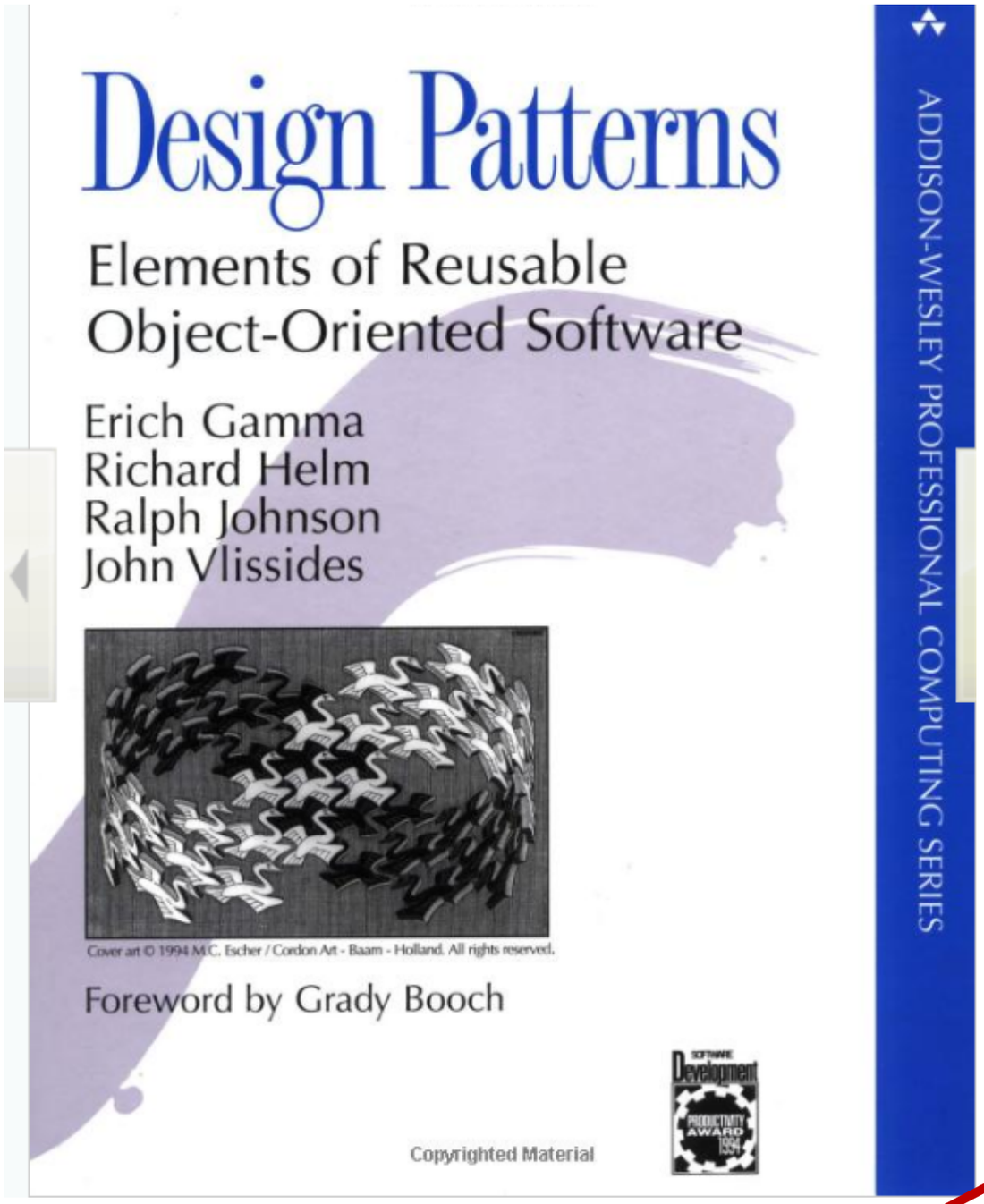
DESIGN PRINCIPLES

CAC 430

Birmingham-Southern College

You should really get this book!



Gang of Four

?????

Amazon Bestsellers Rank: #2,354 in Books (See Top 100 in Books)
#1 in Books > Computers & Internet > Programming > Software Design, Testing & Engineering > **Software Reuse**
#1 in Books > Nonfiction > Foreign Language Nonfiction > **French**
#1 in Books > Computers & Internet > Computer Science > Software Engineering > **Design Tools & Techniques**

What is a Pattern ?
The "Alexandrian" Definition

Each pattern describes a problem
which occurs over and over again in our environment,
and then describes
the core of the solution to that problem,
in such a way that
you can use this solution a million times over,
without ever doing it the same way twice

C. Alexander, "*The Timeless Way of Building*", 1979

Design Patterns

- Design patterns represent solutions to problems that arise when developing software within a particular context
  - Patterns = **Problem/Solution** pair in **Context**
- Capture static and dynamic structure and collaboration among key participants in software designs
- Facilitate reuse of successful software architectures and design
  - i.e. the "*design of masters*"…

What Makes it a Pattern ?

A pattern must...

- ...solve a problem
  - i.e., it must be useful
- ...have a context
  - it must describe where the solution can be used
- ...recur
  - must be relevant in other situations

- ... teach
  - provide sufficient understanding to tailor the solution
- ... have a name
  - referred consistently

GoF Form of a Design Pattern

Pattern name and classification

Intent

what does pattern do

Also known as

other known names of pattern (if any)

Motivation

the design problem

Applicability

situations where pattern can be applied

Structure

a graphical representation of classes in the pattern

Participants

the classes/objects participating and their responsibilities

Collaborations

of the participants to carry out responsibilities

GoF Form of a Design Pattern

Consequences
trade-offs, concerns

Implementation
hints, techniques

Sample code
code fragment showing possible implementation

Known uses
patterns found in real systems

Related patterns
closely related patterns

Why are Patterns Important?

- "Patterns provide an incredibly dense means of efficient and effective communication between those who know the language." *Nate Kirby*
- "Human communication is the bottleneck in software development. If patterns can help developers communicate with their clients, their customers, and each other, then patterns help fill a crucial need in our industry."
  Jim Coplien
- "Patterns don't give you code you can drop into your application, they give you experience you can drop into your head."
  Patrick Logan
- "Giving someone a piece of code is like giving him a fish; giving him a pattern is like teaching him to fish."
  Don Dwiggins

Experiential Reuse Benefits

- Mature engineering disciplines have handbooks of solutions to recurring problems. All certified professional engineers in these fields have been trained in the contents of these handbooks.
- In an experiment, teams of leading heart surgeons from five New England medical centers observed one another's operating room practices and exchanged ideas about their most effective techniques. The result? A 24% drop in their overall mortality rate for coronary bypass surgery = 74 fewer deaths than predicted.

An Analogy: Becoming a Chess Master

- *Learn the rules and physical requirements*
  - e.g., pieces, legal movements, chess board geometry and orientation.

- *Learn the principles*
  - e.g., relative value of certain pieces, strategic values of center squares, power of a threat etc.

- To become a master, one must *study the games of other masters*
  - these games contain patterns that must be understood, memorized and applied repeatedly

- There are hundreds, if not thousands, of such patterns

Becoming a Software Design Master

- *First learn the rules*
  - e.g., the algorithms, data structures and languages of software
- *Then learn the principles*
  - e.g., principles that govern different programming paradigms
    - structured, modular, object-oriented, etc

- To truly master software design, one must *study the design of other masters*
  - these designs contain *patterns* that must be understood, memorized and applied repeatedly
- There are thousands of these patterns

Drawbacks of Design Patterns

- Patterns do not lead to direct code reuse (rather, they enable *experiential* reuse)
- Patterns are deceptively simple
- Integrating patterns into a software development process is a human-intensive activity
- Teams may suffer from patterns overload

When your only tool is a hammer, all the problems look like a nail…

- When first learning patterns, all problems begin to look like the problem under consideration – try to avoid this!
  - Similar to someone just learning to play chess and using the same strategy everywhere – eventually you will get burned!

Problem X
Problem Y
Problem Z

Patterns to help with design changes…

Designing for Change – Causes for Redesign (I)

- Creating an object by specifying a class explicitly
  - Commits to a particular implementation instead of an interface
  - Can complicate future changes
  - Create objects indirectly
  - Patterns: Abstract Factory, Factory Method, Prototype
- Dependence on specific operations
  - Commits to one way of satisfying a request
  - Compile-time and runtime modifications to request handling can be simplified by avoiding hard-coded requests
  - Patterns: Chain of Responsibility, Command

Causes for Redesign (II)

- Dependence on hardware and software platform
  - External OS-APIs vary
  - Design system to limit platform dependencies
  - Patterns: Abstract Factory, Bridge
- Dependence on object representations or implementations
  - Clients that know how an object is represented, stored, located, or implemented might need to be changed when object changes
  - Hide information from clients to avoid cascading changes
  - Patterns: Abstract factory, Bridge, Memento, Proxy

Causes for Redesign (III)

- Algorithmic dependencies
  - Algorithms are often extended, optimized, and replaced during development and reuses
  - Algorithms that are likely to change should be isolated
  - Patterns: Builder, Iterator, Strategy, Template Method, Visitor
- Tight coupling
  - Leads to monolithic systems
  - Tightly coupled classes are hard to reuse in isolation
  - Patterns: Abstract Factory, Bridge, Chain of Responsibility, Command, Facade, Mediator, Observer

