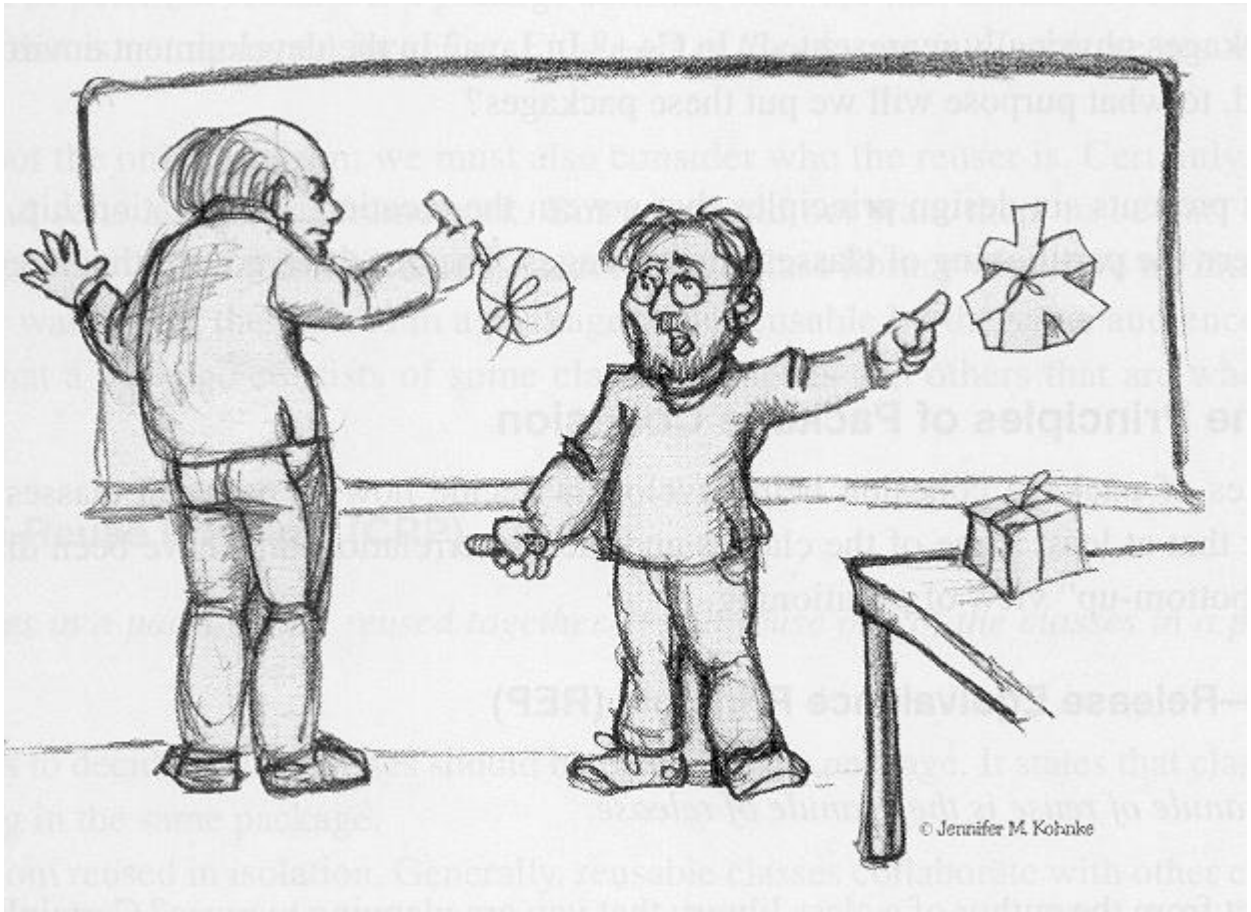
Design Principles 1

For

Software Components

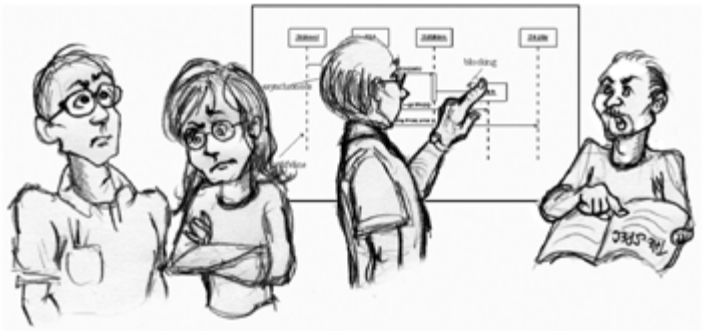


Agenda

- What Is Agile Design?
- Design Principles
  - The Single-Responsibility Principle (SRP)
  - The Open/Closed Principle (OCP)
  - The Dependency-Inversion Principle (DIP)
  - The Interface Segregation Principle (ISP)

What Is Agile Design?

