# Ingineria Programării

Cursul 12

adiftene@infoiasi.ro
mmoruz@infoiasi.ro

19 – 20 mai

# Cuprins

- Previous courses
  - Program quality
  - Metrics
  - Copyright
- Object Oriented Design
  - For classes
  - For packages
- Manifesto for Software Craftsmanship

# Concepte ale calității programelor

- Siguranță (safety)


- Eficiență (efficiency)

- Întreținere (maintenance)


- Uzabilitate (usability)

# COCOMO


- COnstructive COst MOdel (Boehm 1981)
- Folosit pentru evaluarea costurilor
- Trei nivele de rafinare a predicțiilor:
  - COCOMO de bază
  - COCOMO intermediar
  - COCOMO detaliat

# COCOMO 2

- Propus de Boehm în 1995
- Ia în calcul tehnicile moderne de dezvoltare apărute
  - Prototipizare
  - Dezvoltarea pe componente
  - 4GL (fourth generation language)
- Oferă posibilitatea de a face estimări încă din primele faze ale dezvoltării

# Copyright

- Drepturile de autor reprezintă ansamblul prerogativelor de care se bucură autorii cu referire la operele create; instituția dreptului de autor este instrumentul de protecție a creatorilor și operelor lor
- *Copyright gives the creator of an original work exclusive right for a certain time period in relation to that work, including its* ***publication, distribution and adaptation****; after which time the work is said to enter the public domain.*

# Exclusive rights

- Several exclusive rights typically attach to the holder of a copyright:
  - to produce copies or reproductions of the work and to sell those copies (mechanical rights; including, sometimes, electronic copies: distribution rights)
  - to create derivative works (works that adapt the original work)
  - to perform or display the work publicly (performance rights)
  - to sell or assign these rights to others
  - to transmit or display by radio or video (broadcasting rights)

# Object-Oriented Design

- The most common types of programming are Structured Programming and Object Oriented Programming
- It has become difficult to write a program that does not have the external appearance of both structured programming and object oriented programming
  - Do not have **goto**
  - class based and do not support functions or variables that are not within a class
- Programs may look structured and object oriented, but looks can be decieving

# OOD principles for classes

- The **S**ingle Responsibility Principle
- The **O**pen Closed Principle
- The **L**iskov Substitution Principle
- The **I**nterface Segregation Principle
- The **D**ependency Inversion Principle

- SOLID

# OOD principles for deliverables

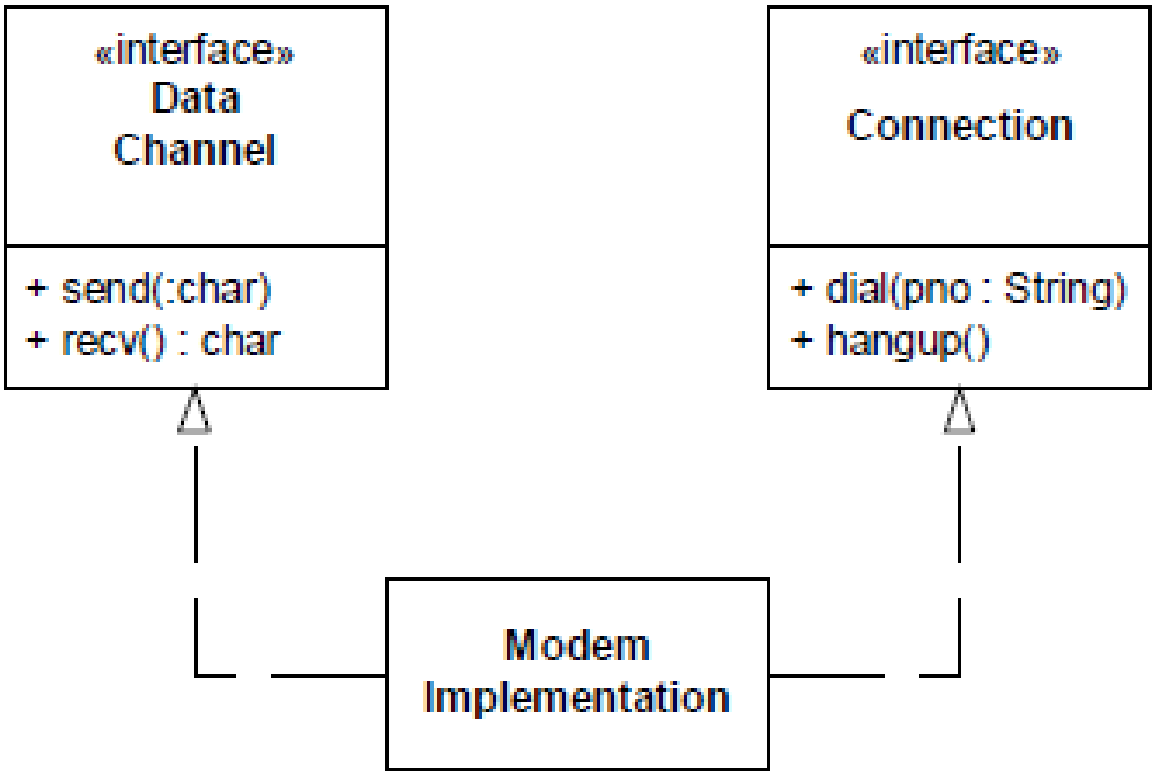
- Cohesion
  - The Release Reuse Equivalency Principle
  - The Common Closure Principle
  - The Common Reuse Principle
- Coupling
  - Acyclic Dependencies Principle
  - The Stable Dependencies Principle
  - The Stable Abstractions Principle

# The Single Responsibility Principle

- *A class should have one, and only one, reason to change.*
- A responsibility to is “a reason for change.”
- Each responsibility is an axis of change.

```
interface Modem
   {
   public void dial(String pno);
   public void hangup();
   public void send(char c);
   public char recv();
   }
```

# The Single Responsibility Principle



# The Single Responsibility Principle

- Should these responsibilities be separated?
  - If the implementations for the communication and connection management change independently, separate
  - If the implementations only change together, do not separate
- Corollary: An axis of change is only an axis of change if the changes actually occur.

# The Open Closed Principle

- *You should be able to extend a class' behavior without modifying it.*
- Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.
- The primary mechanisms behind the Open-Closed principle are abstraction and polymorphism.

