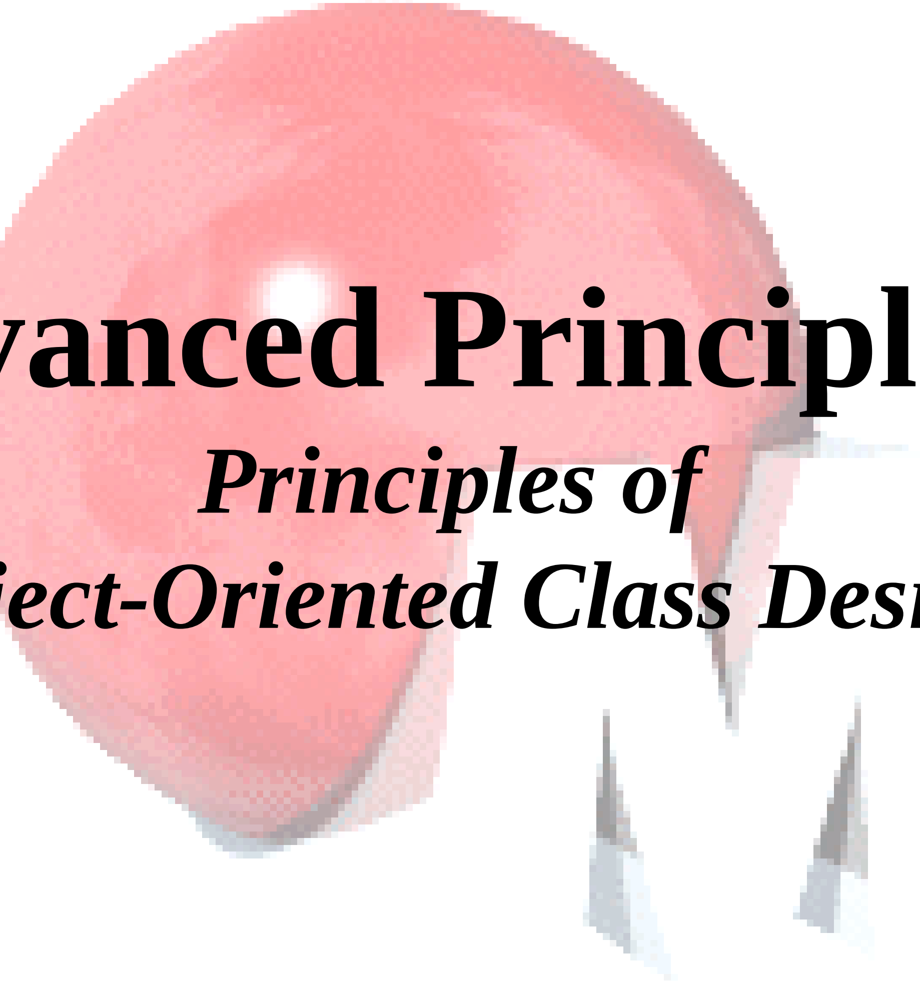
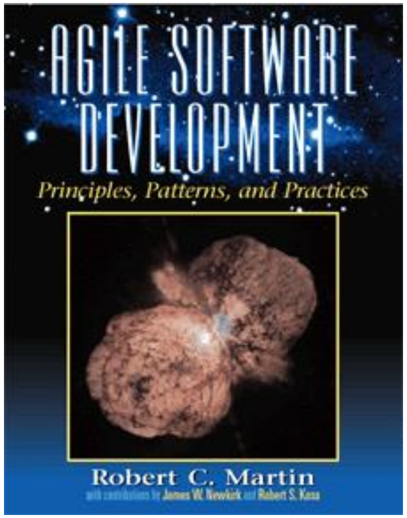
Advanced Principles I

Principles of Object-Oriented Class Design




AGILE SOFTWARE DEVELOPMENT
Principles, Patterns, and Practices
Robert C. Martin
with contributions by James W. Newkirk and Robert S. Koss

Dependency Management

- What is dependency management?
- What bearing does DM have on software?
- What is the result of poor DM?
- What is the advantage of good DM?

What is dependency management?

- A simple idea - as interdependencies increase, features like reusability, flexibility, and maintainability decrease.
- Dependency management is controlling interdependencies.

What bearing does DM have on software?

- Coupling and cohesion are the eternal concerns of software development
- One can say that OO is just a set of tools and techniques for Dependency Management

What is the penalty for practicing poor DM?

A system with poor dependency structure will typically exhibit these four negative traits:

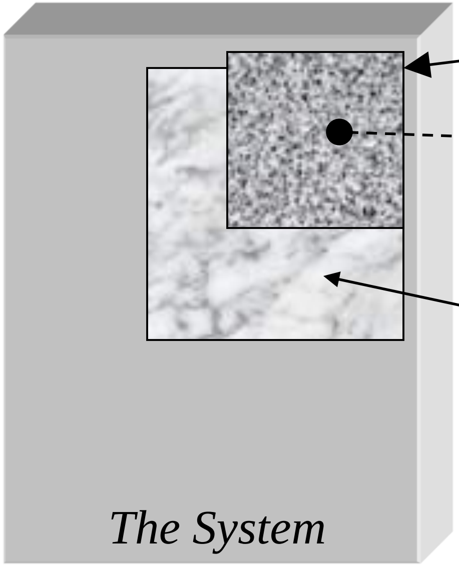
- It is rigid
- It is fragile
- It is not reusable
- It has high viscosity

It is Rigid

Rigidity is the inability to be changed

- The impact of a change cannot be predicted
- If not predicted, it cannot be estimated
- Time and cost cannot be quantified
- Managers become reluctant to authorize change
- Official Rigidity for “Roach Motel” modules

Changes with Rigidity



Officially Rigid Area

Where the change should be made

Where the change must be made now

Are we containing risk, or spreading rot?

