## Symptoms of Poor Design (revisited)

1. Rigidity
2. Fragility
3. Immobility
4. Viscosity
5. Needless complexity
6. Needless repetition
7. Opacity

## Rigidity

- The design is hard to change
  - changes propagate via dependencies to other modules
  - no continuity in the code
- Management reluctance to change anything becomes the policy
- Telltale sign: 'Huh, it was a lot more complicated than I thought.'

## Fragility

- The design is easy to break
  - changes cause cascading effects to many places
  - the code breaks in unexpected places that have no conceptual relationship with the changed area
  - fixing the problems causes new problems
- Telltale signs
  - some modules are constantly on the bug list
  - time is used finding bugs, not fixing them
  - programmers are reluctant to change anything in the code

## Immobility

- The design is hard to reuse
  - the code is so tangled that it is impossible to reuse anything
- Telltale sign: a module could be reused but the effort and risk of separating it from the original environment is too high

## Viscosity

- Viscosity of the software
  - changes or additions are easier to implement by doing the wrong thing
- Viscosity of the environment
  - the development environment is slow and inefficient
  - high compile times, long feedback time in testing, laborious integration in a multi-team project
- Telltale signs
  - when a change is needed, you are tempted to hack rather than to preserve the original design
  - you are reluctant to execute a fast feedback loop and instead tend to code larger pieces

## Needless Complexity

- Design contains elements that are not currently useful
  - too much anticipation of future needs
  - developers try to protect themselves against probable future changes
  - agile principles state that you should never anticipate future needs
- Extra complexity is needed *only* when designing an application framework or customizable component
- Telltale sign: investing in uncertainty

## Needless Repetition

- The same code appears over and over again, in slightly different forms
  - developers are missing an abstraction
  - bugs found in a repeating unit have to be fixed in every repetition
- Telltale sign: overuse of cut-and-paste

## Opacity

- The tendency of a module to become more difficult to understand
  - every module gets more opaque over time
  - a constant effort is needed to keep the code readable
    - easy to understand
    - communicates its design
- Telltale sign: you are reluctant to fix somebody else’s code – or even your own!

## Five Principles to Avoid the Symptoms

1. Single-Responsibility Principle
2. Open–Closed Principle
3. Liskov Substitution Principle
4. Depency-Inversion Principle
5. Interface-Segregation Principle

## SRP: The Single-Responsibility Principle

**A CLASS SHOULD HAVE ONLY ONE REASON TO CHANGE.**

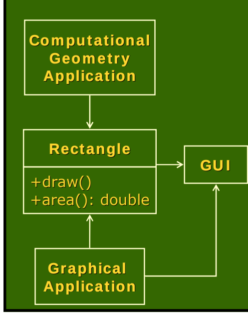
- Cohesion: how good a reason the elements of a module have to be in the same module
- Cohesion and SRP: the forces that cause the module to change

## Responsibility

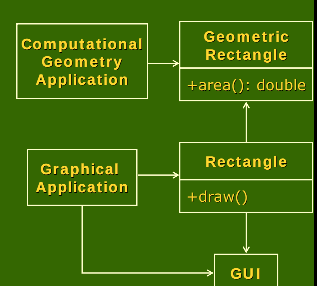
- Rationale behind SRP
  - changes in requirements
    → changes in class responsibilities
  - a ‘cohesive’ responsibility is a single axis of chance
    → a class should have only one responsibility
  - responsibility = a reason to change
- Violation of SRP causes spurious transitive dependencies between modules that are hard to anticipate → fragility
- Separating the responsibilities into interfaces decouples them as far as rest of the application is concerned

## SRP Example: Rectangle

More than one responsibility




Separated responsibilities



## OCP: The Open–Closed Principle

> Software entities should be open for extension, but closed for modification.
>
> – Bertrand Meyer

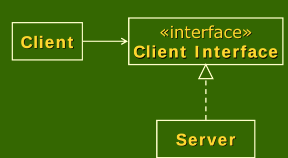
- 'Open for extension': the behaviour of a module can be extended with new behaviours to satisfy the changing requirements
- 'Closed for modification': extending the module must not result in changes to the source or even binary code of the module

## OCP (cont'd)

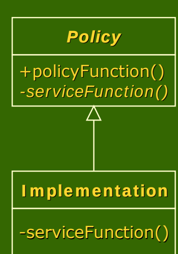
- Reduces rigidity
  - a change does not cause a cascade of related changes in dependent modules
- Changing the module without changing its source code – a contradiction?!
- How to avoid dependency on a concrete class?
  - abstraction
  - dynamic binding