# The Open Closed Principle

```
interface Shape {
public void Draw();
};
class Square implements Shape {
public void Draw();
};
class Circle implements Shape {
public void Draw();
};
void DrawAllShapes(List<Shape> list)
{
for (Shape it: list)
it.Draw();
}
```

# The Open Closed Principle

- **Conventions**
  - **Make all member variables private.** Member variables should never be known to any other class, including derived classes.
  - **No Global Variables -- Ever.** No module that depends upon a global variable can be closed against any other module that might write to that variable.
  - **Run Time Type Identification is Dangerous.** If a use if RTTI does not violate the open-closed principle, it is safe.

# The Liskov Substitution Principle

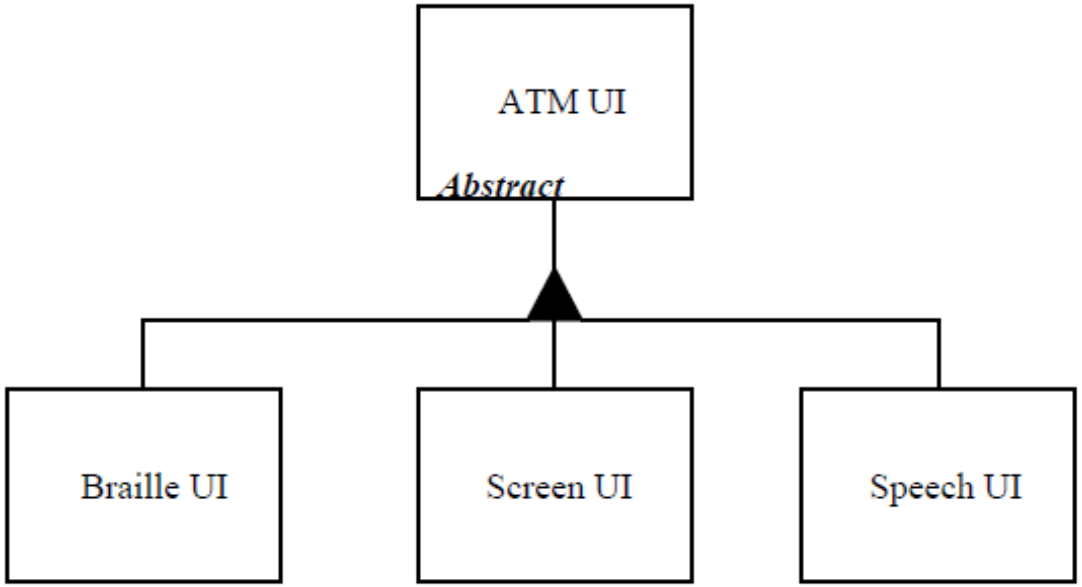
- *Derived classes must be substitutable for their base classes.*
- Makes applications more maintainable, reusable and robust
- If there is a function which does not conform to the LSP, then that function uses a reference to a base class, but must know about all the derivatives of that base class.
- Such a function violates the Open-Closed principle

# The Liskov Substitution Principle

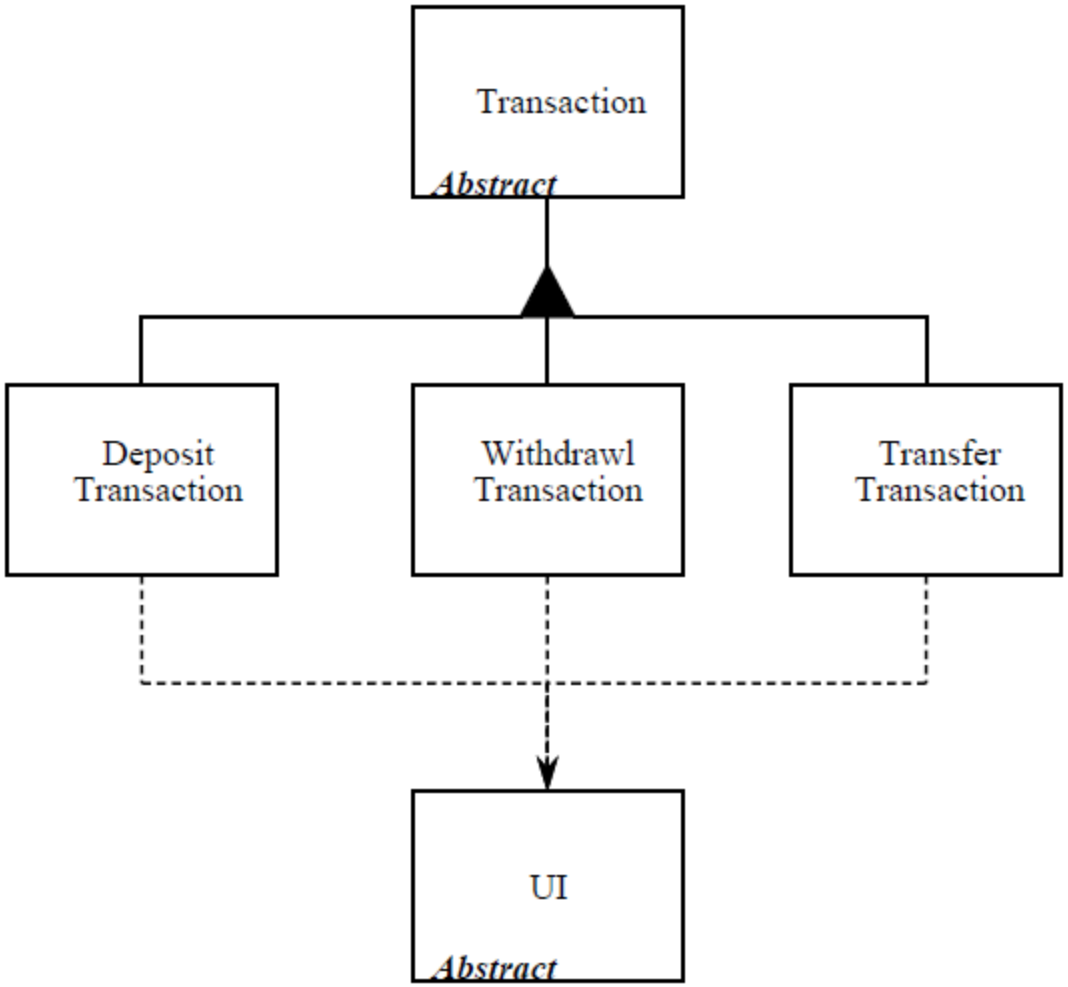
- The Liskov Substitution Principle (Design by Contract) is an important feature of all programs that conform to the Open-Closed principle.
- Only when derived types are completely substitutable for their base types functions which use those base types can be reused with impunity, and the derived types can be changed with impunity.

