Object-Oriented Principles

Comp 47480: Contemporary Software Development

SOLID Principles

- In this section we'll look at the so-called SOLID set of object-oriented principles identified by Bob Martin in the "early days" of OOP:
  - **S**ingle Responsibility Principle (SRP)
  - **O**pen-Closed Principle (OCP)
  - **L**iskov Substitution Principle (LSP)
  - **I**nterface Segregation Principle (ISP)
  - **D**ependency Inversion Principle (DIP)

- We'll also look at two other principles/guidelines:
  - No Concrete Superclasses
  - Law of Demeter

Roadmap

- Single Responsibility Principle (SRP)
- Interface Segregation Principle (ISP)
- Open-Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- 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

Strongly related to **cohesion**

(ii) that responsibility should be entirely encapsulated by the class.

Related to **coupling**

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.

A Simple SRP Violation

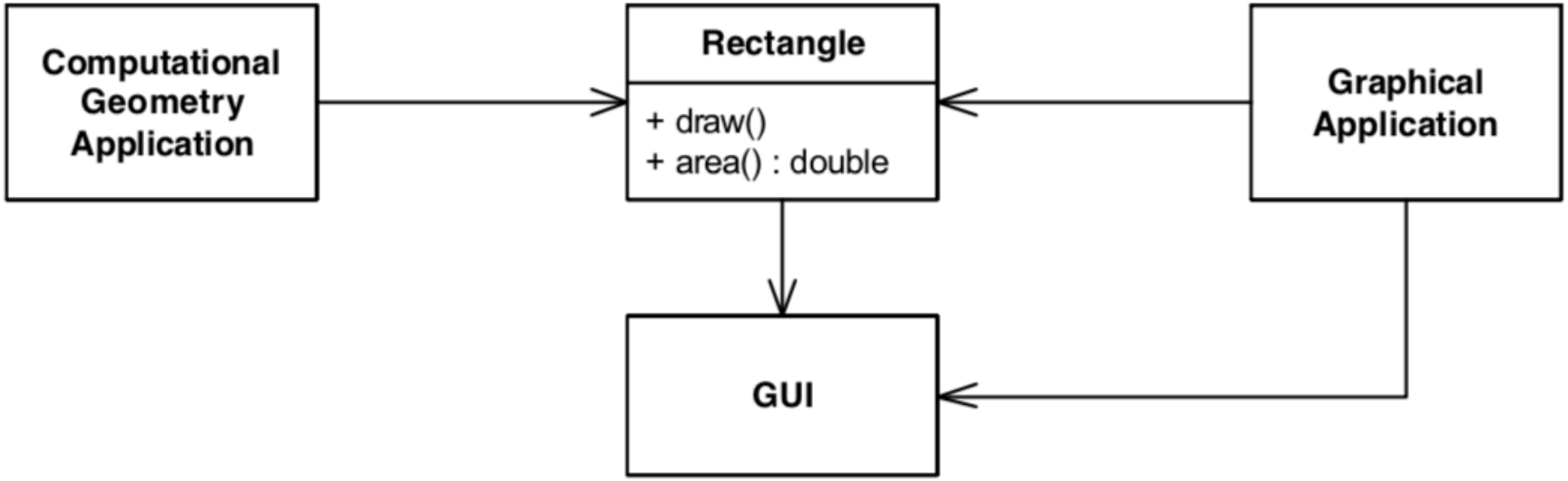
```
class Course {
  void calculateDropoutRate() {
    double rate = ...; // compute dropout rate
    System.out.println(rate);
  }
}
```

It is computing the dropout rate *and* printing the result.

This method has two reasons to change:
(1) a change in how we compute the dropout rate
(2) a change in the UI

See module textbook for discussion of this example.

Violating SRP



The Graphical Application needs to draw and compute area, while the CGA only needs to compute area.

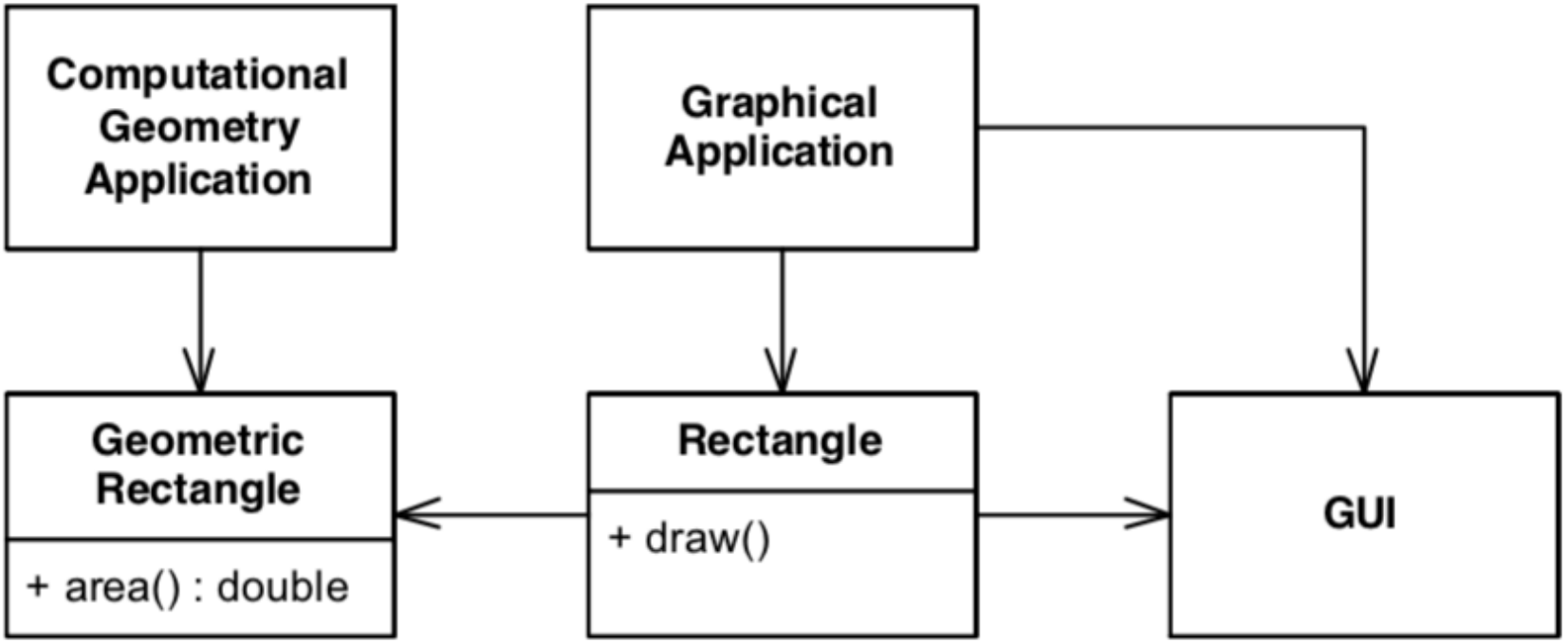
This design violates the SRP.

Why?

Discuss the consequences.

With thanks to Bob Martin for the example.

Redesign to comply with the SRP



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

Another type of SRP Violation

Don’t forget the ‘**entirely encapsulated**’ clause!

This is violated when part of the natural responsibility of a class is placed in another class.

This reduces comprehension, increases coupling and leads to *shotgun surgery* (we'll return to this).

It can also lead to wholesale code duplication, as clients don't find the required functionality where they expect it, and implement it themselves.

Roadmap

- Single Responsibility Principle (SRP)
- Interface Segregation Principle (ISP)
- Open-Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- 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.

