

# **Object-Oriented Principles**

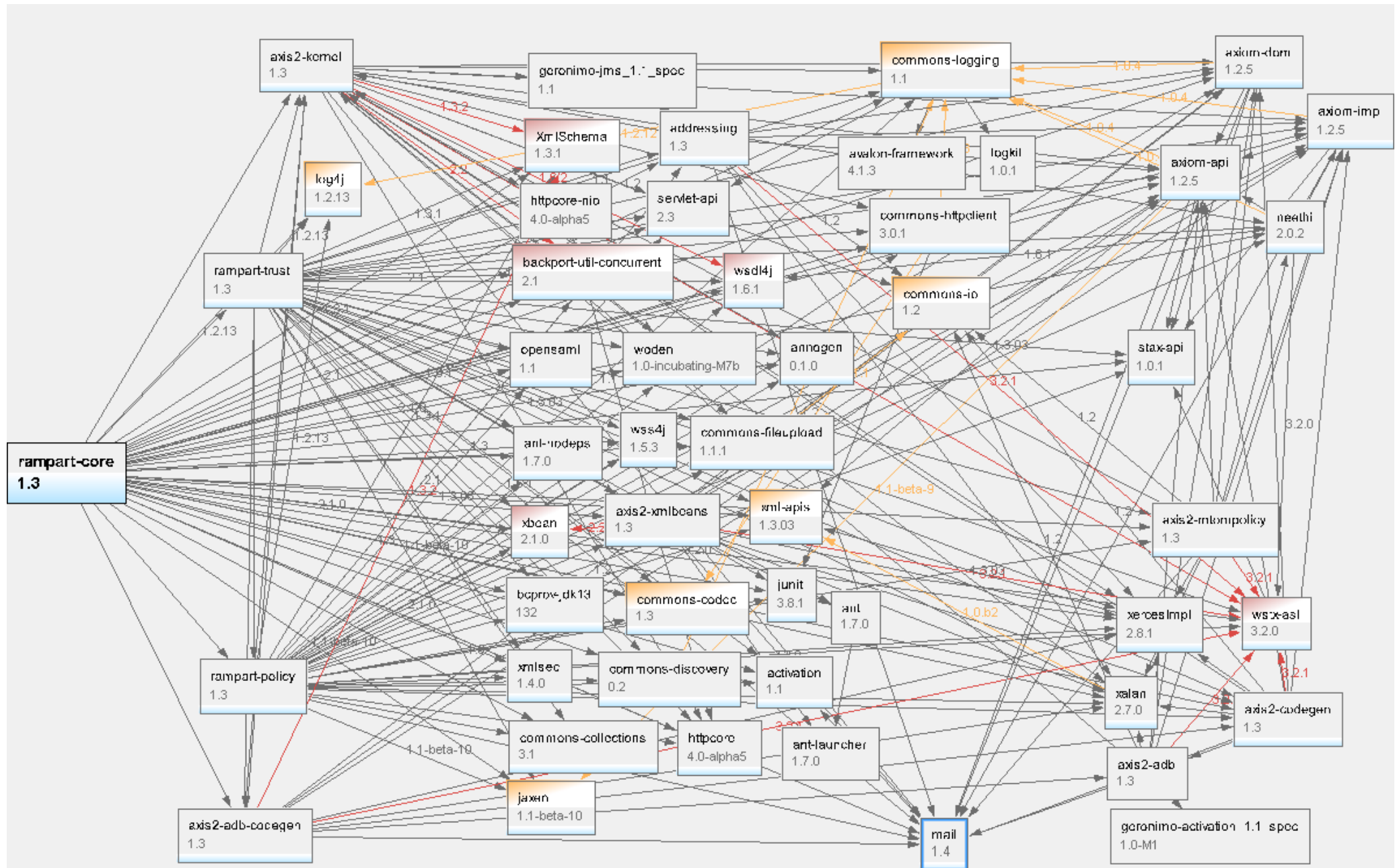
---

Comp 47480: Contemporary Software Development

# Principles and Patterns

- The activity of design can be informed by
  - **principles:** general statements of what properties a good design ought to have
  - **heuristics:** guidelines for creating a good design
  - **metrics:** how to measure the quality of a design
  - **patterns:** concrete examples of successful designs for particular purposes.
- We'll examine these topics in the coming lectures. This section relates to software **PRINCIPLES**.

It's all about **dependency**



# An über principle: Keep Dependency to a minimum

We're concerned with **static (compile-time) dependency**, not dynamic (run-time) dependency.

In a million-line code base, a new requirement means that class A has to be changed...

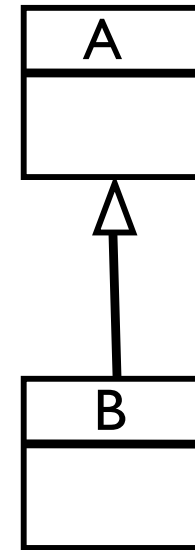
The million-dollar question is:  
what classes depend on class A? i.e.  
what else has to change because A changes?

Strongly related  
to **coupling**

# More about **Dependency**

In designing anything (software, buildings, airplanes), we have components that **depend** on each other.

For example, when class B inherits from class A, what dependency is created?



# Dependency

If a class Foo uses an instance of class FooBar, what dependency is created?

```
class Foo {  
    public Foo {  
        ... = new FooBar  
  
    }  
    ...  
}
```

# Why does dependency matter?

If A depends on B, that means that...

If a new requirement causes us to update B,  
A may well have to be updated as well.

So if B is updated a lot,  
A will have to be updated a lot as well.

Hold this  
thought!

# Stable and Unstable

A **stable** component is one that doesn't change over time.

Conversely, an **unstable** component is inclined to change over time.

In a house, the foundations are stable -- they are intended to be very hard if not impossible to move.

The paint on the walls is unstable -- it is made to be easy to replace.



# Stability and Dependency

**Edict 1:    Separate the stable from the unstable!**

**Edict 2:    The stable must not depend on the unstable.**

If you ignore the 1st edict, your whole system will be unstable.

If you ignore the 2nd edict, your whole system will be unstable.

The unstable may depend on the stable, that's ok.

# An über principle: Don't Repeat Yourself (DRY)

- Aka **Write Once and Once Only**
- Very important principle in software development and evolution
  - “[Duplication is] the bane of the programmer’s existence” — Kent Beck
- Duplicated code => **implicit dependency!**



**Implicit dependency  
is the worst type!**

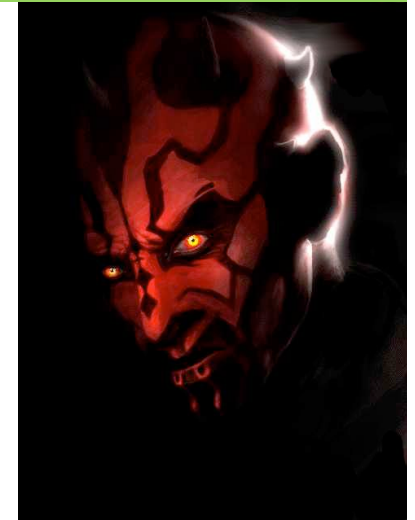
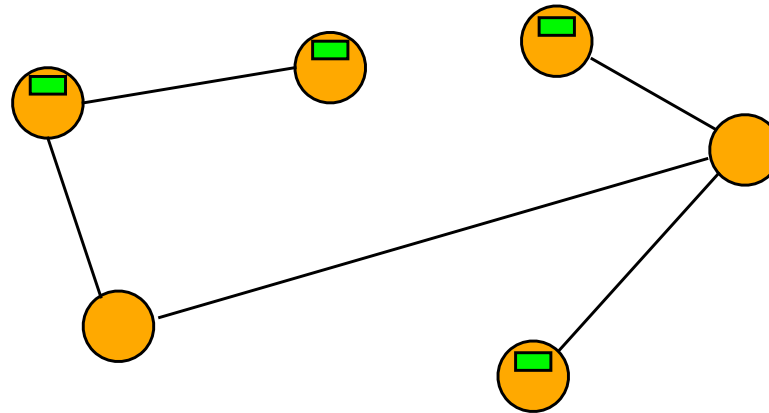
# How common is duplication?

- Lopez et al. analyzed 4.5 million non-fork GitHub projects representing over 428 million files written in Java, C++, Python, and JavaScript.
- Key finding: This corpus has a mere 85 million unique files.
- In other words, 70% of the code on GitHub consists of copies of previously created files!

Lopes, C.V. et al., DéjàVu: A Map of Code Duplicates on GitHub, Proceedings of OOPSLA, October 2017.

# Duplication: the root of all software evil

Information that appears in several places around the system is contrary to this principle.



The green boxes above represent duplicated information; if one has to change, they all have to change.

At best, updating takes longer than when there's no duplication.

At worst, error will occur if you omit one of the updates.