- If **agility** is about building software in tiny increments, how can you ever **design** the software?
- In an agile team, the big picture evolves along with the software. With each iteration, the team improves the design of the system so that it is as good as it can be for the system as it is **now**.
- It is a way to incrementally evolve the most appropriate architecture and design for the system.
- Agile development makes the process of design and architecture **continous**.



What Is a Good Design?

- How do we know how whether the design of a software system is good?
- A design is good if it doesn't have any symptoms of a poor design.
- Symptoms of poor design (design smells) often pervade the overall structure of the software.
- The symptoms are:
  - **Rigidity.** The design is difficult to change.
  - **Fragility.** The design is easy to break.
  - **Immobility.** The design is difficult to reuse.
  - **Viscosity.** It is difficult to do the right thing.
  - **Needless complexity.** Overdesign.
  - **Needless repetition.** Mouse abuse – to much copy-paste.
  - **Opacity.** Disorganized expression.
- These symptoms are similar in nature to code smells, but are at a higher level. They are smells that pervade the overall structure of the software rather than a small section of code.

Why Software Rots

- In nonagile environments, design degrade because requirements change in ways that the initial design did not anticipate.
- Often, these changes need to be made quickly and may be made by developers who are not familiar with the original design philosophy.
- So, though the change to design works, it somehow violates the original design.
- Bit by bit, as changes continue, these violations accumulate until malignancy sets in.

The Cure against Design Smells

- Often, the smell is caused by the violation of one or more **design principles**.
- The object-oriented design principles that help developers eliminate the symptoms of poor design:
  - The Single-Responsibility Principle (SRP)
  - The Open/Closed Principle (OCP)
  - The Liskov Substitution Principle (LSP)
  - The Dependency-Inversion Principle (DIP)
  - The Interface Segregation Principle (ISP)
- These principles are the hard-won product of decades of experience in software engineering.
- They are not the product of a single mind but represent the integration of the thoughts and writings of a large number of software developers and researchers

Only Give Medicine to Patients

- Agile teams apply principles only to solve smells!
- They don't apply principles when there are no smells.
- It would be a mistake to unconditionally conform to a principle just because it is a principle.
- The principles are there to help us eliminate bad smells.
- They are not a perfume to be liberally scattered all over the system.
- Over-conformance to the principles leads to the design smell of needless complexity.

What is the Design?

- A set of UML diagrams may represent parts of a design, but those diagrams are not **the** design.
  - ***But they can be very useful to analyze and discuss or document the design***.
- **The design of a software project is an abstract concept**.
- It has to do with the overall shape and structure of the program, as well as the detailed shape and structure of each component, class, and method.
- The design can be represented by many different media, but its final embodiment is source code.
- In the end, **the source code is the design**.

THE SINGLE-RESPONSIBILITY PRINCIPLE (SRP)

The Single-Responsibility Principle (SRP)

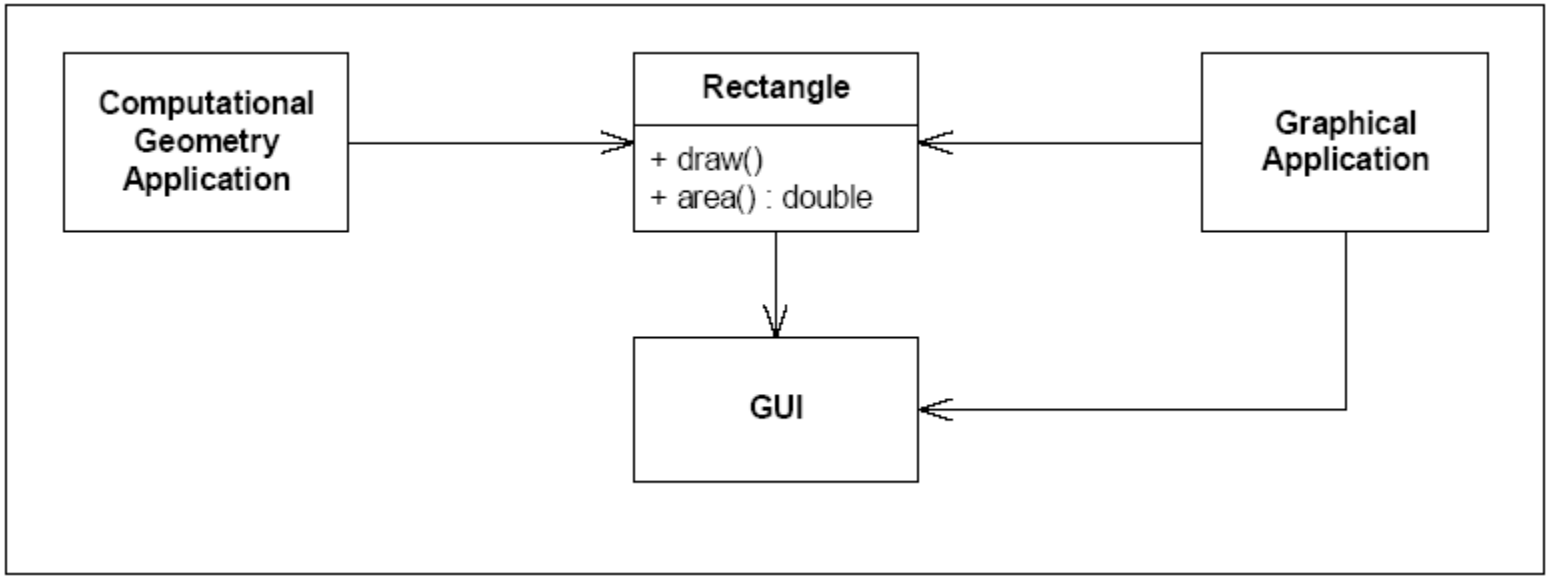
A class should have only one reason to change.

