Introduction to Software Engineering
Agile Design Principles

CMPS115 – Summer 2017
Richard Jullig

Acknowledgements

- Material based on
  Robert Martin, Agile Principles, Patterns, and Practices (2007)
  - Chapters 7 thru 12
  - See Piazza > Resources > Reading Material

- Other worthwhile parts of the book
  - Extreme Programming practices
  - Design patterns (with examples)

The SOLID Design Principles

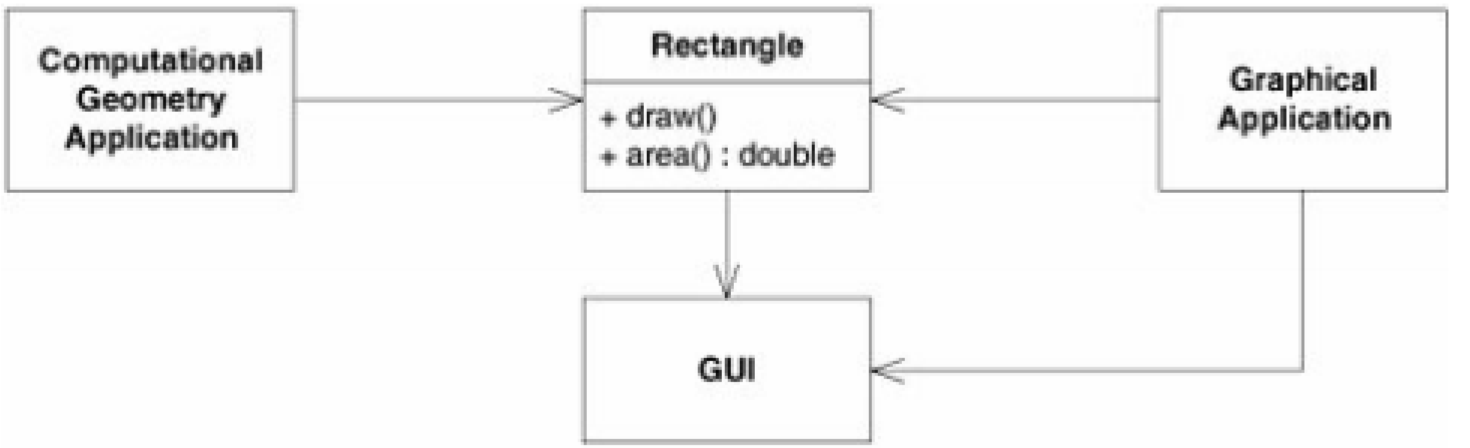
- SRP: the single responsibility principle
- OCP: the open/closed principle
- LSP: the Liskov substitution principle
- ISP: the interface segregation principle
- DIP: the dependency inversion principle

SRP: Single Responsibility Principle

A class should have only one reason to change.

