Object-Oriented Programming Fundamentals in C#

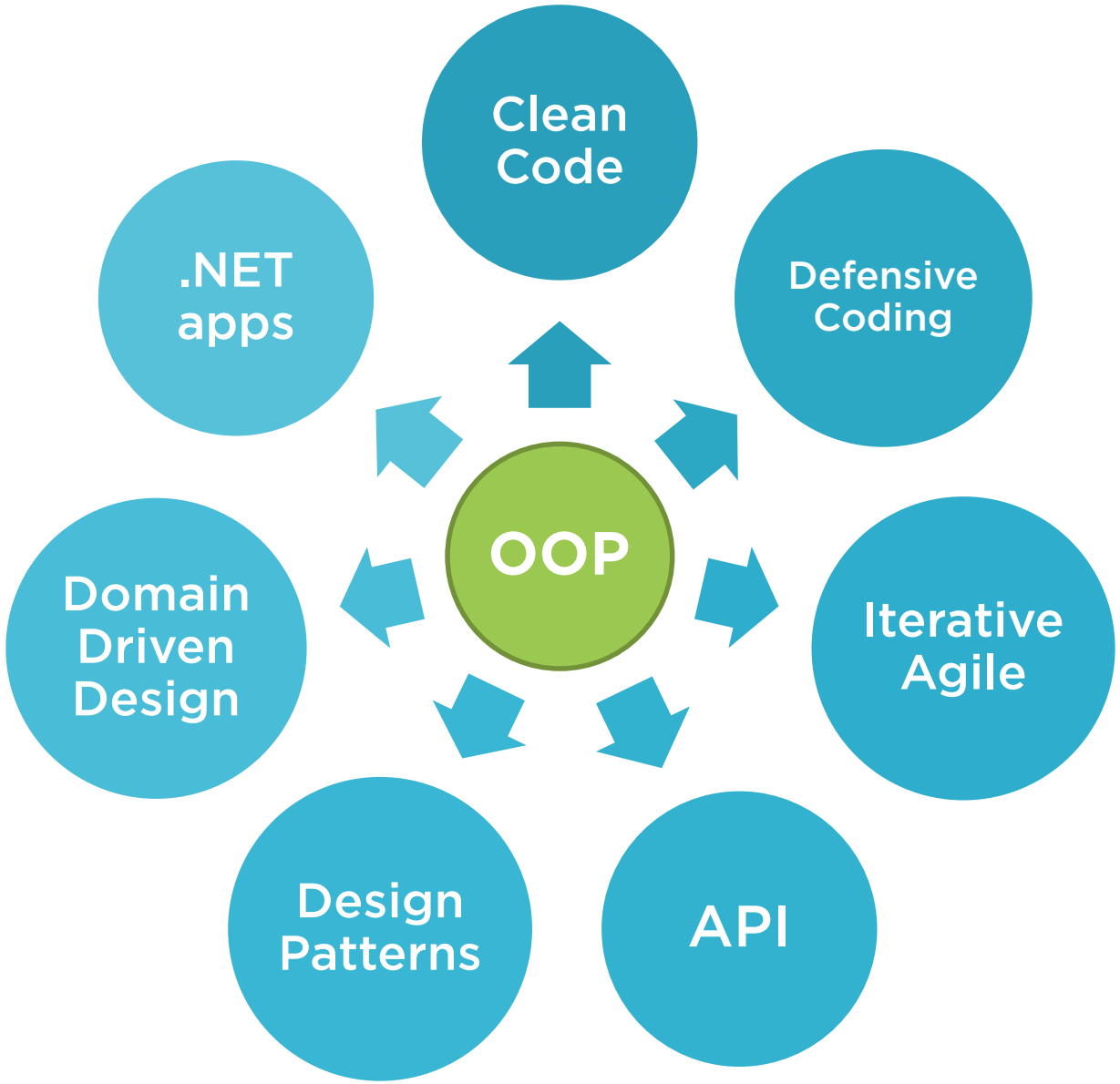
INTRODUCTION

Deborah Kurata

CONSULTANT | SPEAKER | AUTHOR | MVP | GDE

@deborahkurata | blogs.msmvps.com/deborahk/

OOP Is the Foundation



Object != Class

Properties

Methods

```
public class Customer
{
    public int CustomerId { get; set; }

    public string EmailAddress { get; set; }

    public string FirstName { get; set; }

    public string LastName { get; set; }

    public bool Validate(){ ... }
}
```

Object != Class

Members

```
public class Customer
{
    public int CustomerId { get; set; }

    public string EmailAddress { get; set; }

    public string FirstName { get; set; }

    public string LastName { get; set; }

    public bool Validate(){ ... }
}
```

Object != Class

```
var customer = new Customer();
```

Object variable

```
public class Customer
{
    public int CustomerId { get; set; }

    public string EmailAddress { get; set; }

    public string FirstName { get; set; }

    public string LastName { get; set; }

    public bool Validate(){ ... }
}
```

Objects

Class

We need to define the business objects

```
public class Customer
{
    public int CustomerId { get; set; }

    public string EmailAddress { get; set; }

    public string FirstName { get; set; }

    public string LastName { get; set; }

    public bool Validate(){ ... }
}
```

Business Object == Class


Entity

Customer Management System

Entity — customer

Class — Customer
- Last Name
- First Name
- Go On An Adventure

Objects — Bilbo Baggins

Frodo Baggins

Object-Oriented Programming (OOP)

An approach to designing and building applications that are:

- Flexible
- Natural
- Well-crafted
- Testable

by focusing on objects that interact cleanly with one another

Identifying classes

Separating responsibilities

Establishing relationships

Leveraging reuse

Prerequisites

Required

- Basic knowledge of C#
  - Syntax
  - Variables
  - Conditionals
  - Control flow

Suggested

- Visual Studio

Not Required

- OOP experience

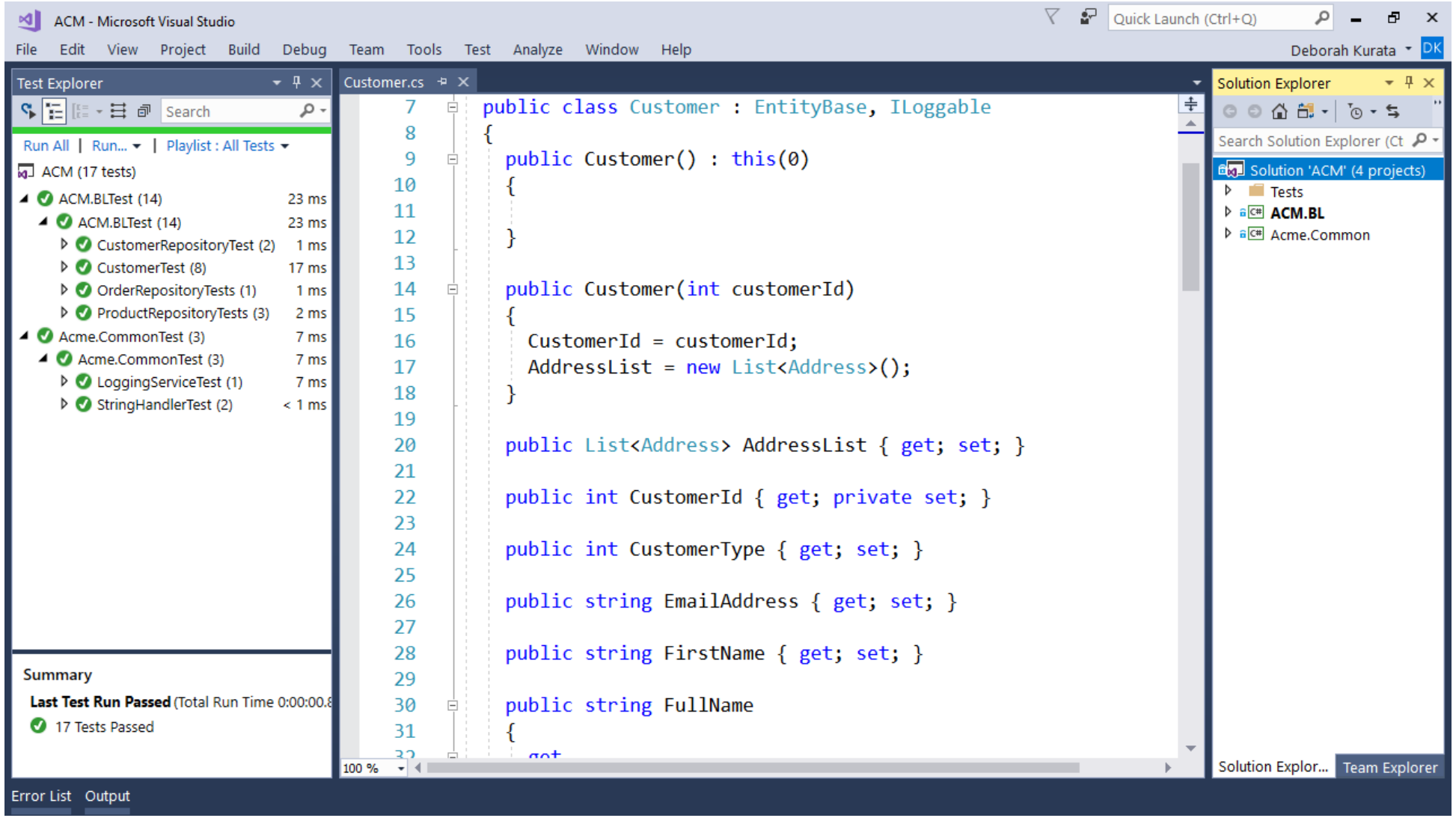
Checklist

Review module concepts

Code along assistance

Revisit as you implement your applications

Coding Along



Course Outline

Identifying classes

- Identifying classes from requirements
- Building entity classes

Separating responsibilities

- Separating responsibilities

Establishing relationships

- Establishing relationships

Leveraging reuse

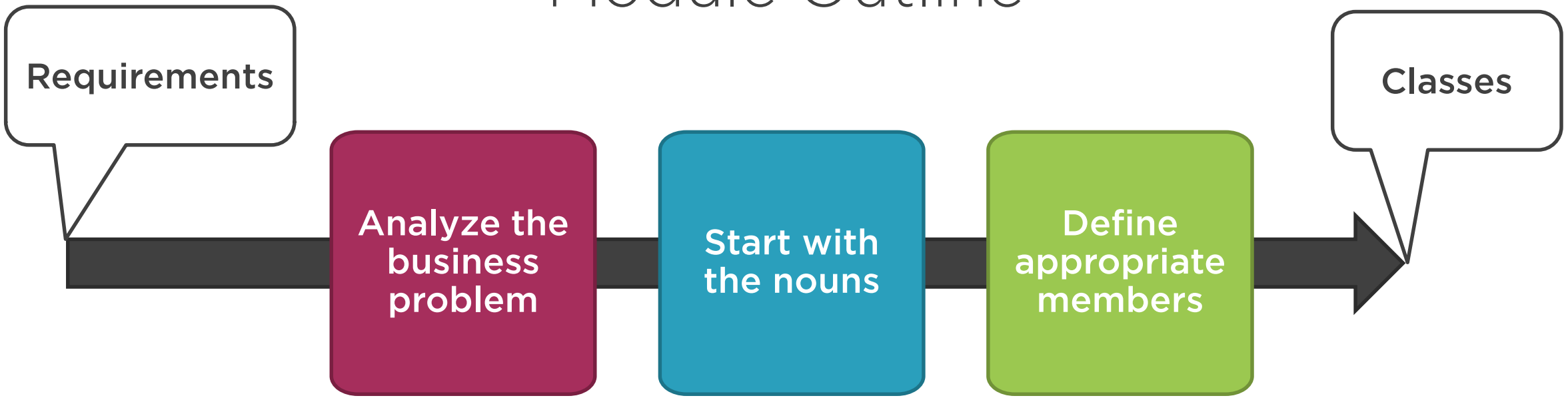
- Leveraging reuse
- Building reusable components
- Understanding interfaces