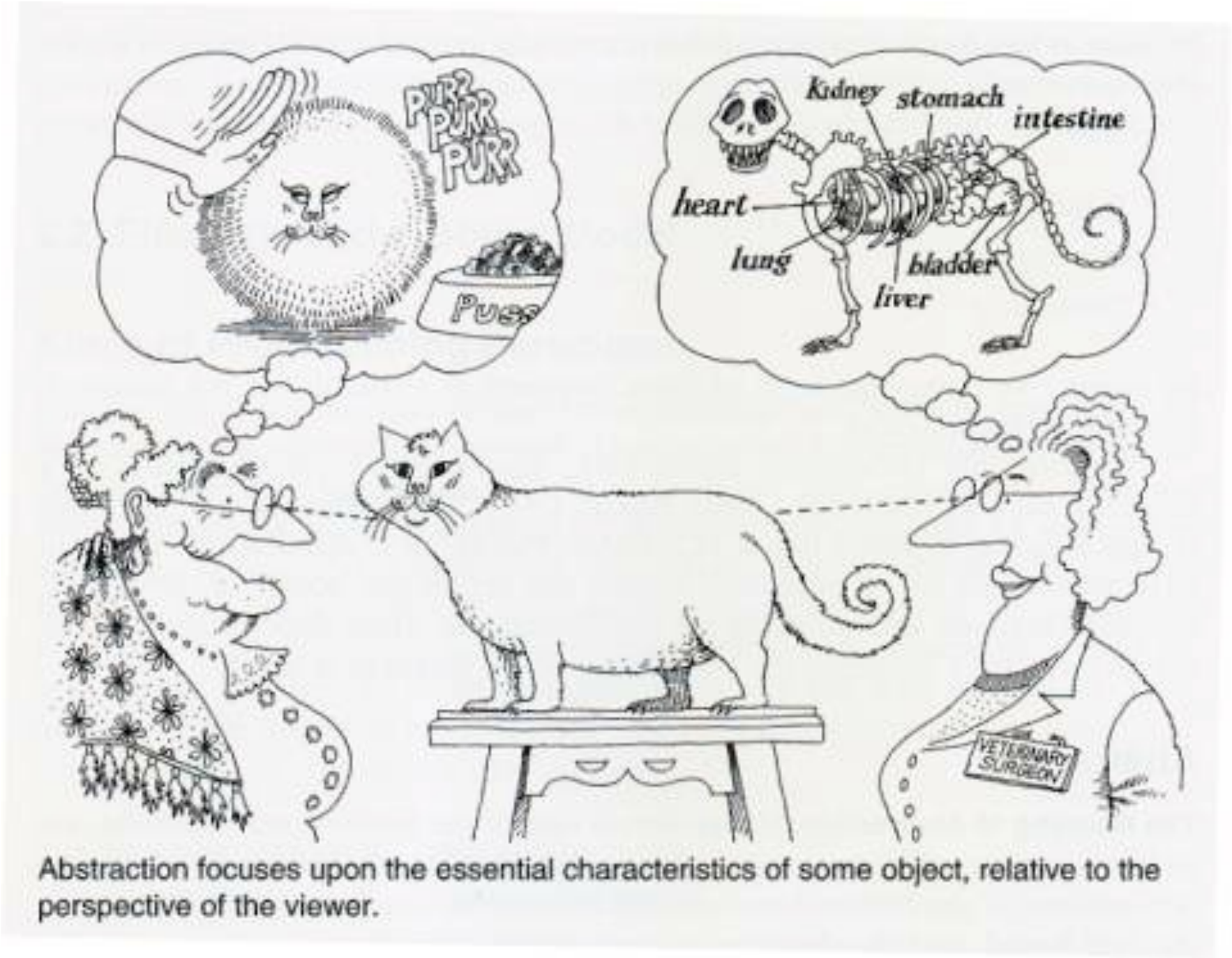
Another way of thinking about this is to say: **interfaces belong to clients**.

We'll look at some examples.

Interface Segregation Principle

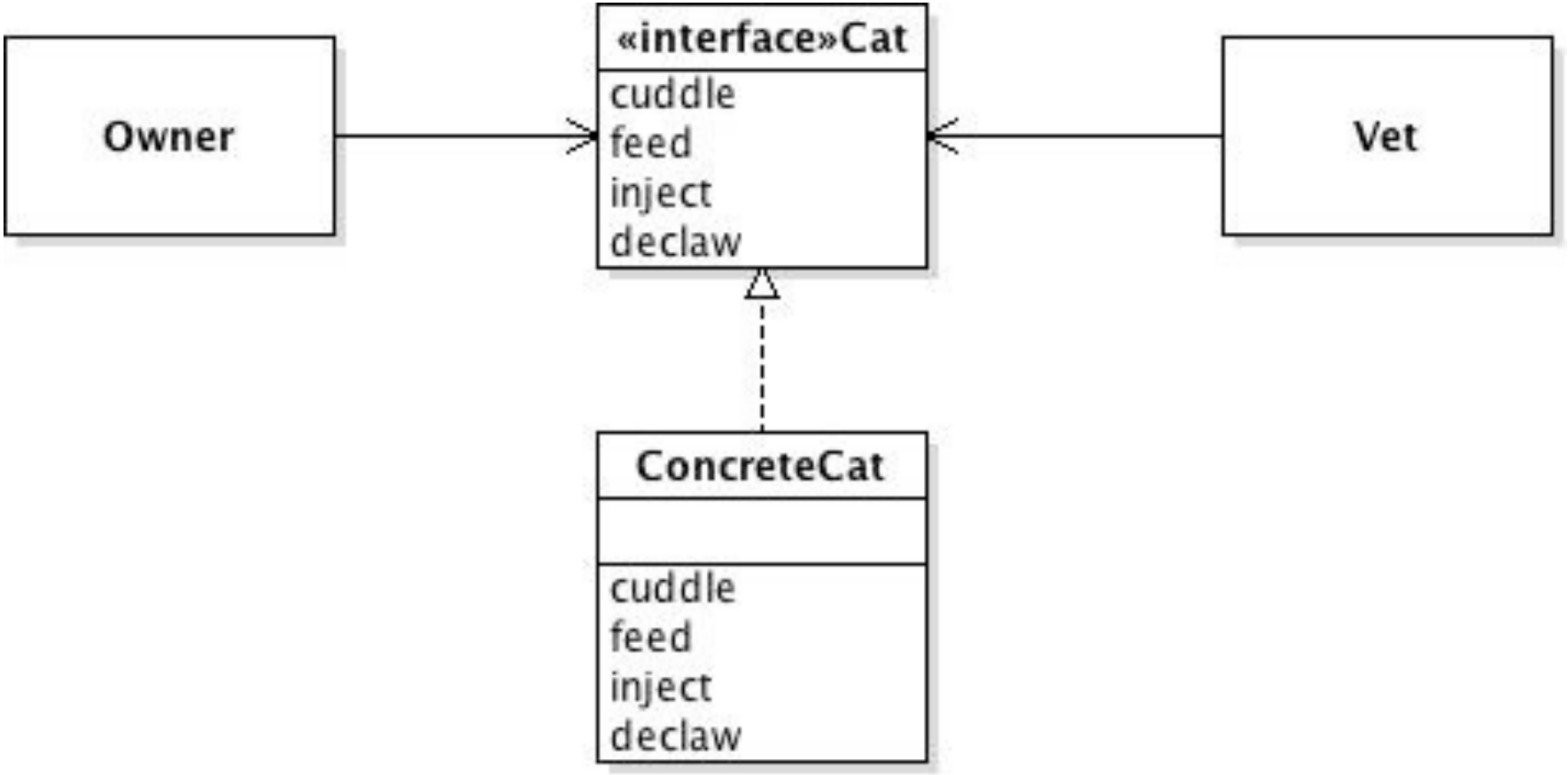
Tailor your Interfaces to the Client's Specific Requirements