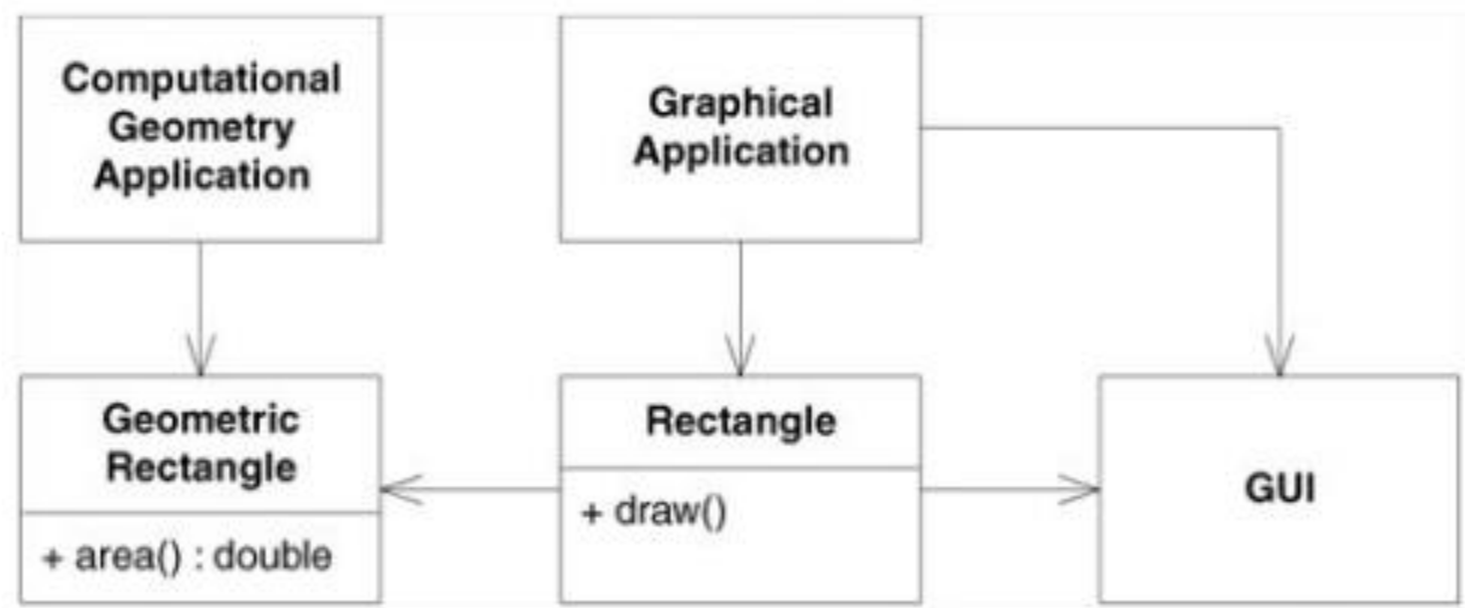
- Adapted from the notion of component cohesion
  - The elements of a component/module should be functionally related
- Each responsibility (group of related functional elements) may be a source of change.
- Multiple responsibilities ➔ multiple reasons for change
- Coupling of responsibilities ➔ fragile designs

SRP: Example (1)



- Rectangle
  - Geometric computation responsibility: area()
  - Graphical application responsibility: draw()
- Design violates SRP

SRP: Example (2)



- Geometric computation and graphical rendering separated
  - Both can be used independently from each other
- Design conforms to SRP

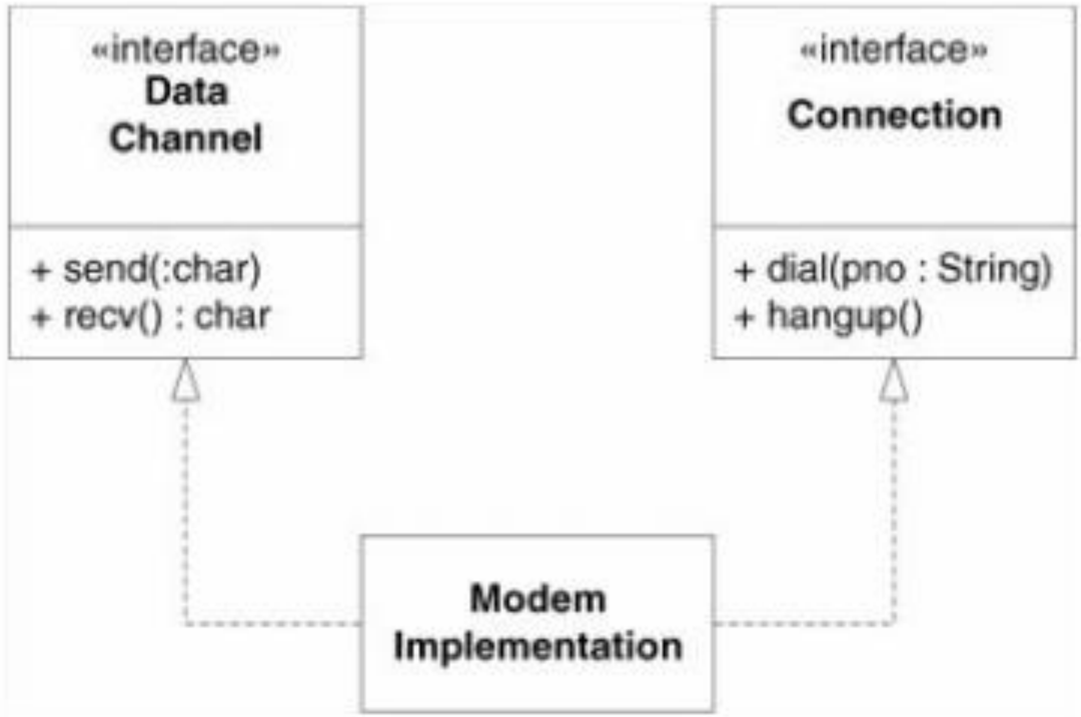
Non scholae sed vitae discimus.

