4 OO Package Design Principles

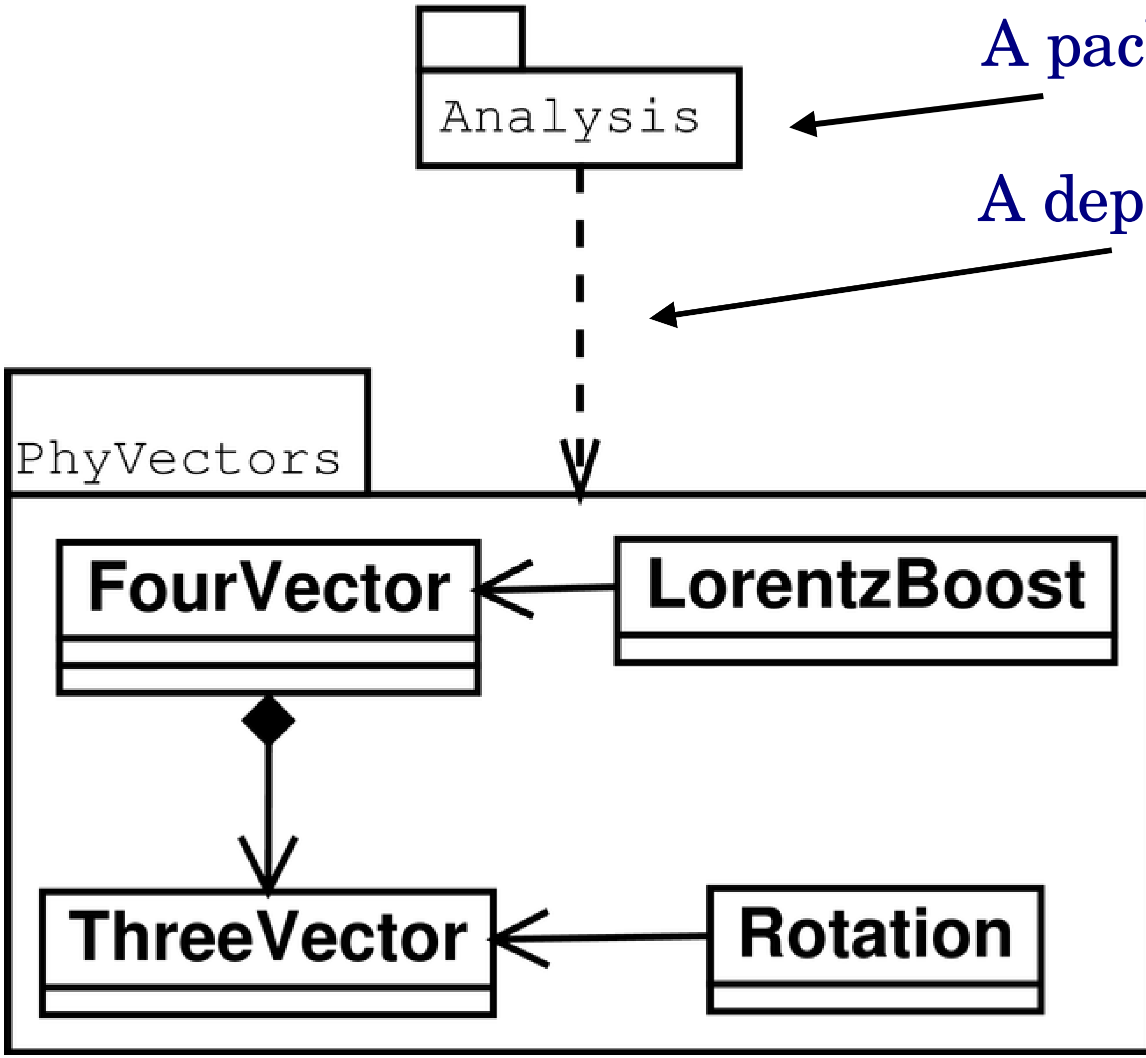
4.1 Packages Introduction

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

4.1 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

4.2 Packages in UML

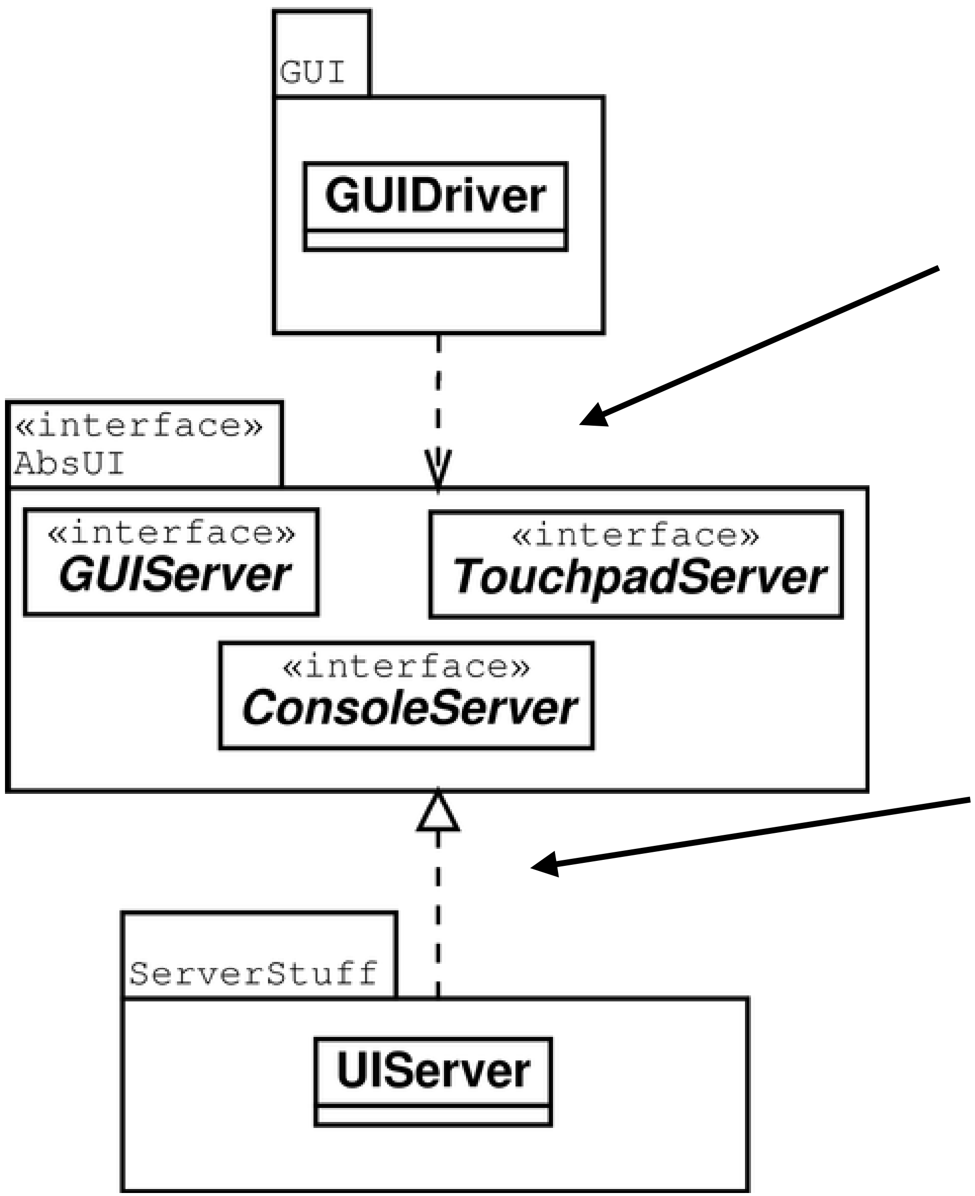


A package

A dependency between packages

A package with classes shown inside

4.2 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

4.3 Three Package Design Principles

- Reuse-Release Equivalency Principle
- Common Closure Principle
- Common Reuse Principle

4.3 Reuse-Release Equivalency Principle (REP)

The unit of reuse is the unit of release

Bob Martin

It is about reusing software.

Reuseable software is external software,
you use it but somebody else maintains it.
There is no difference between commercial
and non-commercial external software for reuse.

