SOLID Design Principles

- **S**RP: The Single Responsibility Principle
- **O**CP: The Open/Closed Principle
- **L**SP: The Liskov Substitution Principle
- **I**SP: The Interface Segregation Principle
- **D**IP: The Dependency Inversion Principle

Source: *Agile Software Development: Principles, Patterns, and Practices.*
Robert C. Martin, Prentice Hall, 2002.

S. O. L. I. D.

- Принцип на едноличната отговорност - Single Responsibility Principle (SRP);
- Отворено-затворен принцип - Open Closed Principle (OCP);
- Принцип на субституцията (Лисков) - Liskov's Substitution Principle (LSP);
- Принцип за разделяне на интерфейсите - Interface Segregation Principle (ISP);
- Принцип за обръщане на зависимостта- Dependency Inversion Principle (DIP)).

Принцип на едноличната отговорност

- Един модул трябва да има само една причина да се променя.
- Този принцип гласи, че ако имаме две неща да се променят в един клас, трябва да се разделят функционалност в два класа.
- Всеки клас ще се справи само с една отговорност и за бъдеще, ако ние трябва да направим една промяна, ние ще го направи в класа, в който се отнася.

Just because you can, doesn't mean you should.

SRP: Example

- Consider a module that compiles and prints a report. Such a module can be changed for two reasons:
  - First, the content of the report can change.
  - Second, the format of the report can change.
- These two things change for very different causes; one substantive, and one cosmetic. The single responsibility principle says that these two aspects of the problem are really two separate responsibilities, and should therefore be in separate classes or modules. It would be a bad design to couple two things that change for different reasons at different times.
- The reason it is important to keep a class focused on a single concern is that it makes the class more robust. Continuing with the foregoing example, if there is a change to the report compilation process, there is greater danger that the printing code will break if it is part of the same class.

single responsibility principle - bad example

```
public class EmployeeService
{
public void SaveEmployeeInfo(Employee e)
{// To do something}

public void UpdateEmployeeInfo(int empId, Employee e)
{//To do Something

}
public Employee GetEmployeeInfo(int empID)
{// To do something
}

public void MAPEmployee(SqlDatareader reader)
{// To do something
}
}
```

single responsibility principle - good example

```
public class EmployeeService
{
        public void SaveEmployeeInfo(Employee e)
        {// To do something}
        public void UpdateEmployeeInfo(int empId, Employee e)
        {//To do Something }
        public Employee GetEmployeeInfo(int empID)
        {// To do something }
}

public class ExtendEmployeeService extend EmployeeService
{
        public void MAPEmployee(SqlDatareader reader)
        {// To do something}
}
```

Отворено-затворен принцип

- Един модул трябва да бъде отворен за разширение, но затворен за модификация.
- Ако има нови изисквания към софтуера, тогава няма да модифицираме вече работещия код, а ще имплементираме нов.

Open chest surgery is not needed when putting on a coat.

Open/Closed Principle

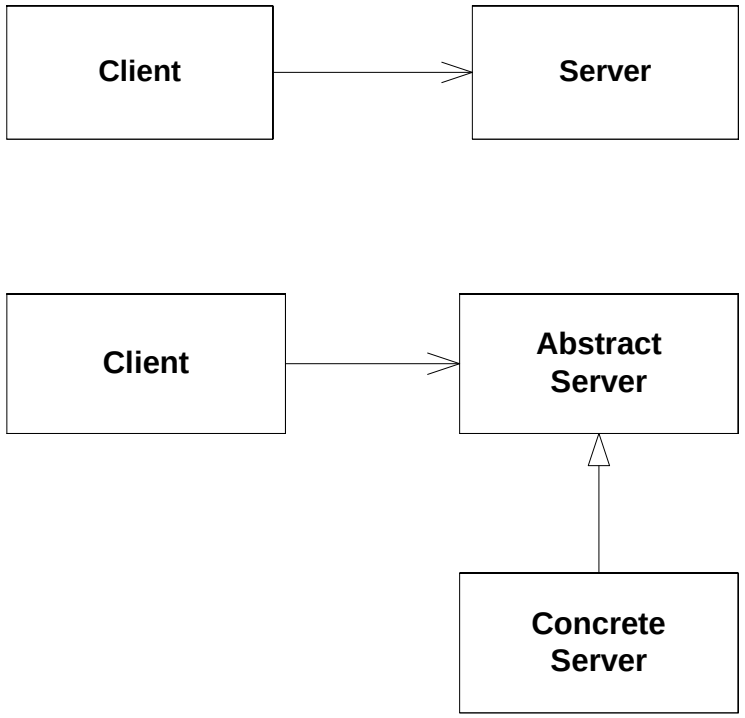
"Modules should be open for extension, but closed for modification"
-Bertrand Meyer

- A principle which states that we should add new functionality by adding new code, not by editing old code.
- Defines a lot of the value of OO programming
- Abstraction is the key