SRP: Modem Example (1)

```
public interface Modem
{
  public void Dial(string pno);
  public void Hangup();
  public void Send(char c);
  public char Recv();
}
```

- Two responsibilities
  - Connection: Dial(), Hangup()
  - Data transfer: Send(), Recv()
- SRP violated
- Should responsibilities be separated?
  - If connection interface changes,
    classes using data transfer interface have to be recompiled
- When this happens (or there is good reason to expect this to happen) then separate responsibilities

SRP: Modem Example (2)



- Separate responsibilities (axes of change)
  - If/when they become axes of change
- Components depending on Data Channel interface
  now independent of components depending on Connection interface

Non scholae sed vitae discimus.

OCP: Open/Closed Principle

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

.

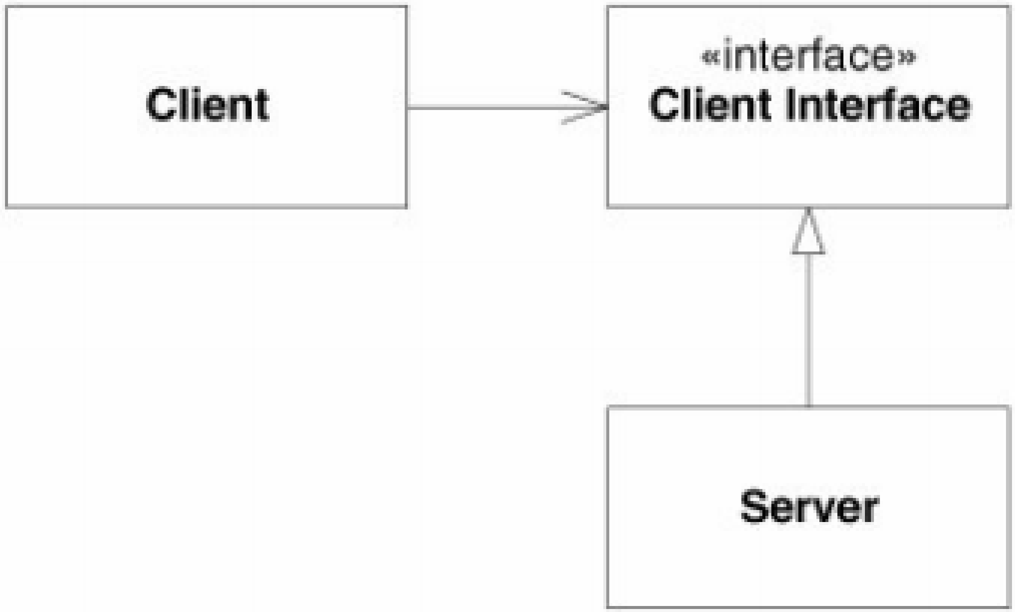
- OCP: introduced by Bertrand Meyer
  - In: *Object-Oriented Software Construction* (1997, 2nd ed.)

- Open for extension
  - Changes can add new elements/behaviors
- Closed for modification
  - Changes should not modify existing elements/behaviors
- Abstraction is key
  - Abstract classes allow extension without modification

OCP: Example (1)

- OCP violated:
  - Client class has reference to Server class
  - If Server class replaced by different class,
    Client needs to be modified (name of new Server class)
- Therefore: Client is not open and closed

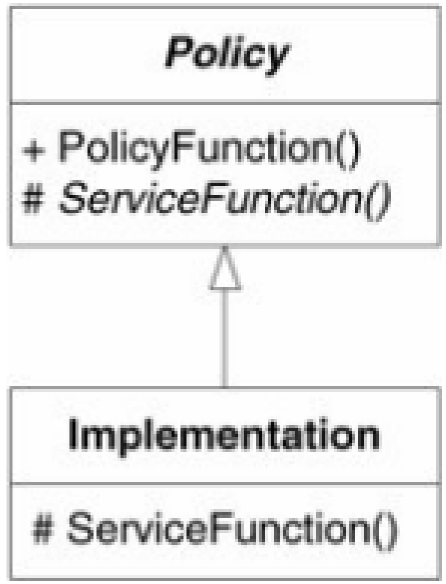
OCP: Example (2)



- Client references Client Interface
  - Unaffected by change of Server class
- Client is open and closed
- Client Interface rather than Server Interface
  - Client depends on the interface it requires, rather than the interface the Server provides
  - If necessary, use Adapter to bridge possible gap

Non scholae sed vitae discimus.