4.3 Reuse-Release Equivalency

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

4.3 Release Control

- Requirements for reuseable software
  - Put reuseable components into a package
  - Track versions of the package (CVS)
  - 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

4.3 REP Summary

- Group components (classes) for reusers
- Single classes are usually not reuseable
  - Collaborating classes make up a package
- Classes in a package should form a reuseable and releaseable module
  - Module provides coherent functionality
  - Dependencies on other packages controlled
  - Requirements on other packages specified
- Reduces work for the reuser

4.3 Common Closure Principle (CCP)

Classes which 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 affects as few packages as possible, because of compile and link time and revalidation

4.3 From OCP to CCP

- OCP: Classes should be open for extension, but closed for modification
  - This is an ideal
  - Classes 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
- Reduces frequency of release of packages

4.3 CCP Summary

- Group classes with similar closure together
  - package closed for anticipated changes
- Confines changes to a few packages
- Reduces package release frequency
- Reduces work for the programmer

4.3 Commom Reuse Principle (CRP)

Classes in packages should be reused together

Bob Martin

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

CRP for packages is analogous to SRP for classes

4.3 Common Reuse

- A package brings in all its dependencies
- User only interested in a few classes
  - 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

4.3 CRP Summary

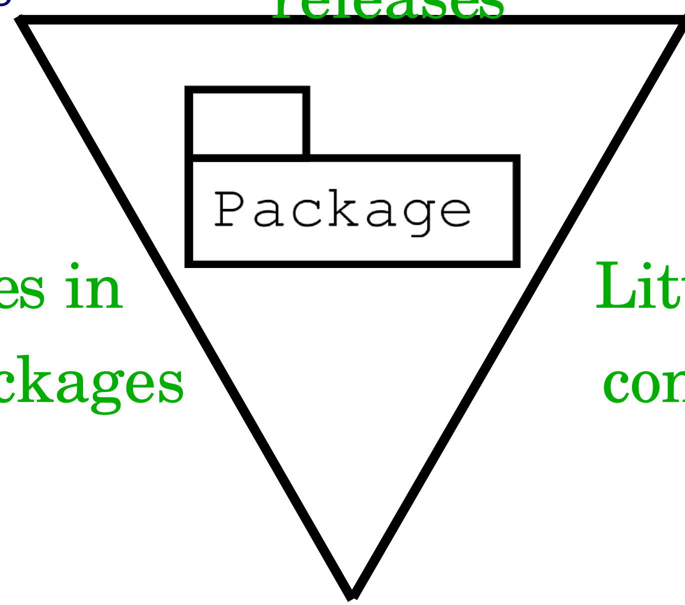
- Group classes according to common reuse
  - avoid unneccessary dependencies for users
- Following the CRP often leads to splitting packages
  - Get more, smaller and more focused packages
- CRP analogous to SRP for classes
- Reduces work for the reuser

4.3 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

4.4 The Development Environment

- Controlling relations between packages
  - Critical for large projects
  - Programming, compile and link time
- Three more package design principles
  - Acyclic Dependencies
  - Stable Dependencies
  - Stable Abstractions
- Other aspects of development environment

