SIEMENS Healthineers

Design Principles

Training material sourced from CT RDA

Design Principles

Design principles are **universally** applicable concrete design rules to **guide** a developer, designer, or architect during **design conceptualization or maintenance** phases.

SOLID

- **S**ingle Responsibility Principle
- **O**pen-closed Principle
- **L**iskov's Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

Example

```
interface Modem {
  public void dial(String pno);
  public void hangup();
  public void send(char c);
  public char recv();
}
```

Seems like good set of modem functions – but is it one or two sets of responsibilities?

- Connection management (dial and hangup)
- Data communication (send and recv)

Separate if connection functions are changing

Don’t separate if no changes expected

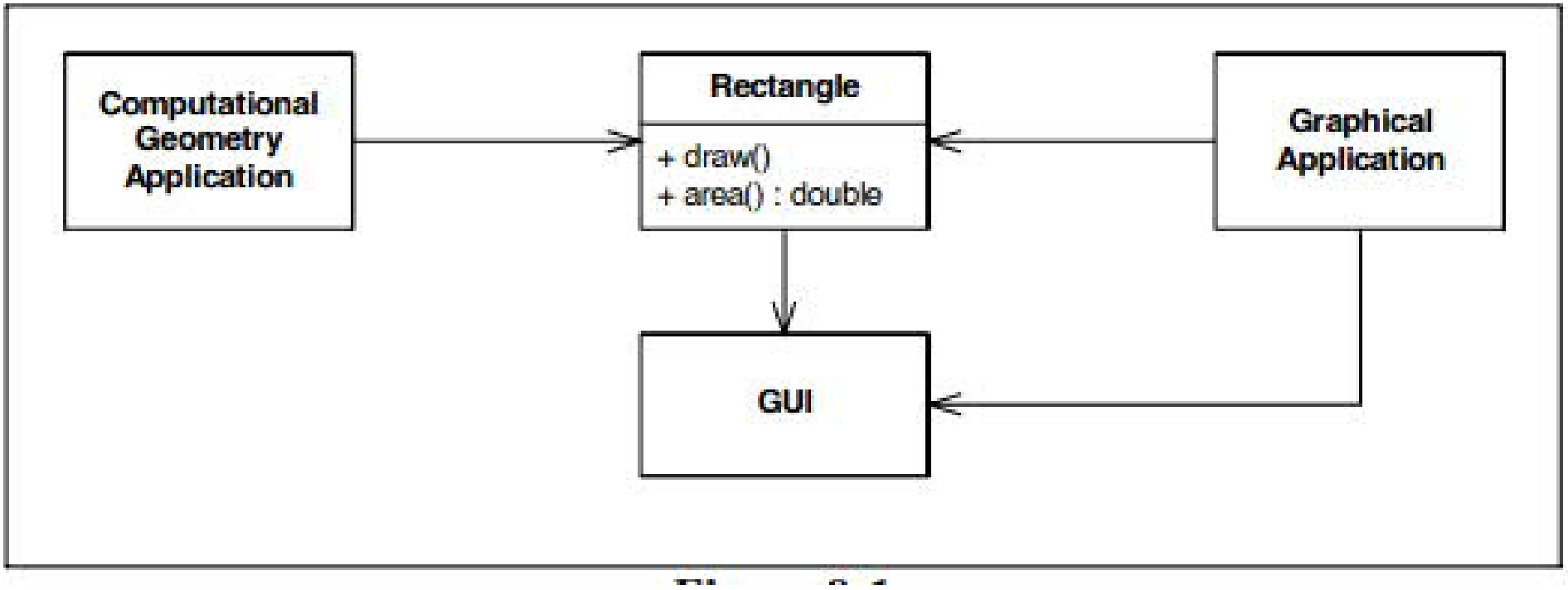
Single Responsibility Principle

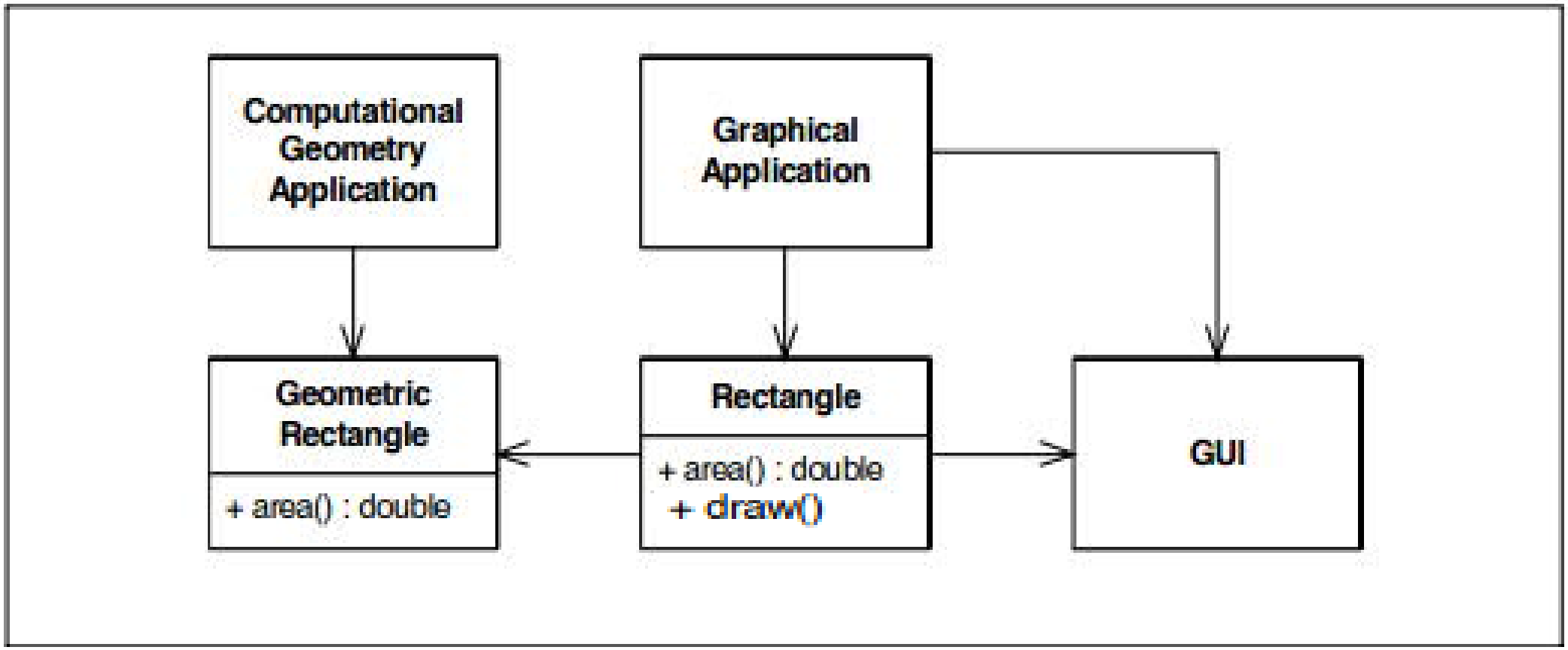
- There should never be more than **one reason** for a class to **change**
- Every class should have a **single purpose**
- **Concentrate** on a single aspect in an abstraction
- **A responsibility is a “reason for change”**

Two different applications use the Rectangle class.

One application does computational geometry. It uses Rectangle to help it with the mathematics of geometric shapes. It never draws the rectangle on the screen.

