Capstone Projects
SE 490

SOLID revisited

Dr. Simon Fan
Professor of Computer Science and Software Engineering
California State University ● San Marcos
sfan@csusm.edu

SOLID Software Design Principles

- **SOLID** principles *are guidelines* that can be applied to remove code smells by refactoring the software's source code until it is both legible and extensible.

S	Single-responsibility principle
O	Open-close principle
L	Liskov substitution principle
I	Interface segregation principle
D	Dependency Inversion Principle

An example (php)

```php
class Circle {
    public $radius;

    public function __construct($radius) {
        $this->radius = $radius;
    }
}

class Square {
    public $length;

    public function __construct($length) {
        $this->length = $length;
    }
}
```

```php
class AreaCalculator {

    protected $shapes;

    public function __construct($shapes = array()) {
        $this->shapes = $shapes;
    }

    public function sum() {
        // logic to sum the areas
    }

    public function output() {
        return implode('', array(
            "<h1>",
                "Sum of the areas of provided shapes: ",
                $this->sum(),
            "</h1>"
        ));
    }
}
```

```php
$shapes = array(
    new Circle(2),
    new Square(5),
    new Square(6)
);

$areas = new AreaCalculator($shapes);

echo $areas->output();
```

Can you identify two reasons to change the AreaCalculator class?

SOLID Software Design Principles

Single Responsibility Principle	A class should have only one responsibility

A responsibility, or functionality, is a family of functions that serves **one particular actor/client**

A tool with a knife and a fork offers two functionalities

- For a class with more responsibilities, when we need to make a change to one, the change might affect the other functionality of the class.
- If we have two reasons to change a class, we should split its responsibility in two classes. Each class will handle only one responsibility

SOLID Software Design Principles

Open Close Principle	Software entities (like classes, modules, packages, libraries) should be open for extension but closed for modifications

Assume you have a library containing a set of classes, there is a need to replace the existing implementation of a class (say, for efficiency reason).

It is better to extend it rather than change the code that was already written.

- Some particular uses of this principle
  - Template Pattern
  - Strategy Pattern

Open Close Principle example

It is **NOT** closed for modifications

```php
class AreaCalculator {
  public function sum() {
      foreach($this->shapes as $shape) {
          if(is_a($shape, 'Square')) {
              $area[] = pow($shape->length, 2);
          } else if(is_a($shape, 'Circle')) {
              $area[] = pi() * pow($shape->radius, 2);
          }
      }
}
```

```php
class Square {
    public $length;

    public function __construct($length) {
        $this->length = $length;
    }
}
```

If we wanted the **sum** method to be able to sum the areas of more shapes, we would have to add more **if/else blocks** and that goes against the Open-closed principle.

It is **now** closed for modifications

```php
class AreaCalculator {
  public function sum() {
      foreach($this->shapes as $shape) {
          $area[] = $shape->area;
      }

      return array_sum($area);
  }
}
```

```php
class Square {
    public $length;

    public function __construct($length) {
        $this->length = $length;
    }

    public function area() {
        return pow($this->length, 2);
    }
}
```

Assuming the same is done for the **Circle** class: an **area** method should be added

A Step Further: Coding to an interface

- Programming in general: using abstract classes (or interfaces) as contracts to client code.
  - The contract enforces a concrete class to implement the behavior defined in the abstract superclass or interface.

closed for modifications

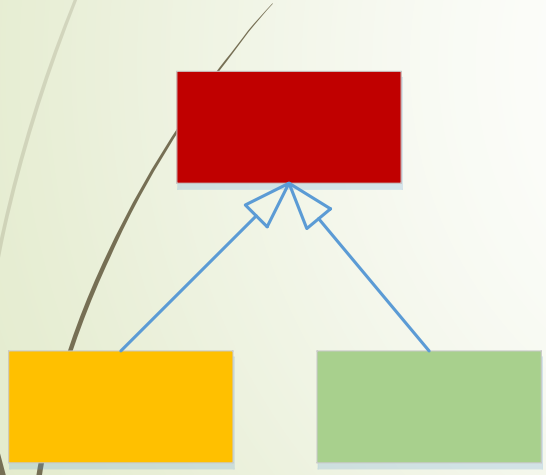
```
public function sum() {
    foreach($this->shapes as $shape) {
        if(is_a($shape, 'ShapeInterface')) {
            $area[] = $shape->area();
            continue;
        }

        throw new AreaCalculatorInvalidShapeException;
    }

    return array_sum($area);
}
```

