CSCI 152
Software Engineering

Shih-Hsi "Alex" Liu

SOLID Principles for OO Development (3 papers from Martin)

SE Principles for OO Development

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

Open-Closed Principle (OCP)

Software entities should be open for extension,
but closed for modification

B. Meyer, 1988 / quoted by **R. Martin**, 1996

- "*Software Systems change during their life time*"
  - When a single change to a program results in a cascade of changes to dependent modules, that program exhibits the undesirable attributes that we have come to associate with "bad" design
  - both better designs and poor designs have to face the changes;
  - good designs are stable

Open-Closed Principle (OCP)

All systems change during their life cycle. This must be borne in mind when developing systems expected to last longer than the first version
but closed for modification

Ivar Jacobson

- You should design modules that *never changes*. When requirements change, you extend the ***behavior*** of such modules by adding new code, **not** by changing old code that already works.

- Be open for extension
  - module's behavior can be *extended*
    - *Make the module behave in new and different ways as the req. of the app. change, or to meet the needs of new app.*
- Be closed for modification
  - source code for the module *must not be changed*
    - *Do not change the working modules*

The Open-Closed Principle (OCP)

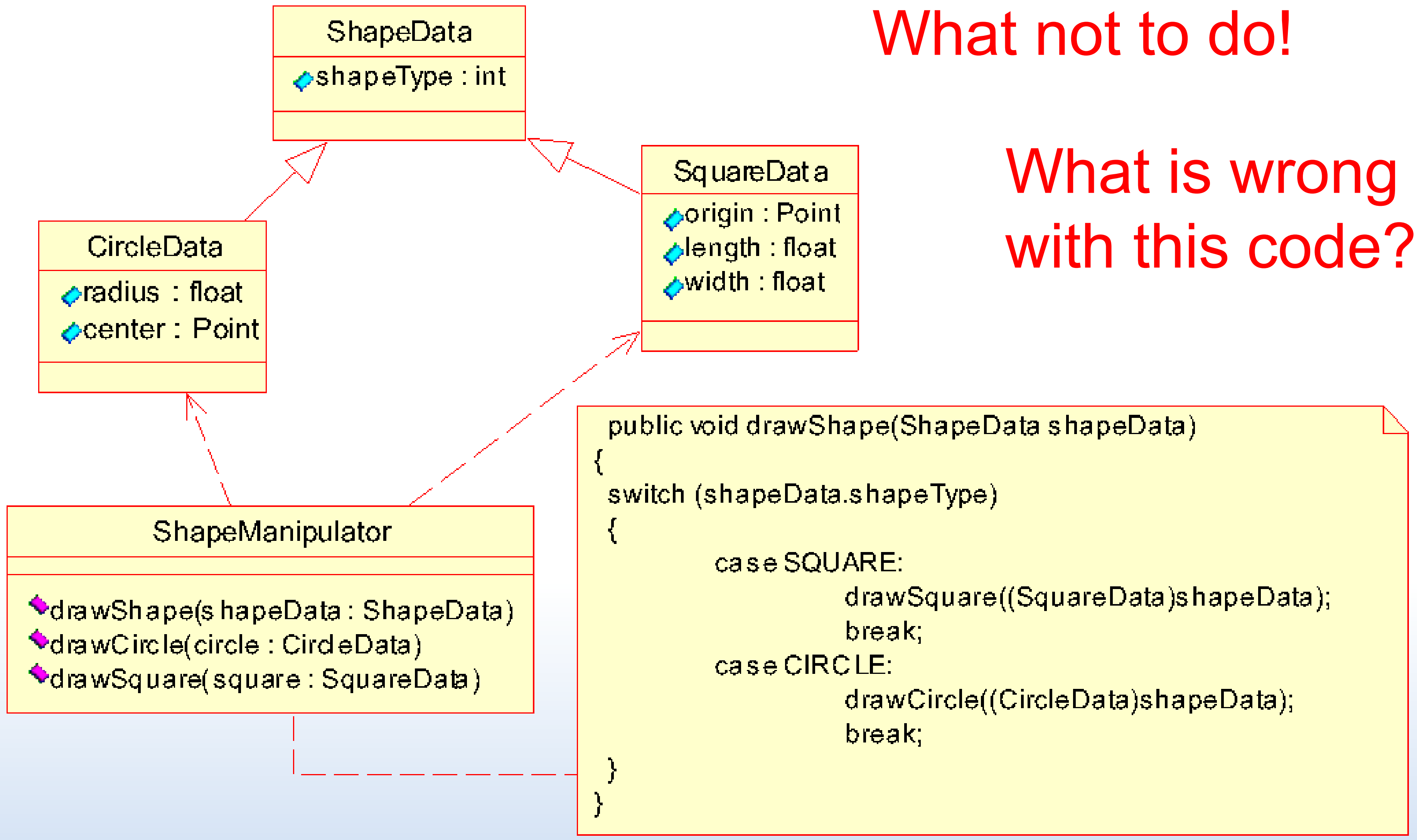
- *Modules should be written so they can be extended without requiring them to be modified*
- We want to change what the modules do, without changing the source code of the modules.
  - When requirements change, we extend the behavior of such modules by **adding new** code, **not** by **changing old** code that already works

- Why is it bad to change source code?
- How is this OCP implemented?

The Open/Closed Principle (OCP) Example

An Example of
What not to do!

What is wrong
with this code?

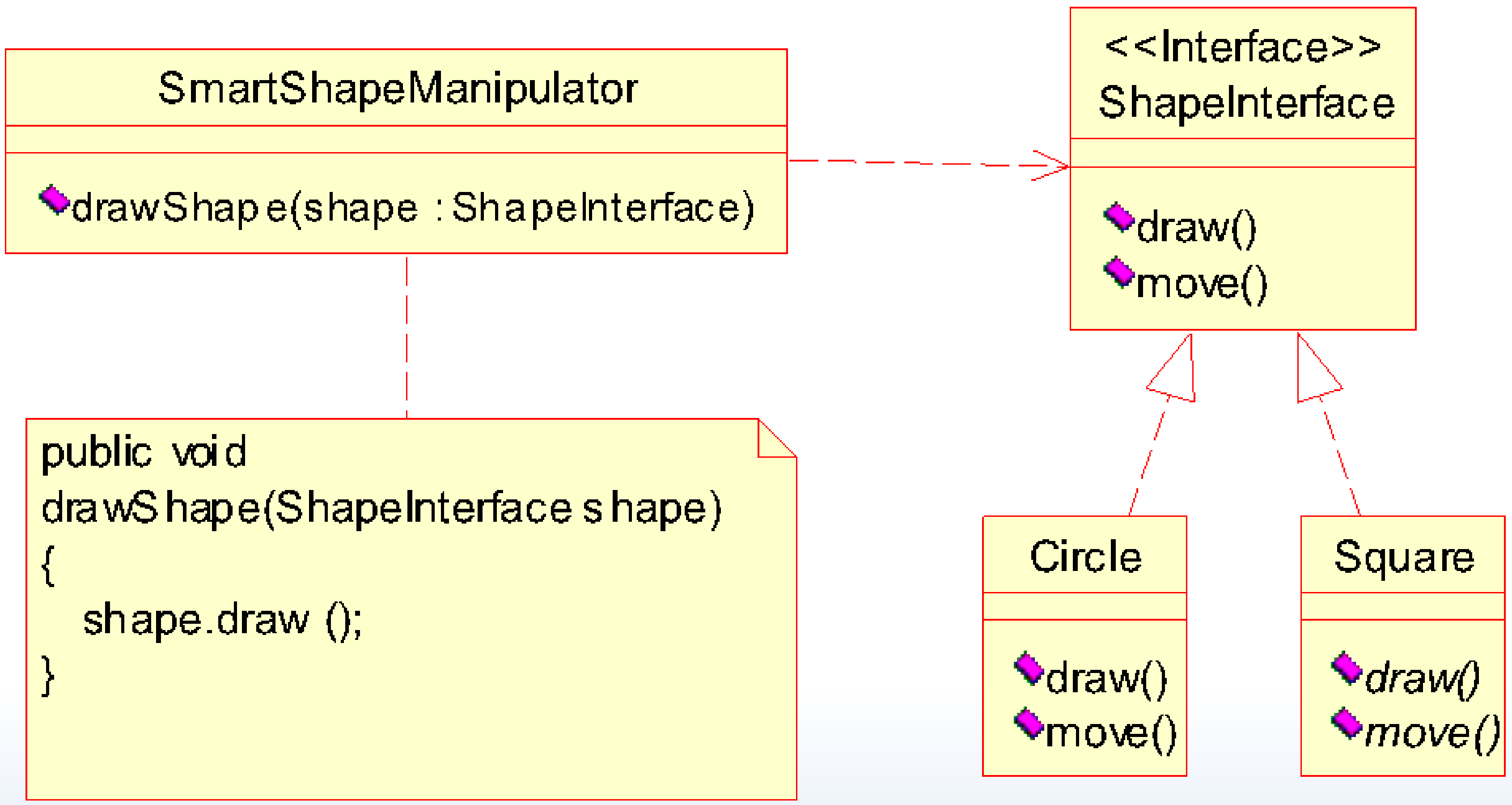


```
 public void drawShape(ShapeData shapeData)
{
 switch (shapeData.shapeType)
 {
         case SQUARE:
                  drawSquare((SquareData)shapeData);
                  break;
         case CIRCLE:
                  drawCircle((CircleData)shapeData);
                  break;
 }
}
```

The Open/Closed Principle (OCP) Example

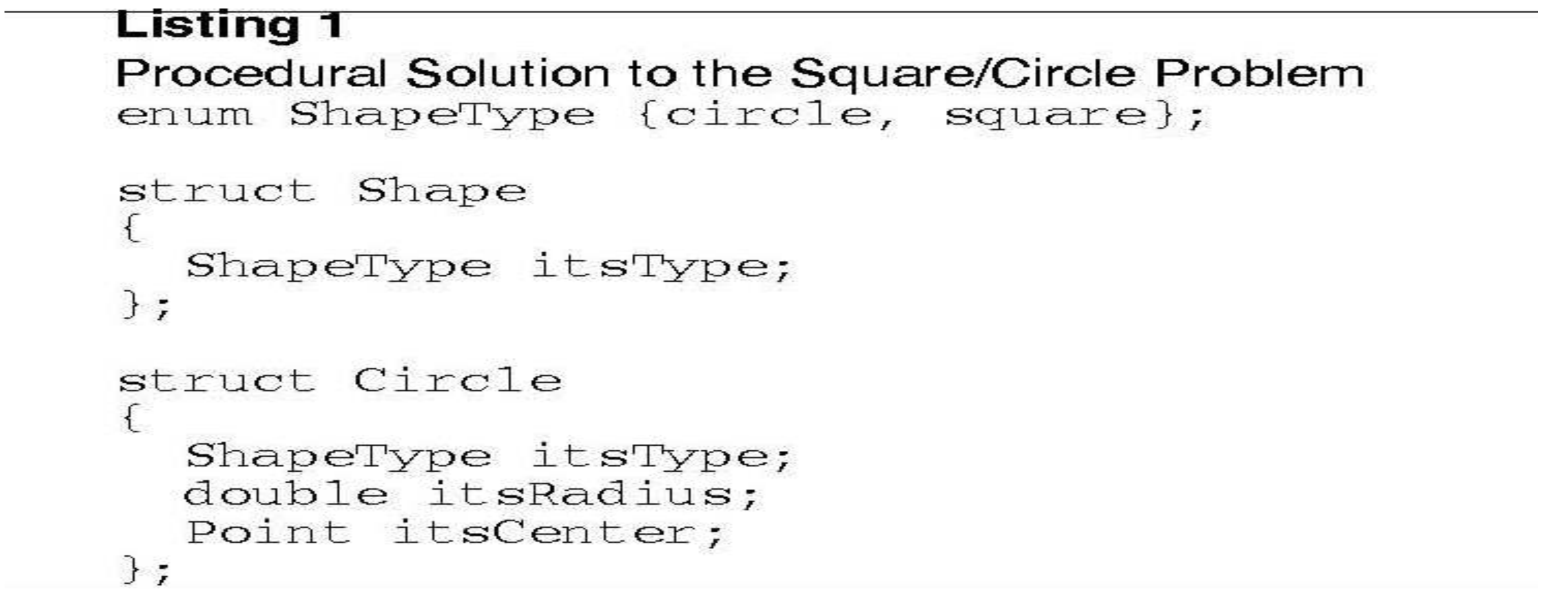
- The Problem: Changeability…
  - If I need to create a new shape, such as a Triangle, I must modify the ‘drawShape()' function.
  - In a complex application the switch/case statement above is repeated over and over again for every kind of operation that can be performed on a shape .
  - Worse, every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

The Open/Closed Principle (OCP) Example



```
public void
drawShape(ShapeInterface shape)
{
   shape.draw ();
}
```