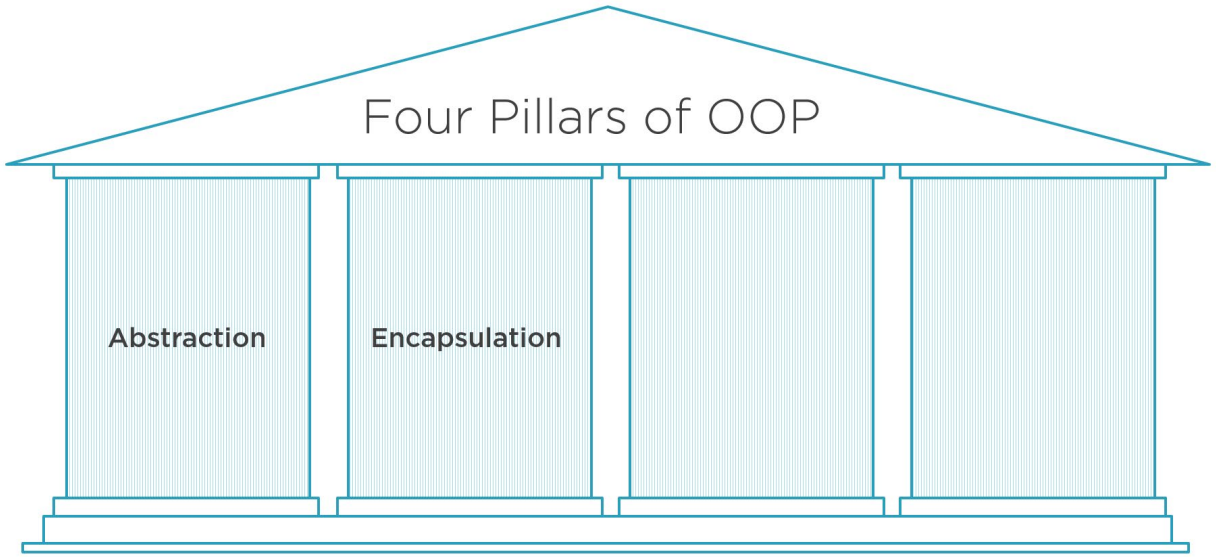
Identifying Classes from Requirements

Deborah Kurata
CONSULTANT | SPEAKER | AUTHOR | MVP | GDE

@deborahkurata | blogs.msmvps.com/deborahk/

Module Outline

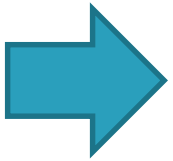




Business Requirements

Object-Oriented Programming (OOP)

Identifying classes



- Represents business entities
- Defines properties (data)
- Defines methods (actions/behavior)

Separating responsibilities

Establishing relationships

Leveraging reuse

Acme Customer Management System

Manage business, residential, government, and educator types of customers

Manage our products

Accept orders from customers online or through our call center

Start with the nouns.

Acme Customer Management System

Manage business, residential, government, and educator types of customers

Customer

Manage our products

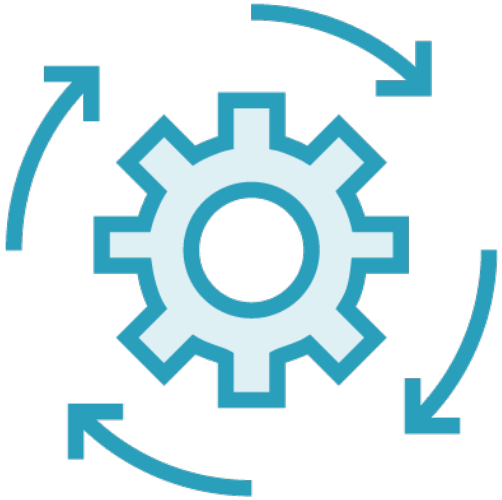
Product

Accept orders from customers online or through our call center

Order

Defining Appropriate Members

Properties: data



Methods: operations

Acme Customer Management System

- Customer's name (Last name, first name)
- Email address
- Home and work addresses

- Product name
- Description
- Current price

- Customer
- Order date
- Shipping address
- Products and quantities ordered

Define Appropriate Members

Customer	Product	Order
•Name •Email address •Home address •Work address	•Product name •Description •Current price	•Customer •Order date •Shipping address •Product •Quantity

Define Appropriate Members

Customer	Product	Order	Order Item
•Name •Email address •Home address •Work address	•Product name •Description •Current price	•Customer •Order date •Shipping address •Order items	•Product •Quantity

Define Appropriate Members

Customer	Product	Order	Order Item
•Name •Email address •Home address •Work address •Validate() •Retrieve() •Save()	•Product name •Description •Current price •Validate() •Retrieve() •Save()	•Customer •Order date •Shipping address •Order items •Validate() •Retrieve() •Save()	•Product •Quantity •Validate() •Retrieve() •Save()

Consider the effect of data changes over time

Define Appropriate Members

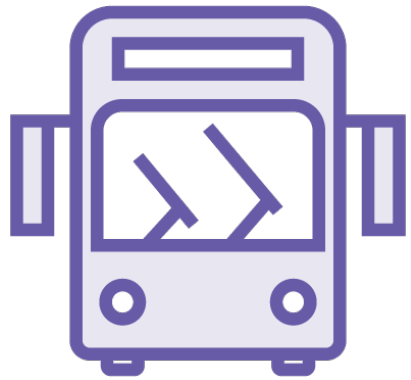
Customer	Product	Order	Order Item
•Name •Email address •Home address •Work address •Validate() •Retrieve() •Save()	•Product name •Description •Current price •Validate() •Retrieve() •Save()	•Customer •Order date •Shipping address •Order items •Validate() •Retrieve() •Save()	•Product •Quantity •Validate() •Retrieve() •Save()

Define Appropriate Members

Customer	Product	Order	Order Item
•Name •Email address •Home address •Work address •Validate() •Retrieve() •Save()	•Product name •Description •Current price •Validate() •Retrieve() •Save()	•Customer •Order date •Shipping address •Order items •Validate() •Retrieve() •Save()	•Product •Quantity •Purchase price •Validate() •Retrieve() •Save()

Abstraction

Manage business, residential, government, and educator types of customers

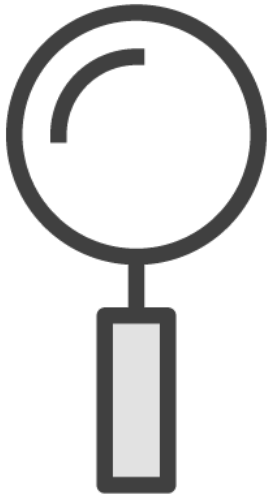


Joe Smith
Joe@aol.com
123 Main St.