# SOLID Principles

- In this section we'll look at the set of object-oriented principles named by Bob Martin in the “early days” of OOP/OOD:
  - Single Responsibility Principle (SRP)
  - Open/Closed Principle (OCP)
  - Liskov Substitution Principle (LSP)
  - Interface Segregation Principle (ISP)
  - Dependency Inversion Principle (DIP)
- We'll also look at two other principles/guidelines:
  - No Concrete Superclasses
  - Law of Demeter

# Object-Oriented Principles: Roadmap

- **Single Responsibility Principle (SRP)**
- Open/Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)
- No Concrete Superclasses
- The Law of Demeter

# Single Responsibility Principle

The Single Responsibility Principle states that:

- (i) every class should have a single responsibility, and
- (ii) that responsibility should be entirely encapsulated by the class.

Strongly related  
to **cohesion**

This is a fundamental principle that expresses the core of good object-oriented design.

Following this principle has this very desirable effect:

*There is only one [type of] reason for a class to change.*

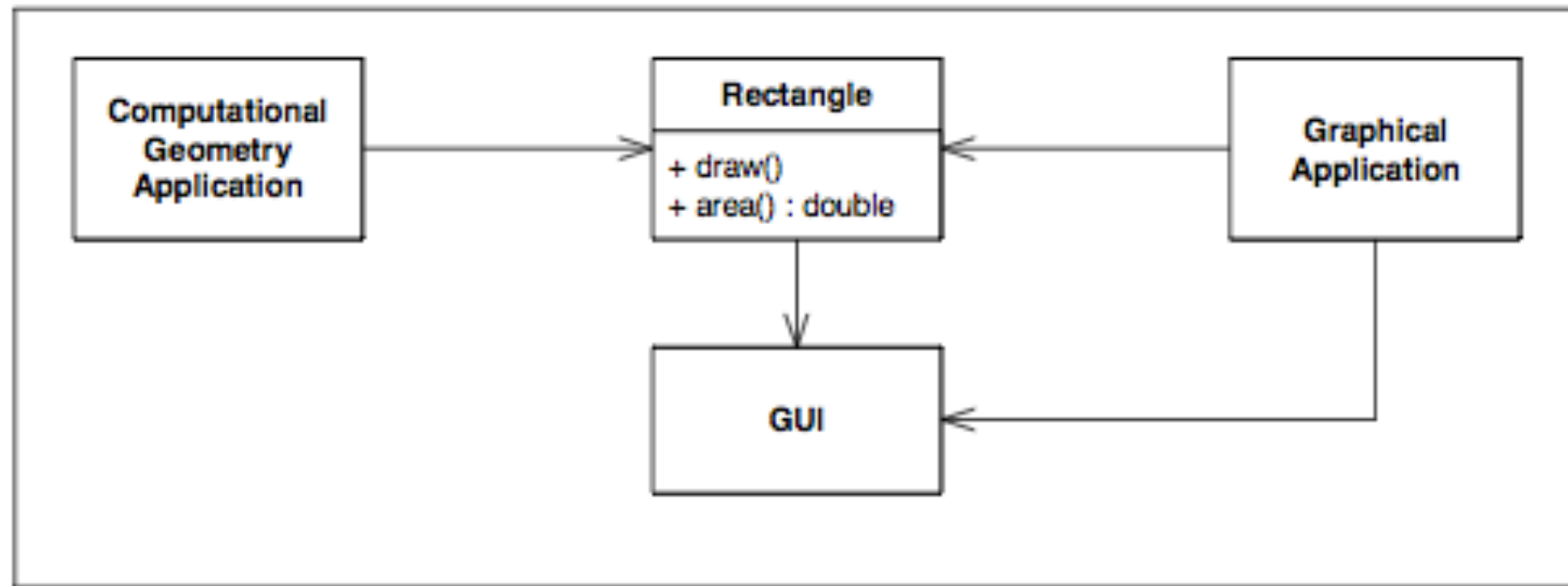


# SINGLE RESPONSIBILITY PRINCIPLE

Every object should have a single responsibility, and all its services should be narrowly aligned with that responsibility.



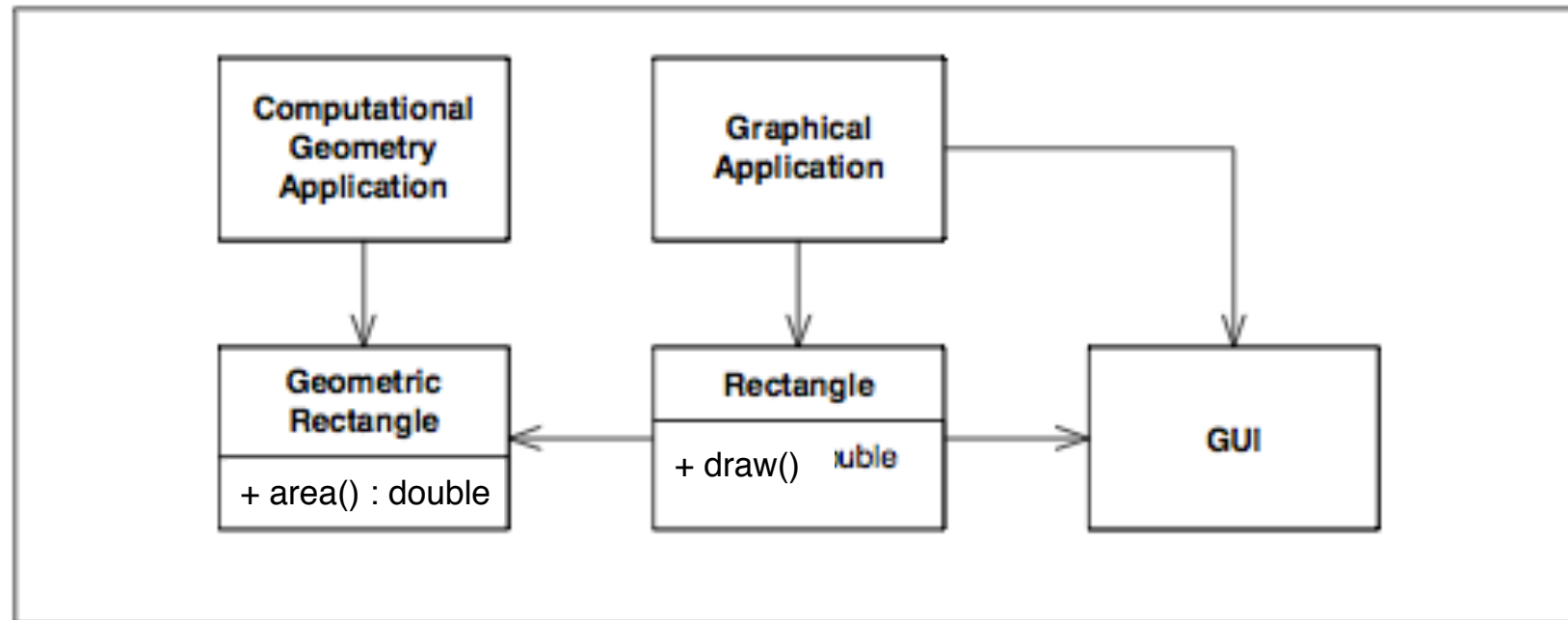
# Violating SRP



**Figure 9-1**  
More than one responsibility

This design violates the SRP. Discuss the consequences.

# Redesign to comply with the SRP



**Figure 9-2**  
Separated Responsibilities

The Rectangle class has been split into two, representing the different responsibilities involved (mathematical model of rectangle and graphical rectangle).

# Another SRP Violation

Don't forget the 'entirely encapsulated' clause!

Look at the sketch of a Person class below.

This class should have an **equals** method that checks if the argument object is a Person and if it has the same PPS number

```
class Person{  
    ...  
  
    private String pps; // unique identifier  
}
```

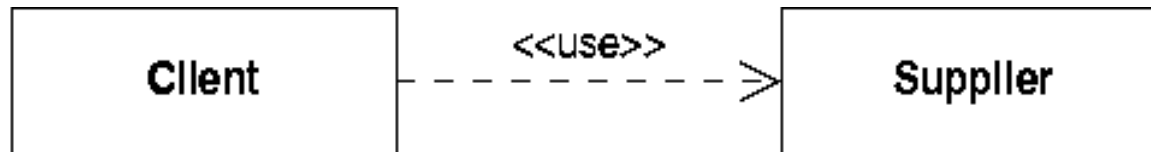
What are the practical consequences if this class doesn't have an **equals** method? In the context of a real software project, how will the software evolve?

# Object-Oriented Principles: Roadmap

- Single Responsibility Principle (SRP)
- **Open/Closed Principle (OCP)**
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)
- No Concrete Superclasses
- The Law of Demeter

# Dealing with **change**

- How can we insulate modules from change?