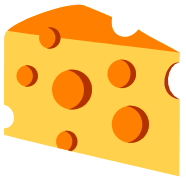
Abstraction is Key

Abstraction is the most important word in OOD

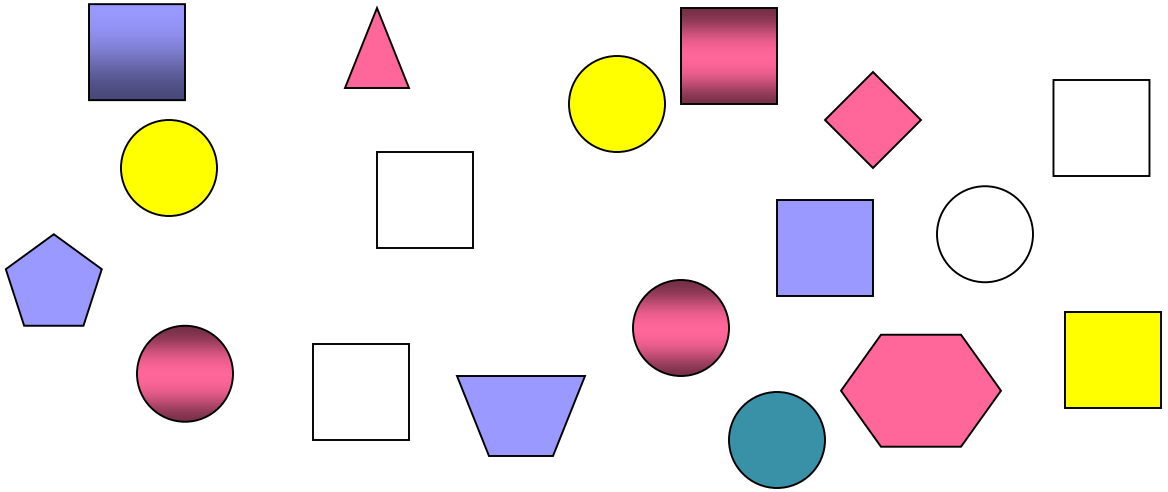
- Client/Server relationships are "open"
- Changes to servers cause changes to clients
- Abstract servers "close" clients to changes in implementation.



The Shape Example



- Procedural (not closed) implementation
- OO (closed) implementation



Procedural (open) version

Shape.h

```
enum ShapeType {circle, square};
struct Shape
     {enum ShapeType itsType;};
```

Circle.h

```
struct Circle
{
   enum ShapeType itsType;
   double itsRadius;
   Point itsCenter;
};
void DrawCircle(struct Circle*)
```

Square.h

```
struct Square
{
   enum ShapeType itsType;
   double itsSide;
   Point itsTopLeft;
};
void DrawSquare(struct Square*)
```

DrawAllShapes.c

```
#include <Shape.h>
#include <Circle.h>
#include <Square.h>

typedef struct Shape* ShapePtr;

void
DrawAllShapes(ShapePtr list[], int n)
{
   int i;
   for( i=0; i< n, i++ )
   {
        ShapePtr s = list[i];
        switch ( s->itsType )
        {
        case square:
             DrawSquare((struct Square*)s);
             break;
        case circle:
             DrawCircle((struct Circle*)s);
             break;
        }
   }
}
```

What is wrong with the code?

It can be demonstrated to work. Isn't that the important thing?

- DrawAllShapes is not closed.
  - *Switch/case* tend to recur in diverse places.
  - If we add a shape, we add to the switch/case.
  - All *switch/case* statements must be found and edited.
  - *Switch/Case* statements are seldom this tidy.
  - When we add to the *enum*, we must rebuild everything.
- The software is both rigid and brittle

A Closed Implementation

Shape.h

```
Class Shape
{
public:
    virtual void Draw() const =0;
};
```

Square.h

```
Class Square: public Shape
{
public:
    virtual void Draw() const;
};
```

Circle.h

```
Class Circle: public Shape
{
public:
    virtual void Draw() const;
};
```

DrawAllShapes.cpp

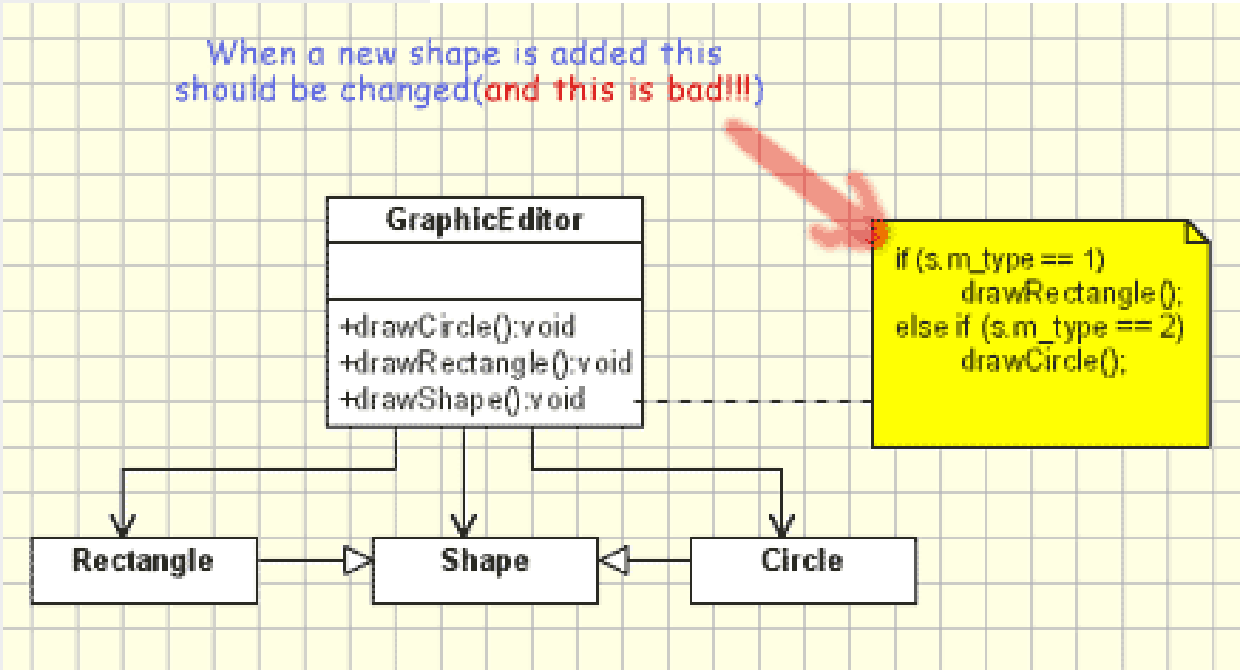
```
#include <Shape.h>

void
DrawAllShapes(Shape* list[],int n)
{
   for(int i=0; i< n; i++)
       list[i]->draw();
}
```