Abstraction

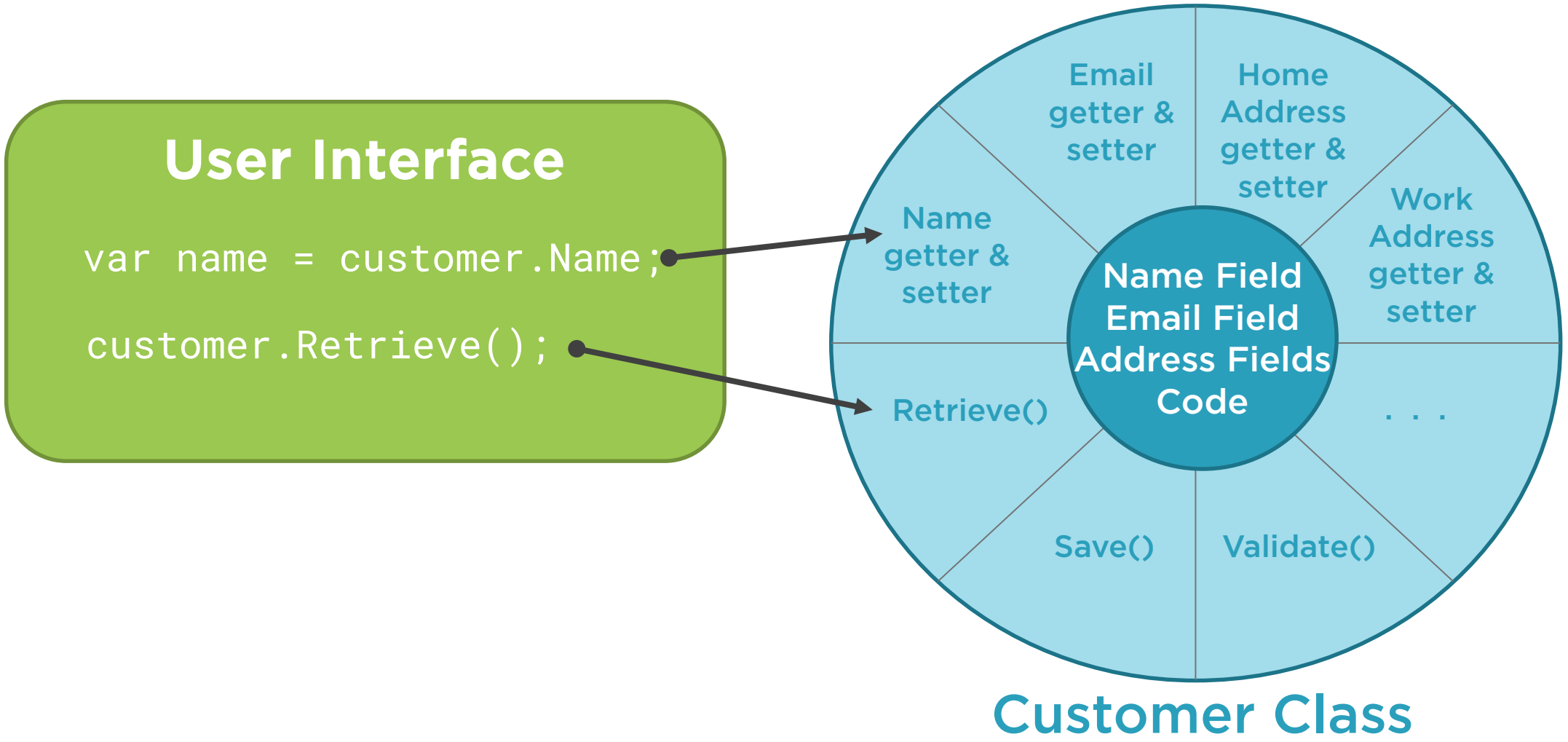
Simplifying reality

Ignoring extraneous details

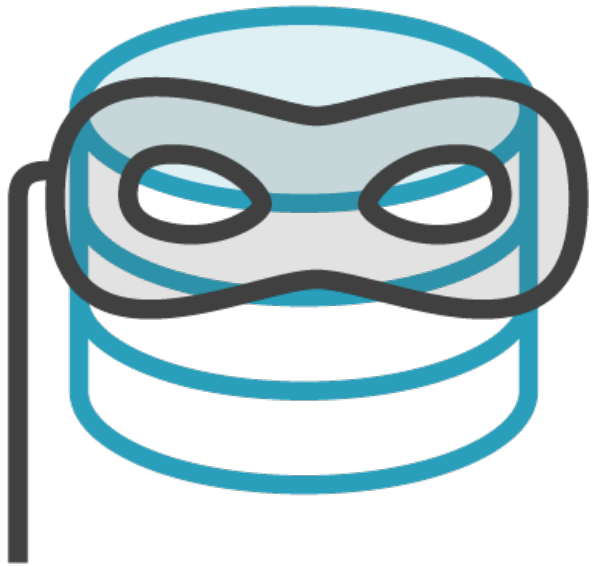


Focusing on what is important for a purpose

Encapsulation



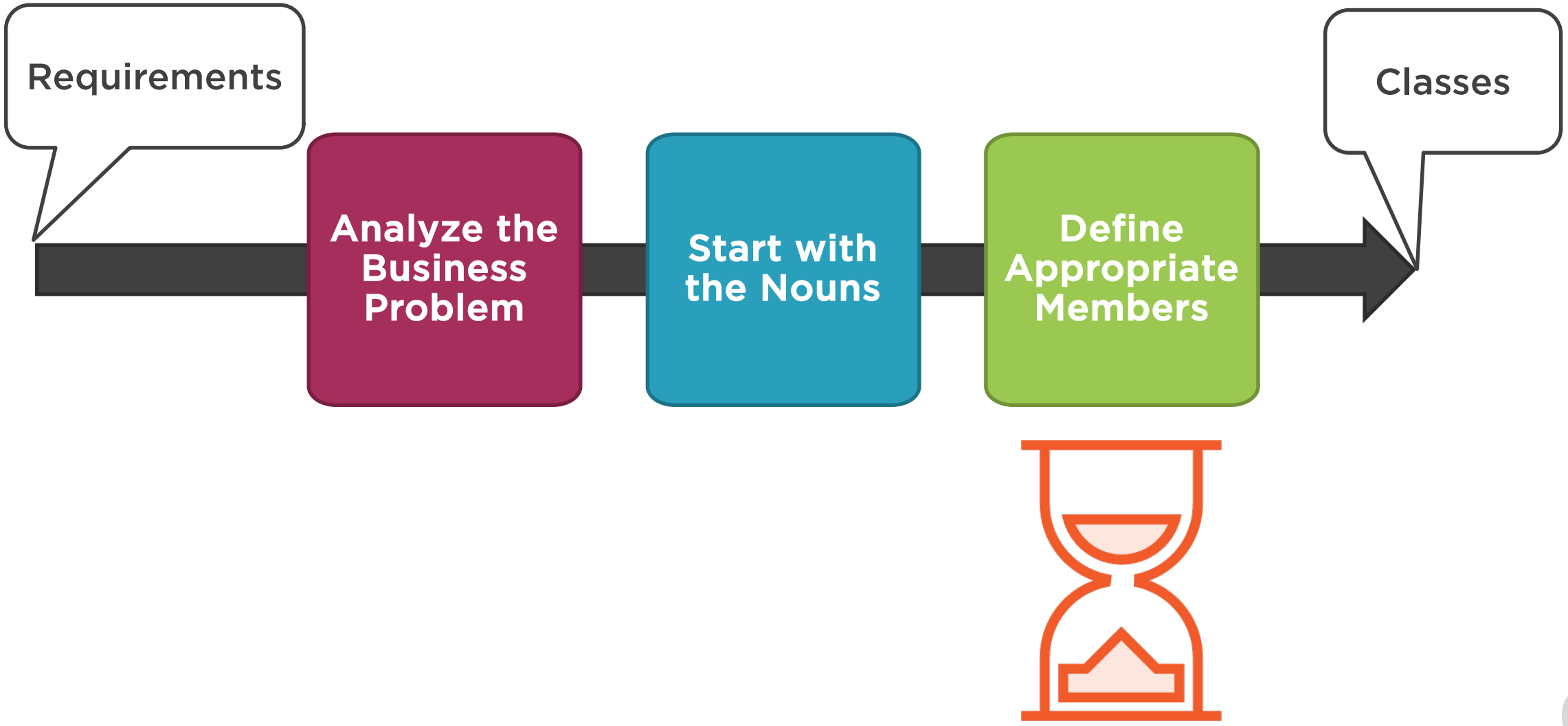
Encapsulation



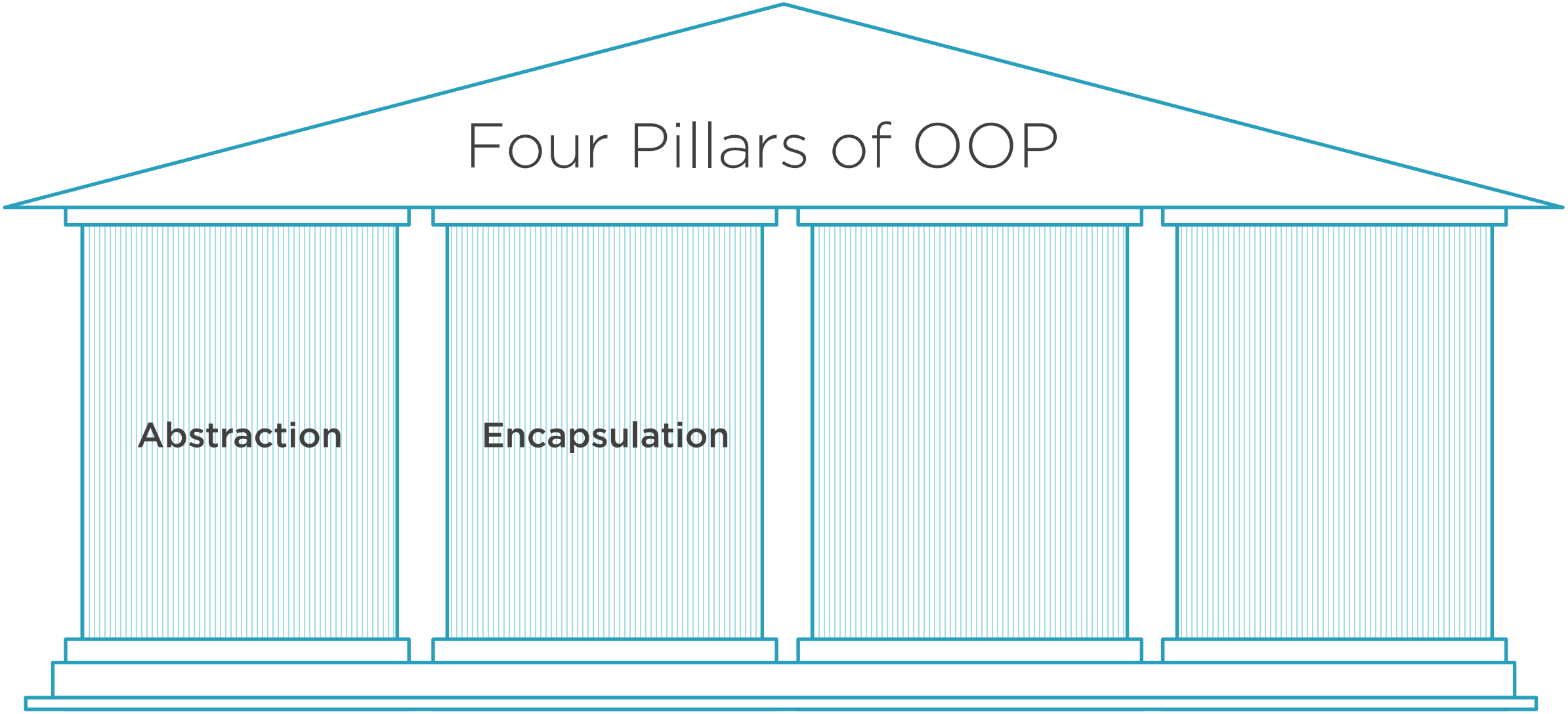
- Protects the data
- Allows for authorization before getting the data
- Allows for validation before setting the data

- Helps manage complexity
- Only the class needs to understand the implementation
- Implementation can be changed without impacting the application

Identifying Classes

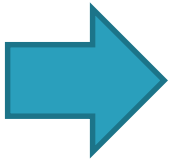


Four Pillars of OOP



Object-Oriented Programming (OOP)

Identifying classes



- Represents business entities
- Defines properties (data)
- Defines methods (actions/behavior)

Separating responsibilities

Establishing relationships

Leveraging reuse

Building Entity Classes

Deborah Kurata
CONSULTANT | SPEAKER | AUTHOR | MVP | GDE

@deborahkurata | blogs.msmvps.com/deborahk/

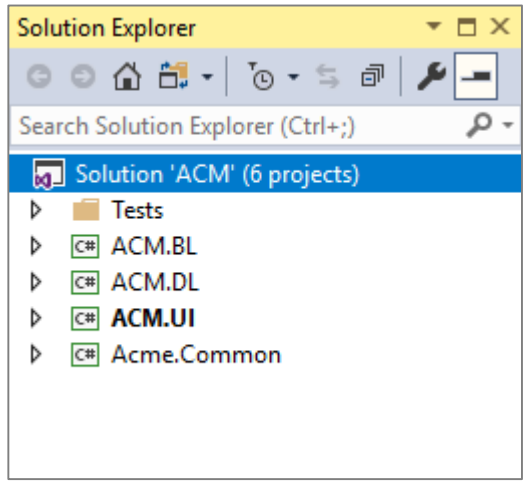
Identified Classes

Customer	Product	Order	Order Item
•Name •Email address •Home address •Work address •Validate() •Retrieve() •Save()	•Product name •Description •Current price •Validate() •Retrieve() •Save()	•Customer •Order date •Shipping address •Order items •Validate() •Retrieve() •Save()	•Product •Quantity •Purchase price •Validate() •Retrieve() •Save()