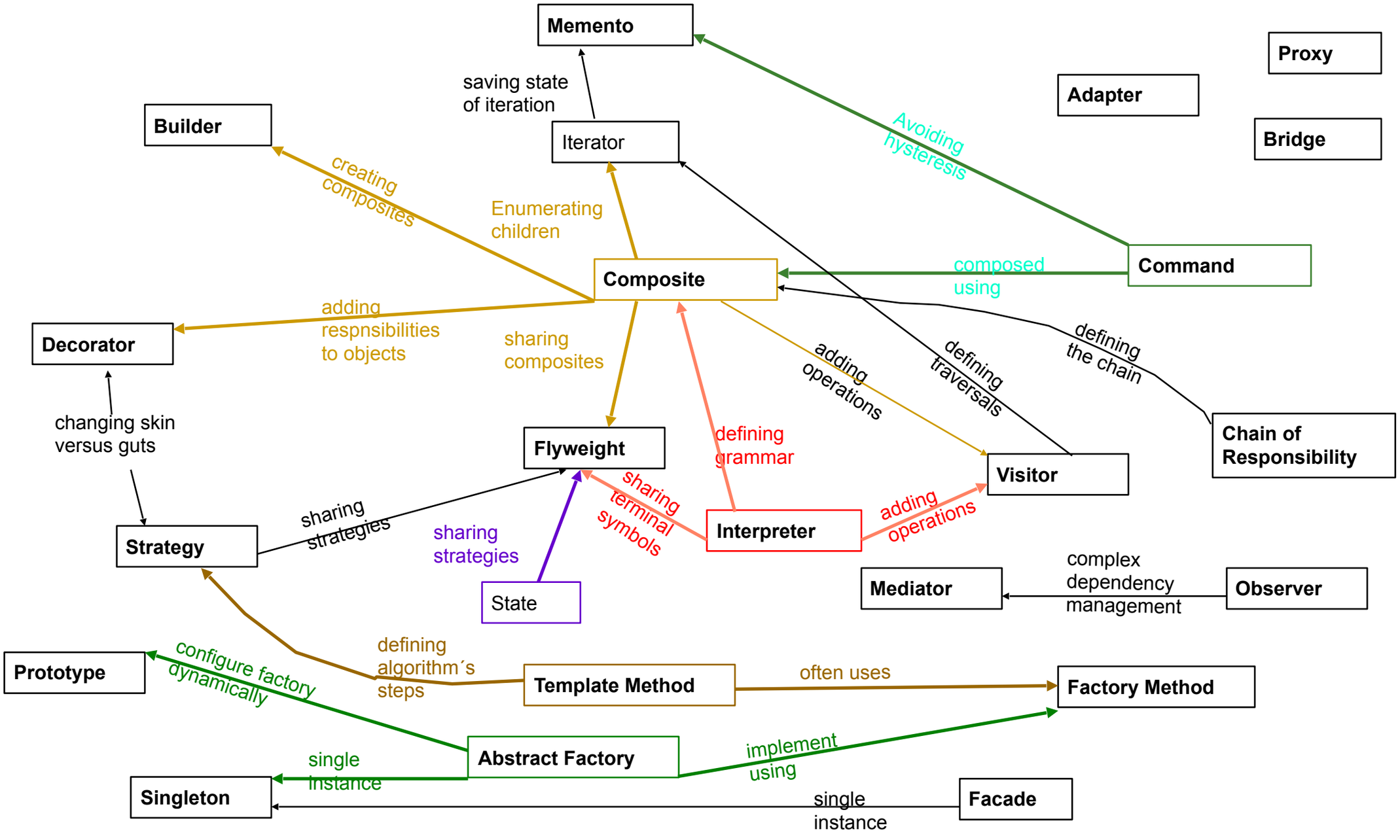
Causes for Redesign (IV)

- Extending functionality by subclassing (can be bad)
  - Requires in-depth understanding of the parent class
  - Overriding one operation might require overriding another
  - Can lead to an explosion of classes (for simple extensions)
  - Patterns: Bridge, Chain of Responsibility, Composite, Decorator, Observer, Strategy
- Inability to alter classes conveniently
  - Sources not available
  - Change might require modifying lots of existing classes
  - Patterns: Adapter, Decorator, Visitor

Design Pattern Space

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Relations among Design Patterns



Many different kinds of patterns

- Small printing industry grew out of patterns:
- Patterns for many different contexts:
  - Analysis/Requirements patterns
  - Middleware/infrastructure
  - Domain-specific patterns
  - Real-time patterns
  - HPC patterns
  - UML and Patterns
  - Anti-patterns

DESIGN PRINCIPLES

- Single Responsibility Principle
- Open-Closed Principle
- Liskov Substitution Principle
- Dependency Inversion Principle
- Interface Segregation Principle

SINGLE RESPONSIBILITY PRINCIPLE (SRP)

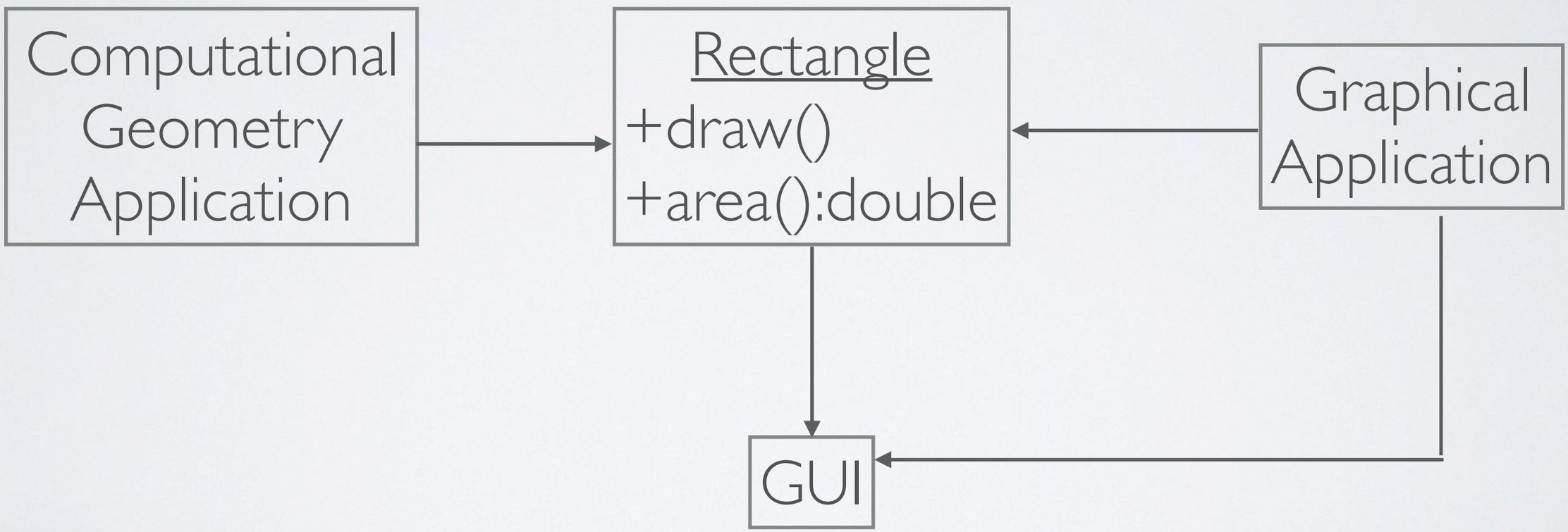
- Cohesion: refers to the degree to which elements inside a module belong together

- SRP: There should never be more than one reason for a class to change

- If a class has more than one responsibility, then the responsibilities become coupled

SINGLE RESPONSIBILITY PRINCIPLE (SRP)

- Example:



SINGLE RESPONSIBILITY PRINCIPLE (SRP)

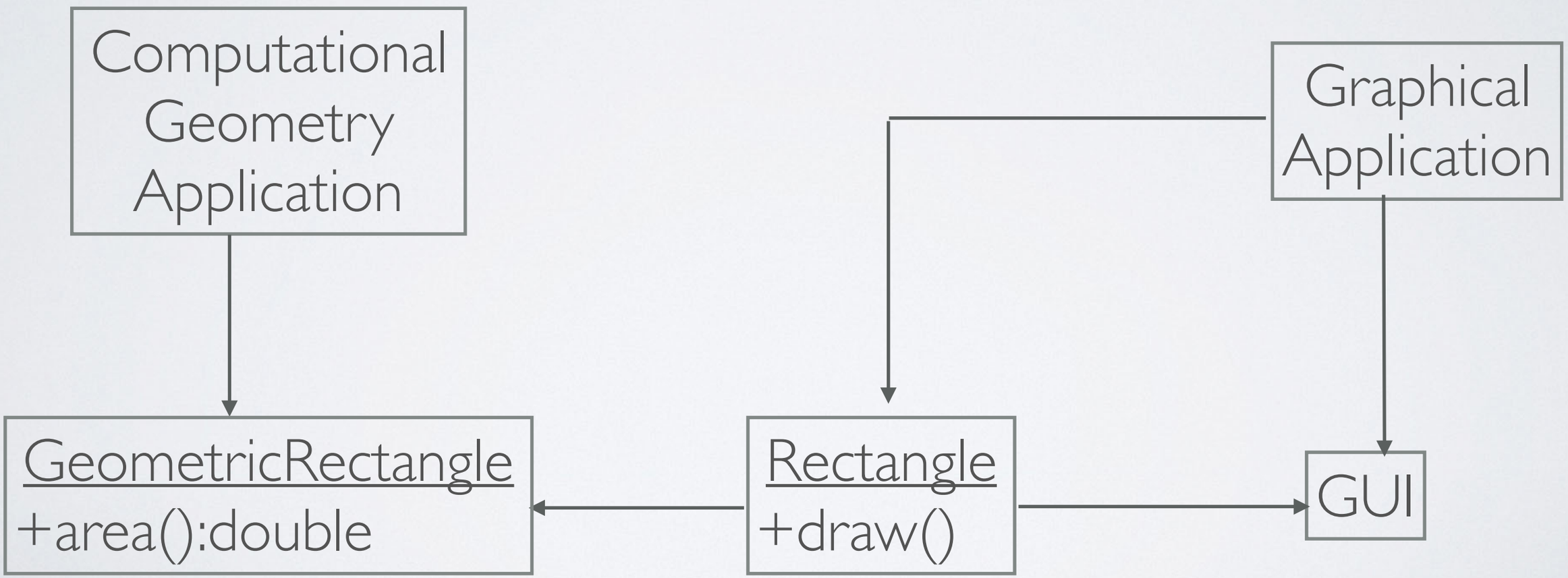
- Why are the two responsibilities a problem?