- This principle was described in the work of Tom DeMarco and Meilir Page-Jones. They called it **cohesion**, which they defined as the functional relatedness of the elements of a module.

SRP

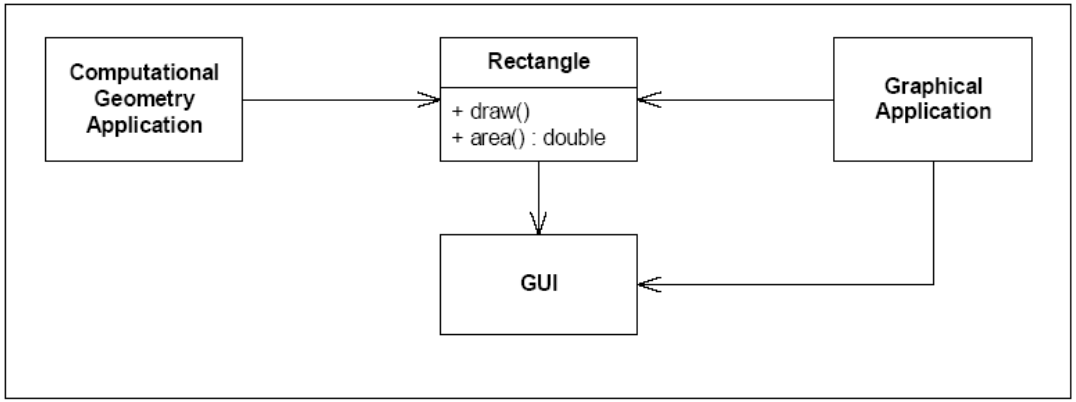
- Why is it important to separate two responsibilities into separate classes?
- The reason is that each responsibility is an axis of change. When the requirements change, that change will be manifest through a change in responsibility among the classes. If a class assumes more than one responsibility, that class will have more than one reason to change.
- If a class has more than one responsibility, the responsibilities become coupled. Changes to one responsibility may impair or inhibit the class's ability to meet the others. This kind of coupling leads to fragile designs that break in unexpected ways when changed.

SRP



This design violates the SRP.
The Rectangle class has two responsibilities:
The first responsibility is to provide a mathematical model of the geometry of a rectangle.
The second responsibility is to render the rectangle on a graphical user interface.

SRP

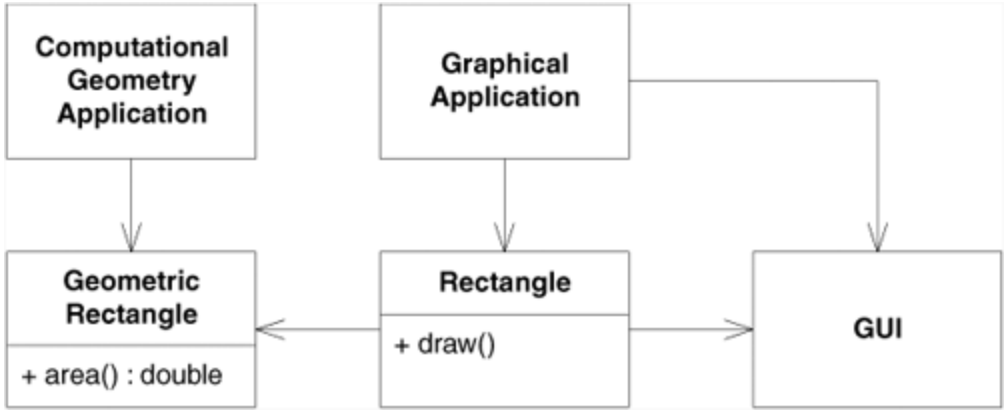


The violation of SRP causes several nasty problems.

Firstly, we must include the GUI in the computational geometry application. If this were a C++ application, the GUI would have to be linked in, consuming link time, compile time, and memory footprint. In a Java application, the .class files for the GUI have to be deployed to the target platform.

Secondly, if a change to the GraphicalApplication causes the Rectangle to change for some reason, that change may force us to rebuild, retest, and redeploy the ComputationalGeometryApplication. If we forget to do this, that application may break in unpredictable ways.

Separated Responsibilities



A better design is to separate the two responsibilities into two completely different classes as shown.

This design moves the computational portions of Rectangle into the GeometricRectangle class.