A BAD example in C/C++



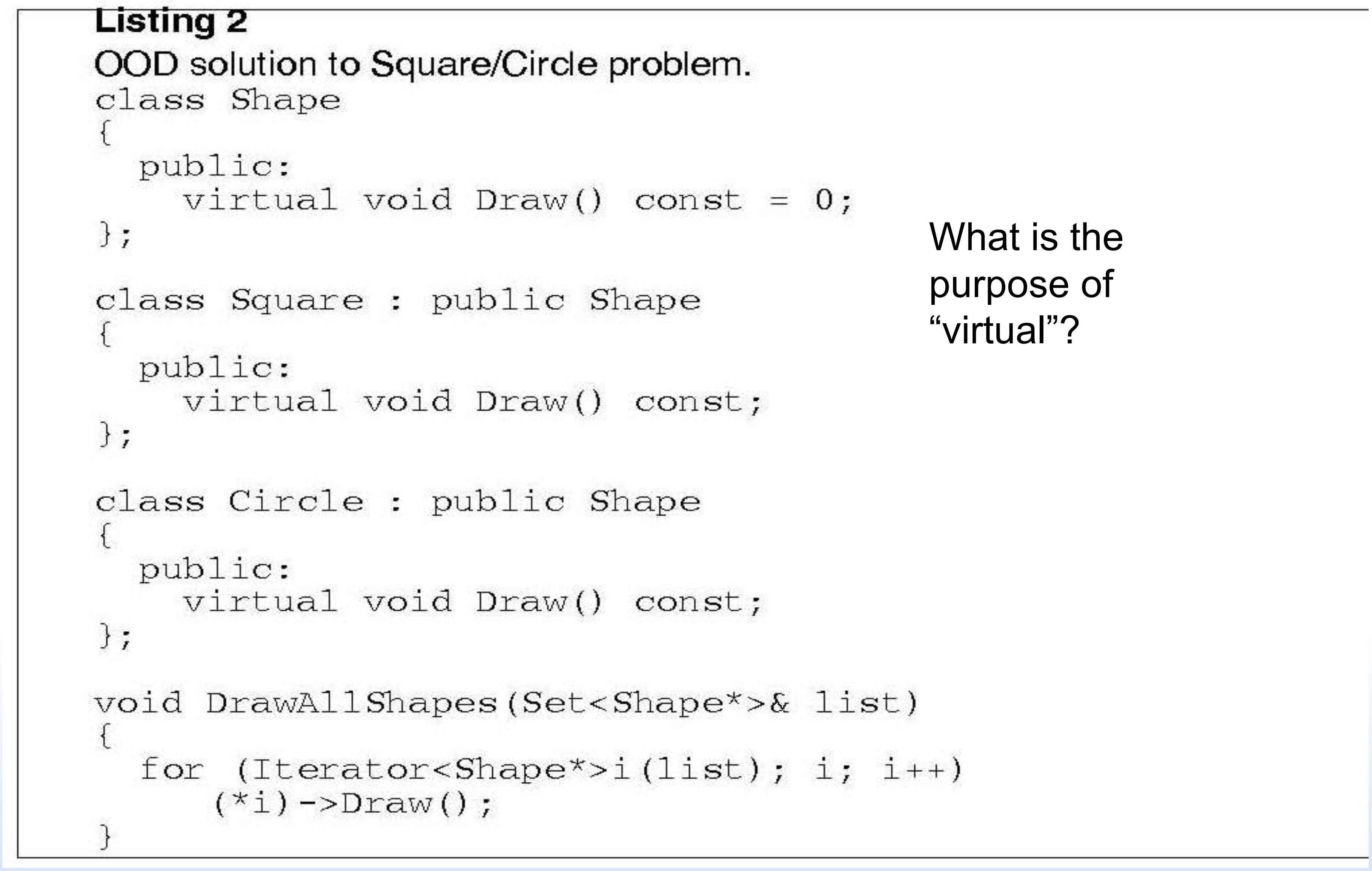
Listing 1

Procedural Solution to the Square/Circle Problem

```
enum ShapeType {circle, square};

struct Shape
{
  ShapeType itsType;
};

struct Circle
{
  ShapeType itsType;
  double itsRadius;
  Point itsCenter;
};
```

```
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;
    }
  }
}
```

A Good C/C++ example



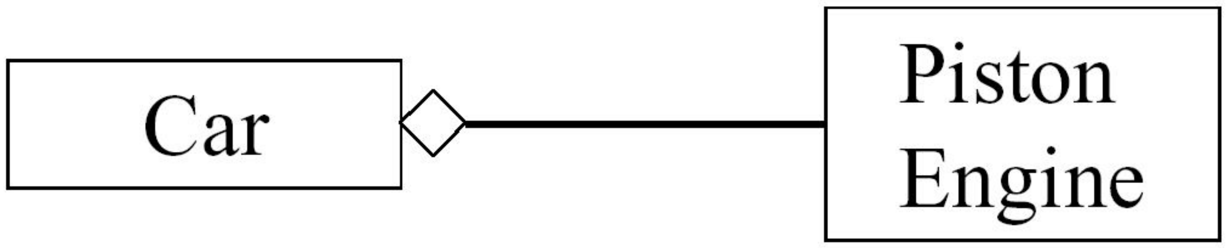
Listing 2

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();
}
```

What is the purpose of “virtual”?

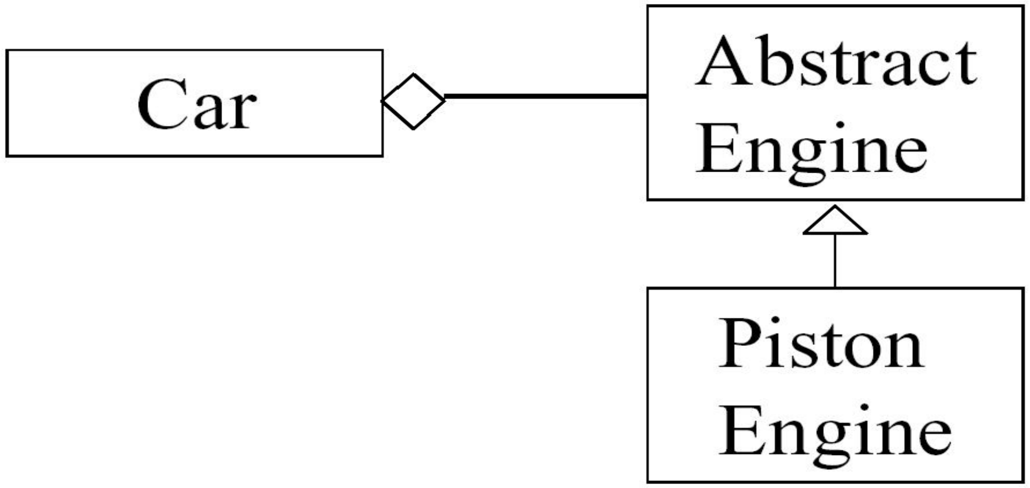
Open the door ...



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

  ...in the given design

... But Keep It Closed!



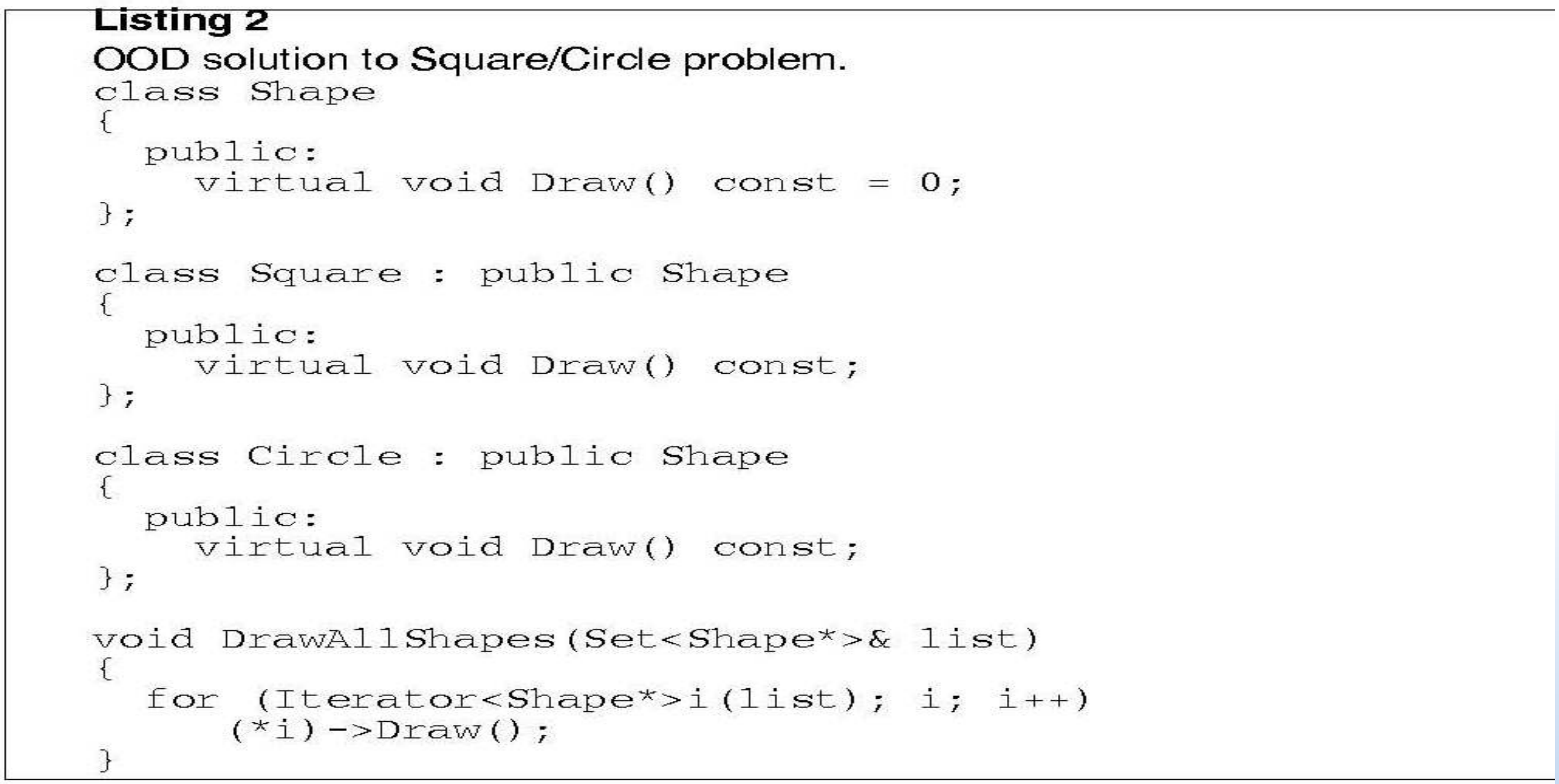
- **A class must *not* depend on a concrete class!**
- It must depend on an abstract class ...
  - Abstraction fixed but can be extended by derived classes
  - ...using polymorphic dependencies (calls)

Key Point

- Since programs that conform to OCP are changed by adding new code, rather than by changing existing code, they do not experience the cascade of changes exhibited by non-conforming programs.
- Abstraction is the key!!

Strategic Closure

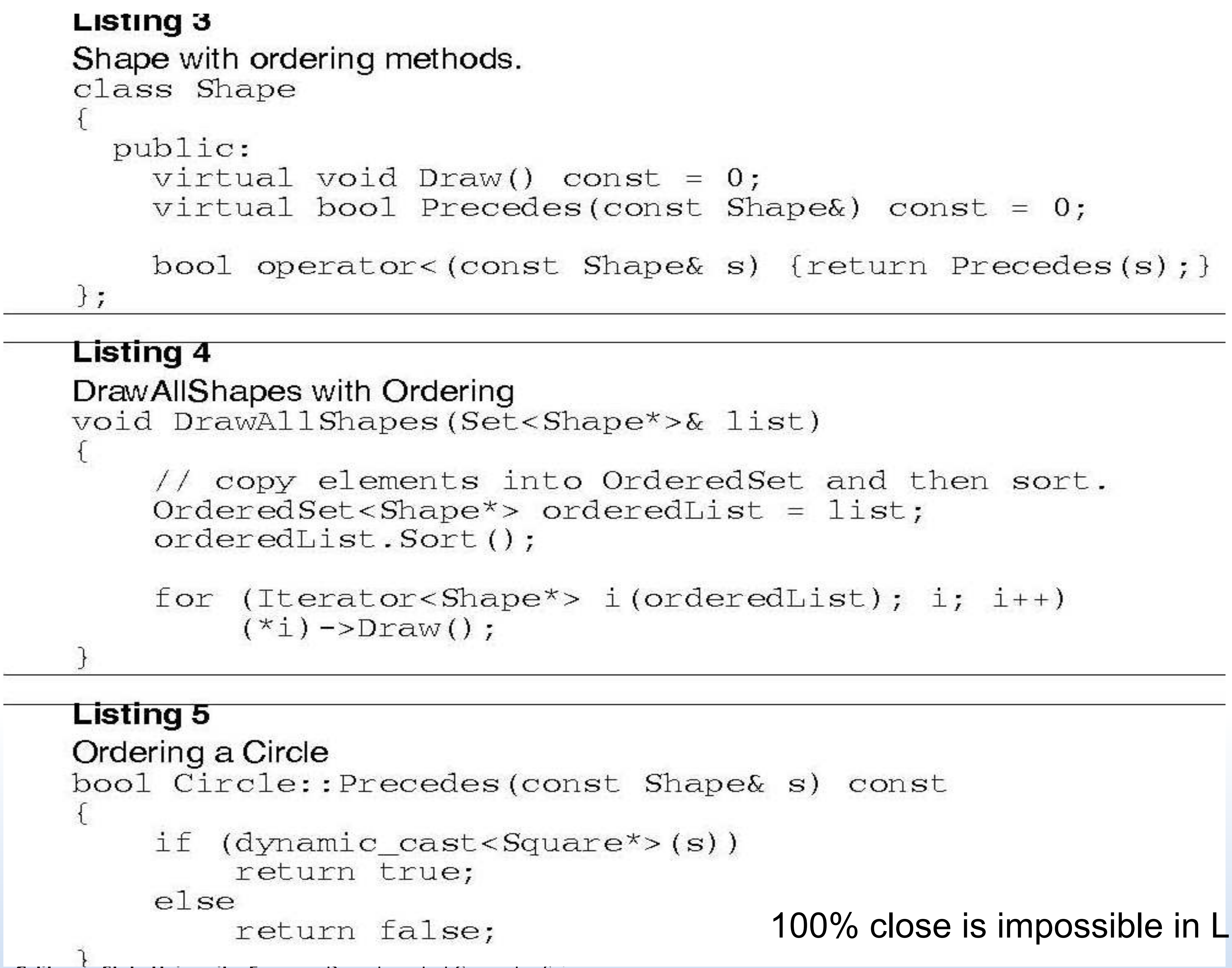
- No significant program can be 100% closed.
  - If all circle should be drawn before square, DrawAllShapes in Listing 2 should be changed
  - Strategic closure: designer must choose the kinds of changes against which to close his design



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();
}
```