# The Interface Segregation Principle

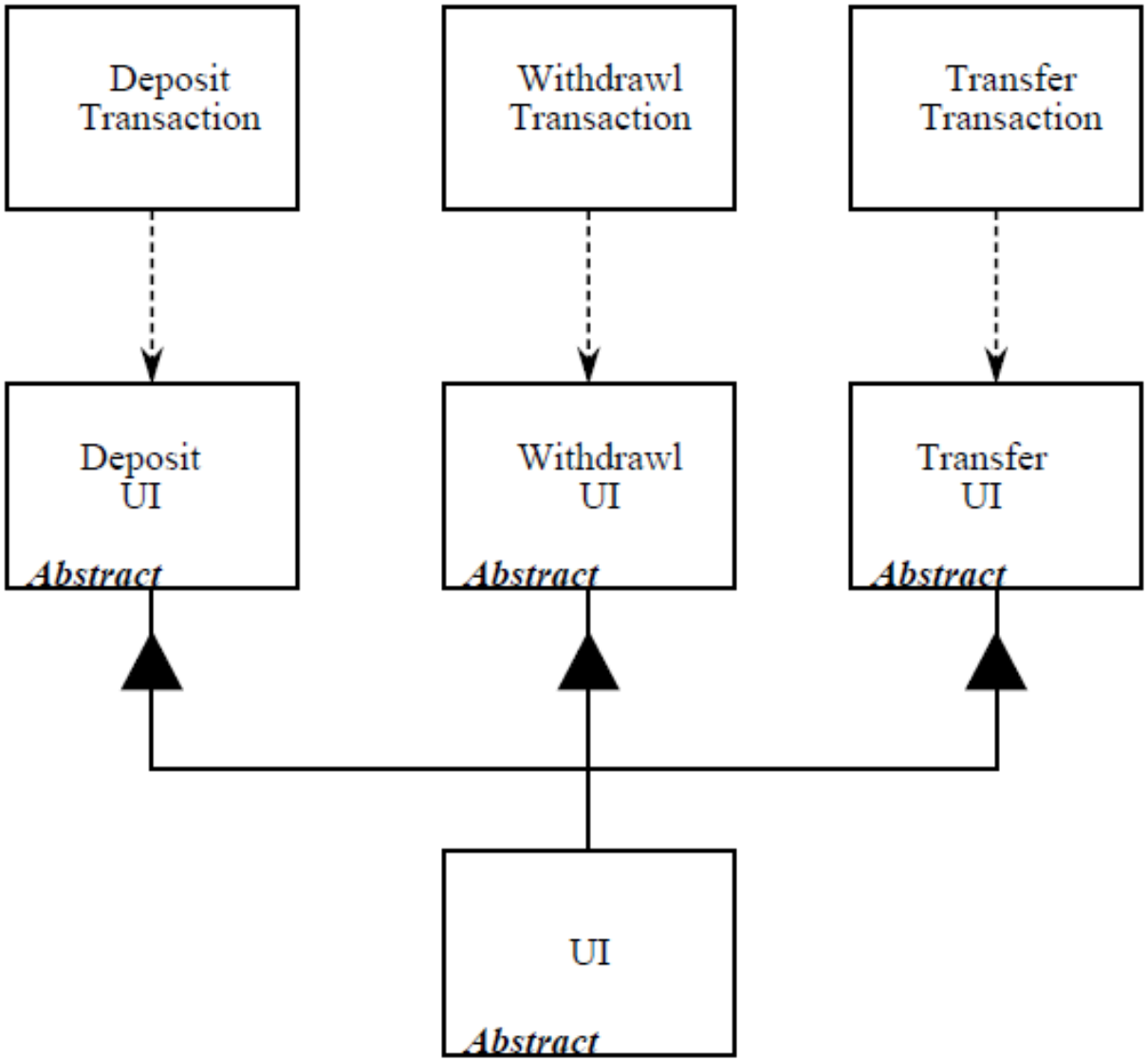
- *Clients should not be forced to depend upon interfaces that they do not use.*
- *Make fine grained interfaces that are client specific.*



# The Interface Segregation Principle



# The Interface Segregation Principle



# The Interface Segregation Principle

- “fat interfaces”; i.e. interfaces that are not specific to a single client must be avoided
- Fat interfaces lead to inadvertent couplings between clients that ought otherwise to be isolated
- By making use of the ADAPTER pattern, fat interfaces can be segregated into abstract base classes that break the unwanted coupling between clients

