SOLID principles in PHP

→ Created by Robert C. Martin in 2000s
→ Quoted in the book Agile software development, principles, patterns and practice
→ Goal: create a structure easy to maintain and extend

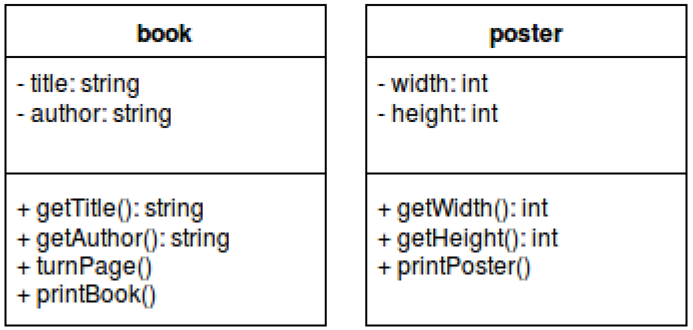
PHP OOP: reminder

- In PHP OOP we can create:
  - **Classes**
  - **Abstract classes**
  - **Interfaces**

Single responsibility principle

A class should have only a single responsibility

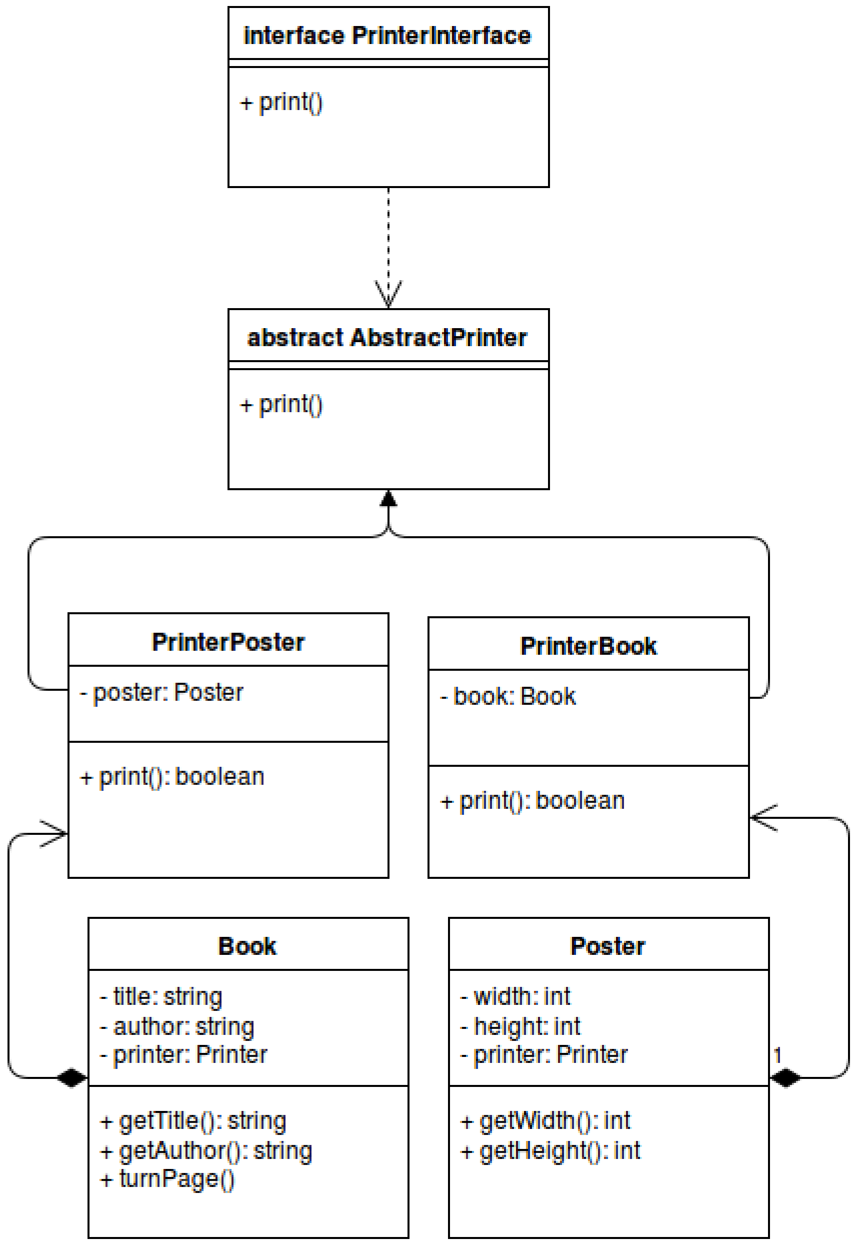
Example - printing different materials



- Multiple responsibility:
  - Display book/poster information
  - Print book/poster