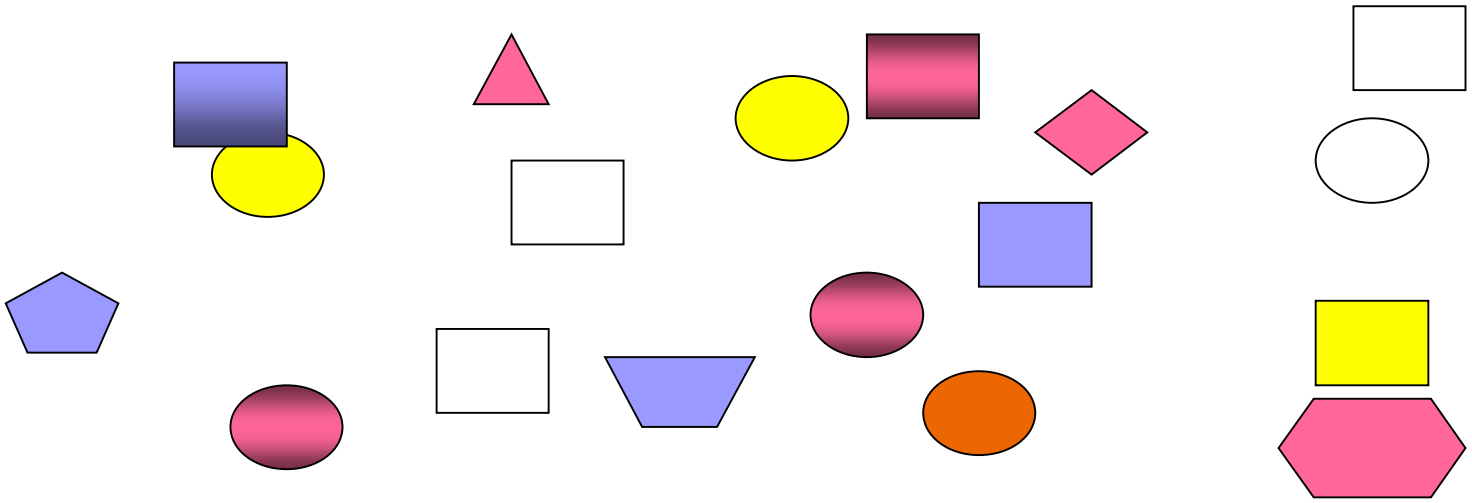
The other application is graphical in nature. It may also do some computational geometry, but it definitely draws the rectangle on the screen.





Single Responsibility Principle

The Shape Example



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;
       }
   }
}
```

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();
}
```

Open-Closed Principle

- Software entities should be **open for extension** and **closed for modification**
  - Open for extension – extend with new behaviors
  - Closed for modification – extending behavior does not result in changes to the existing source code or binary code of module. .exe, DLL, .jar remain untouched!

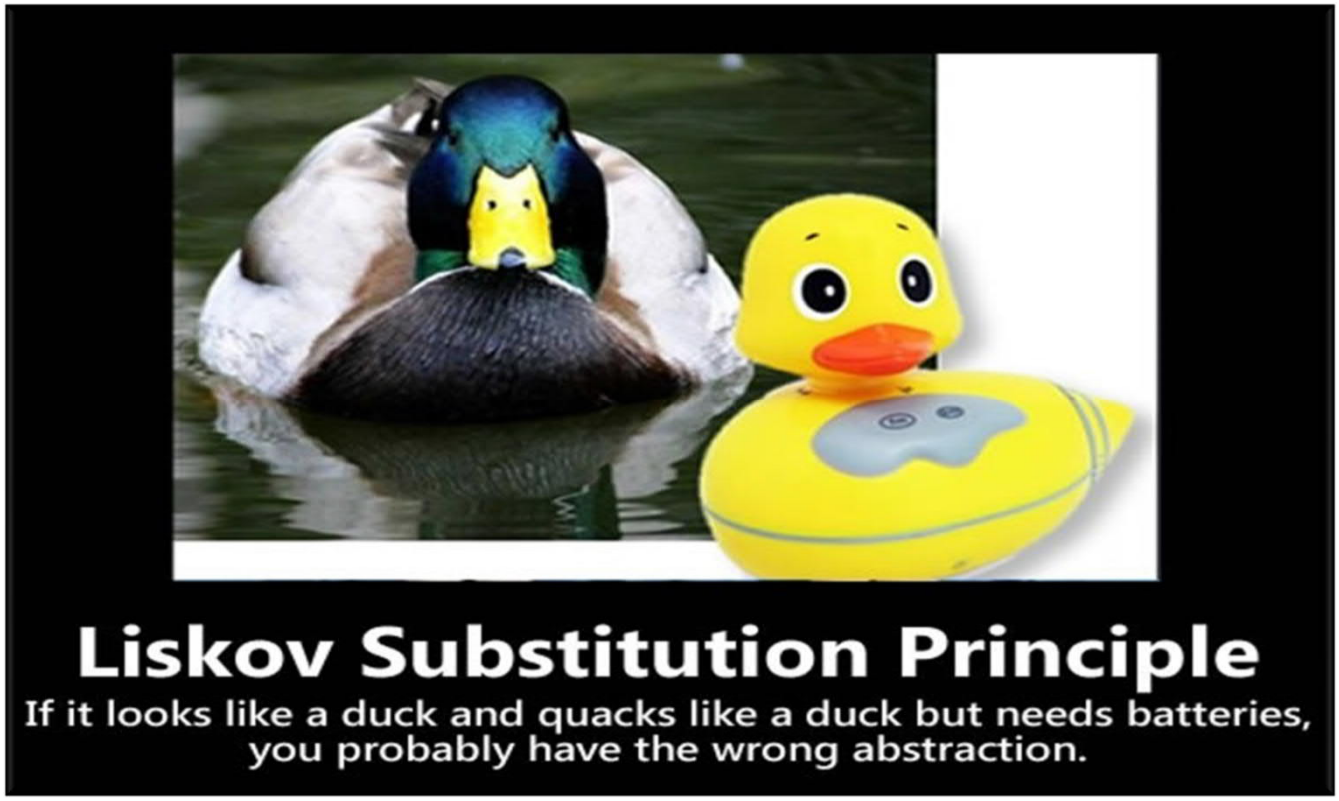
- **Define interface, extend implementation**