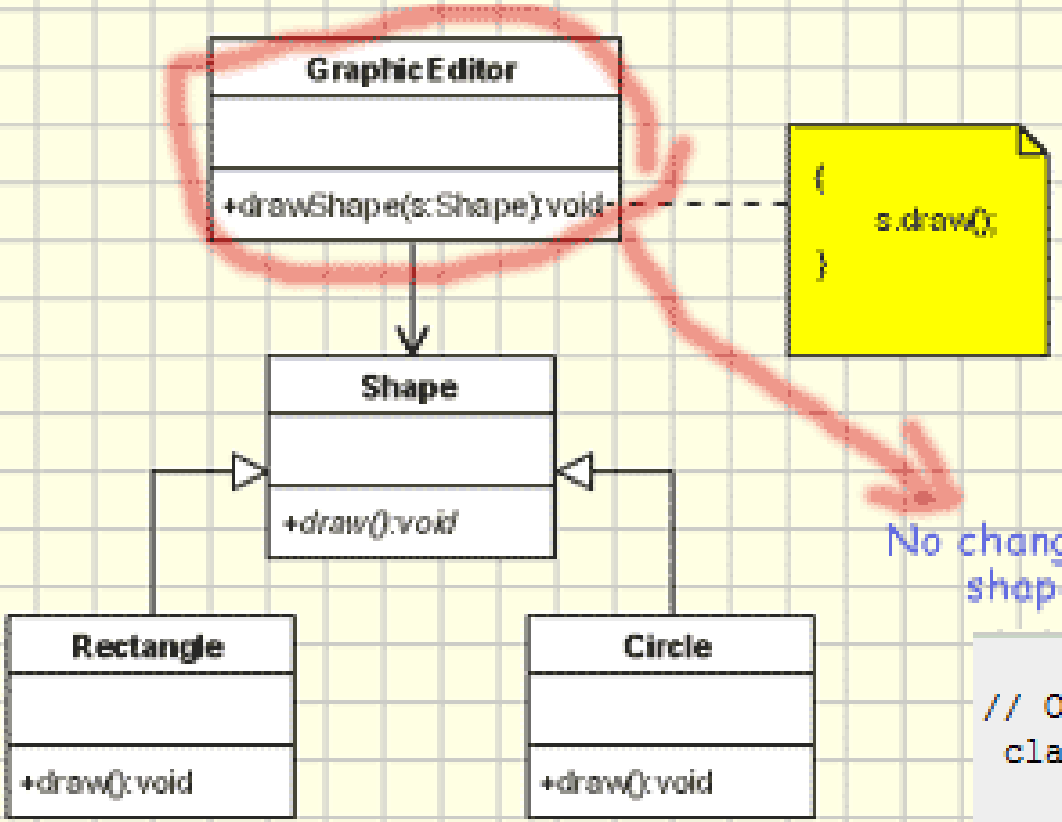
Strategic Closure

No program is 100% closed.

- Closure Against What?
  - Closure is strategic. You have to choose which changes you'll isolate yourself against.
  - What if we have to draw all circles first? Now DrawAllShapes must be edited (or we have to hack something)
- Opened Where?
  - Somewhere, someone has to instantiate the individual shapes.
  - It's best if we can keep the dependencies confined

```
// Open-Close Principle - Bad example
 class GraphicEditor {

        public void drawShape(Shape s) {
                if (s.m_type==1)
                        drawRectangle(s);
                else if (s.m_type==2)
                        drawCircle(s);
        }
        public void drawCircle(Circle r) {....}
        public void drawRectangle(Rectangle r) {....}
 }

 class Shape {
        int m_type;
 }

 class Rectangle extends Shape {
        Rectangle() {
                super.m_type=1;
        }
 }

 class Circle extends Shape {
        Circle() {
                super.m_type=2;
        }
 }
```






```
// Open-Close Principle - Good example
 class GraphicEditor {
        public void drawShape(Shape s) {
                s.draw();
        }
 }

 class Shape {
        abstract void draw();
 }

 class Rectangle extends Shape  {
        public void draw() {
                // draw the rectangle
        }
 }
```