4.4 The Acyclic Dependencies Principle (ACP)

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

Stabilise and release a project in pieces

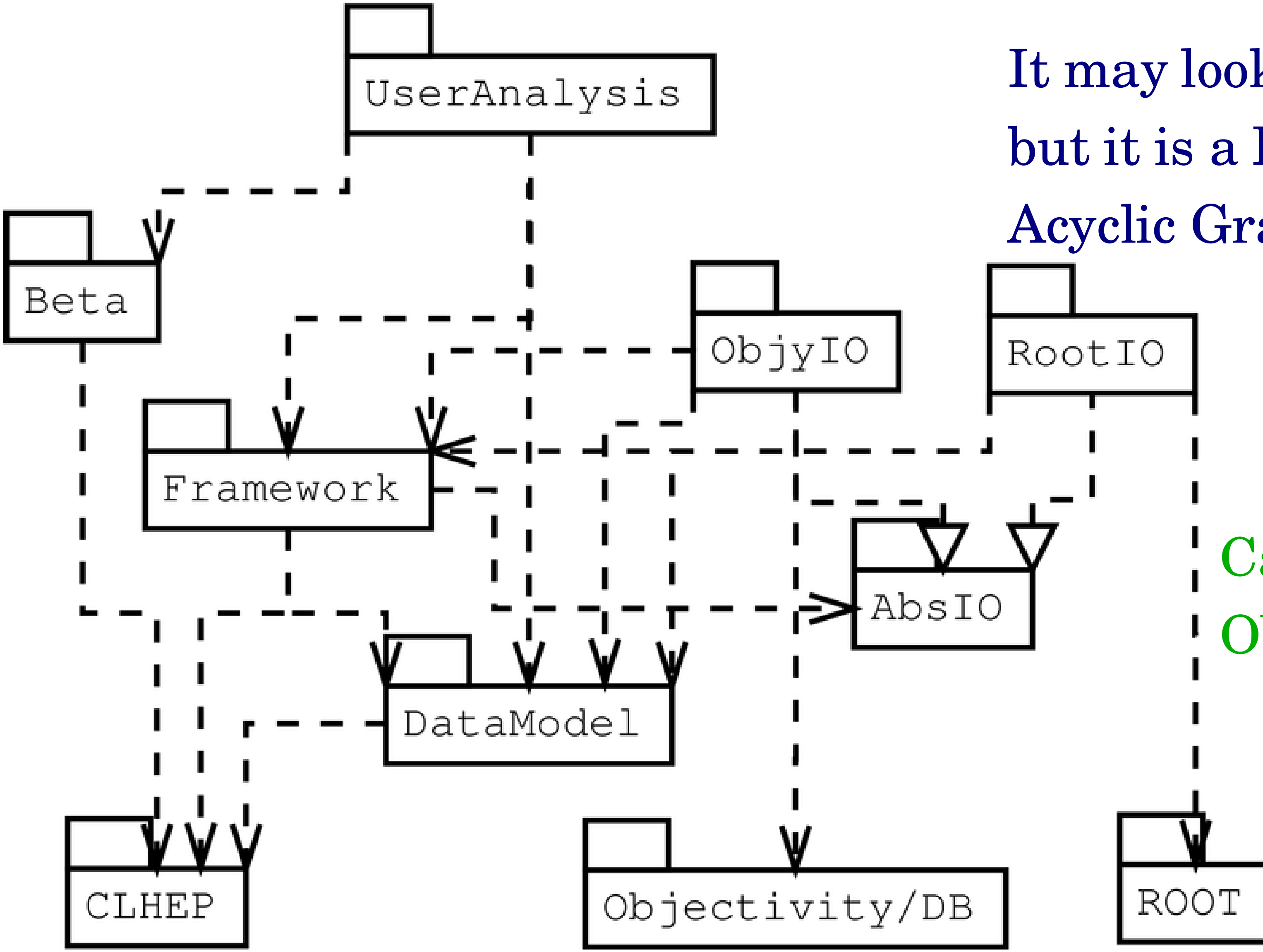
Avoid interfering developers → *Morning after syndrome*

Organise package dependencies in a top-down hierarchy

4.4 Morning-After-Syndrome

- Not the one after an extended pub crawl
- You work on a package and eventually it works → you go home happy
- The next day your package is broken!
  - 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