- We would like to be able to change modules without affecting their clients
  - this will increase the maintainability of the system

# Open and Closed Modules

- A module (class, package, ...) is said to be **closed** if it is not subject to further change
  - clients can then rely on the services the module provides



- A module is said to be **open** if it is available for extension
  - this will make it possible to extend what the module does

# The Open/Closed Principle

- The open/closed principle states that developers should try to produce modules that are both open and closed.
  - closed for modification
  - open for extension
- This sounds like a paradox
  - to resolve it we need a way of extending a module without changing its code
  - We'll see several ways to achieve this in the coming slides

# Interface and Implementation

- The **interface** of a module is what is visible or available to clients of the module
  - public methods
- The **implementation** is the internal details of the module that support the interface
  - private/protected methods/fields
- One approach to open/closed modules is to ensure that clients only depend on the **interfaces** of the modules that they use
  - makes it possible to change implementations easily
  - may also change the behaviour of the client



# Try using Public and Private features

- **Data abstraction** assigns an access level (public/private) to each feature in a class
- Clients can use only the **public** interface



- Thus **private** features can be freely changed without affecting clients

# Example in Code

```
class Driver {  
    public void drive(){  
        car.startEngine();  
        car.pressAccelerator();  
        ...  
    }  
    private FordEscort car;  
}
```

Client

```
class FordEscort {  
    public void startEngine(){  
        ...  
    }  
    public void pressAccelerator(){  
        ...  
    }  
    private float currentSpeed;  
    private boolean doorOpen;  
    ...  
}
```

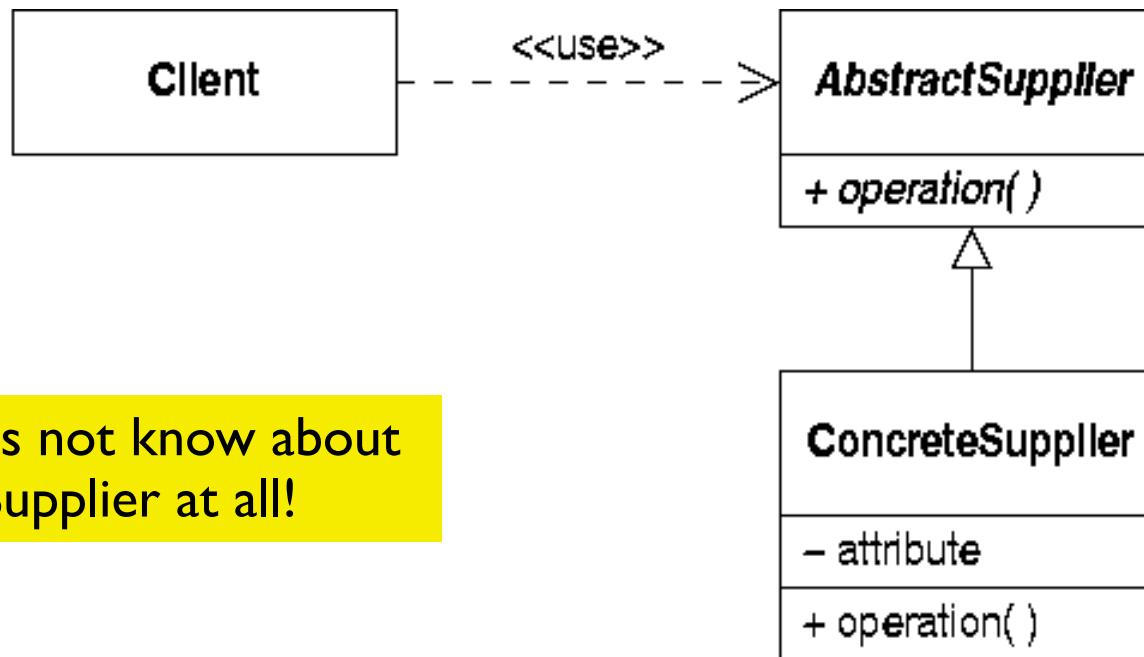
Supplier

# This approach is limited

- Even with data abstraction, Java classes are technically not closed
  - changes to method bodies affect the source code and entail recompilation of clients
- The interface actually used by clients is left implicit
  - The client is still dependent on the **actual class** of the supplier.
- Changing to a new supplier is impossible without also changing the client.

# Make the Client less dependent on the Supplier

- A more flexible approach based on interfaces/abstract classes:



Client does not know about ConcreteSupplier at all!

# Example in Code

```
class Driver {  
    public void drive(){  
        car.startEngine();  
        car.pressAccelerator();  
        ...  
    }  
    private Car car;  
}
```

Client

```
interface Car {  
    public void startEngine();  
    public void pressAccelerator();  
    ...  
}
```

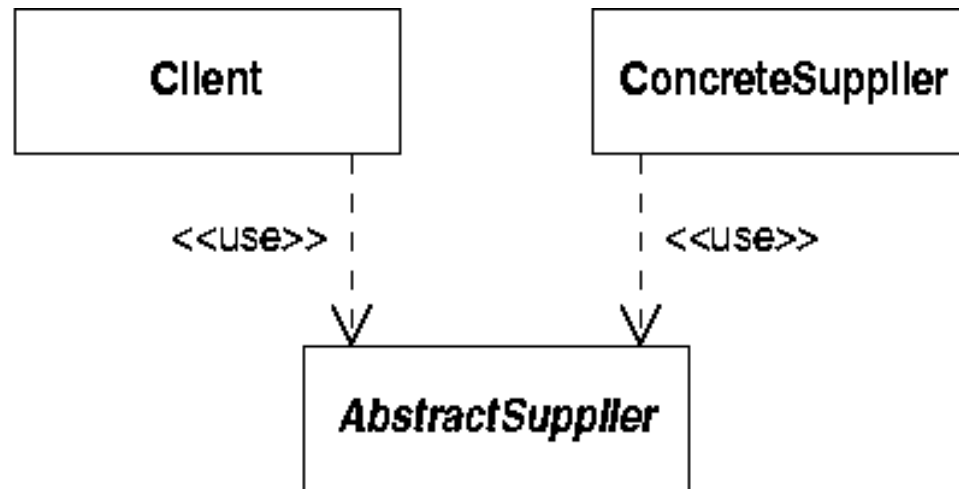
Abstract  
Supplier

```
class FordEscort implements Car {  
    public void startEngine(){  
        ...  
    }  
    public void pressAccelerator(){  
        ...  
    }  
    private float currentSpeed;  
    private boolean doorOpen;  
    ...  
}
```

Concrete  
Supplier

# Looking at the dependency structure

- The **dependency structure** indicates that the client is independent of the concrete suppliers



- Whenever you use e.g. an interface in Java, this is the dependency structure that is actually created.

# Open and Closed

- The Client class is
  - **open** in the sense that its functionality can be extended by adding further supplier subclasses
  - **closed** in that this can be done without changing the client
- However, the client is not protected against changes in the interface exported by the abstract supplier class
  - Something has to be set in stone!

# What to leave open, what to close?

- It may be impossible to predict in what ways a class will be extended.
- In general it's a bad idea to makes classes more open than they need to be.
  - this is **overengineering**
- Current practice is to develop classes in a rather closed style, but apply the OCP as soon as it is necessary.

Many design patterns are related to this principle. Think about this later on in the module.



# Example

```
public class Rectangle
{
    public double Width { get; set; }
    public double Height { get; set; }
}
```

# Example

```
public class AreaCalculator
{
    public double Area(Rectangle[] shapes)
    {
        double area = 0;
        foreach (var shape in shapes)
        {
            area += shape.Width*shape.Height;
        }

        return area;
    }
}
```

OK so far?

# Example

```
public double Area(object[] shapes)
{
    double area = 0;
    foreach (var shape in shapes)
    {
        if (shape is Rectangle)
        {
            Rectangle rectangle = (Rectangle) shape;
            area += rectangle.Width*rectangle.Height;
        }
        else
        {
            Circle circle = (Circle)shape;
            area += circle.Radius * circle.Radius * Math.PI;
        }
    }

    return area;
}
```

Can we **extend** this class  
with **modifying** it?