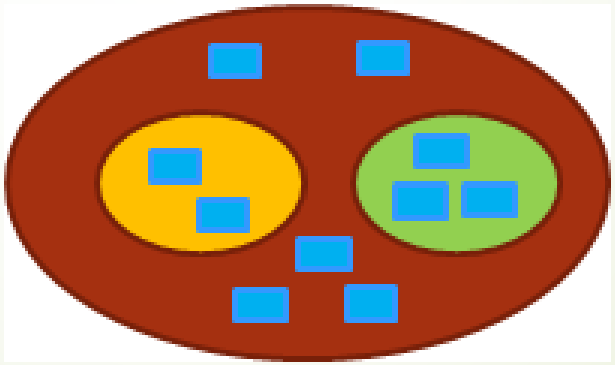
open for extension

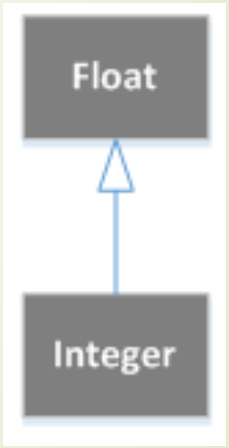
```
interface ShapeInterface {
    public function area();
}
class Circle implements ShapeInterface {
    public $radius;
    public function __construct($radius) {
        $this->radius = $radius;
    }
    public function area() {
        return pi() * pow($this->radius, 2);
    }
}
```

Concept: Data Types

- In OOP, each class also represents a data type.
- The type represented by a subclass is subsumed by the type represented by its superclass.
- In other words, a superclass represents a relatively wider type (**base type, supertype**) and its subclass is a narrower type (**derived type, subtype**).