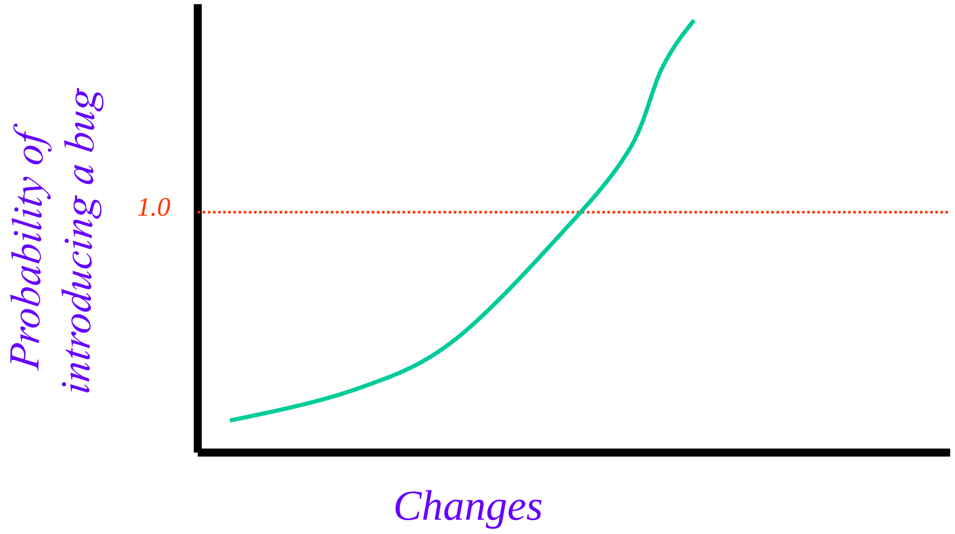
It is Fragile

Software changes seem to exhibit non-local effects

- A single change requires a cascade of subsequent changes
- New errors appear in areas that seem unconnected to the changed areas
- Quality is unpredictable.
- The development team loses credibility

Increasing Risk

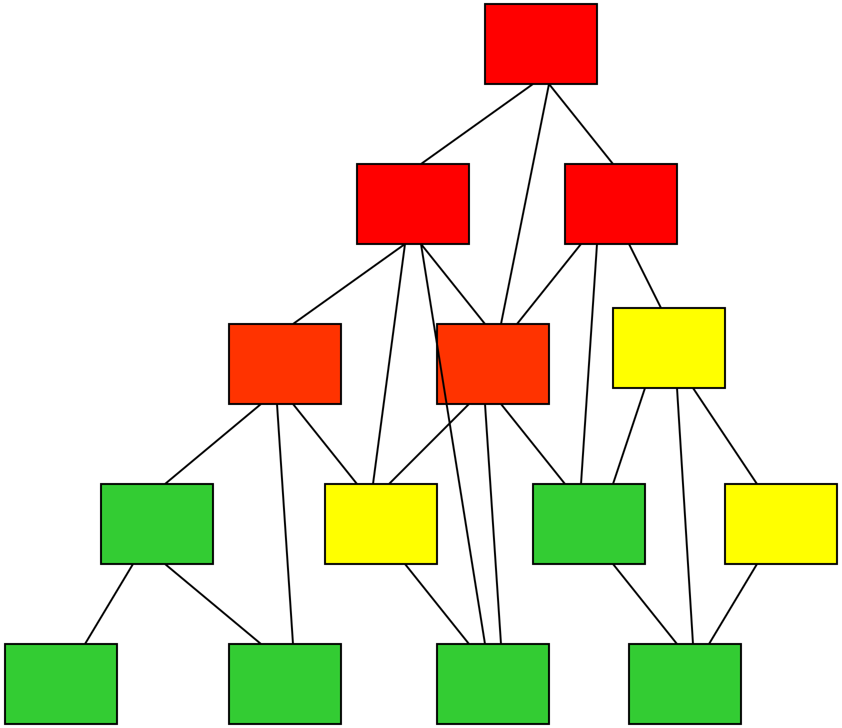
Defects v. Cumulative Modifications



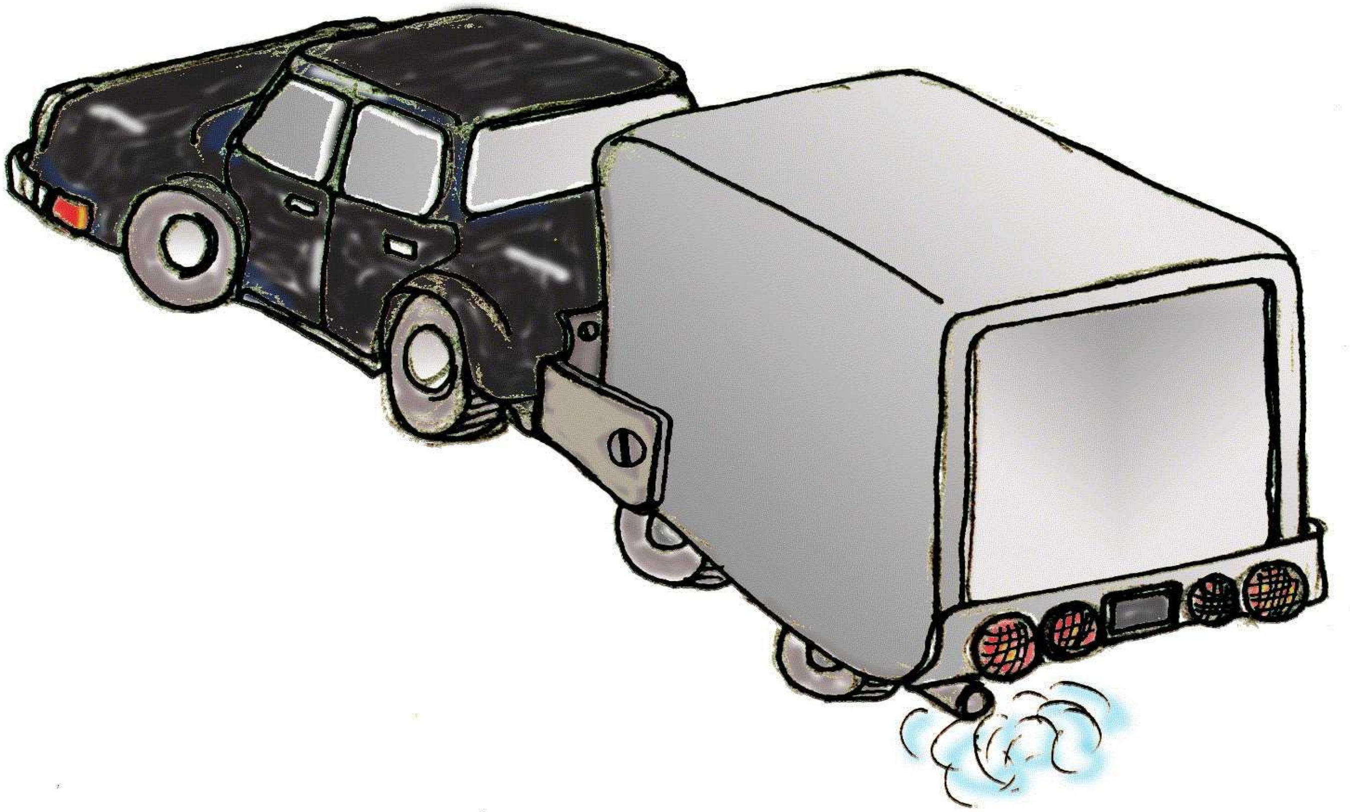
Systems tend to become increasingly fragile over time. Intentional, planned partial rewrites may be necessary to sustain growth and maintenance.