Now changes made to the way rectangles are rendered cannot affect the ComputationalGeometryApplication.

What is a Responsibility?

- In the context of the Single Responsibility Principle (SRP) we define a responsibility to be
  - "**a reason for change**"

- If you can think of more than one motive for changing a class, then that class has more than one responsibility.

- This is sometimes hard to see. We are accustomed to thinking of responsibility in groups.

The Single Responsibility Principle (SRP)

There should never be more than one reason for a class to change.

The SRP is one of the simplest of the principles, and one of the hardest to get right.

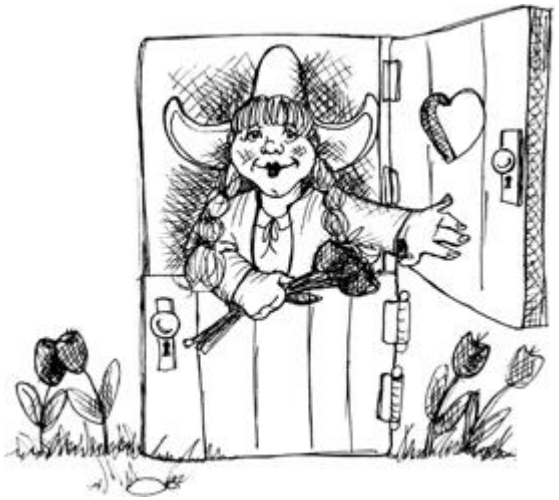
Conjoining responsibilities is something that we do naturally.

Finding and separating those responsibilities is much of what software design is really about.

THE OPEN/CLOSED PRINCIPLE (OCP)

Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.

This principle was described in the work of Bertrand Meyer in 1988.



When to apply OCP?

- When a single change to a program results in a cascade of changes to dependent modules, the design smells of rigidity.
- OCP advises us to refactor the system so that further changes of that kind will not cause more modifications.
- If OCP is applied well, further changes of that kind are achieved by adding new code, not by changing old code that already works.

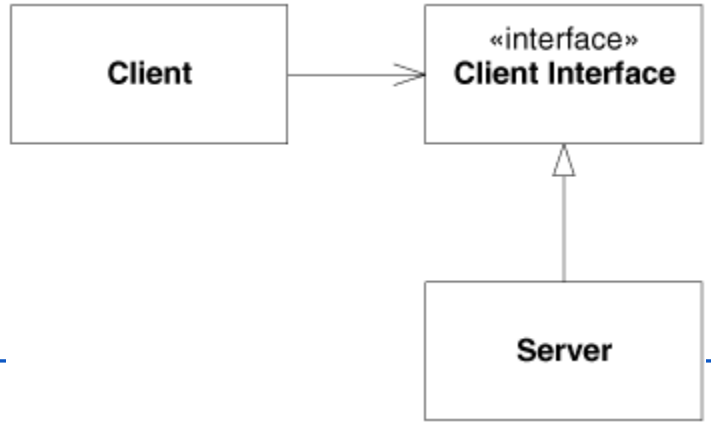
Description of OCP

- Modules that conform to OCP have two primary attributes.
  - They are **open for extension.** This means that the behavior of the module can be extended. As the requirements of the application change, we can extend the module with new behaviors that satisfy those changes. In other words, we are able to change what the module does.
  - They are **closed for modification.** Extending the behavior of a module does not result in changes to the source, or binary, code of the module. The binary executable version of the module—whether in a linkable library, a DLL, or a .EXE file—remains untouched.
- It would seem that these two attributes are at odds. The normal way to extend the behavior of a module is to make changes to the source code of that module. A module that cannot be changed is normally thought to have a fixed behavior.

How to apply OCP?

- A simple design that does not conform to OCP.
  - Both the Client and Server classes are concrete.
  - The Client class **uses** the Server class. If we want for a Client object to use a different server object, the Client class must be changed to name the new server class.

- The corresponding design that conforms to the OCP by using the S**TRATEGY** pattern
  - The Client class uses this abstraction. However, objects of the Client class will be using objects of the derivative Server class.
  - The Client class **uses** the Server class. If we want for a Client object to use a different server object, we must inject the concrete class through a constructor or a setter if the Client has to be closed.



Anticipation and "Natural" Structure

- No matter how "closed" a module is, there will always be some kind of change against which it is not closed.
- **There is no model that is natural to all contexts!**
- Since closure cannot be complete, it must be strategic.
- Experienced designers hope that they know the users and the industry well enough to judge the probability of various kinds of changes.
- These designers then invoke OCP against the most probable changes.