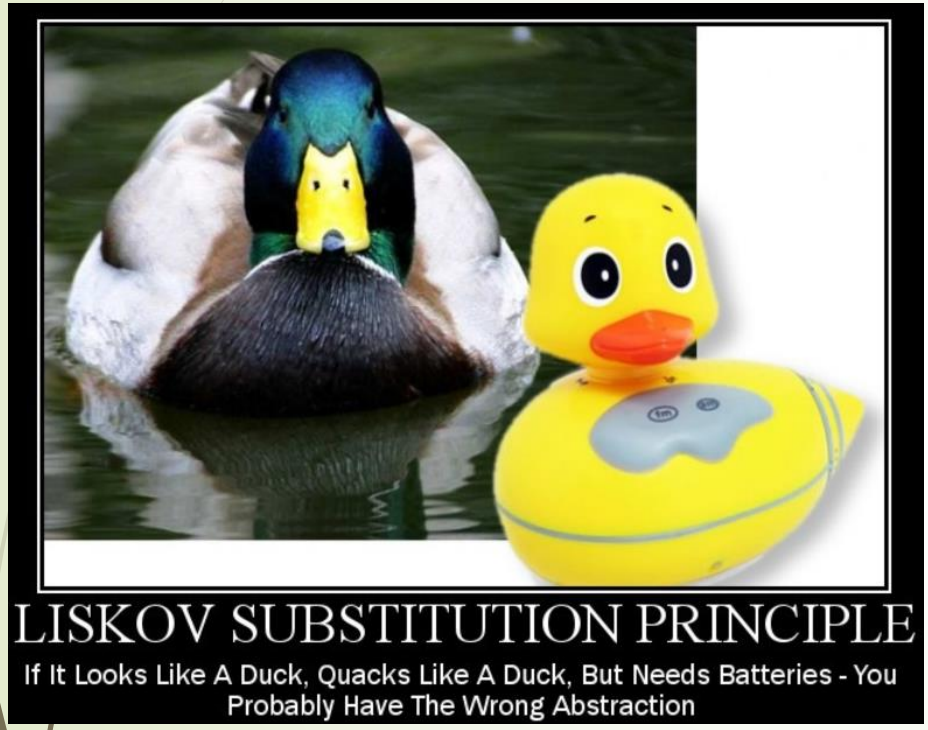


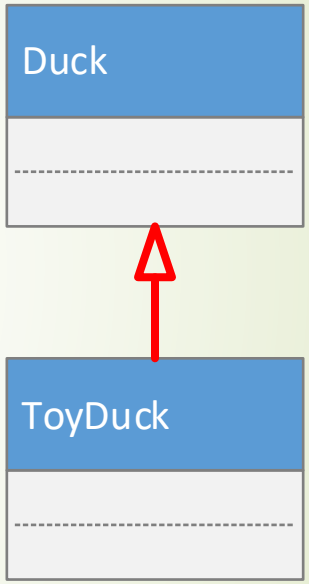




SOLID Software Design Principles

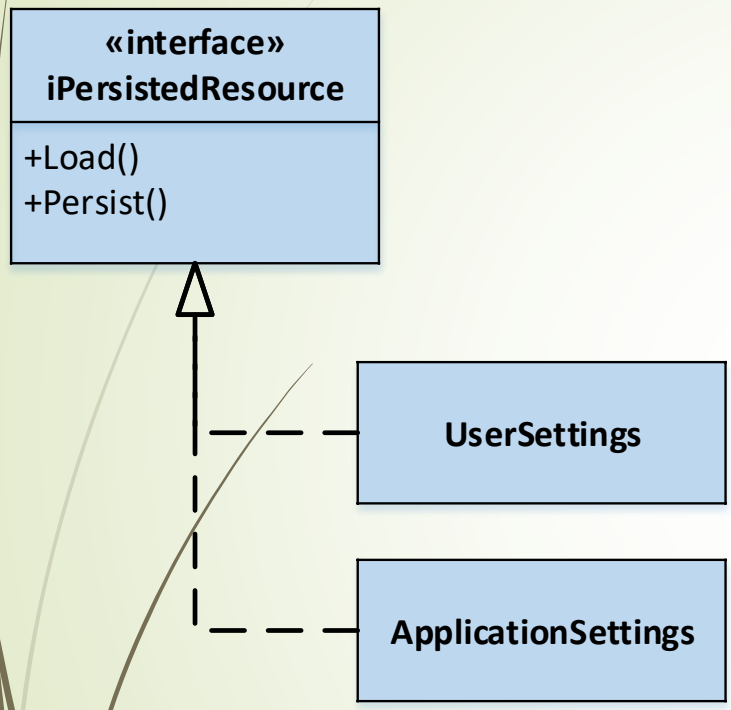
Liskov's Substitution Principle	Derived types must be completely substitutable for their base types. *A derived class should simply extend the base class without changing the contract as declared in the base class*





- A derived class should be able to replace the base class without causing code changes in the client code (or surprise to the client).
- Client code that uses the base class objects must be able to use the objects of the derived class without knowing it

LSP Example



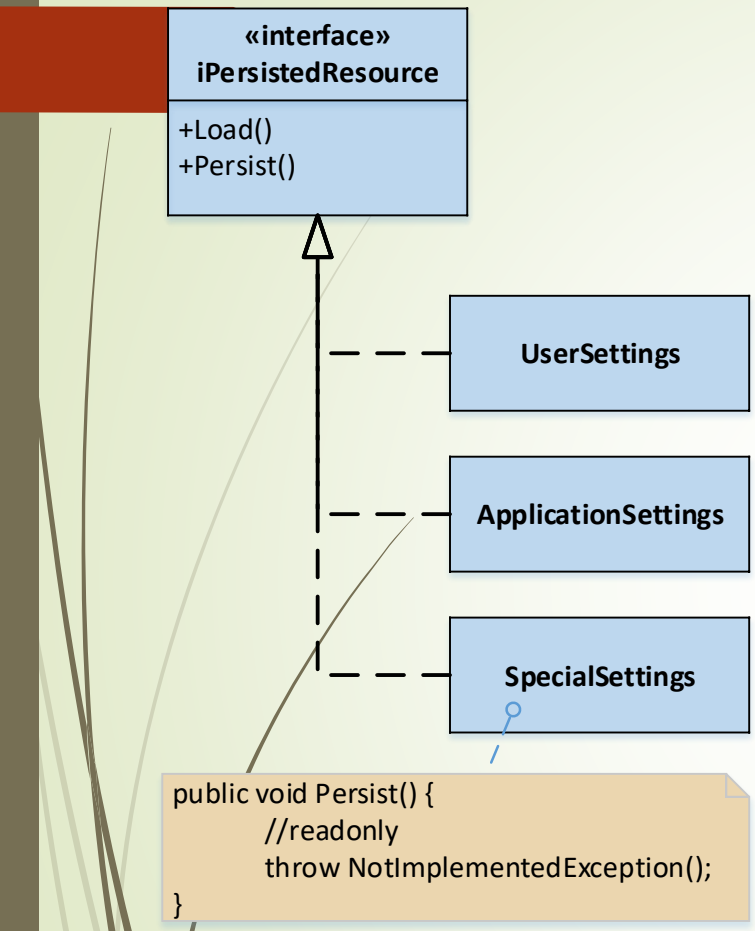

```
static IEnumerable<IPersistedResource> LoadAll()
{
    var allResources = new List<IPersistedResource>
                       {
                           new UserSettings(),
                           new ApplicationSettings()
                       };
    allResources.ForEach(r => r.Load());
    return allResources;
}
```

Client1

```
static void SaveAll(IEnumerable<IPersistedResource> resources)
{
    resources
        .ForEach(r => r.Persist());
}
```

Client2

LSP Example




```
static IEnumerable<IPersistedResource> LoadAll()
{
    var allResources = new List<IPersistedResource>
                       {
                           new UserSettings(),
                           new ApplicationSettings(),
                           new SpecialSettings()
                       };
    allResources.ForEach(r => r.Load());
    return allResources;
}
```

Client1

```
static void SaveAll(IEnumerable<IPersistedResource> resources)
{
    resources
        .ForEach(r => r.Persist());
}
```

Client2

SpecialSettings is not substitutable for IPersistResource: **it behaves differently**. In saveall(), if we call Persist on SpecialSettings object, the app blows up.