Layering the Application

Build with a layered structure

Layering is key to a good application structure



Solution -> application

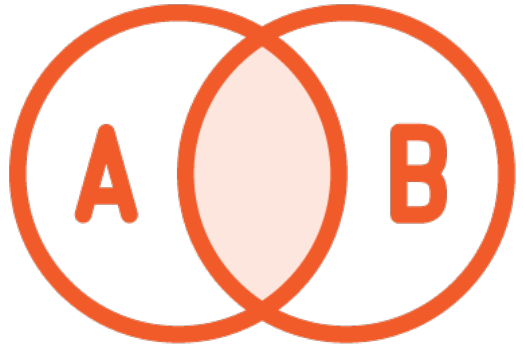
Project -> layer

Common Application Layers

User interface layer

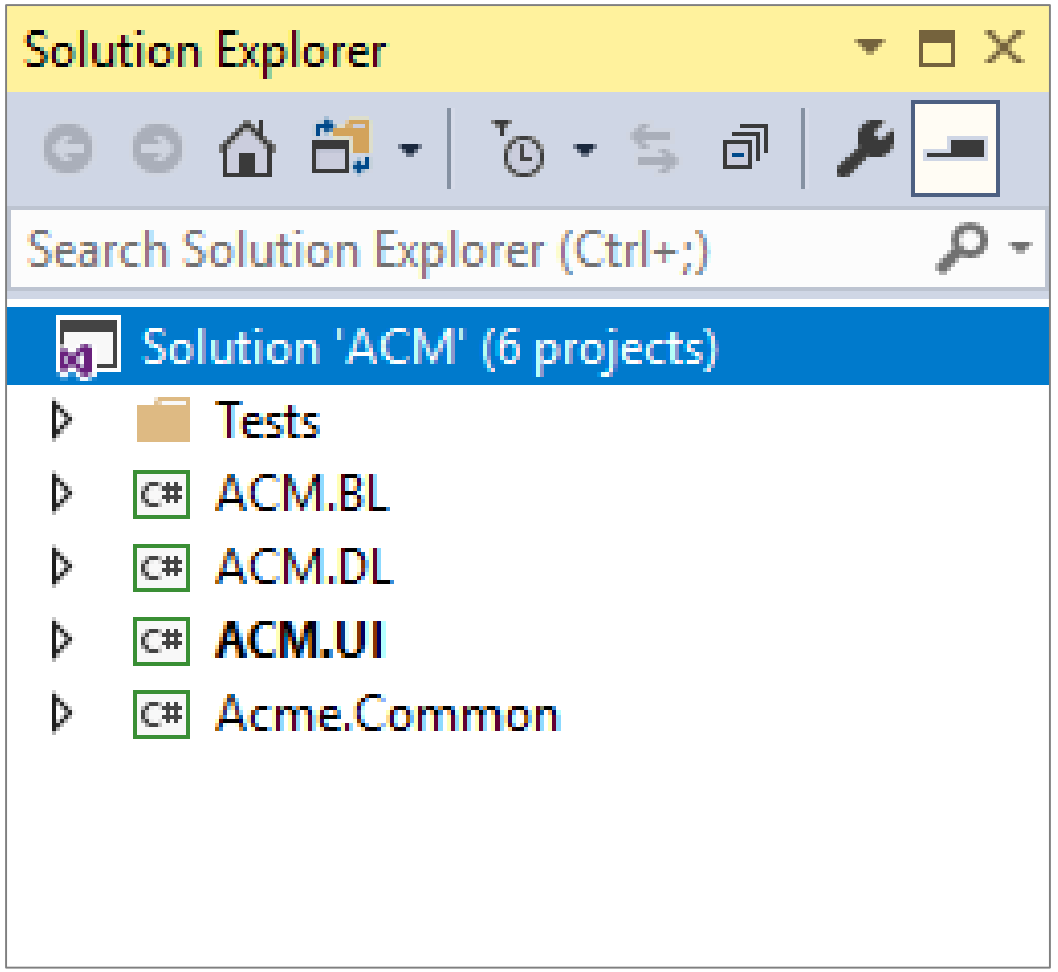
Business logic layer

Data access layer

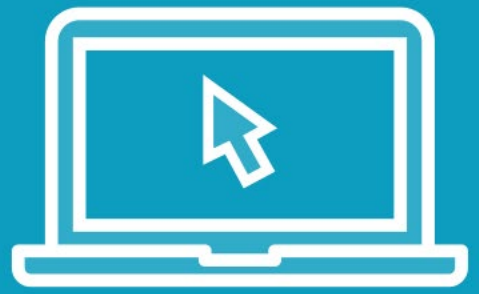


Common code

Visual Studio Solution

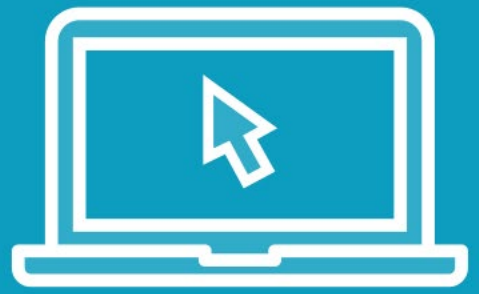


Demo



Building the business logic component

Demo



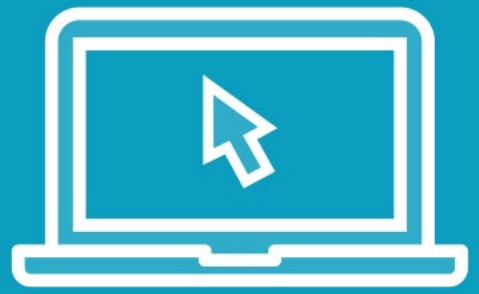
Building a class

Adding properties

Customer

- Name
- Email address
- Home address
- Work address
- Validate()
- Retrieve()
- Save()

Demo



Using snippets

Testing Our Code

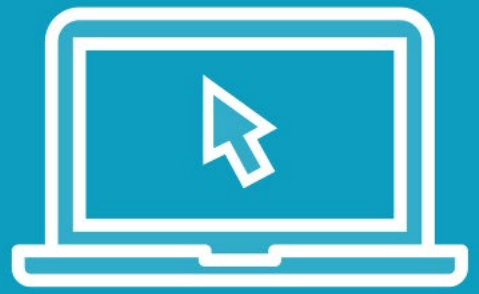
```
public class Customer
{
    public int CustomerId { get; private set; }

    public string EmailAddress { get; set; }

    public string FirstName { get; set; }

    public string FullName
    {
        get
        {
            return LastName + "," + FirstName;
        }
    }

    private string _lastName;
    public string LastName
    {
        get
        {
            return _lastName;
        }
        set
        {
            _lastName = value;
        }
    }

}
```

Our class

```
[TestClass]
public class CustomerTest
{

    [TestMethod]
    public void FullNameTestValid()
    {
        //-- Arrange
        Customer customer = new Customer();
        customer.FirstName = "Bilbo";
        customer.LastName = "Baggins";

        string expected = "Baggins, Bilbo";

        //-- Act
        string actual = customer.FullName;

        //-- Assert
        Assert.AreEqual(expected, actual);
    }
```

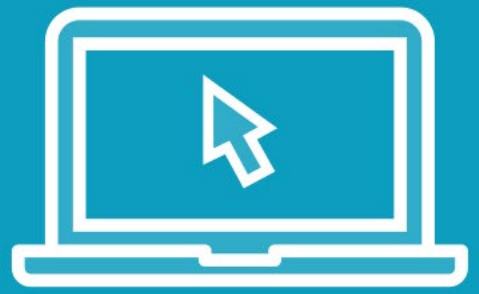
Test for our class

Demo



Testing our class: valid values

Demo



Testing our class: invalid values

Creating a New Object

```
Customer customer = new Customer();
```

```
var customer = new Customer();
```

Accessing Properties

```
var customer = new Customer();
```

```
customer.LastName = "Baggins";
customer.FirstName = "Bilbo";
```

```
var actual = customer.FullName;
```

```
public class Customer
{
    public int CustomerId { get; private set; }

    public string EmailAddress { get; set; }

    public string FirstName { get; set; }

    public string FullName
    {
        get
        {
            string fullName = LastName;
            if (!string.IsNullOrWhiteSpace(FirstName))
            {
                if (!string.IsNullOrWhiteSpace(fullName))
                {
                    fullName += ", ";
                }
                fullName += FirstName;
            }
            return fullName;
        }
    }

    private string _lastName;
    public string LastName
    {
        get
        {
            return _lastName;
        }
        set
        {
            _lastName = value;
        }
    }

}
```

Objects Are Reference Types

```
int i1;
i1 = 42;

int i2 = i1;
i2 = 2;
```

What is i1?

```
var c1 = new Customer();
c1.FirstName = "Bilbo";

var c2 = c1;
c2.FirstName = "Frodo";
```

What is c1.FirstName?

i1

i2

c1

c2

Static Modifier

```
public static int InstanceCount { get; set; }
```

```
Customer.InstanceCount += 1;
```

Layering the Application

- User interface
- Business logic
- Data access
- Common library

Building a Class

Each class defines a type

Give the class a good name

Set the appropriate access modifier

```
public class Customer
{

}
```

Defining Properties: Manually

Declare the backing field

Declare the property

Add the getter and setter

```
private string _lastName;
public string LastName
{
    get
    {
        return _lastName;
    }
    set
    {
        _lastName = value;
    }
}
```

Defining Properties: Auto-implemented

Manages the backing field automatically

Use when the setter and getter don't need logic

```
public string FirstName { get; set; }
```

Use Visual Studio snippets

Unit Testing

Create a separate project

Set a reference to the business layer component

Define tests for valid and invalid scenarios

Organize the test

- Arrange: Set up the test
- Act: Access the member being tested
- Assert: Determine the result

Working with Objects

```
Customer customer = new Customer();
```

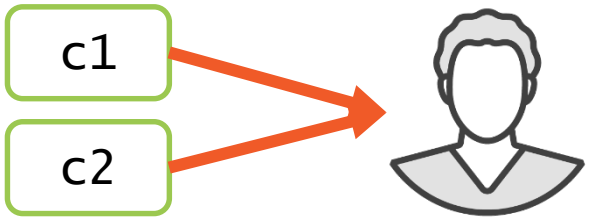
Creating an object

```
var customer = new Customer();
```

var keyword

```
customer.FirstName = "Bilbo";
```

Setting properties



Objects are reference types

```
public static int InstanceCount { get; set; }
Customer.InstanceCount += 1;
```

Static modifier

Customer Class

Building Entity Classes - Methods

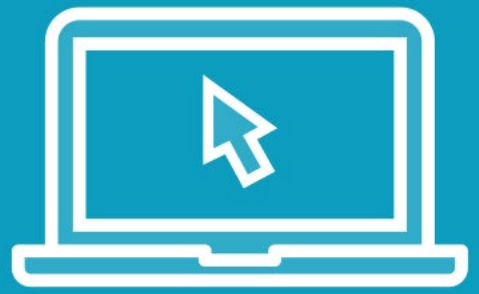
Deborah Kurata
CONSULTANT | SPEAKER | AUTHOR | MVP | GDE

@deborahkurata | blogs.msmvps.com/deborahk/

Identified Classes

Customer	Product	Order	Order Item
•Name •Email address •Home address •Work address •Validate() •Retrieve() •Save()	•Product name •Description •Current price •Validate() •Retrieve() •Save()	•Customer •Order date •Shipping address •Order items •Validate() •Retrieve() •Save()	•Product •Quantity •Purchase price •Validate() •Retrieve() •Save()

Demo

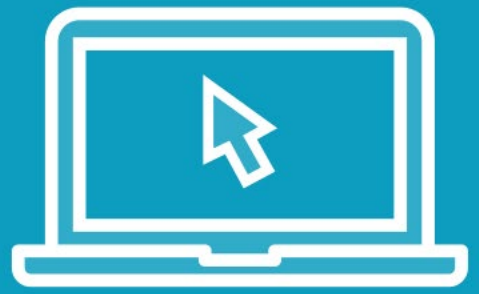


Creating class methods

Customer

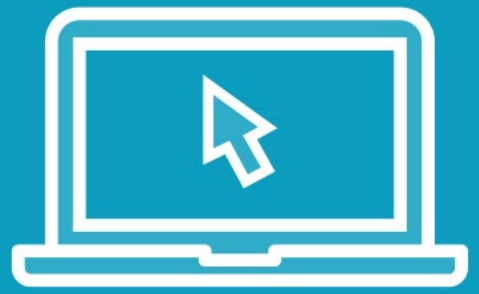
- Name
- Email address
- Home address
- Work address
- Validate()
- Retrieve()
- Save()

Demo



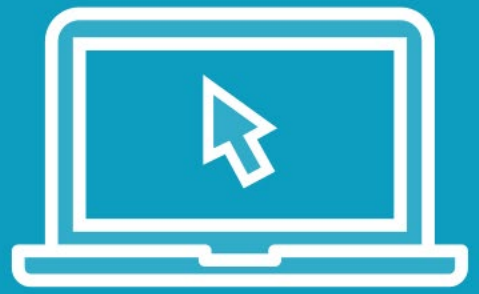
Testing methods

Demo



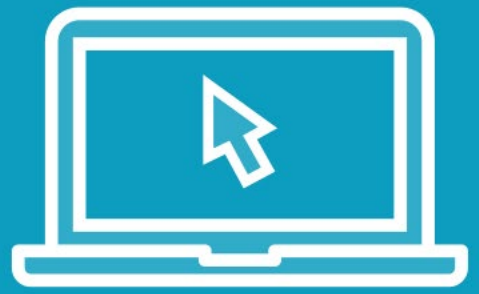
Method terms

Demo



Constructors

Demo



Building the remaining classes

Product	Order	Order Item
•Product name •Description •Current price •Validate() •Retrieve() •Save()	•Customer •Order date •Shipping address •Order items •Validate() •Retrieve() •Save()	•Product •Quantity •Purchase price •Validate() •Retrieve() •Save()