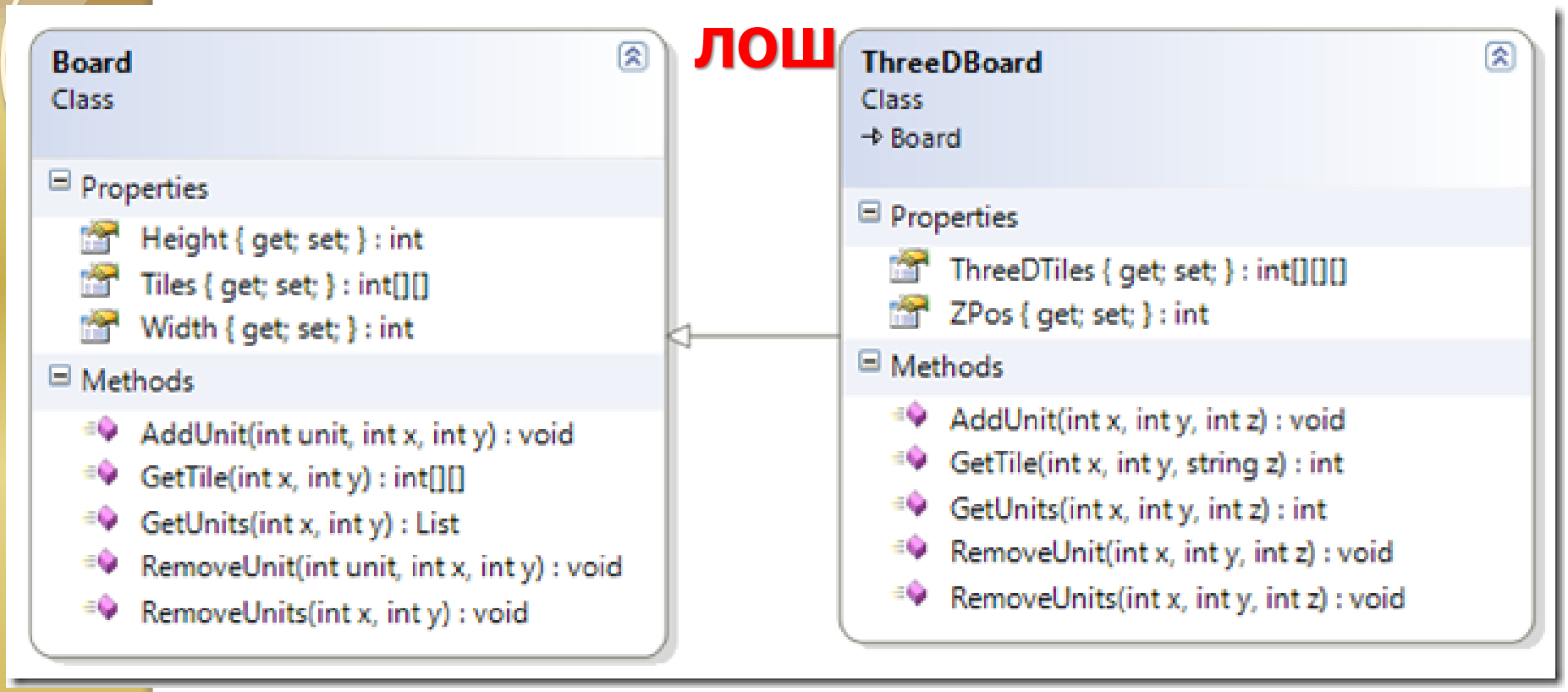
Принцип на субституцията (Лисков)

- Наследниците трябва да бъдат заместим от техните базови класове.
- Правилна йерархия на класовете.
- Методи или функции, които използват тип от базов клас, трябва да могат да работят и с обекти от наследниците без да се налага промяна.

If it looks like a duck, quacks like a duck, but needs batteries – you probably have the wrong abstraction

LSP:The Liskov Substitution Principle

- Barbara Liskov in a 1987: Let q(x) be a property provable about objects x of type T. Then q(y) should be true for objects y of type S where S is a subtype of T.
- Liskov's notion of a behavioral subtype defines a notion of substitutability for mutable objects; that is, if S is a subtype of T, then objects of type T in a program may be replaced with objects of type S without altering any of the desirable properties of that program (e.g., correctness).
- Behavioral subtyping is a stronger notion than typical subtyping of functions defined in type theory, which relies only on the contravariance of argument types and covariance of the return type:
  - Contravariance (converting from narrower to wider) of method arguments in the subtype.
  - Covariance (converting from wider to narrower) of return types in the subtype.
  - No new exceptions should be thrown by methods of the subtype, except where those exceptions are themselves subtypes of exceptions thrown by the methods of the supertype.



Instead of extending Board, ThreeDBoard should be composed of Board objects. One Board object per unit of the Z axis.