Open-Closed Principle

Liskov's Substitution Principle

- Clients that use references (or pointers) to base abstractions **must be able to use** their **derived abstractions** without knowing it
- A subclass must be **replaceable** where its super class is referenced
- Everything that is true of the base type should be true of the subtype

Liskov’s Substitution Principle



Example

A team's task is to create a basic email object

Client Team A, responsible for creating an application that sends an email reminder as a part of their process, creates a client that consumes the server and everything works

```java
import java.util.List;

public interface IMessage {

    public void fillAddresses(List<String> addresses);
    public void fillMessageBody(String message);
    public void fillSubject(String subject);
    public boolean send();
}
```

```java
import java.util.List;

public class SmtpMessage implements IMessage{
    public void fillAddresses(List<String> addresses){
        // do what is required here
    }
    public void fillMessageBody(String message){
        // do what is required here
    }
    public void fillSubject(String subject){
        // do what is required here
    }
    public boolean send(){
        // send message here
        return true;
    }
}
```

From: http://codebalance.blogspot.com/2010/09/oop-solid-rules-interface-segregation.html

Example

```
import java.util.List;

public class SmsMessage implements IMessage{

    public void fillAddresses(List<String> addresses){
        // do what is required here
    }
    public void fillMessageBody(String message){
        // do what is required here
    }
    public void fillSubject(String subject){
        throw new UnsupportedOperationException();
    }
    public boolean send(){
        // send message here
        return true;
    }

}
```

Team now receives a request from Client Team B which wants to support sending a text message

Since it is almost the same as sending an email message, they decide to use polymorphism

Both Clients are happy!
But already LSP is violated

From: http://codebalance.blogspot.com/2010/09/oop-solid-rules-interface-segregation.html

Team now receives a request from Client Team A that they want to enhance the system to support the email to be sent to a list of addresses as a blind copy

```java
import java.util.List;

public interface IMessage {

    public void fillAddresses(List<String> addresses);
    public void fillMessageBody(String message);
    public void fillSubject(String subject);
    public boolean send();
    public void fillBccAddresses(List<String> bccAddresses);
}
```

```java
import java.util.List;
public class SmtpMessage implements IMessage{
    public void fillAddresses(List<String> addresses){
        // do what is required here
    }
    public void fillMessageBody(String message){
        // do what is required here
    }
    public void fillSubject(String subject){
        // do what is required here
    }
    public boolean send(){// send message here
        return true;
    }
    public void fillBccAddresses(List<String> bccAddresses){
        // do what is required here
    }
}
```

```java
import java.util.List;
public class SmsMessage implements IMessage{

    public void fillAddresses(List<String> addresses){
        // do what is required here
    }
    public void fillMessageBody(String message){
        // do what is required here
    }
    public void fillSubject(String subject){
        throw new UnsupportedOperationException();
    }
    public void fillBccAddresses(
            List<String> bccAddresses){
        throw new UnsupportedOperationException();
    }
    public boolean send(){
        // send message here
        return true;
    }
}
```

Example

Client Team A gets the new server and things are great

But Client Team B is now broken and must recompile and go through a round of regression testing because a feature was added for another team

Interfaces may contain groups of methods, where each group serves a different set of clients. Best to separate them.