One (horrible) way to avoid exceptions

```
static void SaveAll(IEnumerable<IPersistedResource> resources)
{
    resources
        .ForEach(r =>
                 {
                     if (r is SpecialSettings)
                         return;

                     r.Persist();
                 });
}
```

Client

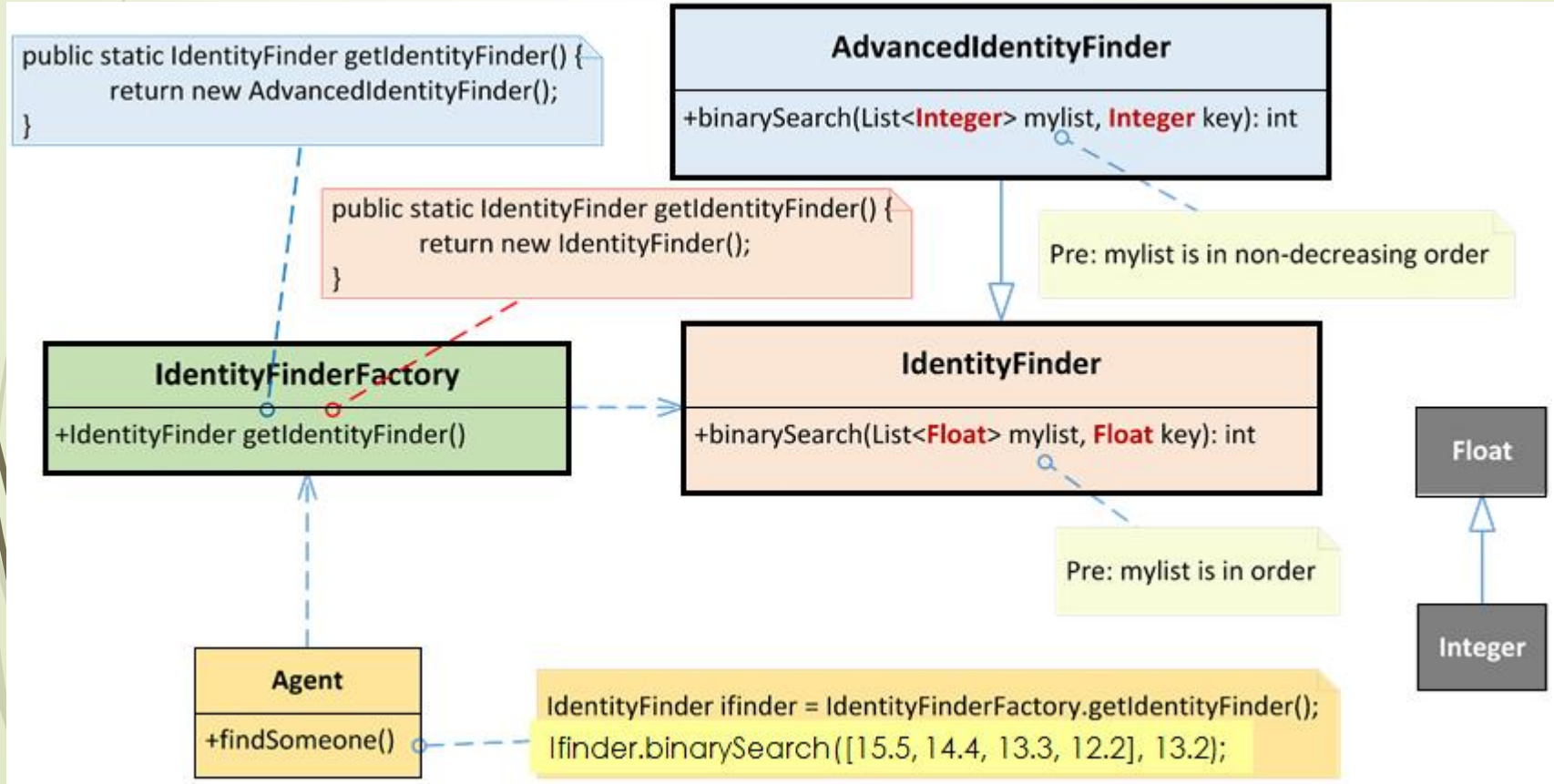
Anytime you see code that takes in some sort of baseclass or interface and then performs a check such as “if (someObject is SomeType)”, there’s a very good chance that that’s an LSP violation

Liskov: Breakout Discussion

Liskov's Substitution	A subclass object can always be used to substitute for an object of its superclass *without causing any surprises to its client*.

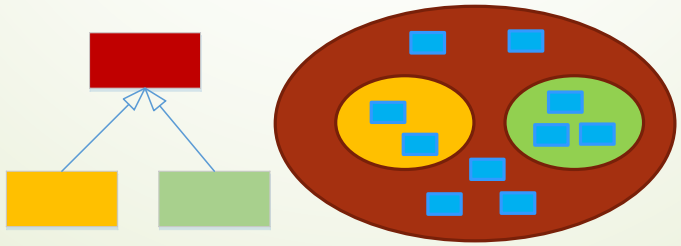
- Client code that uses the base class objects must be able to use the objects of the derived class without knowing it

Would the following design violates the Liskov principle?