OCP: Example (3) – Template Method Pattern



- *Policy*: Abstract base class
- *Policy* is open and closed
- Implementation extends *Policy,* doesn't modify it
  - Overrides abstract method(s), not concrete method(s) of *Policy*

Non scholae sed vitae discimus.

OCP: Example (4)

- DrawAllShapes()
  - Conforms to OCP

- Behavior can be extended without modification
  - By adding more Shape implementing classes
  - E.g. Triangle

```
public interface Shape
{
  void Draw();
}

public class Square : Shape
{
  public void Draw()
  {
    //draw a square
  }
}

public class Circle : Shape
{
  public void Draw()
  {
    //draw a circle
  }
}

  public void DrawAllShapes(IList shapes)
  {
    foreach(Shape shape in shapes)
      shape.Draw();
  }
```

OCP: neither easy nor cheap

- OCP relies on finding the "right" abstraction
  - no particular abstraction is right for every kind of change
- Abstractions
  - Not easy to find the right one
  - Complicate the design
  - Only justified if changes actually happen

LSP: Liskov Substitution Principle

Subtypes must be substitutable for their base types in all contexts.

- Barbara Liskov, MIT, 1988
  - S is a subtype of T if
    for each object s of type S
    there is an object t of type T
    such that
    for all programs P defined in terms of T
    the behavior of P is unchanged if s is substituted for t

- This notion of subtype is stronger than subclass
  - Stronger means more restrictive

Non scholae sed vitae discimus.

LSP: Example (1)

```
public class Rectangle
{
  private Point topLeft;
  private double width;
  private double height;

  public virtual double Width
  {
    get { return width; }
    set { width = value; }
  }

  public virtual double Height
  {
    get { return height; }
    set { height = value; }
  }
}
```

```
public class Square : Rectangle
{
  public override double Width
  {
    set
    {
      base.Width = value;
      base.Height = value;
    }
  }

  public override double Height
  {
    set
    {
      base.Height = value;
      base.Width = value;
    }
  }
}
```

- Square is a subclass of Rectangle but not a subtype

LSP: Example (2)

```
void g(Rectangle r)
{
  r.Width = 5;
  r.Height = 4;
  if(r.Area() != 20)
    throw new Exception("Bad area!");
}
```

- Every square is a rectangle
  BUT:
- Not every (legal) state change for a rectangle is a (legal) state change for a square
  - In rectangles, the width and height can vary independently
  - In squares, width must always equal the height
- Therefore: LSP violated

Non scholae sed vitae discimus.