It is not reusable

- Desirable parts of the design are dependent upon undesirable parts
- The work and risk of extracting the desirable part may exceed the cost of redeveloping from scratch.



The Trailer



It has high viscosity

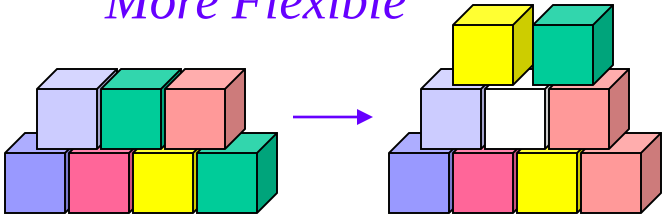
Viscosity is resistance to fluid motion.

- When the "right changes" are *much more difficult* than hacking, the viscosity of the system is high.
- Over time, it will become increasingly difficult to continue developing the product.

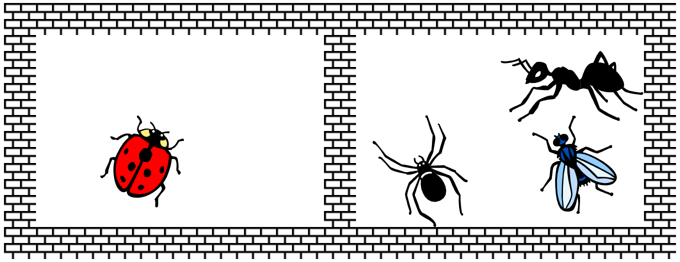
What is the benefit of good DM?

Interdependencies are managed, with firewalls separating aspects that need to vary independently.

More Flexible



Less fragile, the bugs are boxed in



Easier to reuse

Fewer Trailers

Easier to make the right change

Slow the rot

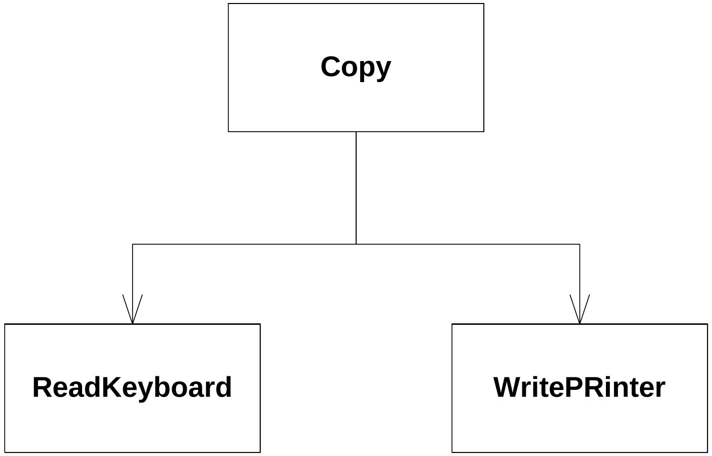
What causes “Code Rot”?

It’s been blamed on stupidity, lack of discipline, and phases of the moon, but...

A case study “The Copy Routine”

First Version

All designs start well



```
void copy(void)
{
     int ch;
     while( (ch=ReadKeyboard()) != EOF)
          WritePrinter(ch);
}
```

The program is an overnight success!
How could it be more simple, elegant, and maintainable?

Second Version

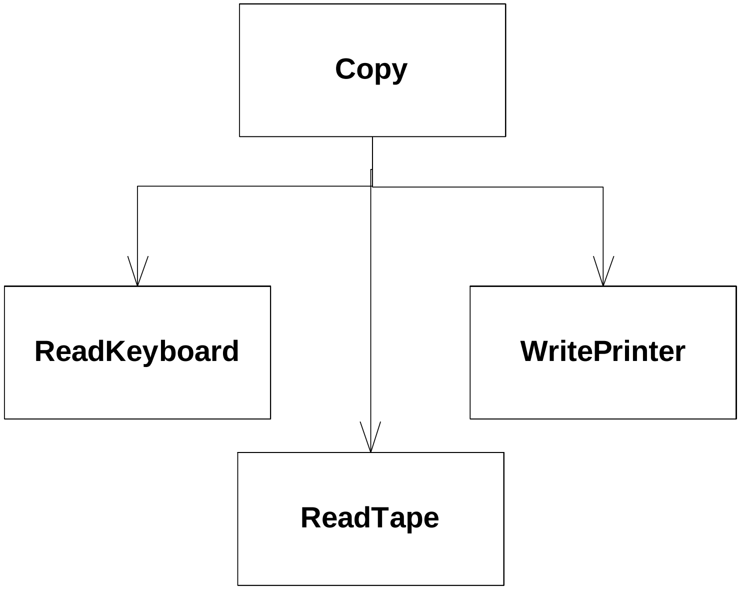
Oh, no! Nobody said the requirements might change!

- We sometimes want to read from paper tape reader.
- We could put a parameter in the call, but we have hundreds of users already!
- No big deal, this is just an exception… we can make it work.

Second Version Design

```
bool GtapeReader = false; // remember to clear

void copy(void)
{
    int ch;
    while( (ch=GtapeReader ? ReadTape() : ReadKeyboard()) != EOF)
       WritePrinter(ch);
}
```



Third Version

How unexpected! Requirements changed again!

It seems that sometimes we need to write to a paper tape punch. We’ve had this problem before, and just added a flag. Looks like it should work again.

```
bool GtapeReader = false;
Bool GtapePunch = false;
// remember to clear

void copy(void)
{
    int ch;
    while( (ch=GtapeReader ? ReadTape() : ReadKeyboard()) != EOF)
       GtapePunch ? WritePunch(ch) : WritePrinter(ch);
}
```

Example of a Good Design

First and only version.

```
void Copy()
{
   int c;
   while( (c=getchar()) != EOF)
      putchar(c);
}
```

But wait! Aren't we supposed to be learning OO design? This isn't OO is it?