SINGLE RESPONSIBILITY PRINCIPLE (SRP)

- Why are the two responsibilities a problem?
  - The GUI would be included in the Computational Geometry Application (taking time and memory)
  - If a change to the Graphical Application requires a change to the Rectangle, what happens to the Computational Geometry Application?

SINGLE RESPONSIBILITY PRINCIPLE (SRP)

- Corrected Example:



SINGLE RESPONSIBILITY PRINCIPLE (SRP)

- Responsibility = "a reason for change"; gather together the items that should change for the same reason

- Examples:
  - Employee class that calculates and reports pay
  - Class responsible for data manipulation and data persistence

OPEN-CLOSED PRINCIPLE (OCP)

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

- One change should not result in a cascade of changes - bad design

- OCP says that modules should *never change*

  - Instead, add new code for new behaviors, don't change existing, working behavior

OPEN-CLOSED PRINCIPLE (OCP)

1. Open for Extension

How is this possible?

- Behavior of the module can be extended, module can behave in new ways

2. Closed for Modification

- No one is allowed to make source code *changes*

OPEN-CLOSED PRINCIPLE (OCP)

- Abstraction!!!
- Shape class

How do I draw triangles?

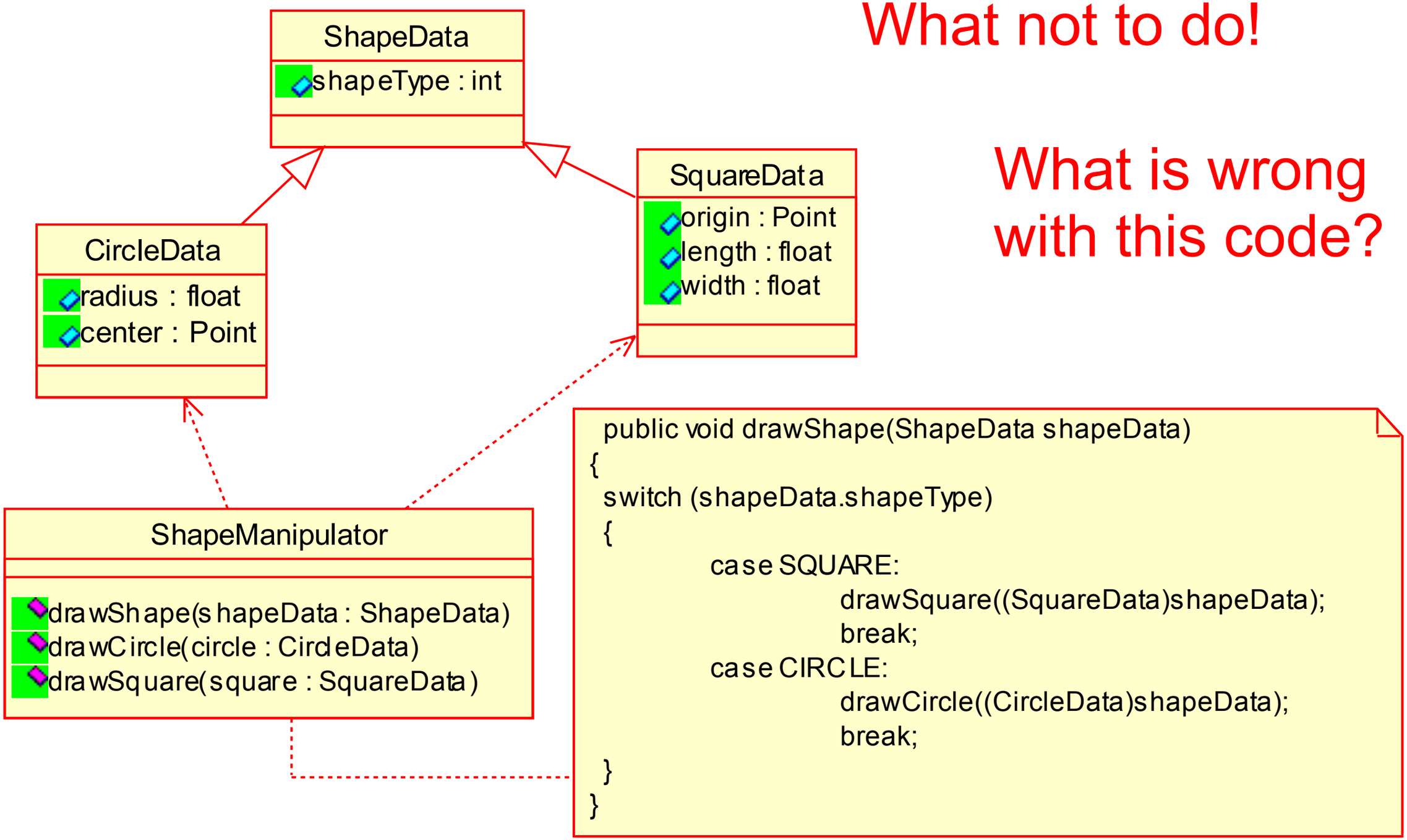
Listing 1 (Continued)
Procedural Solution to the Square/Circle Problem

```
struct Square
{
  ShapeType itsType;
  double itsSide;
  Point itsTopLeft;
};

//
// These functions are implemented elsewhere
//
void DrawSquare(struct Square*)
void DrawCircle(struct Circle*);

typedef struct Shape *ShapePointer;

void DrawAllShapes(ShapePointer list[], int n)
{
  int i;
  for (i=0; i<n; i++)
  {
    struct Shape* s = list[i];
    switch (s->itsType)
    {
    case square:
      DrawSquare((struct Square*)s);
    break;

    case circle:
      DrawCircle((struct Circle*)s);
    break;
    }
  }
```

The Open/Closed Principle (OCP) Example

An Example of
What not to do!

What is wrong
with this code?



The Open/Closed Principle (OCP) Example

- The Problem: Changeability…
  - If I need to create a new shape, such as a Triangle, I must modify the ‘drawShape()' function.
  - In a complex application the switch/case statement above is repeated over and over again for every kind of operation that can be performed on a shape
  - Worse, every module that contains such a switch/case statement retains a dependency upon every possible shape that can be drawn, thus, whenever 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 PRINCIPLE (OCP)

- Create an Abstract Shape class, each with virtual draw methods

- The drawAllShapes method would now never have to be changed

- New shapes would be new extensions of the abstract Shape class

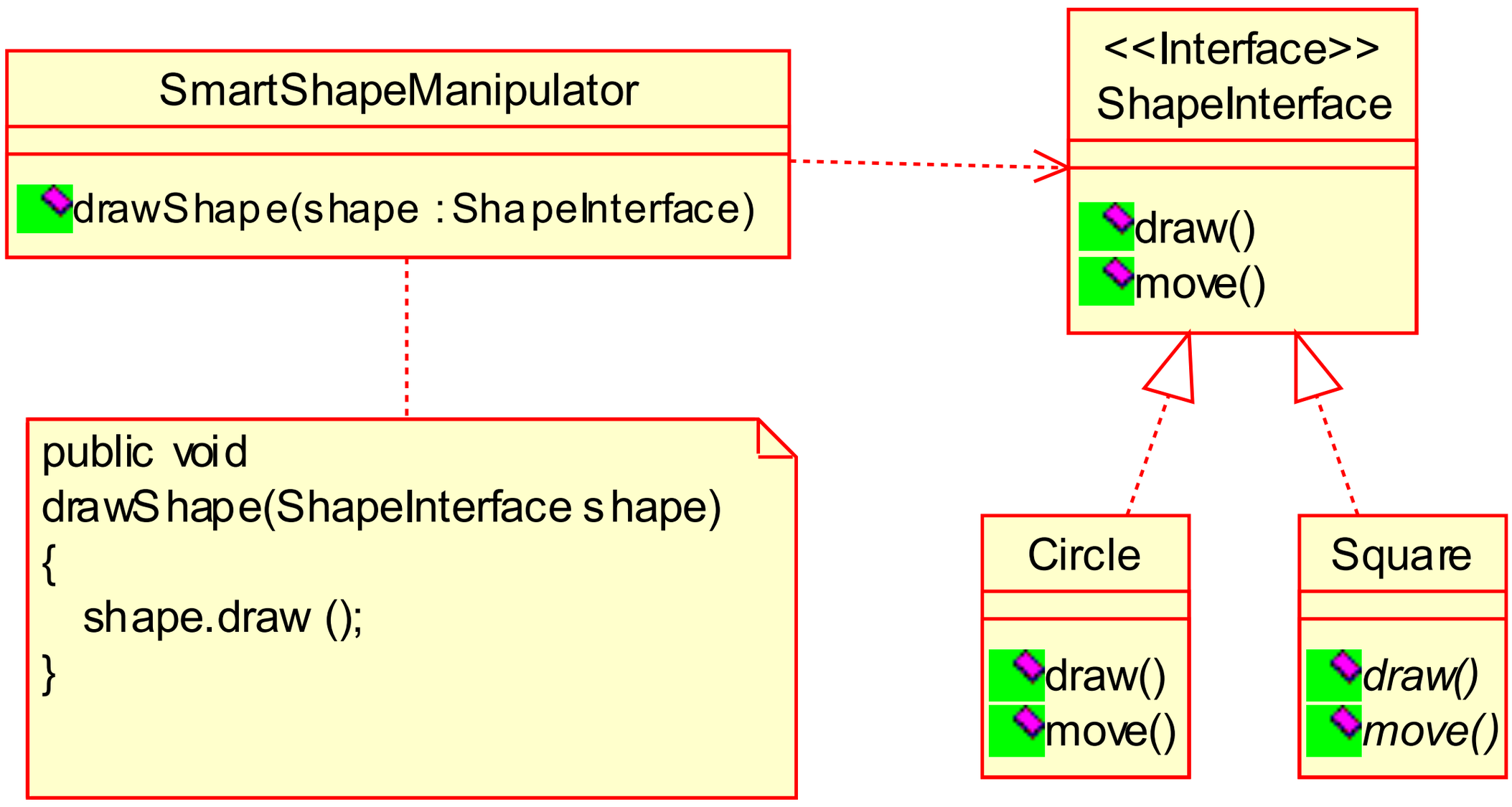
OPEN-CLOSED PRINCIPLE (OCP)