## Basic OCP Designs

Strategy




Template Method




## Strategic Closure

- Conforming to the OCP is expensive, since it can incur needless complexity
- All changes cannot be anticipated
  - apply OCP to the most obvious changes
- Otherwise: 'Fool me once, shame on you. Fool me twice, shame on me.'
  - once a change has occurred, it is more probable that a similar kind of change will occur later
  - apply OCP when it is needed for the first time
- A good strategy: stimulate early changes
  - fast iterations
  - constant feedback

## OCP: Simple Heuristics

- Make all object data private
  - changes to public data are always at risk to 'open' the module
  - all clients of a module with public data members are open to one misbehaving module
  - errors can be difficult to find and fixes may cause errors elsewhere
- No global variables
  - it is impossible to close a module against a global variable

## LSP: The Liskov Substitution Principle

> Subtypes must be substitutable for their base types.
>
> – Barbara Liskov

- Functions that refer to base classes must be able to use objects of both existing and future derived classes without knowing it
- Inheritance must be used in a way that any property proved about supertype objects also holds for the subtype objects

## LSP and OCP

- LSP is motived by OCP (at least partly)
  - abstraction and polymorphism allows us to achieve OCP, but how to use them?
  - key mechanism in statically typed languages: inheritance
- LSP restricts the use of inheritance in a way that OCP holds
- LSP addresses the questions of
  - what are the inheritance hierarchies that give designs conforming to OCP
  - what are the common mistakes we make with inheritance regarding OCP?
- Violation of LSP is a latent violation of OCP

## Example: Inheritance Has Its Limits

```
public abstract class Bird {
    public abstract void fly();
}

public class Parrot extends Bird {
    public void fly()   { /* implementation */ }
    public void speak() { /* implementation */ }
}

public class Penguin extends Bird {
    public void fly() {
        throw new UnsupportedOperationException();
    }
}
```

## Example (cont'd)

```
public static void playWith(Bird bird) {
    bird.fly();
}

Parrot myPet;
playWith(myPet); // myPet "is-a" bird and can fly()

Penguin myOtherPet;
playWith(myOtherPet); // myOtherPet "is-a" bird
                      // and cannot fly()?!
```

## Example (cont'd)

- What went wrong?
  - we did not model 'Penguins cannot fly'
  - we modelled 'Penguins may fly, but if they try it is an error'
- The design fails LSP
  - a property assumed by the client about the base type does not hold for the subtype
  - Penguin is not a subtype of Bird
- Subtypes must respect what the client of the base class can reasonably expect about the base class
  - but how can we anticipate what some client will expect?

## Design by Contract

- A class declares its behaviour
  - requirements (preconditions) that must be fulfilled
  - promises (postconditions) that will hold afterwards
- This forms a *contract* between the class and a client using its services
  - tells explicitly what the client may expect
- B. Mayer (1988): When redefining a method in a derived (or inherited) class
  - the precondition can be replaced only by a weaker one
  - the postcondition can be replaced only by a stronger one
- A derived class should require no more and provide no less than the base class

## LSP: Simple Heuristic

- Telltale signs of LSP violation:
  - degenerate functions in derived classes (i.e. overriding a base-class method with a method that does nothing)
  - throwing exceptions from derived classes
- Solution 1: inverse the inheritance relation
  - if the base class has only additional behaviour
- Solution 2: extract common a base class
  - if both initial and derived classes have different behaviors
  - penguins → Bird, FlyingBird, Penguin
- Sometimes it is not possible to edit the base class
  - example: Java Collections Hierarchy

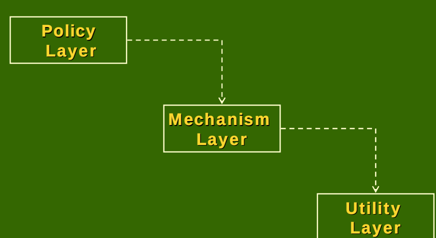
## DIP: The Dependency-Inversion Principle

> HIGH-LEVEL MODULES SHOULD NOT DEPEND ON LOW-LEVEL MODULES. BOTH SHOULD DEPEND ON ABSTRACTIONS.
>
> ABSTRACTIONS SHOULD NOT DEPEND ON DETAILS. DETAILS SHOULD DEPEND ON ABSTRACTIONS.
>
> – ROBERT MARTIN

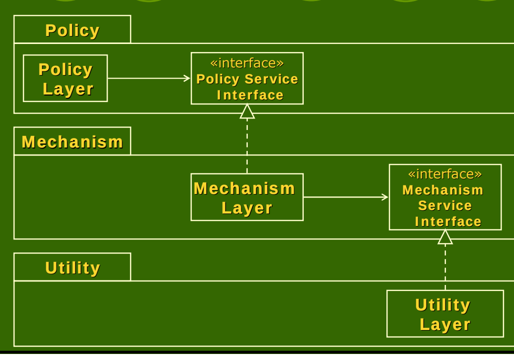
## DIP (cont'd)