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();
}
```

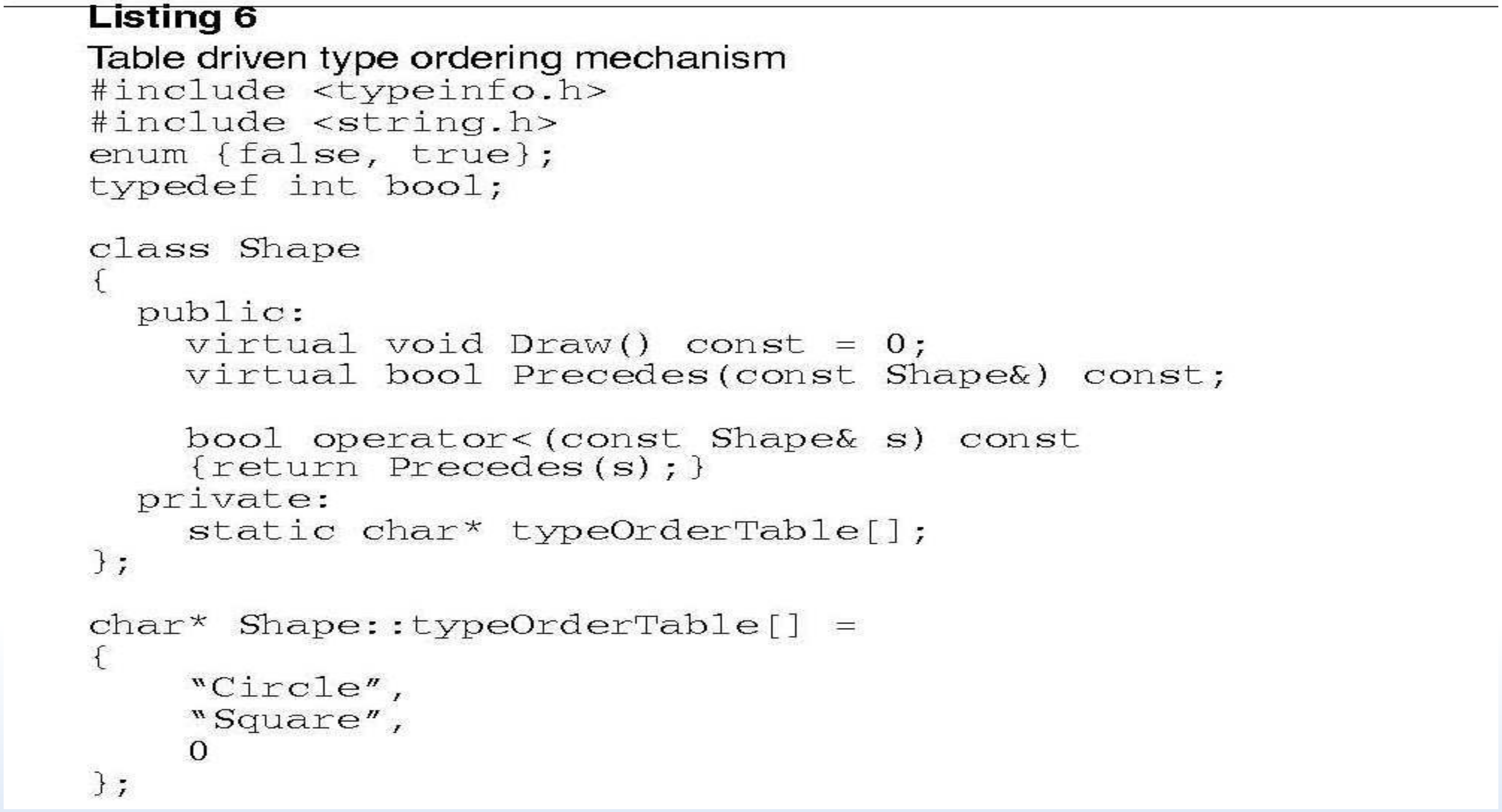
Listing 5

Ordering a Circle

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

100% close is impossible in Listing 5

Data Driven Approach for Strategic Closure

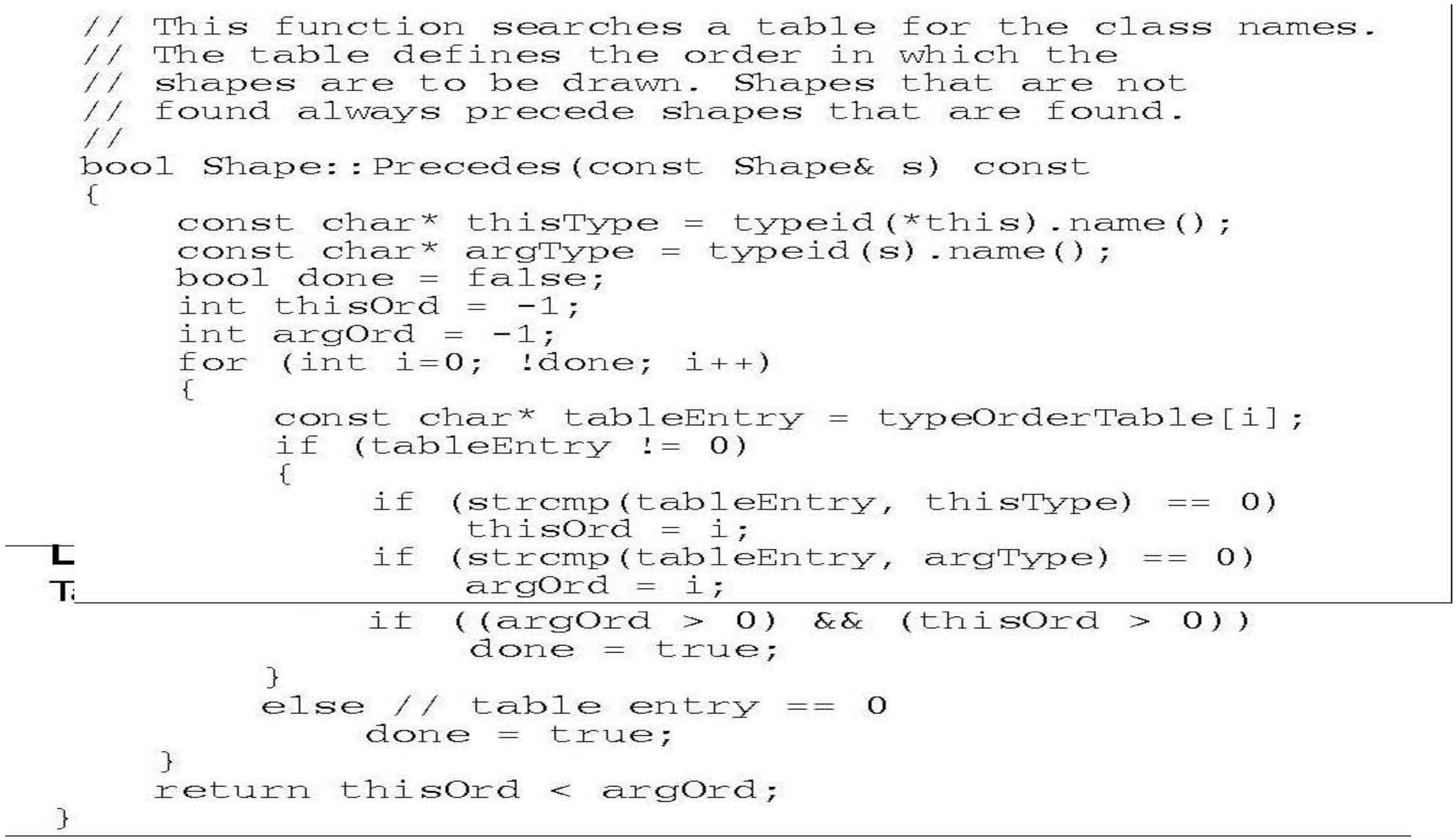


Listing 6

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
};
```

A possible way to solve the problem of Listing 5.



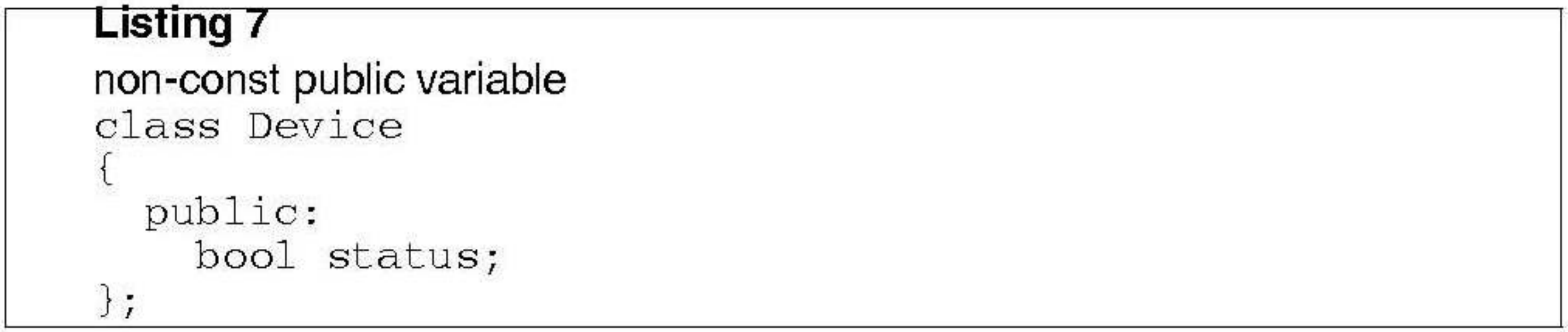
```
// This function searches a table for the class names.
// The table defines the order in which the
// shapes are to be drawn. Shapes that are not
// found always precede shapes that are found.
//
bool Shape::Precedes(const Shape& s) const
{
    const char* thisType = typeid(*this).name();
    const 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;
            if ((argOrd > 0) && (thisOrd > 0))
                done = true;
        }
        else // table entry == 0
            done = true;
    }
    return thisOrd < argOrd;
}
```

OCP Heuristics

Make all object-data private
No Global Variables!

- Changes to public data are always at risk to "open" the module
  - They may have a rippling effect requiring changes at many unexpected locations;
  - Errors can be difficult to completely find and fix. Fixes may cause errors else where.
  - When public data members changed, every function depends upon those members must be changed
- Non-private members are modifiable
  - Case 1: "I swear it will not change"
    - may change the status of the class (next slide)
  - Case 2: a `Time` class with open members
    - may result in inconsistent times (next slide)

Case 1 Example



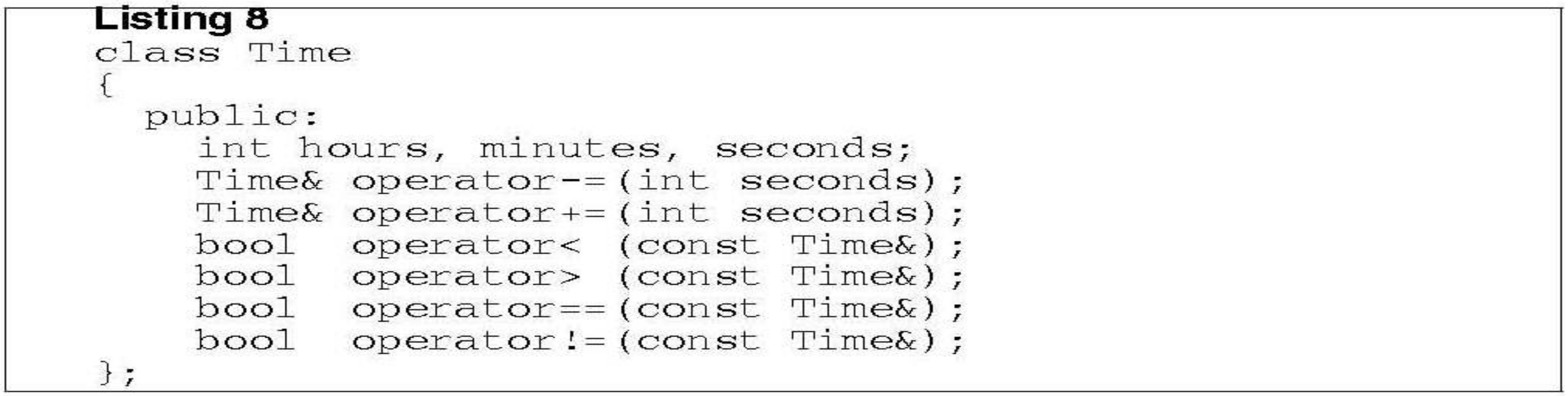
Listing 7

non-const public variable

```
class Device
{
  public:
    bool status;
};
```

- Suppose programmer knows *status* will never change. Do we need to make it private?
  - Some said: we can keep it public for debug purpose. There is no harm since we know no other code can change its value
  - Others said: Clients may take this adv. to modify *status.*
    - All code related to *status* may be affected! Too risky!