# Example

```
public abstract class Shape
{
    public abstract double Area();
}
```

# Example

```
public class Rectangle : Shape
{
    public double Width { get; set; }
    public double Height { get; set; }
    public override double Area()
    {
        return Width*Height;
    }
}
```

```
public class Circle : Shape
{
    public double Radius { get; set; }
    public override double Area()
    {
        return Radius*Radius*Math.PI;
    }
}
```

# Example

```
public double Area(Shape[] shapes)
{
    double area = 0;
    foreach (var shape in shapes)
    {
        area += shape.Area();
    }

    return area;
}
```

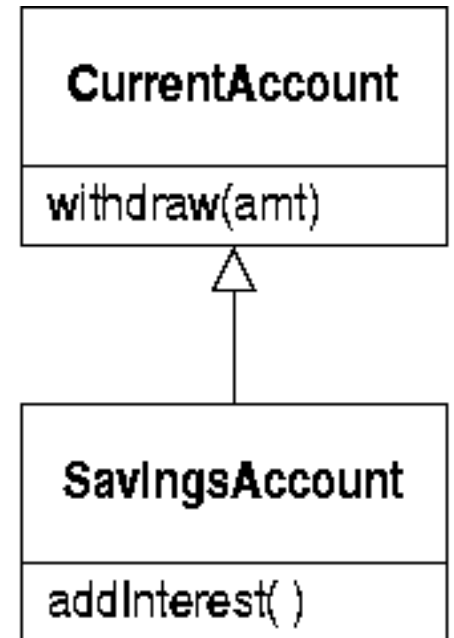
Can we **extend** this class  
with **modifying** it?

# Roadmap

- Single Responsibility Principle (SRP)
- Open/Closed Principle (OCP)
- **Liskov Substitution Principle (LSP)**
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)
- No Concrete Superclasses
- The Law of Demeter

# Liskov Substitution Principle

- States the conditions under which references to superclasses can safely be converted to references to subclasses
- “it must be possible to substitute an instance of a subclass for an instance of a superclass without affecting client classes or modules”
- this defines what **generalisation** means in object-oriented design





# Liskov Substitution Principle

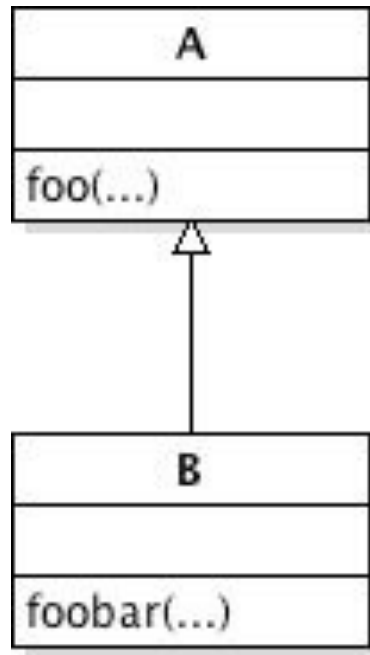
- The **Liskov Substitution Principle** roughly is this
  - “Any context that expects a member of a generalised class must work correctly when provided with an appropriate object of the more specialised class.
  - This has a major impact on how we can override, in a subclass, an operation defined in a superclass
- As a slogan (from the perspective of the subclass):
  - **Demand no more, promise no less!**

# Liskov Substitution Principle

- To state it more formally:
  - an overriding operation must have a weaker (or equal) precondition and a stronger (or equal) postcondition than the one it overrides.
- Pre- and post-conditions?
  - Fancy names for very simple concepts
  - **Precondition:** what the method demands for it to run properly (“give me a non-negative float”)
  - **Postcondition:** what the method promises to do assuming its precondition is true (“I’ll return its square root”)
- Is the Liskov principle surprising?
  - No, but it has some surprising consequences that we now explore.

# Demonstrating Liskov: simple case

Say we have a class A and a subclass B. If B just inherits from A and doesn't remove or override any of its methods, then Liskov substitutivity principle holds, once any new methods in B preserve the class invariant\* of A.



\***class invariant**: a predicate on the fields of a class that should be maintained, e.g.,  $(1 \leq \text{month} \leq 12)$ .

# Now consider overriding...

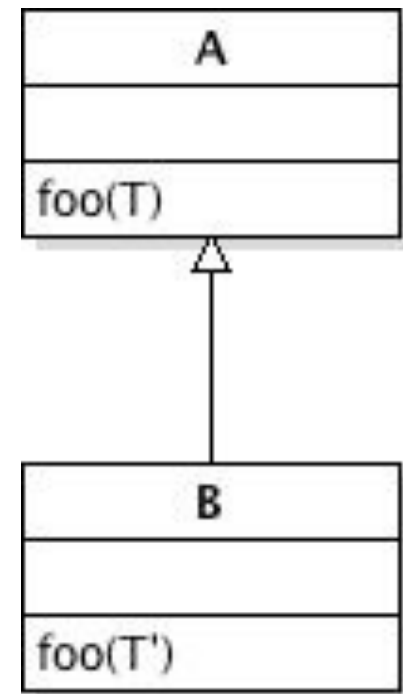
Now we consider method overriding. Our aim is to work out how method overriding must be constrained in order to ensure that the Liskov substitution principle is maintained.

B is a specialisation of A.

A defines the method `foo` which takes a parameter of type `T`.

B overrides `foo`, and `B::foo` takes an argument of type `T'`.

We now work out the relationship that must exist between `T` and `T'` in order for substitutivity to be supported.



# Demonstrating Liskov: substitution

In order to make it clear what is meant by substitution, we consider the following code fragment (Java-ish):

```
A a = new A();  
...  
if (isSunnyDay())  
    a = new B();  
...  
a.foo(t); // so a may refer to an A or a B object!
```

Consider the argument to the message send. Statically it appears to invoke the `A::foo` method, so the programmer will supply an object `t` of type `T`. However....

# Demonstrating Liskov: contravariance

... a may refer to a B object. This means that B::foo must also be able to work correctly with an argument of type T.

The type of argument to B::foo is in fact T'.

=> T must be a subtype of T'.

(Don't forget that the subtyping relation is reflexive.)

This is surprisingly. In the overriding method in the more specialised class, the argument types must remain the same or be made more general.

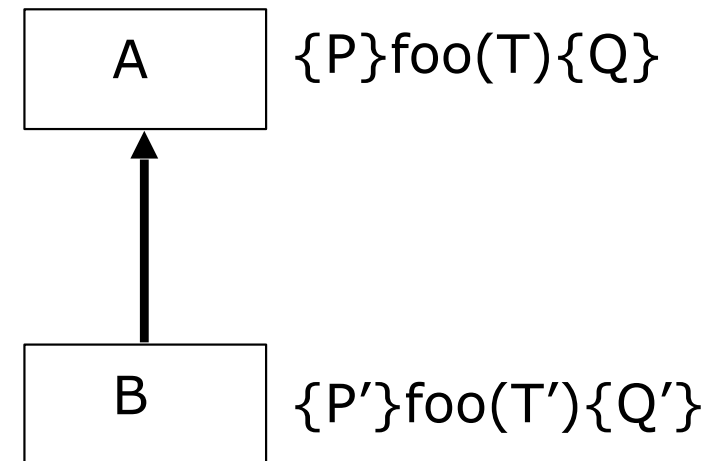
This is termed **contravariance**. The alternative, **covariance**, may lead to run-time type errors if substitutivity is permitted.

# Pre- and post-conditions

Using a similar argument, we can show that the return type of the overriding method must remain the same or be made more specialised.

We have only considered the issue of types. Looking at the method definitions in terms of pre- and postconditions, a similar argument to the above will lead us to:

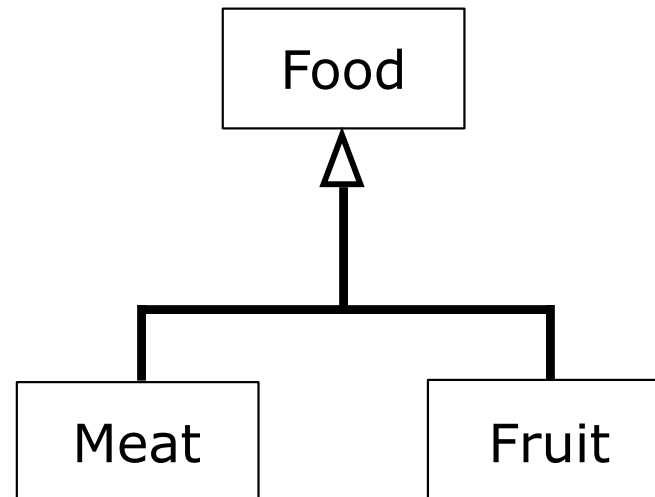
$P'$  must be weaker than  $P$  and  
 $Q'$  must be stronger than  $Q$ .



# A concrete example

See the 47480 web page for this example in Java code.

We assume this simple inheritance hierarchy already exists:



The preceding 5 slides may seem too abstract. If to, you can just focus on this concrete code example.



# Animals eat food

Animals eat food:

```
class Animal{  
    public void eat(Food f){...};  
}
```

Nothing controversial here!

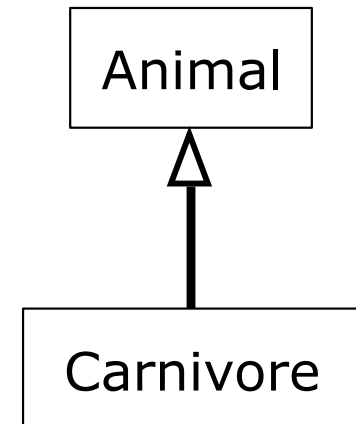
# Carnivores eat meat

Carnivores are a type of animal, so we create a class Carnivore, a subclass of Animal.

