University of Dublin
Trinity College

CS7CS3: Object-Oriented Design Principles

Prof. Siobhán Clarke *(she/her)*
Ext. 2224 – L2.15
www.scss.tcd.ie/Siobhan.Clarke/

Before we start…

Object-Oriented Design Principles

Why?

(let's think about the overall process)

Engineering systems

"The application of science to the design, building and use of machines, constructions etc"

Oxford Reference English dictionary (1998)

Implications

- Repeatable – should be able to re-use tools and techniques across projects
- Responsible – should take account of best practices and ethics
- Systematic – should be planned, documented and analysed
- Measurable – should have objective support both for the products *and* for the process itself

What's involved

Requirements

Functional requirements
Non-functional requirements

Specification

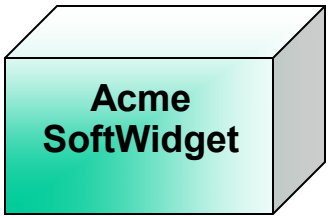
What it must do
What it mustn't do

Design

Architecture
Functional components
Algorithms and data structures

Maintenance

Bug fixes
New features
New platforms
Versions



Coding

Individual components
Synergy

Testing

Black-box testing
White-box testing
Acceptance testing

Deployment

Acceptance
Packaging
Marketing

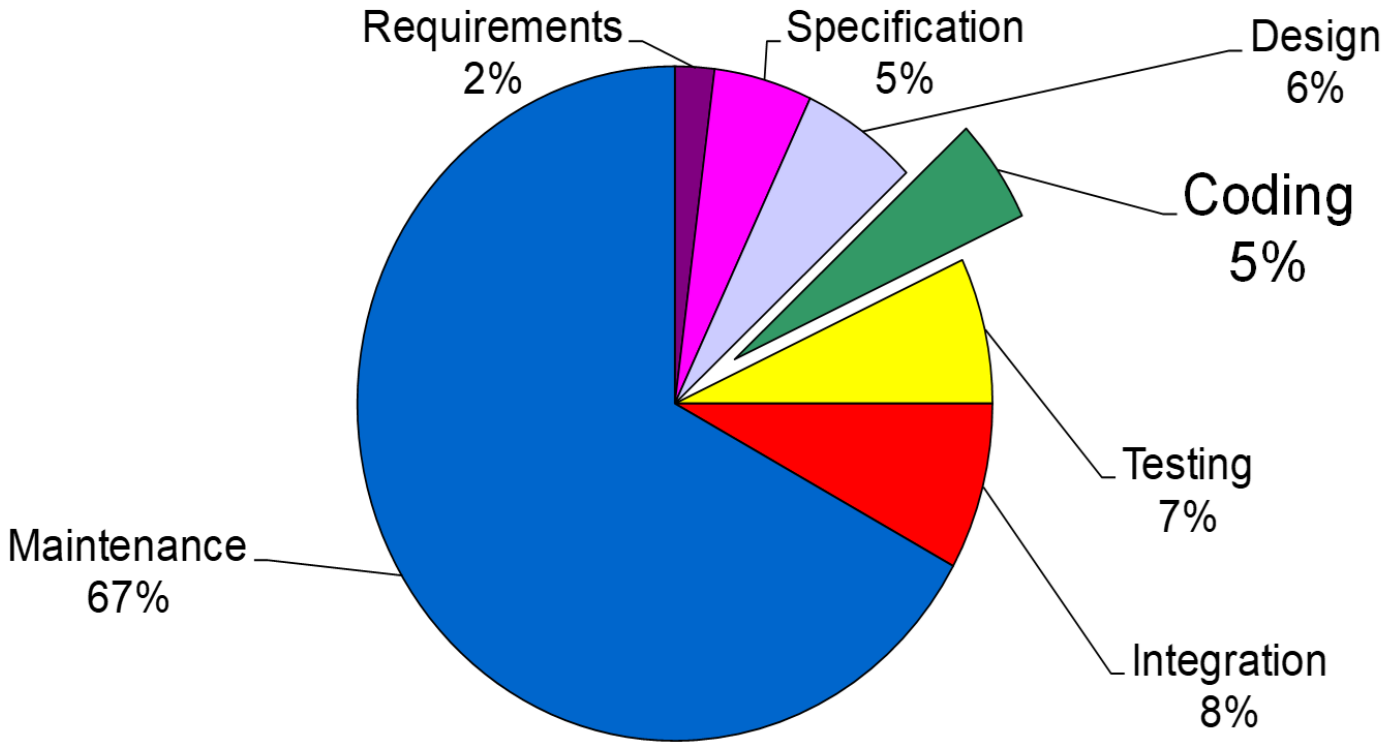
Documentation

Requirements and specification
Design decisions
User documentation

Integration

OS variations
Interworking
Communication

How long each bit takes



From Stephen Schach, *Classical and object-oriented software engineering*, Addison-Wesley (1999)

Object-oriented design

Whenever you think about writing a class, you're doing design

- What functions does the class provide?
- What functions does it rely on?
- What functions can the programmer change?
- What features might change at run-time?
- What features are shared by all instances? What features should be in each instance?

From specification to design

Specification is a precise statement of *what* was agreed to be built

Design is a statement of *how* the system will be built, in technical terms

- The division of functionality between modules/classes/methods
- The relationships between the parts

Design decisions have a major impact on the maintenance and evolution of a system – in other words, **good design** is the key engineering skill

What's to be done?

Where do classes come from?

- Test design – what should this software do? The tests emerge from the requirements, and classes need to be written to pass those tests
- Domain analysis – the nouns in requirements (starting point only!) – the test designs will have come from the requirements
- "Basic" computer science structures – lists, hash tables, …
- Re-used from elsewhere – sometimes have to adopt someone else's view of the world, *e.g.,* Java's view of windowing

There's no substitute for experience

Basic techniques

What classes need to work together to achieve the goals of test?

How is the application's functionality divided into collaborating classes?

What does each class do?

- Its behaviour
- Its assumptions about the world

How is the application's functionality divided into classes?

What specialisations are possible?

- General *versus* specific functions

What can be provided generically, and what must be provided specifically?

What do clients need to know?

- Exposed methods and variables
- Any implications of particular algorithms

Anything exposed limits possible changes

Elements of Bad Design

- Software Rigidity
- Software Fragility
- Software Immobility
- Software Viscosity

(source: article by Robert C. Martin on OOD principles)

Software Rigidity

Rigidity is the tendency for software to be difficult to change, even in simple ways.

Symptom: Every change causes a cascade of subsequent changes in dependent modules.

Effect: When software behaves this way, managers fear (and don't allow) developers fix non-critical problems. This reluctance derives from the fact that they don't know, with any reliability, when the developers will be finished.

Software Fragility

Fragility is the tendency of the software to break in many places every time it is changed. Often the breakage occurs in areas that have no conceptual relationship with the area that was changed.

Symptom: Every fix makes it worse, introducing more problems than are solved.

Effect: Every time mangers/ team leaders authorize a fix, they fear that the software will break in some unexpected way.

Software Immobility

Immobility is the inability to reuse software from other projects or from parts of the same project.

Symptom: A developer discovers that he needs a module that is similar to one that another developer wrote. But the module in question has too much baggage that it depends upon. After much work, the developer discovers that the work and risk required to separate the desirable parts of the software from the undesirable parts are too great to tolerate.

Effect: And so the software is simply rewritten instead of reused.

Software Viscosity

Viscosity is the tendency of the software/ development environment to encourage software changes that are hacks rather than software changes that preserve original design intent.

Symptom: It is easy to do the wrong thing, but hard to do the right thing.

Effect: The software maintainability degenerates due to hacks, workarounds, shortcuts, temporary fixes etc.

Why does bad design happen?

Obvious reasons: lack of design skills/ design practices, changing technologies, time/ resource constraints, domain complexity etc.

Not so obvious:

- Software rotting is a slow process .. Even originally clean and elegant design may degenerate over the months/ years ..
- Unplanned and improper module dependencies creep in; Dependencies go unmanaged.
- Requirements often change in the way the original design or designer did not anticipate ..

Some opinions on clean code:

Bjarne Stroustrup, inventor of C++:

- I like my code to be **elegant** and **efficient**. The logic should be **straightforward** and make it hard for bugs to hide, the **dependencies minimal** to ease maintenance, error handling complete according to an articulated strategy, and **performance close to optimal** so as not to tempt people to make the code messy with unprincipled optimizations. **Clean code does one thing well.**

Some opinions on clean code:

Grady Booch, author of Object-Oriented Analysis and Design with Applications:

- Clean code is **simple and direct**. Clean code reads like well-written prose. Clean code never obscures the **designers' intent** but rather is full of crisp abstractions and straightforward lines of control.

Some opinions on clean code:

Dave Thomas, founder of OTI and godfather of the Eclipse strategy:

- Clean code **can be read, and enhanced by a developer other than its original author**. It has unit and acceptance tests. It has meaningful names. It provides **one way rather than many** ways for doing one thing. It has **minimal** dependencies, which are explicitly defined, and provides a clear and minimal API. Code should be literate since, depending on the language, not all necessary information can be expressed clearly in code alone.

Some opinions on clean code:

Michael Feathers, author of Working Effectively with Legacy Code:

- I could list all of the qualities that I notice in clean code, but there is one overarching quality that leads to all of them. **Clean code always looks like it was written by someone who cares.** There is **nothing obvious that you can do to make it better**. All of those things were thought about by the code's author, and if you try to imagine improvements, you are led back to where you are, sitting in appreciation of the code someone left for you—code written by someone who cared deeply about the craft.

Some opinions on clean code:

Ward Cunningham, inventor of Wiki and Fit, co-inventor of Extreme Programming. The force behind Design Patterns. Smalltalk and OO thought leader.

- You know you are working with clean code when each routine you read turns out to be **pretty much what you expected**. You can call it beautiful code when the code also makes it look like the language was made for the problem.