Interface Segregation Principle (ISP)

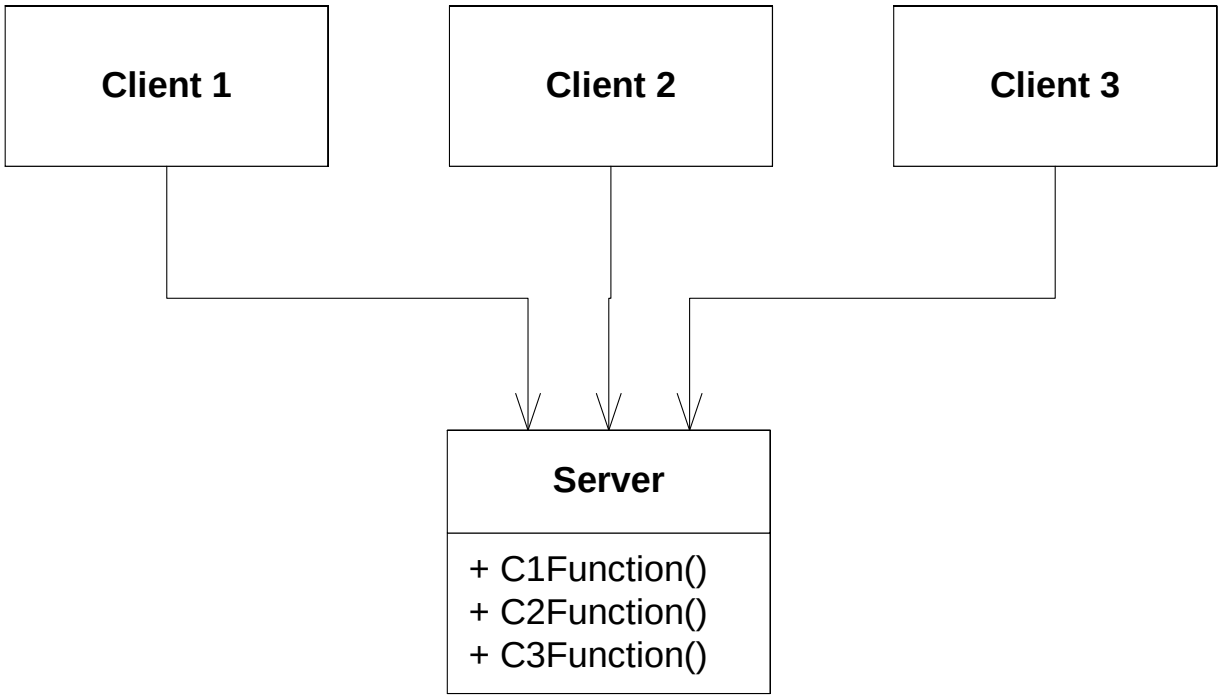
Classes should not depend on interfaces that they not use.

- The meaning of this phrase is to avoid tying a client class to a big interface if only a subset of this interface is really needed.
- Many times you see an interface which has lots of methods.
- This is a bad design choice since probably a class implementing it will infringe Single Responsibility Principle and for many other issues which arises when interfaces grow.

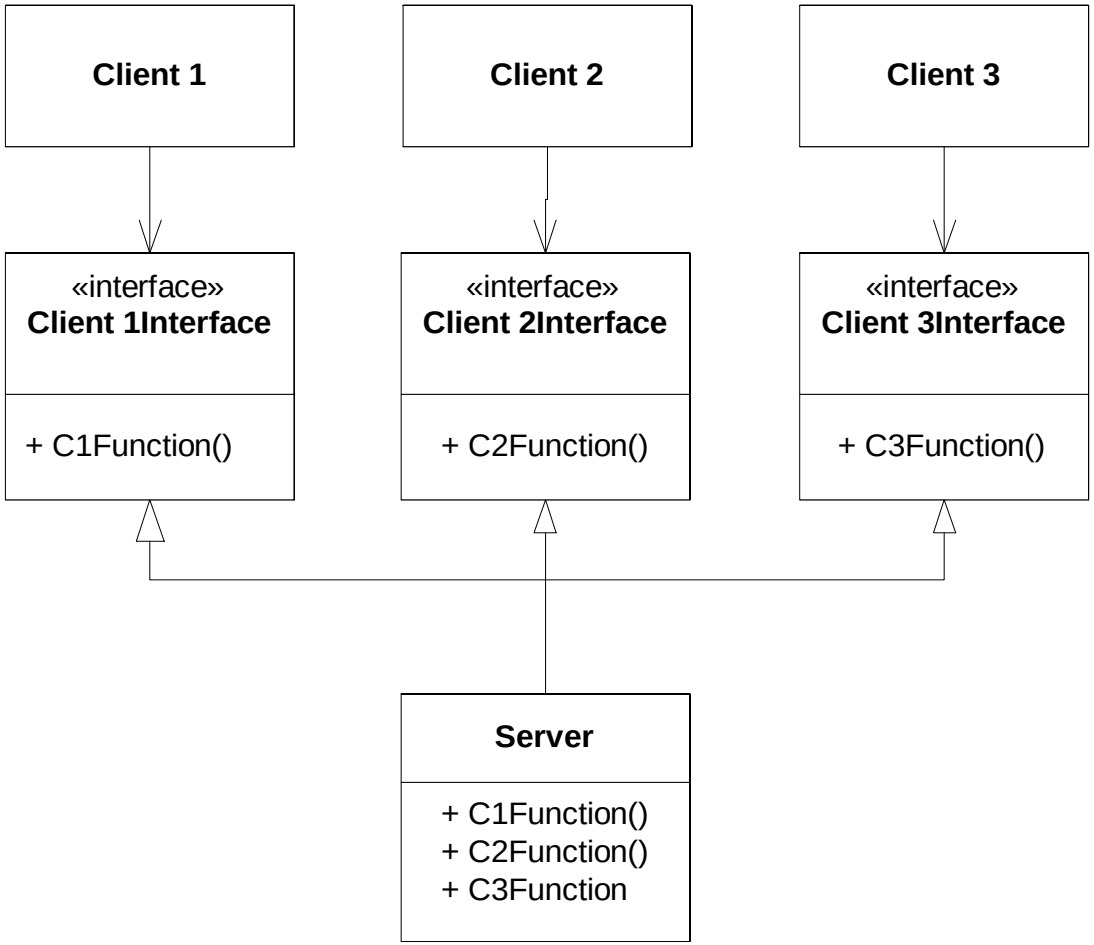
You want me to plug this in, where?