Listing 2

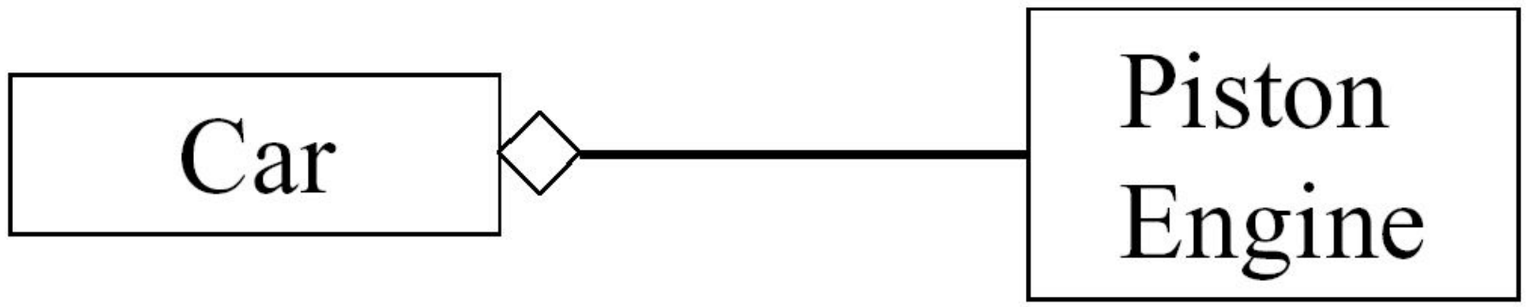
OOD solution to Square/Circle problem.

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

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

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

void DrawAllShapes(Set<Shape*>& list)
{
  for (Iterator<Shape*>i(list); i; i++)
     (*i)->Draw();
}
```

The Open/Closed Principle (OCP) Example

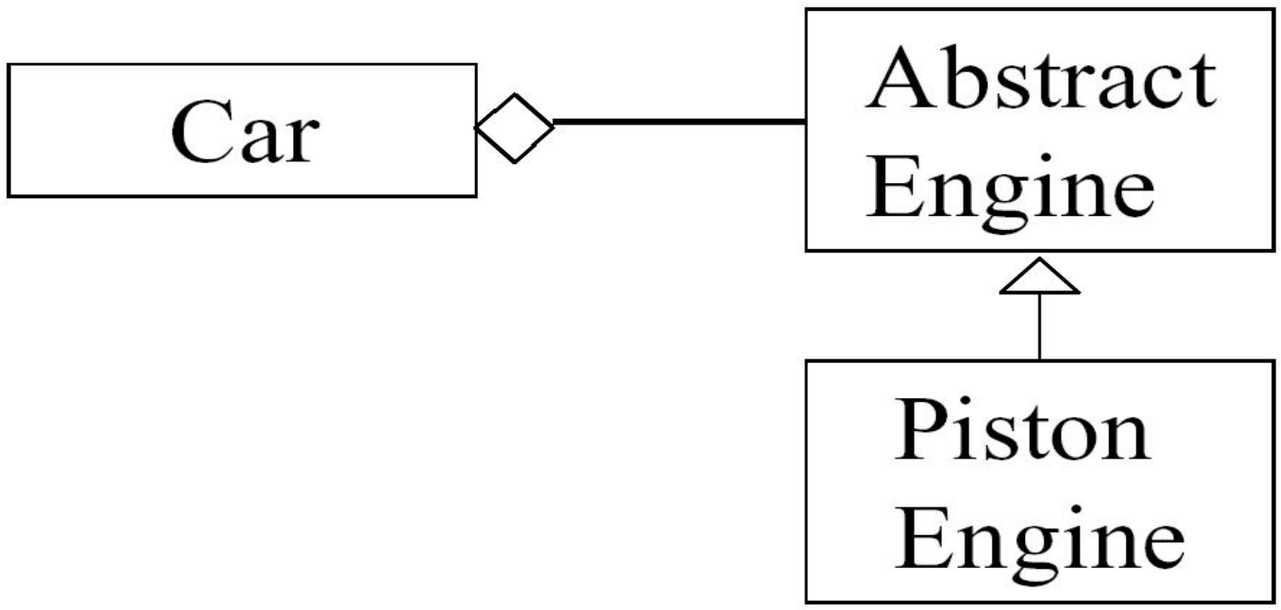


Open the door ...



- How to make the Car run efficiently with a TurboEngine?
- Only by changing the Car!
  - ...in the given design

... But Keep It Closed!



- A class must not depend on a concrete class!
- It must depend on an abstract class ...
- ...using polymorphic dependencies (calls)

OPEN-CLOSED PRINCIPLE (OCP)

- 100% closure is unreasonable
- Closure should be strategic
- Abstraction
- Data Driven Approach

OPEN-CLOSED PRINCIPLE (OCP)

Listing 3

Shape with ordering methods.

```
class Shape
{
  public:
    virtual void Draw() const = 0;
    virtual bool Precedes(const Shape&) const = 0;

    bool operator<(const Shape& s) {return Precedes(s);}
};
```

Listing 4

DrawAllShapes with Ordering

```
void DrawAllShapes(Set<Shape*>& list)
{
    // copy elements into OrderedSet and then sort.
    OrderedSet<Shape*> orderedList = list;
    orderedList.Sort();

    for (Iterator<Shape*> i(orderedList); i; i++)
        (*i)->Draw();
}
```

Listing 5

Ordering a Circle

```
bool Circle::Precedes(const Shape& s) const
{
    if (dynamic_cast<Square*>(s))
        return true;
    else
        return false;
}
```

Listing 6

Table driven type ordering mechanism

```
#include <typeinfo.h>
#include <string.h>
enum {false, true};
typedef int bool;

class Shape
{
  public:
    virtual void Draw() const = 0;
    virtual bool Precedes(const Shape&) const;

    bool operator<(const Shape& s) const
    {return Precedes(s);}
  private:
    static char* typeOrderTable[];
};

char* Shape::typeOrderTable[] =
{
    "Circle",
    "Square",
    0
};

// This function searches a table for the class names.
// The table defines the order in which the
// shapes are to be drawn. Shapes that are not
// found always precede shapes that are found.
//
bool Shape::Precedes(const Shape& s) const
{
    const char* thisType = typeid(*this).name();
    const char* argType = typeid(s).name();
    bool done = false;
    int thisOrd = -1;
    int argOrd = -1;
    for (int i=0; !done; i++)
    {
        const char* tableEntry = typeOrderTable[i];
        if (tableEntry != 0)
        {
            if (strcmp(tableEntry, thisType) == 0)
                thisOrd = i;
            if (strcmp(tableEntry, argType) == 0)
                argOrd = i;
```

Listing 6 (Continued)

Table driven type ordering mechanism