From: http://codebalance.blogspot.com/2010/09/oop-solid-rules-interface-segregation.html

Interface Segregation Principle

- Clients should **not be forced to depend** upon interfaces that they **do not need**
  - When a client A depends on a class that contains methods that it does not use, but which other clients use, client A will be affected by changes those other clients force upon the class
- **Don’t pollute interfaces**
- **Avoid fat interfaces**
- **Interfaces here do not mean Java Interfaces. They mean the public methods that are exposed by a class**

ISP in Action

```java
import java.util.List;
public interface IMessage {

    public void fillAddresses(List<String> addresses);
    public void fillMessageBody(String message);
    public boolean send();
}
```

```java
import java.util.List;
public class SmtpMessage implements IEmailMessage{
    public void fillAddresses(List<String> addresses){
        // do what is required here
    }
    public void fillMessageBody(String message){
        // do what is required here
    }
    public void fillSubject(String subject){
        // do what is required here
    }
    public boolean send(){// send message here
        return true;
    }
    public void fillBccAddresses(List<String> bccAddresses){
        // do what is required here
    }
}
```

```java
import java.util.List;
public interface IEmailMessage extends IMessage{

    public void fillSubject(String subject);
    public void fillBccAddresses(List<String> bccAddresses);
}
```

```java
import java.util.List;
public class SmsMessage implements IMessage{

    public void fillAddresses(List<String> addresses){
        // do what is required here
    }
    public void fillMessageBody(String message){
        // do what is required here
    }
    public boolean send(){
        // send message here
        return true;
    }
}
```

Now Client Team B has a specific interface IMessage that it can use

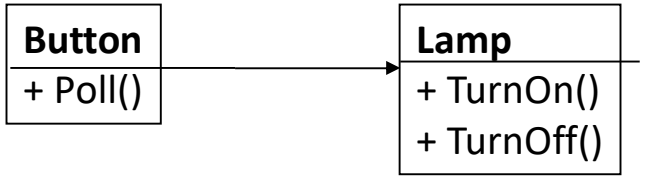
Similarly, Client Team A has a specific interface IEmailMessage that it can use

ISP allows IEmailMessage to be changed without impacting Client Team B

Interface Segregation Principle

Example

Button object has Poll method to determine whether or not user has “pressed” it. The Button could be represented as an icon on GUI, home security system, physical button, etc. The Poll method helps detect if the button is activated or deactivated.



Naïve Model

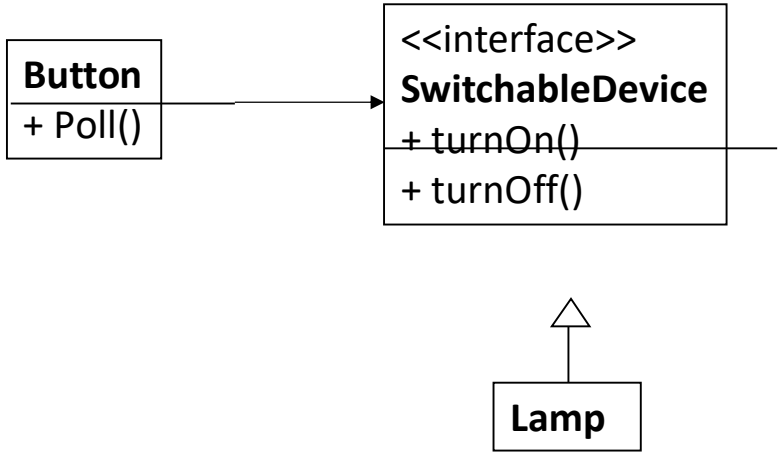
```
public class Button {
  private Lamp itsLamp;
  public boolean poll() {
    return (itsLamp.turnOn());
  }
  …
}
```

Button can’t be reused (for e.g. for a fan)
Changes to Lamp might affect Button.