Classes

Customer	Product	Order	Order Item
•Name •Email address •Home address •Work address •Validate() •Retrieve() •Save()	•Product name •Description •Current price •Validate() •Retrieve() •Save()	•Customer •Order date •Shipping address •Order items •Validate() •Retrieve() •Save()	•Product •Quantity •Purchase price •Validate() •Retrieve() •Save()

Creating Methods

Set the appropriate access modifier

Specify the desired return type or void

Give the method a good name

```
public bool Validate()
{
  var isValid = true;

  if (string.IsNullOrWhiteSpace(LastName)) isValid = false;
  if (string.IsNullOrWhiteSpace(EmailAddress)) isValid = false;

  return isValid;
 }
```

Unit Testing Methods

Define tests for valid and invalid scenarios

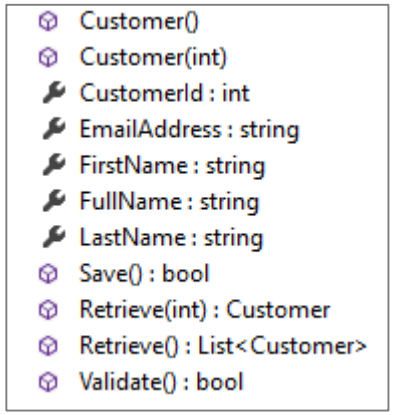
Organize the test

- Arrange: Set up the test
- Act: Call the method being tested
- Assert: Determine the result

Method Terminology

```
public Customer Retrieve(int customerId)
```

Signature



Contract

```
public Customer Retrieve(int customerId)

public List<Customer> Retrieve()
```

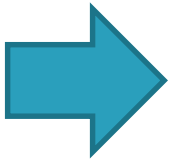
Overloading

```
public Customer() { }

public Customer(int customerId) { }
```

Constructor

Object-Oriented Programming (OOP)

Identifying classes



- Represents business entities
- Defines properties (data)
- Defines methods (actions/behavior)