Case 2 Example



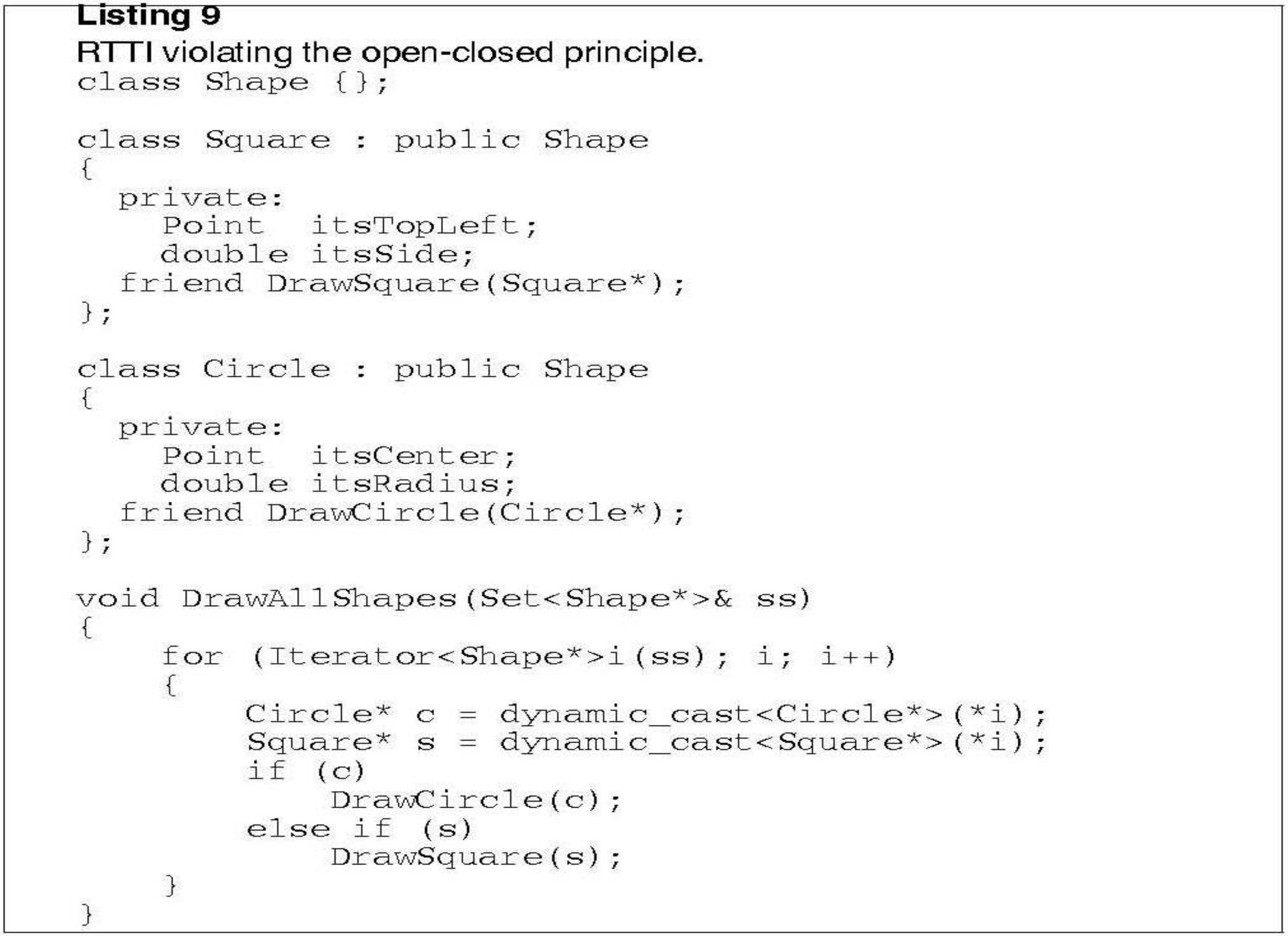
Listing 8

```
class Time
{
  public:
    int hours, minutes, seconds;
    Time& operator-=(int seconds);
    Time& operator+=(int seconds);
    bool  operator< (const Time&);
    bool  operator> (const Time&);
    bool  operator==(const Time&);
    bool  operator!=(const Time&);
};
```

- Some said: Let’s make this class public
  - What if we change *hours*, *minutes*, *seconds* independently?
    - It violates to atomic feature of Time!
  - But, there is no overriding reason to make these variables private (friend is still needed if private)
    - Of course, a better “style” is still to make them private

Probably called bad “style” not bad “design”

RTTI (or dynamic cast) is dangerous?

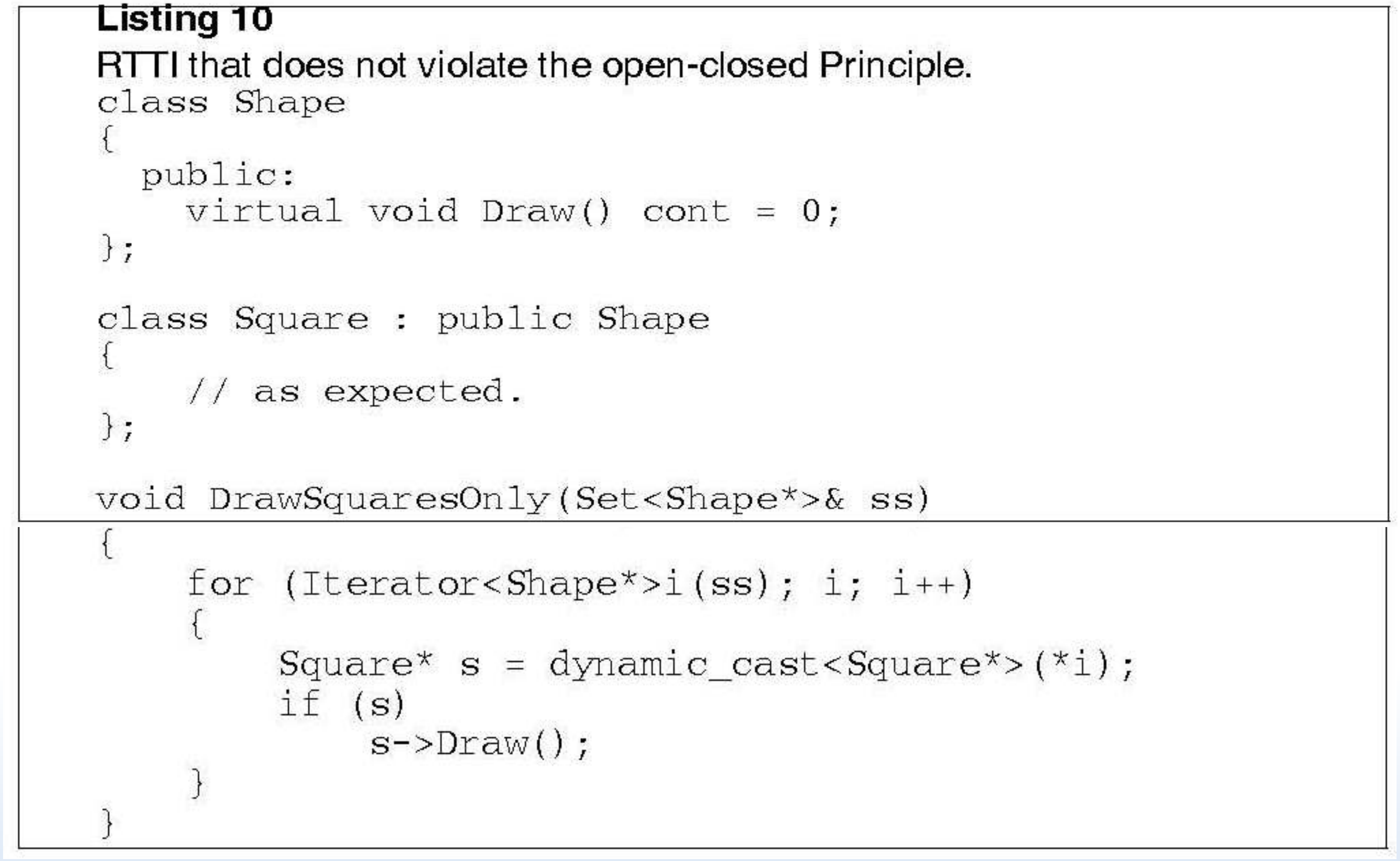


Listing 9

RTTI violating the open-closed principle.

```
class Shape {};

class Square : public Shape
{
  private:
    Point  itsTopLeft;
    double itsSide;
  friend DrawSquare(Square*);
};

class Circle : public Shape
{
  private:
    Point  itsCenter;
    double itsRadius;
  friend DrawCircle(Circle*);
};

void DrawAllShapes(Set<Shape*>& ss)
{
    for (Iterator<Shape*>i(ss); i; i++)
    {
        Circle* c = dynamic_cast<Circle*>(*i);
        Square* s = dynamic_cast<Square*>(*i);
        if (c)
            DrawCircle(c);
        else if (s)
            DrawSquare(s);
    }
}
```

RTTI (or dynamic cast) is dangerous?



Listing 10

RTTI that does not violate the open-closed Principle.

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

class Square : public Shape
{
    // as expected.
};

void DrawSquaresOnly(Set<Shape*>& ss)
{
    for (Iterator<Shape*>i(ss); i; i++)
    {
        Square* s = dynamic_cast<Square*>(*i);
        if (s)
            s->Draw();
    }
}
```

OCP Conclusion

- A heart of OO Design
  - Use the concepts of abstraction and polymorphisms
  - Reusability and Maintainability
  - The next two principles have to follow OCP
  - This principle is heavily applied in design patterns
    - E.g., Abstract Factory: One product that can be run in different OS

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.

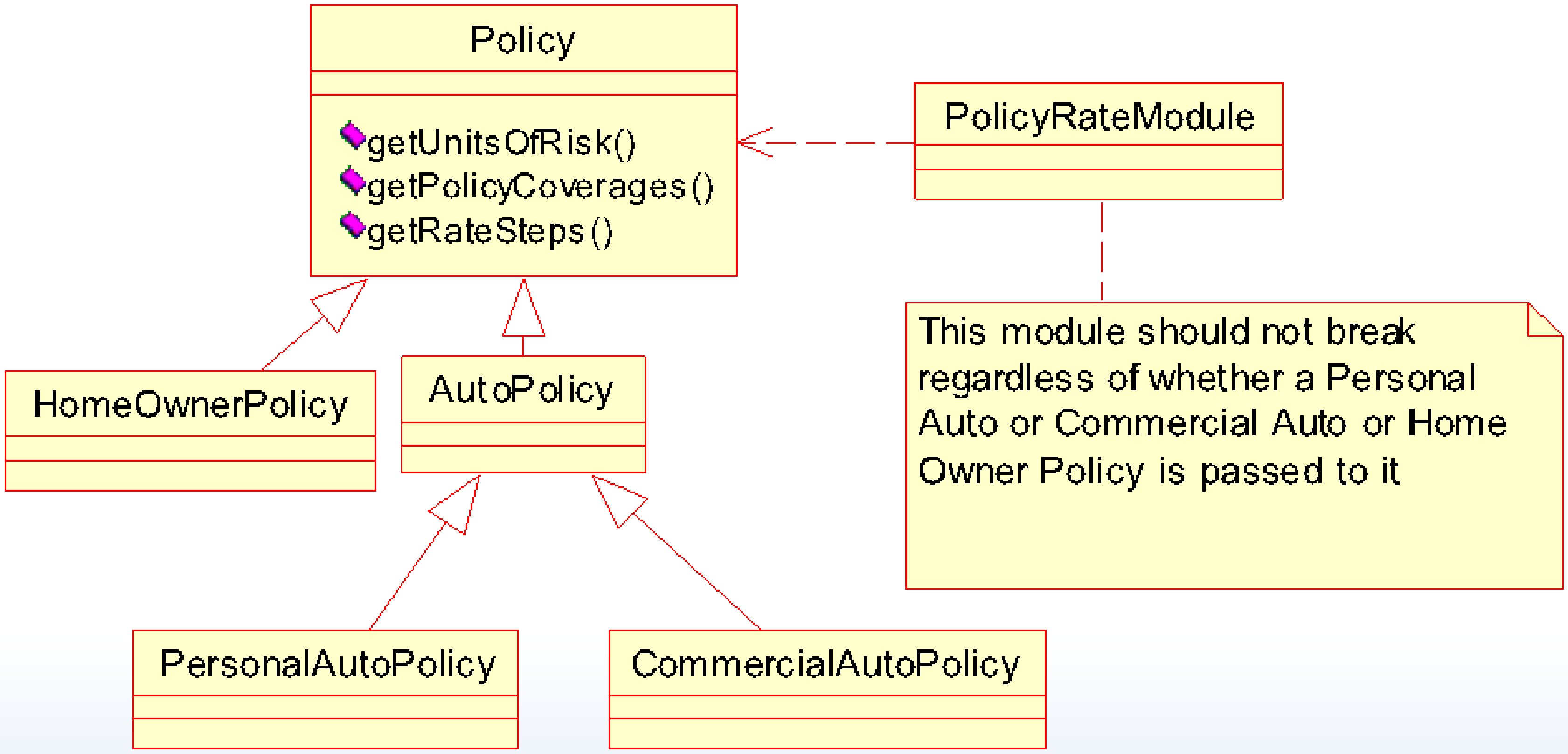
R. Martin, 1996

Liskov Substitution Principle (LSP)

- *"What is wanted here is something like the following substitution property: If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T."*

  --- B. Liskov

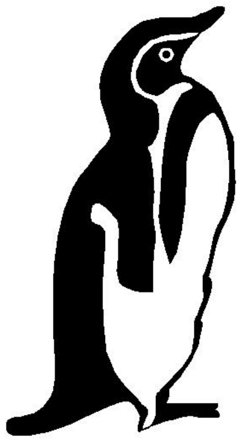
The Liskov Substitution Principle (LCP) Example



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!

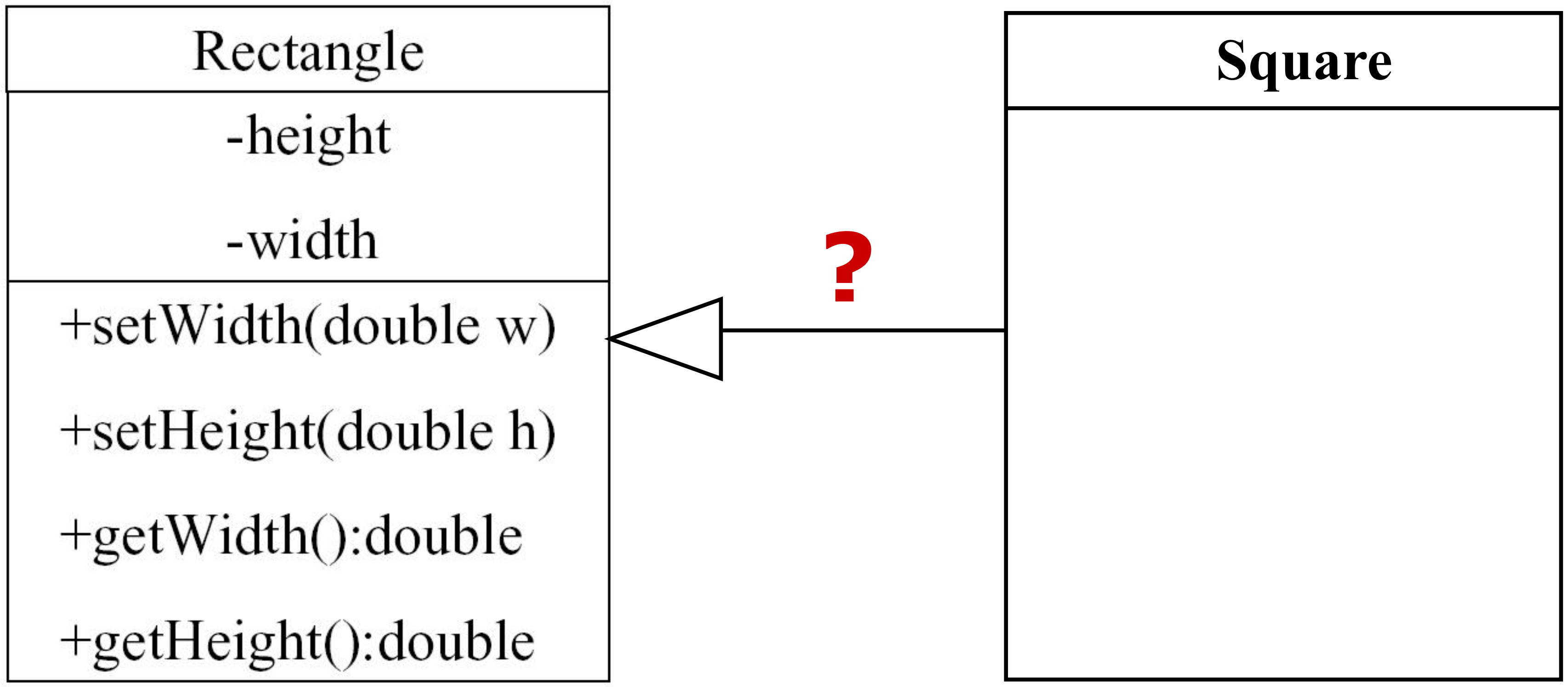