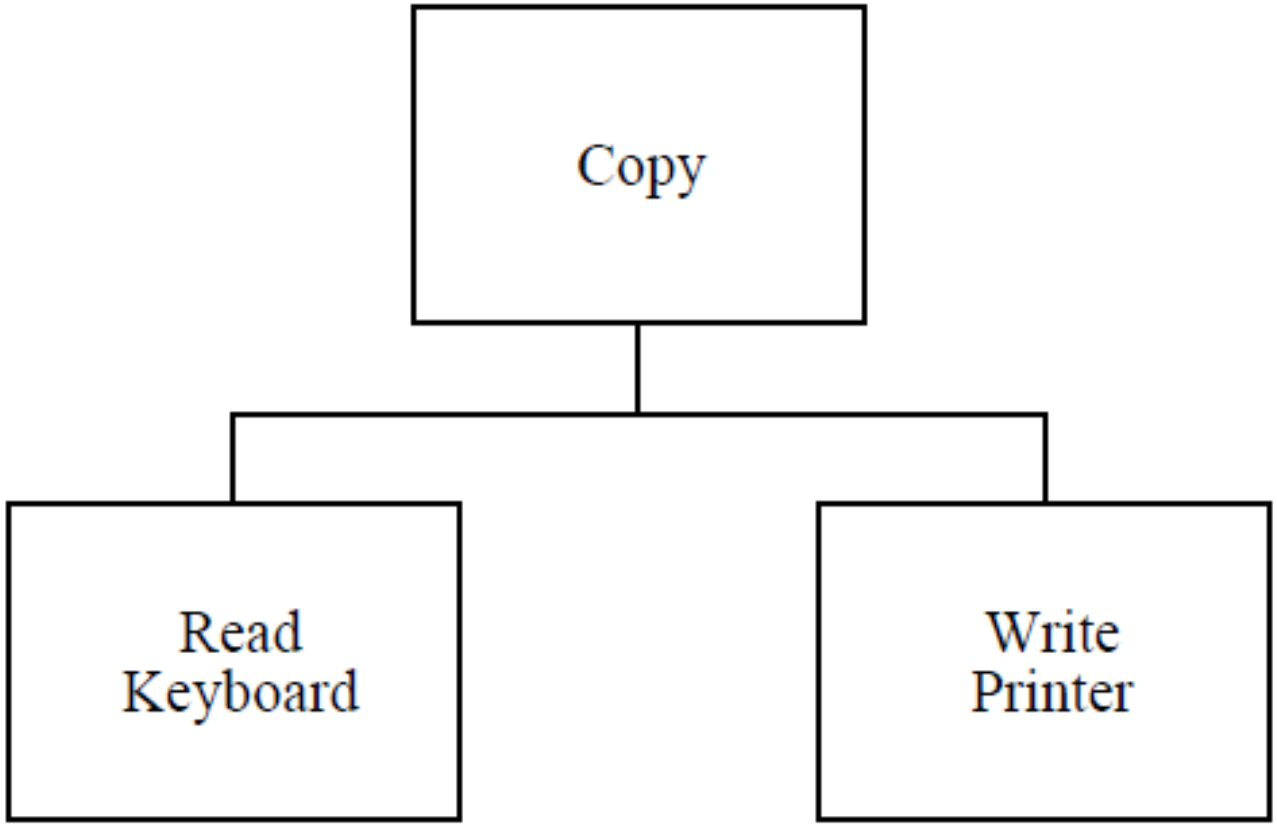
# The Dependency Inversion Principle

- What is it that makes a design bad?
  - most software eventually degrades to the point where someone will declare the design to be unsound
  - Because of the lack of a good working definition of “bad” design.

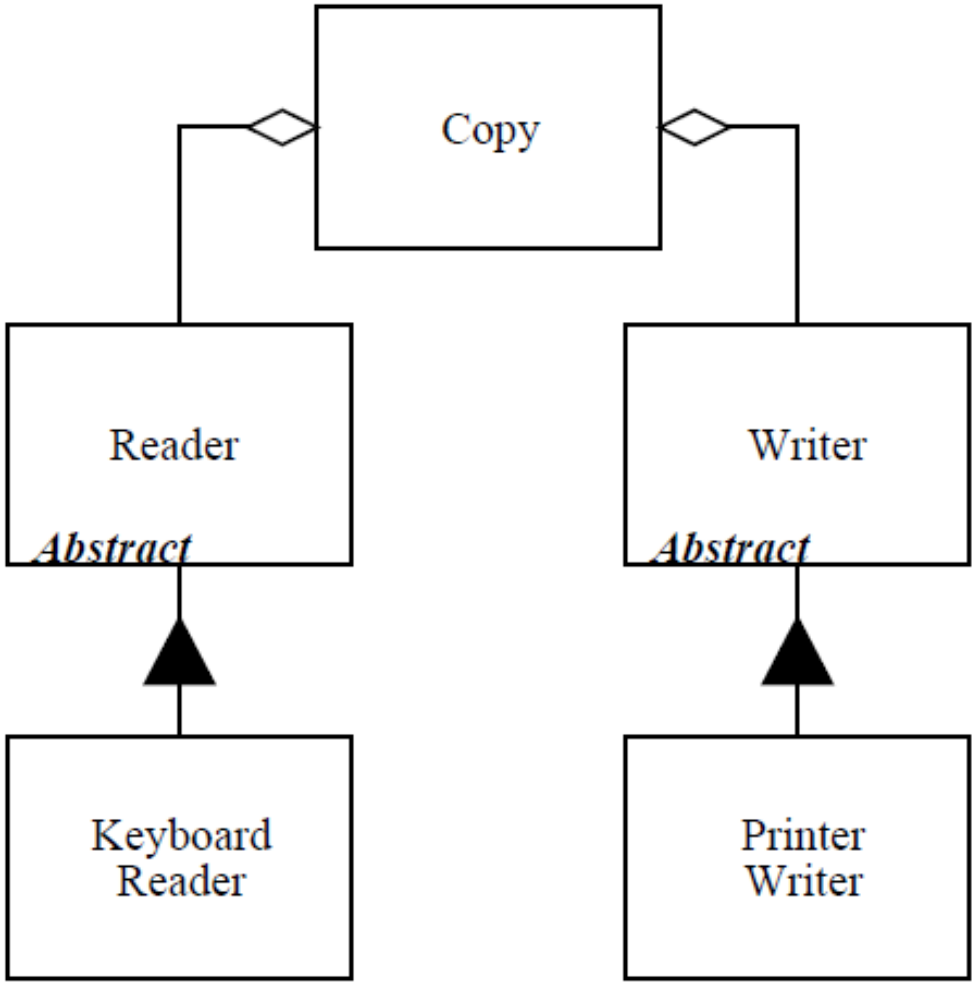
# The Dependency Inversion Principle

- The Definition of a “Bad Design”
  - It is hard to change because every change affects too many other parts of the system. (Rigidity)
  - When you make a change, unexpected parts of the system break. (Fragility)
  - 3. It is hard to reuse in another application because it cannot be separated from the current application. (Immobility)

