Object-Oriented DESIGN

CSCI-4448 - Boese

University of Colorado **Boulder**

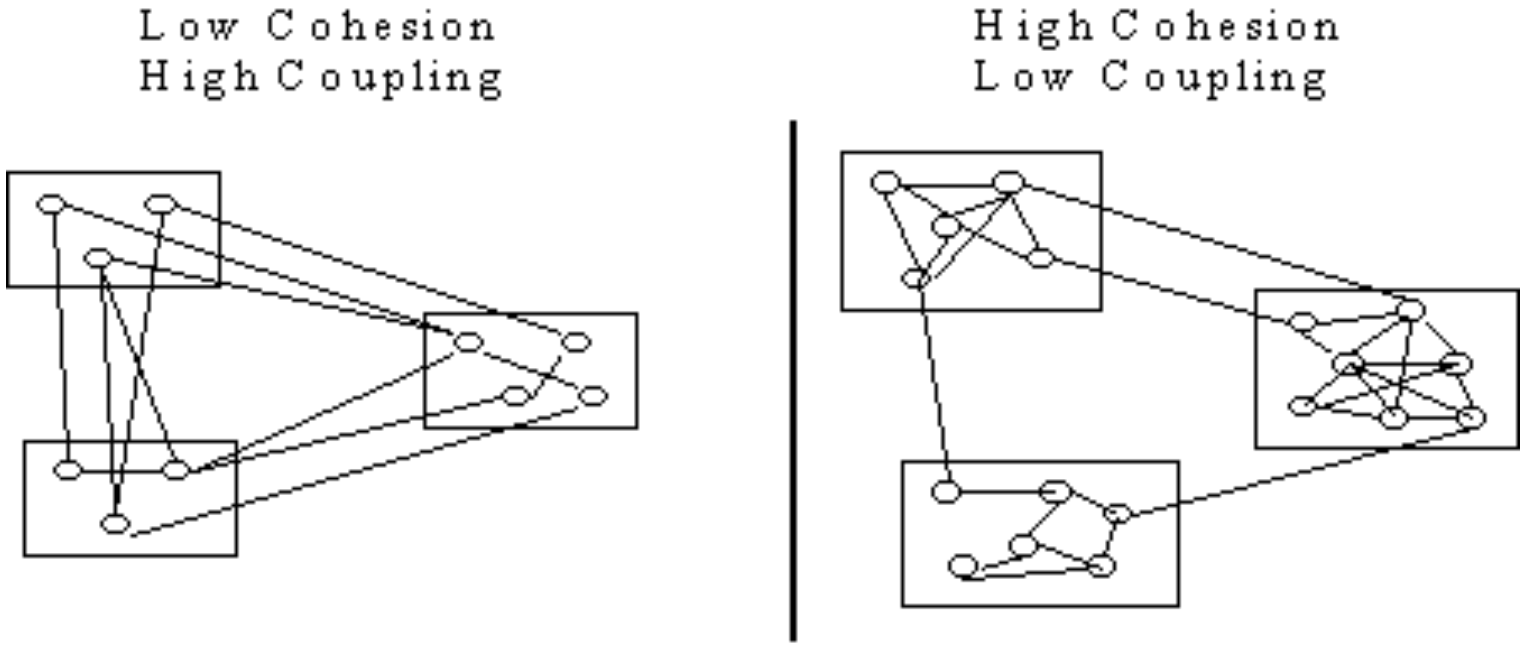
Objectives

- Understand and be able to apply OOD concepts including:
  - Cohesion vs Coupling
  - SOLID Principles
    - *Single-Responsibility Principle*
    - *Open-closed Principle*
    - *Liskov Substitution Principle*
    - *Interface Segregation Principle*
    - *Dependency Inversion Principle*
  - Composition vs Inheritance

Cohesion vs Coupling

Cohesion vs Coupling

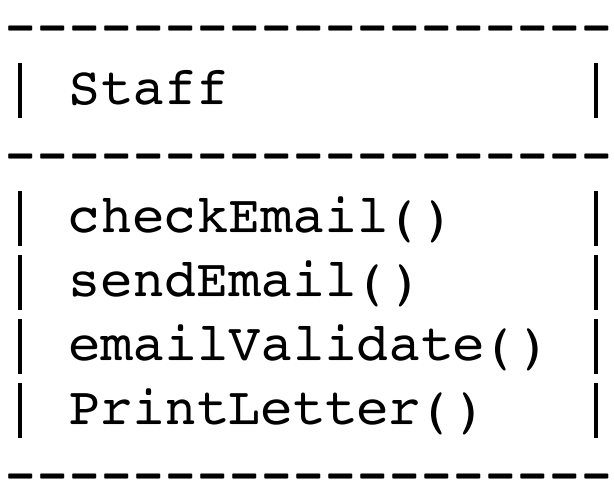
- Goal:
  - ***High cohesion*** *within* modules
  - ***Low coupling*** *between* modules



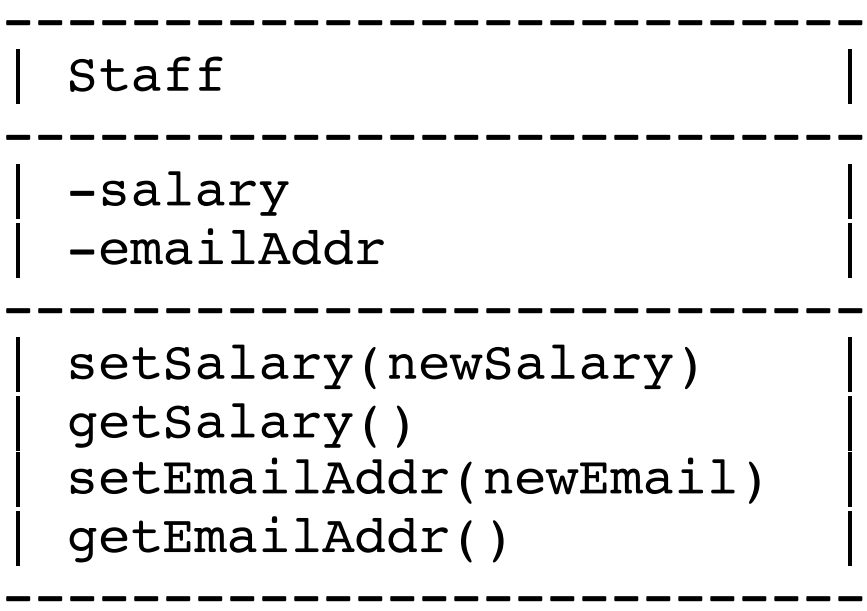
Cohesion vs Coupling

- **Cohesion** refers to how related the elements in a class/ module/framework are
  - **Low cohesion:** class performs many unrelated tasks
  - **Strong cohesion:** class focused on what it should be doing (single purpose)