OO Design Principles

- **The Open/Closed Principle (OCP)**
- **The Liskov Substitution Principle (LSP)**
- **The Dependency Inversion Principle (DIP)**
- **The Interface Segregation Principle (ISP)**

Class-related principles

- **The Reuse/Release Equivalency Principle (REP)**
- **The Common Closure Principle (CCP)**
- **The Common Reuse Principle (CRP)**

Package cohesion principles

- **The Acyclic Dependencies Principle (ADP)**
- **The Stable Dependencies Principle (SDP)**
- **The Stable Abstractions Principle (SAP)**

Package coupling principles

(source in detail: a series of articles by Robert C. Martin on OOD principles, published in *The C++ Report*, 1996)

The Open/Closed principle (OCP)

"A module should be open for extension but closed for modification."

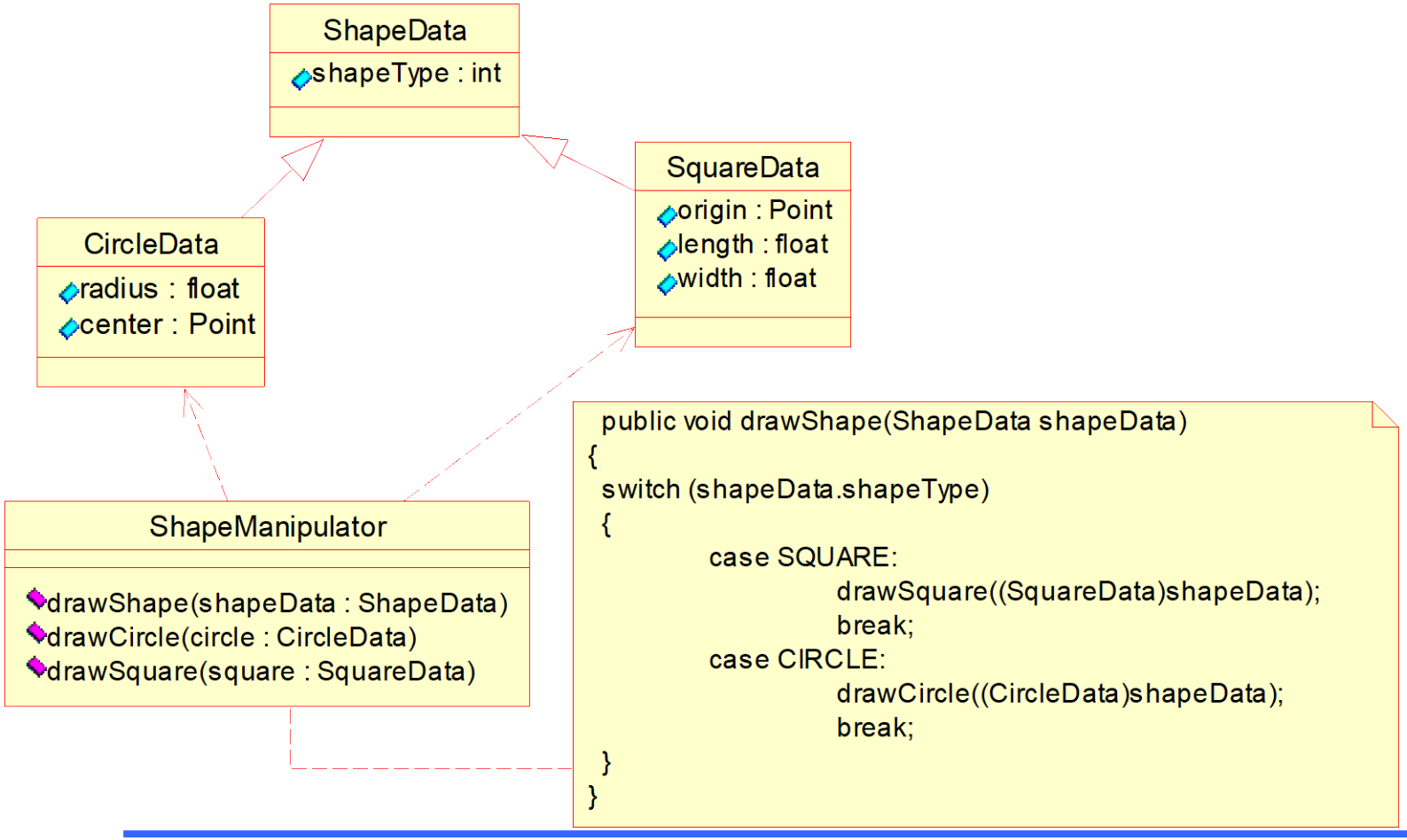
We should write our modules so that they can be extended, without requiring them to be modified. In other words, we want to be able to extend (add to) what the modules do, without changing the existing source code of the modules.

How?: Abstraction and Polymorphism

(originated from Bertrand Meyer)

(Open/Closed)

OCP Example



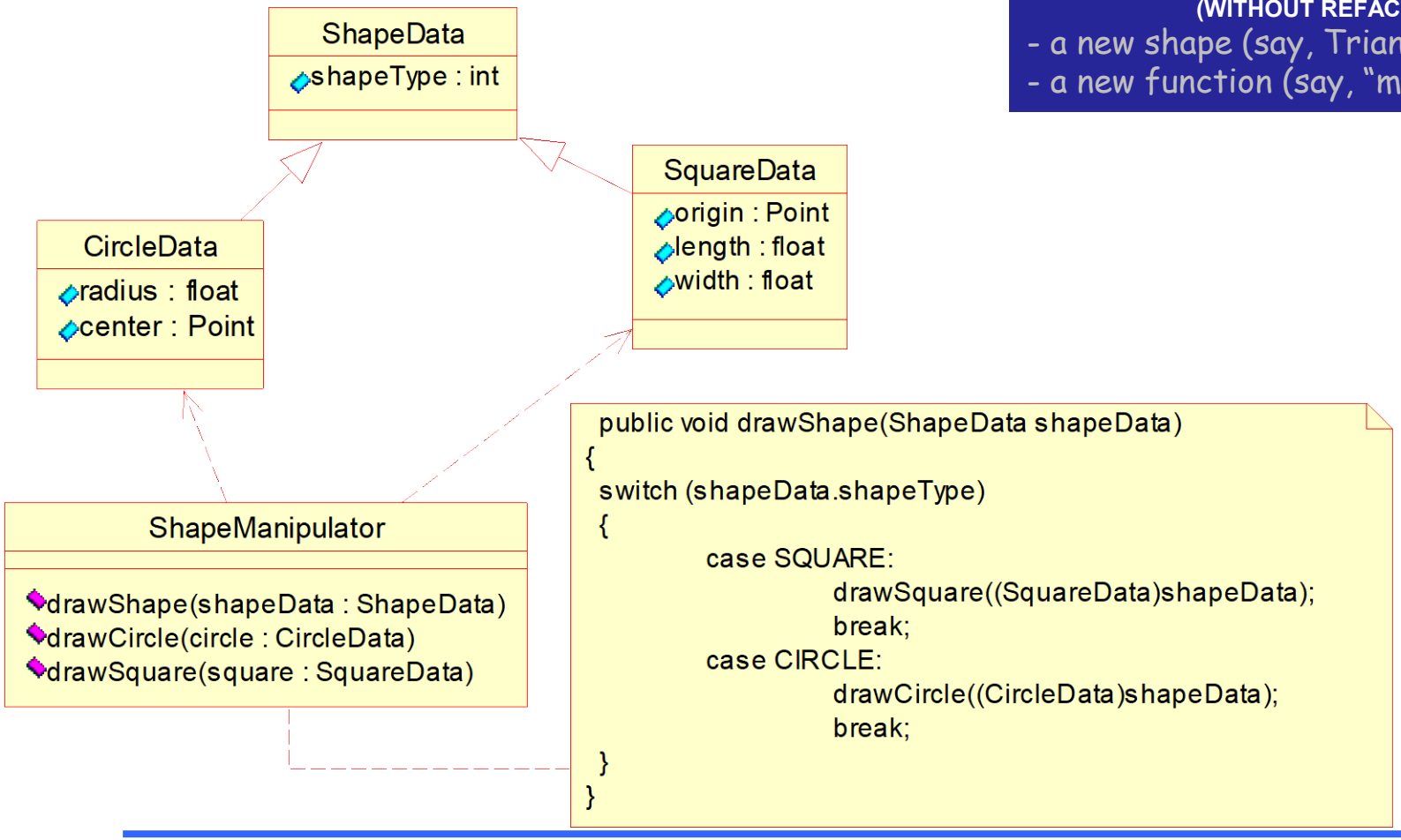
OCP Example

To illustrate effect of bad design

What if I add

(WITHOUT REFACTORING STRUCTURE):

- a new shape (say, Triangle)?
- a new function (say, "move")?



(Open/Closed)

OCP Discussion

To create a new shape (e.g., a Triangle), have to (at least) modify `drawShape()`

In a complex application, switch/case statement above repeated over and over again for every kind of operation that can be performed on a shape.