```
class Penguin : public Bird {
   public: void fly() {
     error ("Penguins don't fly!"); }
};
void PlayWithBird (Bird& abird) {
     abird.fly();    // OK if Parrot.
     // if bird happens to be Penguin...OOOPS!!
}
```

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

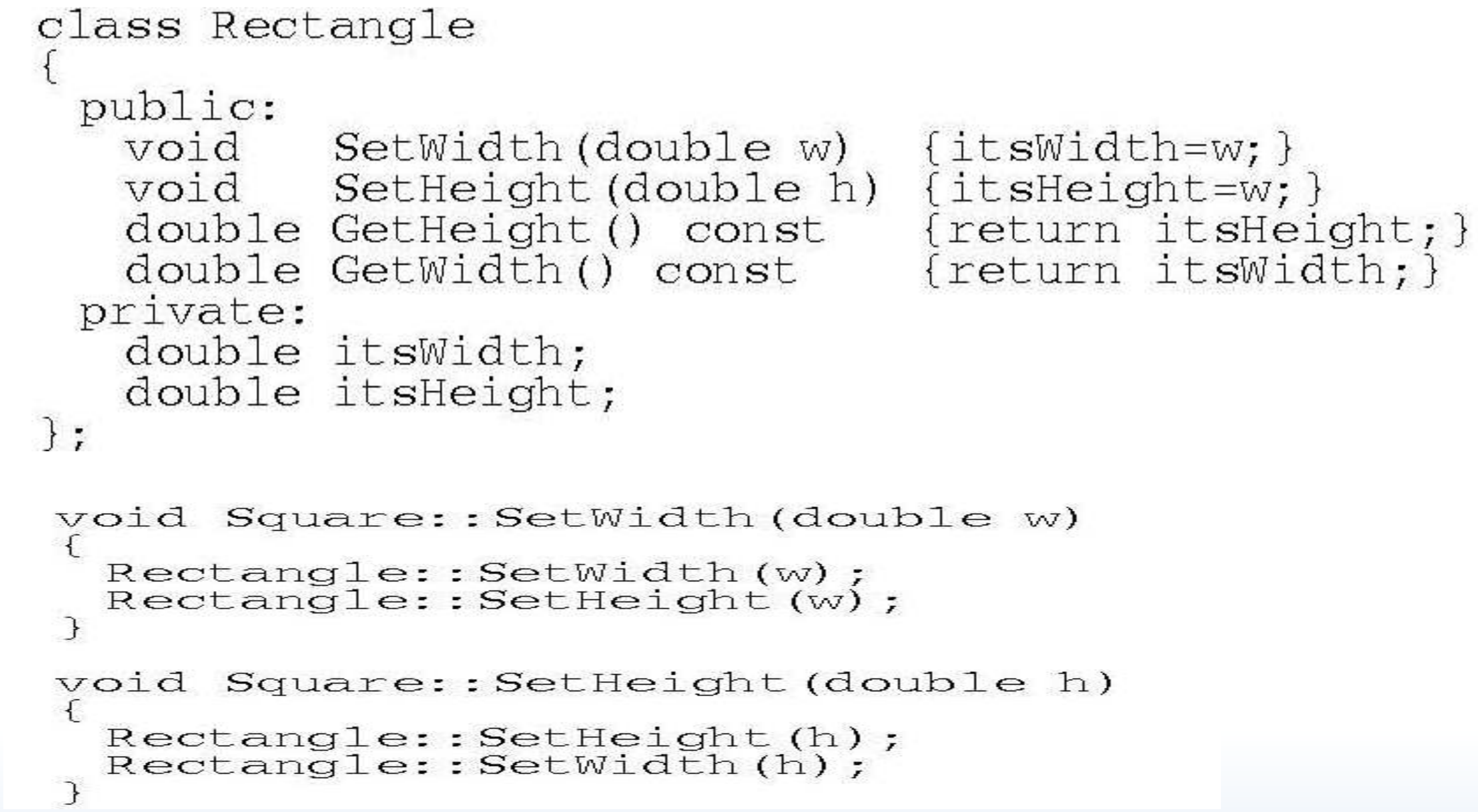
Subclass Penguin cannot replace superclass Bird, because it CANNOT fly!!

Square IS-A Rectangle?



- Should I inherit Square from Rectangle?

Square IS-A Rectangle?



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

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

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

If we call the following 3 functions

- Square s;
- s.SetWidth(1); //both Width and Height are 1
- s.SetHeight(2);//both Height and Width are now 2

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

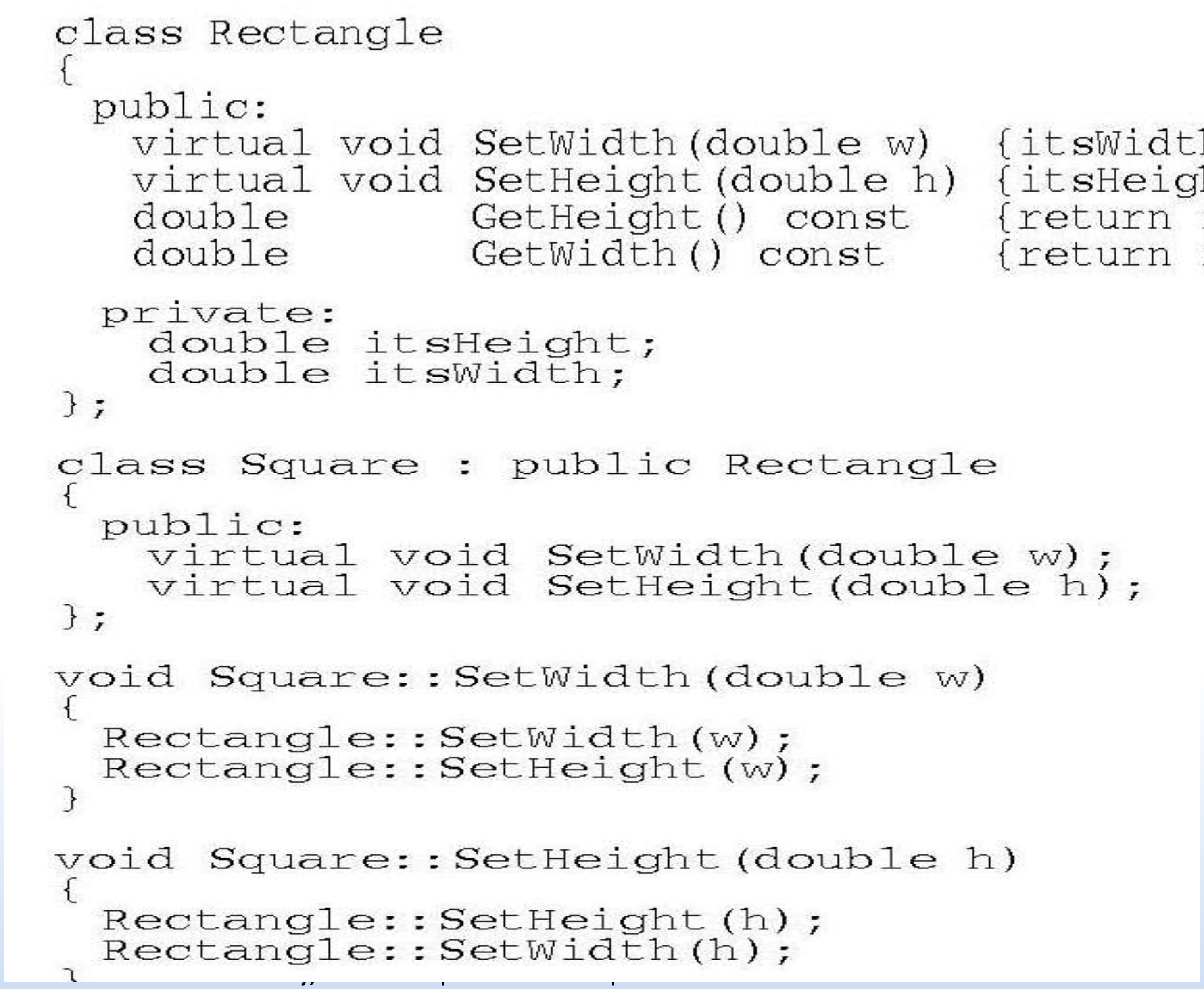
- What about the above function?

The Answer is ...

- It will call Rectangle’s setWidth function even though object r is Square.

- Solution:
  - Override **`setHeight`** and **`setWidth`**
    - change base class to set methods **`virtual`**
    - Keep in mind, without virtual, r will call setWidth of Rectangle, even if r is an object of Square. Then in this case, width and height may no longer be consistent.

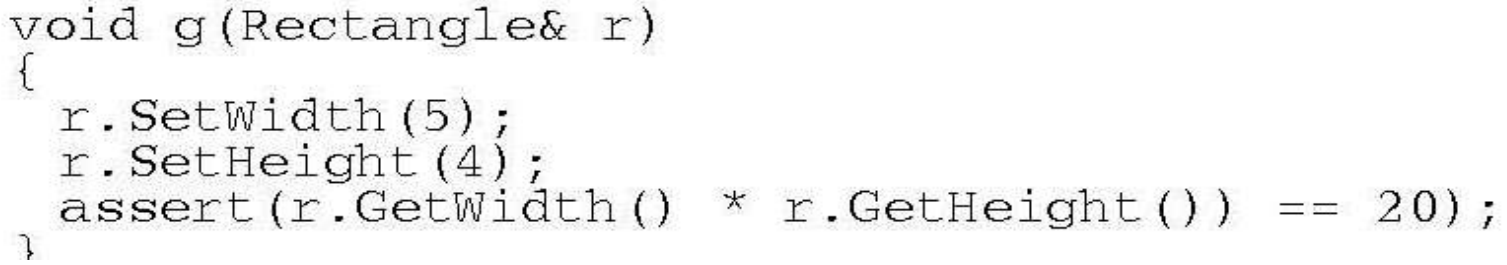
Square IS-A Rectangle?



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

  private:
    double itsHeight;
    double itsWidth;
};

class Square : public Rectangle
{
  public:
    virtual void SetWidth(double w);
    virtual void SetHeight(double h);
};

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

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

Square IS-A Rectangle?

- The real problem:
  - Was the programmer who wrote that function justified in assuming that changing the width of a *Rectangle* leaves its height unchanged?



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

Square IS-A Rectangle?

- If the real object passed in is a Square object, the assert will be wrong…should be 16!
  - There exists functions that take references or pointers to Rectangle objects but cannot operate properly upon Square objects. So it violates LSP and OCP.
- A Square might be a rectangle, but square object is-not a rectangle object. Because the "behavior" of square is not a rectangle.
  - LSP makes it clear that in OOP, IS-A relationship pertains to BEHAVIOR – not intrinsic private behavior but Extrinsic public behavior: behavior that clients depend upon

Design by Contract

- Advertised Behavior of an object:
  - advertised **Requirements** (Preconditions)
  - advertised **Promises** (Postconditions)

When redefining a method in a derivate class, you may only replace its precondition by a **weaker** one, *and its* postcondition by a **stronger** one

B. Meyer, 1988

⇒ Derived class services should **require no more** and **promise no less**