  Problems:
  → code duplication between the two print method
  → not easy to extend

Possible solution



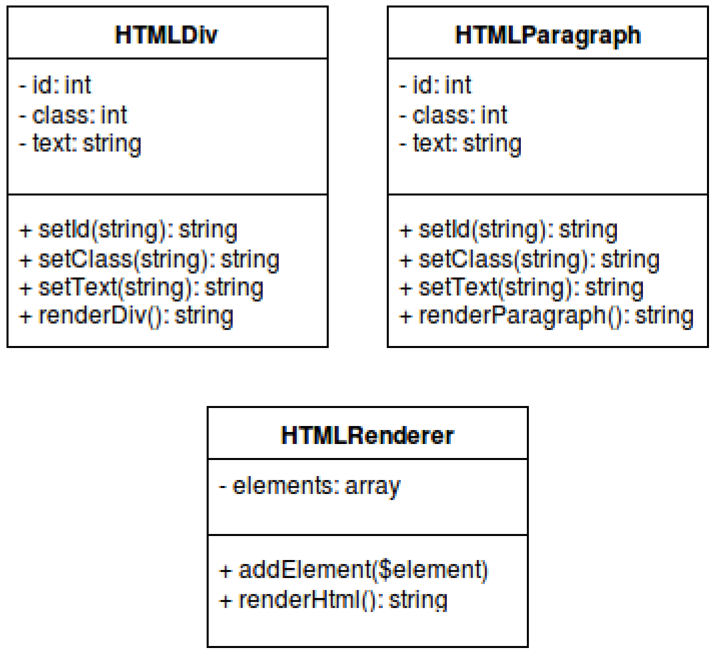
→ book and poster have only one responsibility

→ no duplicated code (common code is in the abstract class)

Open/Closed principle

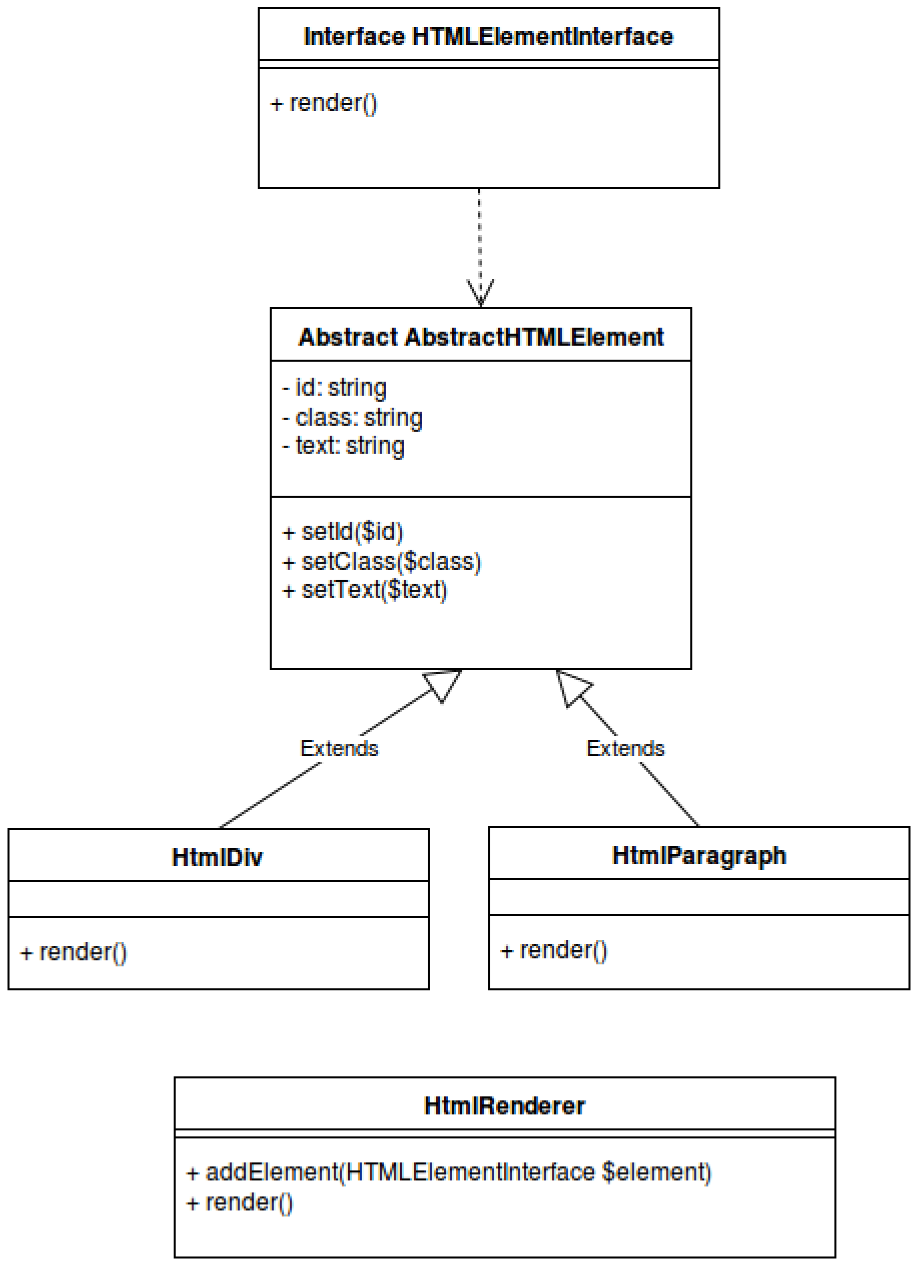
Software entities (class, modules...) should be open for extension, but closed for modification.