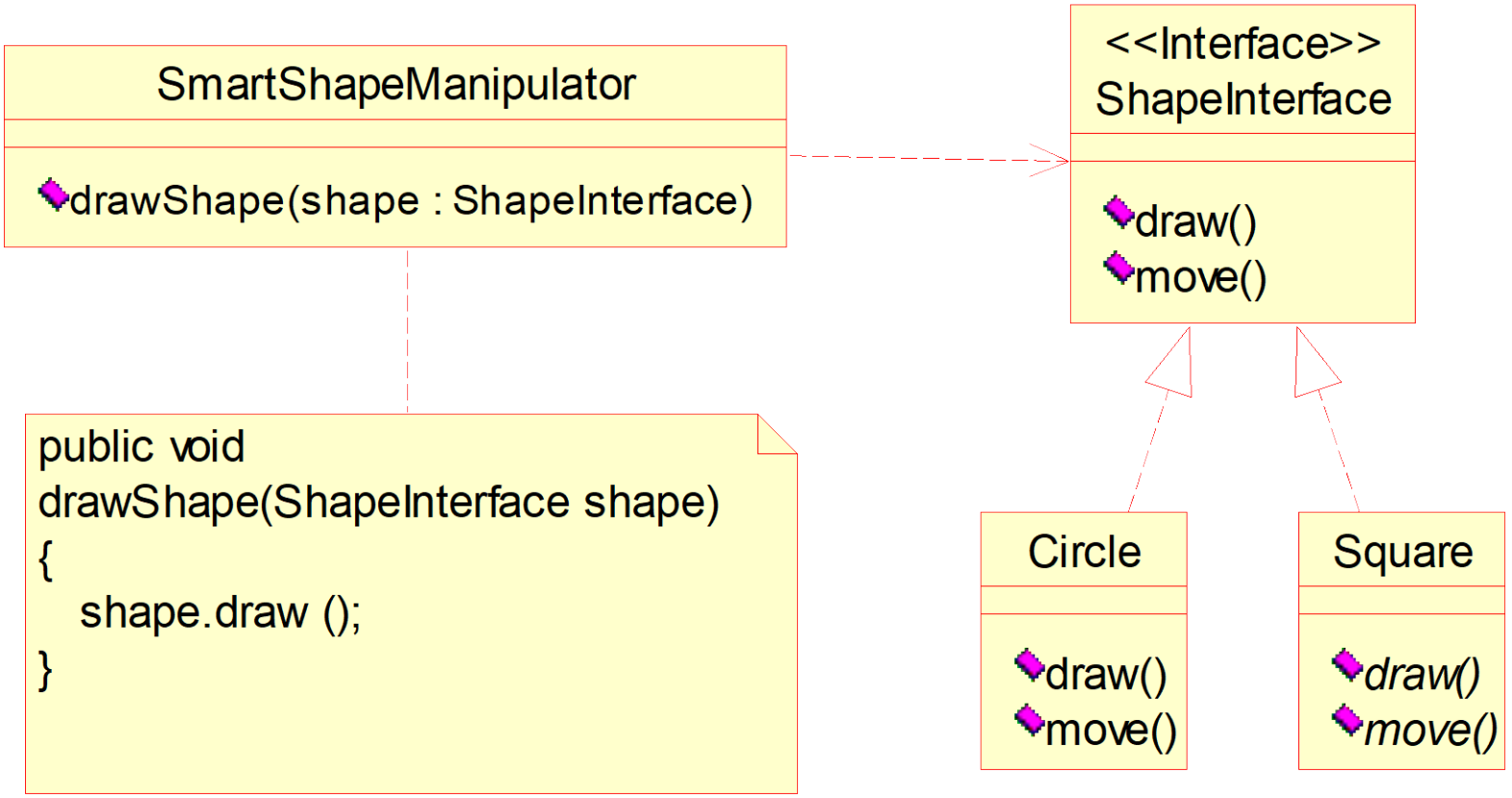
Worse, every module that has such a switch/case statement has a dependency on every possible shape that can be drawn

--> one of the shapes is modified in any way, the modules all need recompilation, and possibly modification

However, when the majority of modules in an application conform to the open/closed principle, then new features can be added to the application by adding new code rather than by changing working code. Thus, the working code is not exposed to breakage.

(Open/Closed)

OCP Example



The Liskov Substitution Principle (LSP)

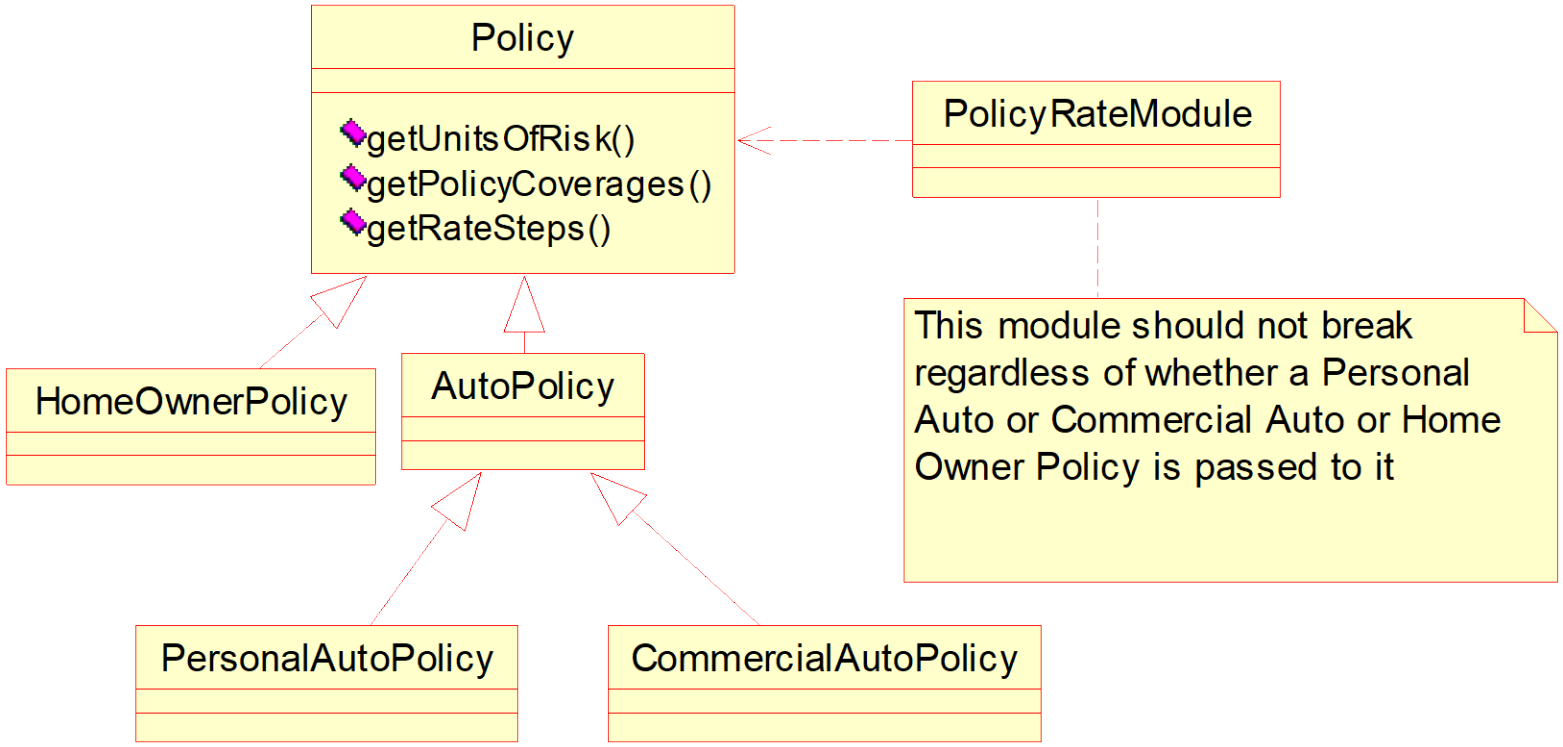
"Subclasses should be substitutable for their base classes."

A client of a base class should continue to function properly if a derivative of that base class is passed to it.

In other words, if some function takes an argument of type `Policy`, then it should be legal to pass in an instance of `PersonalAutoPolicy` to that provided `PersonalAutoPolicy` is directly/ indirectly derived from `Policy`.

(Liskov Substitution)

LSP Example



(Liskov Substitution)

LSP Discussion

Another example - Is a Square a Rectangle ?

(Liskov Substitution)

LSP Discussion

Another example - Is a Square a Rectangle ?

Mathematically, yes. Behaviourally, a Square is not a Rectangle and it is *behaviour* that software is really all about.

It is only when derived types are completely substitutable for their base types that functions which use those base types can be reused with impunity, and the derived types can be changed with impunity.

This implies a “contract” with a client that should be explicit.

The Dependency Inversion Principle (DIP)

"Depend upon Abstractions. Do not depend upon concretions."

Dependency Inversion is the strategy of depending on interfaces or abstract functions and classes, rather than on concrete functions and classes.

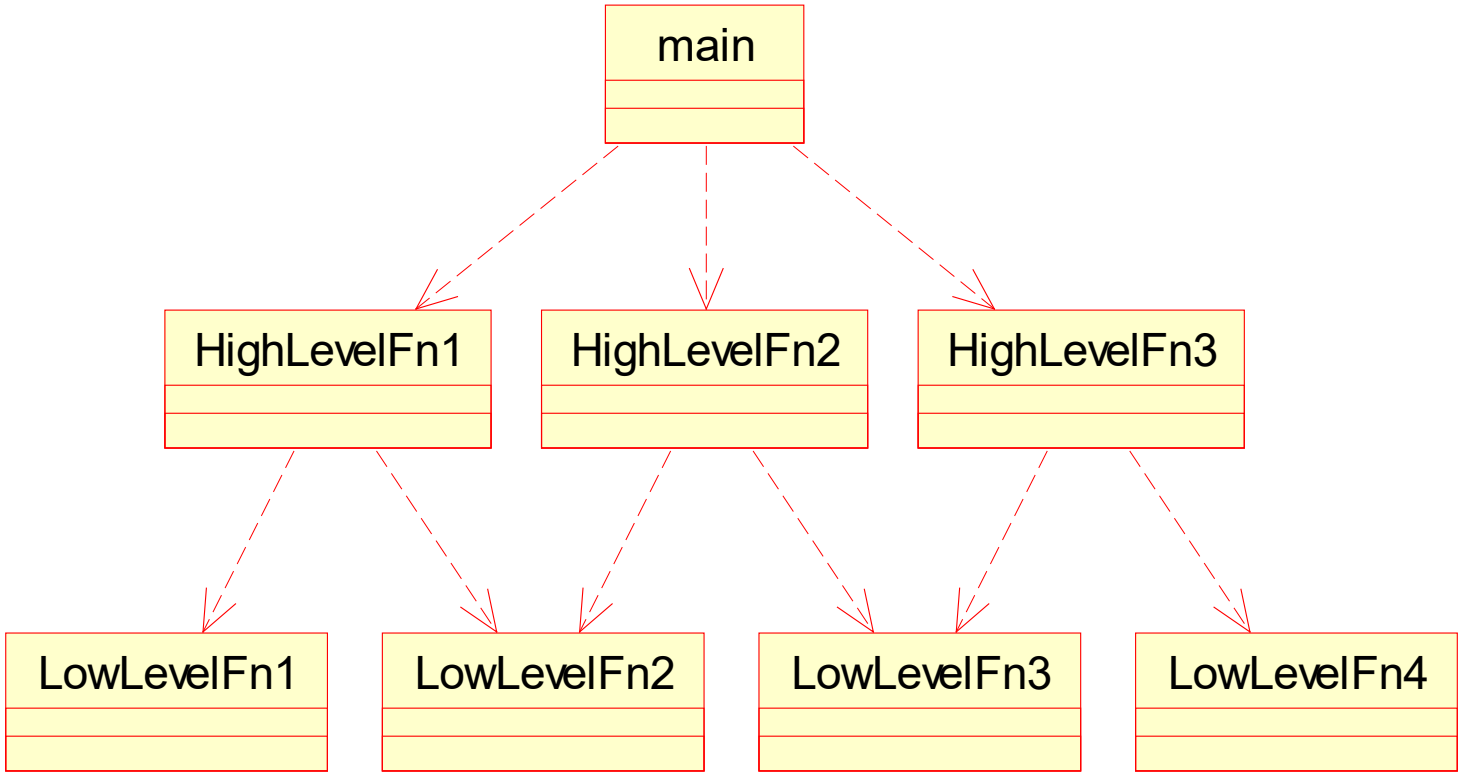
Every dependency in the design should target an interface, or an abstract class. No dependency should target a concrete class.