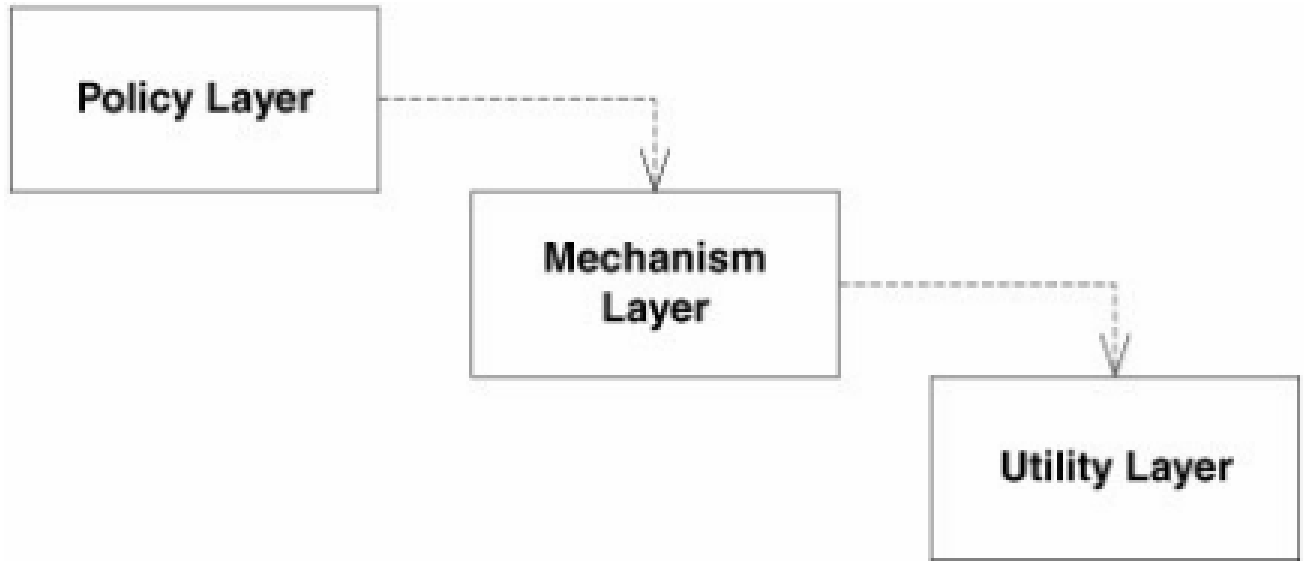
DIP: Dependency Inversion Principle

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

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

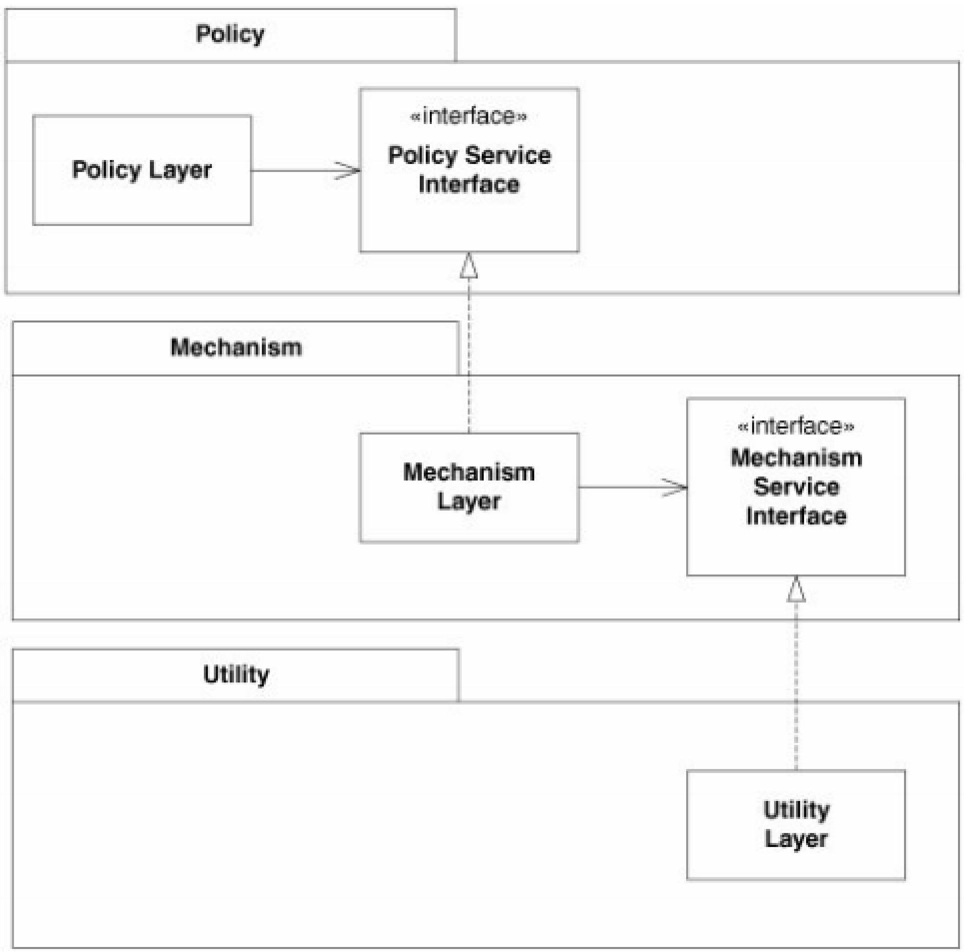
- High-level modules should not directly reference low-level modules
- Rather: low-level modules should support the interface high-level modules require
- Makes high-level modules reusable in other contexts