Carnivores eat a type of food, namely meat, so we override the eat method in Carnivore to accept only meat.

```
class Carnivore extends Animal{  
    public void eat(Meat m){...};  
}
```

All ok? It looks fine...



# That violated the Liskov substitutivity principle!

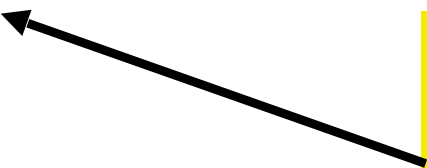
This will lead to problems when substitution occurs, i.e. when a context that is expecting an instance of the superclass (Animal) is given an instance of the subclass (Carnivore) to work with.

Here's how we construct such an example:

```
Fruit apple = new Fruit();  
Carnivore rex = new Carnivore();  
Animal my_animal = new Animal();
```

```
if (coin_toss() == heads)  
    my_animal = rex;
```

```
my_animal.eat(apple);
```



This creates a run-time type error whenever `my_animal` refers to an instance of `Carnivore`.

# What went wrong?

We overrode the `eat` method and made its argument more specific, i.e. we “demanded more” in the overriding method.

Note the code snippet above as it shows how this permits the new method to run without the new demands being met, hence a run-time type error occurs.

# Liskov in Practice

Most languages demand that the argument type in an overriding method be **contravariant**. Java is even more specific — the arguments must be of the same type.

**Covariance** permits run-time type errors to occur. Eiffel is the only ‘mainstream’ language that permits covariance.

As a programmer you should observe the following:

- When adding a new method to a subclass, take care not to break any invariants (documented or implicit) of the superclass.
- An overriding method should *extend* the method it overrides, not do something entirely different
  - One the best ways of achieving this is to *invoke* the overridden method in the overriding method.

# Liskov Summary

The Liskov Substitution Principle explains what substitution means in the context of inheritance.

This principle limits what you can do in a subclass, e.g.,

- inherited methods cannot be rejected
- Overriding methods must have contravariant arguments

Most object-oriented languages try to prevent you from violating the Liskov principle, e.g., in Java:

- you cannot reject an inherited method
- the arguments to an overriding method must be of the same type as the arguments to the overridden method.

# Roadmap

- Single Responsibility Principle (SRP)
- Open/Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- **Interface Segregation Principle (ISP)**
- Dependency Inversion Principle (DIP)
- No Concrete Superclasses
- The Law of Demeter

# Interface Segregation Principle

The Interface-Segregation Principle (ISP) states simply that a client should not be forced to depend on methods it does not use.

Another way of thinking about this is to say: **interfaces belong to clients.**

We'll look at some examples.



