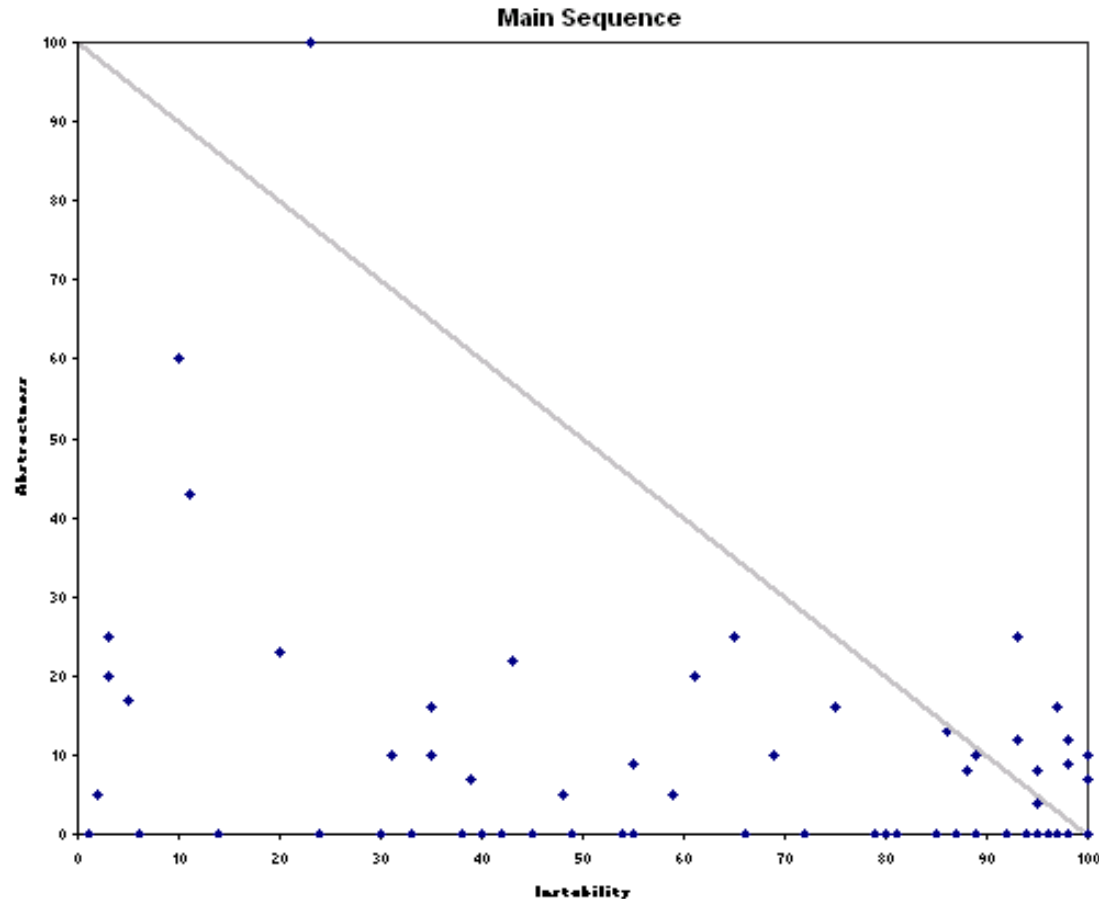
$$D \in [0, \sim 0.707]$$

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

$$\text{Normalised so that } D' \in [0, 1]$$

- ▶ A design can be analyzed for its overall conformance to the main sequence
- ▶ Any package that has a D value that is not near zero can be reexamined and restructured.
- ▶ Helps define packages that are more reusable, and less sensitive to change

Scatter Plot on A-I graph



Much of the code at
was too concrete
for its stability

At least the problem was
recognised ...

Statistical Analysis on D

- ▶ Calculate the mean and variance of all the D metrics within a design
- ▶ A conformant design to have a mean and variance which were close to zero
- ▶ Variance can be used to establish “control limits” which can identify packages that are “exceptional” in comparison to all the others

SAP Summary

- ▶ Stable packages should be abstract
- ▶ In a large project packages should have a balance of Abstractness and Instability
 - Lie close to the main sequence in A-I-plot
- ▶ Metrics I and A help to quantify design quality

Ten Commandments of OO Programming

1. Software entities (classes, modules, etc) should be open for extension, but closed for modification. (The open/closed principle -- Bertrand Meyer)
2. Derived classes must be usable through the base class interface without the need for the user to know the difference. (The Liskov Substitution Principle)
3. Details should depend upon abstractions. Abstractions should not depend upon details. (Principle of Dependency Inversion)
4. The granule of reuse is the same as the granule of release. Only components that are released through a tracking system can be effectively reused. (Release-Reuse Equivalency)
5. Classes within a released component should share common closure. That is, if one needs to be changed, they all are likely to need to be changed. What affects one, affects all. (Common Closure)

Ten Commandments of OO Programming

6. The dependency structure for released components must be a DAG. There can be no cycles. (Acyclic Dependencies Principle)
7. Dependencies between released components must run in the direction of stability. The dependee must be more stable than the depender. (Stable Dependencies Principle)
8. The more stable a released component is, the more it must consist of abstract classes. A completely stable component should consist of nothing but abstract classes. (Stable Abstractions)
9. Where possible, use proven patterns to solve design problems.
10. When crossing between two different paradigms, build an interface layer that separates the two. Don't pollute one side with the paradigm of the other.

OO Design Principle Summary

- ▶ Structure of classes and packages that keeps the software application flexible, robust, reusable, and developable