(mechanism for achieving OCP)

(Dependency Inversion)

DIP Example

(Dependency Structure of a Procedural Architecture)



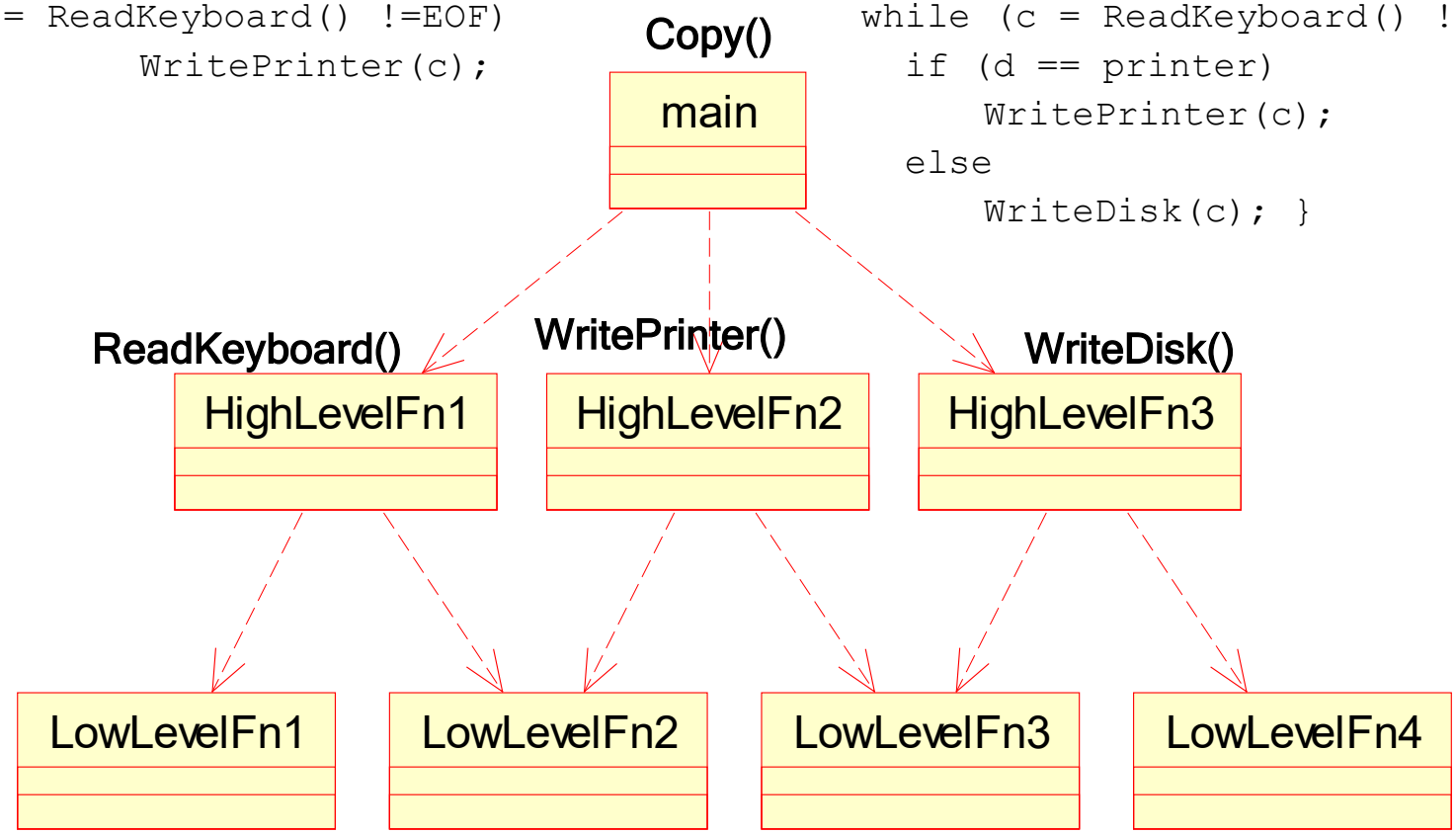
(Dependency Inversion)

DIP Example

(Dependency Structure of a Procedural Architecture)

```
void Copy(Device d) {
  while (c = ReadKeyboard() !=EOF)
                  WritePrinter(c);
}
```

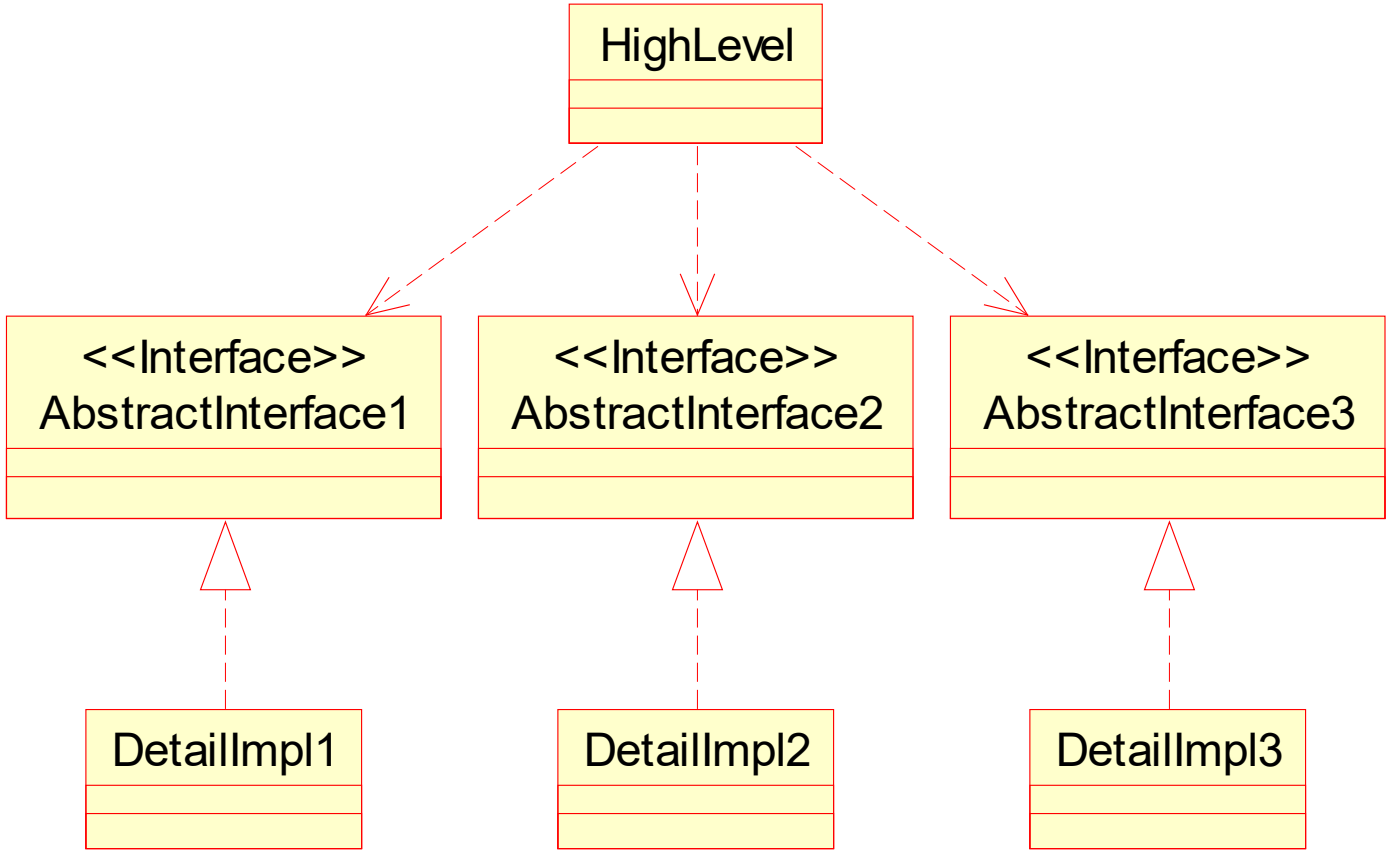
```
void Copy(Device d) {
   while (c = ReadKeyboard() !=EOF)
     if (d == printer)
        WritePrinter(c);
     else
        WriteDisk(c); }
```



(Dependency Inversion)

DIP Example

(Dependency Structure of an Object Oriented Architecture)