Low Cohesion



Strong Cohesion



Strong Cohesion

- **Strong cohesion** in classes is desired, weak cohesion should be avoided
  - Consider a change in the requirements of the software – some object(s) is/are responsible for that requirement
  - If the classes have strong cohesion, then only one (or a few) classes need modification
  - If the classes have weak cohesion, then many more will need to be modified to account for the change

How to achieve high Cohesion

- High cohesion can be achieved if most of the interaction is within subsystems, rather than across subsystem boundaries
- Questions to ask:
  - Does one subsystem always call another one for a specific service?
    - *Yes: Consider moving them together into the same subsystem.*
  - Which of the subsystems call each other for services?
    - *Can this be avoided by restructuring the subsystems or changing the subsystem interface?*
  - Can the subsystems even be hierarchically ordered (in layers)?

Coupling

- **Coupling** refers to how strongly related to two or more classes (or methods, routines, etc.) are.
  - Tight coupling
    - *Implies a strong relationship between two objects.*
    - *Depend on many other classes to perform a task.*
    - *Changes to one subsystem will have high impact on the other subsystem*
  - Loose coupling
    - *Implies a weaker relationship between objects.*
    - *Have few dependencies on other classes.*
    - *A change in one subsystem does not affect any other subsystem.*

Cohesion vs Coupling

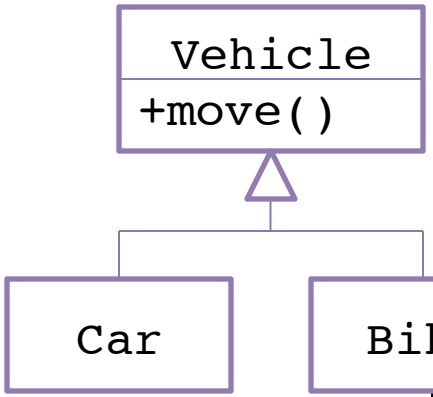
Tight Coupling

```
class Traveler
{
   Car c=new Car();
   void startJourney()
   {
     c.move();
   }
}

class Car
{
  void move()
  {
    // logic...
  }
}
```

What if need to change from Car to Bike?

Loose Coupling



Vehicle is an *abstraction*

```
class Traveler
{
   Vehicle v;
   public void setV(Vehicle v)
   {
     this.v = v;
   }

   void startJourney()
   {
     v.move();
   }
}
```

- **High coupling**:
  - iPods are a good example of tight coupling: once the battery dies you might as well buy a new iPod because the battery is soldered fixed and won't come loose, thus making replacing very expensive. A loosely coupled player would allow effortlessly changing the battery.

How to achieve Low Coupling

- Low coupling can be achieved if a calling class does not need to know anything about the internals of the called class (Principle of information hiding)
- Questions to ask:
  - Does the calling class really have to know any attributes of classes in the lower layers?
  - Is it possible that the calling class calls only operations of the lower level classes?

Cohesion & Coupling

- Cohesion and coupling are typically complimentary

OOD: SOLID Principles

OOD: SOLID Principles

- **S** = Single Responsibility Principle
- **O** = Opened Closed Principle
- **L** = Liskov Substitution Principle
- **I** = Interface Segregation Principle
- **D** = Dependency Inversion Principle

S = Single Responsibility Principle
O = Opened Closed Principle
L = Liskov Substitution Principle
I = Interface Segregation Principle
D = Dependency Inversion Principle

Single-Responsibility Principle

Single Responsibility Principle

Single Responsibility Principle

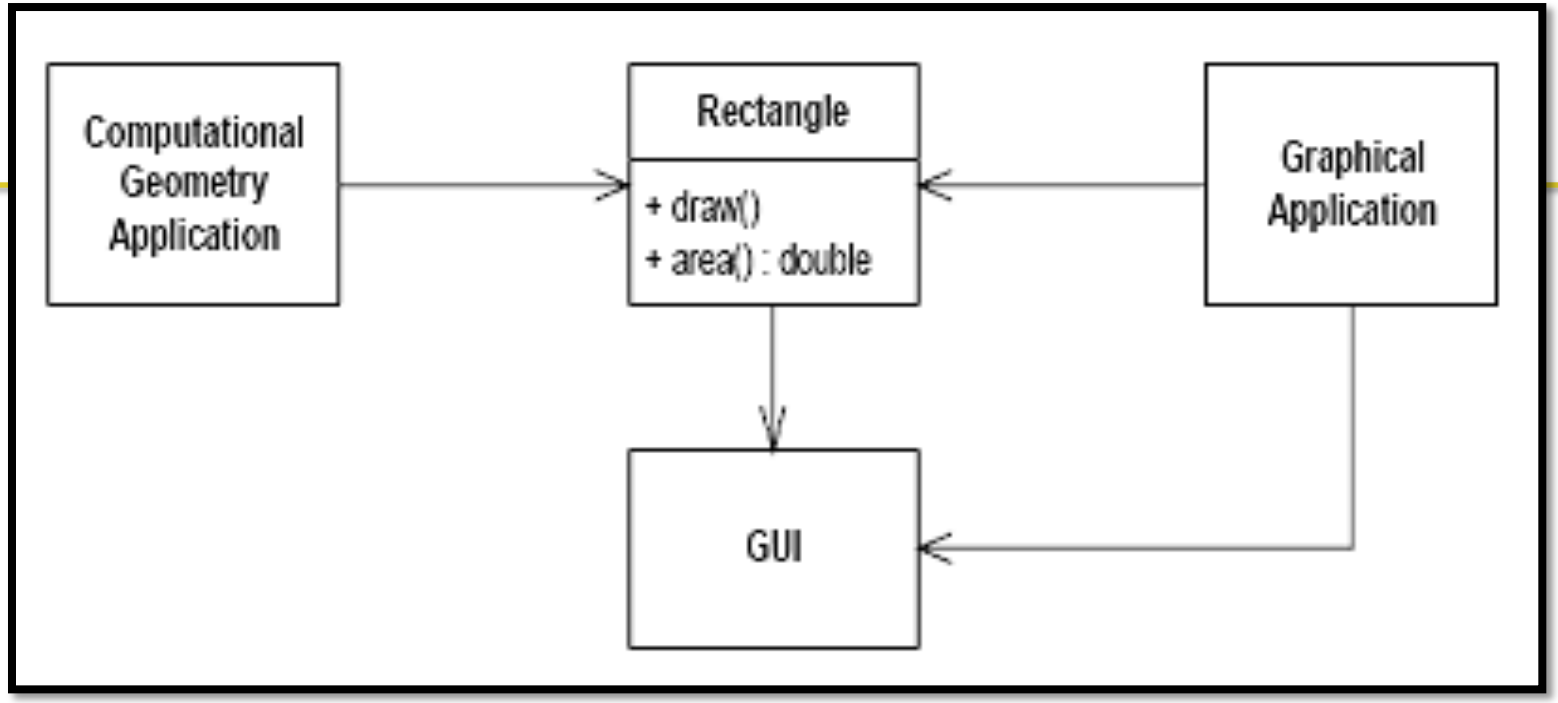
"Just because you can implement all the features in a single device, you shouldn't".
Why?
Because it adds a lot of manageability problems for you in the long run.