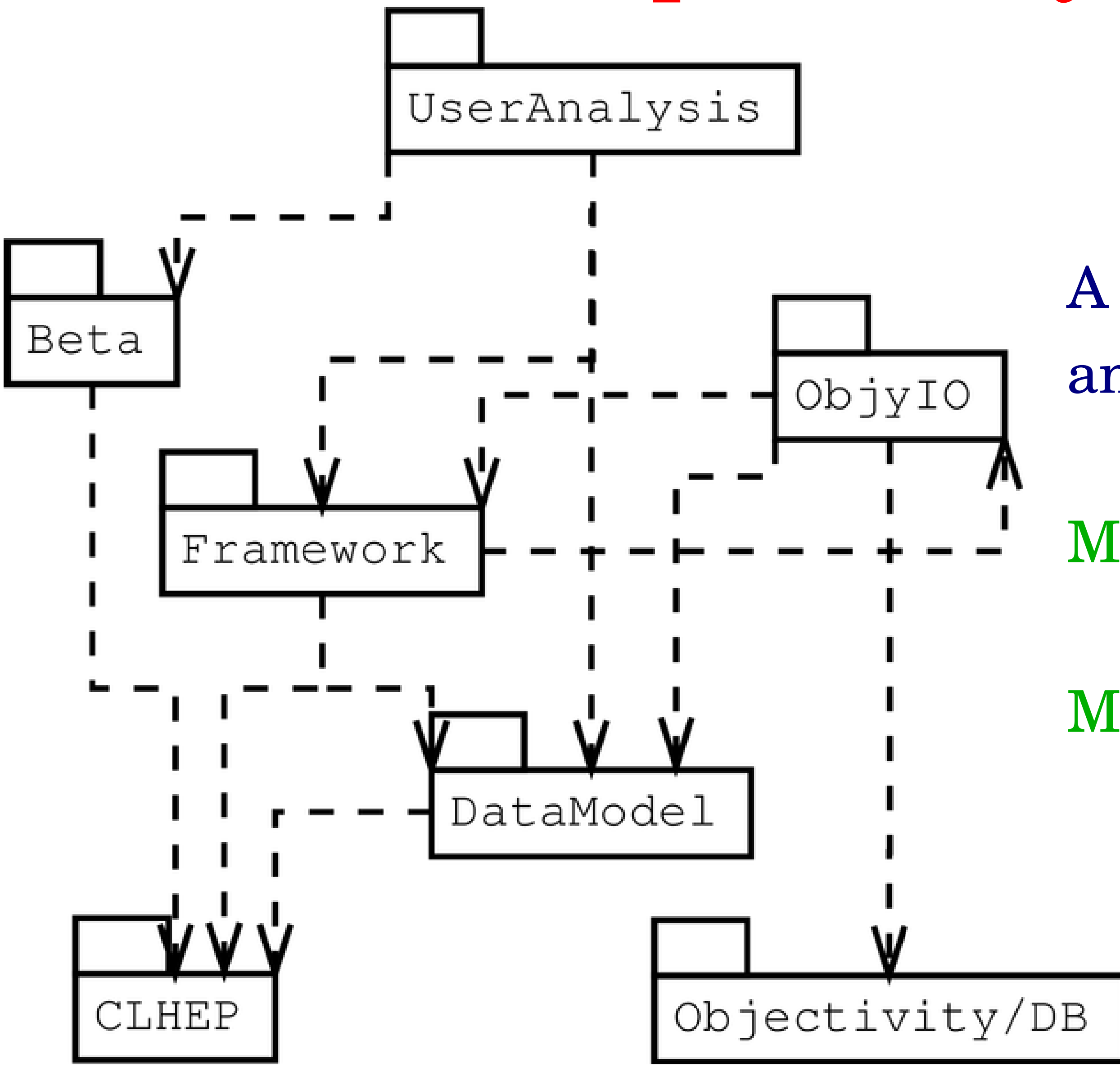
4.4 Dependencies are a DAG



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

Can exchange ObjyIO and RootIO

4.4 Dependency Cycles



A cycle between Framework and ObjyIO

Must develop together

May need multipass link

4.4 ADP Summary

- Dependency structure of packages is a DAG
- Dependency cycles → Morning-After-Syndrome
- Dependency hierarchy should be shallow
- Break cycles with
  - Abstract interfaces (DIP)
  - Splitting packages (CRP)
  - Reorganising packages