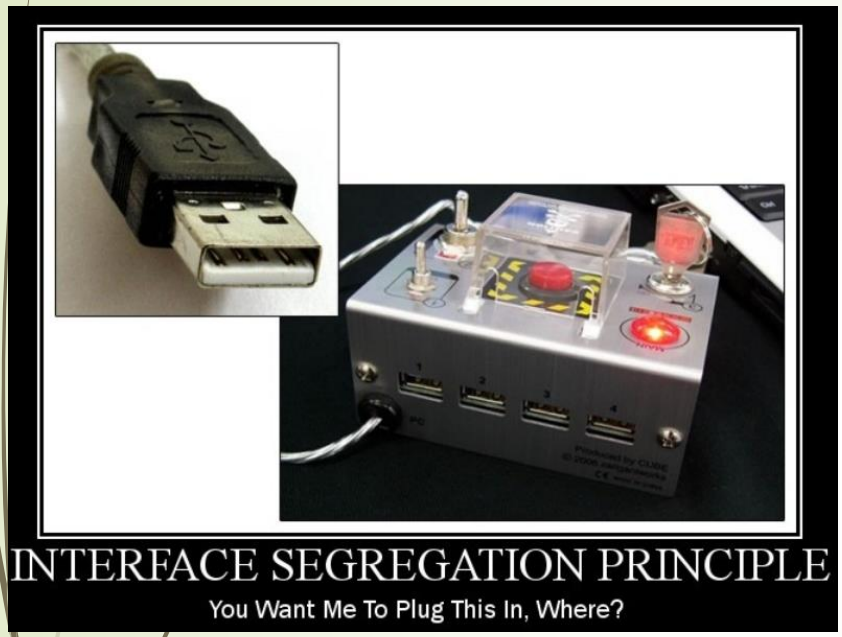
Liskov substitution principle: summary

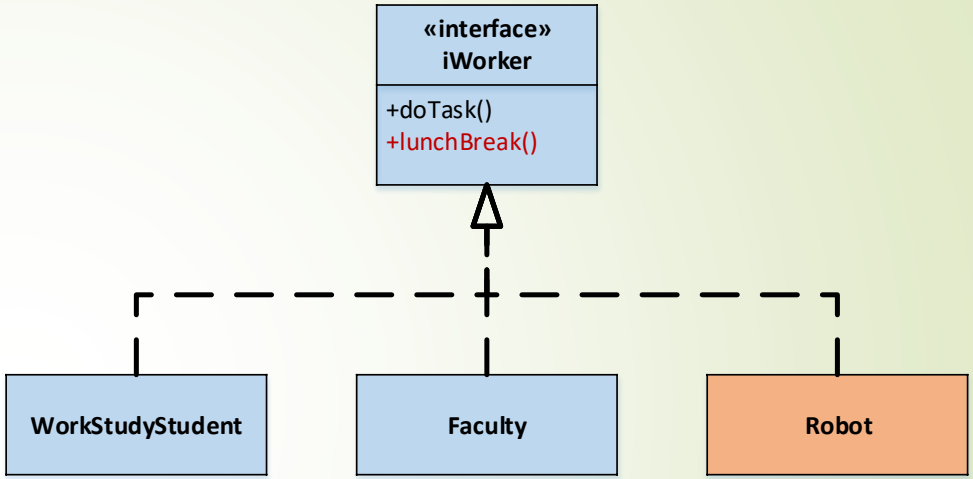
- ***Liskov substitution principle*** implies that any subclass object must be at least able to do what the superclass objects can. This imposes some **standard requirements on subclassing**:
  1) Sustain public operations
  2) Method parameters in the subclass can have wider types
  3) Return values in the subclass can have narrower types (stricter).
  4) Preconditions cannot be strengthened in a subclass
  5) Postconditions cannot be weakened in a subclass
  6) Invariants of the superclass must be preserved in a subclass



SOLID Software Design Principles

Interface Segregation Principle	Clients should not depend upon interfaces that they don't use





- An interface contains only methods that should be there. If we add methods that should not be in an interface, the classes implementing it will have to implement those methods as well.
- Interfaces containing methods that are not specific to it are called **polluted** or fat interfaces. We should avoid them.