# The Dependency Inversion Principle



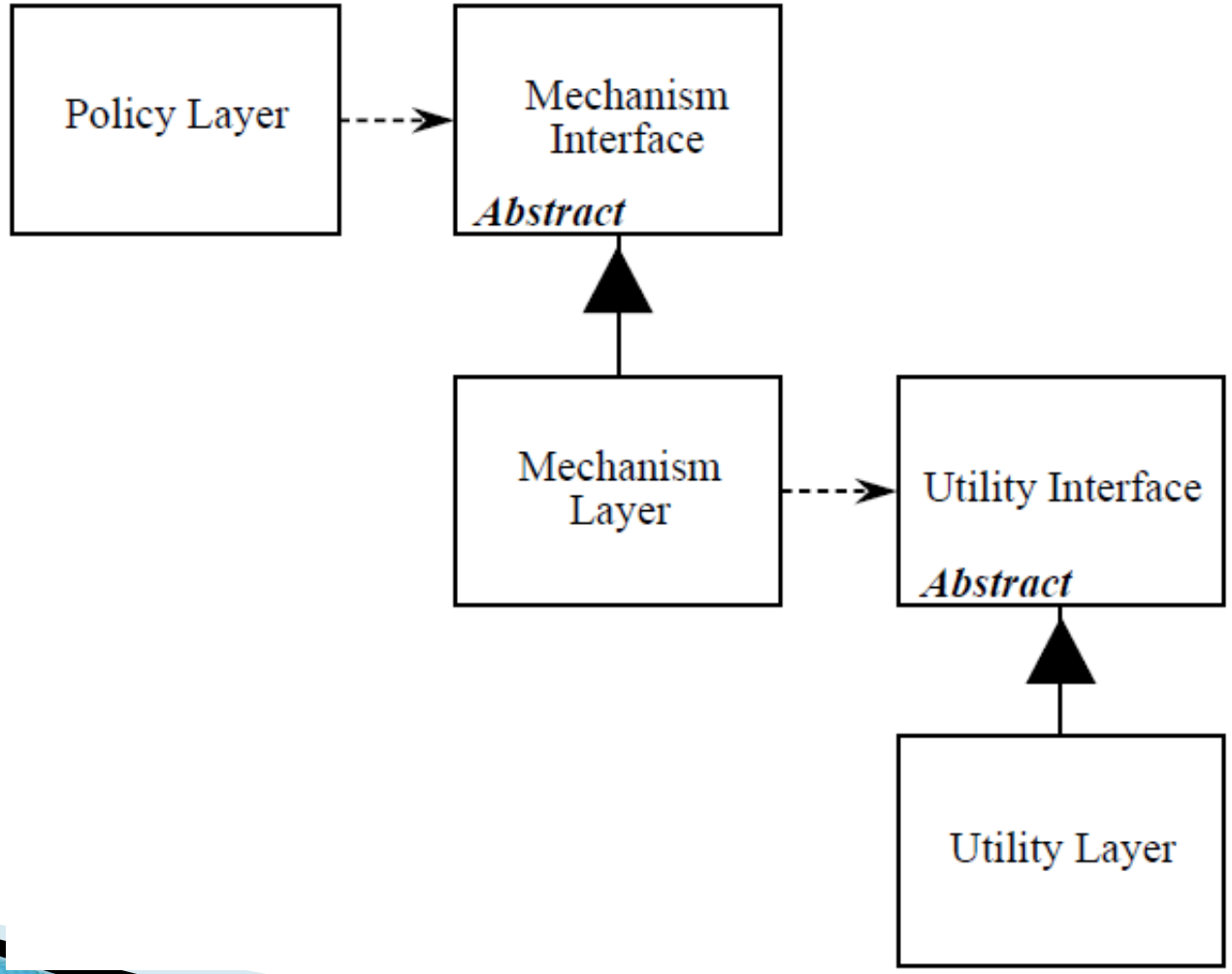
# The Dependency Inversion Principle



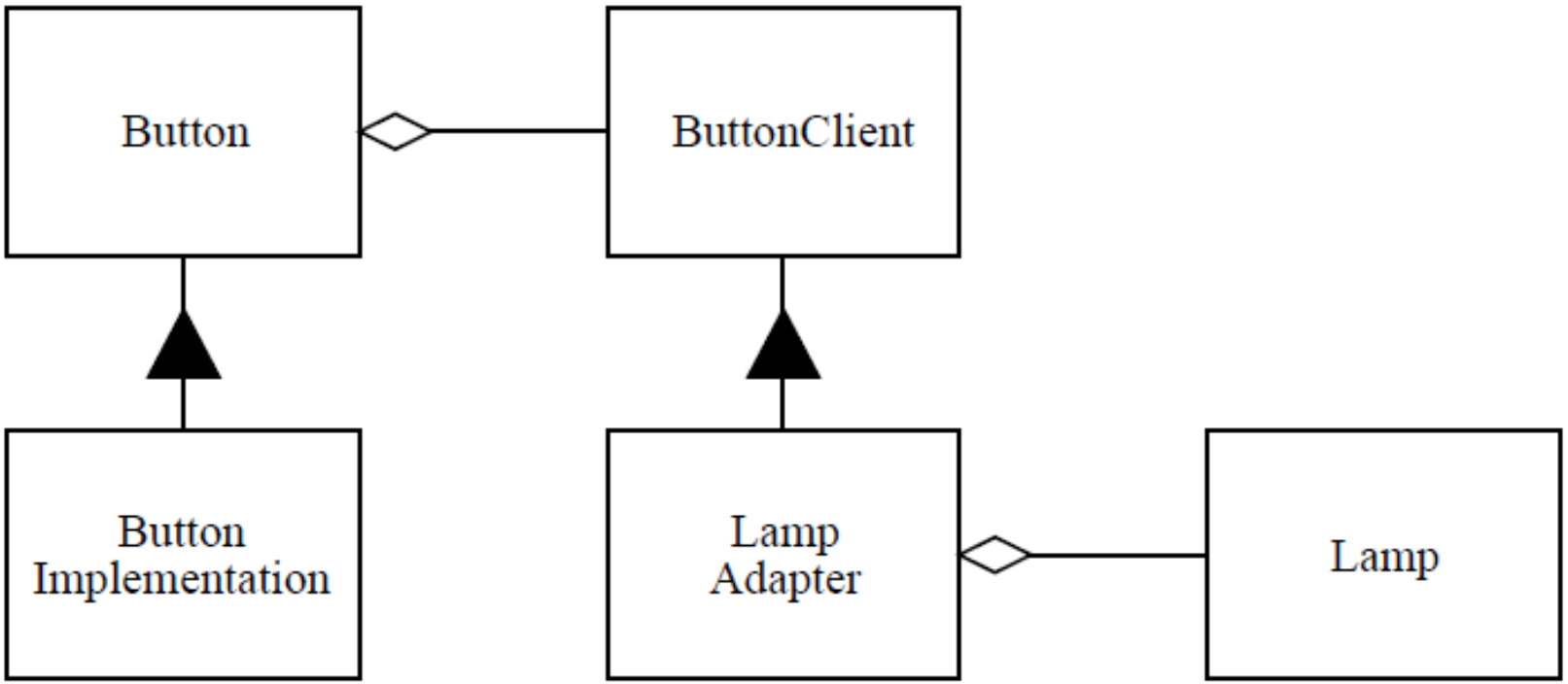
# The Dependency Inversion Principle

- A. High level modules should not depend upon low level modules. Both should depend upon abstractions.
- B. Abstractions should not depend upon details. Details should depend upon abstractions.

# The Dependency Inversion Principle – Layering



# The Dependency Inversion Principle – Layering



# Granularity (Cohesion)

- As software applications grow in size and complexity they require some kind of high level organization.
- The class is too finely grained to be used as an organizational unit for large applications.
- Something “larger” than a class is needed => packages.