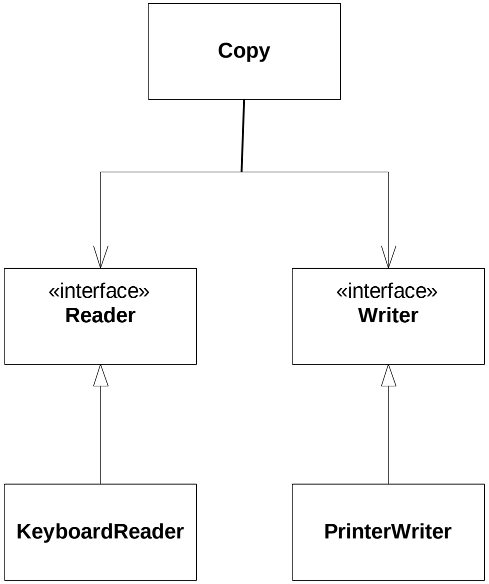
...is it?

It is a small program based on abstractions!

- FILE is an abstraction
  - It represents some kind of byte stream
  - It has many variations
- It has methods
  - Read, Write, getchar, putchar, etc
  - The methods are *dynamically* bound

FILE is a class, just implemented differently.

Rephrased in OO



```
interface Reader
{ char read(); }

interface Writer
{ void write(char c); }

public class Copy
{
   Copy(Reader r, Writer w)
   {
       itsReader = r;
       itsWriter = w;
   }
   public void copy()
   {
       int c;
       while( (c==itsReader.read()) != EOF )
           itsWriter.write(c);
   }
   private Reader itsReader;
   private Writer  itsWriter;
}
```

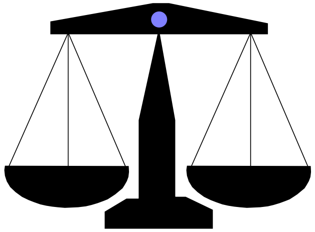
Dependency Management Review

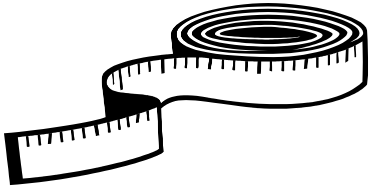
- Why do programs tend to rot over time?
- What is dependency management?
- What are four qualities of good designs?
- Are OO programs always simpler than non-OO versions?
- Why would anyone want to use a paradigm that may result in more complex designs?
- Why are compile and link time important?

Class Design Principles

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

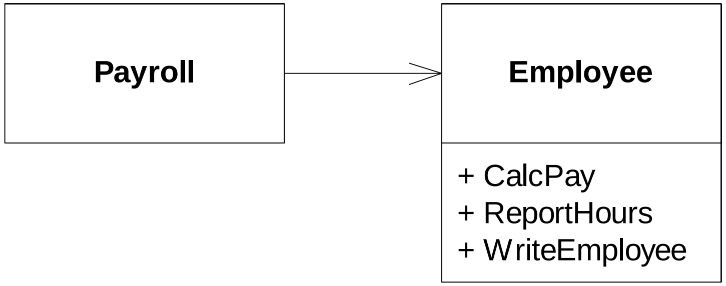
- SRP: The Single Responsibility Principle
- OCP: The Open/Closed Principle
- LSP: The Liskov Substitution Principle
- ISP: The Interface Segregation Principle
- DIP: The Dependency Inversion Principle



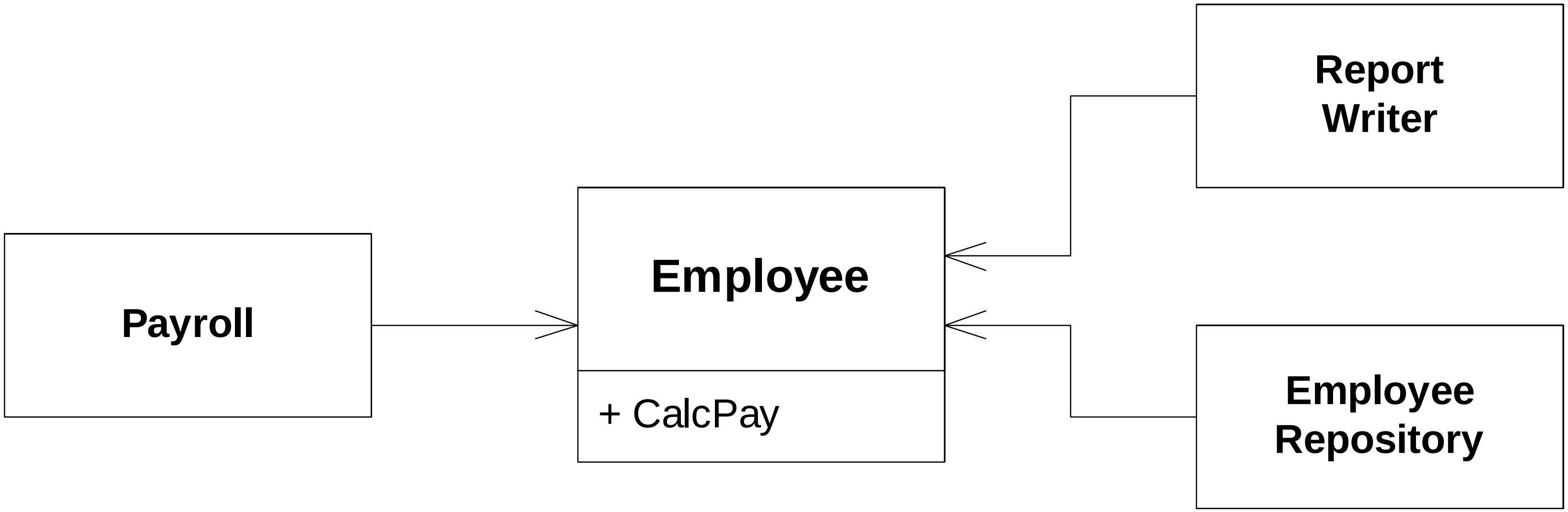


The Single Responsibility Principle

- A class should have one, and only one, reason to change.