Single Responsibility Principle

- If you have a class that has
  - more than one reason to change
  - or has more than one overall responsibility

  you need to split the class into multiple classes based upon their responsibility.

- Why split?
  - Each responsibility is an axis of change.
  - Code becomes coupled if classes have more than one responsibility.



- Rectangle class does the following:
  - Calculates the value of the rectangular area
  - Renders the rectangle in the UI
- And, two applications are using this Rectangle class:
  - A computational geometry application uses this class to calculate the area
  - A graphical application uses this class to draw a rectangle in the UI
- This is violating the SRP (Single Responsibility Principle)!

Single Responsibility Principle

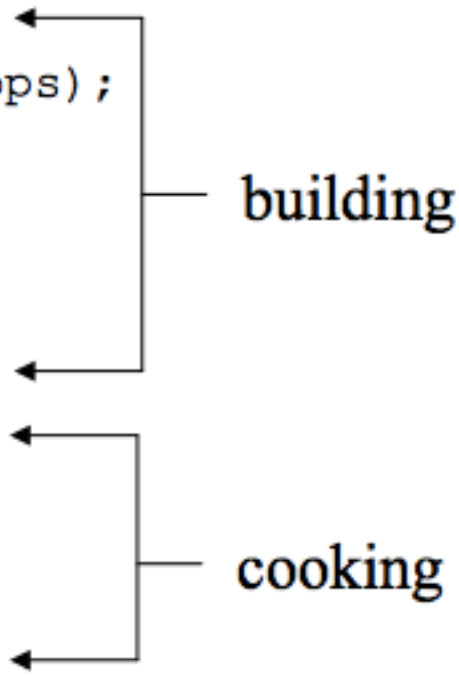
- Fix:

Separate the responsibilities into two different classes, such as:

  - **`Rectangle`**: define the `area()` method.
  - **`RectangleUI`**: inherit the `Rectangle` class and define the `Draw()` method.

Single Responsibility Principle

```
public interface Pizza {
  public List<Topping> getToppings();
  public void setToppings(List<Topping> tops);
  public PizzaSize getSize();
  public void setSize(PizzaSize size);
  public PizzaCrust getCrust();
  public void setCrust(PizzaCrust crust);
  pub void cook(int temp, int minutes);
  public TasteRating rateTaste();
  public SmellRating rateSmell();
  public boolean isBurnt();
}
```



S = Single Responsibility Principle
O = Opened Closed Principle
L = Liskov Substitution Principle
I = Interface Segregation Principle
D = Dependency Inversion Principle

Open-Closed Principle


OPEN CLOSED PRINCIPLE
Open Chest Surgery Is Not Needed When Putting On A Coat

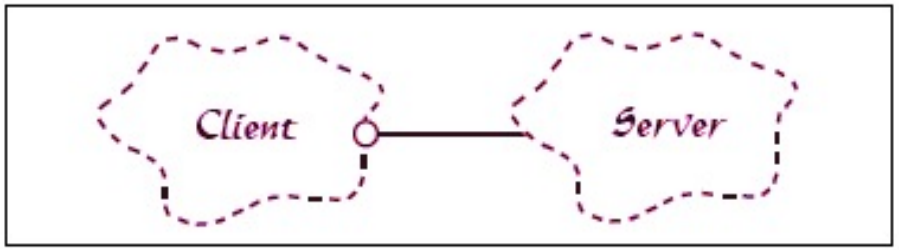
Open-Closed Principle

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

Open-Closed Principle

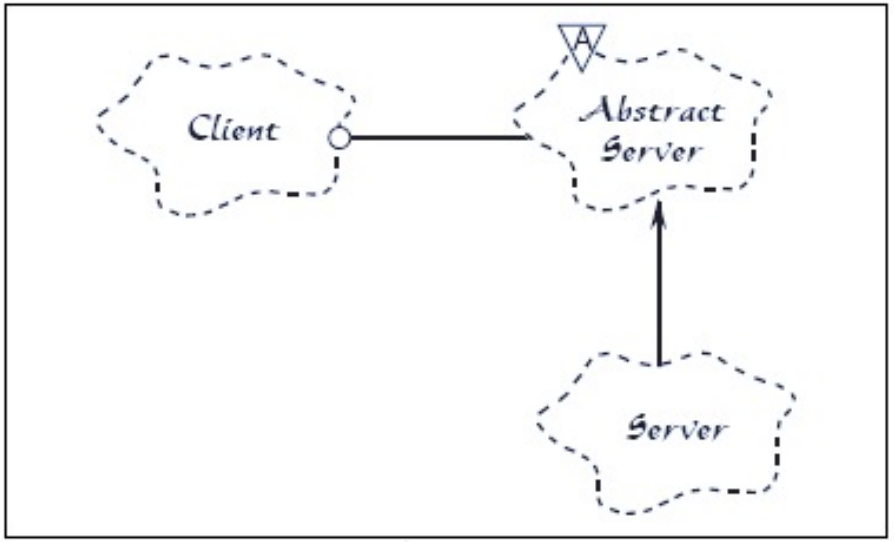
Able to *extend* a class's behavior *without modifying* it.