# 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 are packages physically represented? In the programming language? In the development environment?
- Once created, how will we use these packages?

# The Reuse/Release Equivalence Principle

- Code copying vs. code reuse
- I reuse code if, and only if, I never need to look at the source code. The author is responsible for maintenance
  - I am the customer
  - When the libraries that I am reusing are changed by the author, I need to be notified
  - I may decide to use the old version of the library for a time
  - I will need the author to make regular releases of the library
  - I can reuse nothing that is not also released

# The Reuse/Release Equivalence Principle

- The granule of reuse is the granule of release. Only components that are released through a tracking system can be effectively reused. This granule is the package.

# The Common Reuse Principle

- *The classes in a package are reused together. If you reuse one of the classes in a package, you reuse them all.*
- Which classes should be placed into a package?
  - Classes that tend to be reused together belong in the same package.
- Packages to have physical representations that need to be distributed.

# The Common Reuse Principle

- I want to make sure that when I depend upon a package, I depend upon every class in that package or I am wasting effort.

# The Common Closure Principle

- 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.
- 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.

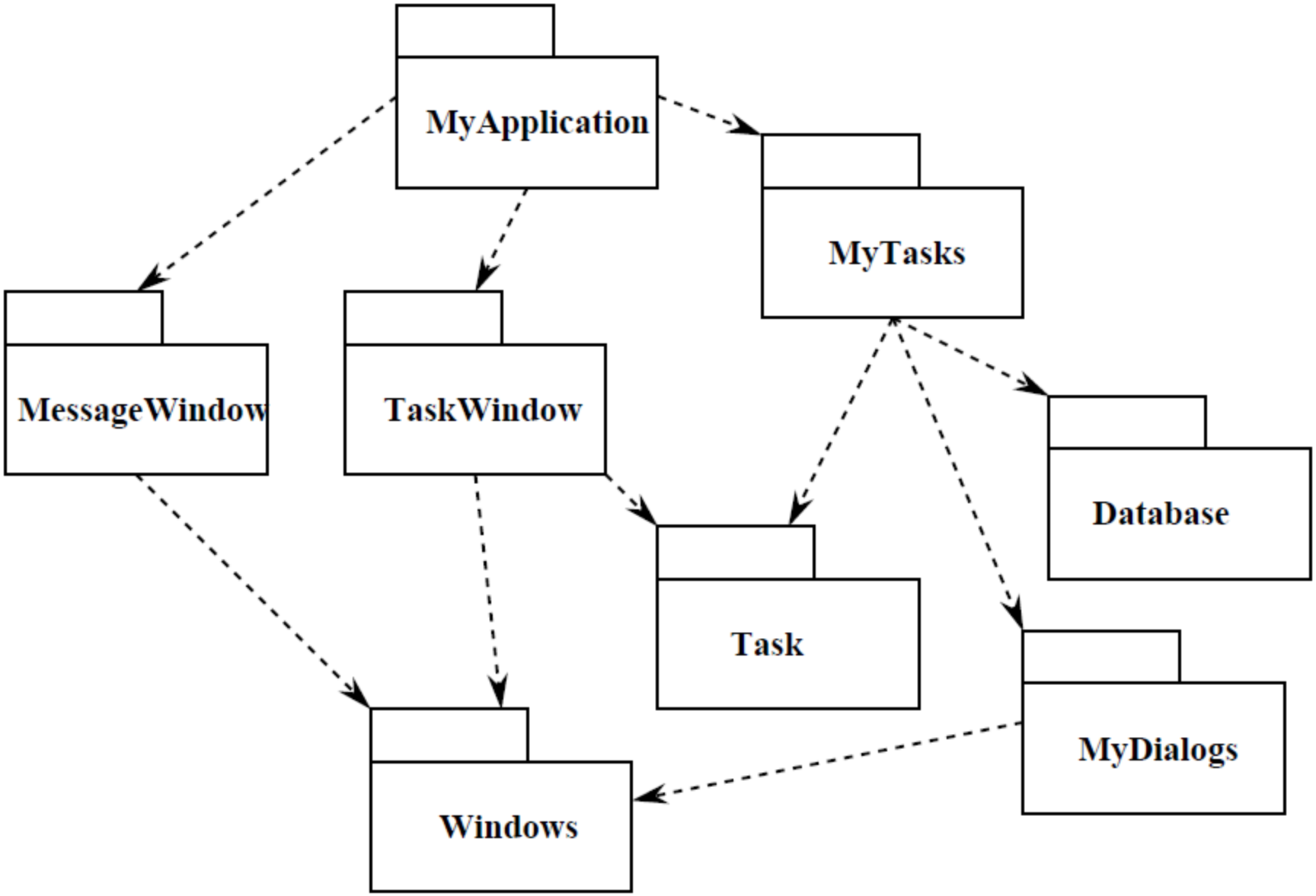
# The Acyclic Dependencies Principle

- The morning after syndrome: you make stuff work and then gone home; next morning it longer works? Why? Because somebody **stayed later than you**!
- Many developers are modifying the same source files.
- Partition the development environment into releasable packages
- You must *manage the dependency structure of the packages*