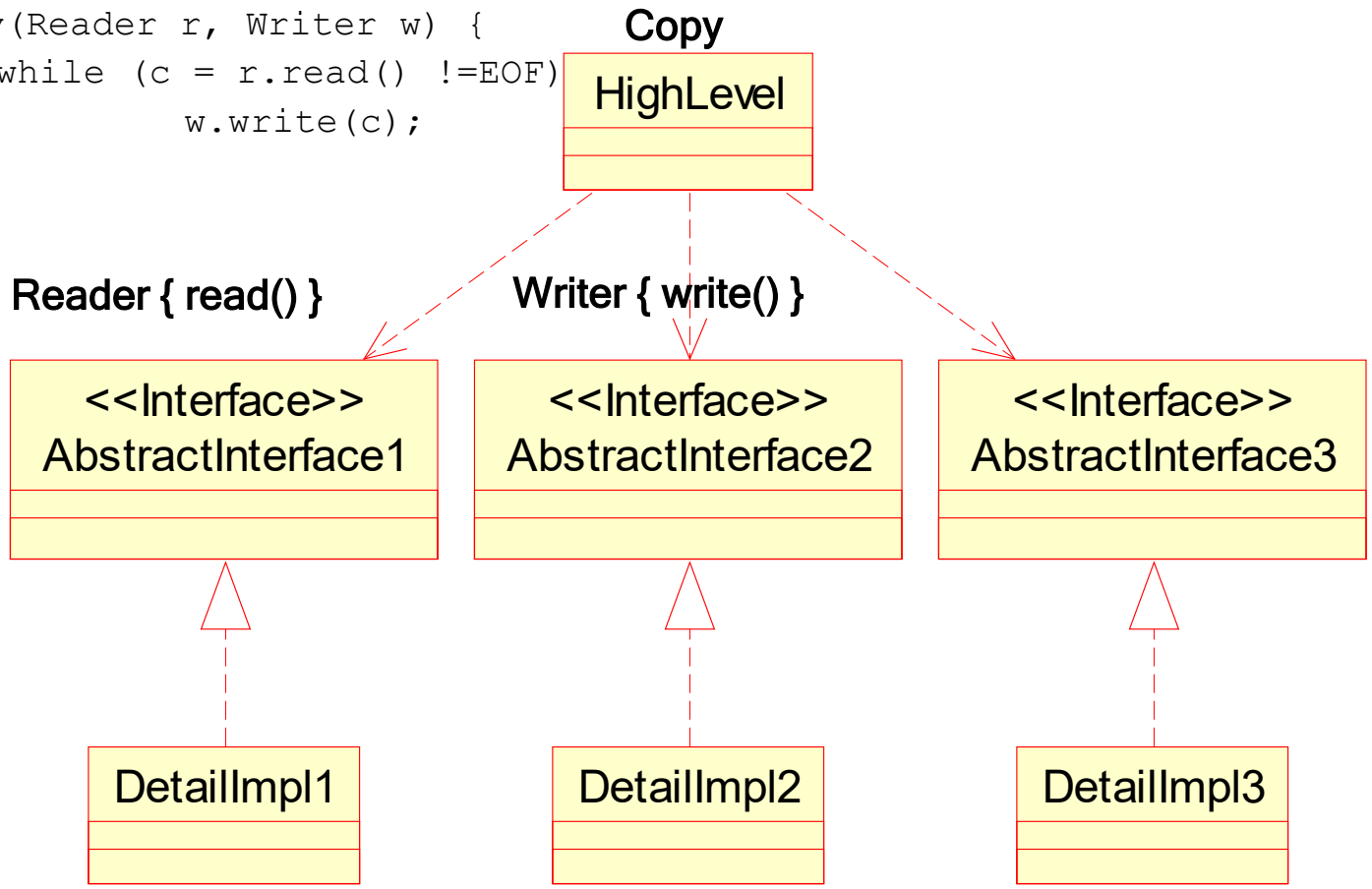


(Dependency Inversion)

DIP Example

(Dependency Structure of an Object Oriented Architecture)

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



(Dependency Inversion) DIP Discussion (1 of 2)

One motivation behind the DIP is to **prevent you from depending upon volatile modules**.

Typically, concrete things change a lot, abstract things change much less frequently.

Not reasonable in every circumstance, though useful goal

- Decision point:
  - *How likely is class to change?*
  - *Do you "own" the class?*
  - *Can you envisage different versions of the class appropriate under different circumstances?*

(Dependency Inversion) DIP Discussion (2 of 2)

Abstractions are “hinge points”, they represent the places where the design can bend or be extended, without themselves being modified (OCP)

One of the most common places that designs depend on concrete classes is when those designs create instances. By definition, you cannot create instances of abstract classes. There is an elegant solution to this problem named Abstract Factory *(look up as an exercise!)*

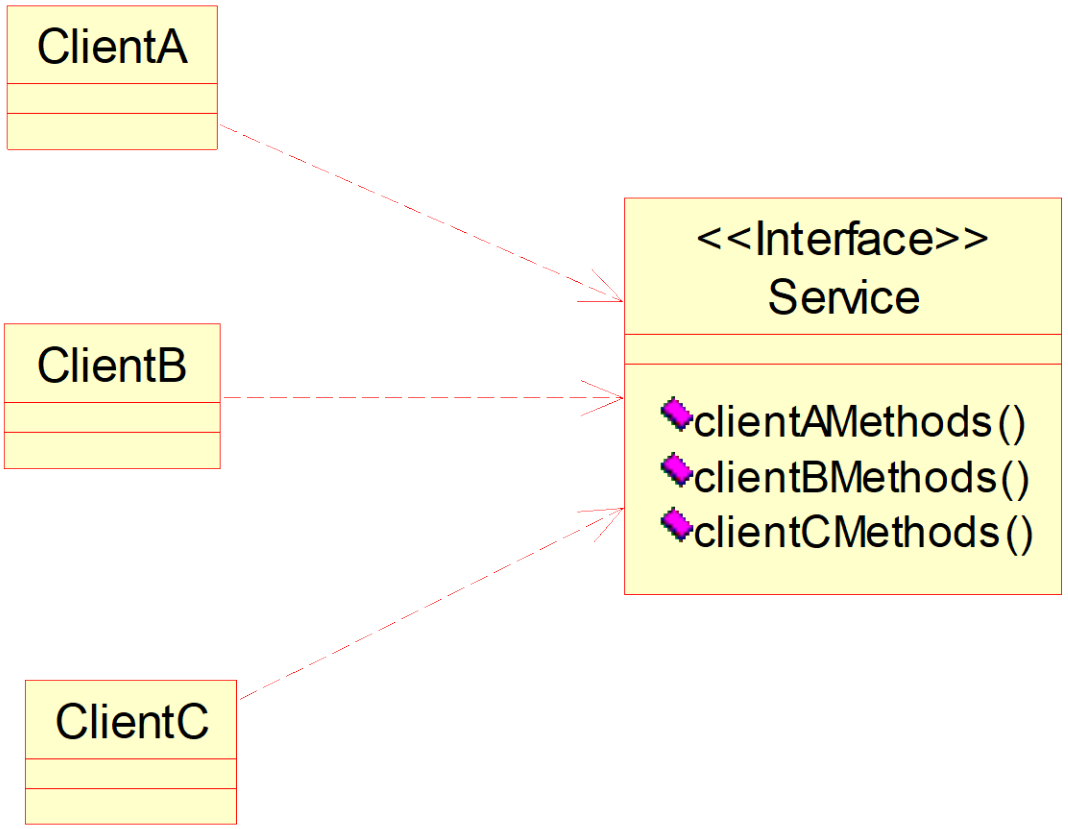
The Interface Segregation Principle (ISP)

"Many client specific interfaces are better than one general purpose interface"

If you have a class that has several clients, rather than loading the class with all the methods that the clients need, create specific interfaces for each type of client and multiply inherit them into the class (C++ parlance) or implement them where required (Java parlance).

ISP Example

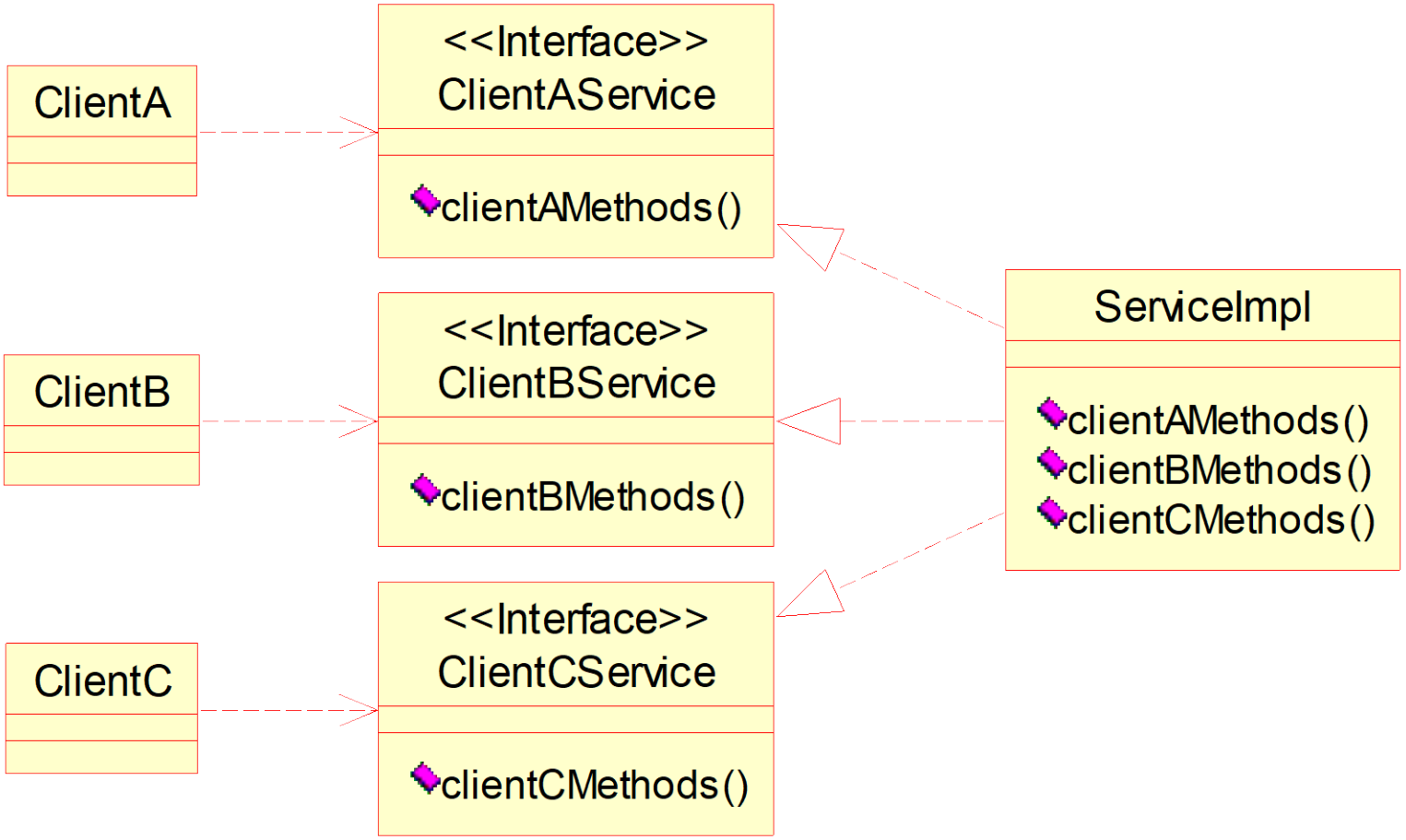
Fat Service with Integrated Interfaces



(Interface Segregation)