Example - Render HTML



Problem:
→ Need to modify HTMLRenderer each time we add a new HTML element

Possible solution

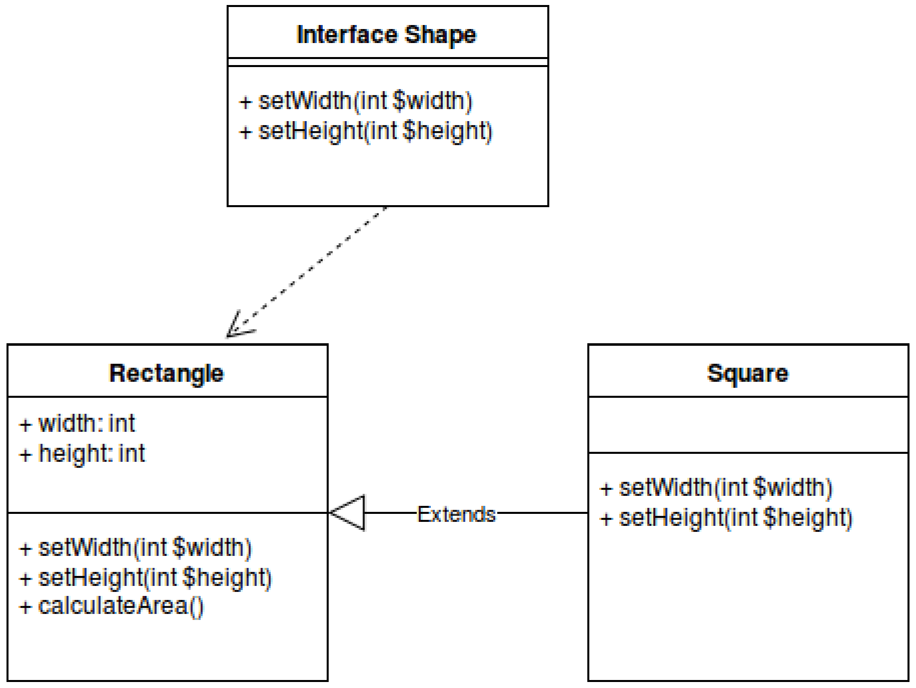


→ create a new class for a new element without modifying anything (close to modification, open to extension)

Liskov substitution principle

Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.

Example – The famous square/rectangle problem



→ In an application, if we replace the object Rectangle by the object Square, the behavior and result won't change

Implementation

```
class Rectangle {
    public function setWidth($width){
        $this->width = $width;
    }
    public function setHeight($height){
        $this->height = $height;
    }
    public function calculateArea(){
        return $this->width * $this->height;
    }
}

class Square extends Rectangle {
    public function setWidth($width){
        $this->width = $width;
        $this->height = $width;
    }
    public function setHeight($height){
        $this->height = $height;
        $this->width = $height;
    }
}
```

```
$rectangle = new Rectangle();
$rectangle->setWidth(2);
$rectangle->setHeight(3);

var_dump($rectangle->area());
// Good Result: int(6)
```

Following the principle, replacing Rectangle by Square should provide the same output