Interface Pollution by "collection"

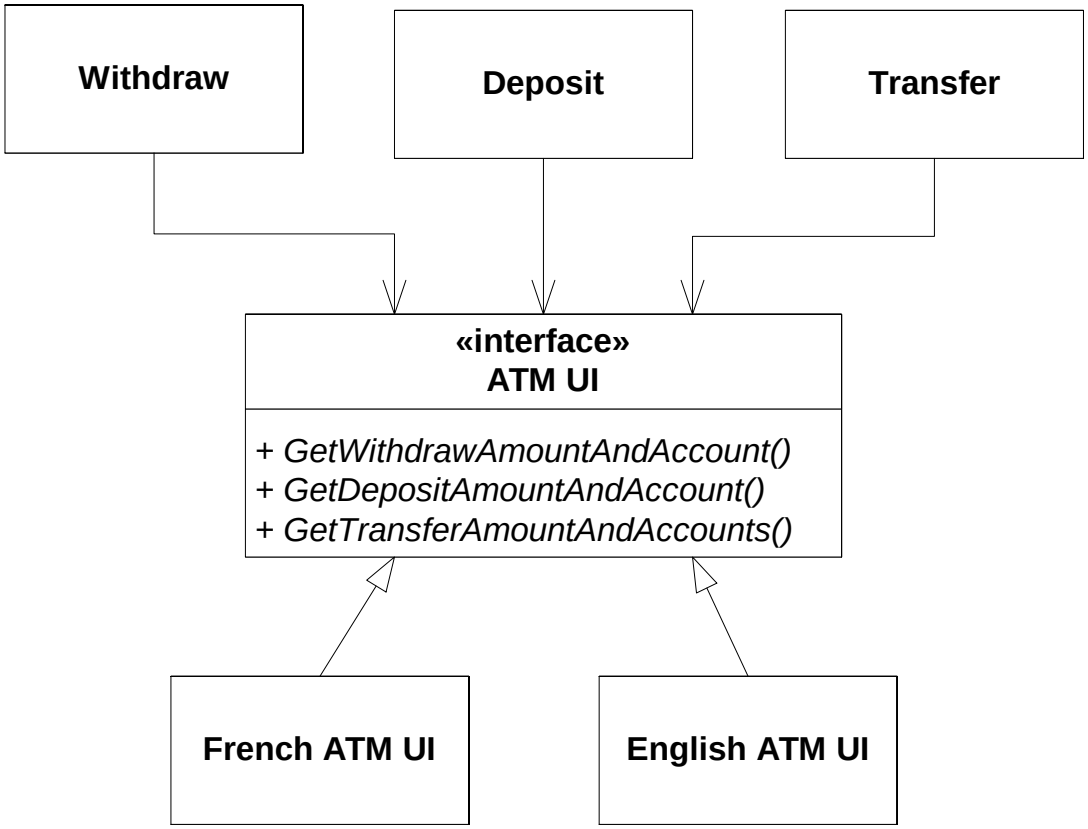
Distinct clients of our class have distinct interface needs.