```
            if ((argOrd > 0) && (thisOrd > 0))
                done = true;
        }
        else // table entry == 0
            done = true;
    }
    return thisOrd < argOrd;
}
```

Table is the only item not closed. This can be placed in a separate module so that changes to it do not affect the other modules.

OPEN-CLOSED PRINCIPLE (OCP)

- Encapsulation follows the OCP
- No Global variables
  - No module that depends upon a global variable can be closed against any other module that may write to that variable
- Run time type identification (RTTI) - dynamic casting is not recommended unless it doesn't violate OCP

ACTIVITY

- Want to draw vehicles: cars, trucks, etc.
- Draw a class diagram of how to represent these groups that does not violate the OCP
- Write the Java code to implement this (note: you do not have to write the explicit draw code for the individual cars/trucks…)

Liskov Substitution Principle (LSP)

- The key of OCP: Abstraction and Polymorphism
  - Implemented by inheritance
  - How do we measure the quality of inheritance?

Inheritance should ensure that any property proved about supertype objects also holds for subtype objects

B. Liskov, 1987

Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.

R. Martin, 1996

LISKOV SUBSTITUTION PRINCIPLE (LSP)

- If for each object o_1 of type S, there is an object o_2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2, then S is a subtype of T.

- Basically…Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it

LISKOV SUBSTITUTION PRINCIPLE (LSP)

- Consider the consequences
  - If a function doesn't conform to the LSP, then the function uses a reference to a base class requiring the function to know about all derivatives of the base class
- Violates OCP

LISKOV SUBSTITUTION PRINCIPLE (LSP)

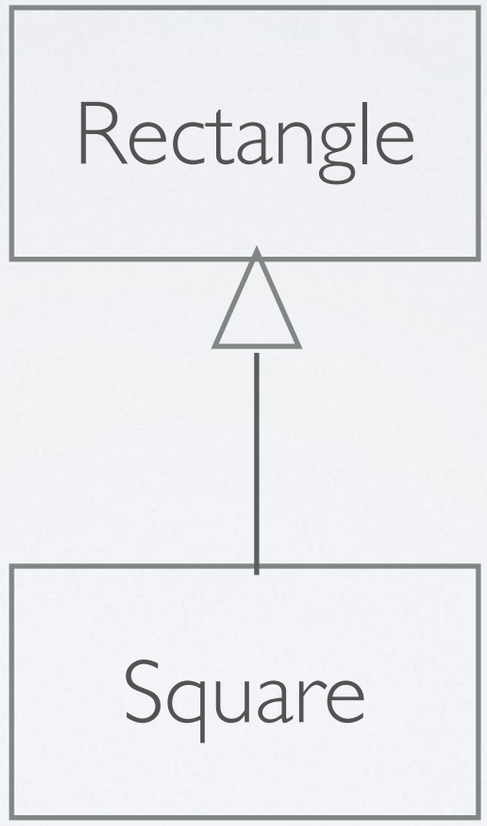
```
void DrawShape(const Shape& s)
{
    if(typeid(s) == typeid(Square))
        DrawSquare(static_cast<Square&>(s));
    else if(typeid(s) == typeid(Circle))
        DrawCircle(static_cast<Circle&>(s));
}
```

LISKOV SUBSTITUTION PRINCIPLE (LSP)

```
class Rectangle
{
   public:
      void SetWidth(double w) {itsWidth=w;}
      void SetHeight(double h) {itsHeight=h;}
      double GetHeight() const {return itsHeight;}
      double GetWidth() const {return itsWidth;}
   private:
      double itsWidth;
      double itsHeight;
}
```

LISKOV SUBSTITUTION PRINCIPLE (LSP)

- What if I want a square?



Issue?

LISKOV SUBSTITUTION PRINCIPLE (LSP)

```
void Square::SetWidth(double w)
{
   Rectangle::SetWidth(w);
   Rectangle::SetHeight(w);
}
void Square::SetHeight(double h)
{
   Rectangle::SetHeight(h);
   Rectangle::SetWidth(h);
}
```

Problem Solved?

LISKOV SUBSTITUTION PRINCIPLE (LSP)

```
void f(Rectangle& r)
{
    r.SetWidth(32);   //calls Rectangle::SetWidth
}
```

- Rectangle's methods weren't defined as Virtual
- Editing them now violates OCP

LISKOV SUBSTITUTION PRINCIPLE (LSP)

```
void g(Rectangle r)
{
    r.SetWidth(5);
    r.SetHeight(4);
    assert(r.GetWidth() * r.GetHeight() == 20);
}
```

LISKOV SUBSTITUTION PRINCIPLE (LSP)

- Must view design from perspective of users of the design

- Cannot view/evaluate in isolation

- Square and Rectangle worked fine in isolation but not from perspective of a programmer making assumptions about the base class

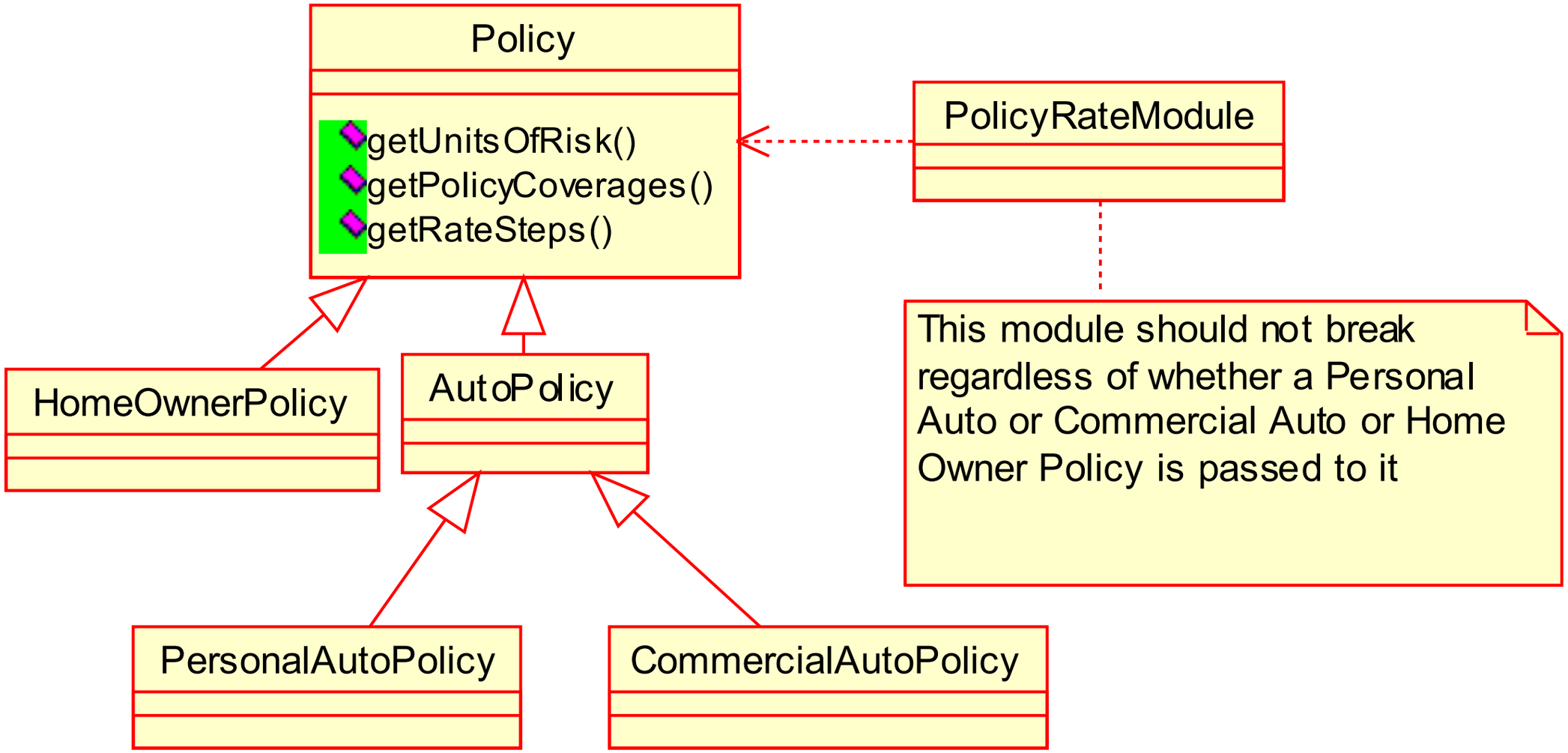
LISKOV SUBSTITUTION PRINCIPLE (LSP)

- What was the real problem with the Rectangle and Square design?
- Did the ISA relationship not hold?

LISKOV SUBSTITUTION PRINCIPLE (LSP)

- Behaviorally, a square is not a rectangle
- When we program, it is behavior that we are seeking to implement
- In order for LSP to hold, and thus the OCP, all derivatives must conform to the *behavior* of the base class they use

The Liskov Substitution Principle (LSP) Example



Inheritance *Appears* Simple

```
class Bird {                          // has beak, wings,...
  public: virtual void fly();  // Bird can fly
};

class Parrot : public Bird {       // Parrot is a bird
  public: virtual void mimic();  // Can Repeat words...
};