Interfaces are in the eye of the Client



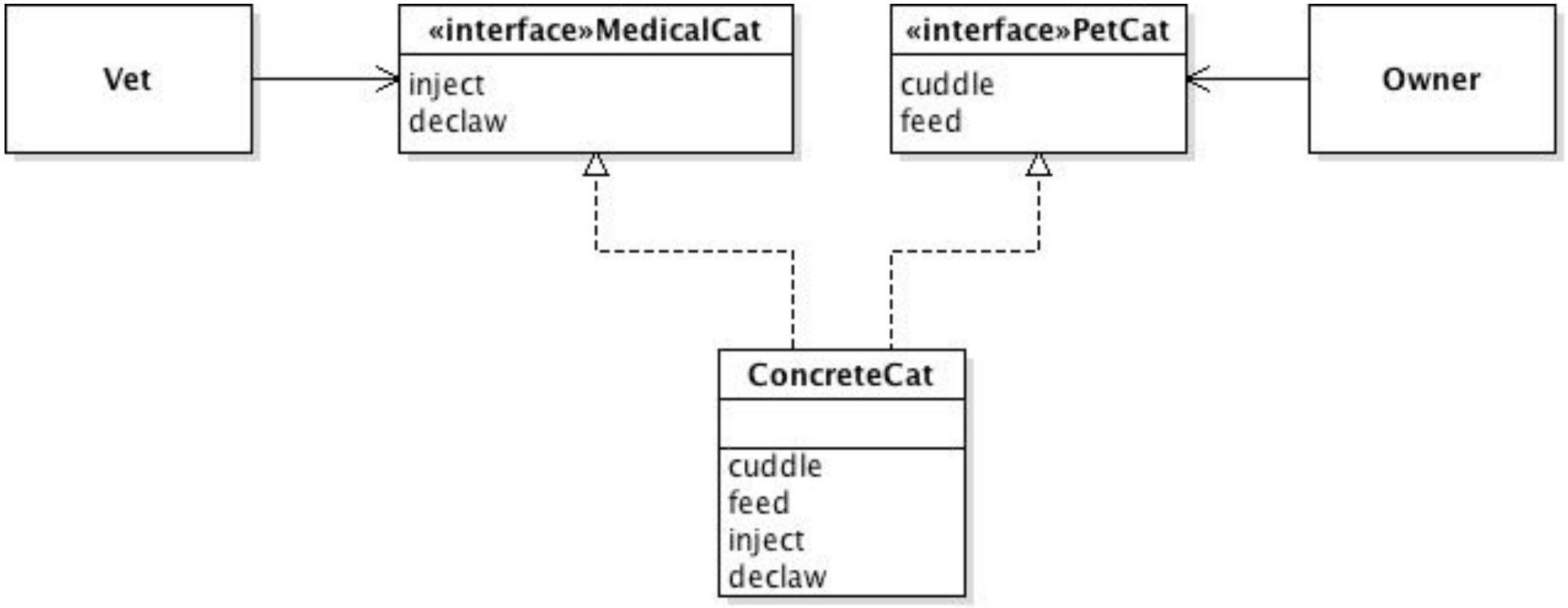
Abstraction focuses upon the essential characteristics of some object, relative to the perspective of the viewer.

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.

Note that the `ConcreteCat` class still looks uncohesive, and could be split to comply better with the SRP.

Object-Oriented Principles: Roadmap

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

Open-Closed Principle

A module is **closed** if it is not subject to modification. Clients can rely on the services the module provides.

A module is said to be **open** if it is available for extension.

Closed is good -- we can depend on it.

Open is good -- we can extend it.

Is is possible to create modules that are both **open** and **closed**?

i.e. *open for extension* and *closed for modification*?

i.e. can we create modules whose behaviour we can change without actually changing the code of the module?

The Open-Closed Principle

The open-closed principle states that developers should try to produce modules that are both open and closed.

- **open** for extension
- **closed** for modification

This sounds like a paradox. To resolve it we need a way of extending what a module does without changing its code.

We'll see several ways to achieve this in the coming slides.

Achieving Open Modules

Many basic programming techniques contribute towards making modules open:

Method Parameters: enable methods to be used in a variety of contexts.

Higher-Order Functions (e.g. `map`, `reduce` etc.): take functions as arguments, which makes them very tailorable.

Type Parameters (e.g. `Stack[T]`): enable classes work with a variety of types.

Inheritance: a method is open to accepting as argument a subclass of the class of the argument. Also applies to interfaces.

Design Patterns: usually open the program to a new *axis of change*. We'll cover design patterns later in the module.

OCP in Action

A naive implementation of the `List` class in Scala:

```
class List[T]:
  def sortWith(before: (T, T) => Boolean): List =
    ...

  def reduce(f: (T, T) => T): T =
    ...
```

The `List` class is **open** to storing objects of any type `T`.

The `sortWith` method is **open** to sorting in any order, as given in the `before` argument.

The `reduce` hof is **open** to performing any kind of reduction, as given in the `f` argument.

Axes of Change

The notion of *axis of change* of often used in the context of the Open-Closed Principle.

We visualise the program as a point in a vector space, where each dimension represents a type of change.

E.g the List class is **open** to change on these axes:

- to create a `List` of a new type of object
- to sort the `List` in a new way
- to reduce the `List` in a new way

Other the other hand, it is **closed** to changing the data structure employed to store the list.

(we could open it on this axis using the Bridge design pattern.)

Example of using Inheritance for OCP

```
class Driver {
  private FordPuma car;
  ...
  public void drive(){
    car.startEngine();
    car.accelerate();
    ...
  }
}
```

Client

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

Supplier

Problems with this Model?

Firstly, if drivers only ever drive Ford Puma cars, there is nothing wrong with this model!

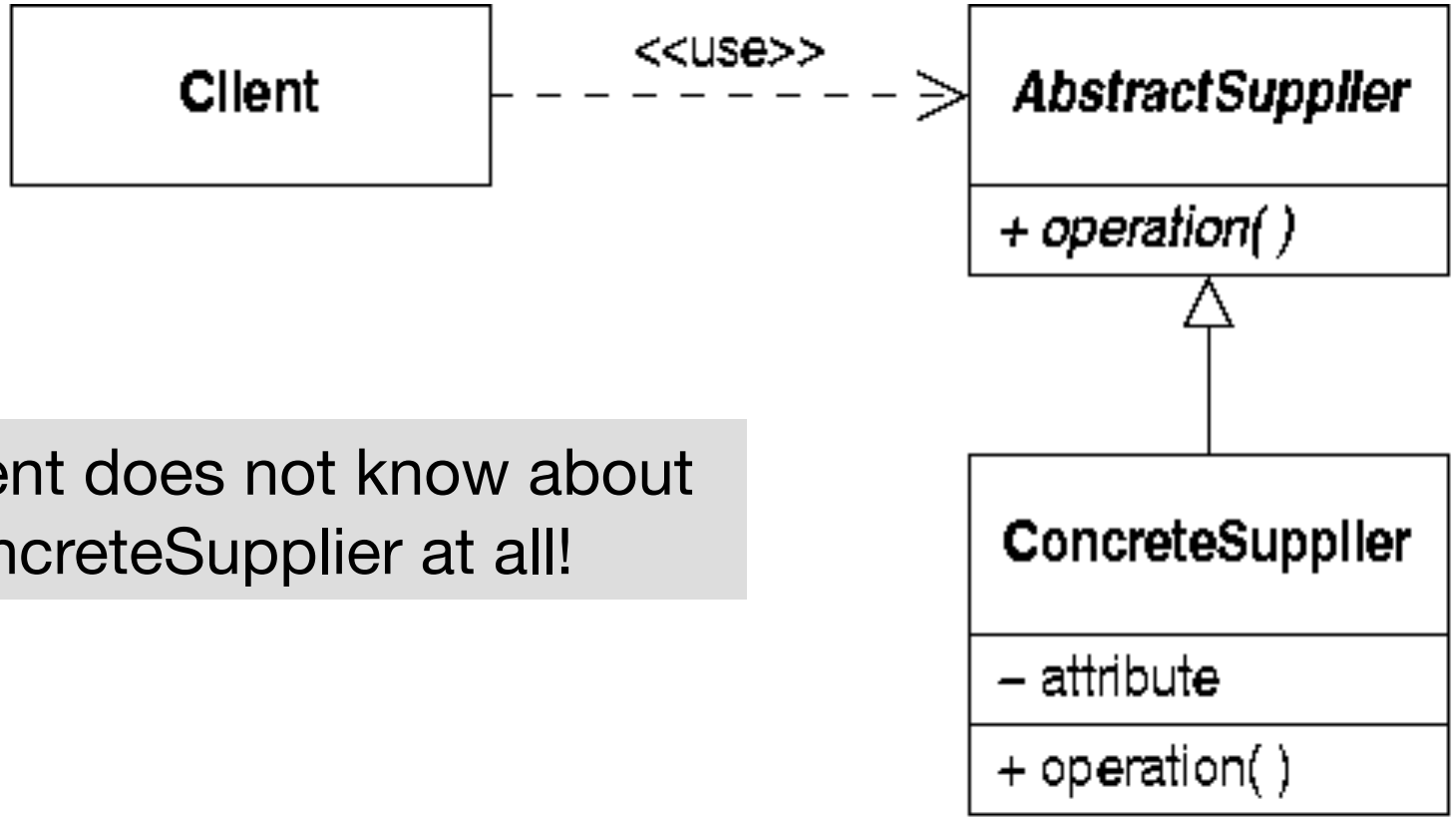
However if we envisage that drivers may drive other car types, this is a poor design.