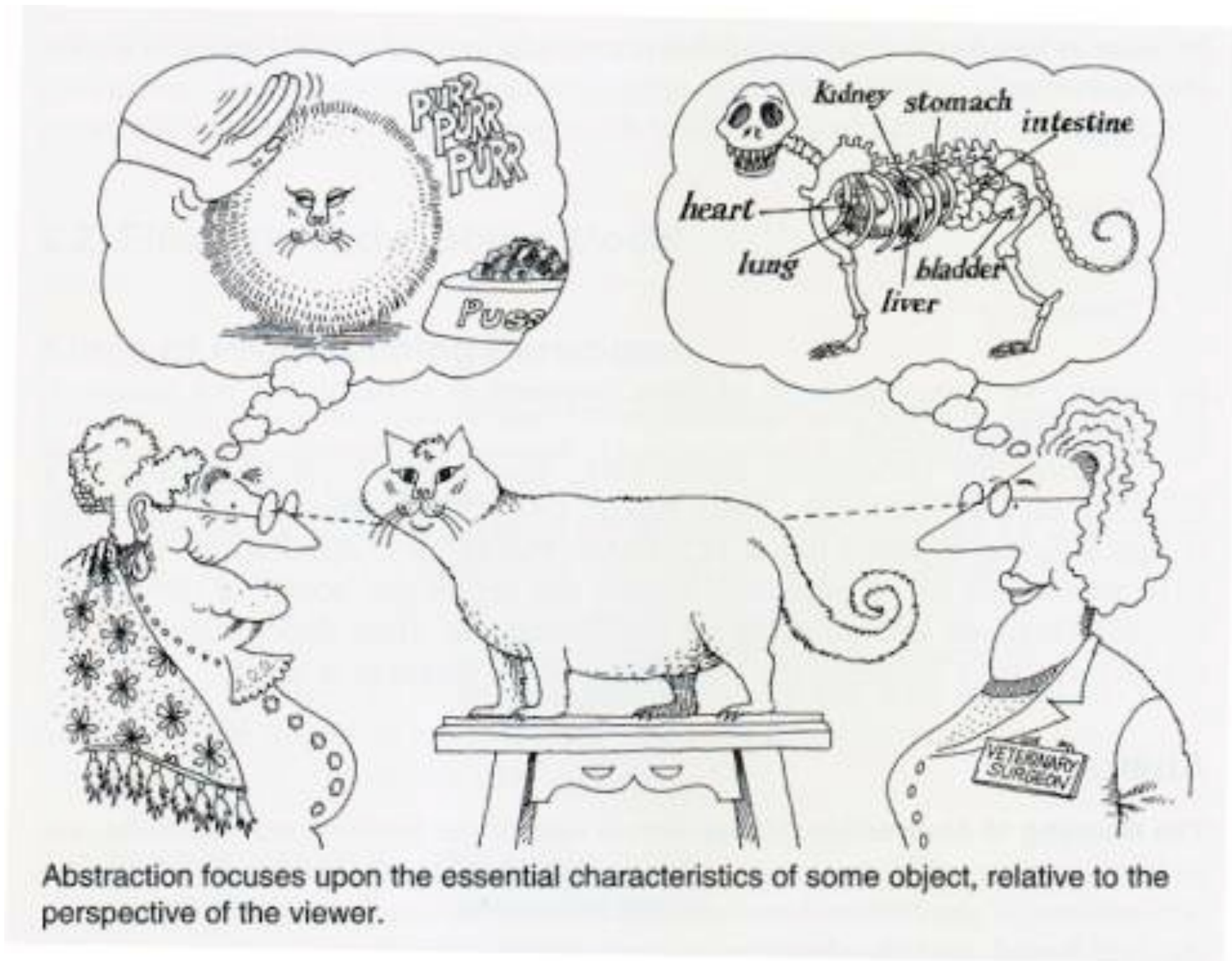


# Interface Segregation Principle

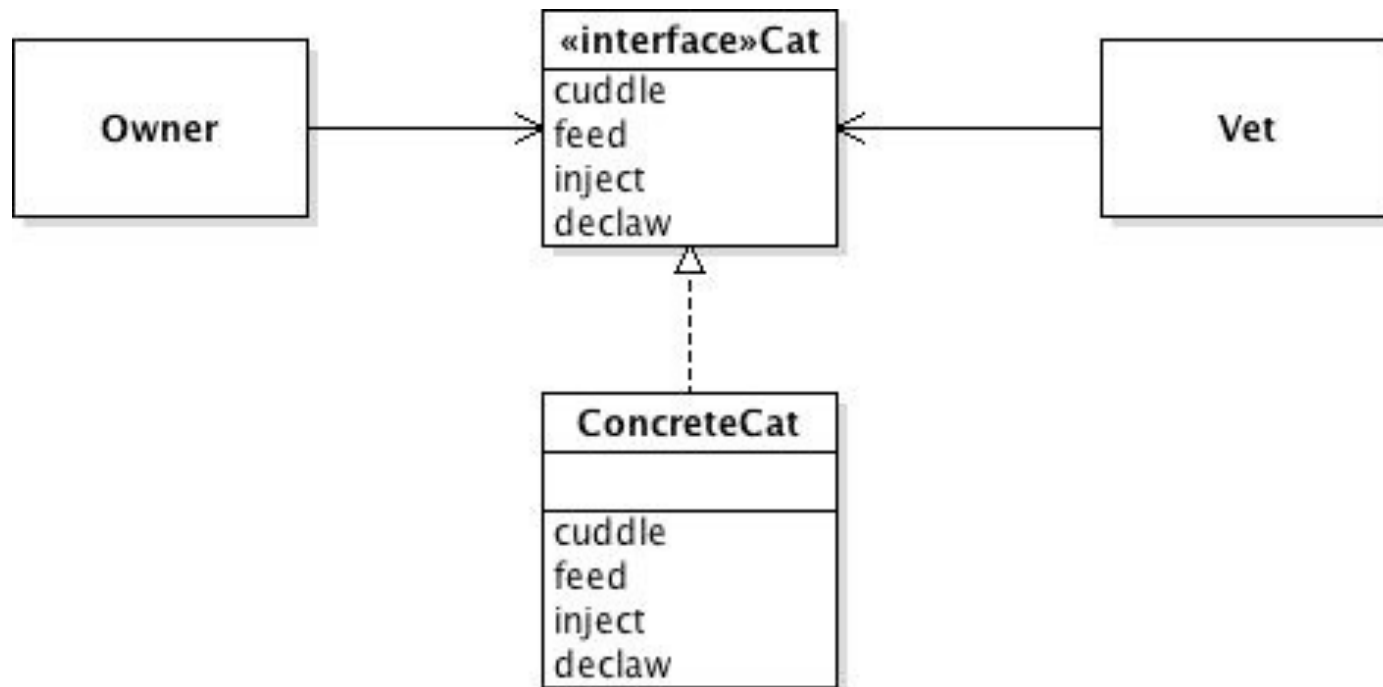
---

Tailor your Interfaces to the Client's Specific Requirements

# Interfaces are in the eye of the Client

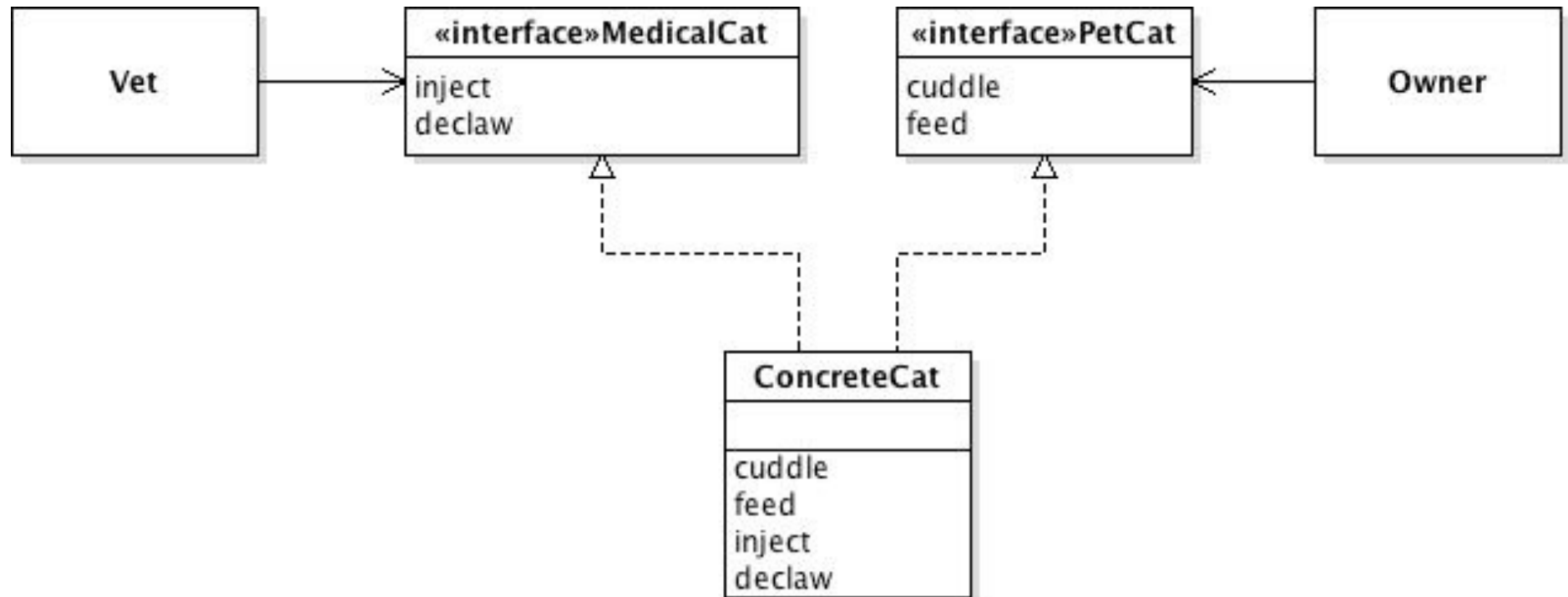


# Cat example without ISP



Owners and Vets can do what they want with the Cat, but each sees an interface that is too wide.

# Cat example with ISP



Again, Owners and Vets can do what they want with the Cat, and each sees only the interface they need to use.

# Roadmap

- Single Responsibility Principle (SRP)
- Open/Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- **Dependency Inversion Principle (DIP)**
- No Concrete Superclasses
- The Law of Demeter

# Dependency Inversion

The Dependency Inversion principle can be simply stated as follows:

*Abstractions should not depend on details.*

How is that **inversion**?

We tend to think of systems in a **top-down** way, where the higher levels depend on the lower ones.

But then changes in the lower levels cause the higher levels to change (“the tail wagging the dog”).

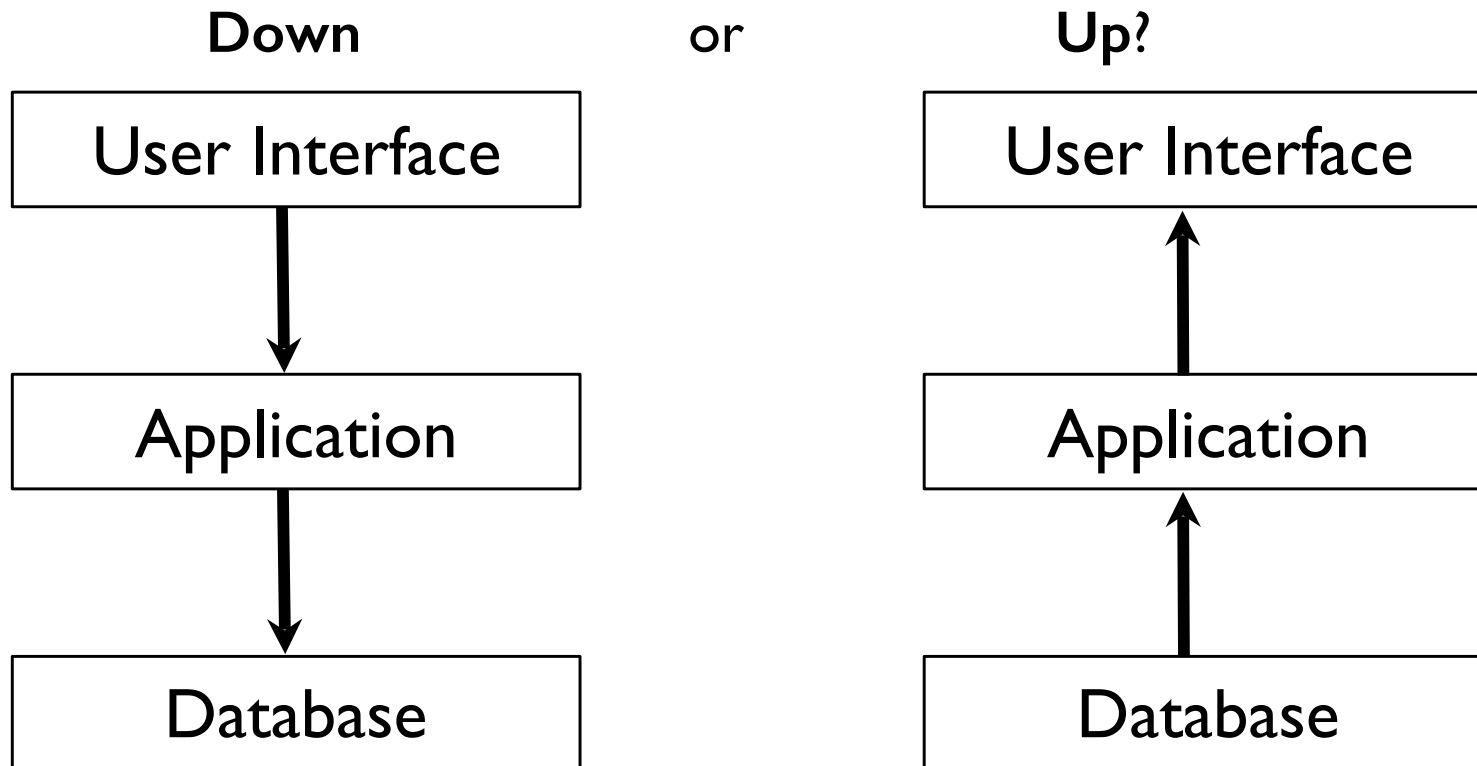
Dependency Inversion means that we invert dependencies to make them **bottom-up**.

# Layered Architecture

We often build system using a **layered architecture**, e.g.

- user interface layer
- application layer
- database layer

How do the dependencies go in a layered architecture like this?



# Dependency Inversion Example

We tend to build systems so that the upper layers depend on the lower ones. It seems natural in some way...

However this means that if the lower levels change, the upper layers must change as well. This is highly undesirable.