4.4 Stable Dependencies Principle (SDP)

Dependencies should point in the direction of stability

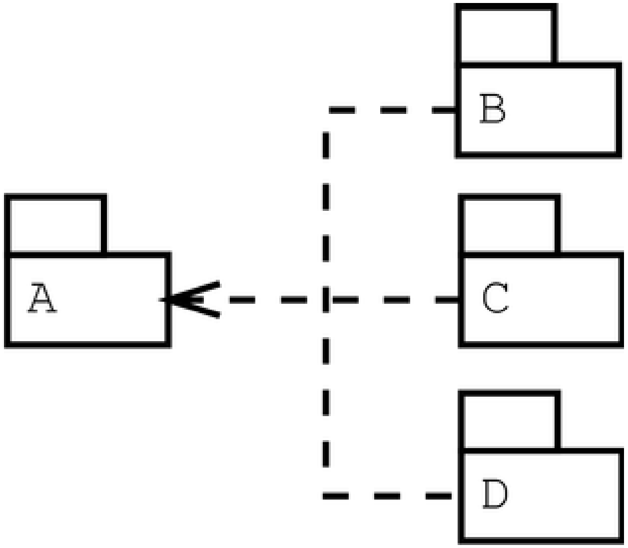
Robert Martin

Stability: corresponds to effort required to change a package

stable package → hard to change within the project

Stability can be quantified

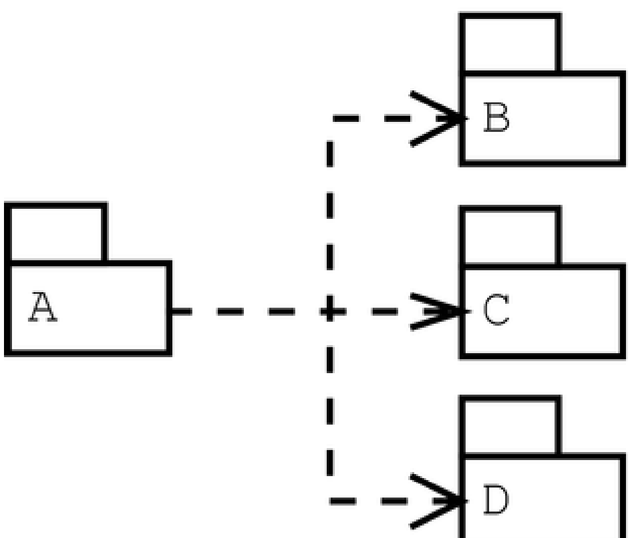
4.4 Quantifying Stability



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

→ Responsible

I = 0



A is unstable, it depends on many other packages

→ Irresponsible

I = 1

C_a = # classes outside the package which depend on classes inside the package (incoming dependencies)

C_e = # classes outside the package which classes inside the package depend upon (outgoing dependencies)

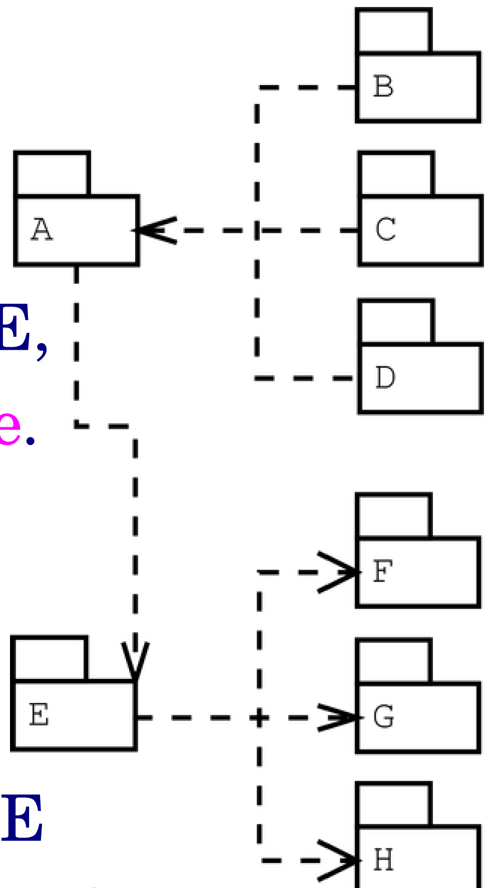
$$I = \frac{C_e}{C_a + C_e}$$

Instability I-Metric

4.4 SDP Example

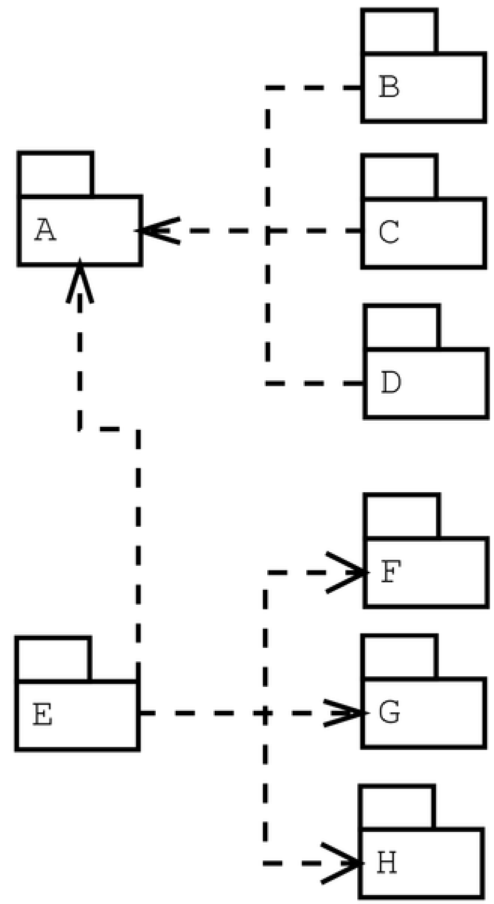
Bad

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



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

Good



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

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

4.4 SDP Summary

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

4.4 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.