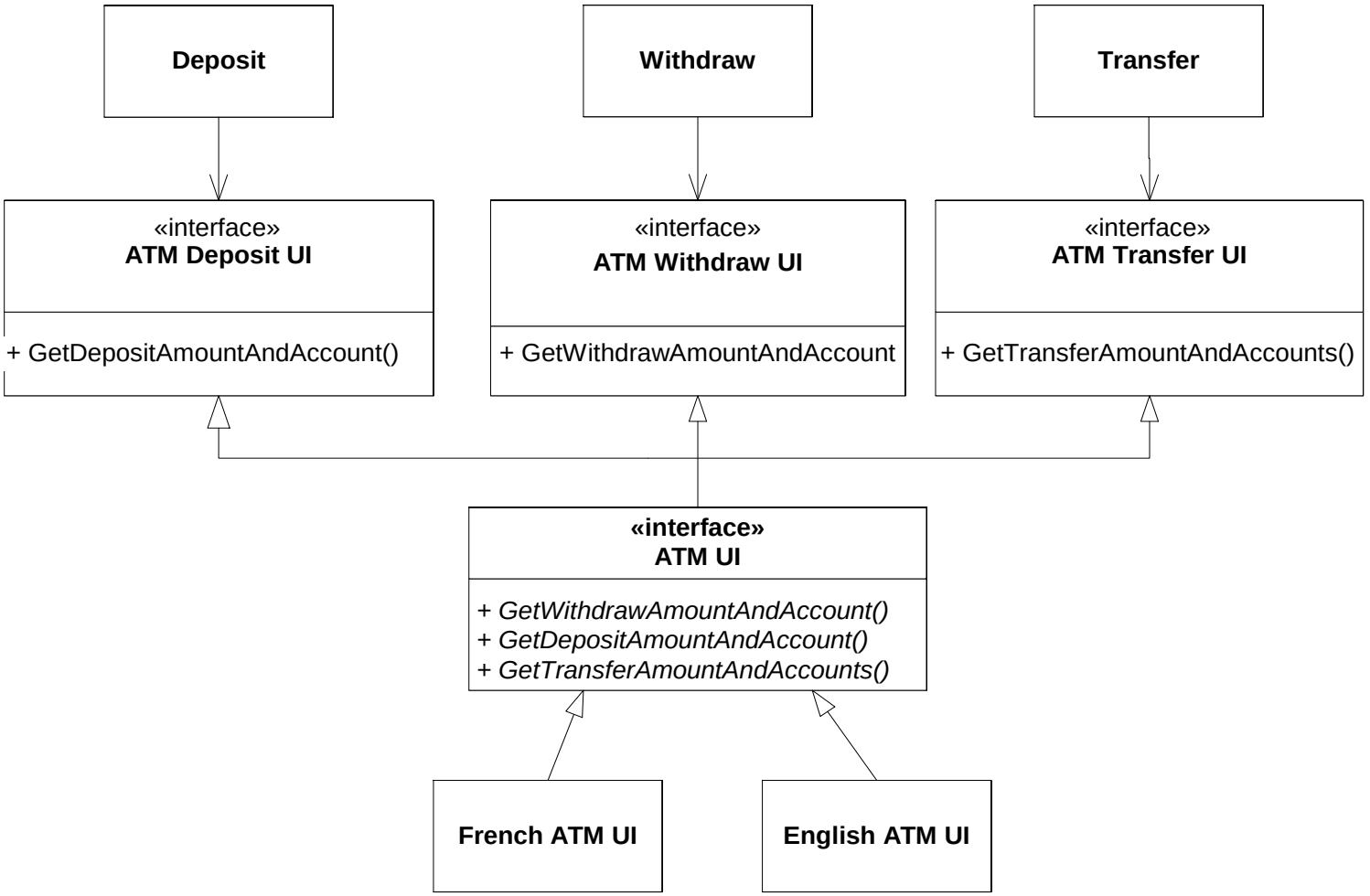
A Segregated Example



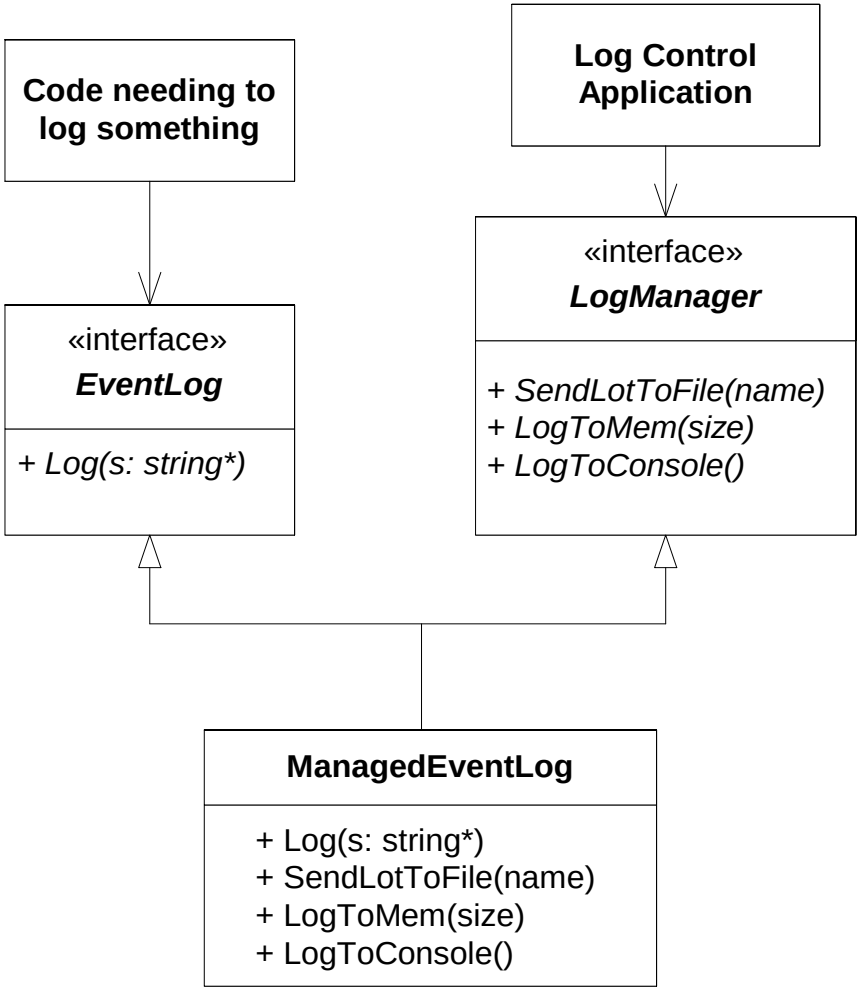
ATM UI Example



A Segregated ATM UI Example



Logger Example



```
// interface segregation principle - bad example
interface IWorker {
        public void work();
        public void eat();
}

class Worker implements IWorker{
        public void work() {
                // ....working
        }
        public void eat() {
                // ...... eating in launch break
        }
}

class SuperWorker implements IWorker{
        public void work() {
                //.... working much more
        }

        public void eat() {
                //.... eating in launch break
        }
}

class Manager {
        IWorker worker;

        public void setWorker(IWorker w) {
                worker=w;
        }

        public void manage() {
                worker.work();
        }
}
```

```
// interface segregation principle - good example
interface IWorker extends Feedable, Workable {
}

interface IWorkable {
        public void work();
}

interface IFeedable{
        public void eat();
}

class Worker implements IWorkable, IFeedable{
        public void work() {