The client (`Driver`) is dependent on the **actual class** of the supplier (`FordPuma`).

This means that the `Driver` class is **closed** to being evolved to drive a new type of car.

Applying OCP to Reduce Dependency on Supplier

A more flexible approach is one based on an interfaces/ abstract class:



Client does not know about ConcreteSupplier at all!

Applying OCP in Code

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

Client

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

Abstract Supplier

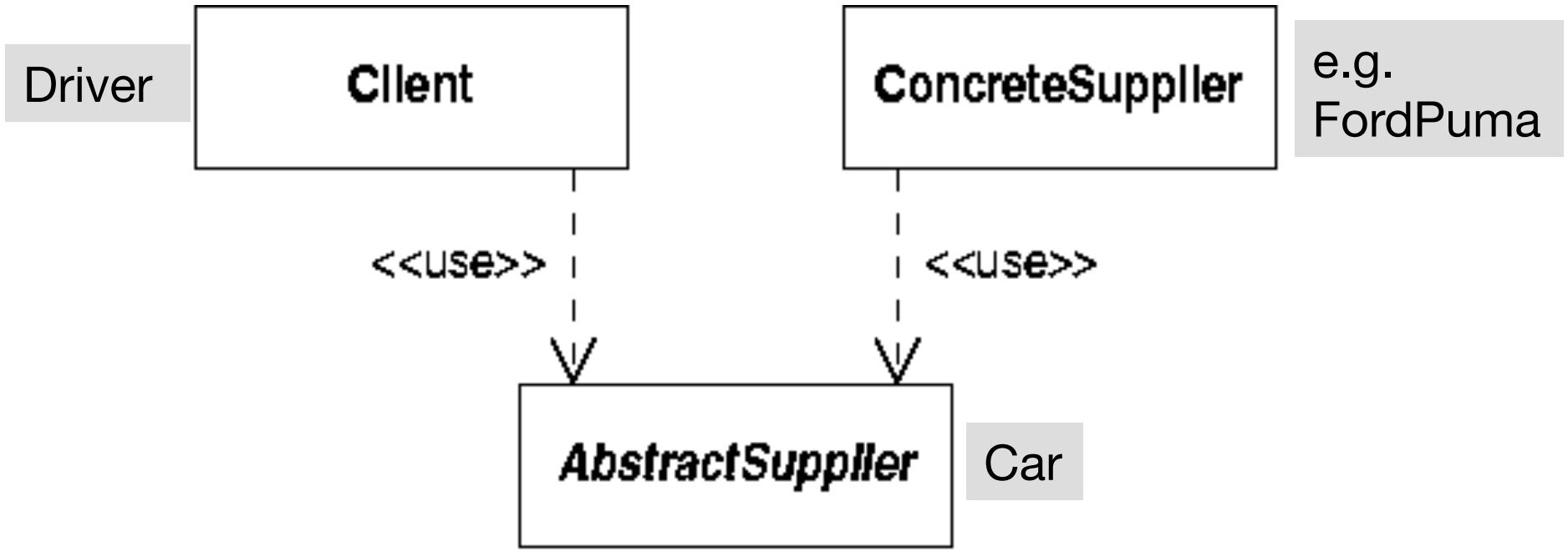
```
class FordPuma implements Car {
  private float currentSpeed;
  private boolean doorOpen;
  ...
  public void startEngine(){...}
  public void accelerate(){...}
}
```

Concrete Supplier

Note that `Driver` is now **open** to being evolved to drive a new type of car, without modification.

Looking at the dependency structure

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



This makes `Client` **open** to working with new types of `ConcreteSupplier` without modification.

What to leave open, what to close?

It is usually impossible to predict in what ways a class will be extended.

Makes classes open for extensions that never become necessary is a bad idea.

- waste of time
- violates YAGNI
- it's **overengineering** (makes code harder to work with)

Current Agile practice is to develop software in a 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.

An Evolving Shapes Example (in C#)

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

Does this code look ok so far?

Do you see the likely SRP violation?

From http://joelabrahamsson.com/a-simple-example-of-the-openclosed-principle

Area Calculator for Rectangles

```
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?

This code clearly violates the SRP.

Adding Circles

```
public class AreaCalculator {
  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;
}
```

This code is now looking really bad.

Plus we can't **extend** this class to handle new `Shape` types without **modifying** it.

Let's fix this...

Start applying the OCP

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

This is our new interface for all `Shape` classes.

Rectangle and Circle are Shapes

```
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;
  }
}
```

Our `Shape` classes now comply much better with the SRP as a by-product.

The Area Calculator Updated

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

Again, the `AreaCalculator` class now complies with the SRP as well.

And we **can** extend this class to handle new `Shape` types without **modifying** it so it now complies the the OCP.

Object-Oriented Principles: Roadmap

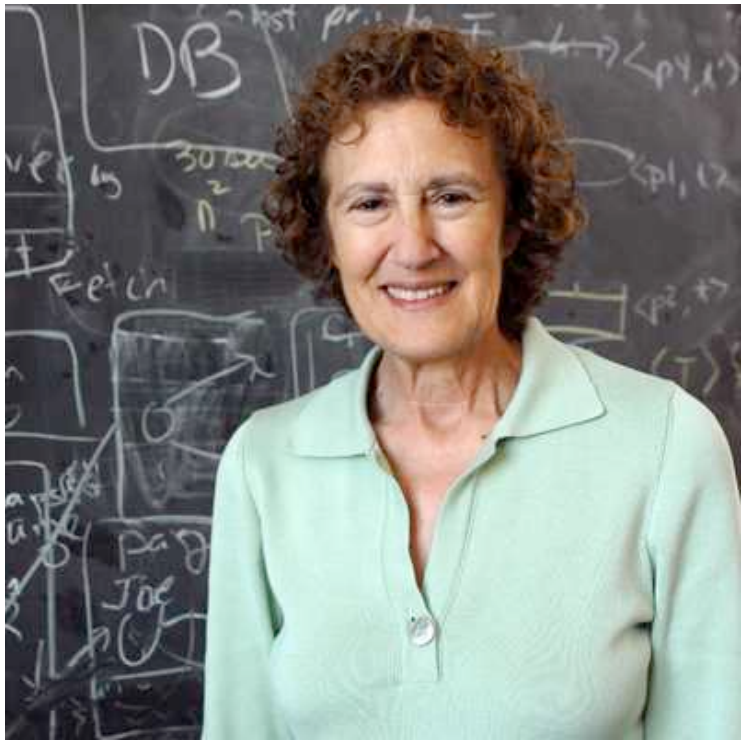
- Single Responsibility Principle (SRP)
- Interface Segregation Principle (ISP)
- Open-Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Dependency Inversion Principle (DIP)
- No Concrete Superclasses
- The Law of Demeter

Liskov Substitution Principle

The **Liskov Substitution Principle** roughly stated is this:

> "Any context that expects an instance of a Class A should work correctly when provided with an instance of a subclass of Class A."

Named after **Barbara Liskov**, ACM Turing Award winner 2008.



Liskov Substitution Principle

So e.g. a method that expects a **`Vehicle`** should work correctly when provided a **`Car.`**

The Liskov Substitution Principle is essential in understanding how inheritance and polymorphism interplay in object-oriented programming.

It also has a major impact on how we can override, in a subclass, a method defined in a superclass.

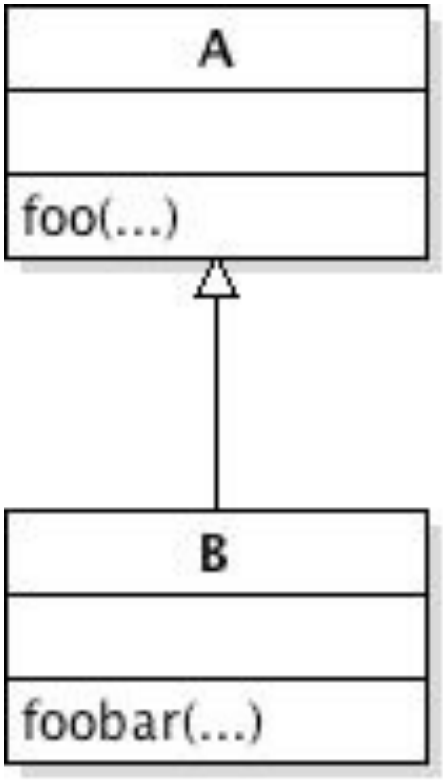
As a slogan (from the perspective of the subclass):

Demand no more, promise no less!

We now explore what the LSP permits and disallows.

Demonstrating Liskov: simple case

Say B is a subclass of A. If B doesn't remove or override any of A's methods, then LSP holds, once any new methods in B preserve any class invariants* of A.



Observing the LSP is trickier when it comes to method overriding, which is what we look at now.

*__class invariant__: a predicate on the fields of a class that should be maintained, e.g., `(1 <= month <= 12)`.

Pre- and Post-Conditions

The **precondition** of method is what the method requires to be true in order for it to run properly.

E.g. the precondition for this method:

```
float sqrt(float n);
```

would be e.g. "**`n`** is a non-negative floating-point value"

Calling `sqrt` with a `String` as argument will lead to a type error, and the result of calling it with a negative value is undefined.

A method's **postcondition** is what is guaranteed to be true once its precondition is true when the method is called (e.g. "return value is square root of **`n`**").

LSP and Method Overriding

The "demand no more, promise no less" slogan applied to method overriding can be expressed this way:

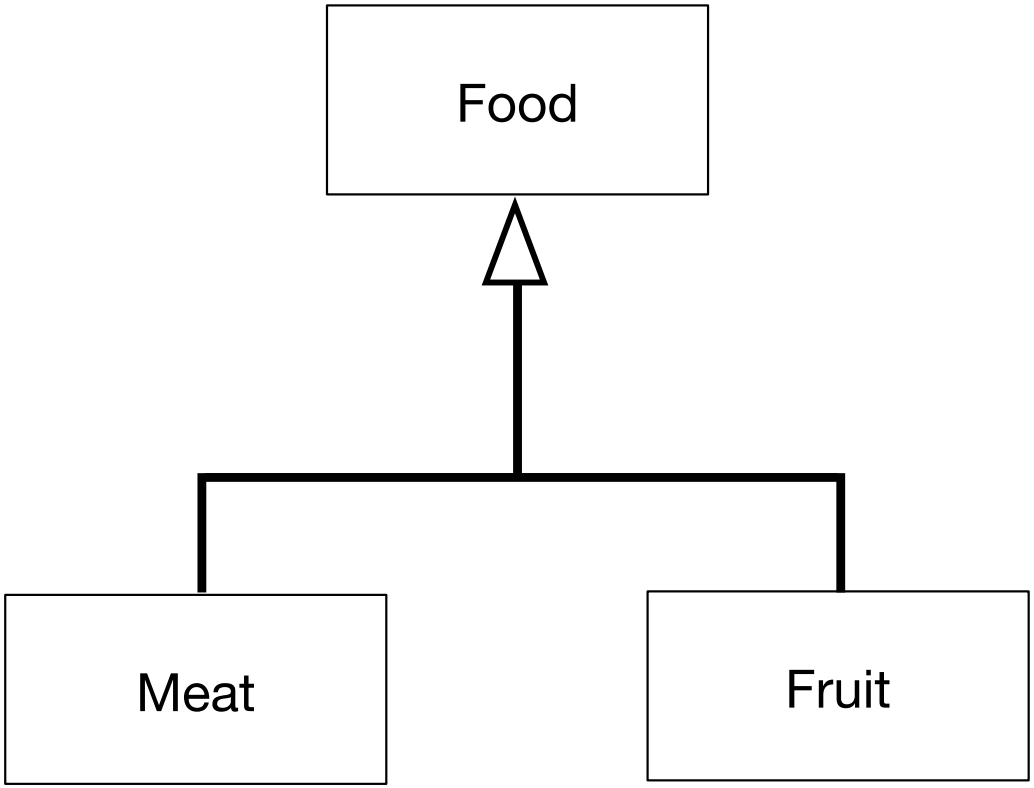
> an overriding method must have a weaker (or equal) precondition than the method it overrides, and
>
> a stronger (or equal) postcondition than the method it overrides.

Is this surprising?

Probably not, but is has some surprising consequences which we now explore.

A concrete example in Java

We assume initially that this simple inheritance hierarchy already exists:



Animals eat food

We want to model that 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 Substitution Principle!

This causes problems when a context that expects an instance of the superclass (`Animal`) is given an instance of the subclass (`Carnivore`).

Here's how we construct such an example:

```
Fruit apple = new Fruit();
Carnivore felix = new Carnivore();
Animal myAnimal = new Animal();

if (coinToss() == heads)
  myAnimal = felix;

myAnimal.eat(apple);
```

This creates a run-time type error whenever **`myAnimal`** 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.

This violates the LSP.

The code snippet above 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**, i.e. become more general in the more specific class.

(Java is even more specific — the arguments in an overriding method must be of the same type.)

The alternative, **covariance**, allows more "natural" modelling (as in the Animal example), but permits run-time type errors to occur.

(Covariance is not mainstream. The Eiffel programming language permits it, but this is no longer a mainstream language.)

Example: the `equals` method in Java

The `Object` class in Java defines an `equals` method as with this signature:

```
public boolean equals(Object o)
```

It would be natural to override this e.g. in a class `Person` as follows:

```
public boolean equals(Person o)
```

This won't work. Why?

The argument type must be *contravariant*, so you can only override this way:

```
public boolean equals(Object o)
```

Example: A Person/Child class hierarchy

Say we have a **`Person`** class as follows:

```
class Person {
  ...
  public void setAge(int newAge) {
    if (newAge >= 0 && newAge <= 125)
      age = newAge
  }
  ...
  private int age;
}
```

Now we want to create a **`Child`** class...

Overriding `setAge` in Person

A **`Child`** is a **`Person`** under the age of 16, so we implement it quickly with:

```
class Child extends Person {
  ...
  public void setAge(int newAge) {
    if (newAge >= 0 && newAge <= 16)
      super.setAge(newAge)
  }
  ...
}
```

Looks ok?

Violating Liskov

`Person::setAge` and `Child::setAge` have the same precondition. No issue there.

Let's consider the postconditions:

Method	Postcondition
`Person::setAge(newAge)`	if newAge in [0, 125] age == newAge
`Child::setAge(newAge)`	if newAge in [0, 16] age == newAge

The postcondition has been made weaker, thus violating the "promise no less" clause of the LSP.

The compiler can't help us here -- we've broken the *behavioural* aspect of the LSP.

This causes real problems (1)

Where **`person`** is of type **`Person`**, what is the effect of this line of code?