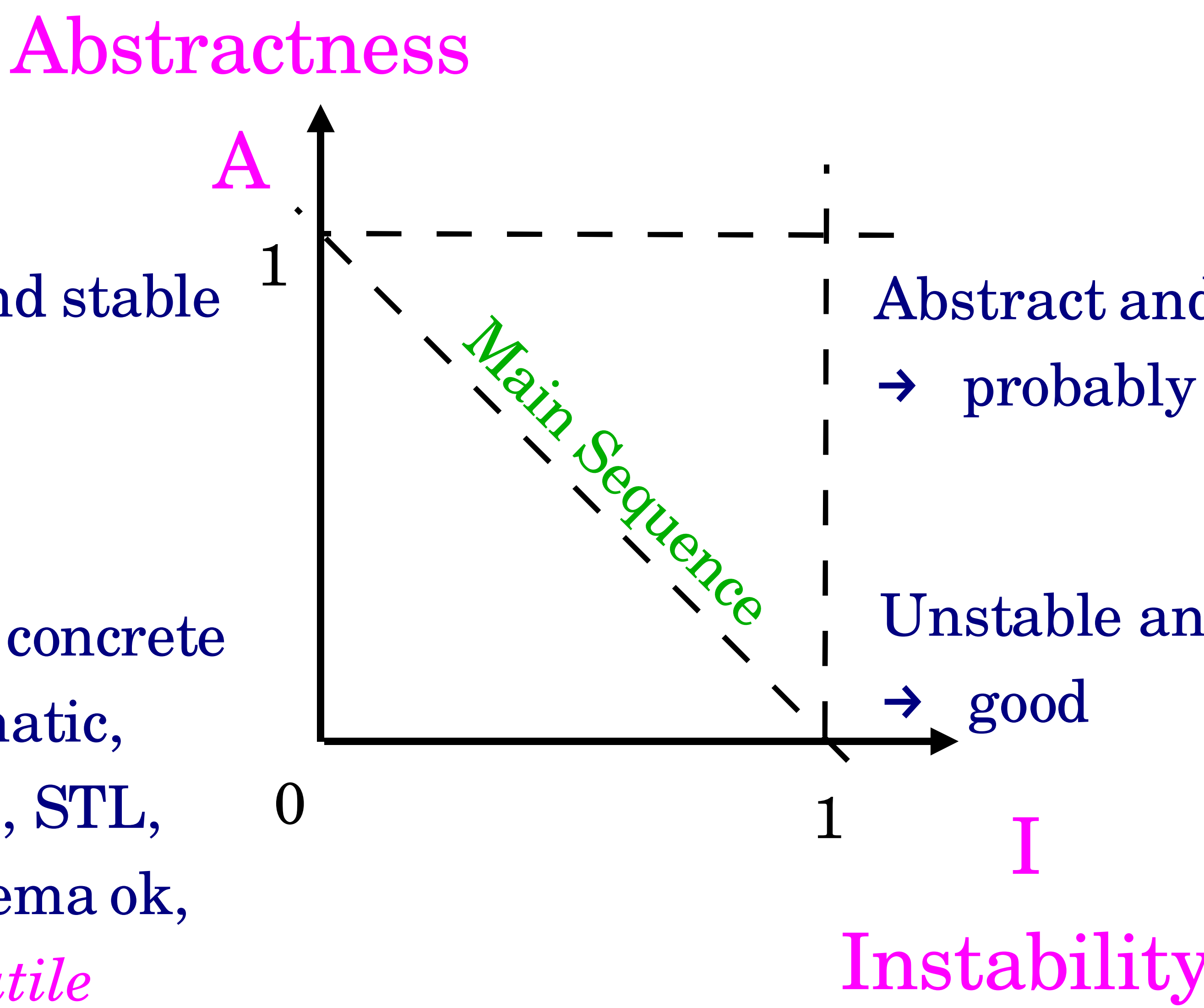
4.4 Quantifying Abstractness

- 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
  - Abstract packages have at least one abstract class