What is the dependency relationship between an electrical device and the wall socket??



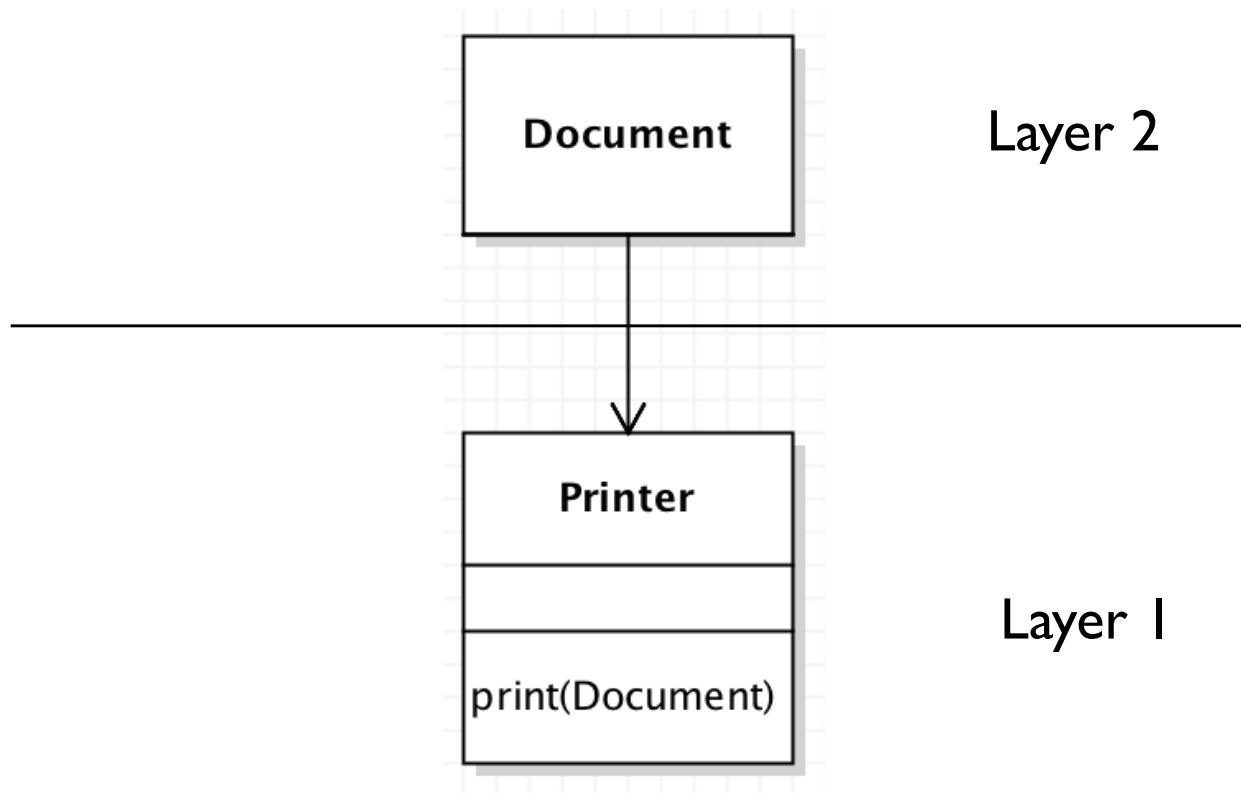


# DEPENDENCY INVERSION PRINCIPLE

Would You Solder A Lamp Directly To The Electrical Wiring In A Wall?

# Top-down dependency...

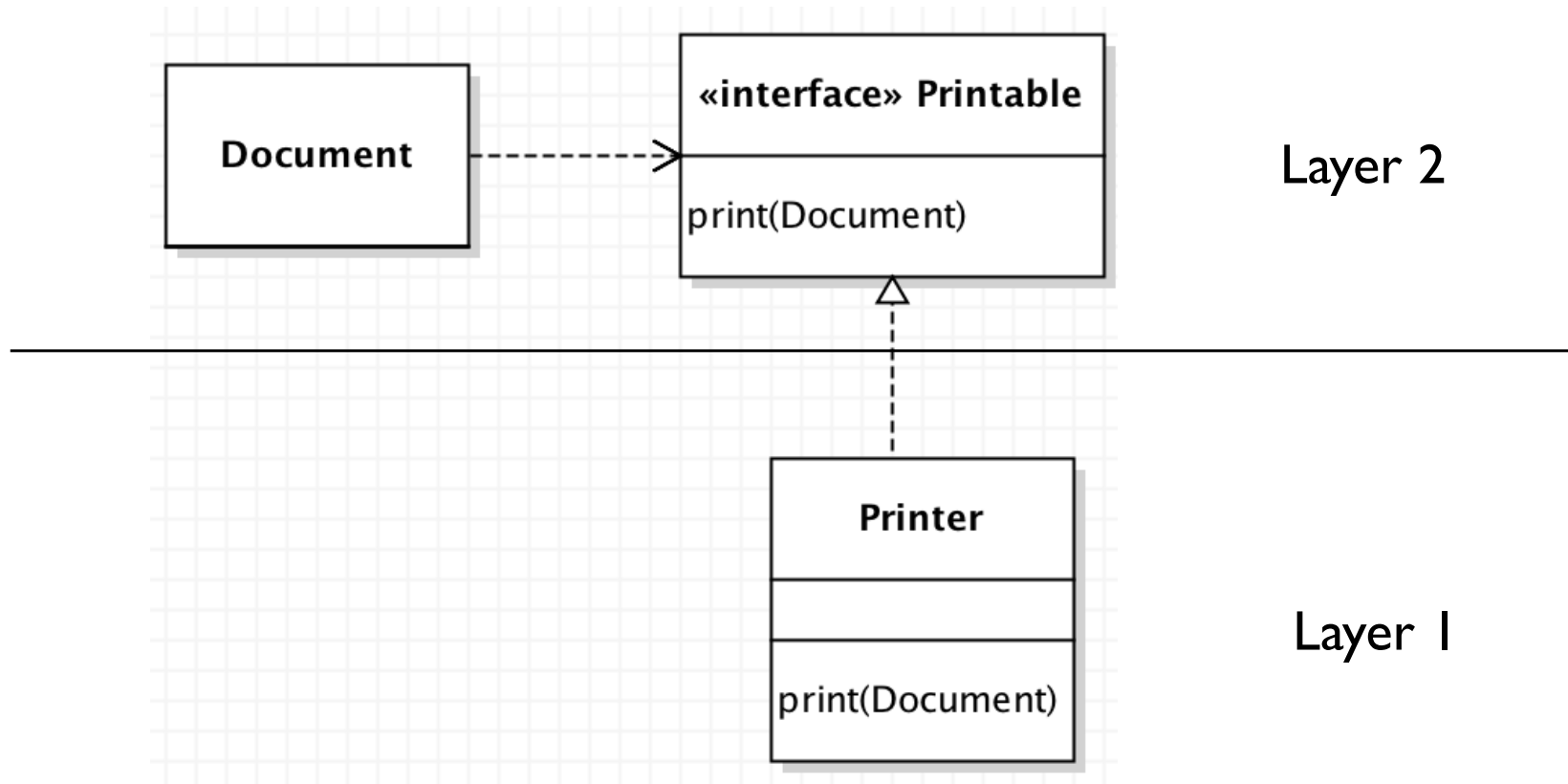
Say a system has a top-down dependency as follows:



How can we invert the dependency from Layer 2 to Layer 1?  
It may seem impossible...

# Inverted to bottom-up dependency

We can however flip the dependency this way:



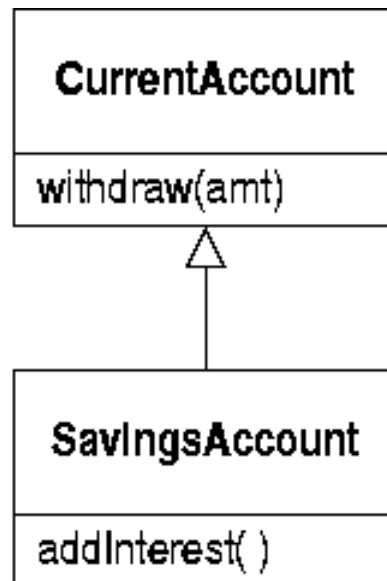
Note that at run-time, a **Document** object will still have a reference to a **Printer** object (through the interface), but the top-down code dependency has disappeared.