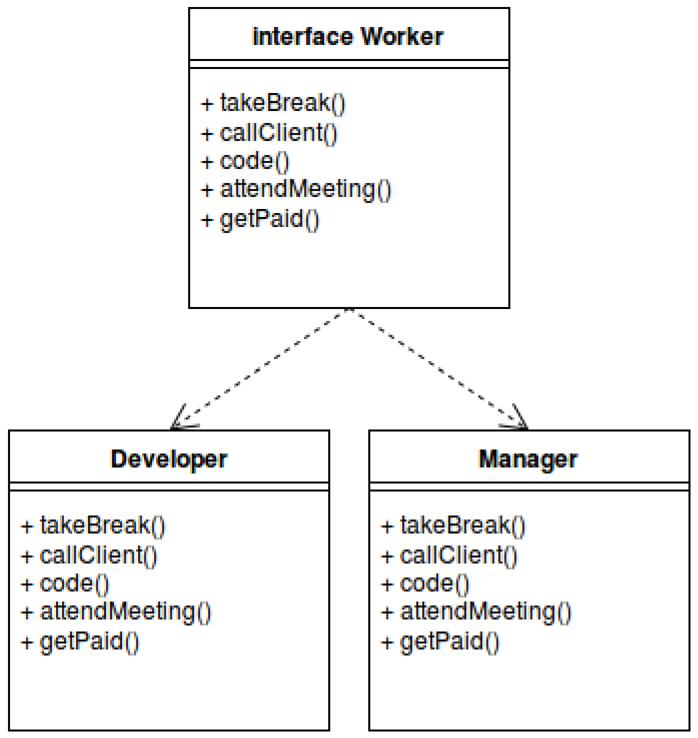
```
$rectangle = new Square();
$rectangle->setWidth(2);
$rectangle->setHeight(3);

var_dump($rectangle->area());
// Bad Result: int(9) instead of int(6) !
```

Interface segregation principle

Many client-specific interfaces are better than one general-purpose interface.