The Single Responsibility Principle. (SRP)



Open/Closed Principle

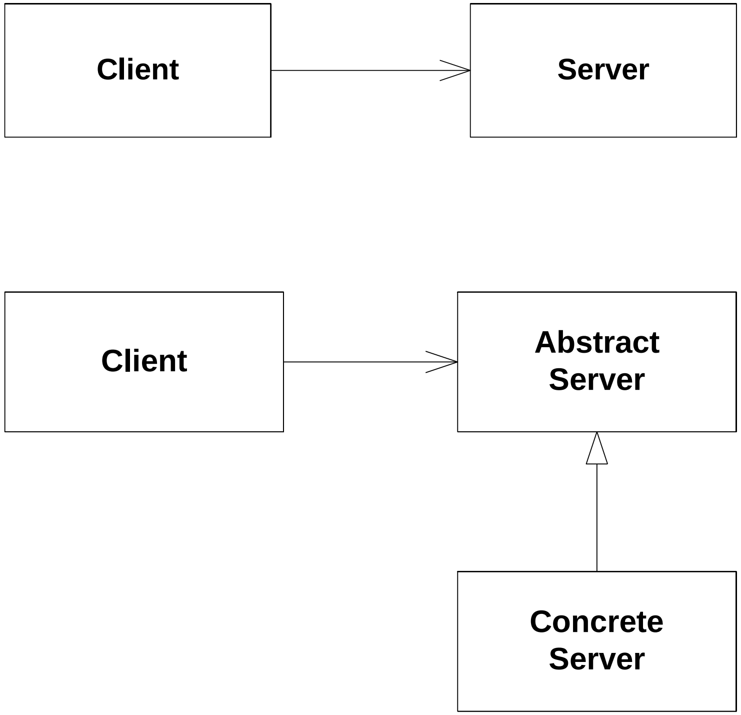
> *"Modules should be open for extension, but closed for modification"*
> *-Bertrand Meyer*

- 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

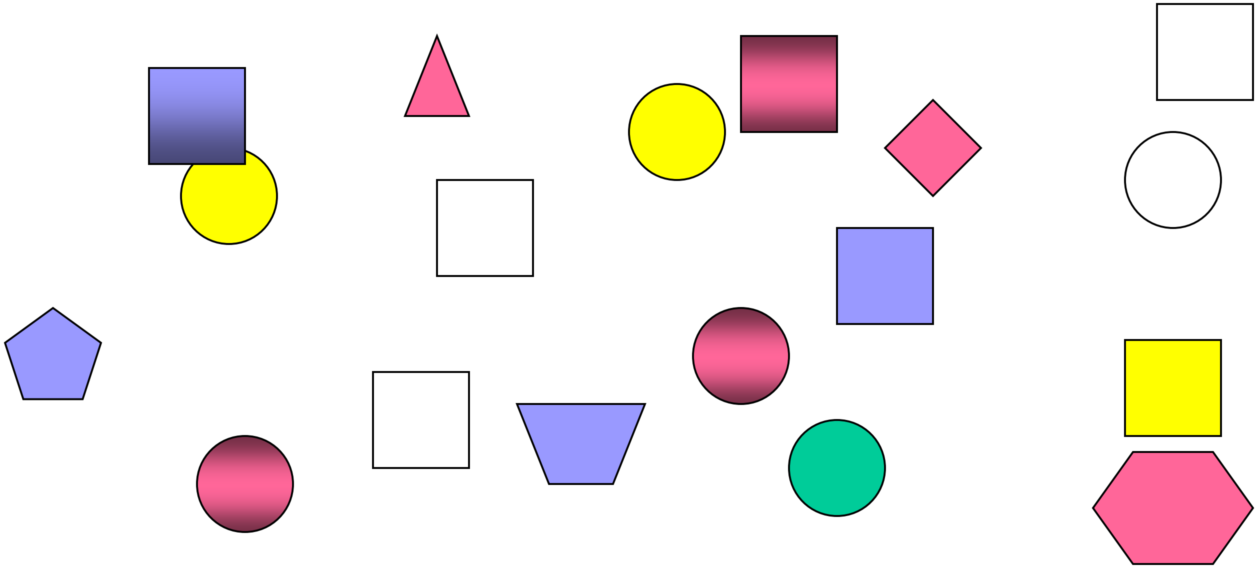
Abstraction is the most important word in OOD

- 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 editd.
  - *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;
};
```

Circle.h

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

Square.h

```
Class Square: 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

Picking Targets

- Technicians and domain users list
  - Ways that the system is expected to change
  - Ways that the system has already changed
- Isolate these to *kinds of* changes, not specific changes
  - schema changes
  - sensor hardware changes
  - data store technology
- Keep this list handy throughout design
- Don’t *make* the changes, just *allow* for them.

Open/Closed Review

- What does the open/closed principle say?
- What does that mean practically?
- How can it be achieved?
- What is strategic closure?
  - How can this be achieved in design?
  - What if you can’t close completely?

Liskov Substitution Principle

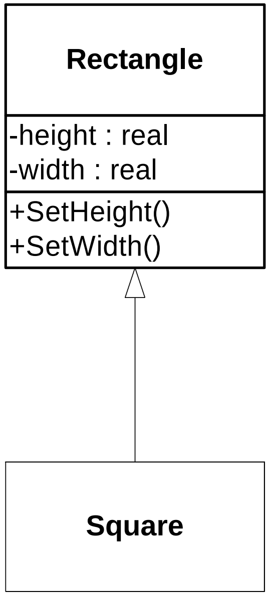
Derived classes must be usable through the base class interface, without the need for the user to know the difference.

- All derived classes must be substitutable for their base classes
- This principle guides us in the creation of abstractions.

Square/Rectangle

A square is-a rectangle, right? So lets consider Square as a subtype of Rectangle.

Uh, oh. This doesn’t quite seem to fit



We can ***make*** *it work:*

```
void Square::SetWidth(double w)
{
   width = w;
   height = w;
}
void Square::SetHeight(double h)
{
   width = h;
   height = h;
}
```

Substitution… denied!

- It is reasonable for users of a rectangle to expect that height and width may change independently.
- These expectations are preconditions and postconditions
  - Bertrand Meyer calls it “Design by Contract”
  - Post condition contract for rectangle is
    - width = new Width
    - height = old height
- Square violates Rectangle’s contract

Liskov Substitution Principle (cont.)

- A client of rectangle expects height and width to be changed independently
  - void setAspectRatio(Rectange* r, double ratio);
- By deriving Square from Rectangle, we are allowing someone to set the aspect ratio of a Square !
- We can still make it work
  - if (typeid(r) == typeid(Rectangle))
  - Violates Open/Closed Principle !

Liskov Substitution Principle (cont.)