```
int Base::f(int x);
// REQUIRE: x is odd
// PROMISE: return even int
```

```
int Derived::f(int x);
// REQUIRE: x is int
// PROMISE: return 8
```

Design by Contract

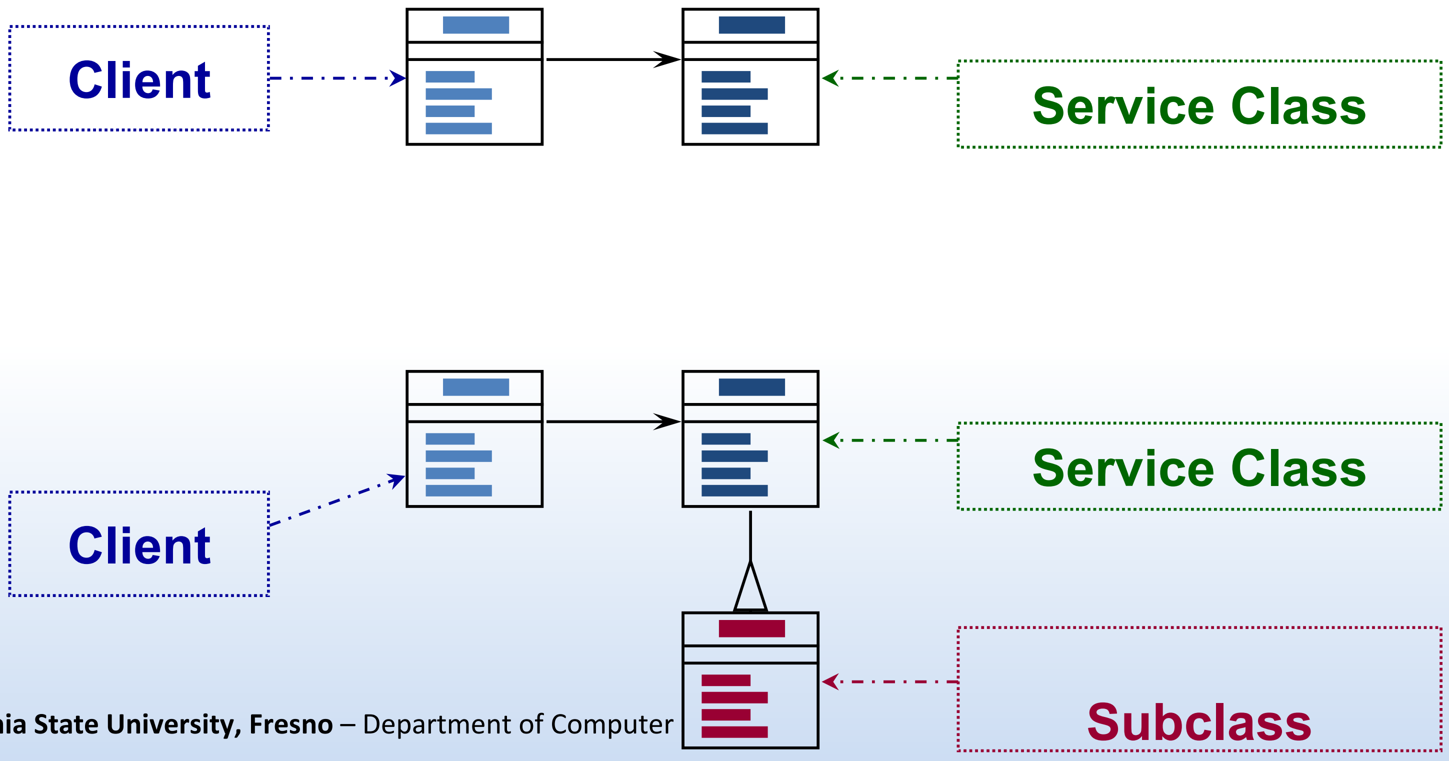
- Back to the Square and Rectangle's *SetWidth(double w)* example:
  - Postcodition of Rectangle is
    - assert((itsWidth == w) && (itsHeight == old.itsHeight))
    - This is stronger than Square's postcondition (i.e., itsWidth == w). So Square IS NOT a Rectangle!!
      - Subclass' post-condition should be stronger.
    - Namely, the behavior of a Square object is not consistent with the behavior of a Rectangle object

LSP is about Semantics and Replacement

- The meaning and purpose of every method and class must be clearly documented
  - Lack of user understanding will induce violations of LSP
  - In previous example, we have intuition about squares/rectangles, but this is not the case in most other domains
- Replaceability is crucial
  - Whenever any class is referenced by any code in any system, any future or existing subclasses of that class must be 100% replaceable
  - Because, sooner or later, someone **will** substitute a subclass;
    - it’s almost inevitable.
- *Violations of LSP are latent violations of OCP.*

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 (2)

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

- NOP = a method that does nothing
- Solution 1: Inverse Inheritance Relation
  - if the initial base-class has only additional behavior
    - e.g. **`Dog - DogNoWag`**
- Solution 2: Extract Common Base-Class
  - if both initial and derived classes have different behaviors
  - for **`Penguins`** $\rightarrow$ **`Birds`**, **`FlyingBirds`**, **`Penguins`**

Two reasons: replaceable and stronger postcondition.

LSP conclusion

- LSP is about inheritance
  - IS-A relationship pertains to **extrinsic** behavior (i.e., behavior that clients depend on)
    - Programmers need to make sure function g (clients depend on this function to compute Rectangle/Square area size) holds even though setHeight and setWidth are correct
- LSP has to conform to OCP
- 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

Dependency Inversion Principle

What are Bad Designs?

- Rigidity: It is hard to change because every change affects too many other parts of the system ~~("Extension and contraction")~~
  - OCP does not encourage changes in working parts
  - Rigidity is due to the fact that a single change to heavily interdependent software begins a cascade of changes in dependent modules. When these cascade changes cannot be predicted by designer or maintainer, the impact of changes cannot be estimated – so changes are not easily AUTHORIZED

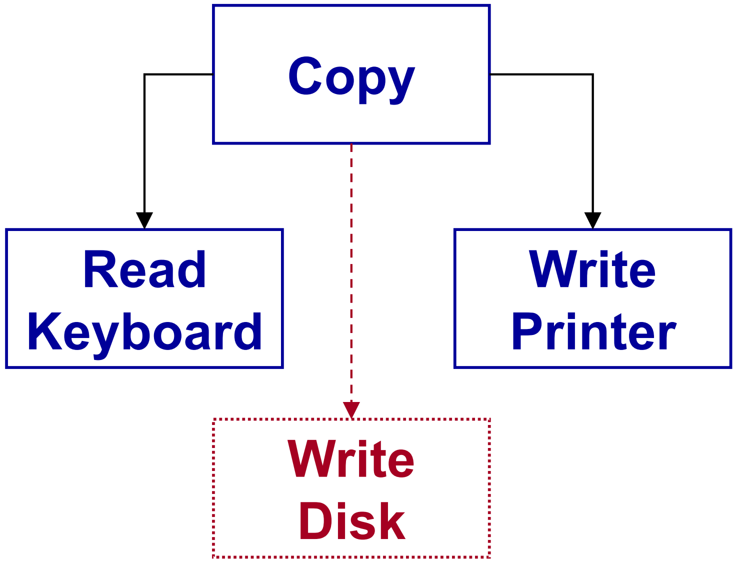
What are Bad Designs?

- Fragility: When you make a change, unexpected parts of the system break.
  - Regression faults!
  - Fragility decreases the credibility of the design and maintenance organization. we are unable to predict the quality of their product changes.

What are Bad Designs?

- Immobility: It is hard to reuse in another application because it cannot be disentangled from the current application.
  - ~~Code reuse in M1 of “on the criteria”~~
  - High coupling examples showed before.
  - Desirable (reusable) parts are depending on undesirable parts – it’s hard to separate desirable parts from undesirable parts, the cost of separation is too high so reusability is low.
- What causes them?
  - Interdependences between modules (i.e., high coupling)

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);
}
```

Copy cannot be reused! Higher level modules are more desirable to be reused – designs/decisions/policies in higher level module are more costly to design.

Copy has many dependencies with lower level IO modules, which becomes rigid and fragile.

Dependency Inversion Principle

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

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

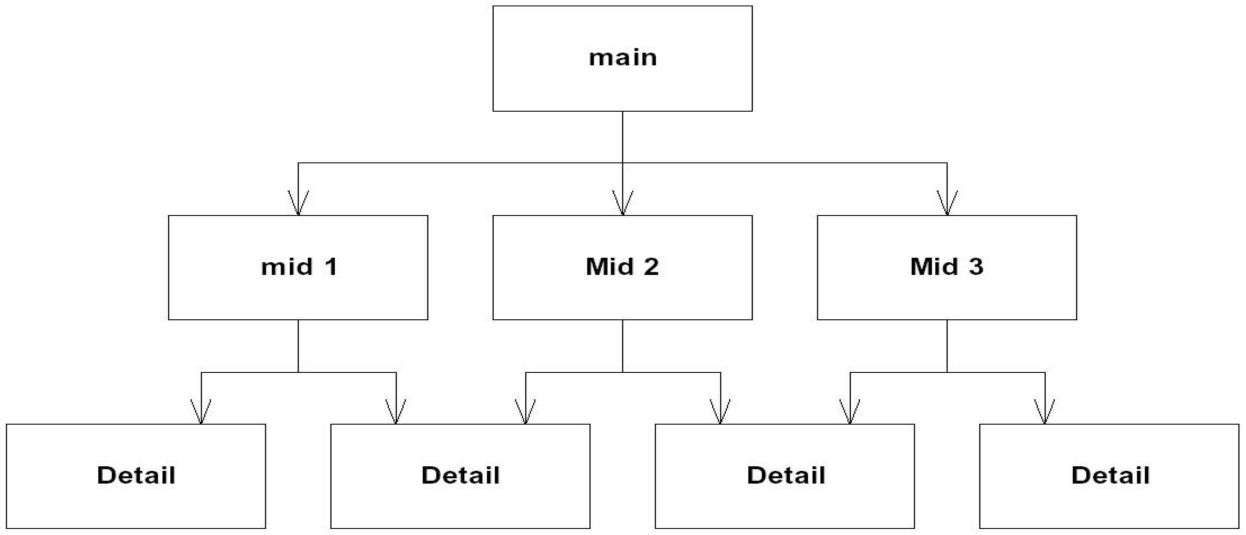
- High level modules contain *important policy decision and business module* of an application.
- It is these modules that contain the identity of an app. The fact is *high level should force lower level to change, not in reverse order*.
- We want to reuse high level (because high level contains business logics) as we already do in lower level.

Dependency Inversion Principle

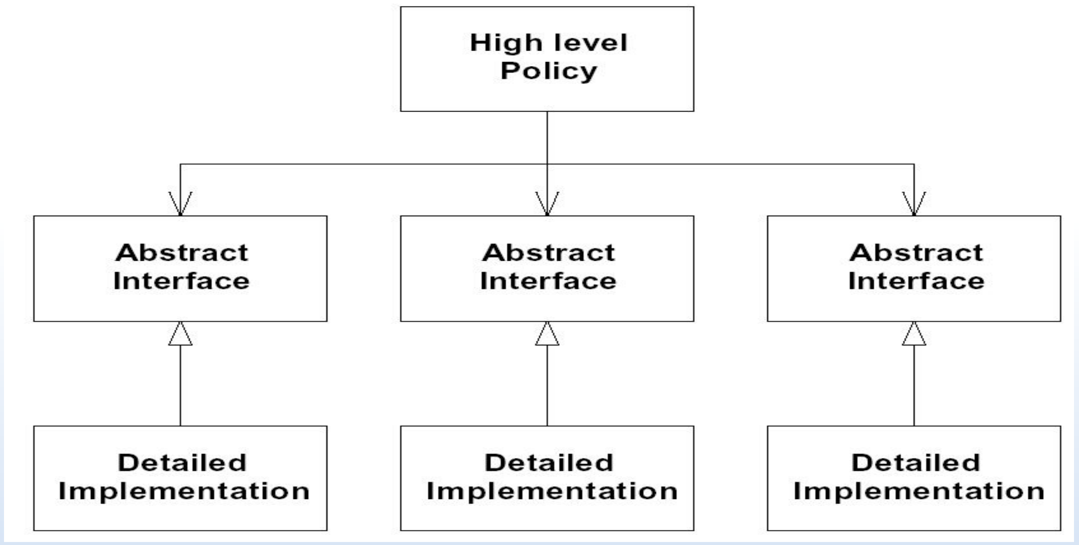
- Dependency Inversion is the strategy of depending upon *interfaces* or *abstract* functions and classes, rather than upon ***concrete*** functions and classes.
- Every dependency in the design should target an interface, or an abstract class. No dependency should target a concrete class.