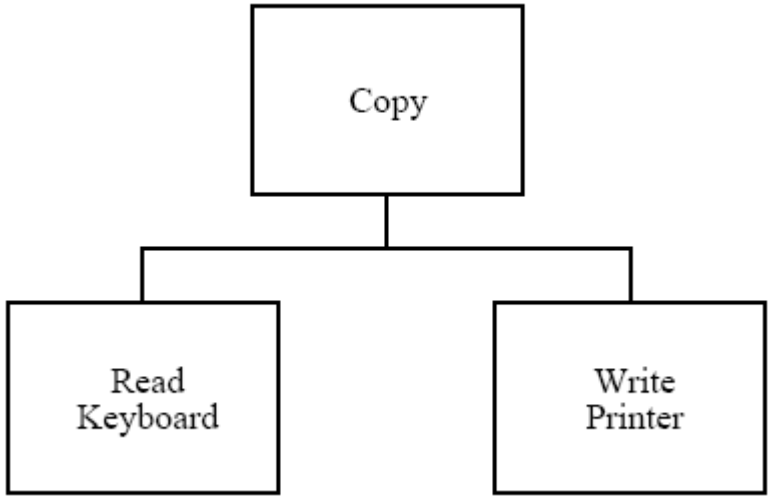
When to apply OCP

- Conforming to OCP is expensive.
- It takes development time and effort to create the appropriate abstractions.
- Those abstractions also increase the complexity of the software design.
- There is a limit to the amount of abstraction that the developers can afford.
- **Clearly, we want to limit the application of OCP to changes that are likely.**

THE DEPENDENCY INVERSION PRINCIPLE (DIP)

DIP Ex. The Copy Program 1

Typical structural program. High-level modules are dependent of low-level modules.



"structure chart" for the copy program

The two low level modules are nicely reusable.
They can be used in many other programs to gain access to the keyboard and the printer.
This is the same kind of reusability that we gain from subroutine libraries.

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

DIP Ex. The Copy Program 2

Consider a new program that copies keyboard characters to a disk file.
Certainly we would like to reuse the "Copy" module since it encapsulates the high level policy that we need.

— ***It knows how to copy characters from a source to a sink***.

Unfortunately, the "Copy" module is dependent upon the "Write Printer" module, and so cannot be reused in the new context.

We could certainly modify the "Copy" module to give it the new desired functionality.

However this adds new interdependencies to the system. As time goes on, and more and more devices must participate in the copy program, the "Copy" module will be littered with if/else statements **and will be dependent upon many lower level modules**.
It will eventually become rigid and fragile.

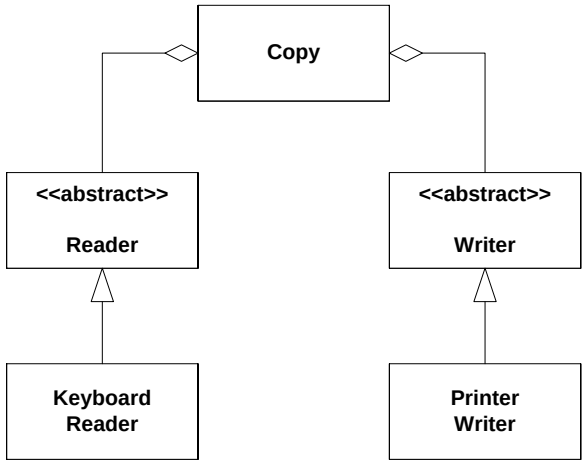
```
enum OutputDevice {printer, disk};
void Copy(outputDevice dev)
{
  int c;
  while ((c=ReadKeyboard()) != EOF)
    if (dev == printer)
      WritePrinter(c);
    else
      WriteDisk(c);
}
```

DIP Ex. The Copy Program 3

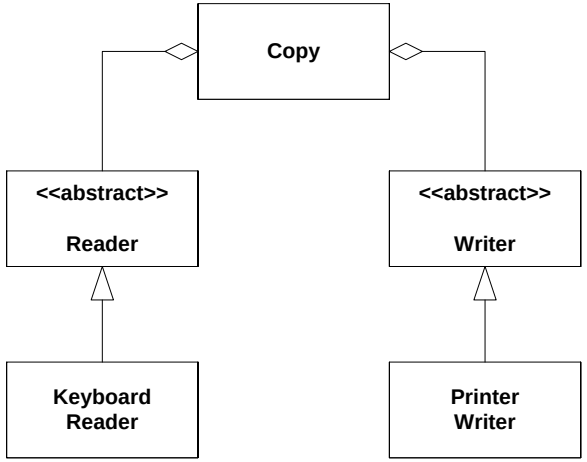
One way to characterize the problem above is to notice that the module containing the high level policy, i.e. the Copy() module, is dependent upon the low level detailed modules that it controls.

OOD gives us a mechanism to *invert* this *dependency*!

This “Copy” class does not depend upon the “Keyboard Reader” nor the “Printer Writer” at all. Thus the dependencies have been *inverted*. ***The “Copy” class depends upon abstractions, and the detailed readers and writers depend upon the same abstractions.***



DIP Ex. The Copy Program 4



We can invent new kinds of "Reader" and "Writer" derivatives that we can supply to the "Copy" class.