- Design by Contract
  - Bertrand Meyer
  - Preconditions, postconditions, invariants
- Rectangle's postconditions for setWidth()
  - width = newWidth
  - length = oldLength
- Square can require no more of clients, nor promise any less
  - Doesn't maintain invariant of length
  - Violates the contract

LSP Guides the Creation of Abstractions

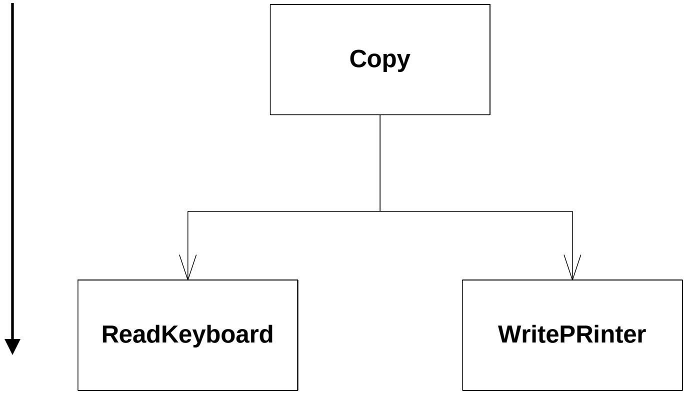
- Abstractions do not stand alone
- Abstractions don’t always conform to *real world* expectations
- Violating LSP is tantamount to violating the OCP

LSP Review

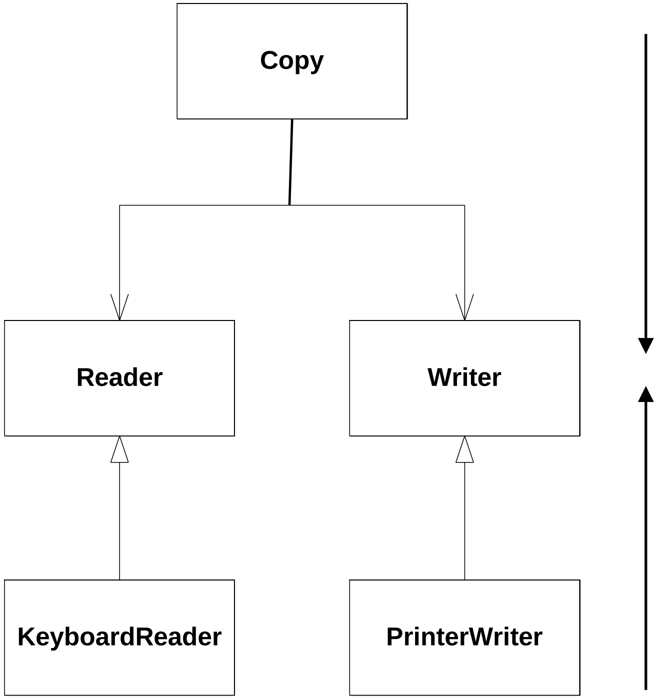
- What does the LSP state?
- What is the risk if LSP is violated?
- What is Design by Contract?

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

Dependency Inversion Principle (cont.)

- Legitimate reasons to violate
  - Creation of objects
    - new Circle   creates a dependency on concrete class
    - localize these dependencies using factories
  - Nonvolatile classes
    - string, vector, etc.
      - providing they are stable