DIP: Example – Naïve Layers



- Higher layers depend directly on lower layers
  - i.e. methods in higher layers reference objects (of classes) defined in lower layers
- Makes each layer sensitive to changes to layers below
  - Dependency possibly transitive

DIP: Dependency Inversion



- Each lower layer depends on (required) interface of upper layer
- Upper layer "owns" interface
  - Hollywood Principle: "Don't call us; we'll call you."
- Heuristic: Depend on abstractions
  - Variables typed by interfaces not concrete classes
  - Only subclass abstract classes
  - Only override abstract methods

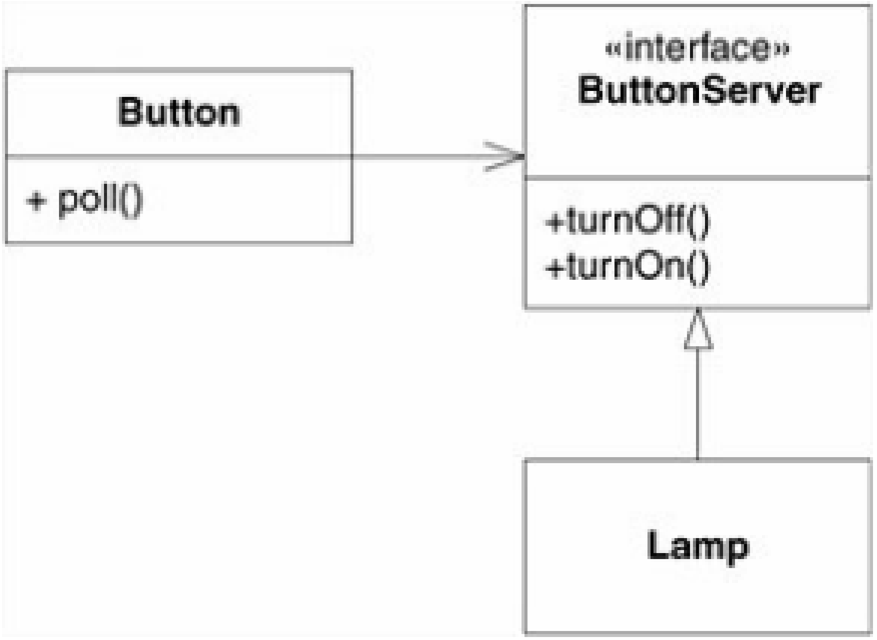
Non scholae sed vitae discimus.

DIP: Example (1)


```
public class Button
{
  private Lamp lamp;
  public void Poll()
  {
    if (/*some condition*/)
      lamp.TurnOn();
  }
}
```

- Button depends directly on Lamp
- Can’t reuse Button in other contexts; e.g. coffee grinder
- Button class sensitive to changes of Lamp class

DIP: Example (2)



- Button can now control device implementing ButtonServer interface
- Button no longer dependent on Lamp
- Button does not need to “own” interface ButtonServer
  - Could replace by more generic SwitchServer

Non scholae sed vitae discimus.

ISP: Interface Segregation Principle

Clients should not be forced to depend on methods they do not use.

- Some objects need “fat”, i.e. non-coherent interfaces.
- Clients should only need to know methods they need.
- Use abstract classes/interfaces
  - Expose different slices of fat interface to different clients

Non scholae sed vitae discimus.

ISP: Exampe (1)

```
public interface Door
{
  void Lock();
  void Unlock();
  bool IsDoorOpen();
}
```