// ...
Parrot mypet;
mypet.mimic();    // my pet being a parrot can Mimic()
mypet.fly();      // my pet "is-a" bird, can fly
```

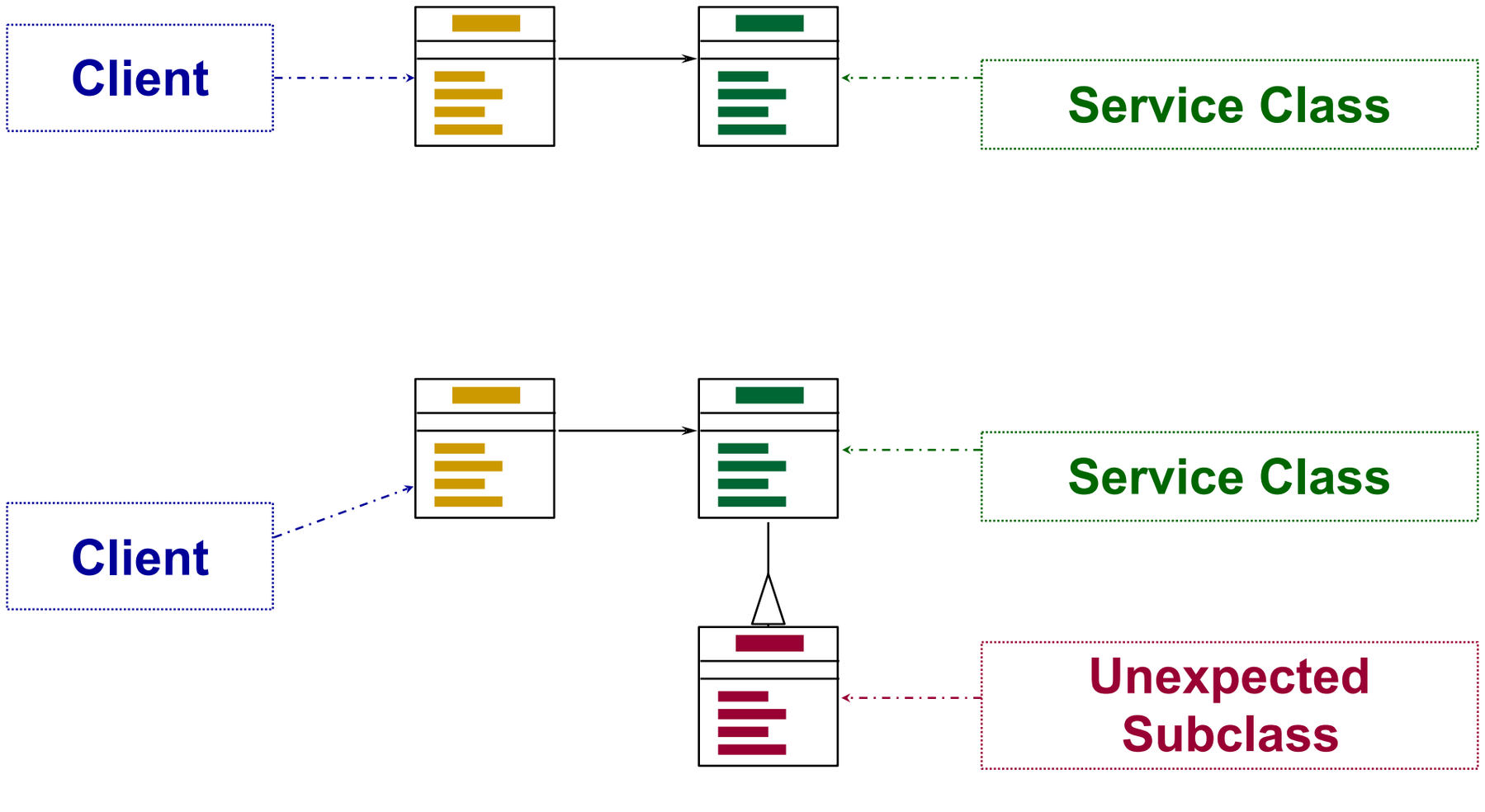
Penguins Fail to Fly!