Violates Open-Closed



the Client and Server classes are both concrete.
So if for any reason the server implementation is changed, the client also needs a change.

Supports Open-Closed



if for any reason the Server implementation is changed, the Client is not likely to require any change.

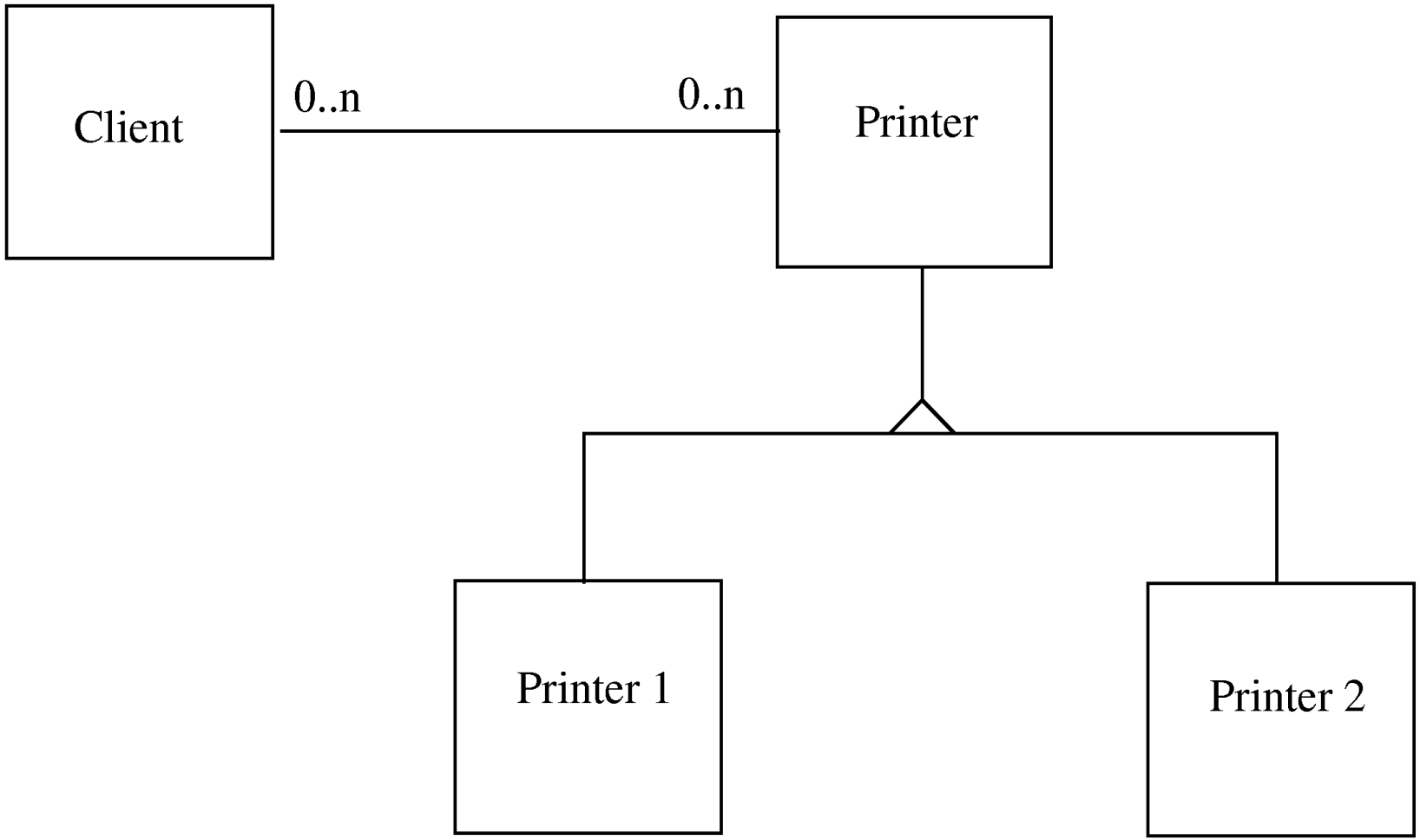
Open-closed Principle…

- In OO this principle is satisfied by using **inheritance** and **polymorphism**
  - Inheritance allows creating a new class to extend behavior without changing the original class

Example..

- Example: client object interacts with a printer object for printing
- Client directly calls methods on Printer1
- If another printer is to be allowed
  - A new class Printer2 will be created
  - But the client will have to be changed if it wants to use Printer 2
- Alternative approach
  - Have Printer1 a subclass of a general Printer
  - For modification, add another subclass Printer 2
  - Client does not need to be changed