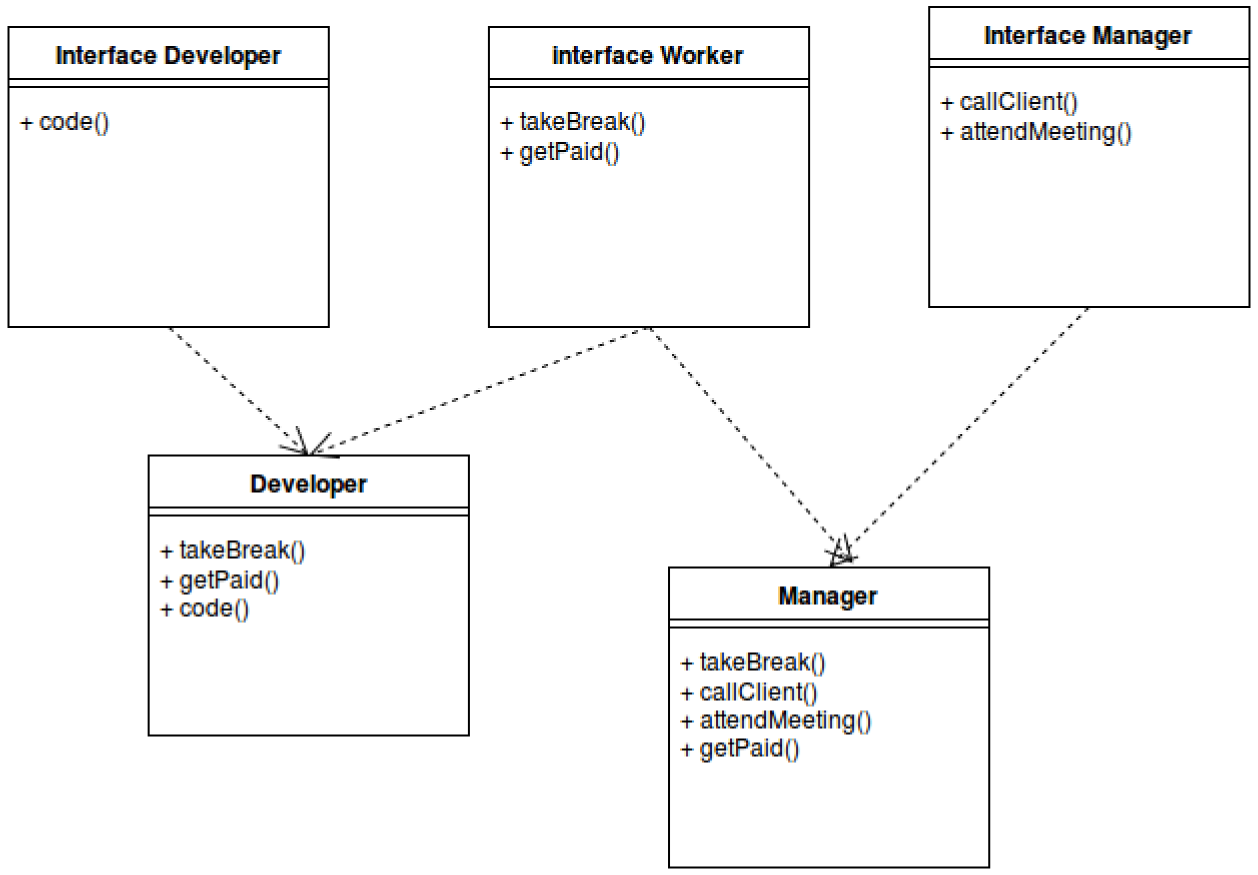
Example – The worker



Problem:
→ Is the code() method necessary for an object manager?

What about the callClient() method for a developer?

Possible solution

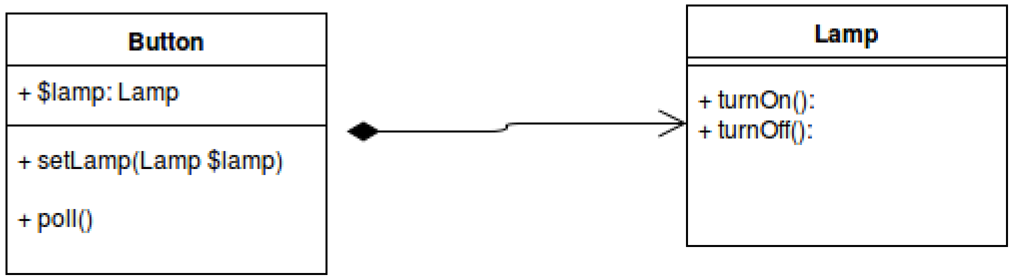


→ Multiple interfaces for multiple behaviours

Dependency inversion principle

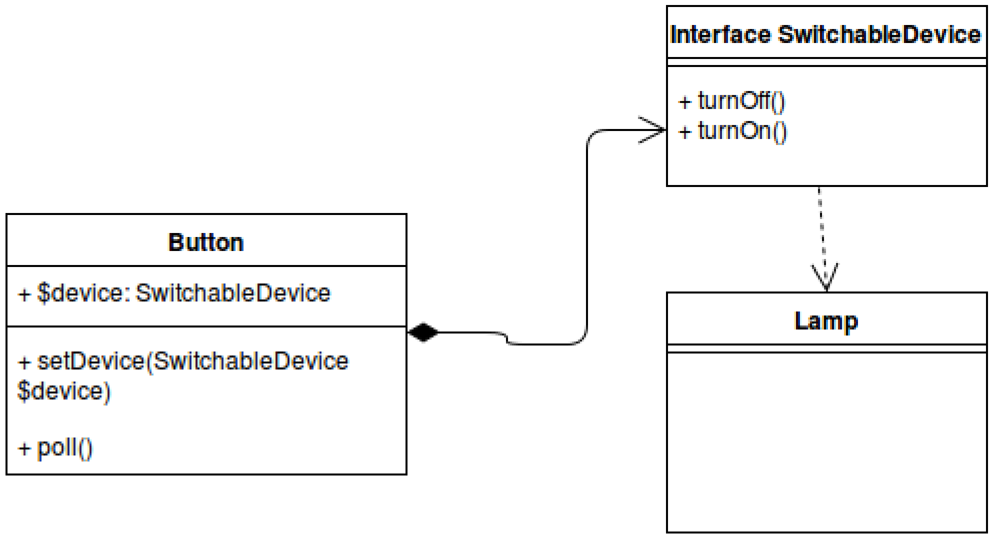
Classes depend upon Abstractions. Do not depend upon concretions.

<u>Example – The lamp</u>



→ Button depends directly on Lamp – changes to Lamp may require change to button
→ Button is not reusable (I can't control a new object Motor with it)

Possible solution


Interface SwitchableDevice
+ turnOff()
+ turnOn()
Button
+ $device: SwitchableDevice
+ setDevice(SwitchableDevice $device)
+ poll()
Lamp

→ Buttons can control any device that implements SwitchableDevice interface (like a new Motor object)
→ Any object can implement SwitchableDevice and control the lamp and other SwitchableDevice