ISP Example

Segregated Interfaces



(Interface Segregation)

ISP Example

Segregated Interfaces

```
interface IDrivable
{
    void accelerate();
    void brake();
}
```

```
class Car implements IDrivable
{
   void accelerate()
   { System.out.println("Vroom"); }

   void brake()
   { System.out.println("Queeeeek");}
}
```

```
class Bike implements IDrivable
{
   void accelerate()
   { System.out.println("Rattle, Rattle, ..."); }

   void brake()
   { System.out.println("..."); }
}
```

```
List<IDrivable> vehicleList = new ArrayList<IDrivable>();
list.add(new Car());
list.add(new Car());
list.add(new Bike());
list.add(new Car());
list.add(new Bike());
list.add(new Bike());
```

```
for(IDrivable vehicle: vehicleList)
{
  vehicle.accelerate(); //this could be a bike or a car,
}
```

(Interface Segregation)

ISP Discussion

Without segregation whenever a change is made to one of the methods that `ClientA` calls, `ClientB` and `ClientC` may be affected. It may be necessary to recompile and redeploy them. With segregation, if the interface for `ClientA` needs to change, `ClientB` and `ClientC` will remain unaffected.

The ISP does not recommend that every class that uses a service have its own special interface class that the service must inherit from. Rather, clients should be categorized by their type, and interfaces for each type of client should be created. If two or more different client types need the same method, the method should be added to both of their interfaces.

OO Design Principles

- **The Open/Closed Principle (OCP)**
- **The Liskov Substitution Principle (LSP)**
- **The Dependency Inversion Principle (DIP)**
- **The Interface Segregation Principle (ISP)**

Class-related principles

- **The Reuse/Release Equivalency Principle (REP)**
- **The Common Closure Principle (CCP)**
- **The Common Reuse Principle (CRP)**

Package cohesion principles

- **The Acyclic Dependencies Principle (ADP)**
- **The Stable Dependencies Principle (SDP)**
- **The Stable Abstractions Principle (SAP)**

Package coupling principles

(source in detail: a series of articles by Robert C. Martin on OOD principles, published in *The C++ Report*, 1996)

The Release Reuse Equivalency Principle (REP)

"The granule of reuse is the granule of release."

A reusable element, be it a component, a class, or a cluster of classes, cannot be reused unless it is managed by a release system of some kind.

Clients will/ should refuse to reuse an element unless the author promises to keep track of version numbers, and maintain old versions for awhile.Therefore, one criterion for grouping classes into packages is reuse.

Since packages are the unit of release in Java, they are also the unit of reuse. Therefore architects would do well to group reusable classes together into packages.

The Common Closure Principle (CCP)

"Classes that change together, belong together."

The work to manage, test, and release a package is non-trivial in a large system. The more packages that change in any given release, the greater the work to rebuild, test, and deploy the release. Therefore we would like to minimize the number of packages that are changed in any given release cycle of the product.

To achieve this, we group together classes that we think will change together.

The Common Reuse Principle (CRP)

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

A dependency on a package is a dependency on everything within the package. When a package changes, and its release number is bumped, all clients of that package must verify that they work with the new package - even if nothing they used within the package actually changed.

Hence, classes that aren't reused together should not be grouped together in a package.

(not as easy as it sounds: look back at the different kinds of cohesion: logical, sequential, temporal, functional, data-sharing - arch. notes)

The Package Cohesion Principles (REP/CCP/CRP) Discussion

These three cannot simultaneously be satisfied.

The REP and CRP makes life easy for re-users, whereas the CCP makes life easier for maintainers.

The CCP strives to make packages as large as possible (after all, if all the classes live in just one package, then only one package will ever change). The CRP, however, tries to make packages very small.

Early in a project, architects may set up the package structure such that CCP dominates for ease of development and maintenance. Later, as the architecture stabilizes, the architects may re-factor the package structure to maximize REP and CRP for the external re-users.

REP: Release Reuse Equivalency
CCP: Common Closure
CRP: Common Reuse

OO Design Principles

- **The Open/Closed Principle (OCP)**
- **The Liskov Substitution Principle (LSP)**
- **The Dependency Inversion Principle (DIP)**
- **The Interface Segregation Principle (ISP)**

Class-related principles

- **The Reuse/Release Equivalency Principle (REP)**
- **The Common Closure Principle (CCP)**
- **The Common Reuse Principle (CRP)**

Package cohesion principles

- **The Acyclic Dependencies Principle (ADP)**
- **The Stable Dependencies Principle (SDP)**
- **The Stable Abstractions Principle (SAP)**

Package coupling principles

(source in detail: a series of articles by Robert C. Martin on OOD principles, published in *The C++ Report*, 1996)

The Acyclic Dependencies Principle (ADP)

"The dependencies between packages must not form cycles."

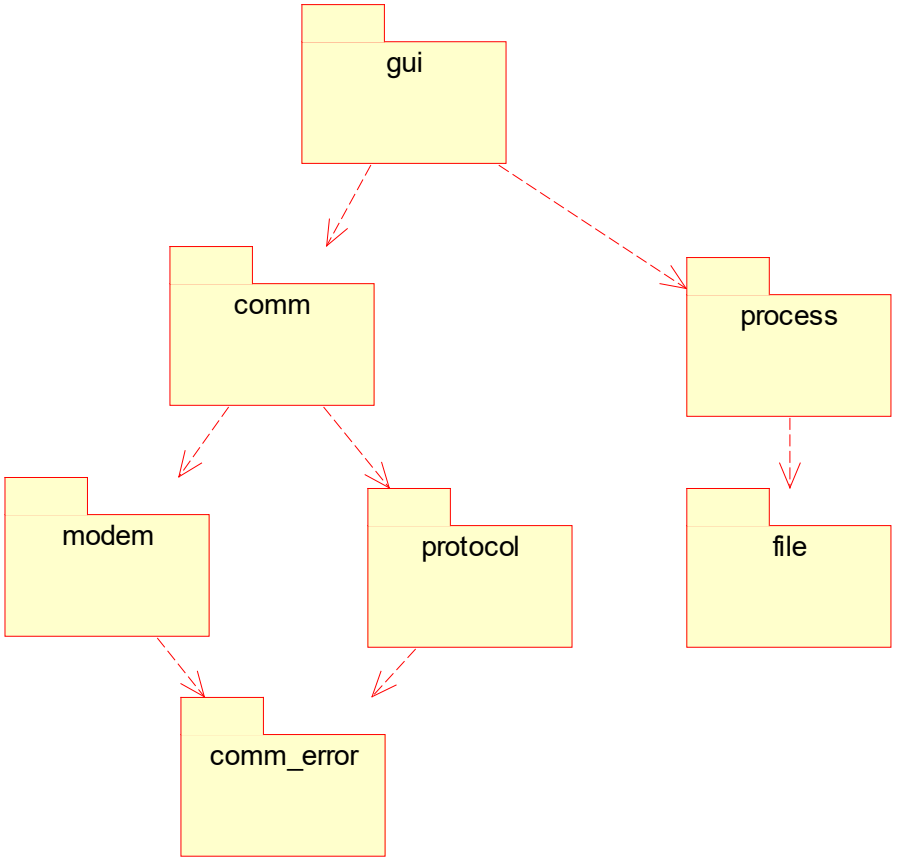
Once changes to a package are made, developers can release the packages to the rest of the project. Before they can do this release, however, they must test that the package works. To do that, they must compile and build it with all the packages it depends upon.

A single cyclic dependency that gets out of control can make the dependency list very long.

Hence, someone needs to be watching the package dependency structure with regularity, and breaking cycles wherever they appear.