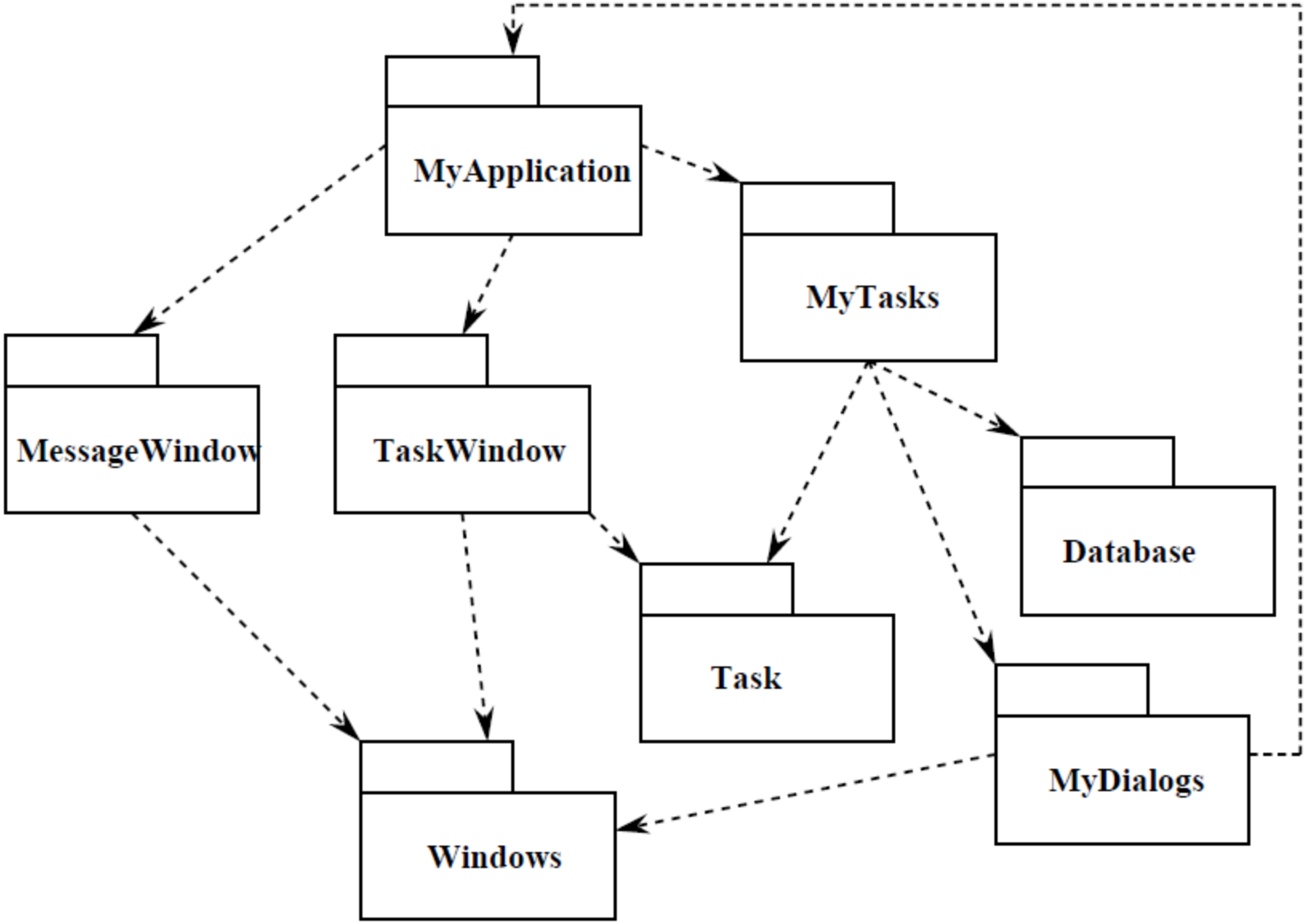
# The Acyclic Dependencies Principle

- The dependency structure between packages must be a directed acyclic graph (DAG). That is, there must be no cycles in the dependency structure.

# The Acyclic Dependencies Principle



# The Acyclic Dependencies Principle



# The Acyclic Dependencies Principle

- Breaking the Cycle
  - Apply the Dependency Inversion Principle (DIP). Create an abstract base class
  - Create a new package that both MyDialogs and MyApplication depend upon. Move the class(es) that they both depend upon into that new package.
- The package structure cannot be designed from the top down.

# Stability

- Not easily moved
- A measure of the difficulty in changing a module
- Stability can be achieved through
  - Independence
  - Responsibility
- The most stable classes are Independent and Responsible. They have no reason to change, and lots of reasons not to change.

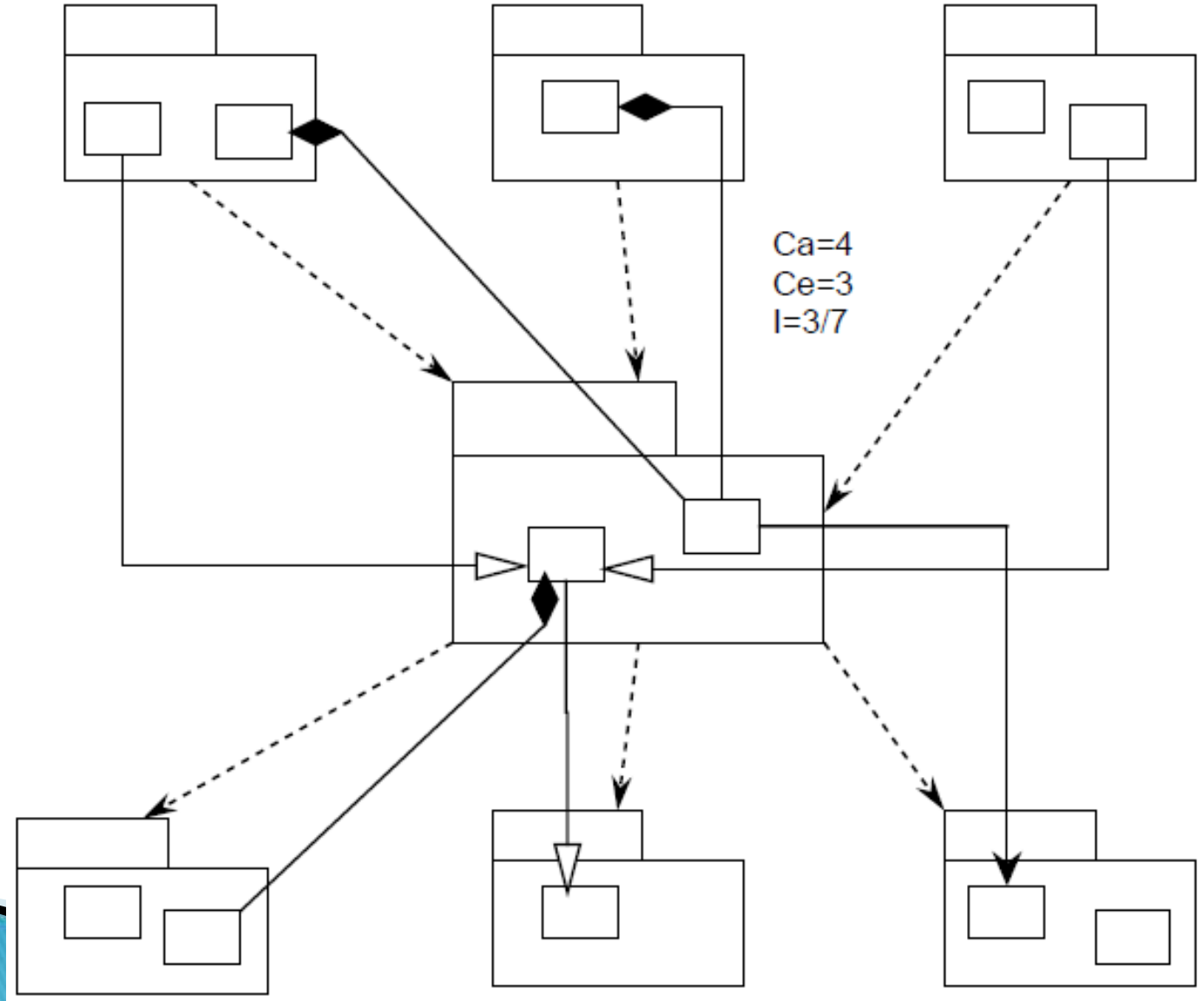
# The Stable Dependencies Principle

- The dependencies between packages in a design should be in the direction of the stability of the packages. A package should only depend upon packages that are more stable that it is.
- we ensure that modules that are designed to be instable are not depended upon by modules that are more stable

# Stability Metrics

- **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 : (Ce/(Ca+Ce)) I=0 maximally stable package. I=1 maximally instable package.