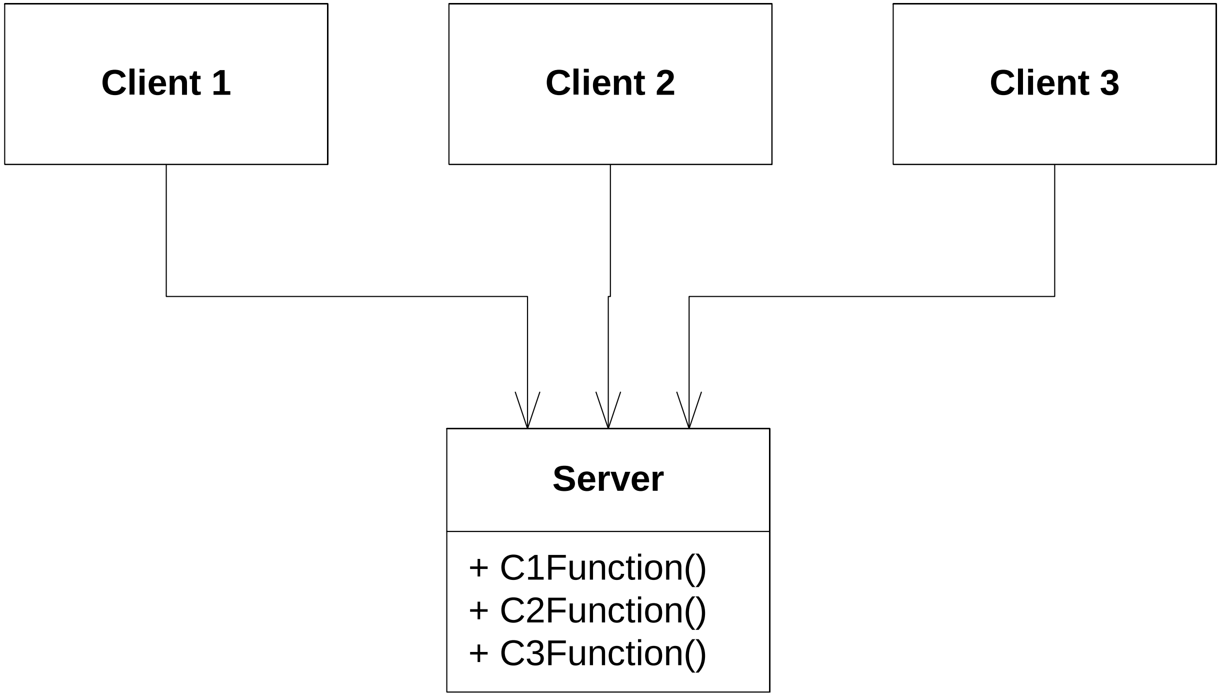
Interface Segregation Principle

Helps deal with “fat” or inappropriate interfaces

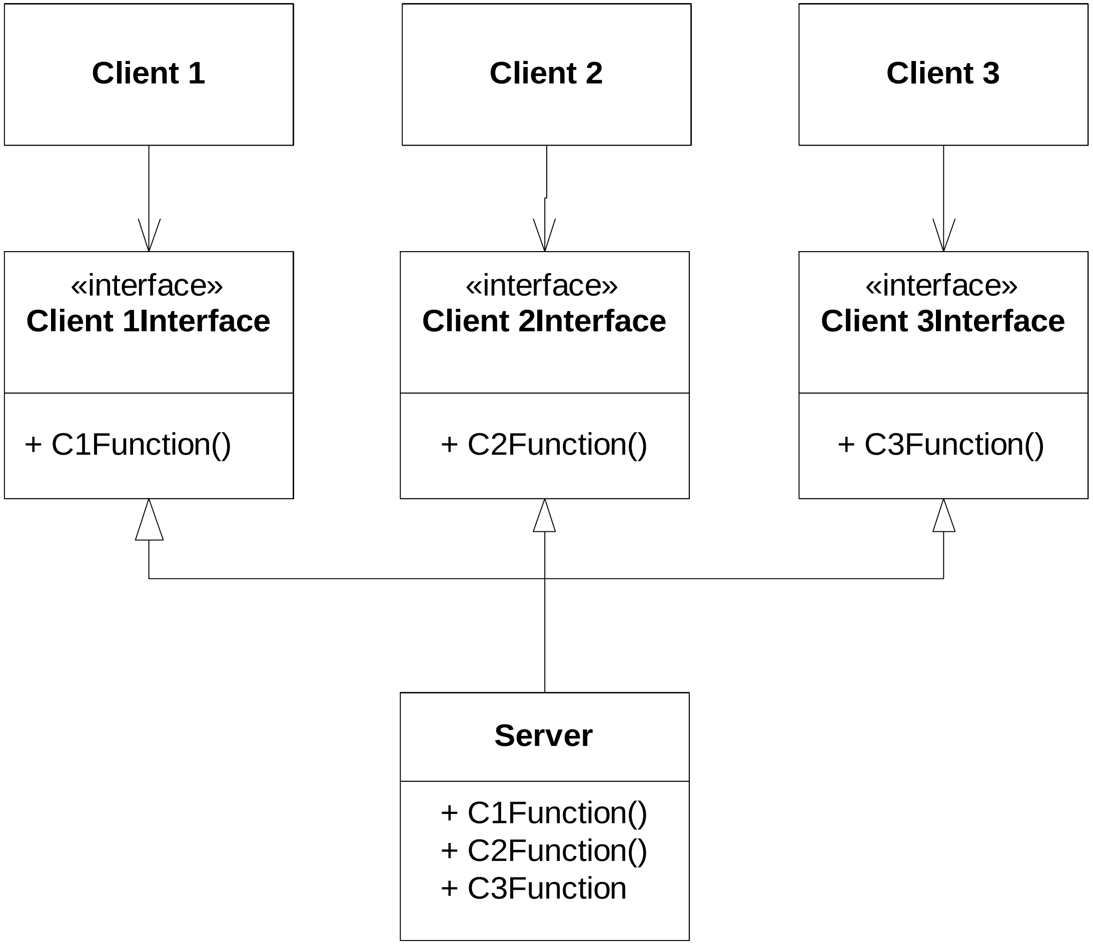
- Sometimes class methods have various groupings.
- These classes are used for different purposes.
- Not all users rely upon all methods.
- This lack of cohesion can cause serious dependency problems
- These problems can be refactored away.

Interface Pollution by "collection"

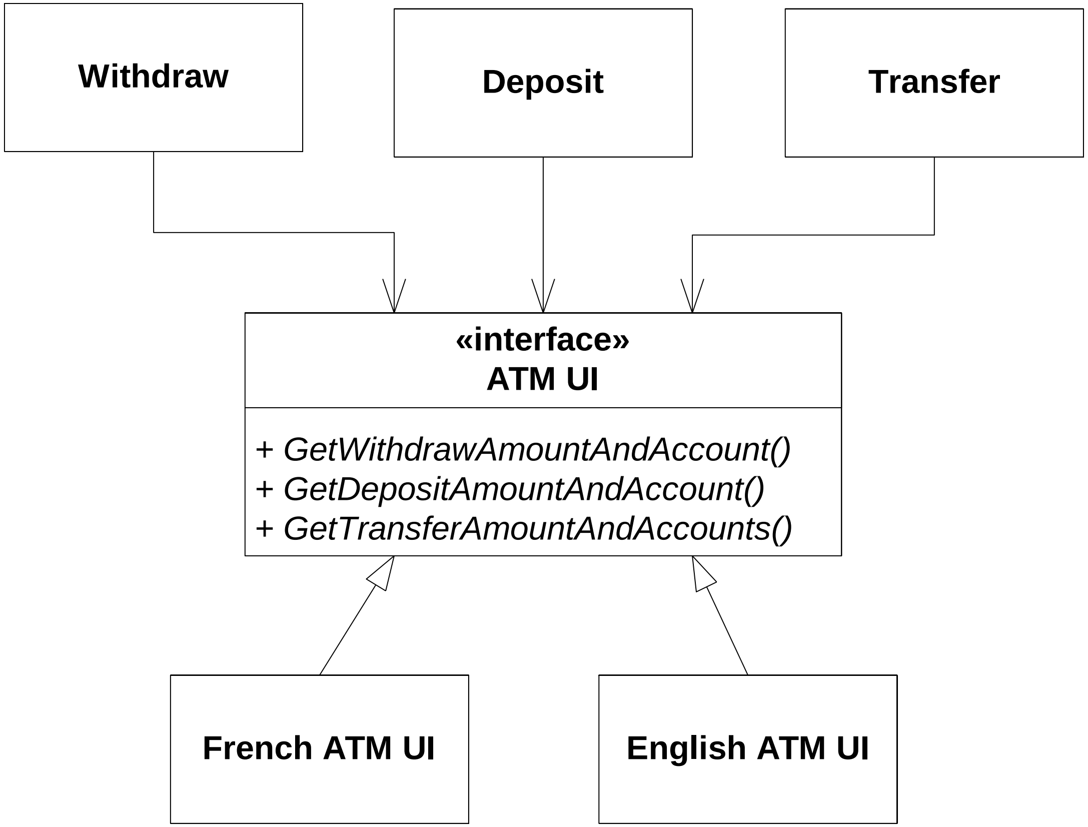
Distinct clients of our class have distinct interface needs.