```
person.setAge(27)
```

When only the **`Person`** class existed, this line would set the age of **`person`** to 27.

However a colleague has subclassed **`Person`** to create a **`Child`** class as on the earlier slide.

Now this line may have no effect on **`person`** (if it refers to a **`Child`** object).

Very confusing!

This causes real problems (2)

Say we add this method to **`Person`**:

```
public void incrAge{
  if (age < 125)
    age += 1
}
```

What problems may this cause?

Could allow a child's age to exceed 16.

The problem isn't just about the postcondition of the **`setAge`** method, it's that the `Child` class makes the inherited constraint on `age` more specific.

Violating Liskov has made our superclass dependent on its subclasses. Whenever we change it, we'll need to revisit all its subclasses as well. Highly undesirable.

How to model this better?

One possible solution is as follows.

All **Person** objects should have their age limited to a certain range.

Store this range in the **Person** class, and use it in the range checks in **`setAge`** and **`incrAge`**.

In e.g. the **Child** class constructor, set this range to the appropriate one for a child ([0..16]).

This is a safer model, and doesn't violate Liskov.

It would also help to observe the "No Concrete Superclasses" guideline -- see later slides.

This model in code (scala)

```scala
class Person(
  var age: Int,
  ageRange: Range = Range.inclusive(0, 125)
):
  def setAge(newAge: Int) =
    if ageRange.contains(newAge) then
      age = newAge

  def incrAge =
    if age < ageRange.max then
      age += 1

class Child extends Person(
  0,
  Range.inclusive(0, 16)
)
```

Liskov in Practice

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 different.

> One the best ways of achieving this is to *invoke* the overridden method in the overriding method.
>
> This is called **method refinement**.

Liskov Summary

The Liskov Substitution Principle explains what substitution means in object-oriented programming.

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 the same type as the arguments to the overridden method.

However behavioural subtyping must be observed as well, and the compiler cannot help with this.

Object-Oriented Principles: Roadmap

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

Dependency Inversion

The Dependency Inversion principle can be simply stated as:
Abstractions should not depend on details, or
High-level classes should not depend on low-level ones.

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").

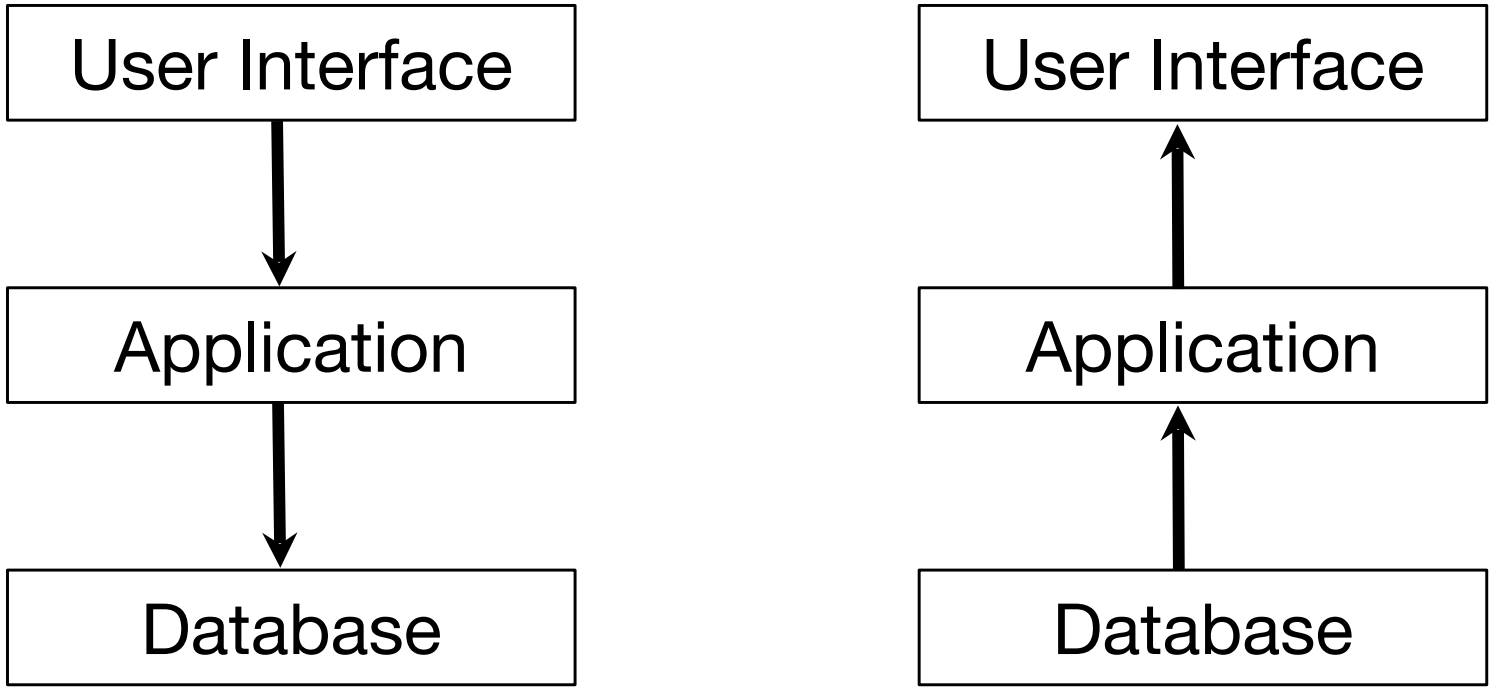
Dependency Inversion means that we invert dependencies to make them **bottom-up**.

Layered Architecture

Systems sometimes employ a **layered architecture**, e.g.
- user interface layer
- application layer
- database layer

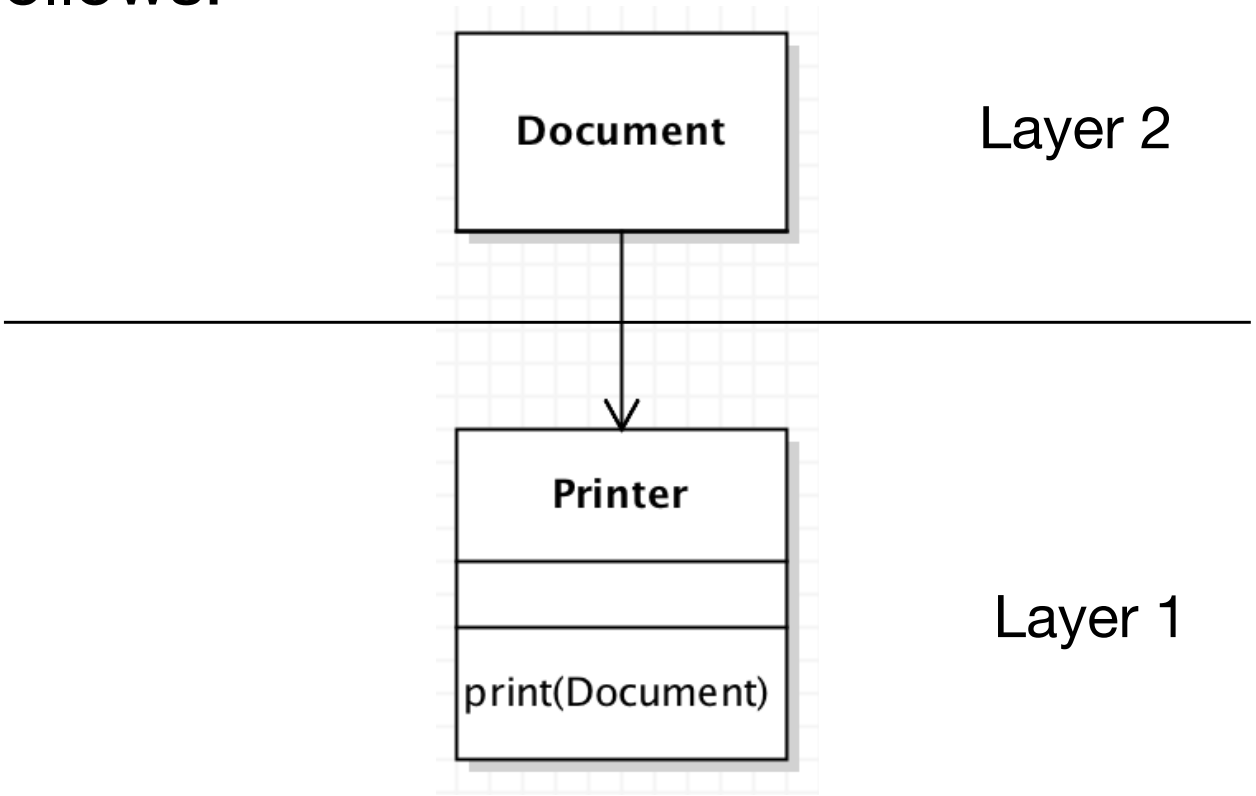
How should the dependencies go in a layered architecture?

Down or **Up**?



How to invert a dependency

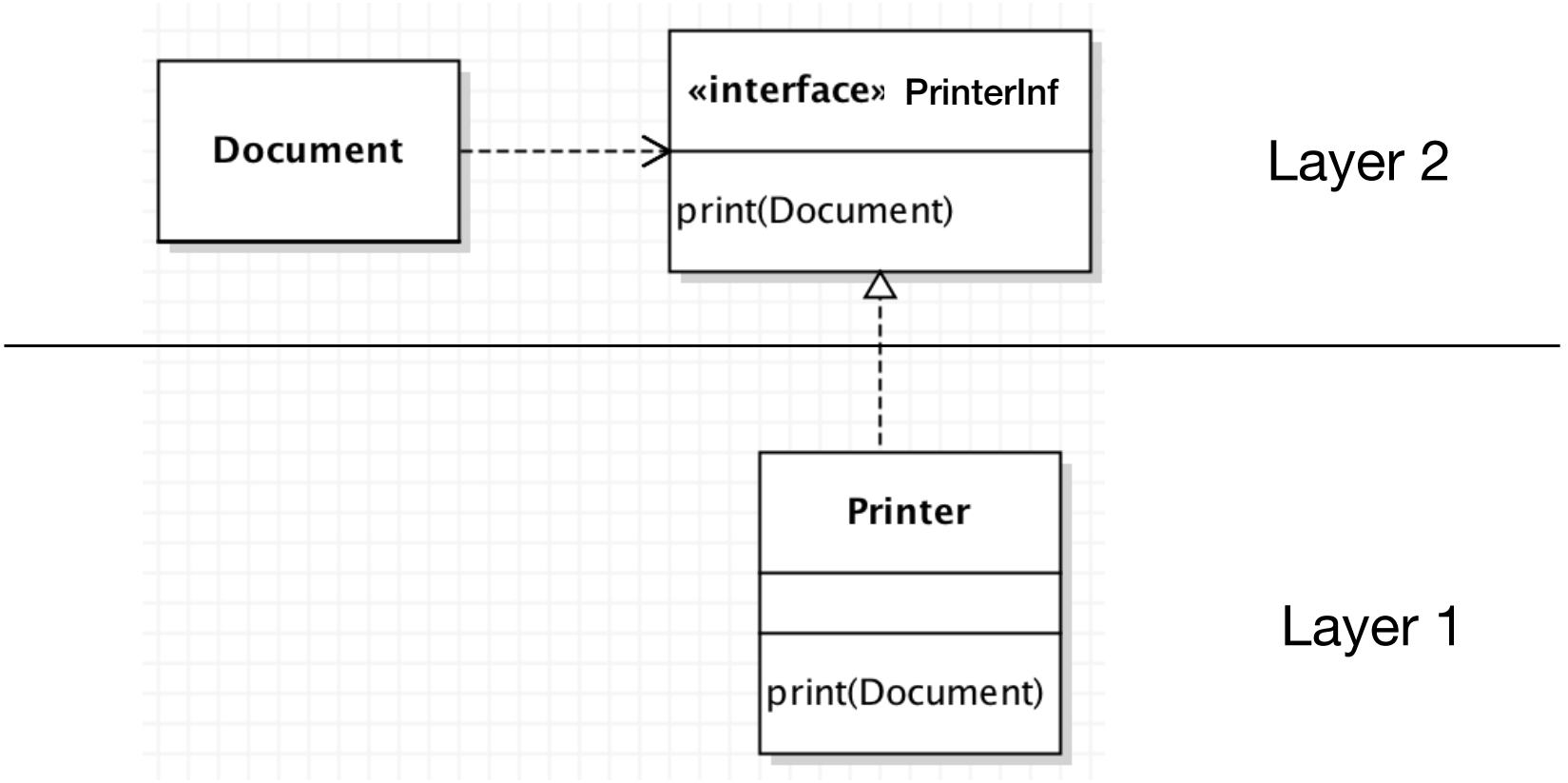
Say a system has a top-down dependency as follows:



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, but the top-down *static* dependency has disappeared.

Original design in code