# Stability Metrics



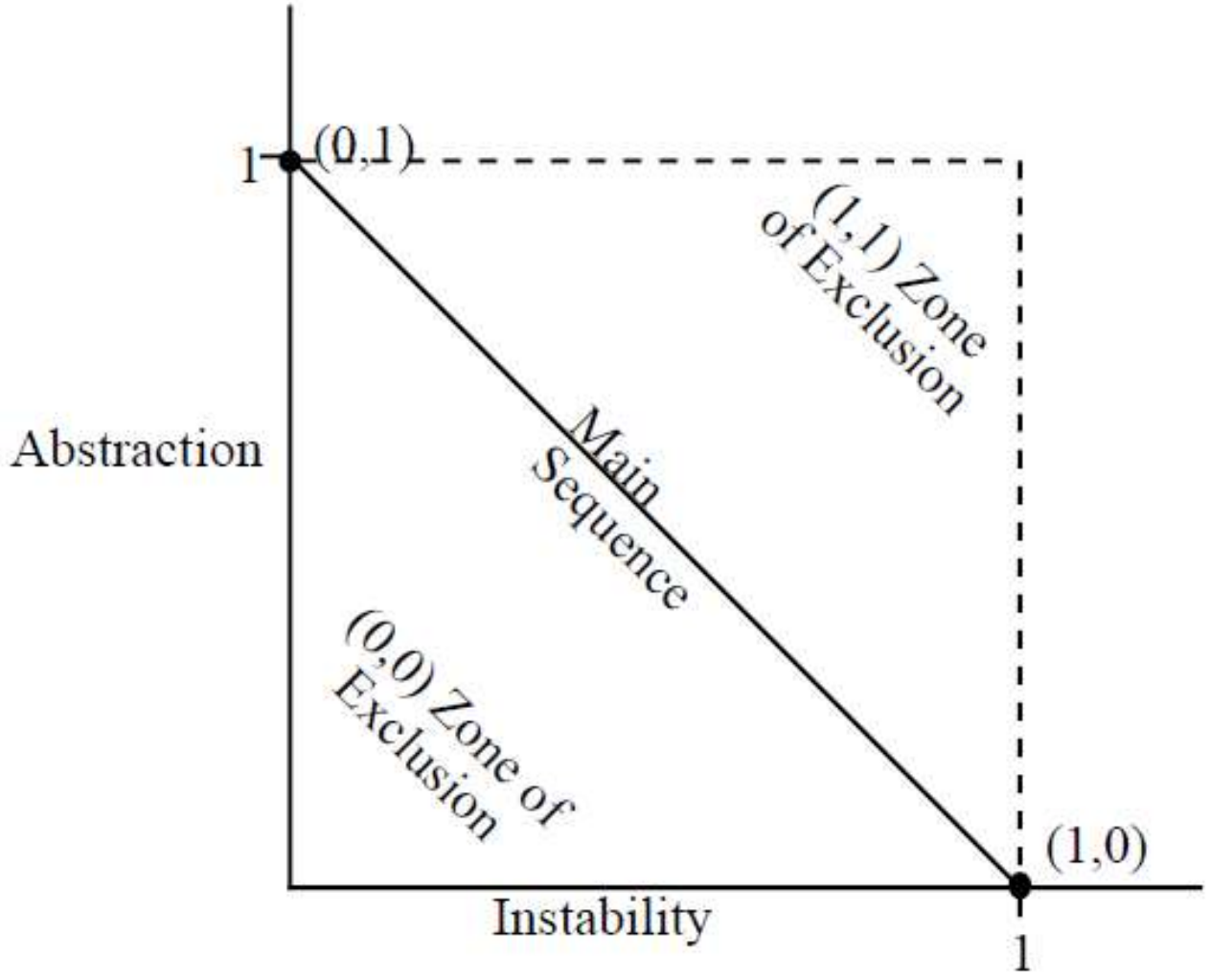
# The Stable Dependencies Principle

- Not all packages should be stable
- The software the encapsulates the high level design model of the system should be placed into stable packages
- How can a package which is maximally stable (I=0) be flexible enough to withstand change?
  - classes that are flexible enough to be extended without requiring modification => abstract classes

# The Stable Abstractions Principle

- 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.
- Abstraction (A) is the measure of abstractness in a package. $A = AC/TC$

# Main Sequence



# Software craftsmanship

- Software craftsmanship is an approach to software development that emphasizes the coding skills of the software developers themselves.
- A response to the problems of the mainstream software industry, including financial concerns over developer accountability.
- A previously rigorous approach becomes increasingly engineering in nature

# Manifesto for Agile Software Development

- We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- **Individuals and interactions** over processes and tools
- **Working software** over comprehensive documentation
- **Customer collaboration** over contract negotiation
- **Responding to change** over following a plan
  - That is, in pursuit of the items on the left we have found the items on the right to be indispensable.

# Manifesto for Software Craftmanship

- As aspiring Software Craftsmen we are raising the bar of professional software development by practicing it and helping others learn the craft. Through this work we have come to value:

- Not only working software, but also **well-crafted software**
- Not only responding to change, but also **steadily adding value**
- Not only individuals and interactions, but also **a community of professionals**
- Not only customer collaboration, but also **productive partnerships**
  - That is, while there is value in the items on the right, we value the items on the left more.

# Bibliography

- All images and examples courtesy of Robert Cecil Martin
- Robert C. Martin, Engineering Notebook columns for The C++ Report

# Useful Links

- Agile Manifesto: http://www.agilemanifesto.org/
- Software Craftsmanship Manifesto: http://manifesto.softwarecraftsmanship.org/