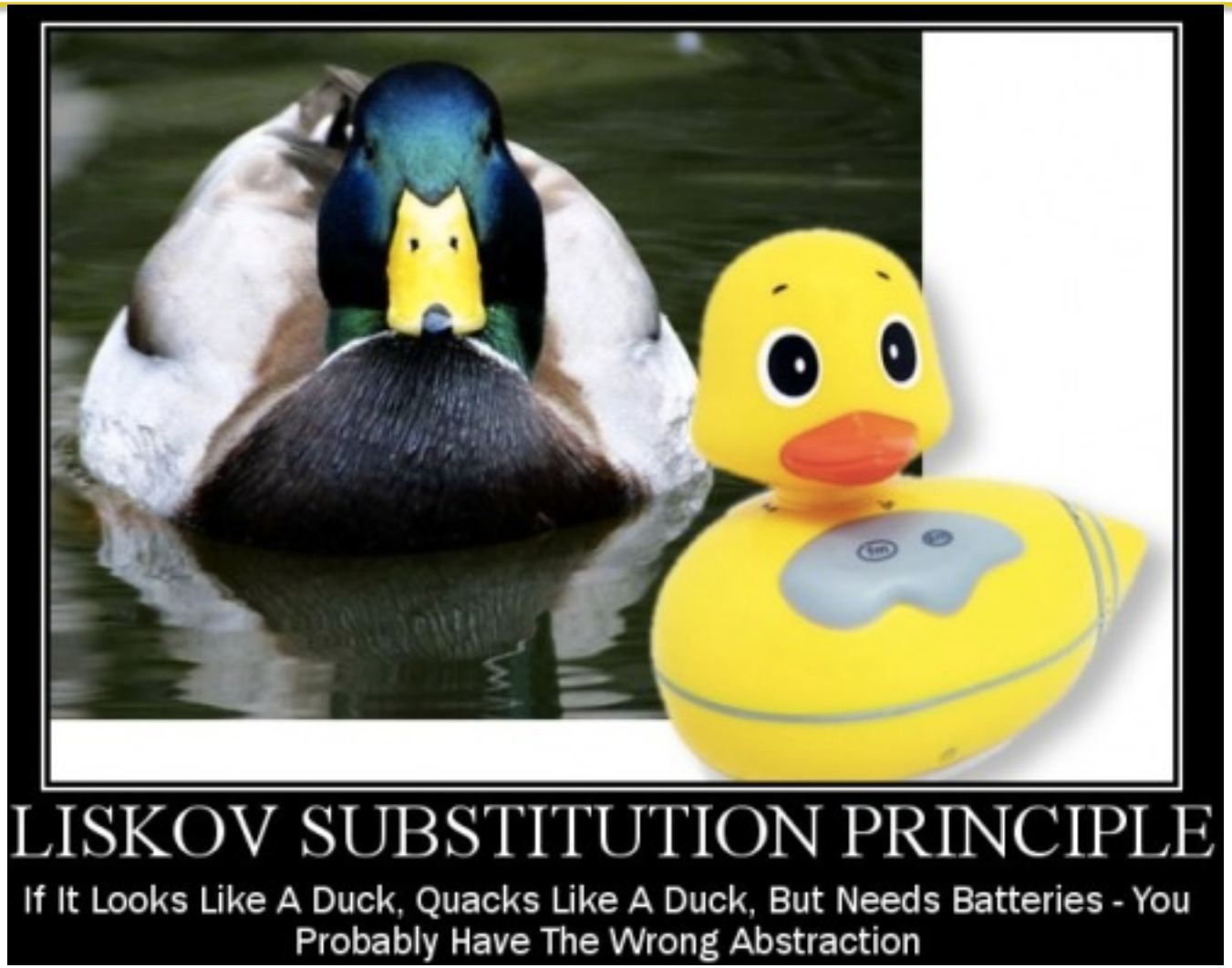
Example…



S = Single Responsibility Principle
O = Opened Closed Principle
L = Liskov Substitution Principle
I = Interface Segregation Principle
D = Dependency Inversion Principle

Liskov Substitution Principle

Liskov Substitution Principle



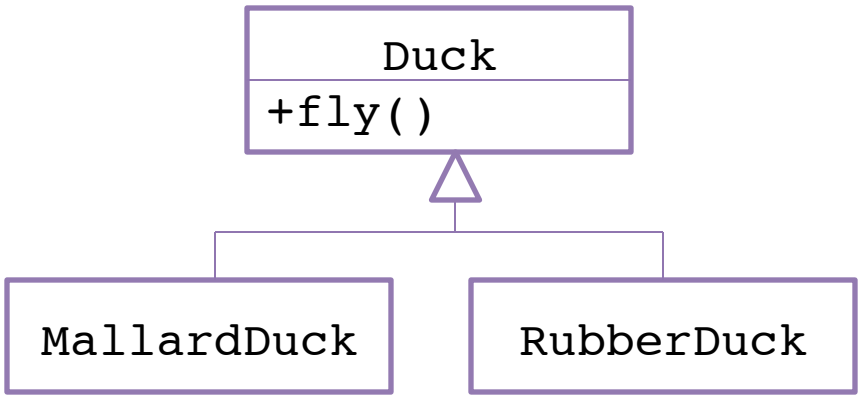
Liskov Substitution Principle

"Subtypes
must be
substitutable
for their
base types."

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

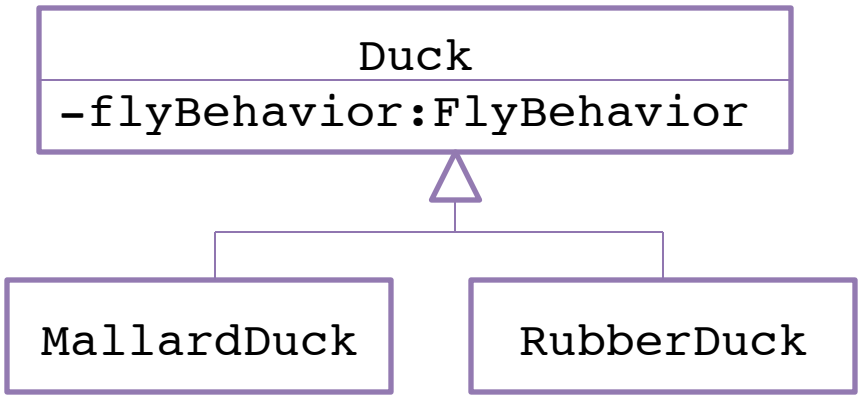
Liskov Substitution Principle

Violates LSP



Rubber ducks can't fly!
But it *is* a 'duck'!

Supports LSP



Separate flying behavior to a separate class or interface. RubberDuck's fly behavior is none