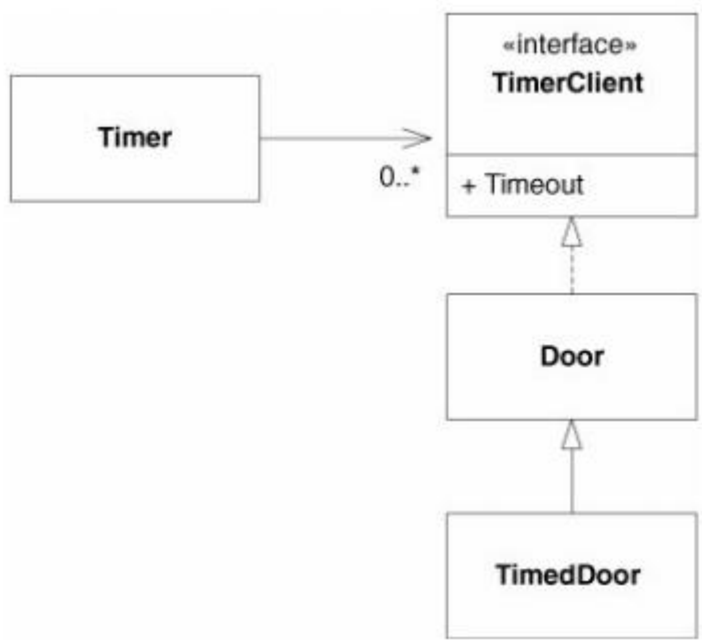
- Suppose a TimedDoor wants to use a Timer to detect if door is left open for too long.
- TimedDoor needs to register with timer

```
public class Timer
{
  public void Register(int timeout, TimerClient client)
  {/*code*/}
}

public interface TimerClient
{
  void TimeOut();
}
```


Non scholae sed vitae discimus.

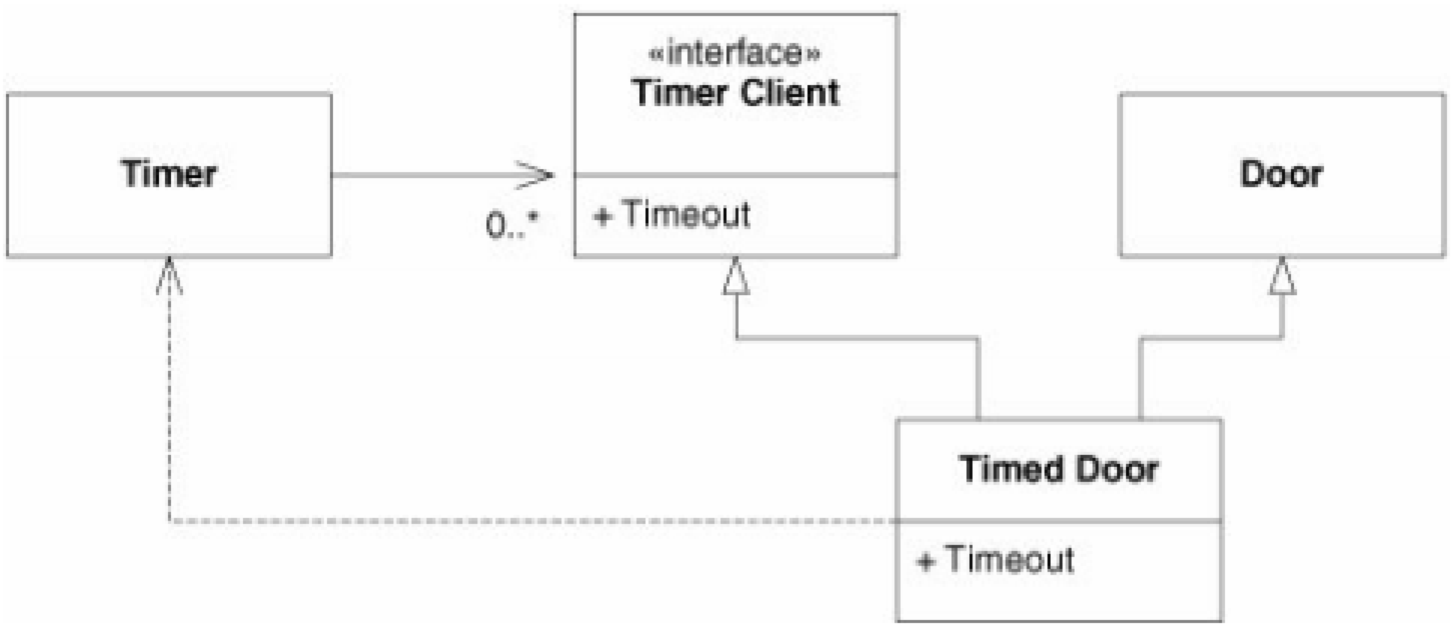
ISP: Example – Interface Pollution



- Possible solution: force Door to implement Timer client
  - Pollutes Door with TimerClient methods
  - Not every door has use for Timer
- Useless complexity (a design smell)

Non scholae sed vitae discimus.