Separating responsibilities

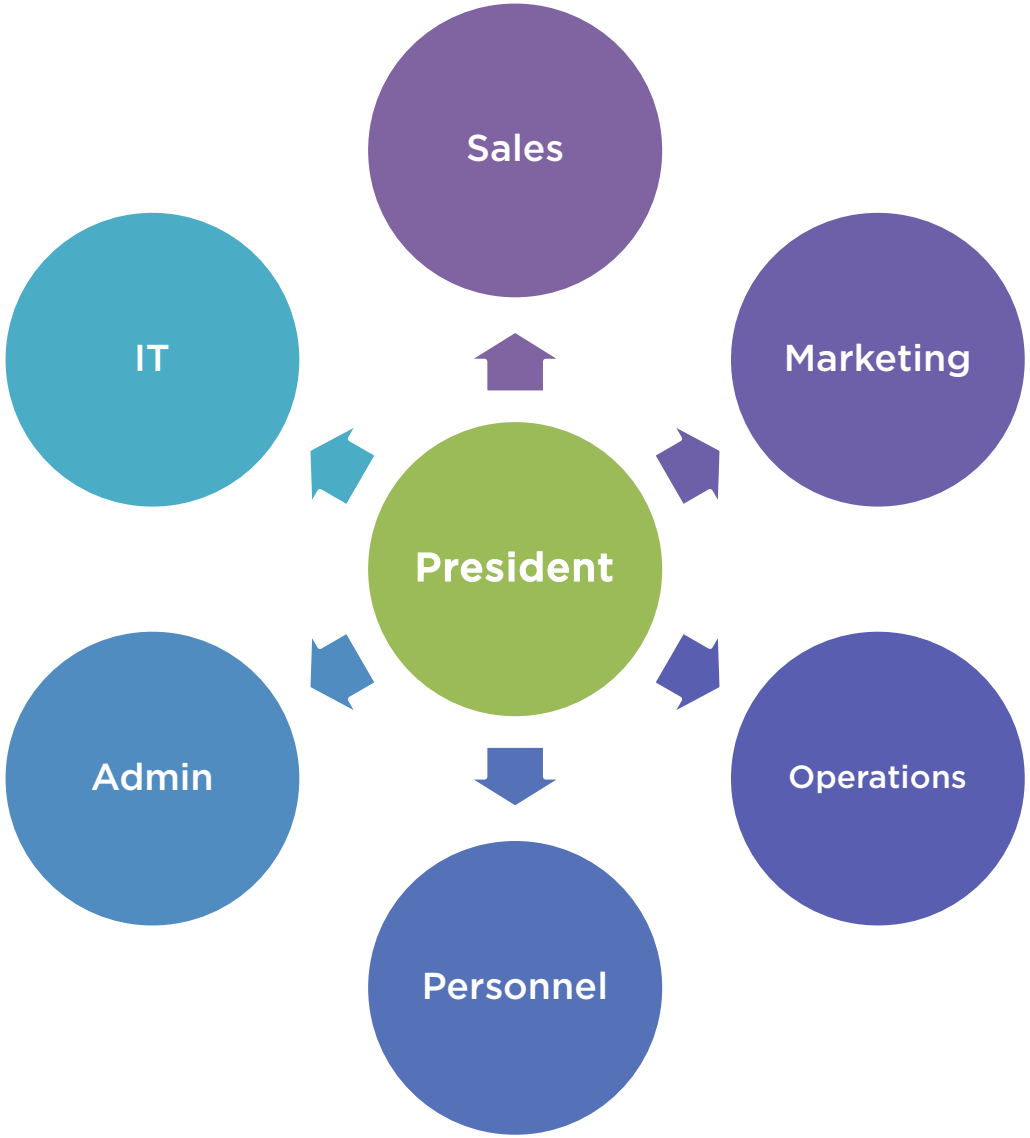
Establishing relationships

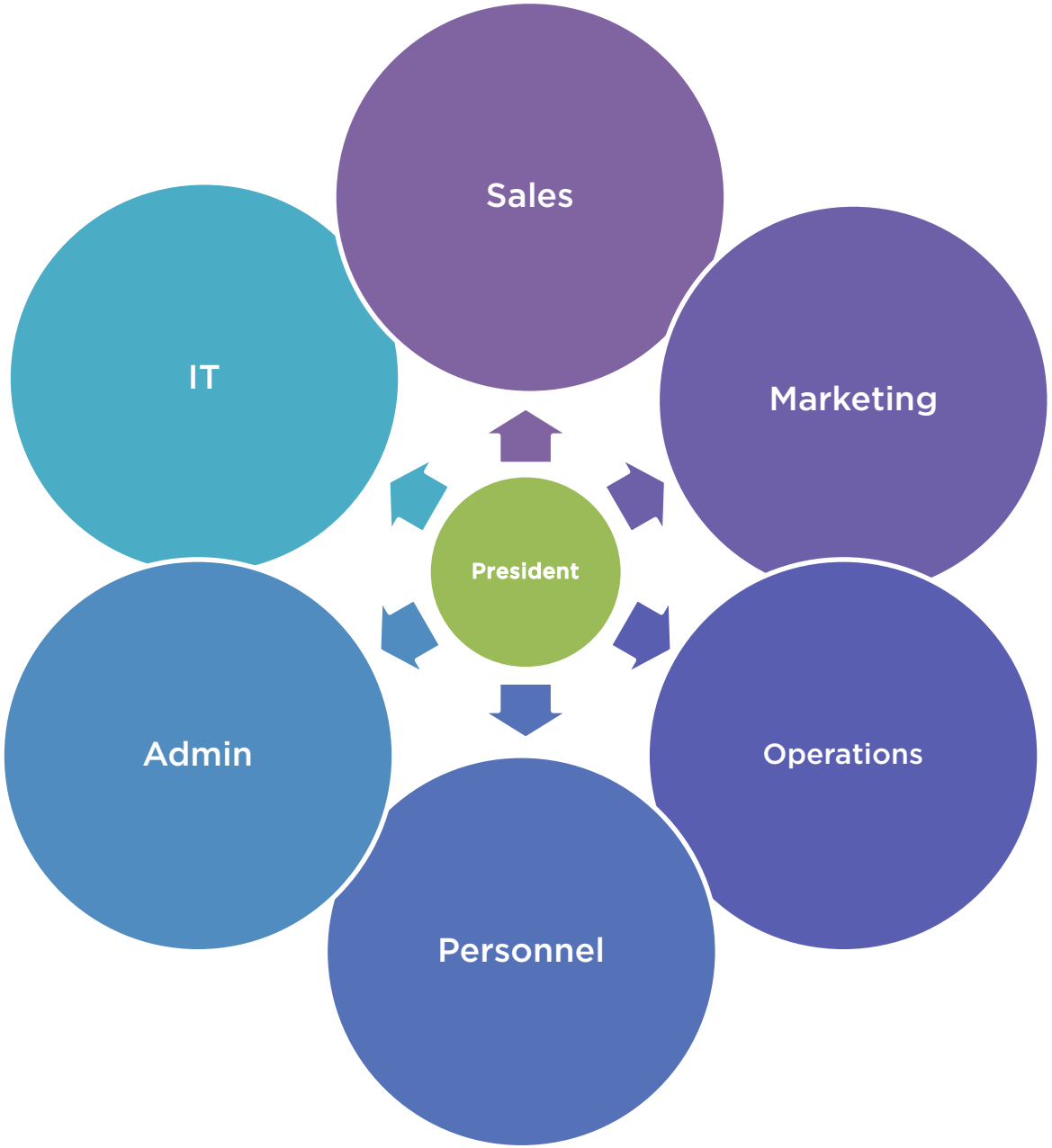
Leveraging reuse

Separation of Responsibilities

Deborah Kurata
CONSULTANT | SPEAKER | AUTHOR | MVP | GDE

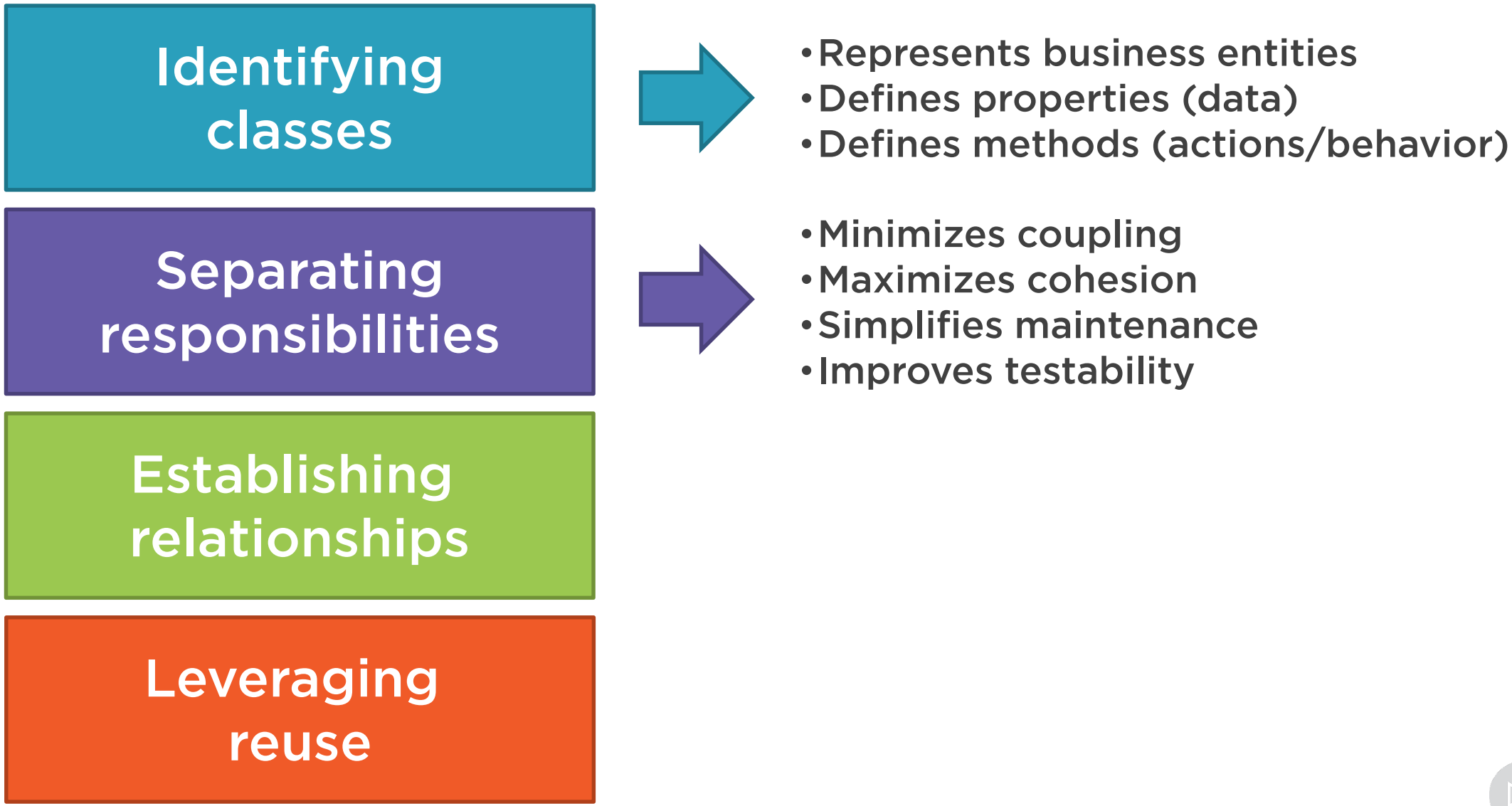
@deborahkurata | blogs.msmvps.com/deborahk/


Sales
IT
Marketing
President
Admin
Operations
Personnel


Sales
Marketing
Operations
Personnel
Admin
IT
President


Name
Email Address
Postal Address
Validate
Retrieve
Save
Customer

Object-Oriented Programming (OOP)



Minimizing Coupling

Customer	Product	Order	Order Item
•Name •Email address •Home address •Work address •Validate() •Retrieve() •Save()	•Product name •Description •Current price •Validate() •Retrieve() •Save()	•Customer •Order date •Shipping address •Order items •Validate() •Retrieve() •Save()	•Product •Quantity •Purchase price •Validate() •Retrieve() •Save()

Data Access Layer

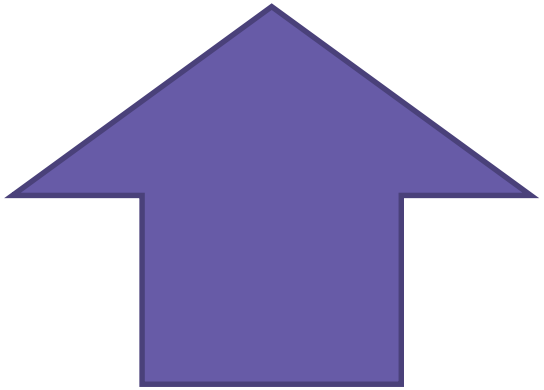
Maximizing Cohesion

Customer	Product	Order	Order Item
•Name •Email address •Home address •Work address •Validate() •Retrieve() •Save()	•Product name •Description •Current price •Validate() •Retrieve() •Save()	•Customer •Order date •Shipping address •Order items •Validate() •Retrieve() •Save()	•Product •Quantity •Purchase price •Validate() •Retrieve() •Save()

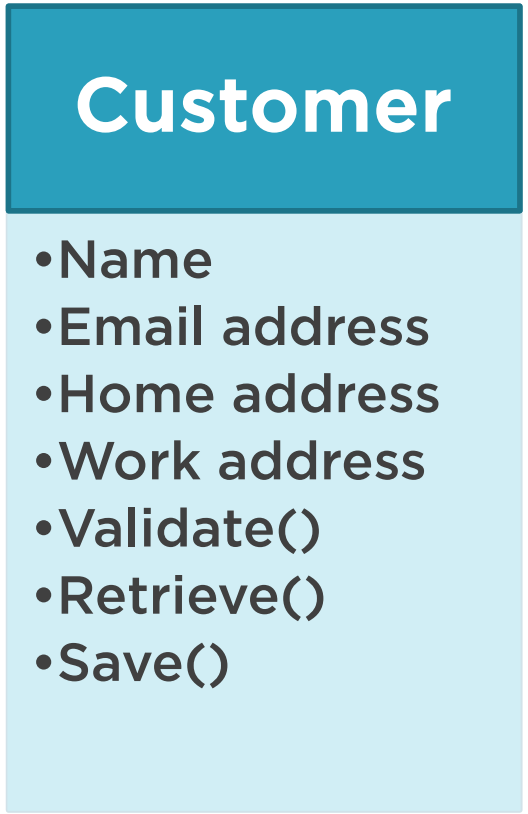
Coupling and Cohesion

Low Coupling

High Cohesion



Separating Responsibilities



Separating Responsibilities

Customer

- Name
- Email address
- Home address
- Work address
- Validate()
- Retrieve()
- Save()

Address

- Street line 1
- Street line 2
- City
- State/Province
- Postal Code
- Country
- Address type
- Validate()

Separating Responsibilities

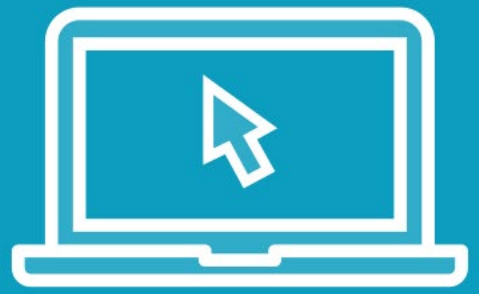
Customer Repository	Customer	Address
•Retrieve() •Save()	•Name •Email address •Home address •Work address •Validate()	•Street line 1 •Street line 2 •City •State/Province •Postal Code •Country •Address type •Validate()

Separating Responsibilities

Customer	Product	Order	Order Item
•Name •Email address •Home address •Work address •Validate()	•Product name •Description •Current price •Validate()	•Customer •Order date •Shipping addr. •Order items •Validate()	•Product •Quantity •Purchase price •Validate()

Customer Repository	Product Repository	Order Repository	Address
•Retrieve() •Save()	•Retrieve() •Save()	•Retrieve() •Save()	•Street line 1 + 2 •City •State/Province •Postal Code •Country •Address type •Validate()

Demo

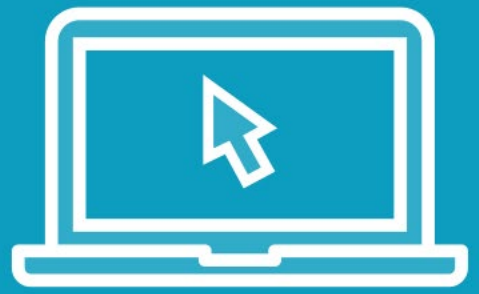


Building the Address class

Address

- Street line 1
- Street line 2
- City
- State/Province
- Postal Code
- Country
- Address type
- Validate

Demo

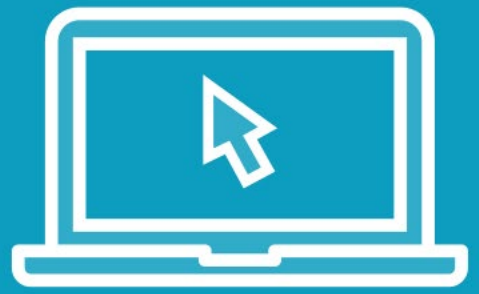


Building the Customer Repository Class

Customer Repository

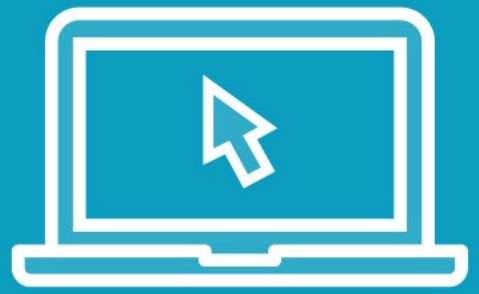
- Retrieve()
- Save()

Demo



Testing the Customer Repository Class

Demo



Building the remaining repository classes

Product Repository	Order Repository
•Retrieve() •Save()	•Retrieve() •Save()

Separating Responsibilities

Customer	Product	Order	Order Item
•Name •Email address •Home address •Work address •Validate()	•Product name •Description •Current price •Validate()	•Customer •Order date •Shipping addr. •Order items •Validate()	•Product •Quantity •Purchase price •Validate()

Customer Repository	Product Repository	Order Repository	Address
•Retrieve() •Save()	•Retrieve() •Save()	•Retrieve() •Save()	•Street line 1 + 2 •City •State/Province •Postal Code •Country •Validate()

Evaluate Coupling

What: Dependence on other classes or external resources

How: Extract dependencies into their own classes

Why: Easier to test and maintain

Example: Move the responsibility for accessing the data store to a repository class

Evaluate Cohesion

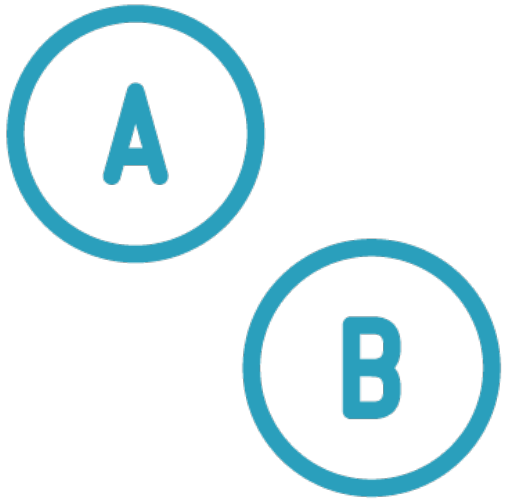
What: Class members should relate to the class purpose