A Segregated Example



ATM UI Example



Withdraw

Deposit

Transfer

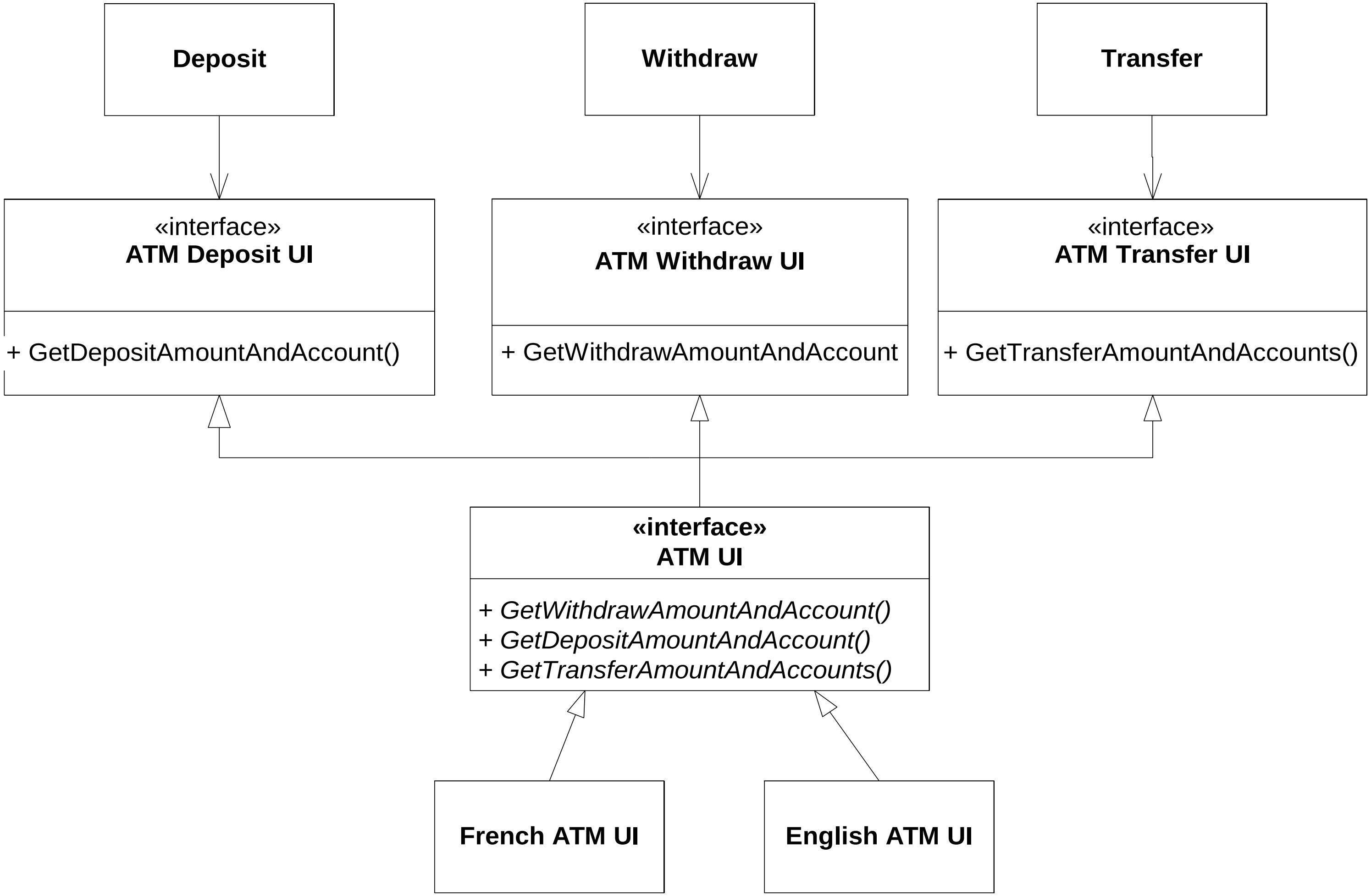
«interface»
ATM UI

+ *GetWithdrawAmountAndAccount()*
+ *GetDepositAmountAndAccount()*
+ *GetTransferAmountAndAccounts()*

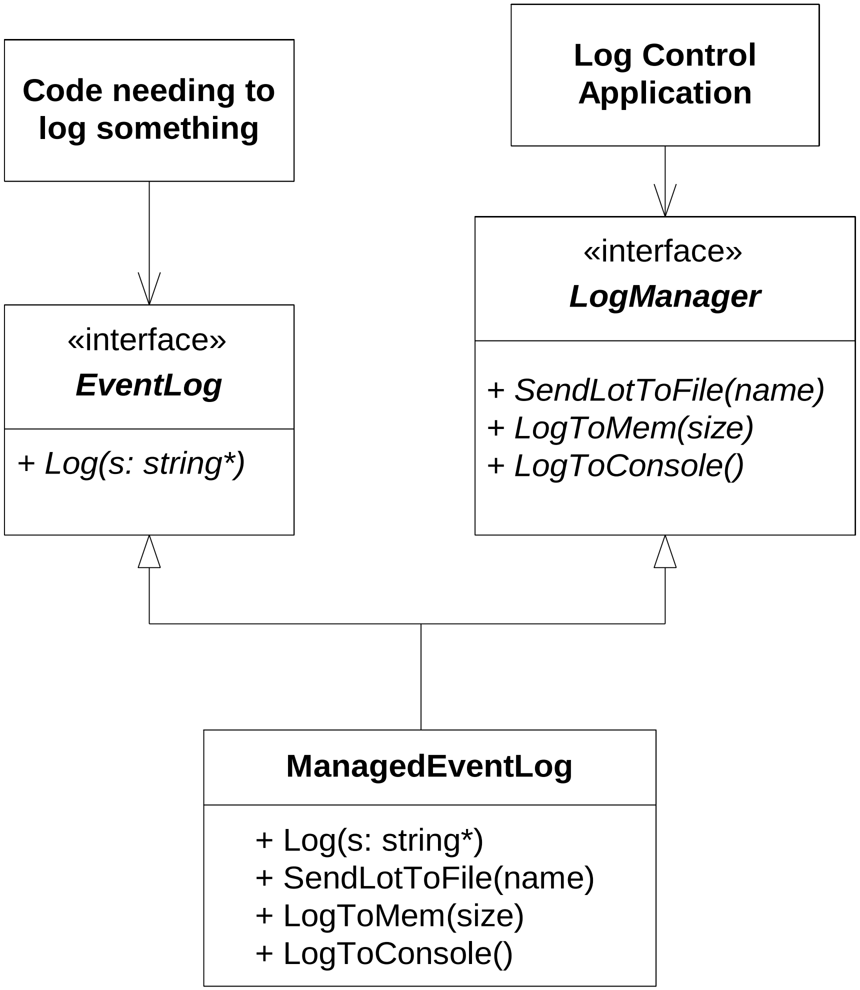
French ATM UI

English ATM UI

A Segregated ATM UI Example


Deposit
Withdraw
Transfer
«interface»
ATM Deposit UI
+ GetDepositAmountAndAccount()
«interface»
ATM Withdraw UI
+ GetWithdrawAmountAndAccount
«interface»
ATM Transfer UI
+ GetTransferAmountAndAccounts()
«interface»
ATM UI
+ GetWithdrawAmountAndAccount()
+ GetDepositAmountAndAccount()
+ GetTransferAmountAndAccounts()
French ATM UI
English ATM UI

Logger Example



ISP Review

- What is the ISP?
- What does it affect?
- How do these problems arise?
- Does it really provide a real solution or just a restructuring of the problem?
- When is it worth the effort?

Four Class Design Principles - Review

- OCP Extend function without editing code
- LSP Child instances substitute cleanly for base
- DIP Depend on abstractions instead of details
- ISP Split interfaces to manage dependencies