```
class Penguin : public Bird {
   public: void fly() {
      error ("Penguins don't fly!"); }
};
void PlayWithBird (Bird& abird) {
     abird.fly();     // OK if Parrot.
     // if bird happens to be Penguin...OOOPS!!
}
```

- Does not model: "*Penguins can't fly*"
- It models "*Penguins may fly, but if they try it is an error*"
- Run-time error if attempt to fly → not desirable
- ***Think about Substitutability - Fails LSP***

LSP and Replaceability

- Any code which can legally call another class's methods
  - must be able to substitute any subclass of that class without modification:



LSP Related Heuristic

It is illegal for a derived class to override a base-class method with a NOP method

- NOP = a method that does nothing, “no operation”
- Solution: Extract Common Base-Class
  - if both initial and derived classes have different behaviors
  - for **`Penguins`** →
    - **`Birds`**, **`FlyingBirds`**, **`Penguins`**

Dependency Inversion Principle

I. High-level modules should ***not*** depend on low-level module implementations. Both levels should depend on abstractions.

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

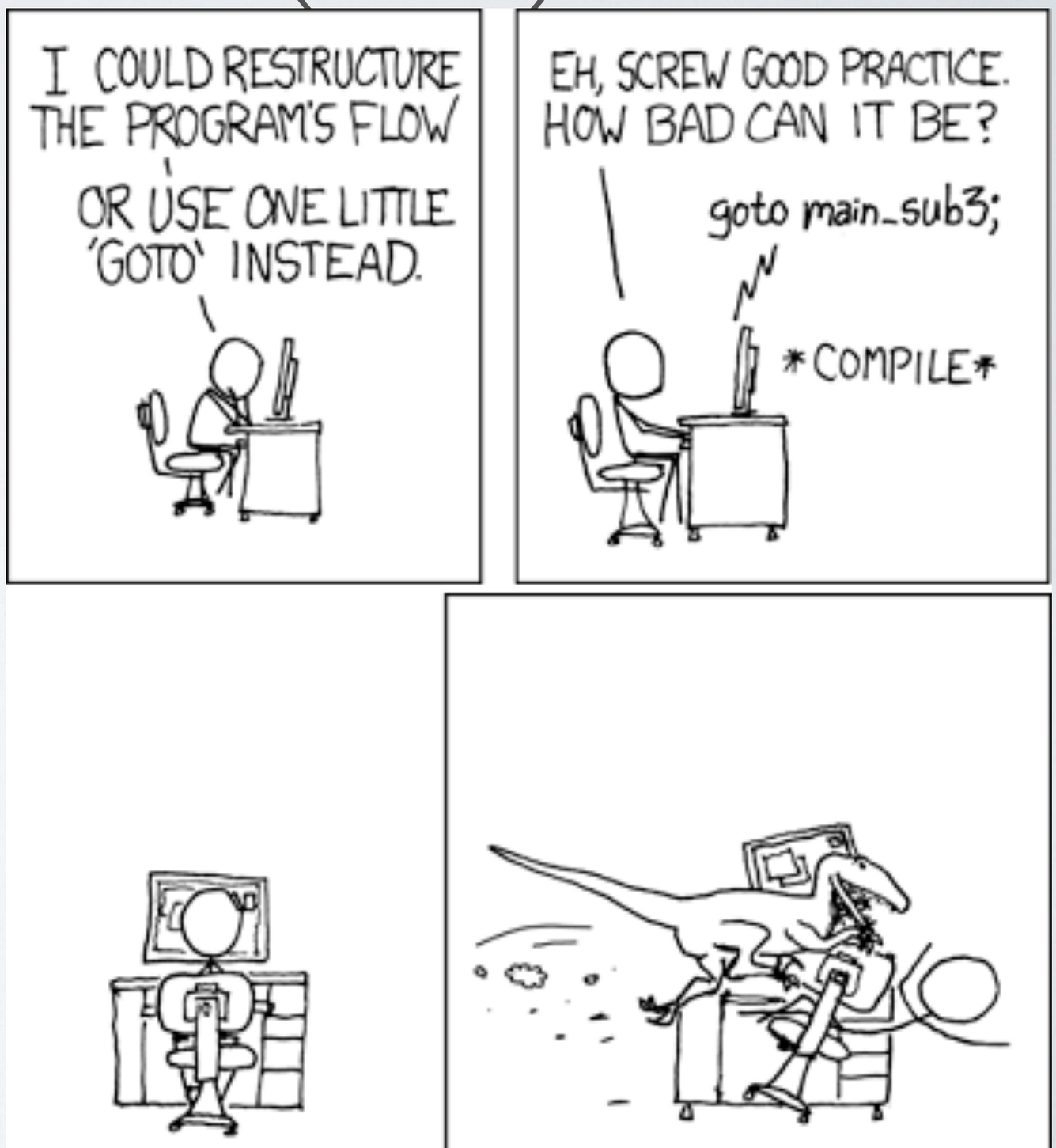
R. Martin, 1996

- OCP states the goal; DIP states the mechanism
- A base class in an inheritance hierarchy should not know any of its subclasses
- Modules with detailed implementations are not depended upon, but depend themselves upon higher abstractions

Specification inheritance vs. implementation inheritance

DEPENDENCY INVERSION PRINCIPLE (DIP)

- How do we define "bad design"?
- TNTWIWHDI criterion - "That's not the way I would have done it"



DEPENDENCY INVERSION PRINCIPLE (DIP)

- Another definition is if the software behaves the way it is supposed to (meets requirements) but fulfills any or all of the following three traits:

1. It is hard to change because every change affects too many other parts of the system (rigidity)

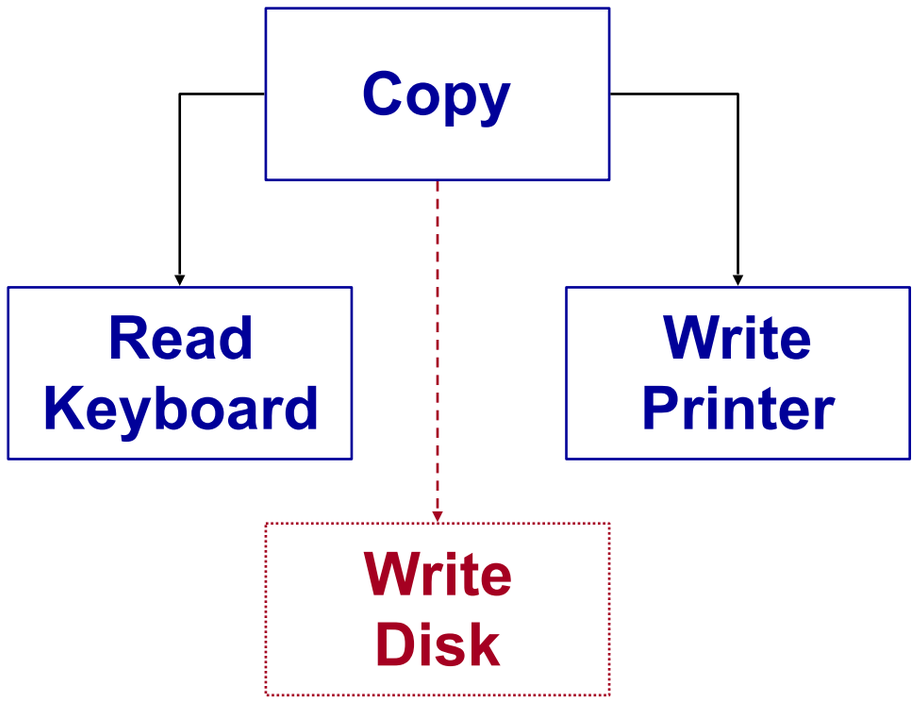
2. When you make a change, unexpected parts of the system break (fragility)

3. It is hard to reuse in another application because it cannot be disentangled from the current application (immobility)

Dependency Inversion Principle

- Dependency Inversion is the strategy of depending upon interfaces or abstract functions and classes, rather than upon concrete functions and classes.
- Every dependency in the design should target an interface, or an abstract class. No dependency should target a concrete class.

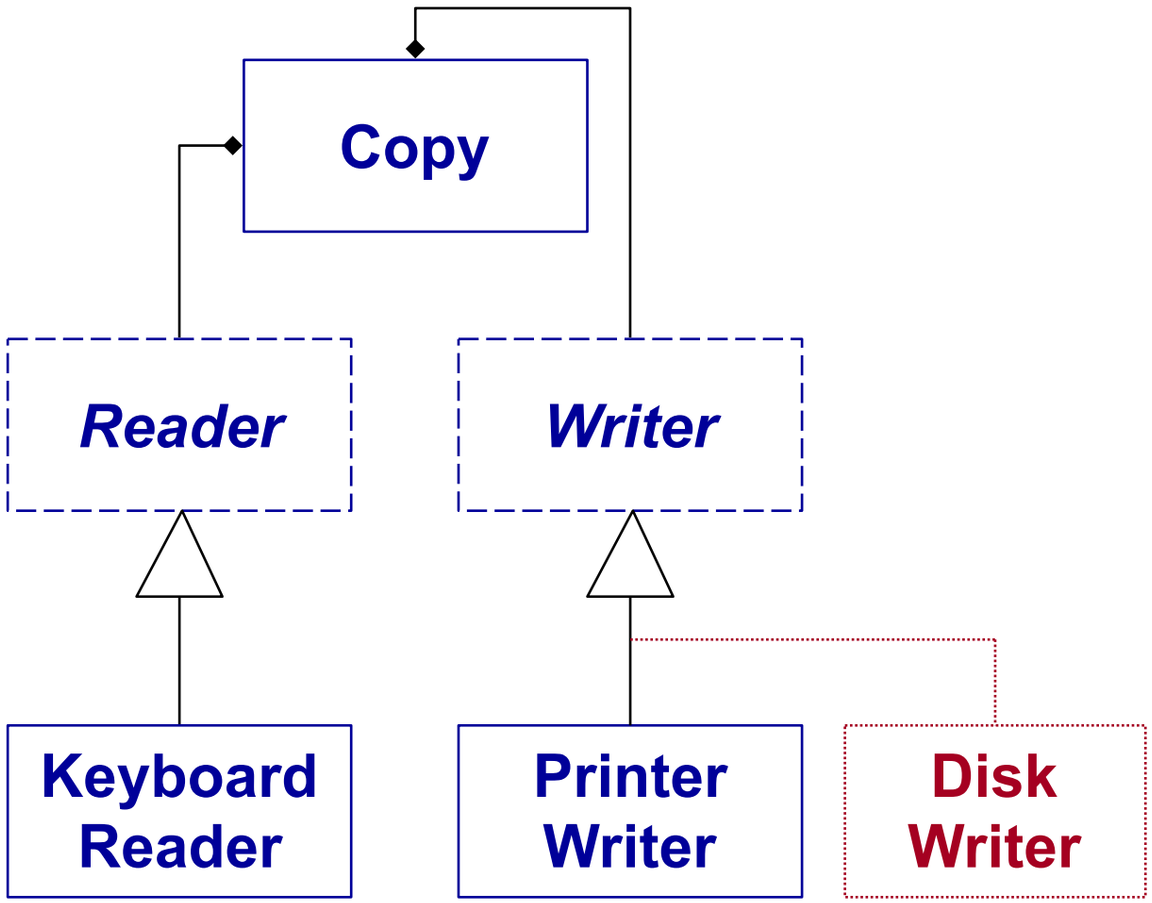