Interface Segregation Principle example

```
public interface IPersistedResource
{
    void Load();
    void Persist();
}
```

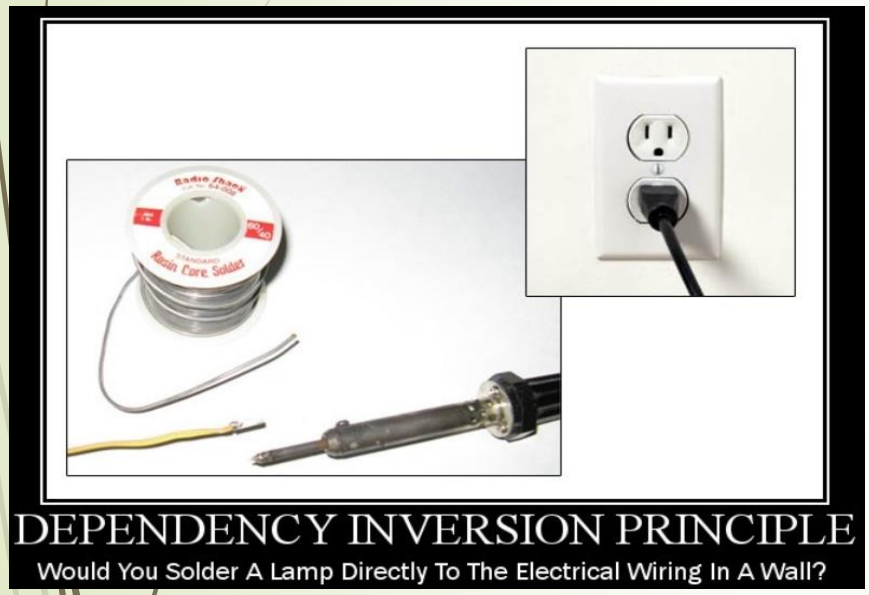
```
public interface ILoadResource
{
    void Load();
}

public interface IPersistResource
{
    void Persist();
}
```

```
static void SaveAll(IEnumerable<IPersistResource> resources)
{
    resources
        .ForEach(r => r.Persist());
}

static IEnumerable<ILoadResource> LoadAll()
{
    var allResources = new List<ILoadResource>
                       {
                           new UserSettings(),
                           new ApplicationSettings(),
                           new SpecialSettings()
                       };
    allResources.ForEach(r => r.Load());
    return allResources;
}
```

SOLID Software Design Principles

Dependency Inversion Principle	High-level modules should not directly depend on low-level modules

In the classical way when a "requester" module(class, framework) need some "supplier" module, it initializes and **holds a direct reference** to it. This will make the 2 modules tightly coupled.