Procedural vs. OO Architecture

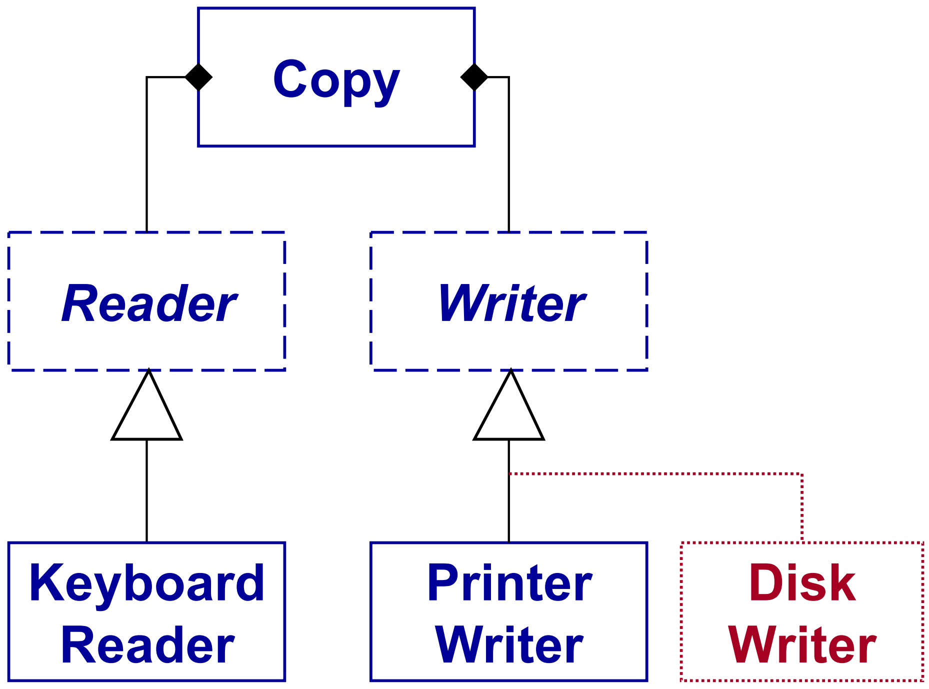
Procedural Architecture



Object-Oriented Architecture



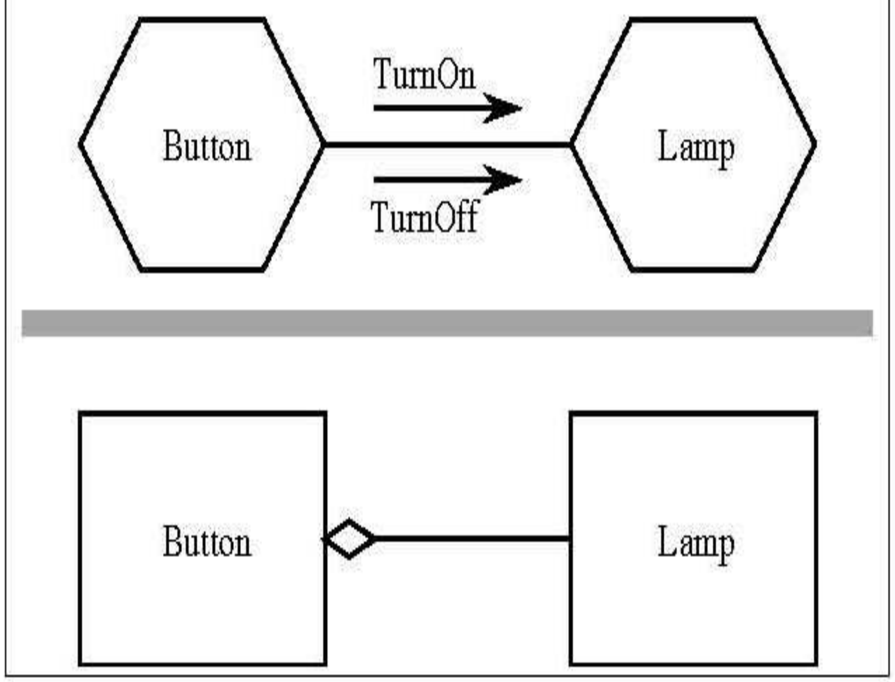
DIP Applied on Example

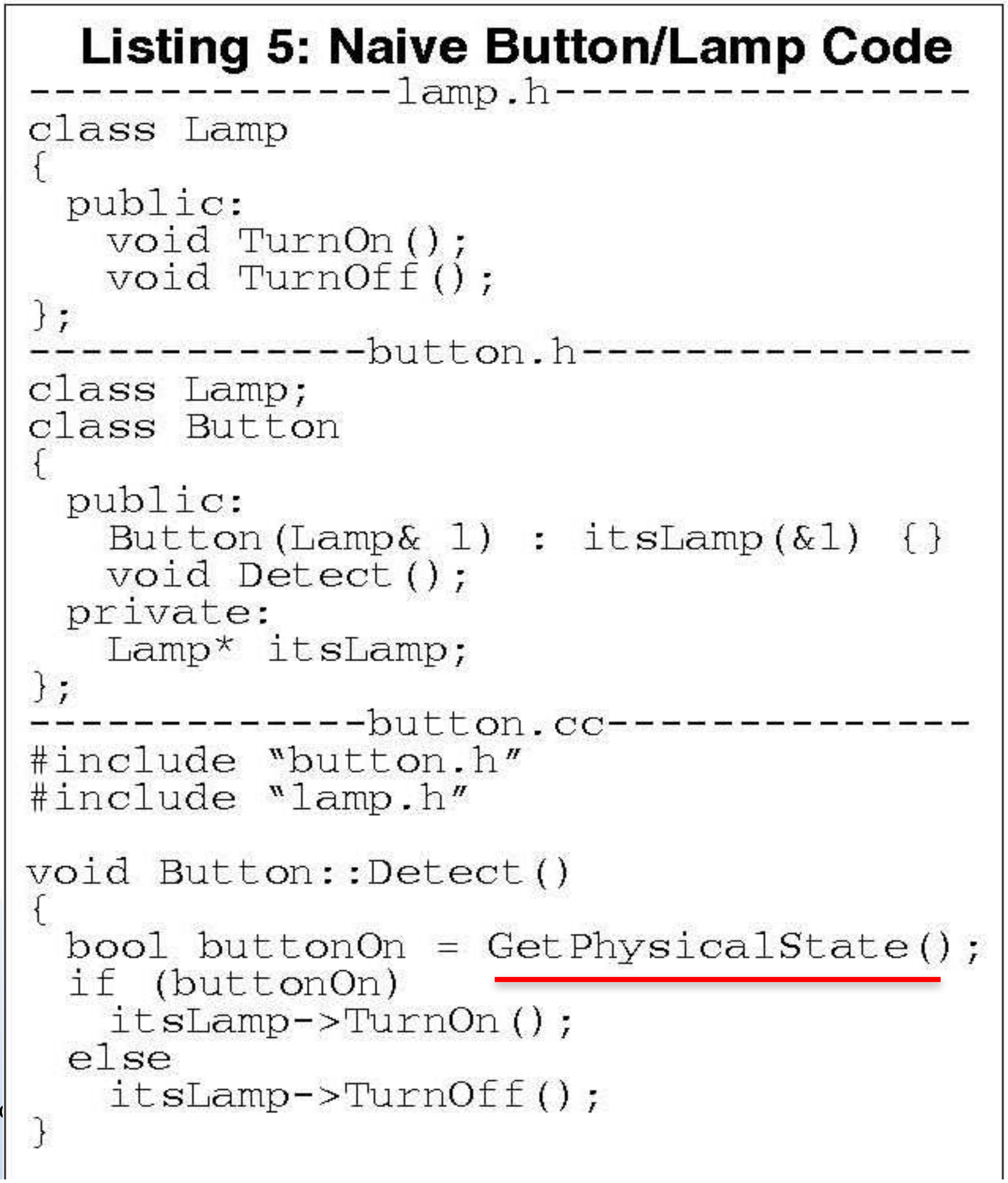


```
class Reader {
  public:
    virtual int read()=0;
};

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

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

Button/Lamp Example

Figure 5: Naive Button/Lamp Model



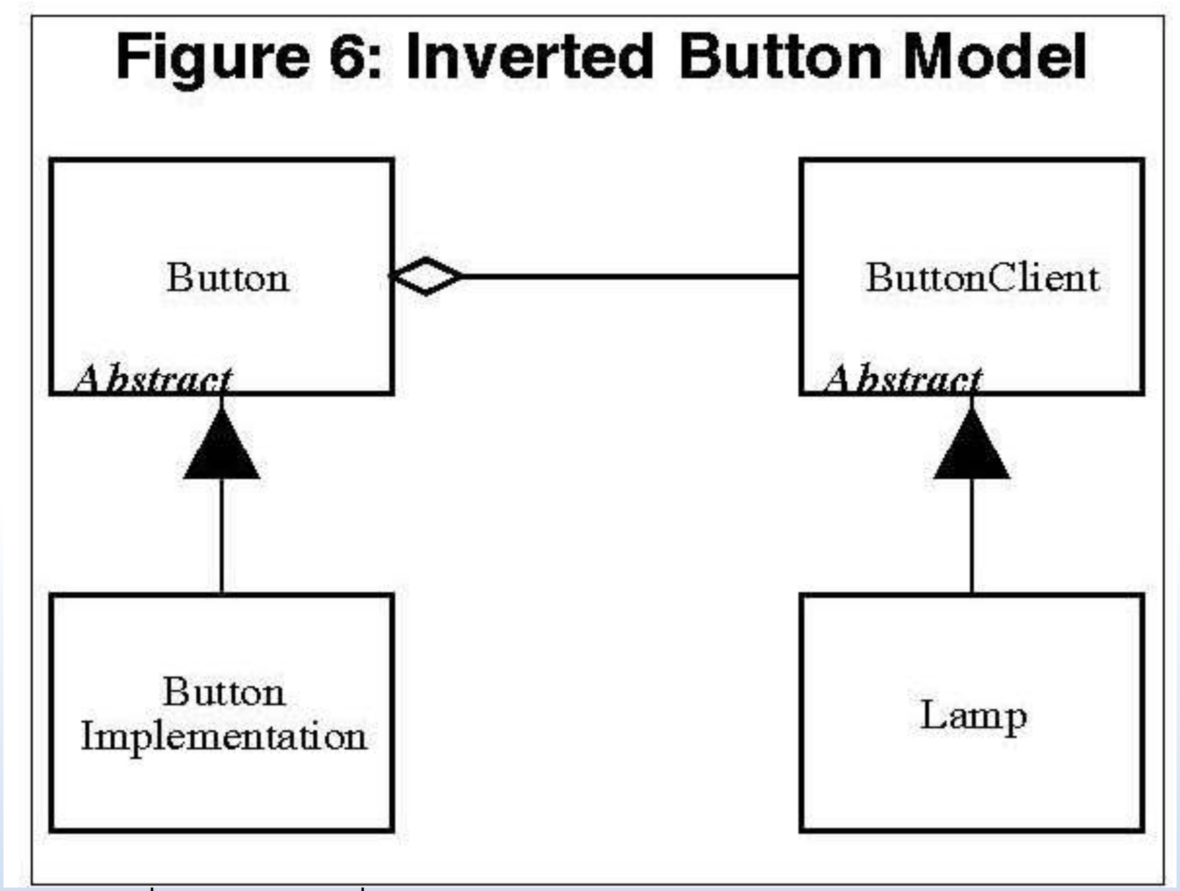


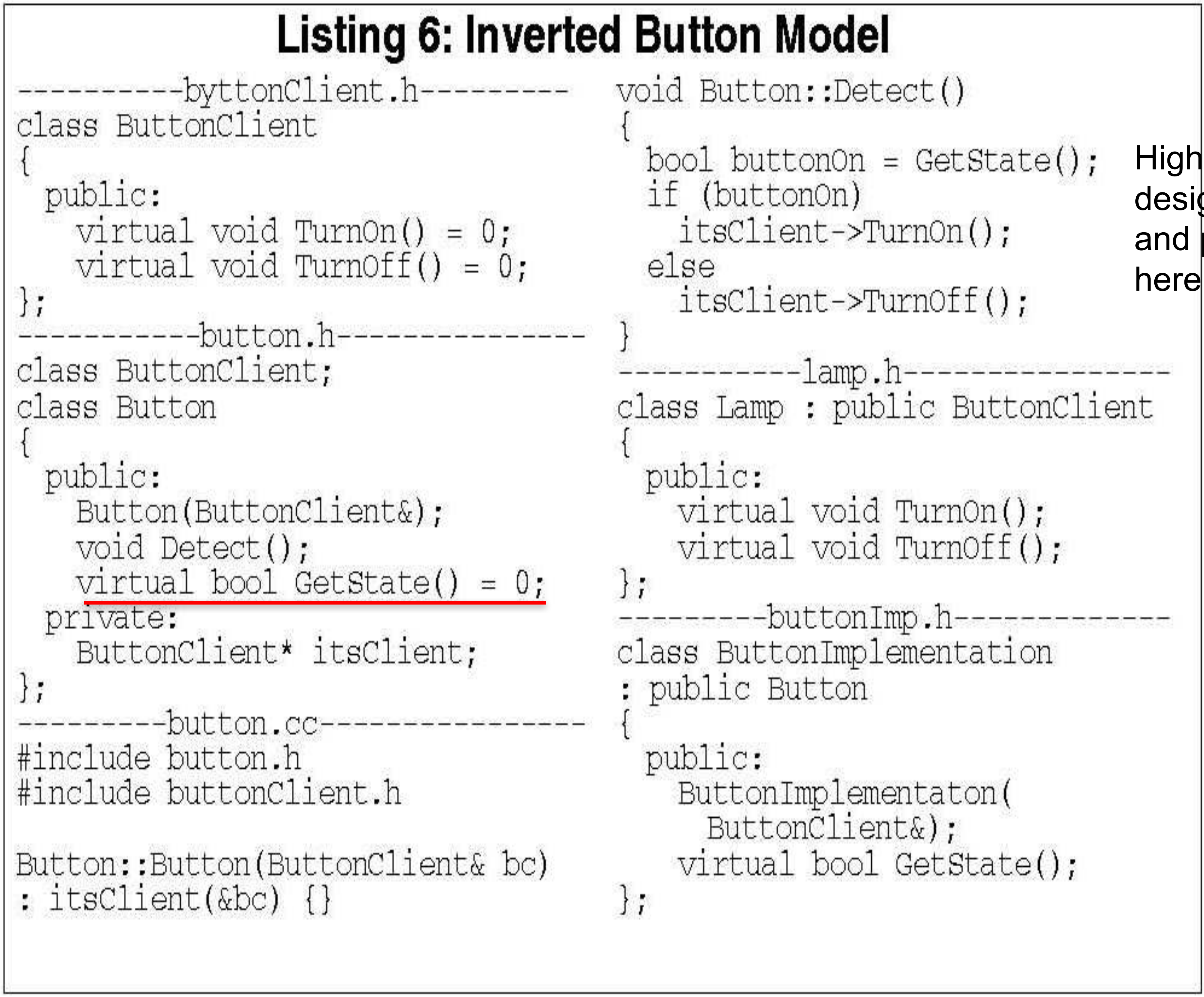
Listing 5: Naive Button/Lamp Code