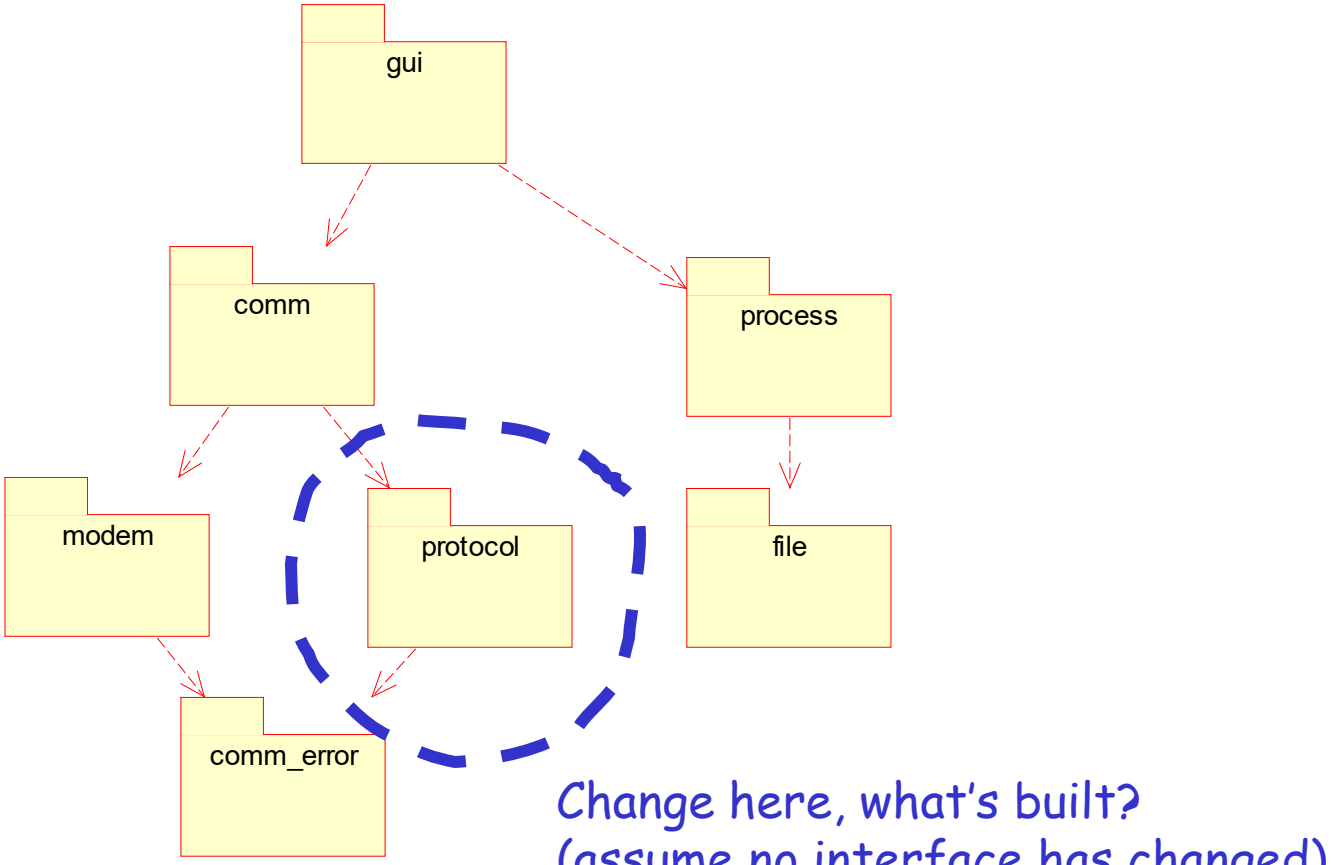
ADP Example

Acyclic Package Network



ADP Example

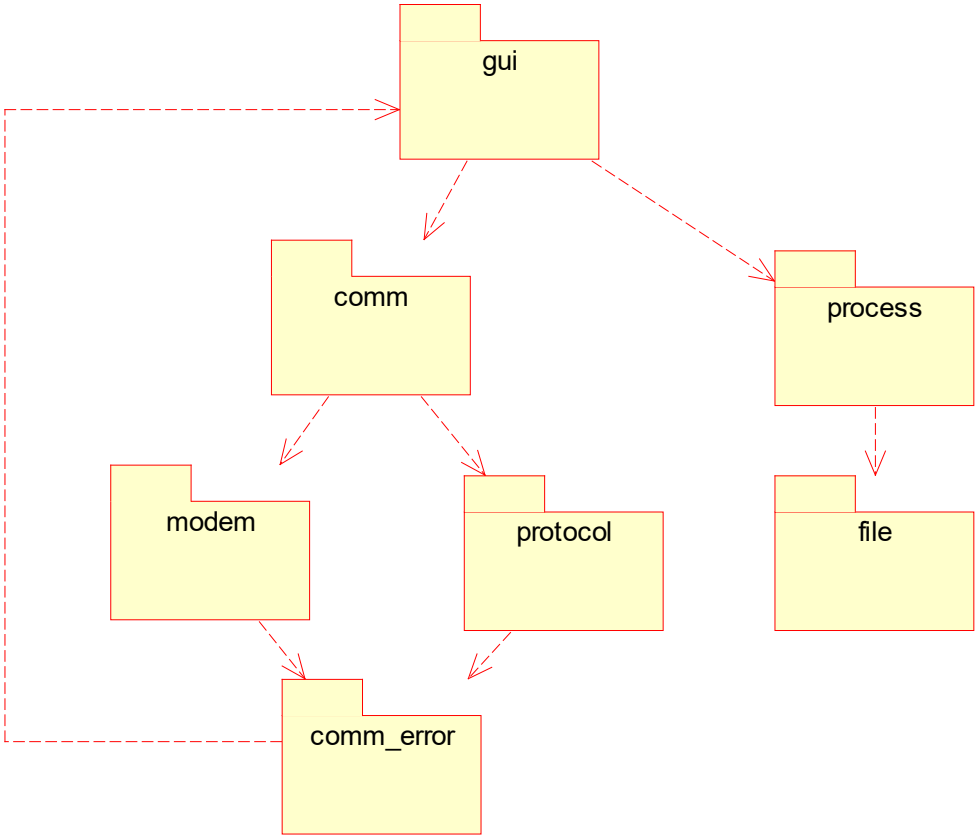
Acyclic Package Network



ADP Example

Cyclic Package Network

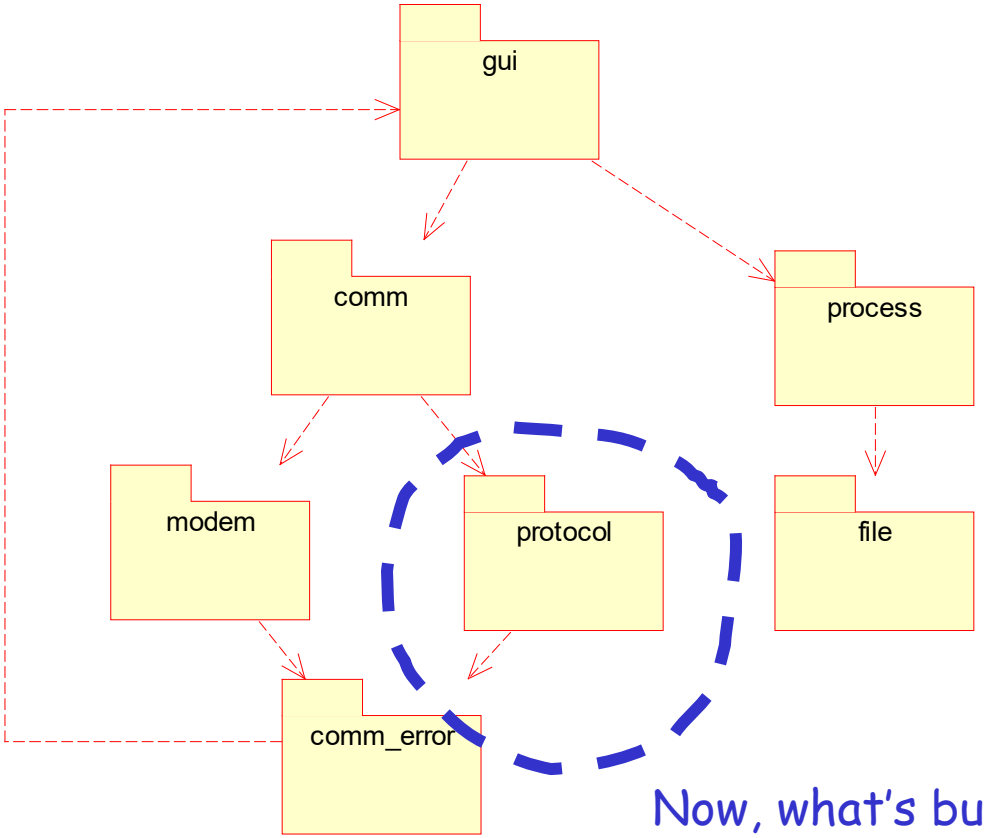
comm_error wants to put message on screen, so….



ADP Example

Cyclic Package Network

comm_error wants to put message on screen, so….



Now, what's built?
(again, no interface has changed)

ADP Discussion

In the acyclic scenario to release the `protocol` package, the engineers would have to build it with the latest release of the `comm_error` package, and run their tests.

In the cyclic scenario to release protocol, the engineers would have to build it with the latest release of `comm_error`, `gui`, `comm`, `process`, `modem`, `file` and run their tests.

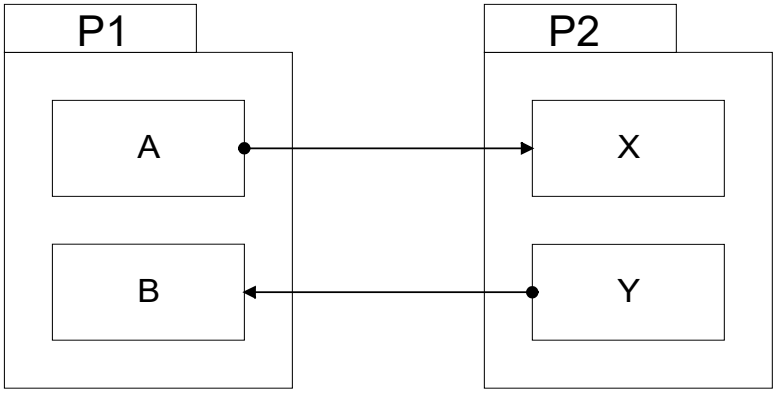
How to break the cycle?

exercise!

(Acyclic Dependencies)

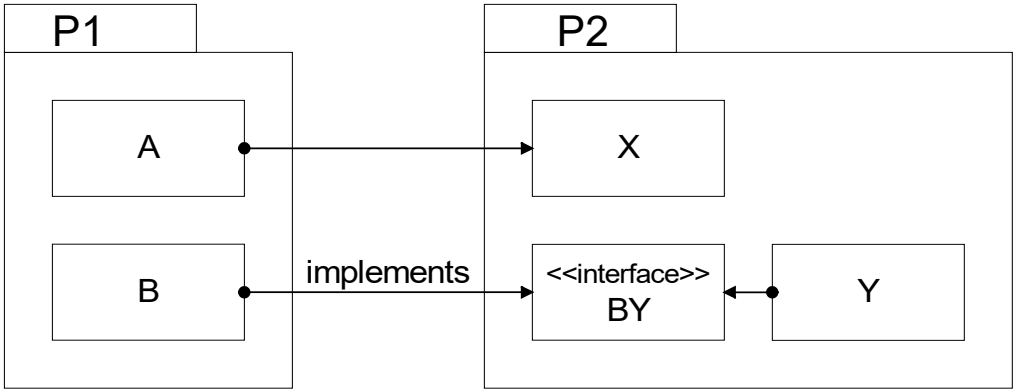
ADP Discussion

Breaking Cycle by introducing an Interface



- A depends on X
- Y depends on B

- Add a new interface or abstract base class to package that uses it
- Inherit or implement in package that implements



(alternatively, create new package containing classes they both depend on)

The Stable Dependencies Principle (SDP)

"Depend in the direction of stability."