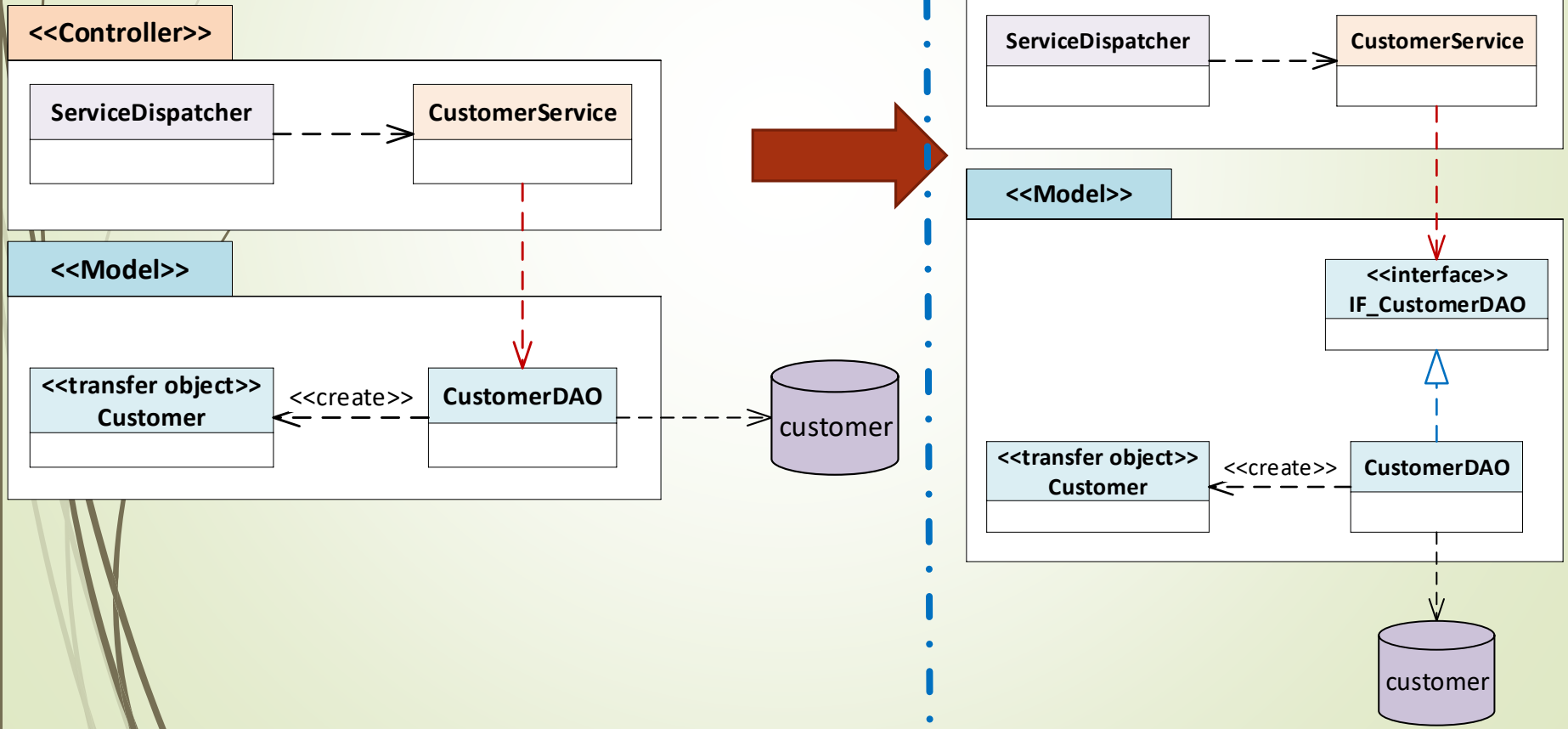


Dependency Inversion Principle states that we should decouple high level classes (service requester classes) from low level classes (service supplier classes), introducing an abstraction layer between them.

In other words, high level classes are **not working directly** with low level classes, they are using interfaces as an abstract layer

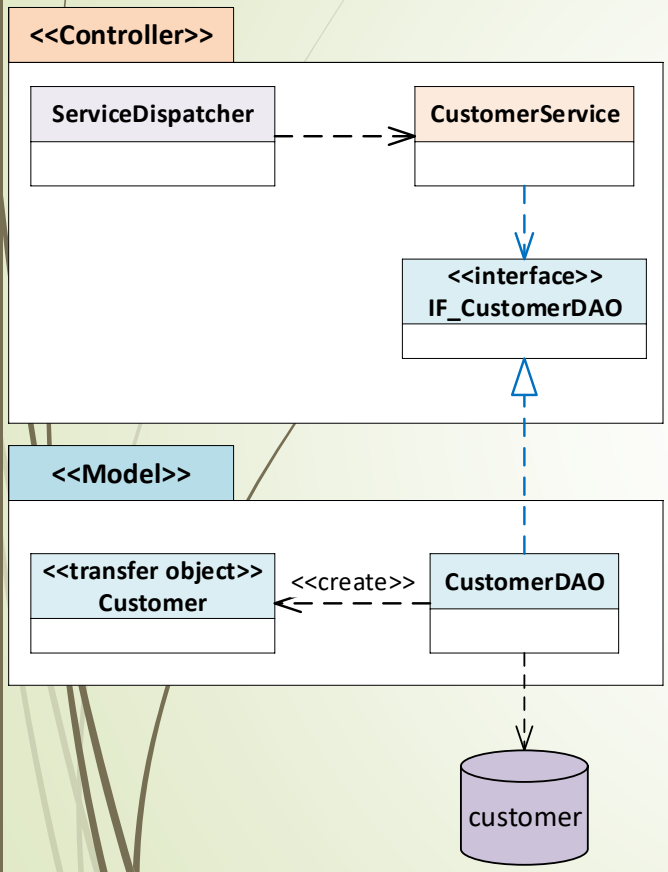
Dependency Inversion principle: architecture design