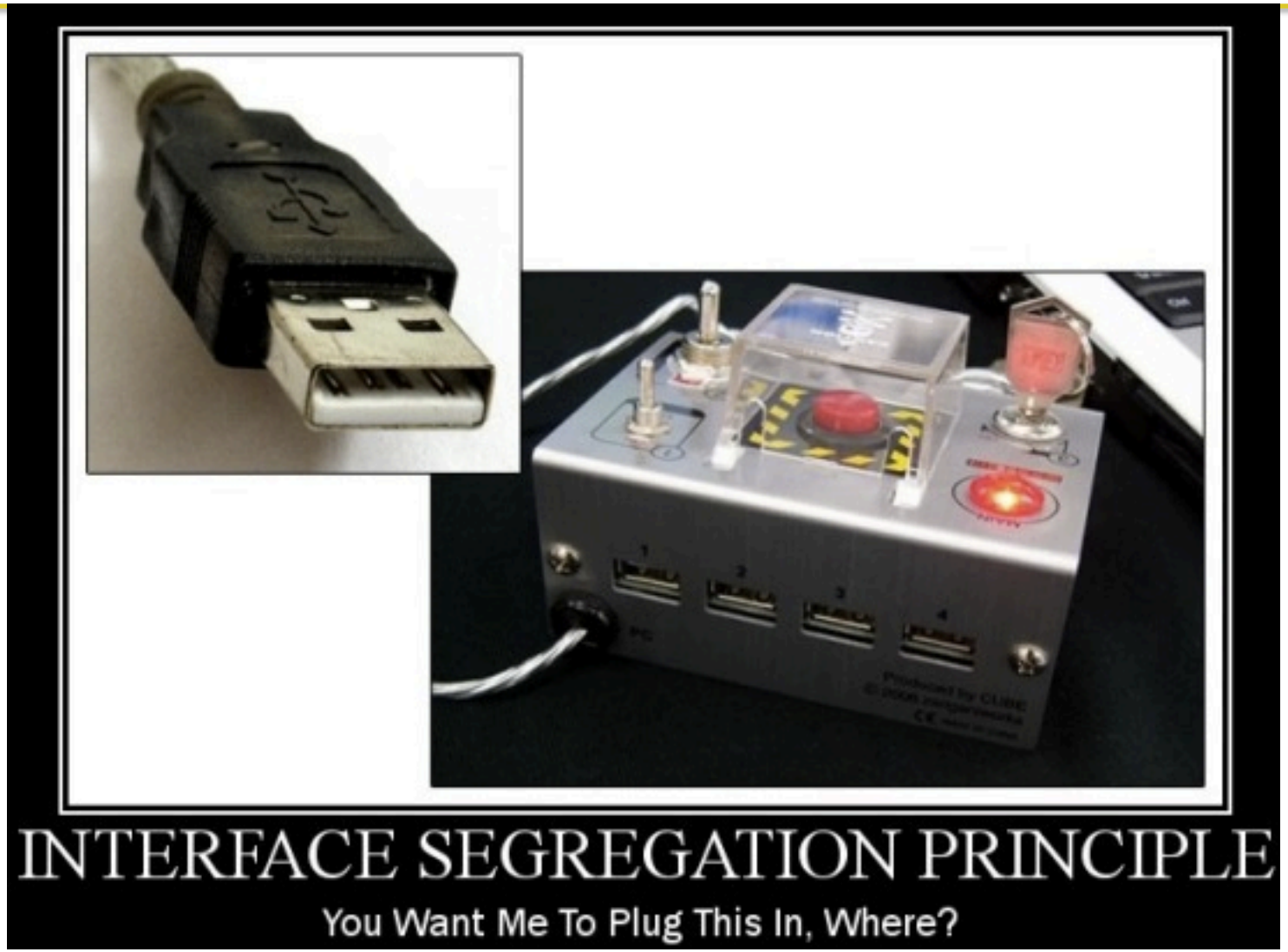
Liskov's Substitution Principle

- Principle: Program using object O1 of base class C should remain unchanged if O1 is replaced by an object of a subclass of C
- If hierarchies follow this principle, the open-closed principle gets supported

S = Single Responsibility Principle
O = Opened Closed Principle
L = Liskov Substitution Principle
I = Interface Segregation Principle
D = Dependency Inversion Principle

Interface Segregation Principle

Interface Segregation Principle



Interface Segregation Principle

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

Interface Segregation Principle

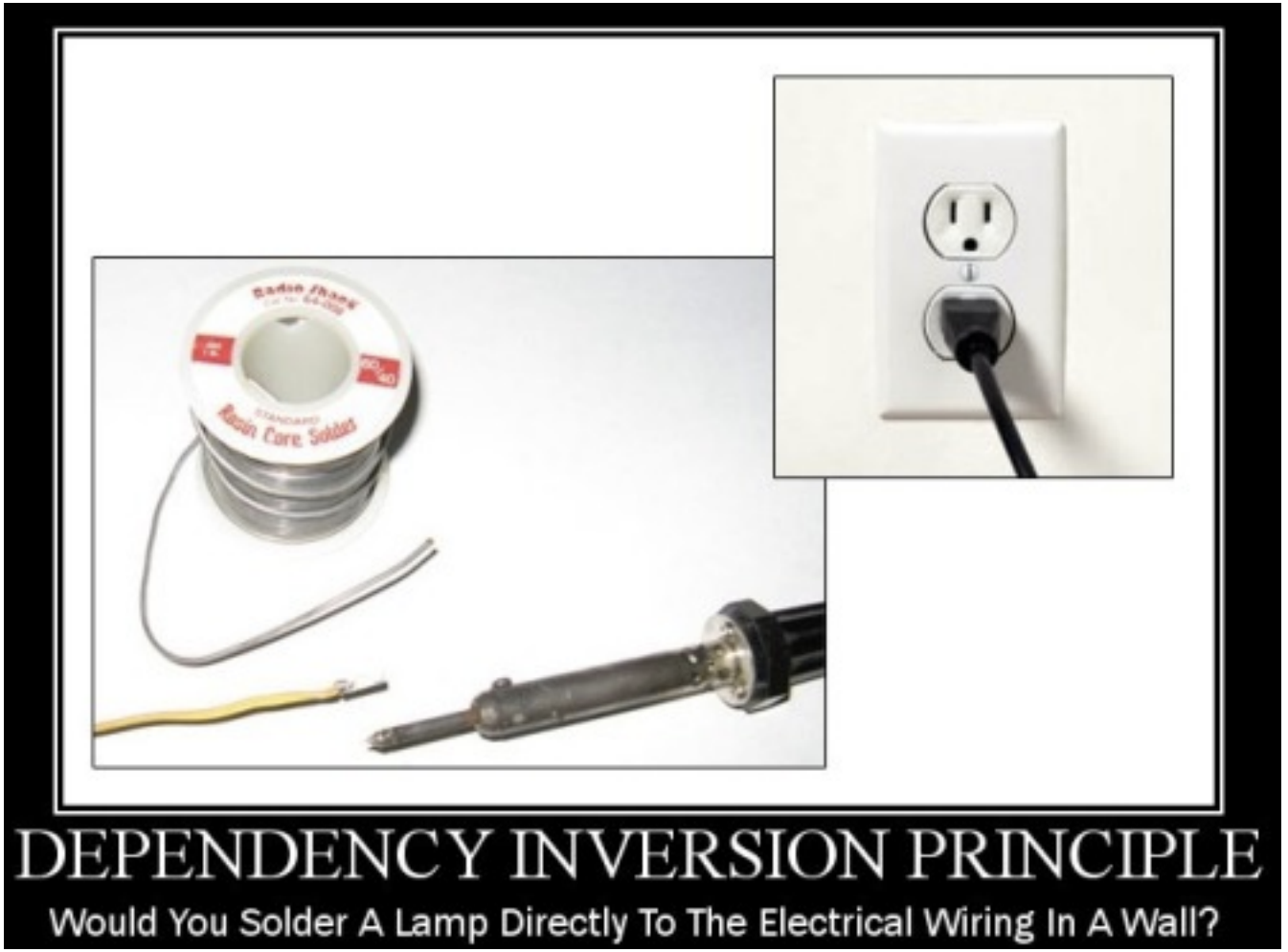
- One sign that you've violated ISP: you have to write lots of do-nothing methods in derived classes
  - Java's old graphics style:
    implement ActionListener required you to implement all these:

Modifier and Type	Method and Description
`void`	**`mouseClicked(MouseEvent e)`** Invoked when the mouse button has been clicked (pressed and released) on a component.
`void`	**`mouseEntered(MouseEvent e)`** Invoked when the mouse enters a component.
`void`	**`mouseExited(MouseEvent e)`** Invoked when the mouse exits a component.
`void`	**`mousePressed(MouseEvent e)`** Invoked when a mouse button has been pressed on a component.
`void`	**`mouseReleased(MouseEvent e)`** Invoked when a mouse button has been released on a component.

S = Single Responsibility Principle
O = Opened Closed Principle
L = Liskov Substitution Principle
I = Interface Segregation Principle
D = Dependency Inversion Principle

Dependency Inversion Principle

Dependency Inversion Principle



Dependency Inversion Principle

"High level modules
should not depend upon
low level modules.
Rather,
both should depend upon
abstractions."

"pluggable nature"

Dependency Inversion Principle

Making an EBookReader class to use PDFBook class is a violation of DIP

- because it requires to change the EBookReader class to read other types of e-books.

- A better design is to
  - let EBookReader use an interface EBook
  - let PDFBook and other types of e-book classes implement EBook.
  - Now adding or changing e-book classes will not require any change to EBookReader class.

Dependency Inversion Principle

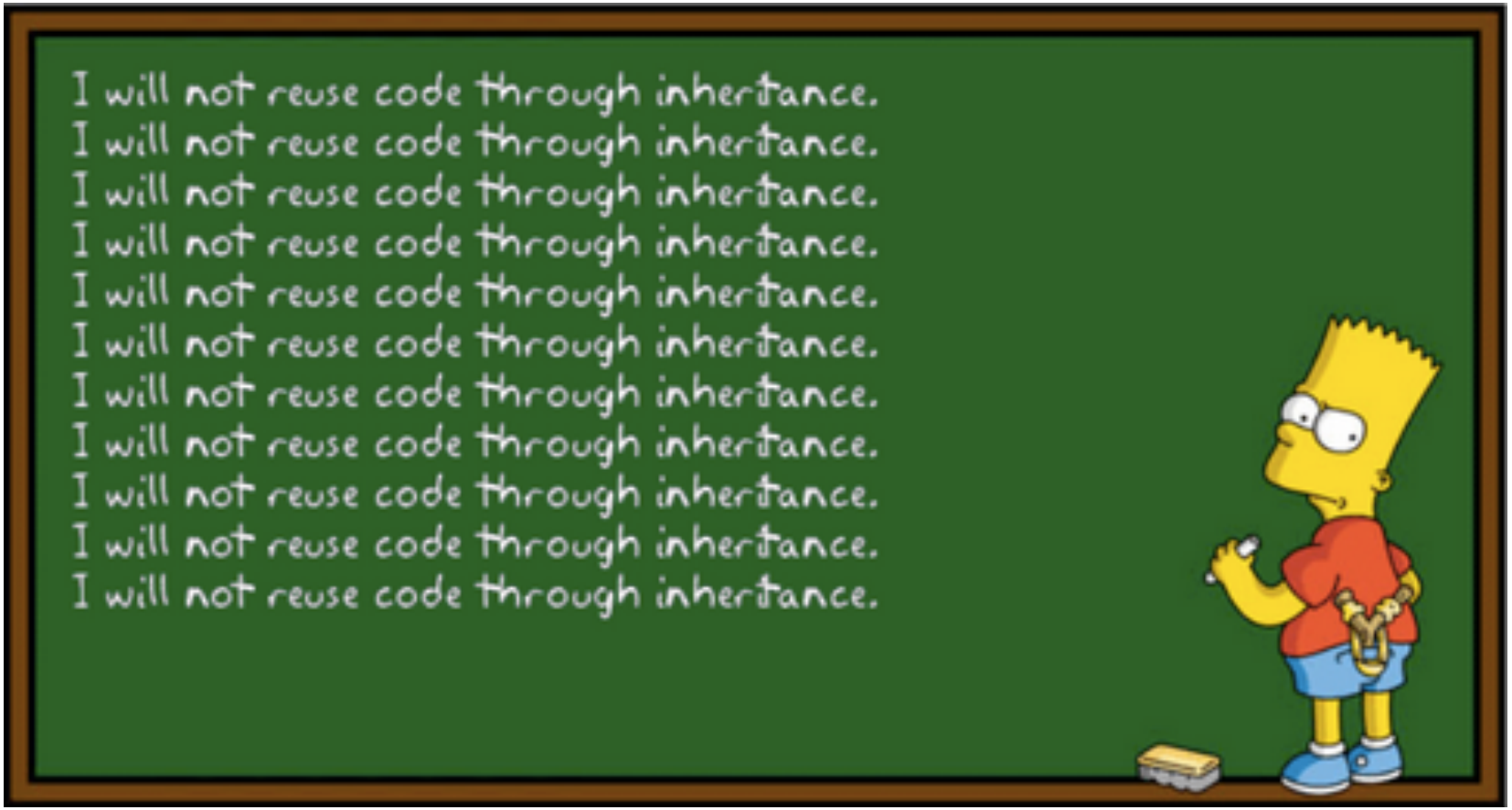
Problem without it:

- Damaging the higher level code that uses the lower level classes.
- Requiring too much time and effort to change the higher level code when a change occurs in the lower level classes.
- Producing less-reusable code.