Stability is related to the amount of work required to make a change. A package with lots of incoming dependencies is very stable because it requires a great deal of work to reconcile any changes with all the dependent packages (developer has to think responsibly – very good reasons *not to* change!).

A package with lots of outgoing dependencies may *have to* change.

(Stable Dependencies)

SDP Metrics

C_a Afferent coupling: number of classes outside package that depend on classes inside package, i.e., incoming dependencies (high number makes package "responsible")

C_e Efferent coupling: number of classes inside package that depend on classes outside package , i.e., outgoing dependencies (low number makes package "independent")

I Instability:

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

I is in range [0,1]: 0 = no outgoing dependencies, lots of incoming dependencies
stable (responsible and independent)
1 = no incoming dependencies, lots of outgoing dependencies
instable (irresponsible and dependent)

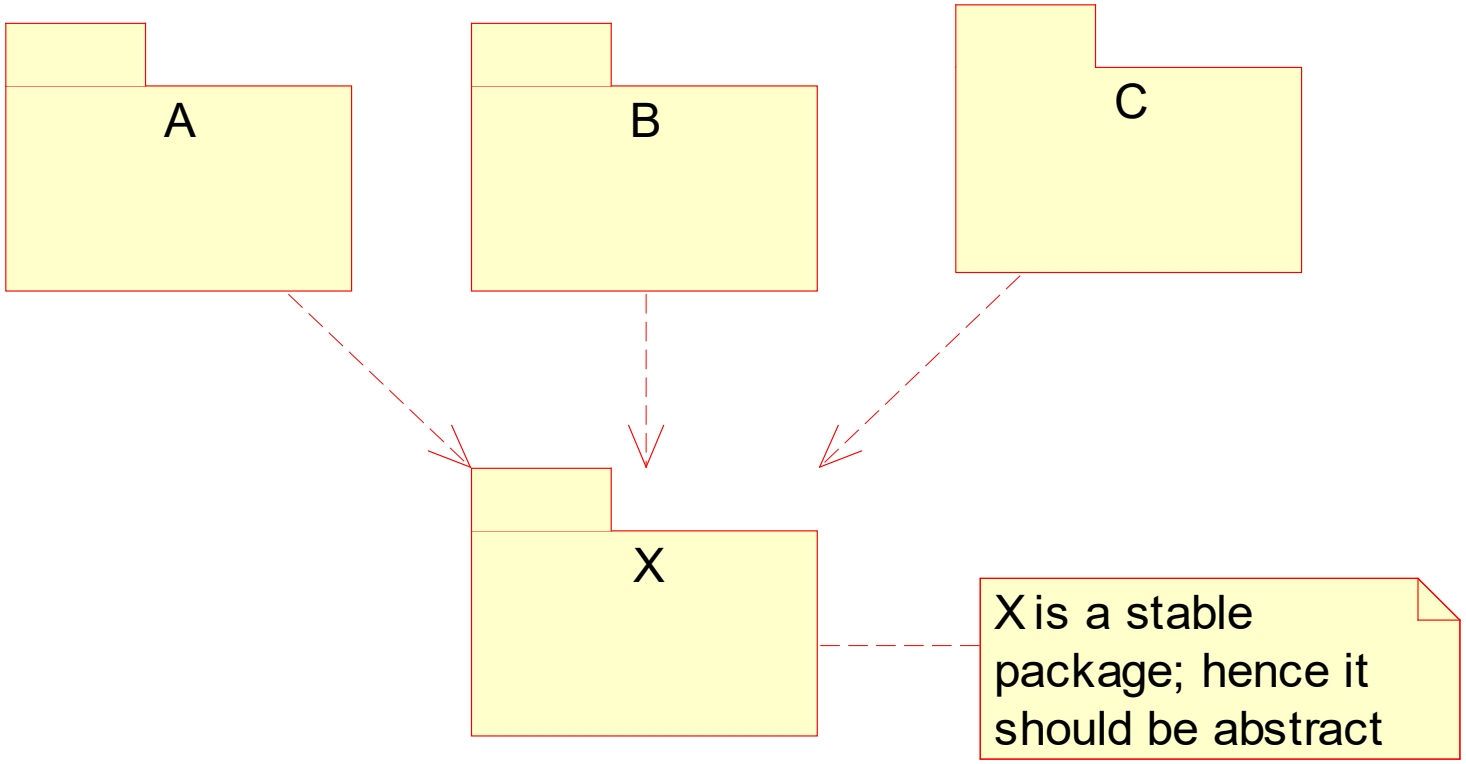
Restating SDP:
"Depend on packages whose instability metric is lower than your own"

The Stable Abstractions Principle (SAP)

“Stable packages should be abstract packages.”

Stability is related to the amount of work required to make a change. A package with lots of incoming dependencies is very stable because it requires a great deal of work to reconcile any changes with all the dependent packages.

SAP Example



(Stable Abstractions)

SAP Metrics

N_c Number of classes in package

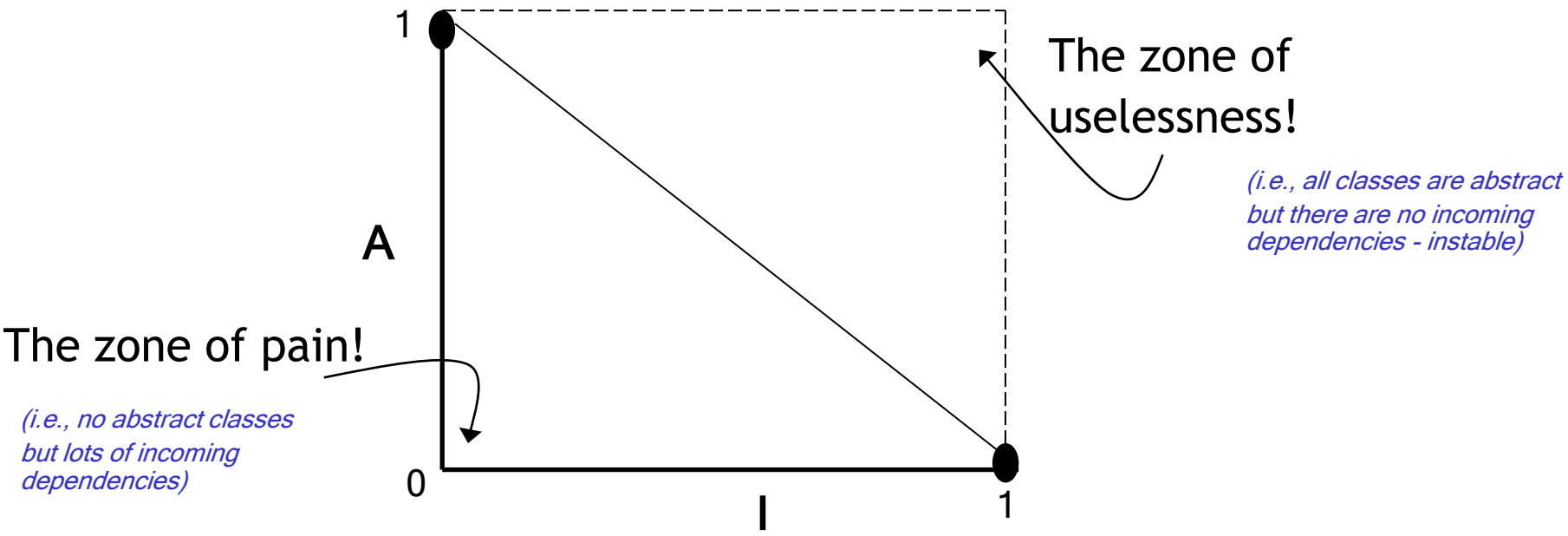
N_a Number of abstract classes in package (an abstract class is a class with at least one abstract interface, and cannot be instantiated)

A Abstractness:

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

A is in range [0,1]: 0 = package has no abstract classes
1 = package contains nothing but abstract classes

Stability vs. Abstractness



I is in range [0,1]: 0 = no outgoing dependencies, lots of incoming dependencies
stable (responsible and independent)
1 = no incoming dependencies, lots of outgoing dependencies
instable (irresponsible and dependent)

A is in range [0,1]: 0 = package has no abstract classes
1 = package contains nothing but abstract classes

SAP Discussion

The packages at the top of a dependency hierarchy are instable and flexible. But those at the bottom are very difficult to change.

The highly stable packages at the bottom of the dependency network may be very difficult to change, but according to the OCP they do not have to be difficult to extend. If the stable packages at the bottom are also highly abstract, then they can be easily extended.

It is possible to compose our application from instable packages that are easy to change, and stable packages that are easy to extend.

The SAP is just a restatement of the DIP.

OO Design Principles

- **The Open/Closed Principle (OCP)**
- **The Liskov Substitution Principle (LSP)**
- **The Dependency Inversion Principle (DIP)**
- **The Interface Segregation Principle (ISP)**

Class-related principles

- **The Reuse/Release Equivalency Principle (REP)**
- **The Common Closure Principle (CCP)**
- **The Common Reuse Principle (CRP)**

Package cohesion principles

- **The Acyclic Dependencies Principle (ADP)**
- **The Stable Dependencies Principle (SDP)**
- **The Stable Abstractions Principle (SAP)**

Package coupling principles

(source in detail: a series of articles by Robert C. Martin on OOD principles, published in *The C++ Report*, 1996)