```
class Document {
  public Document(Printer printer) {
    this.printer = printer;
  }

  public void foo() {
    printer.print(this);
  }

  private Printer printer;
}

class Printer {
  public void print(Document d) {
    ...
  }
}
```

Refactored to apply DIP

Layer 2

```
class Document {
  public Document(IPrinter printer) {
    this.printer = printer;
  }

  ...

  private IPrinter printer;
}

interface IPrinter {
  public void print(Document d);
}
```

Layer 1

```
class Printer implements IPrinter {
  public void print(Document d) {
    ...
  }
}
```

Comments on this example

Layer 2 has no dependency on Layer 1.

Replacing the `Printer` class in Layer 1 will not affect Layer 2.

Both `Document` and `Printer` depend on `PrinterInf`. This represents the contract between the two classes.

The entire Layer 2 can be reused in another context, once a Layer 1 with a suitable `Printer` is provided.

Reusing Layer 1 is trickier as its interface is buried in Layer 1. This can be remedied by putting `PrinterInf` into its own layer.

Related but Different: *Dependency Injection*

Dependency Injection (DI) is where classes declare their dependencies through their APIs, rather than creating them directly themselves.

Without DI, the dependencies between classes are distributed across the classes themselves.

With DI, the dependencies are centralised and easier to change.

Dependency Injection can be done in a limited way by hand, or it can be done across an entire application using a **dependency injection framework**.

Dependency Injection Example

Say some client code creates a database connection:

```
static void ClientCode() {
  IDbConnection dbConnection = new SqlConnection();
  dbConnection.Open();
  ...
}
```

This hardcodes the `ClientCode -> SqlConnection` dependency. Applying DI avoids this dependency:

```
static void ClientCode(IDbConnection dbConnection) {
   dbConnection.Open();
}
```

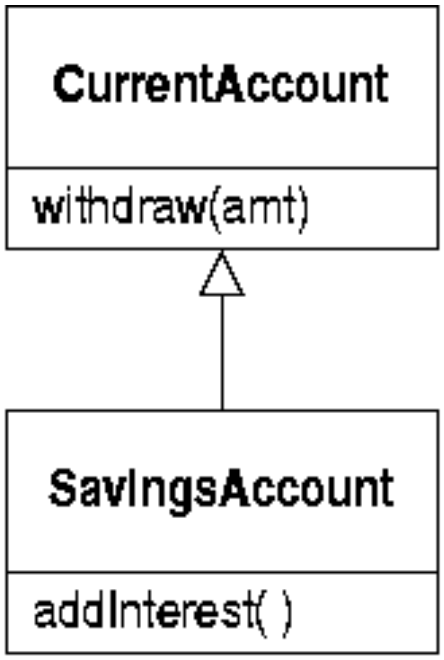
```
static void DICode() {
  IDbConnection dbConnection = new SqlConnection();
  ClientCode(dbConnection);
}
```

Object-Oriented Principles: Roadmap

- Single Responsibility Principle (SRP)
- Interface Segregation Principle (ISP)
- Open-Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- 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
- Subclasses are often added to concrete classes as a system evolves e.g.,



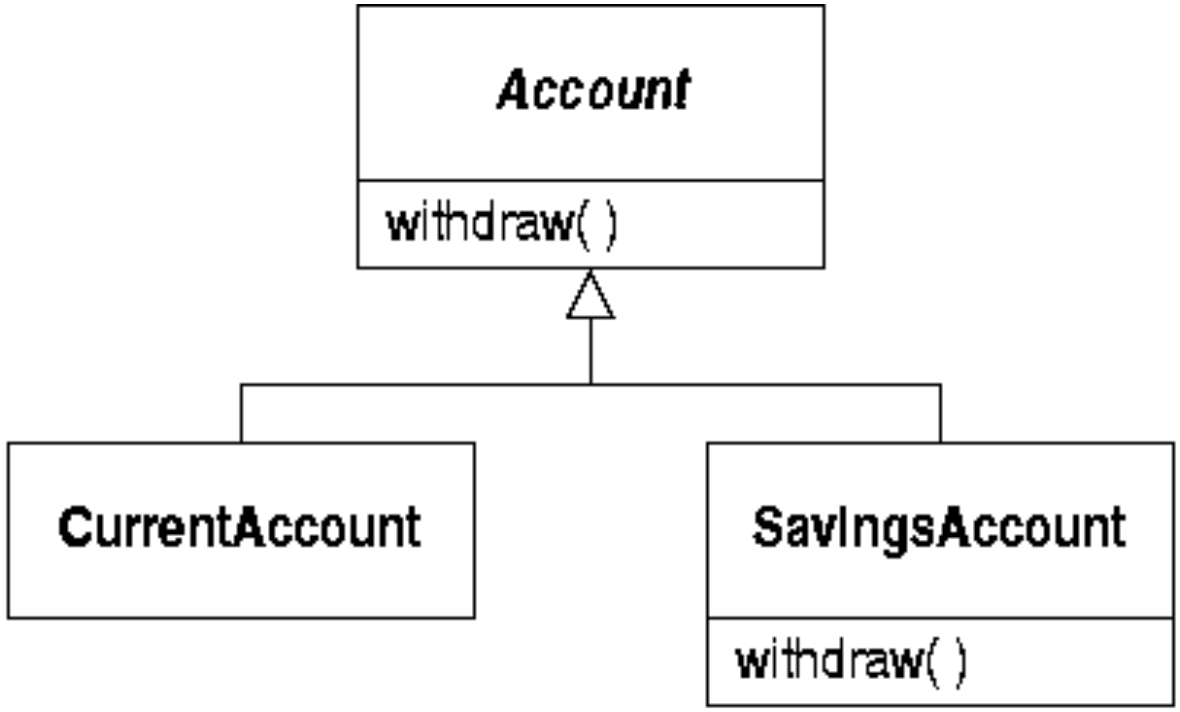
A Dependency Problem

- 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
  - it may not be desirable to apply these changes savings accounts
  - the code may become cluttered with checks for special cases

Refactoring the Design

- A better approach is to introduce an abstract class



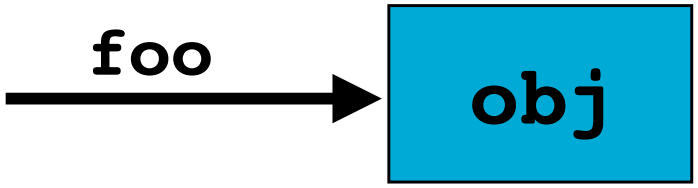
- This is a special case of Dependency Inversion
- Previously SavingAccount depended on CurrentAccount for its implementation; this has been replaced by a shared dependency on the Account abstract class.

Object-Oriented Principles: Roadmap

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

The Law of Demeter

What objects should an object **`obj`**, on receipt of a message **`foo`**, be allowed to send messages to?



So the object **`obj`**, receives the message **`foo`**, and executes some method called **`foo`**.

As the method **`foo`** executes, which objects is it able to send messages to?

The Law of Demeter

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:

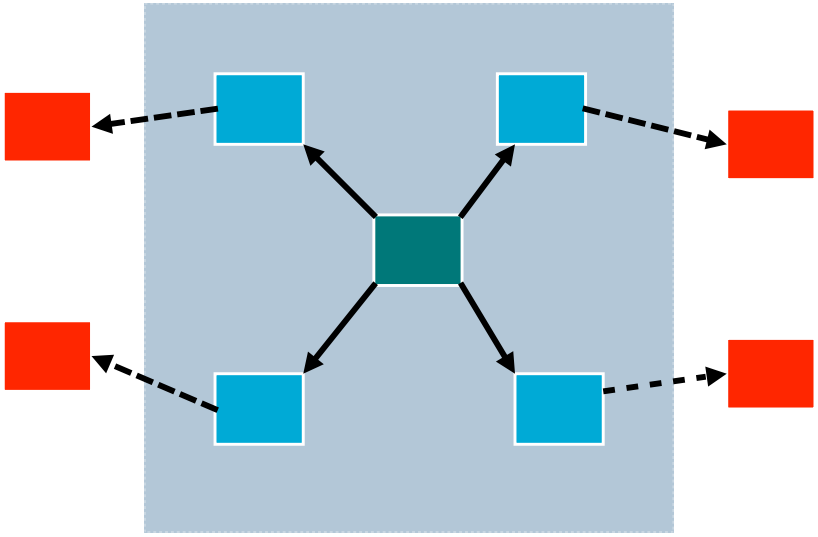
- **`obj`** itself;
- objects that are passed as arguments to **`foo`**;
- objects that are created in **`foo`**;
- objects that are attributes of **`obj`**;
- global objects accessible by **`obj`**.

So what does the Law of Demeter actually disallow?

Who shouldn’t an object talk to?

Demeter explicitly disallows sending a message to a subobject extracted from an accessible object.

This constrains how much an object can know about its environment. i.e. it **reduces dependency**.



Aka "Don't talk to strangers"

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

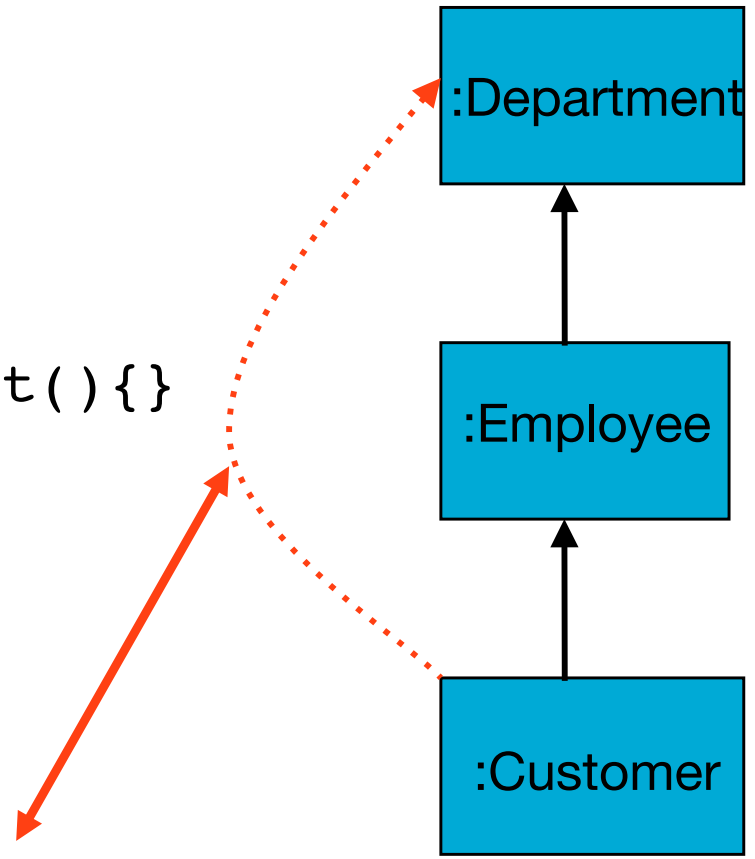
A Violation of Demeter

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

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

class Employee {
   Department getDept(){}
   Department dept;
}

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



Why is this bad? How would you solve it?

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 variables.

- This amounts to providing the same level of data abstraction to subclasses as is given to class clients.

Should I observe Demeter?

Observing Demeter reduces dependency between classes, with all the benefits that brings.

However, it also results in the addition of a lot of small methods and increases the size of public class interfaces.

As with every principle, whether you observe it in any given context is a design decision.

Learner: violates Demeter everywhere
Intermediate: observes Demeter everywhere
Master: observes/violates judiciously

Demeter Summary

- The Law of Demeter essentially avoids encoding the details of system structure in individual methods.

- It is widely regarded as a valuable factor in creating loosely-coupled software that is easier to maintain.

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 follow these principles blindly — there are often good reasons to ignore them in a particular context, but you should be aware of the trade-offs in your decision.