```
---------------lamp.h-----------------
class Lamp
{
  public:
    void TurnOn();
    void TurnOff();
};
-------------button.h----------------
class Lamp;
class Button
{
  public:
    Button(Lamp& l) : itsLamp(&l) {}
    void Detect();
  private:
    Lamp* itsLamp;
};
-------------button.cc---------------
#include "button.h"
#include "lamp.h"

void Button::Detect()
{
  bool buttonOn = GetPhysicalState();
  if (buttonOn)
    itsLamp->TurnOn();
  else
    itsLamp->TurnOff();
}
```

Button/Lamp Example

- What if we want to use different mechanisms (e.g., gesture) for “getState”?





Listing 6: Inverted Button Model

```
----------byttonClient.h---------
class ButtonClient
{
  public:
    virtual void TurnOn() = 0;
    virtual void TurnOff() = 0;
};
-----------button.h--------------
class ButtonClient;
class Button
{
  public:
    Button(ButtonClient&);
    void Detect();
    virtual bool GetState() = 0;
  private:
    ButtonClient* itsClient;
};
---------button.cc----------------
#include button.h
#include buttonClient.h

Button::Button(ButtonClient& bc)
: itsClient(&bc) {}
```

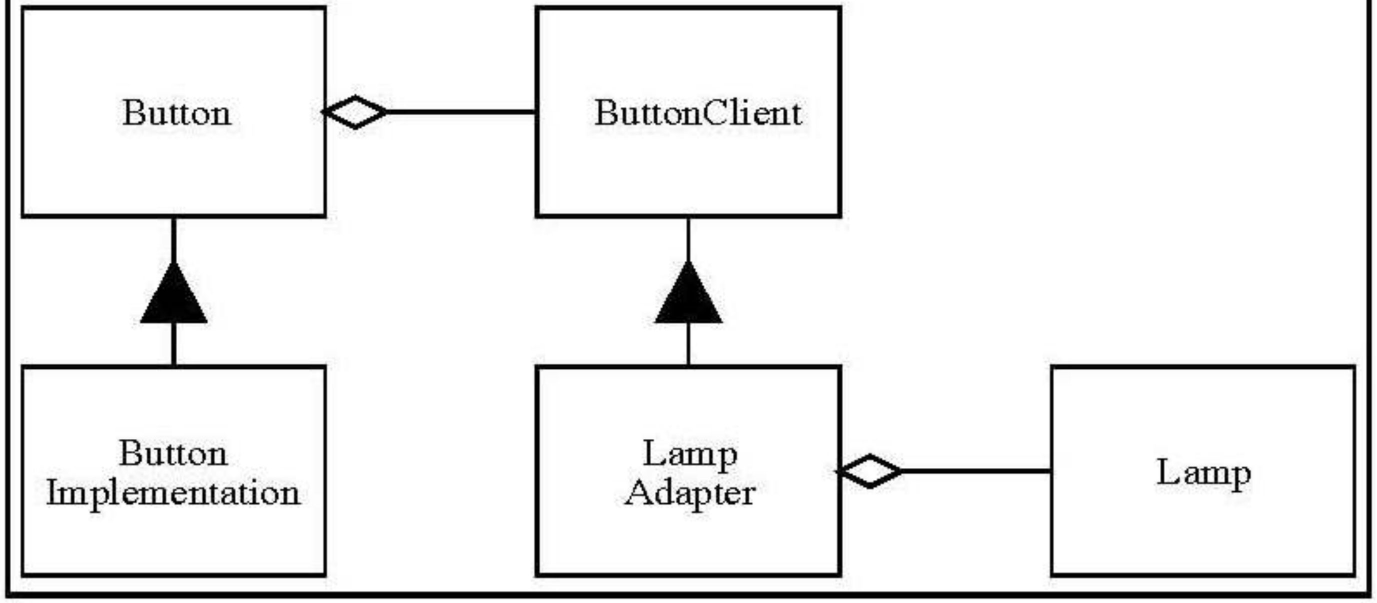
```
void Button::Detect()
{
  bool buttonOn = GetState();
  if (buttonOn)
    itsClient->TurnOn();
  else
    itsClient->TurnOff();
}
-----------lamp.h----------------
class Lamp : public ButtonClient
{
  public:
    virtual void TurnOn();
    virtual void TurnOff();
};
---------buttonImp.h-------------
class ButtonImplementation
: public Button
{
  public:
    ButtonImplementaton(
      ButtonClient&);
    virtual bool GetState();
};
```

High level design and policy here

Button/Lamp Example (SKIP for now)

- What if Lamp comes from a third party library and we don't have permission to view or modify the source code?
  - Adapter Design Pattern!

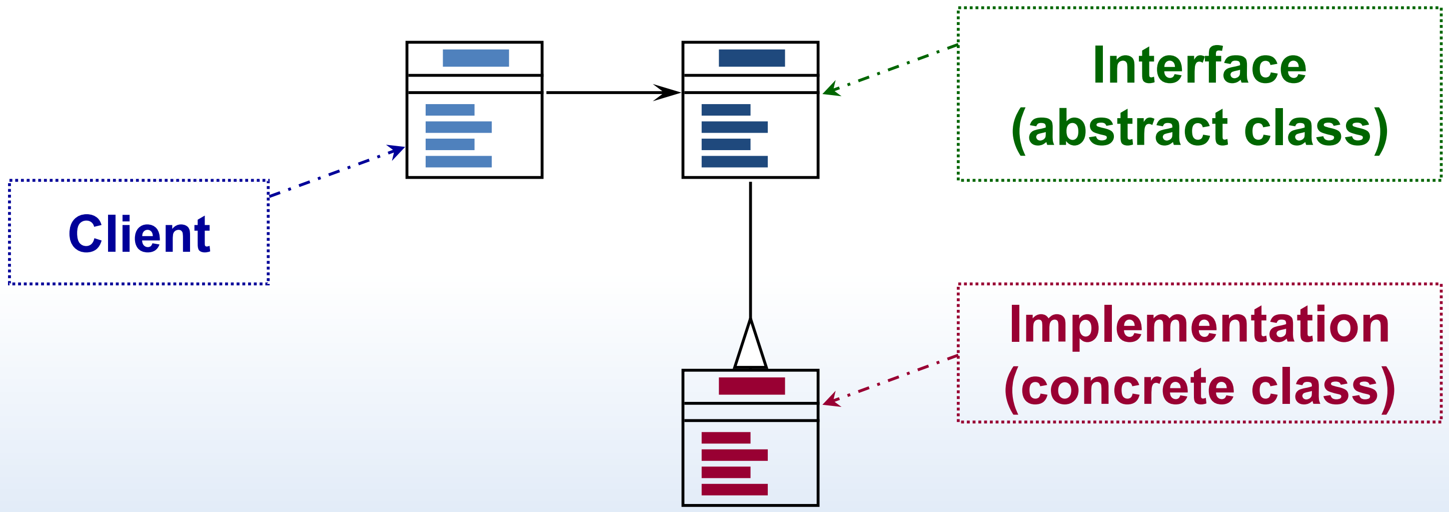
Figure 7: Lamp Adapter



DIP Related Heuristic

Design to an interface, not an implementation!

- Use inheritance to avoid direct bindings to classes:



Design to an Interface

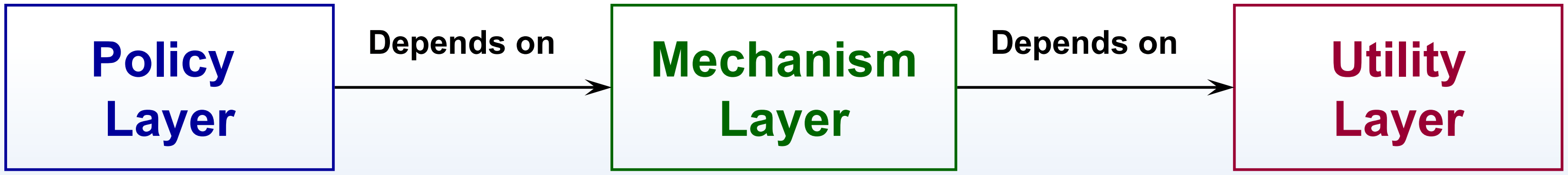
- **Abstract classes/interfaces:**
  - tend to change much <u>less frequently</u>
  - abstractions are 'hinge points' where it is easier to <u>extend</u>
  - shouldn't have to modify classes/interfaces that represent the abstraction (OCP)

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

DIP Related Heuristic

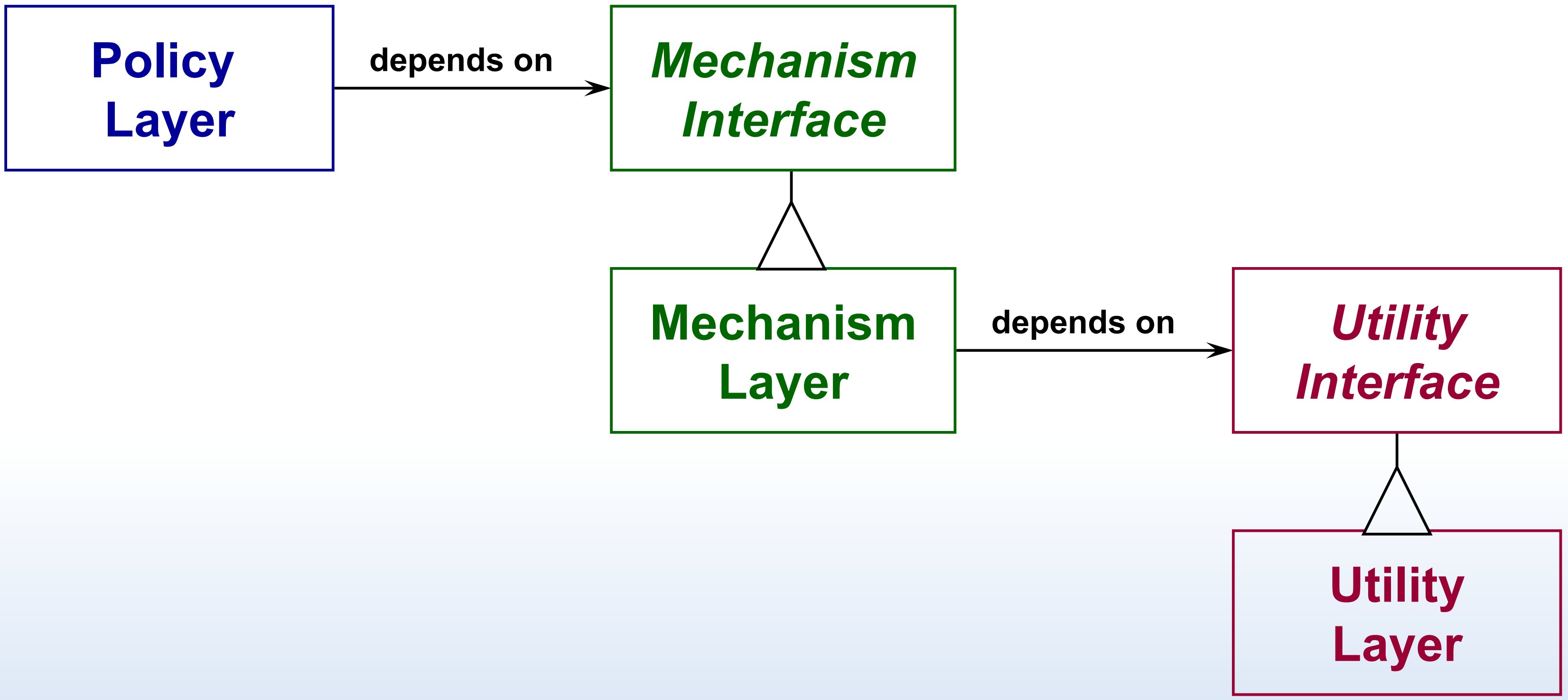
Avoid Transitive Dependencies

- Avoid structures in which higher-level layers depend on lower-level abstractions:
  - In example below, Policy layer is ultimately dependant on Utility layer.



Solution to Transitive Dependencies

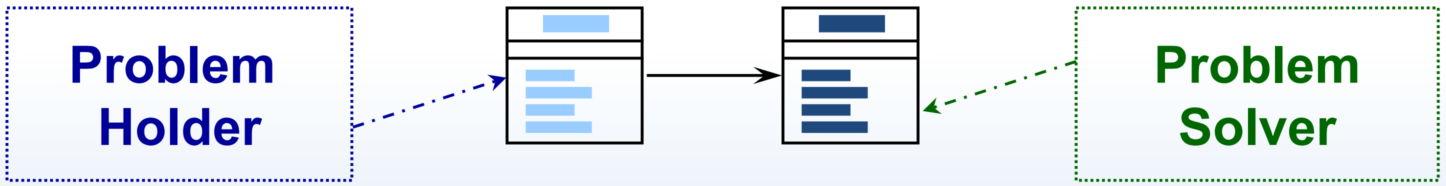
- Use inheritance and abstract ancestor classes to effectively eliminate transitive dependencies:



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:



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 ;)

Two additional principles

- The Single Responsibility Principle
  - This principle discusses the need to place things that change for different reasons in different classes.
- The Interface Segregation Principle
  - Fat class interfaces are all too common. This article describes a principle that helps designers partition fat classes into multiple interfaces