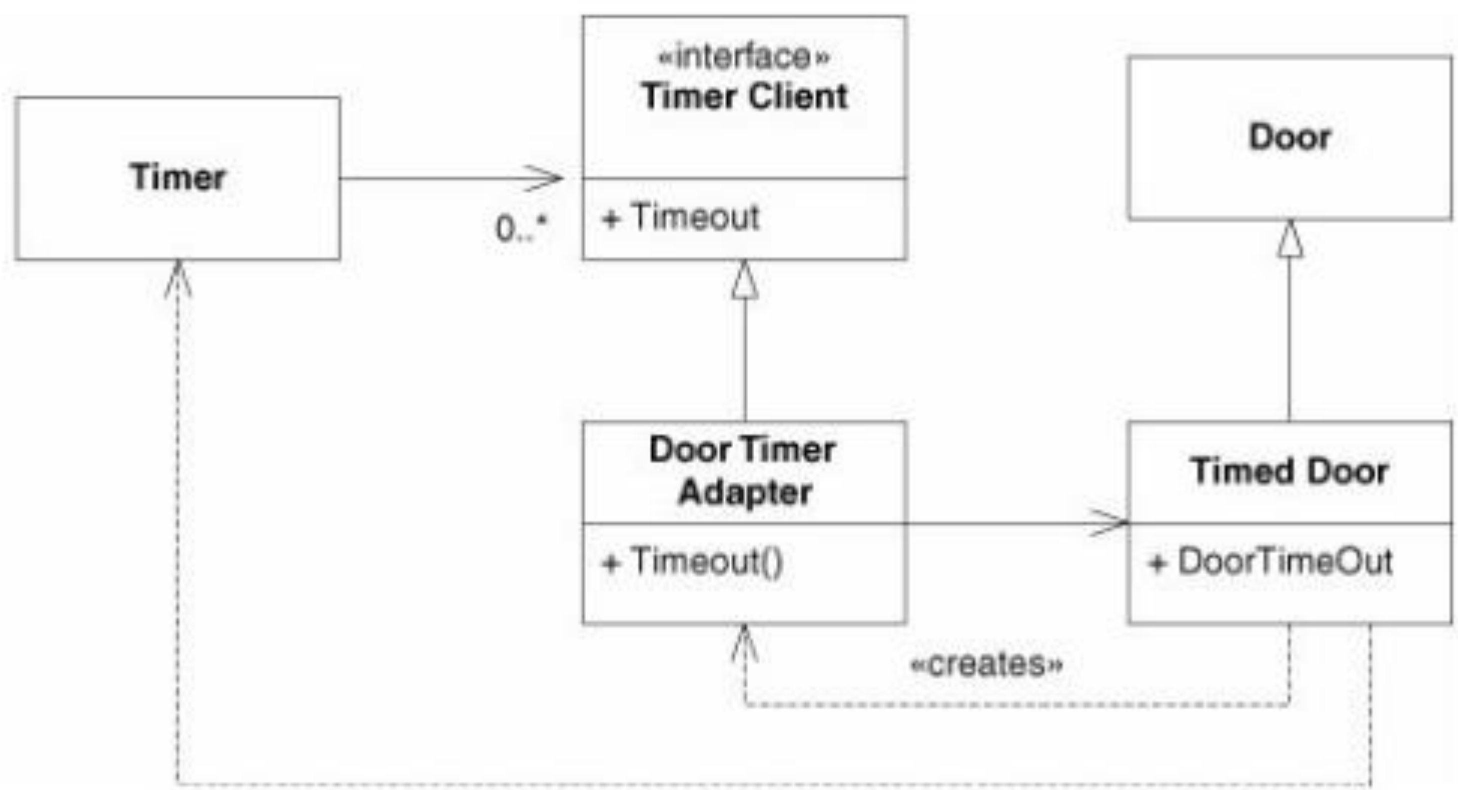
ISP: Separation through multiple inheritance



- Timed Door has a “fat” interface
- Timer Client and Door expose different slices of the Timed Door interface.

Non scholae sed vitae discimus.

ISP: Separation by delegation



- Door interface no longer polluted by Timer Client
- Door Timer Adapter adapts Timer Client Timeout to Timed Door time out method