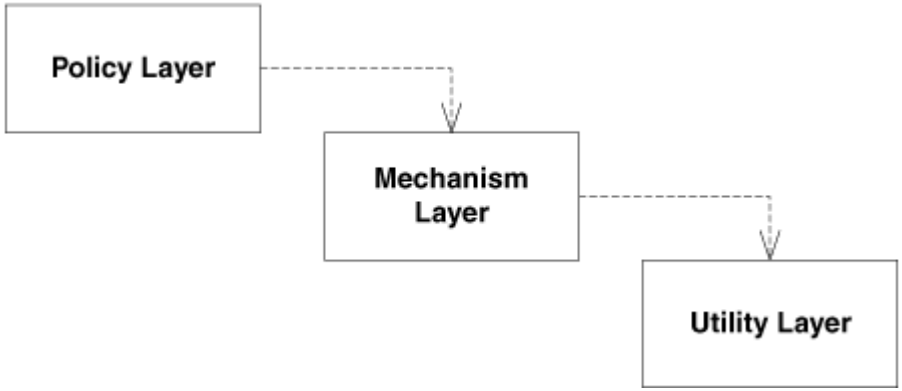
```
class Reader
{
public:
  virtual int Read() = 0;
};

class Writer
{
public:
  virtual void Write(char) = 0;
};

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

But, no matter how many kinds of "Readers" and "Writers" are created, "Copy" will depend upon none of them. There will be no interdependencies to make the program fragile or rigid.

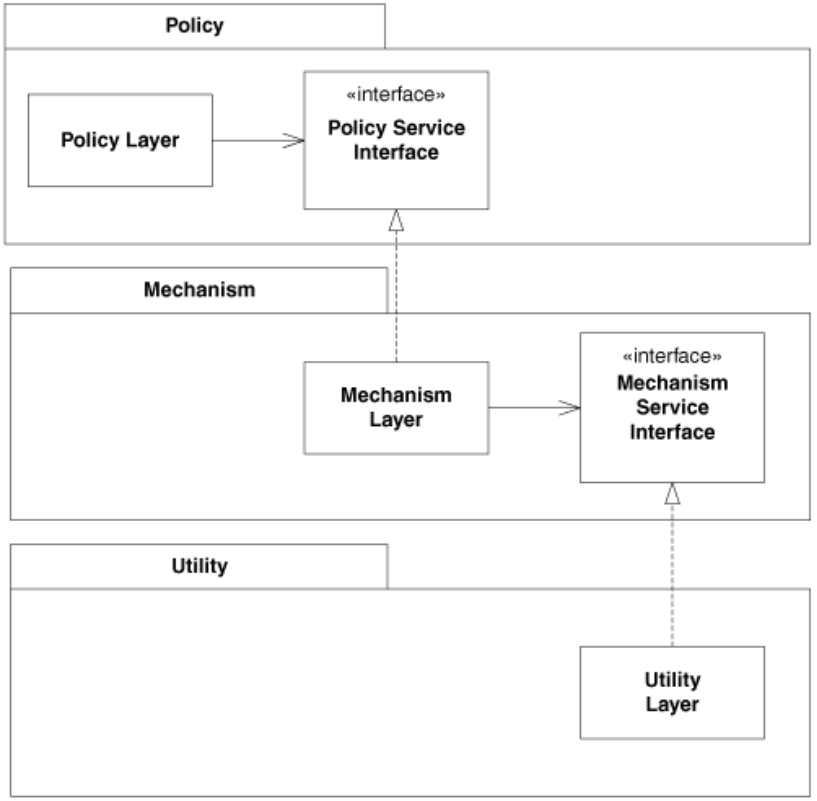
Layering

- According to Booch, "all well structured object-oriented architectures have clearly-defined layers, with each layer providing some coherent set of services through a well-defined and controlled interface.

Naive layering scheme



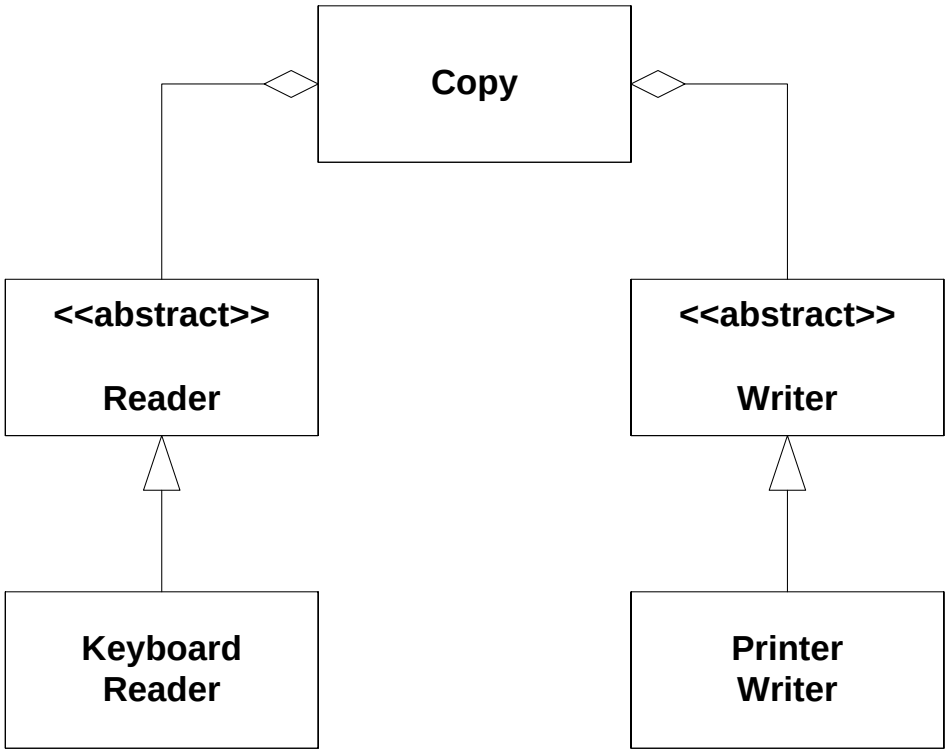
Inverted layers



The **D**ependency **I**nversion **P**rinciple

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.***



Depend on Abstractions

- A somewhat more naive interpretation of DIP is the simple heuristic:
  - **"Depend on abstractions."**
- Simply stated, this heuristic recommends that you should not depend on a concrete class
  - all relationships in a program should terminate on an abstract class or an interface.
- This heuristic is usually violated at least once in every program.
  - Somebody has to create the instances of the concrete classes, and whatever module does that will depend on them.
  - Moreover, there seems no reason to follow this heuristic for classes that are concrete but nonvolatile.
  - If a concrete class is not going to change very much, and no other similar derivatives are going to be created, it does very little harm to depend on it.

THE INTERFACE SEGREGATION PRINCIPLE (ISP)

ISP Ex. Security Door 1

We have a Door class, a Timer class and a TimerClient class.

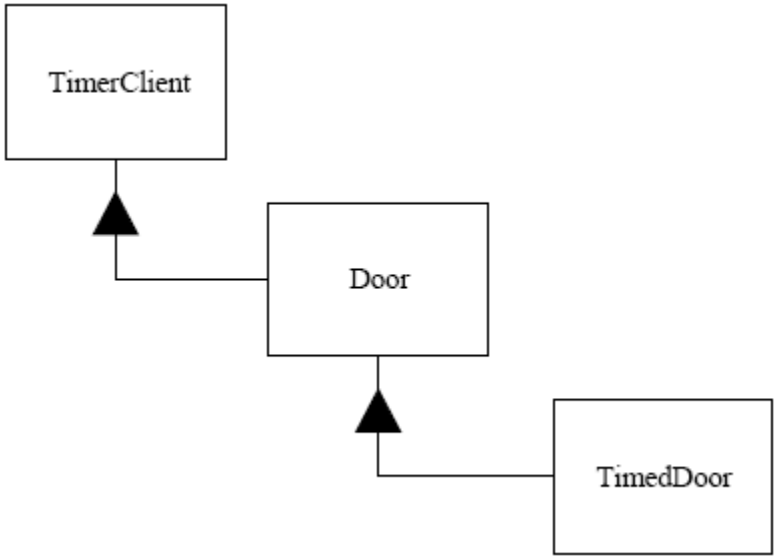
How do we make a TimedDoor class from that?