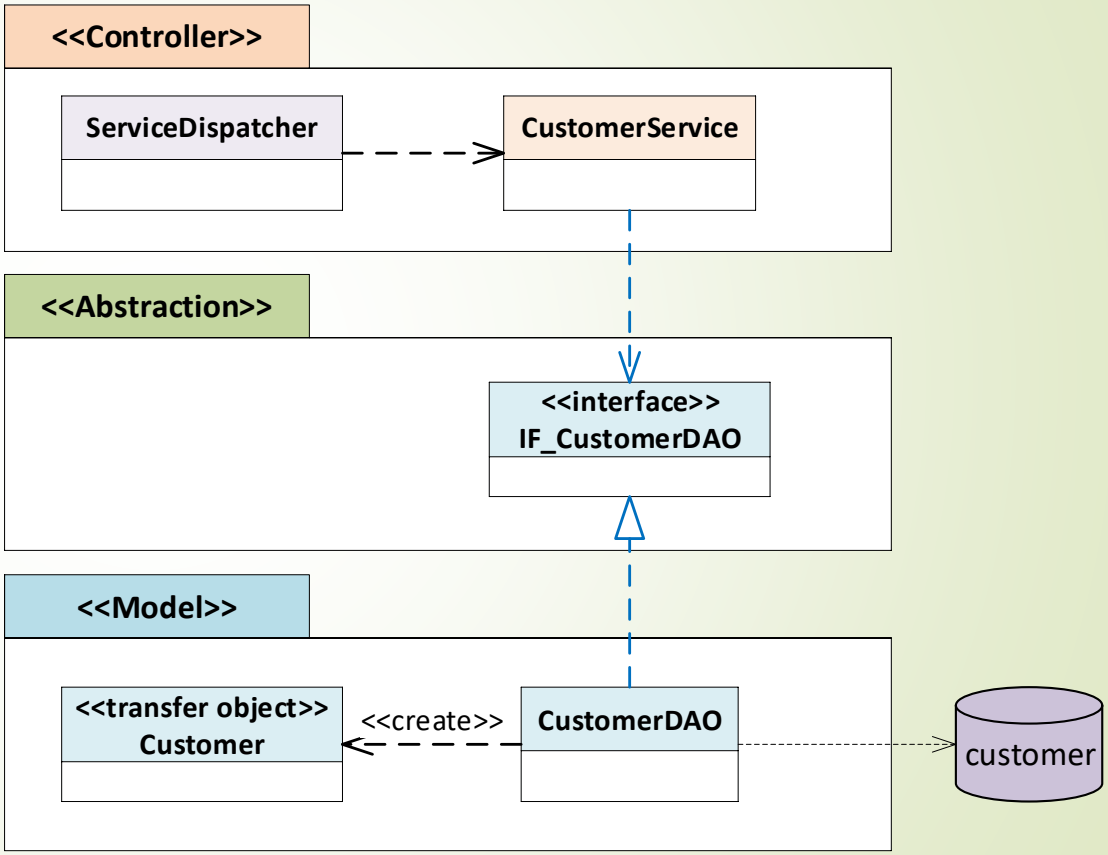
High-level modules should not directly depend on low-level modules.
【 Inversion】 Both should depend on abstractions (an interface or an abstract class**).**



Dependency Inversion principle: architecture design

High-level modules should not directly depend on low-level modules.
【 Inversion】Both should depend on abstractions.



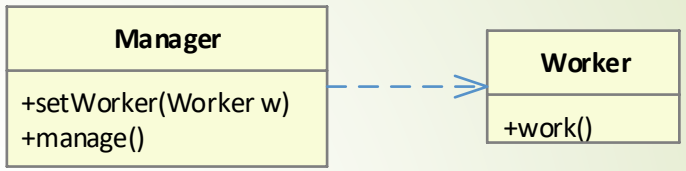


Dependency Inversion: lower-level design Example

High-level classes should not directly depend on low-level classes.
【 Inversion】 Both should depend on abstractions.

```
class Worker {
    public void work() {

        // ....working
    }
}

class Manager {
    Worker worker;

    public void setWorker(Worker w) {
        worker = w;
    }

    public void manage() {
        worker.work();
    }
}
```

```
class SuperWorker {
    public void work() {
        //.... working much more
    }
}
```



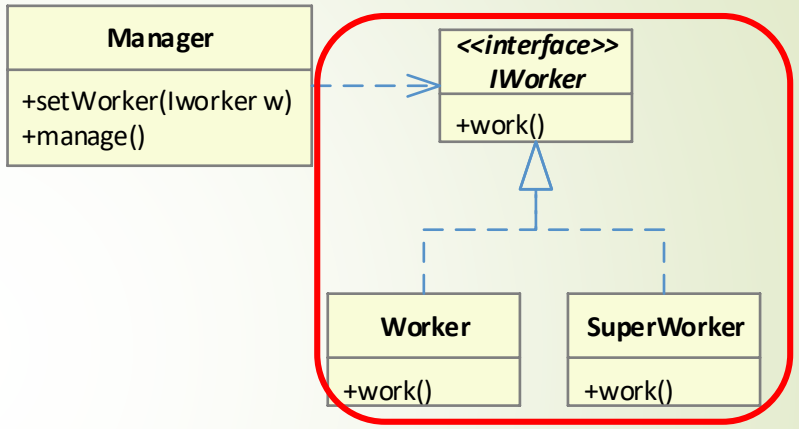
High-level class directly depend on low-level class.

1. What if Manager also needs to manage super workers?
2. Can we break the dependency?

Dependency Inversion: lower-level design Example

High-level classes should not directly depend on low-level classes.
【 Inversion】 Both should depend on abstractions.

```
// Dependency Inversion Principle
interface IWorker {
    public void work();
}

class Worker implements IWorker{
    public void work() {
        // ....working
    }
}
class SuperWorker  implements IWorker{
    public void work() {
        //.... working much more
    }
}
class Manager {
    IWorker worker;

    public void setWorker(IWorker w) {
        worker = w;
    }

    public void manage() {
        worker.work();
    }
}
```



- The supplier provides an **abstract interface**
- The requester provides a **hook** (a property, parameter) that recognizes the abstract interface only

By applying the Dependency Inversion the requester behavior can be easily changed by just changing the dependencies.