Example of Rigidity and Immobility




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

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

DIP Applied on Example




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

class Writer {
  public:
    virtual void write(int)=0;
};

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

DEPENDENCY INVERSION PRINCIPLE (DIP)

A. High level modules should not depend on low level modules. Both should depend upon abstractions.

B. Abstractions should not depend upon details. Details should depend upon abstractions.

Design to an Interface

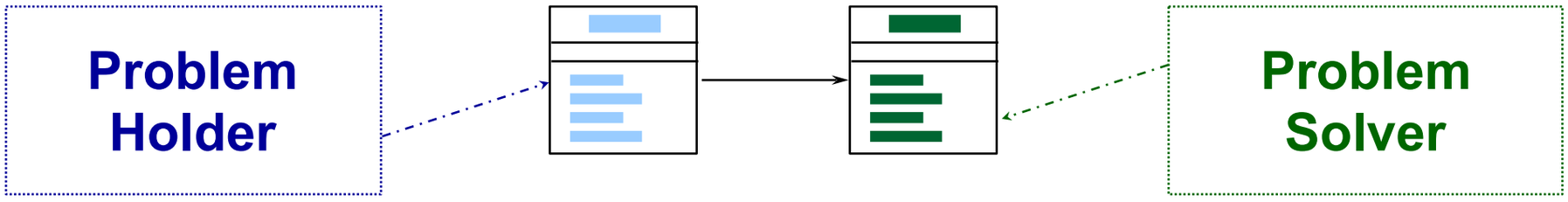
- **Abstract classes/interfaces:**
  - tend to change much less frequently
  - abstractions are 'hinge points' where it is easier to extend/ modify
  - shouldn't have to modify classes/interfaces that represent the abstraction (OCP)

- Exceptions
  - Some classes are very unlikely to change;
    - Therefore, little benefit to inserting abstraction layer
    - Example: String class
  - In cases like this you can use concrete class directly
    - as in Java or C++

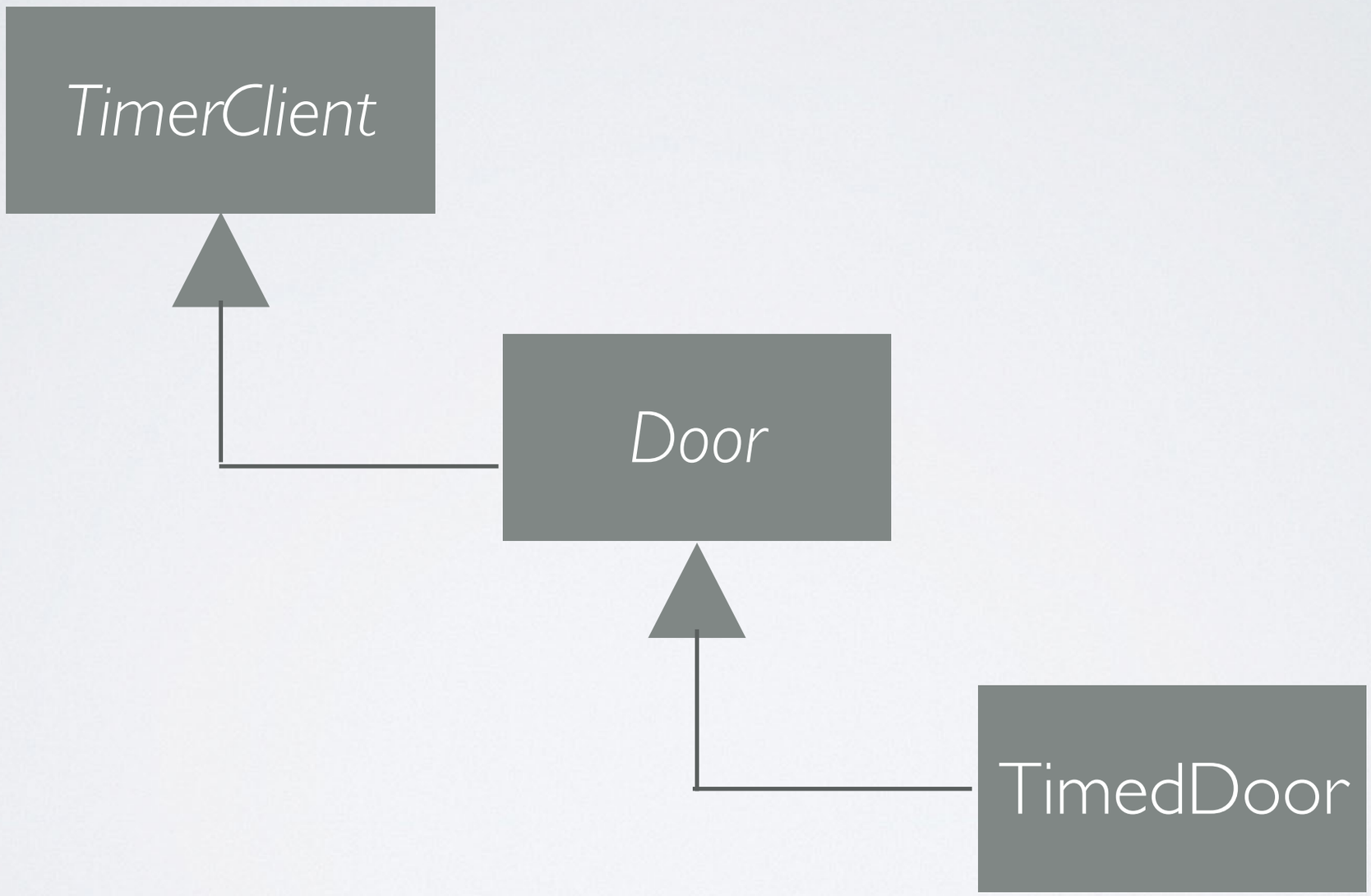
DIP - Related Heuristic

When in doubt, add a level of indirection

- If you cannot find a satisfactory solution for the class you are designing, try delegating responsibility to one or more classes:



INTERFACE SEGREGATION PRINCIPLE (ISP)



INTERFACE SEGREGATION PRINCIPLE (ISP)

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

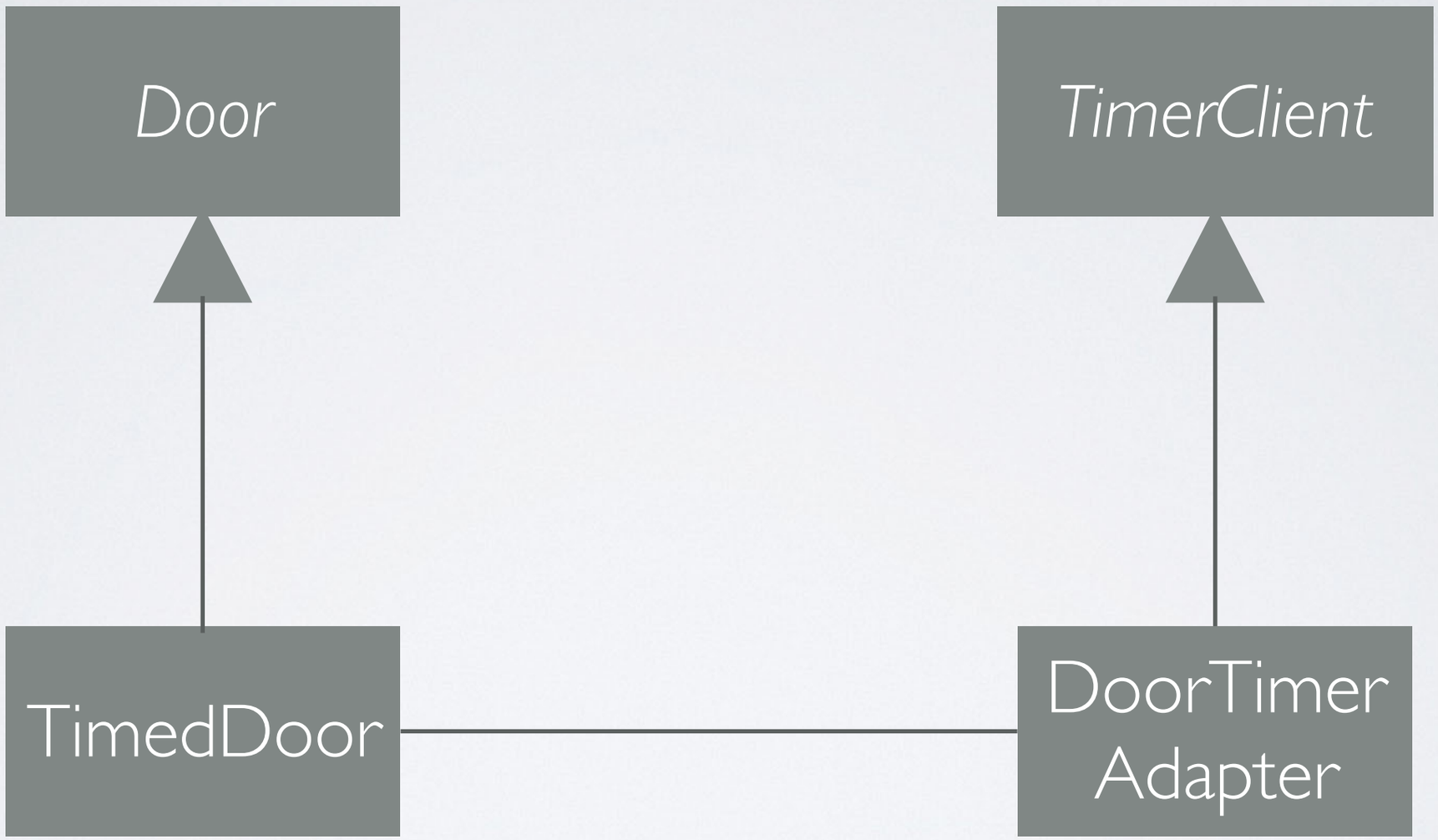
INTERFACE SEGREGATION PRINCIPLE (ISP)

- If clients are forced to depend upon interfaces they don't use, those clients are subject to changes to those interfaces
- Results in inadvertent coupling between all clients

INTERFACE SEGREGATION PRINCIPLE (ISP)

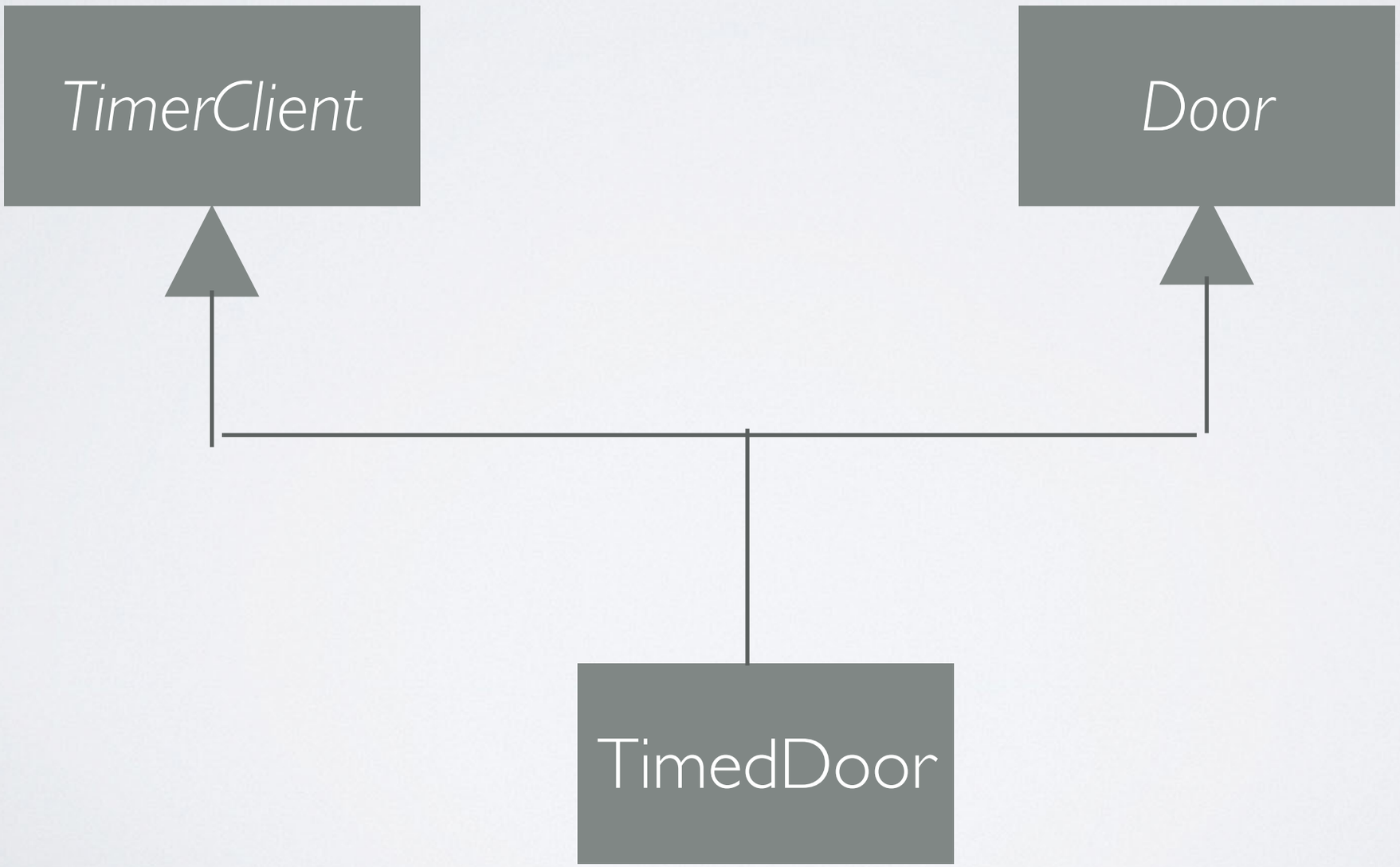
- Separation through delegation
- Adapter pattern

INTERFACE SEGREGATION PRINCIPLE (ISP)



INTERFACE SEGREGATION PRINCIPLE (ISP)

- Separation through multiple inheritance



INTERFACE SEGREGATION PRINCIPLE (ISP)

- Activity

1. ATM UI….need UIs for Braille, Screen, and Speech

2. UI consists of transactions: Deposit, Withdrawal, and Transfer

- Write the class diagrams for these two scenarios

The Founding Principles

- The three principles are closely related
- Violating either LSP or DIP invariably results in violating OCP
  - LSP violations are latent violations of OCP
- It is important to keep in mind these principles to get most out of OO development...
- ... and go beyond buzzwords and hype ;)