How: Extract unrelated members into their own classes

Why: Easier to understand, test and maintain

Example: Move the responsibility for managing addresses into a separate class

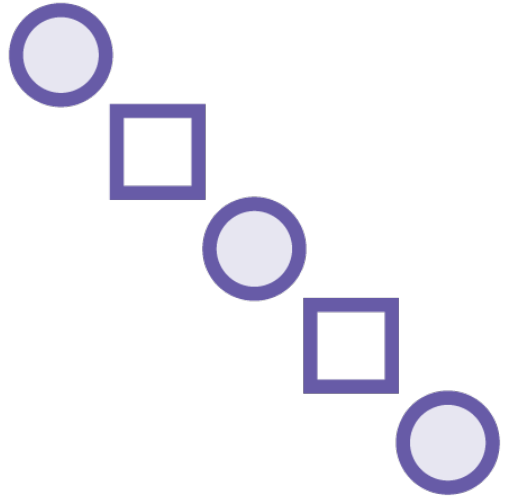
Additional Concepts



Separation of concerns

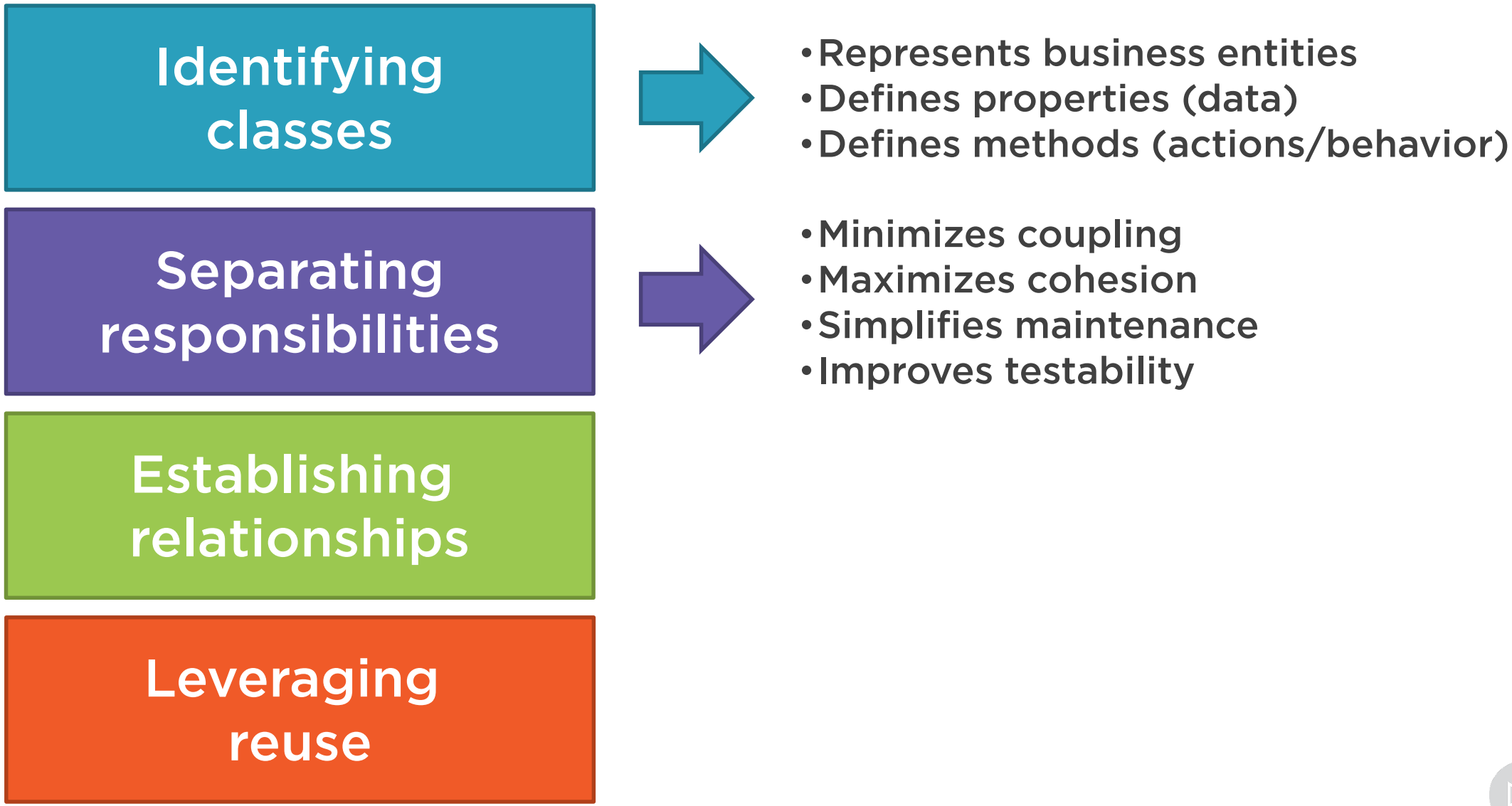
YAGNI

You aren't going to need it



Design patterns

Object-Oriented Programming (OOP)

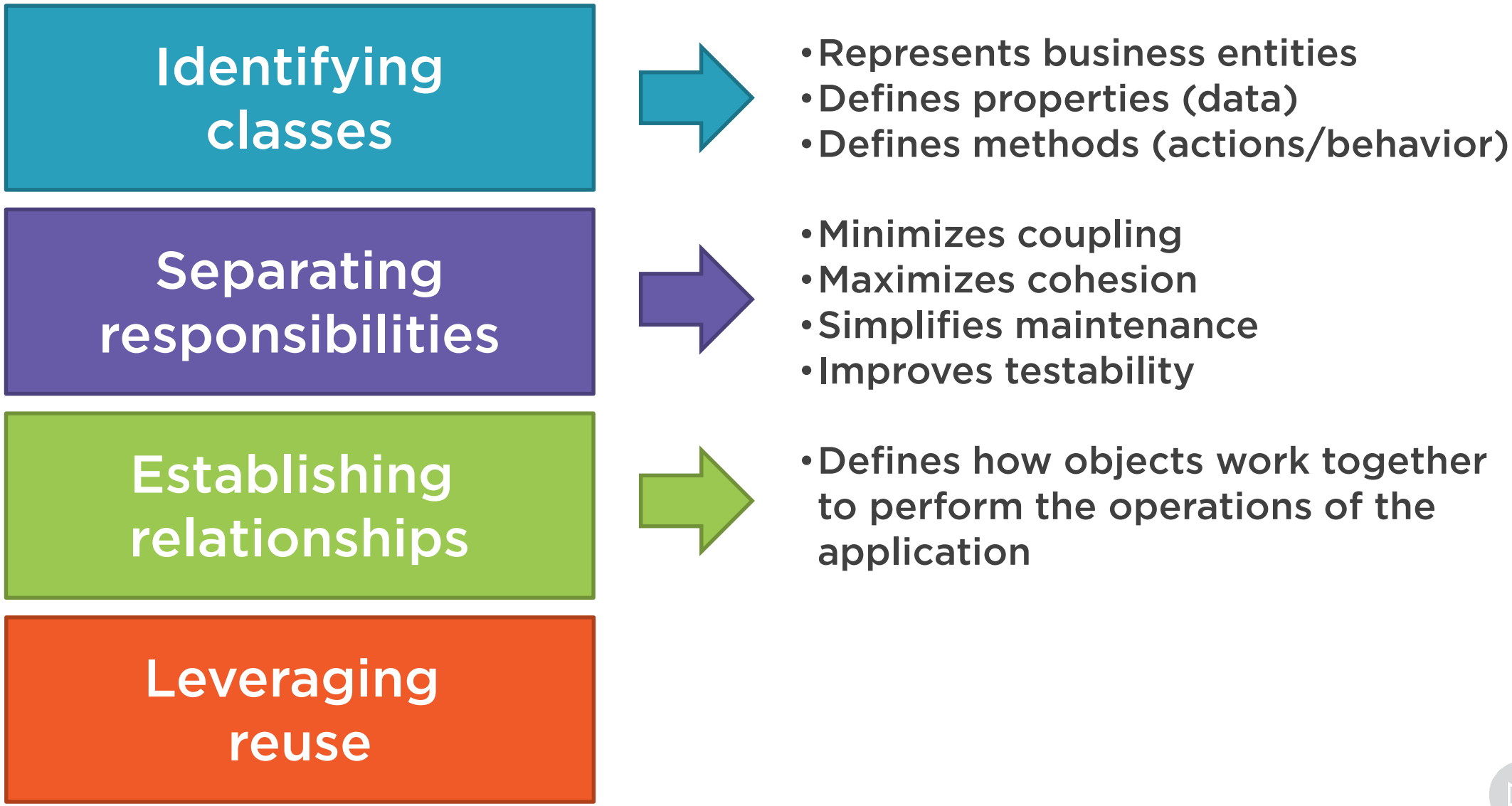


Establishing Relationships

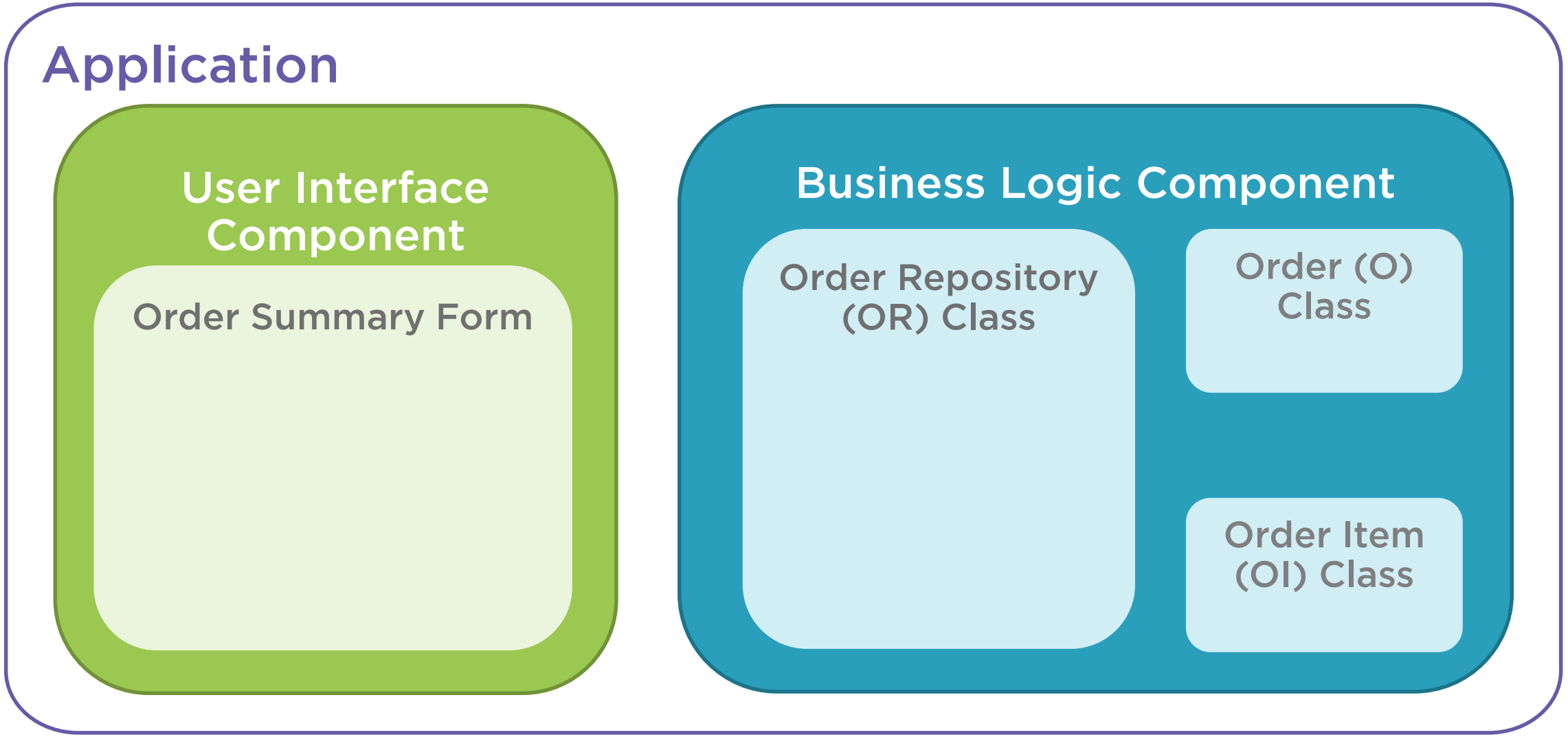
Deborah Kurata
CONSULTANT | SPEAKER | AUTHOR | MVP | GDE