- Modules with detailed implementations are not depended upon, but depend themselves upon abstractions
- High-level modules contain the important business model of the application, the policy
  - independent of details
  - should be the focus of reuse
  - greatest benefits are achievable here
- Results from the rigorous use of LSP and OCP
  - OCP states the goal
  - LSP enables it
  - DIP shows the mechanism to achieve the goal

## Example: Naïve Layering Scheme




## Example: Inverted Layers




## Design to an Interface

- Rationale
  - abstract classes/interfaces tend to change less frequently
  - abstractions are 'hinge points' where it is easier to extend/modify
  - no need to modify classes/interfaces that represent the abstraction
- All relationships should terminate to an abstract class or interface
  - no variable should refer to a concrete class
    - use inheritance to avoid direct bindings to concrete classes
  - no class should derive from a concrete class
    - concrete classes tend to be volatile
  - no method should override an implemented method of any of its base classes
- Exceptions
  - some classes are very unlikely to change → a little benefit in inserting an abstraction layer
    - you can depend on a concrete class that is not volatile (e.g. String class)
  - a module that creates objects automatically depends on them

## ISP: The Interface-Segregation Principle

> CLIENTS SHOULD NOT BE FORCED TO DEPEND UPON METHODS THAT THEY DO NOT USE.

- Many client specific interfaces are better than one general purpose interface
  - no 'fat' interfaces
  - no non cohesive interfaces
- Related to SRP

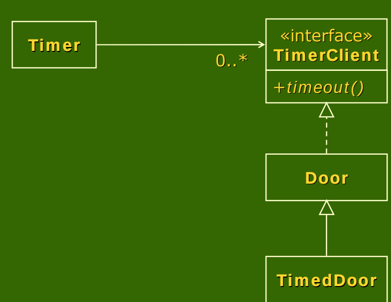
## Fat Interfaces

- Fat interface = general purpose interface ≠ client-specific interface
  - can cause bizarre couplings between its clients
  - when one client forces a change, all other clients are affected
- Break a fat interface into many separate interfaces
  - targeted to a single client or a group of clients
  - clients depend only on the methods they use (and not on other clients' needs)
  - impact of changes to one interface are not as big
  - probability of a change reduces
  - no interface pollution

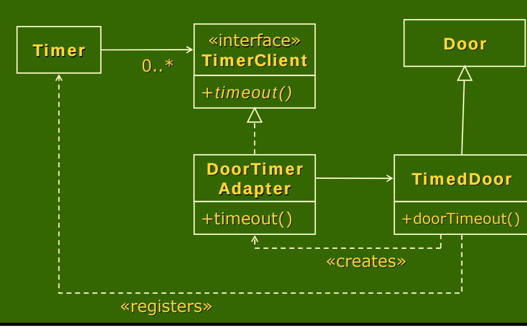
## Example: Door and Timer

```
public class Door {
    public void lock() { /* implementation */ }
    public void unlock() { /* implementation */ }
    public boolean isOpen() { /* implementation */ }
}

public class Timer {
    public void register(int timeout,
                         TimerClient client) {
        /* implementation */
}   }

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

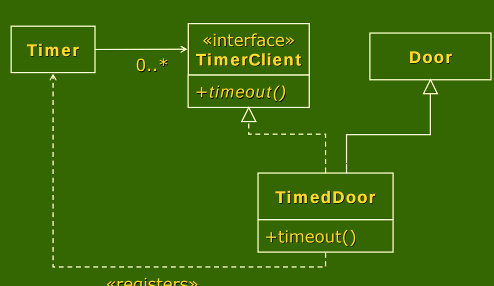
## Example: Timer Client at Top of Hierarchy




## Example: Separation Through Delegation




## Example: Separation Through Multiple Inheritence




## Role-Based Interface Design

- Interfaces are designed from the viewpoint of the service user, not the service provider
  - clients own the interfaces
- Interfaces should represent roles that clients take when using the services of a class or component
- Classes implement many interfaces, interfaces are implemented by many classes
  - example: flying birds (as well as bats) implement interface FlyingCreature, but penguins do not
- Version control by adding new interfaces for clients requiring new services → less viscosity

## Reading for the Next Week

- Section 3: The Payroll Case Study
  - Chapter 13: COMMAND and ACTIVE OBJECT
  - Chapter 14: TEMPLATE METHOD & STRATEGY: Inheritance vs. Delegation
  - Chapter 15: FACADE and MEDIATOR
  - Chapter 16: SINGLETON and MONOSTATE
  - Chapter 17: NULL OBJECT
  - Chapter 18: The Payroll Case Study: Iteration One Begins
  - Chapter 19: The Payroll Case Study: Implementation