                // ....working
        }

        public void eat() {
                //.... eating in launch break
        }
}

class Robot implements IWorkable{
        public void work() {
                // ....working
        }
}

class SuperWorker implements IWorkable, IFeedable{
        public void work() {
                //.... working much more
        }

        public void eat() {
                //.... eating in launch break
        }
}
```

Dependency Inversion Principle (DIP)

- Depend upon abstractions. Do not depend upon concretions.

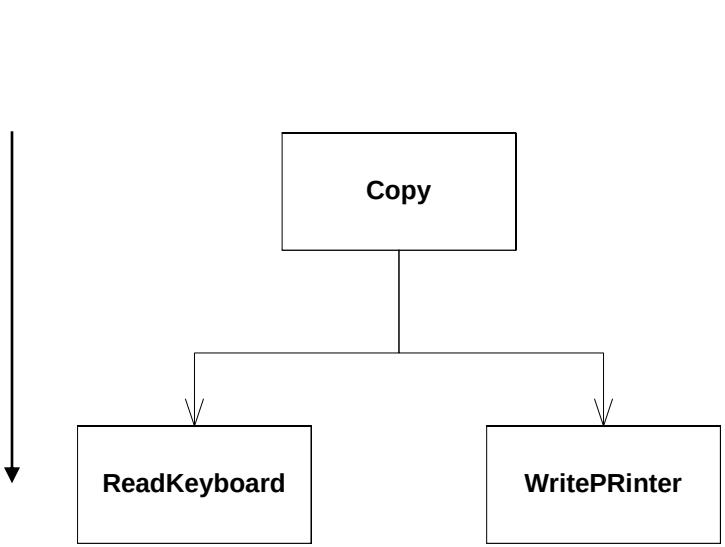
- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend on details. Details should depend on abstractions.

High Level Classes --> Abstraction Layer --> Low Level Classes

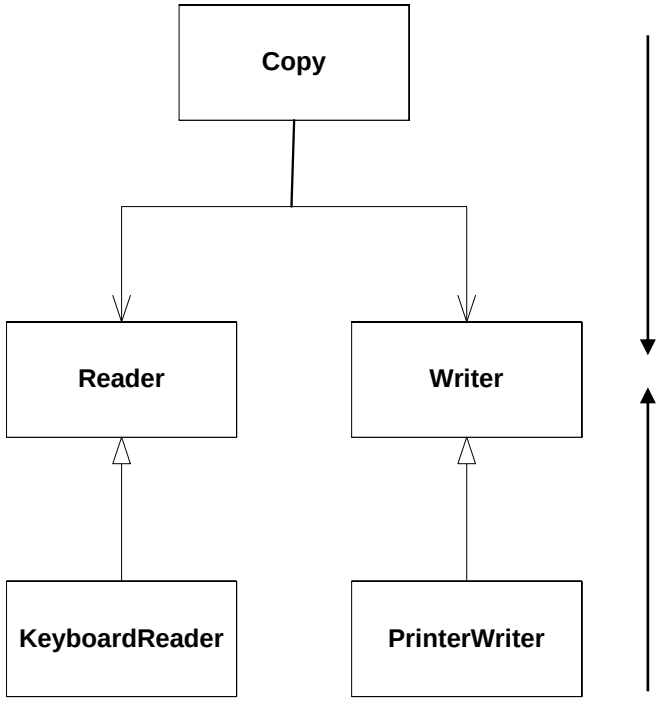
Would you solder a lamp directly to the electrical wiring in a wall?

Dependency Inversion Principle

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



V.



DIP Implications

Everything should depend upon abstractions

- Avoid deriving from concrete classes
- Avoid associating to concrete classes
- Avoid aggregating concrete classes
- Avoid dependencies on concrete components