```
class Door
{
public:
virtual void Lock() = 0;
virtual void Unlock() = 0;
virtual bool IsDoorOpen() = 0;
};
```

```
class Timer
{
public:
void Regsiter(int timeout,
TimerClient* client);
};
class TimerClient
{
public:
virtual void TimeOut() = 0;
};
```

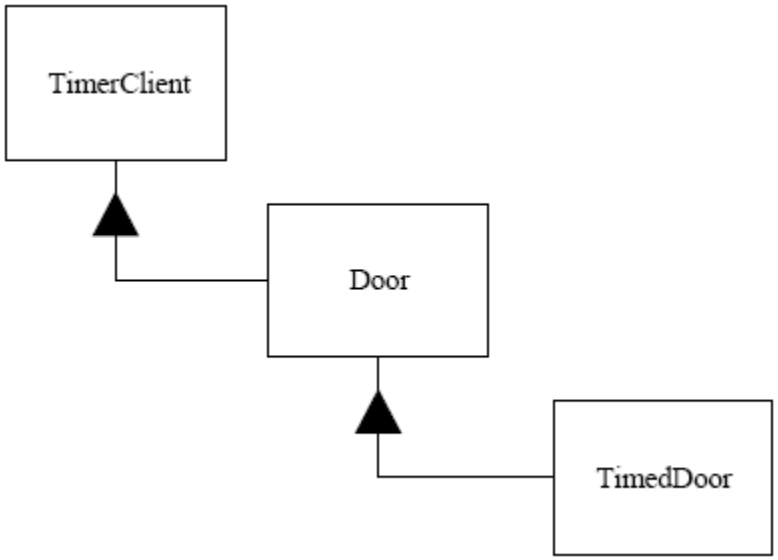
ISP Ex. Security Door 2

One solution is to force Door, and therefore TimedDoor, to inherit from TimerClient. This ensures that TimerClient can register itself with the Timer and receive the TimeOut message.



Although this solution is common, it is not without problems. Chief among these is that the Door class now depends upon TimerClient. Not all varieties of Door need timing. Indeed, the original Door abstraction had nothing whatever to do with timing. If timing free derivatives of Door are created, those derivatives will have to provide nil implementations for the TimeOut method. Moreover, the applications that use those derivatives will have to #include the definition of the TimerClient class, even though it is not used.

ISP Ex. Security Door 3



This is interface pollution.

The interface of Door has been polluted with an interface that it does not require. It has been forced to incorporate this interface solely for the benefit of one of its subclasses.

If this practice is pursued, then every time a derivative needs a new interface, that interface will be added to the base class.

This will further pollute the interface of the base class, making it “fat”.

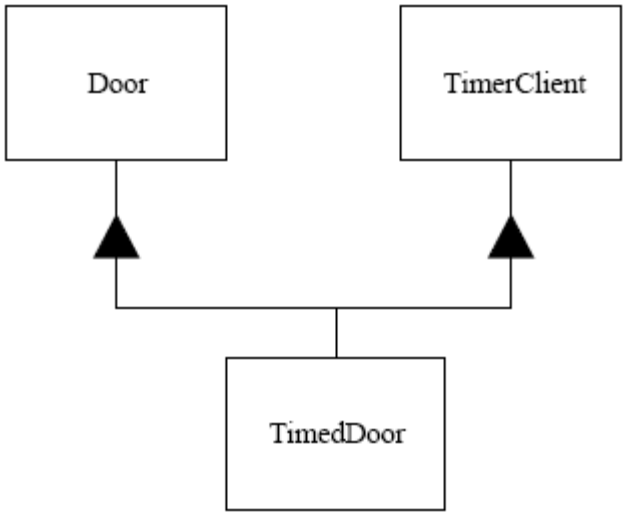
The Interface Segregation Principle (ISP)

Clients should not be forced to depend upon interfaces that they do not use.

When clients are forced to depend upon interfaces that they don't use, then those clients are subject to changes to those interfaces.
This results in an inadvertent coupling between all the clients.

We would like to avoid such couplings where possible, and so we want to separate the interfaces where possible.

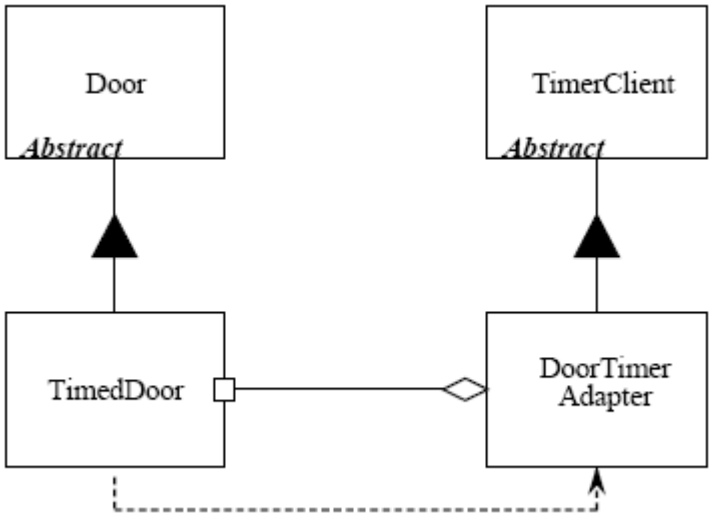
ISP Separation through Multiple Inheritance




```
class TimedDoor : public Door, public TimerClient
{
  public:
    virtual void TimeOut(int timeOutId);
};
```

ISP Separation through Delegation

We can employ the Adapter pattern to the TimedDoor problem.
The solution is to create an adapter object that derives from TimerClient and delegates to the TimedDoor.




```
class TimedDoor : public Door
{
public:
  virtual void DoorTimeOut(int
  timeOutId);
};

class DoorTimerAdapter : public
TimerClient
{
public:
  DoorTimerAdapter(TimedDoor& theDoor)
  : itsTimedDoor(theDoor)
  {}
  virtual void TimeOut(int timeOutId)
  {itsTimedDoor.DoorTimeOut(timeOutId);}
private:
  TimedDoor& itsTimedDoor;
};
```

How to design components?

- In many ways, the criteria that define a well-designed component are no different to the criteria that have been used to underpin good modular software design for many years!

Component Functionality

- A component should implement the *right* functionality.
- This means the component designer must have a good understanding of the context(s) in which their component is to be used.
- For customizable components it is important that the designer understand the variations in functionality that may be required.

Component Interface

- Interfaces must be well designed and easy to use.
- The Interface Segregation Principle (ISP) helps here, stating that different interfaces should be provided for different types of client developers
- It is also important that component interfaces present a *clean abstraction* to the component user.
- An interface that presents a clean abstraction will use concepts (classes, types) that are familiar to its *client* developers, and in particular will hide irrelevant implementation detail.