Composition over Inheritance Principle

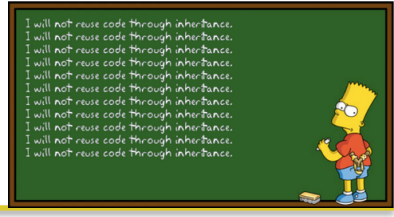
Yet another principle

Composition over Inheritance



This section info from: http://javarevisited.blogspot.com/2013/06/why-favor-composition-over-inheritance-java-oops-design.html

Composition over Inheritance

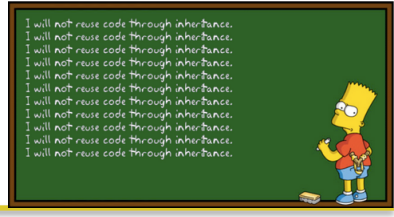


Inheritance is *way* over-used!

There are other better solutions for code re-use.

There are still reasons to use inheritance, but use it *correctly!*

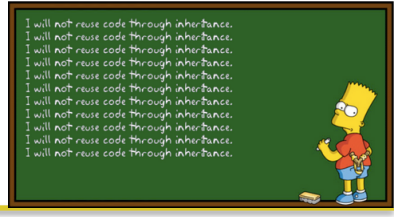
Composition over Inheritance



- Composition creates
  - Flexible code
  - Maintainable code
  - Re-usable code

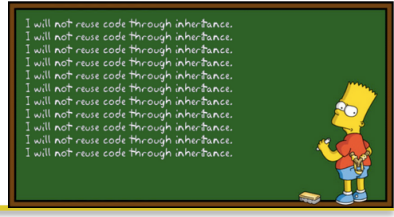
- Composition
  - Holds a reference to class object as an instance variable
  - “has-a” relationship
- Inheritance
  - Inherits from an existing class
  - “is-a” relationship

Composition over Inheritance



- Why choose composition over inheritance?
  - Need multiple functionality but language doesn't support multiple inheritance
    - *Multiple inheritance is considered dangerous and recommended to be avoided unless you really know what you're doing!*
  - Testing easier: TDD, Unit testing
  - Inheritance breaks encapsulation
    - *Subclass depends on super class for functionality…*
  - Each class focused on a single task
  - The composition can be defined dynamically at run-time through objects acquiring references to other objects of the same type
  - "Black-box" reuse, since internal details of contained objects are not visible

Composition over Inheritance



- Disadvantages
  - Resulting systems tend to have more objects

When to use Inheritance?

Use inheritance only when all of the following criteria are satisfied:

- A subclass expresses *"is a special kind of"*
  and **not** *"is a role played by a"*
- An instance of a subclass never needs to become an object of (instance of) another class
- A subclass extends, rather than overrides or nullifies, the responsibilities of its superclass
- A subclass does not extend the capabilities of what is merely a utility class
- For a class in the actual Problem Domain, the subclass specializes a role, transaction or device

- Inheritance is good for
  - Extensibility
  - Reusability
  - Abstraction
  - Eliminate redundant code

Composition

- Method of reuse in which new functionality is obtained by creating an object composed of other objects
- The new functionality is obtained by delegating functionality to one of the objects being composed
- For example:
  - **Aggregation** - when one object owns or is responsible for another object and both objects have identical lifetimes (GoF)
  - **Aggregation** - when one object has a collection of objects that can exist on their own (UML)
  - **Containment** - a special kind of composition in which the contained object is hidden from other objects and access to the contained object is only via the container object (Coad)

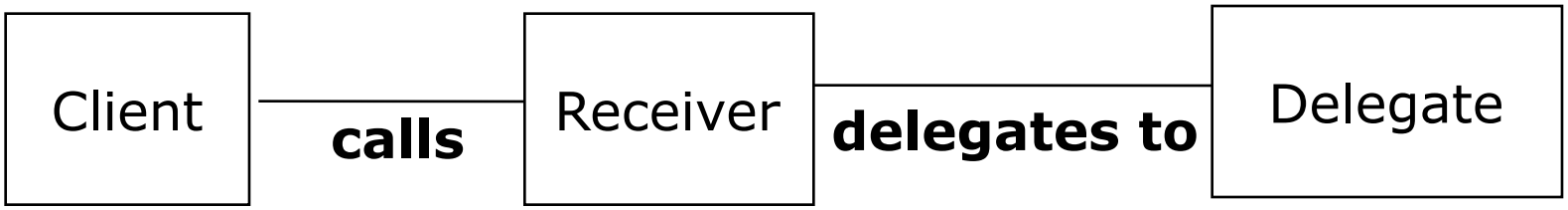
Composition vs Aggregation

- A "owns" B = **Composition** : B has no meaning or purpose in the system without A
- A "uses" B = **Aggregation** : B exists independently (conceptually) from A
- Example 1:
  - A Company is an aggregation of People. A Company is a composition of Accounts. When a Company ceases to do business its Accounts cease to exist but its People continue to exist.

- Example 2: (very simplified)
  - A Text Editor owns a Buffer (composition). A Text Editor uses a File (aggregation). When the Text Editor is closed, the Buffer is destroyed but the File itself is not destroyed.

Delegation

- Delegation is a way of making composition as powerful for reuse as inheritance
- In delegation two objects are involved in handling a request from a Client
  - The Receiver object delegates operations to the Delegate object
  - The Receiver object makes sure, that the Client does not misuse the Delegate object.



Comparison: Delegation vs Implementation Inheritance

- Delegation
  - ☺ Flexibility: Any object can be replaced at run time by another one (as long as it has the same type
  - ☹ Inefficiency: Objects are encapsulated.
- Inheritance
  - ☺ Straightforward to use
  - ☺ Supported by many programming languages
  - ☺ Easy to implement new functionality
  - ☹ Inheritance exposes a subclass to the details of its parent class
  - ☹ Any change in the parent class implementation forces the subclass to change (which requires recompilation of both)

Comparison: Delegation v. Inheritance

- Code-Reuse can be done by delegation as well as inheritance
- Delegation
  - Flexibility: Any object can be replaced at run time by another one
  - Inefficiency: Objects are encapsulated
- Inheritance
  - Straightforward to use
  - Supported by many programming languages
  - Easy to implement new functionality
  - Exposes a subclass to details of its super class
  - Change in the parent class requires recompilation of the subclass.