4.4 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

4.4 The A vs I Plot



Abstract and stable
→ good

Abstract and unstable
→ probably useless

Stable and concrete
→ problematic, CLHEP, STL, DB schema ok, not *volatile*

Unstable and concrete
→ good

4.4 Distance from Main Sequence D-Metric

$D = | A + I - 1 |$

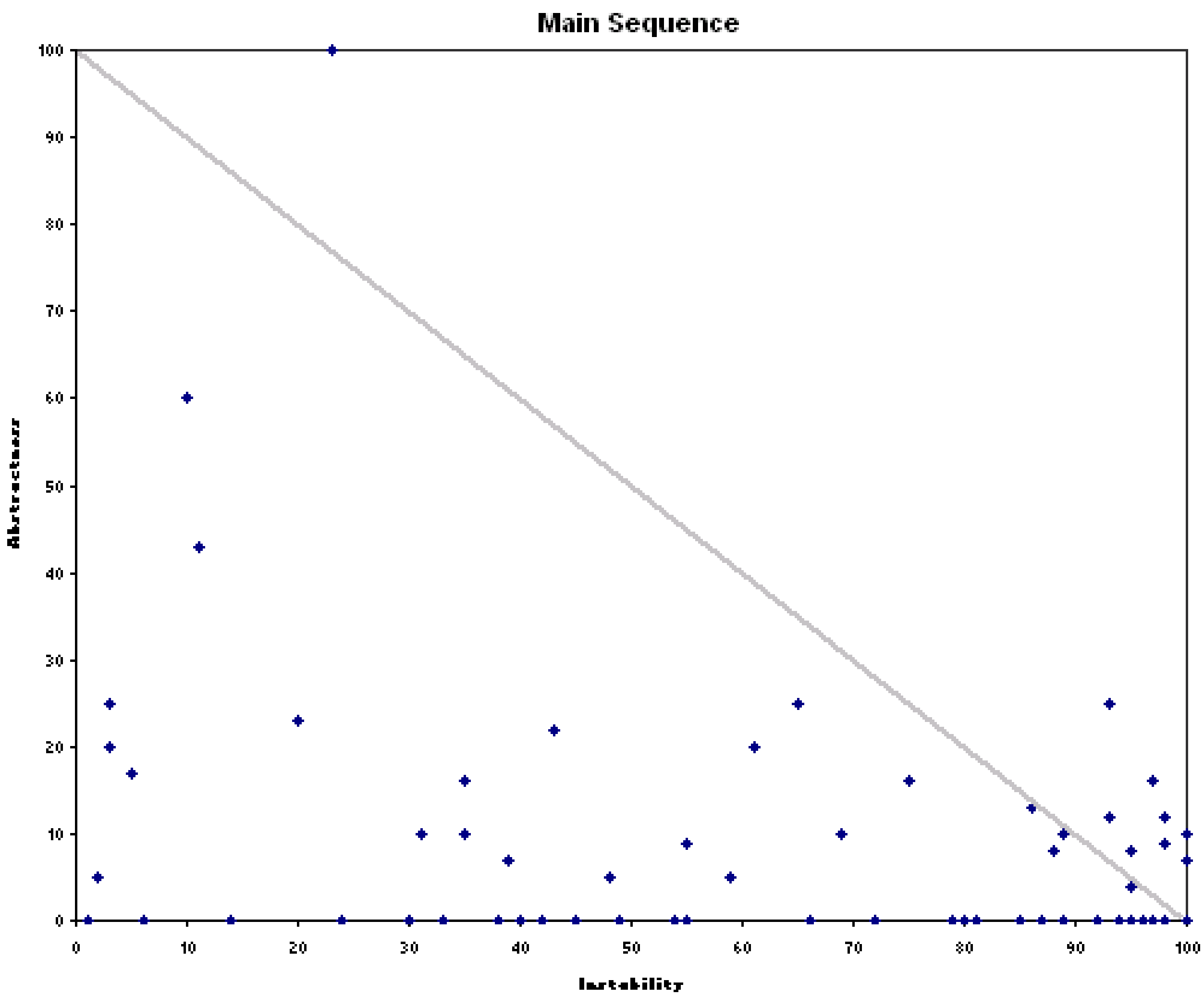
Normalised so that $D \in [0,1]$

Can use mean and standard deviation to set control limits

Can find troublesome packages

Concrete and stable packages like CLHEP or STL will have $D \approx 1$

4.4 Examples from BaBar



Offline code packages release 6.0.1 (early 1999)

Much of the BaBar code at the time was too concrete for its stability

At least the problem was recognised ...

4.4 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 code quality
- Other metrics exist too
  - Code volume and code growth rate
  - Bug discovery and extinction rate

4.5 Mapping Packages

- The BaBar example
  - Each package corresponds to a directory with flat structure under individual CVS control
  - Contains headers (.hh), code (.cc), docs
  - GNUmakefile fragment for building and dependencies
  - Build target: link library and possibly binaries
  - #include “package/class.hh”
- Works well ... easy to understand

4.5 OOD Summary

- Class Design Principles
  - Single Responsibility
  - Open-Closed
  - Liskov Substitution
  - Dependency Inversion
  - Interface Segregation
- Package Design Principles
  - Reuse-Release Equivalence
  - Common Closure
  - Common Reuse
  - Acyclic Dependencies
  - Stable Dependencies
  - Stable Abstractions