# Roadmap

- Single Responsibility Principle (SRP)
- Open/Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)
- **No Concrete Superclasses**
- The Law of Demeter

# No Concrete Superclasses

- It is usually recommended that all superclasses in a system be abstract
  - This guideline is actually an example of Dependency Inversion
- Subclasses are often added to concrete classes as a system evolves:

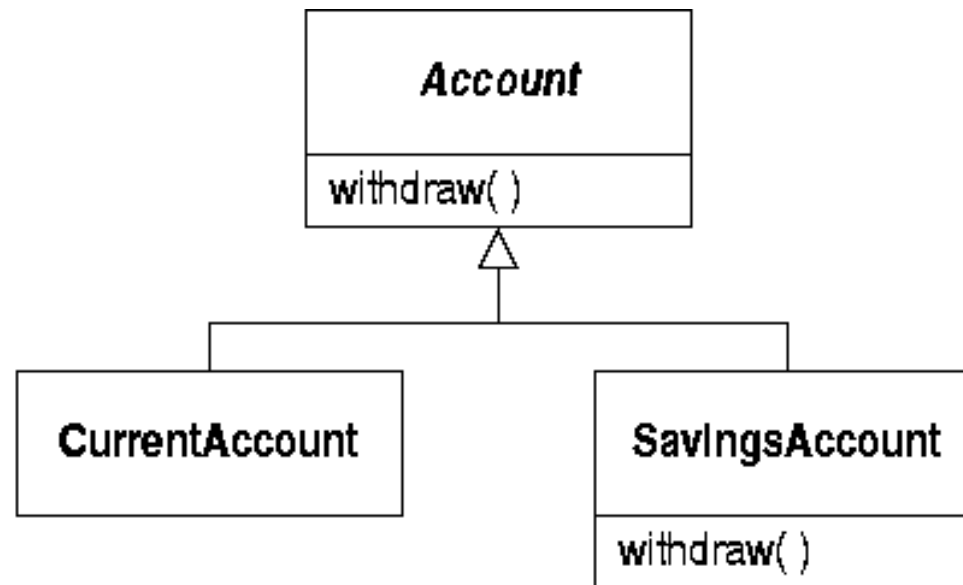


# A Dependency Problem

- The savings account class is now dependent on the current account class
  - changes to the current account -- altering functionality or adding and removing operations -- will affect savings accounts
- This leads to problems
  - the changes may not apply to savings accounts
  - the code may become cluttered with checks for special cases

# Refactoring the Design

- A better approach is to introduce an abstract class
  - even if it initially seems unnecessary



- In a sense, this is a special case of Dependency Inversion
  - In the earlier class model, **SavingsAccount** depended on **CurrentAccount** for its implementation; this has been replaced by a shared dependency on the **Account** abstract class.

# Roadmap

- Single Responsibility Principle (SRP)
- Open/Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)
- No Concrete Superclasses
- **The Law of Demeter**



# The Law of Demeter

What objects should an object  $O$ , on receipt of a message  $m$ , be allowed to send messages to?

If we say "any object," we may end up with a very complicated model.

The so-called "Law of Demeter" proposes to limit it to:

- $O$  itself;

- objects that are passed as arguments to  $m$ ;

- objects that are created in  $m$ ;

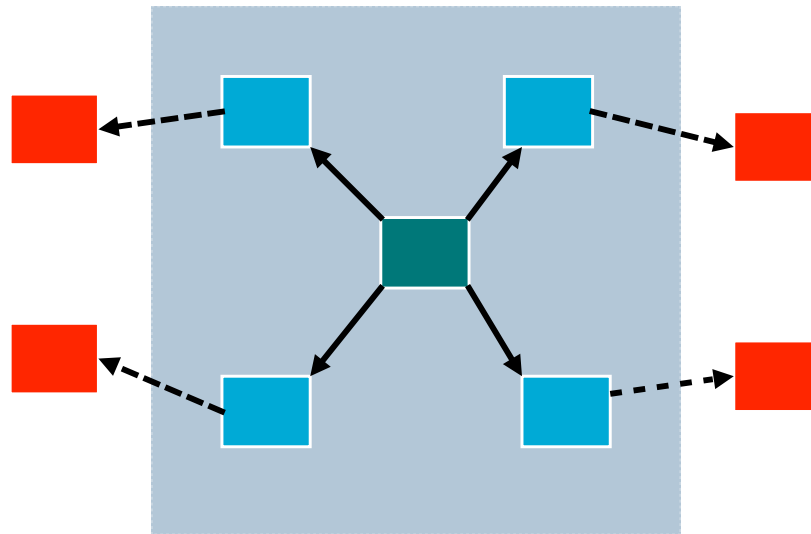
- objects that are attributes of  $O$ .

So what does the Law of Demeter actually disallow?

# Who shouldn't an object talk to?

Demeter explicitly disallows extracting a subobject from an accessible object and sending a message to this subobject.

This constrains how much an object can know about its environment (a good thing!). It may only talk to objects "next to it."



The object in the centre may only traverse the links to its immediate neighbours.

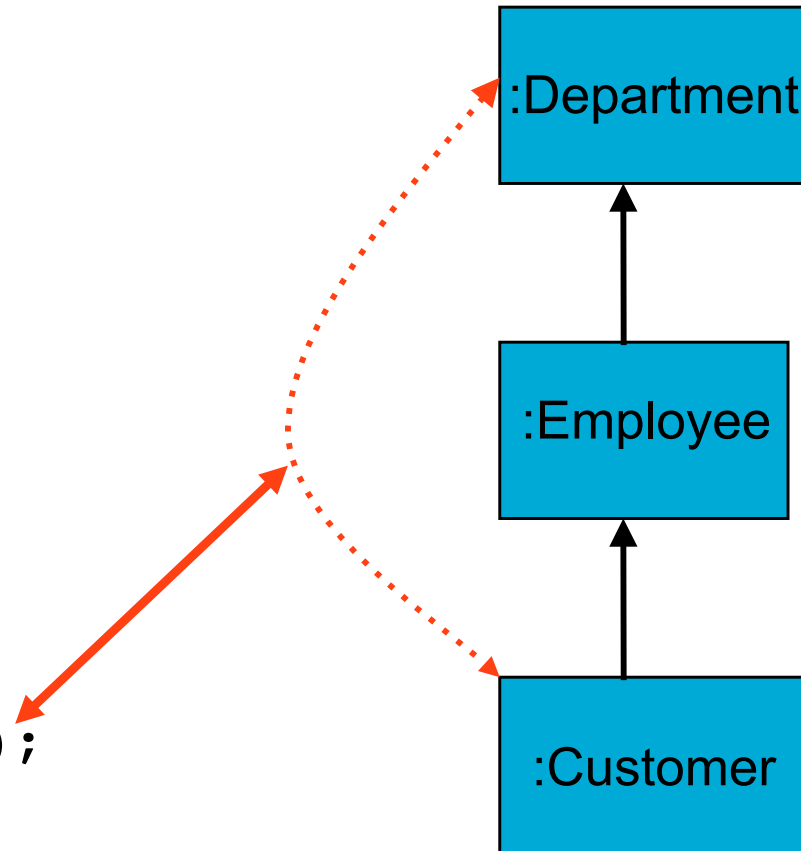
# A Violation of Demeter

A violation of the Law of Demeter in program code might look like this:

```
class Department {  
    double getName();  
}
```

```
class Employee{  
    Department dept;  
}
```

```
class Customer{  
    void foo(){  
        ...  
        emp.dept.getName();  
    }  
    Employee emp;  
}
```



Why is this bad?  
How would you  
solve it? Note  
that making dept  
private won't  
help... why not?

# Refactoring to correct

- Refactoring this code to resolve this issue produces:
- So the Customer class doesn't even know the Department class exists. This may seem like a small matter, but it can be a major simplification in a large system.

```
class Department {
    String getName();
}

class Employee{
    String getDeptName(){
        return dept.getName();
    }
    Department dept;
}

class Customer{
    void foo(){
        ...
        emp.getDeptName();
    }
    Employee emp;
}
```

# Demeter, strong and weak forms

- The **strong form** of the Law of Demeter prohibits access to objects accessed through inherited instance variable.
- This amounts to providing the same level of data abstraction to subclasses as is given to class clients.
- Observing Demeter results in the addition of a lot of small methods, but makes classes much less dependent on external object structures and therefore more flexible.

# Demeter Summary

- The Law of Demeter essentially avoids encoding the details of system structure in individual methods.
- Is widely regarded as a valuable factor in creating loosely-coupled software that is easier to maintain.

# Impact of Software Principles

A very rough measure of impact is number of Google results (April 2017):

Single Responsibility Principle: 742k

Law of Demeter: 95k

Open Closed Principle: 88k

Liskov Substitution Principle: 75k

Dependency Inversion Principle: 74k

Interface Segregation Principle: 68k

(For comparison, “Software Quality” yields about 1 million results.)

# Summary

- Object-Oriented Principles are general statements of properties that a good design should have.
- By way of contrast, patterns are examples of concrete good design in a certain context, and design heuristics are “rules of thumb” that help in creating a good design.
- In this section we looked at several well-known object-oriented design principles.
- Don’t observe these blindly — there are frequently good reasons good reasons to ignore them, but you should be aware of the consequences.