Dependency Inversion Principle

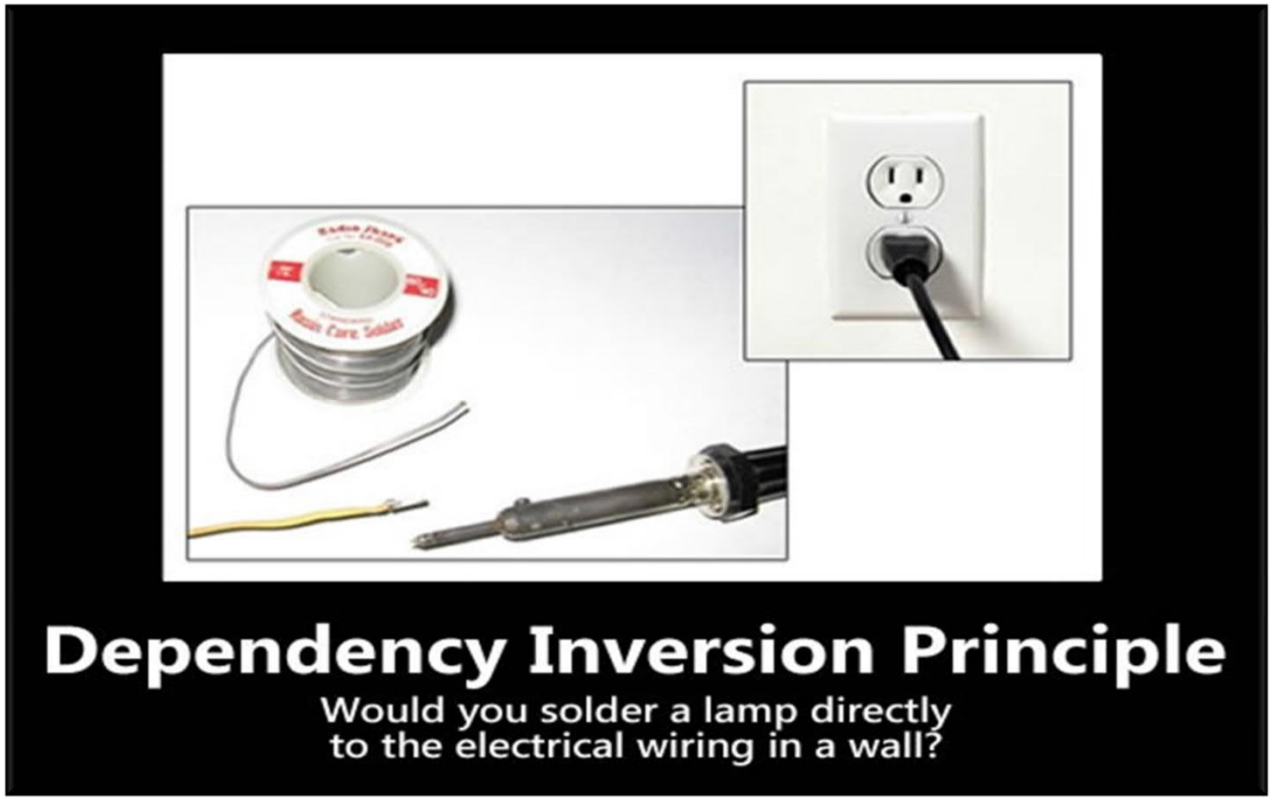
- **Abstractions** should **not** depend upon **implementation details**. Details should be dependent upon abstractions.
- **High-level** abstractions should not depend on **low-level** abstractions.
- **Clients** should depend on **interfaces** not implementation (loose coupling)

Applying the DIP



Now Button can control any device that implements SwitchableDevice
Notice that SwitchableDevice does NOT depend on Button

Dependency Inversion Principle



Some Other Commonly Used Principles

Don’t Repeat Yourself Principle (DRY) - “Every piece of knowledge must have a single, unambiguous, authoritative representation within a system” [HT99].

Keep it simple Silly (KISS) – Avoid unnecessary complexity. Keep things simple since it helps improve understandability and helps detect deficiencies easily.

Composition Preference Principle (CPP) - Shorter name for the principle “favor object composition over class inheritance” [EGV94].

Variation Encapsulation Principle (VEP) – Information hiding wherein the concept that is varying is encapsulated within a class or a hierarchy.

Acyclic Dependencies Principle – Dependencies between packages or classes should not form cycles.

Summary

The OO design principles help us:

- As guidelines when designing flexible, maintainable and reusable software
- As standards when identifying the bad design
- As laws to argue when doing code review

Keep the design of a system as simple, clean, and expressive as possible

- Don’t allow broken windows
- Apply them in iterations (not to a big, up-front design)
- Sometimes you have to make trade-offs

Thank you!