@deborahkurata | blogs.msmvps.com/deborahk/

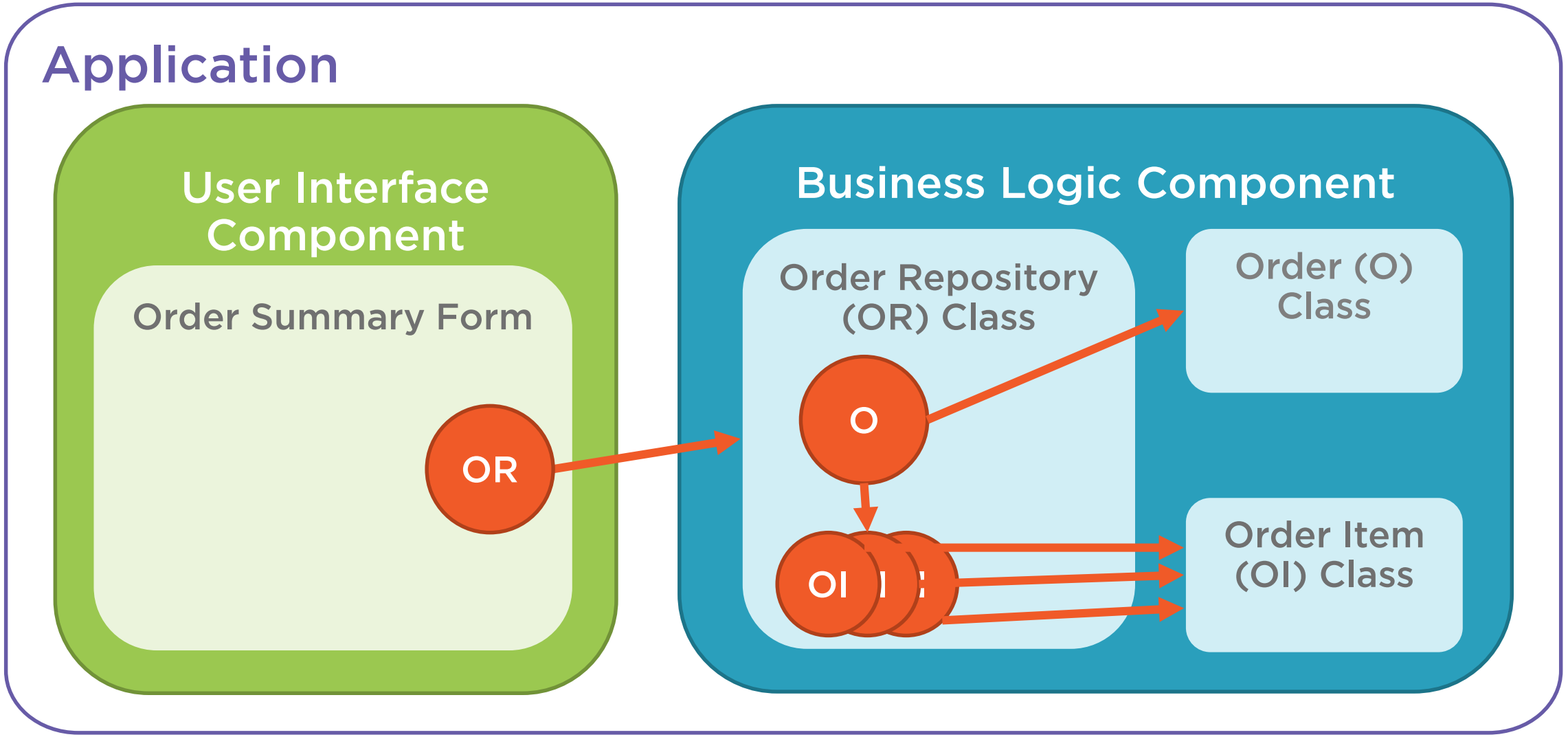
Object-Oriented Programming (OOP)



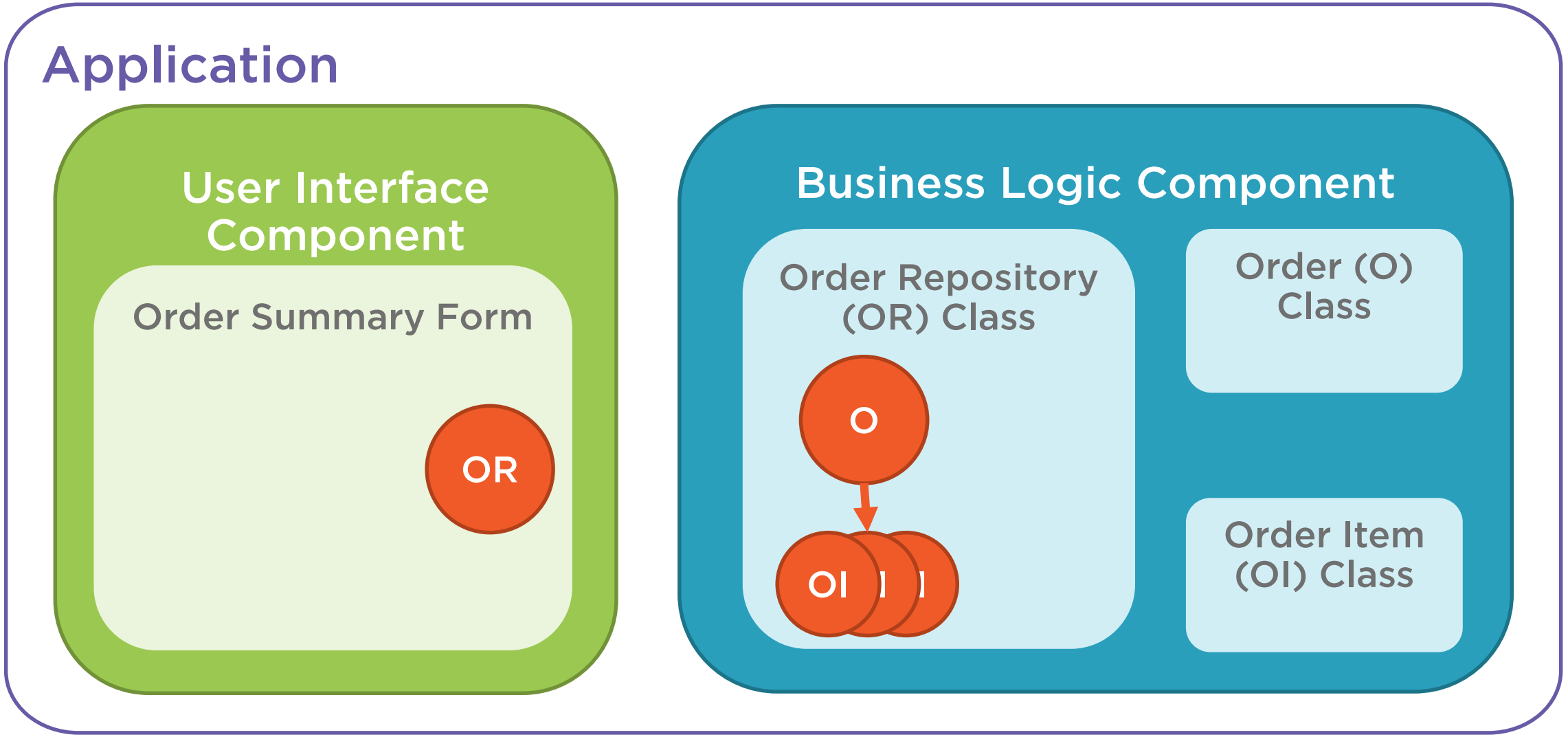
Working Together



Working Together



Working Together



Module Outline

Defining relationships

Types of relationships

Collaboration

Composition

Composition: references

Composition: IDs

Inheritance

Defining Relationships

Customer	Product	Order	Order Item
•Name •Email address •Home address •Work address •Validate()	•Product name •Description •Current price •Validate()	•Customer •Order date •Shipping addr. •Order items •Validate()	•Product •Quantity •Purchase price •Validate()

Customer Repository	Product Repository	Order Repository	Address
•Retrieve() •Save()	•Retrieve() •Save()	•Retrieve() •Save()	•Street line 1 + 2 •City •State/Province •Postal Code •Country •Address type •Validate()

Defining Relationships

Product Repository

Product

Order Repository

Order

Order Item

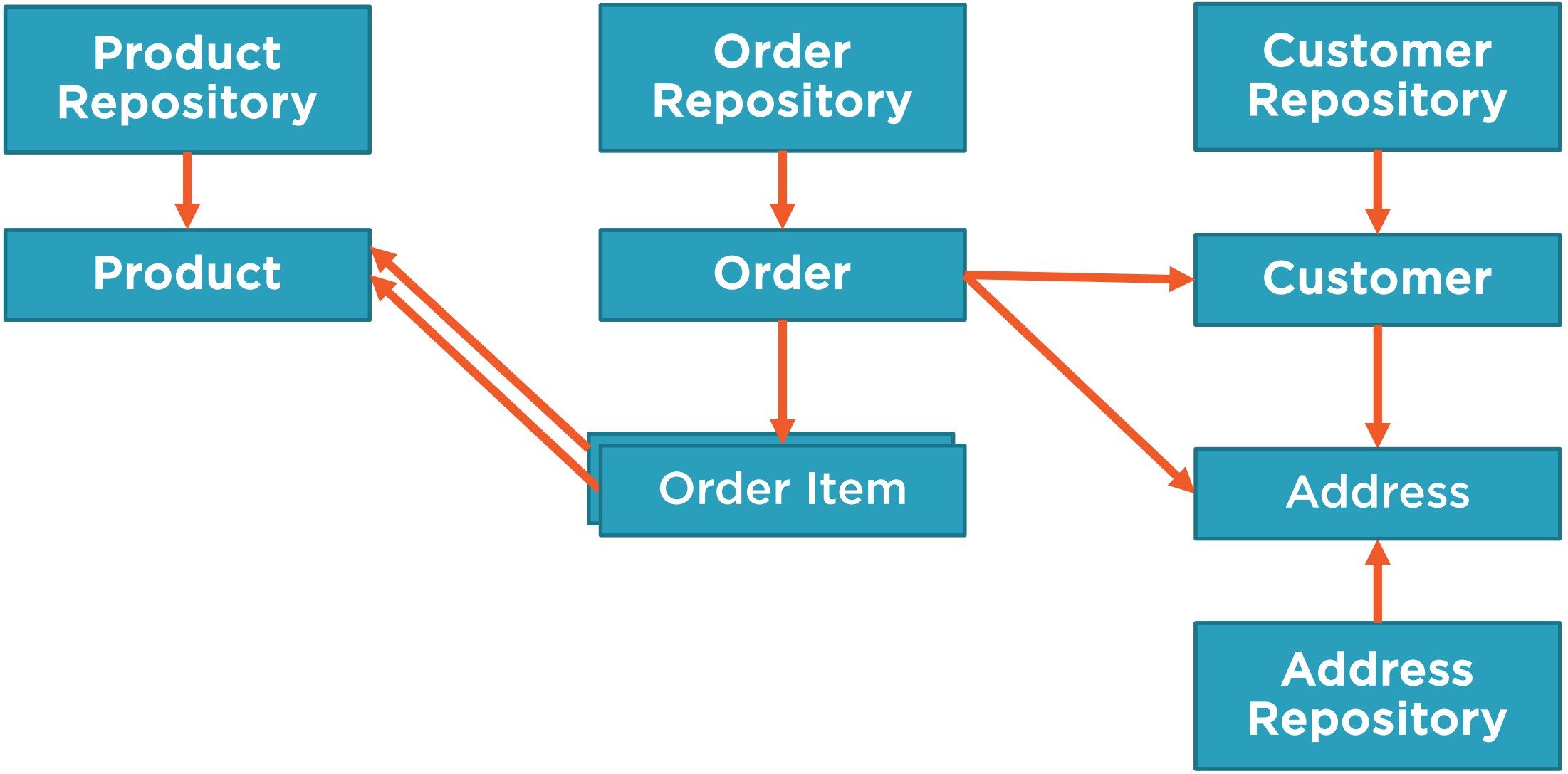
Customer Repository

Customer

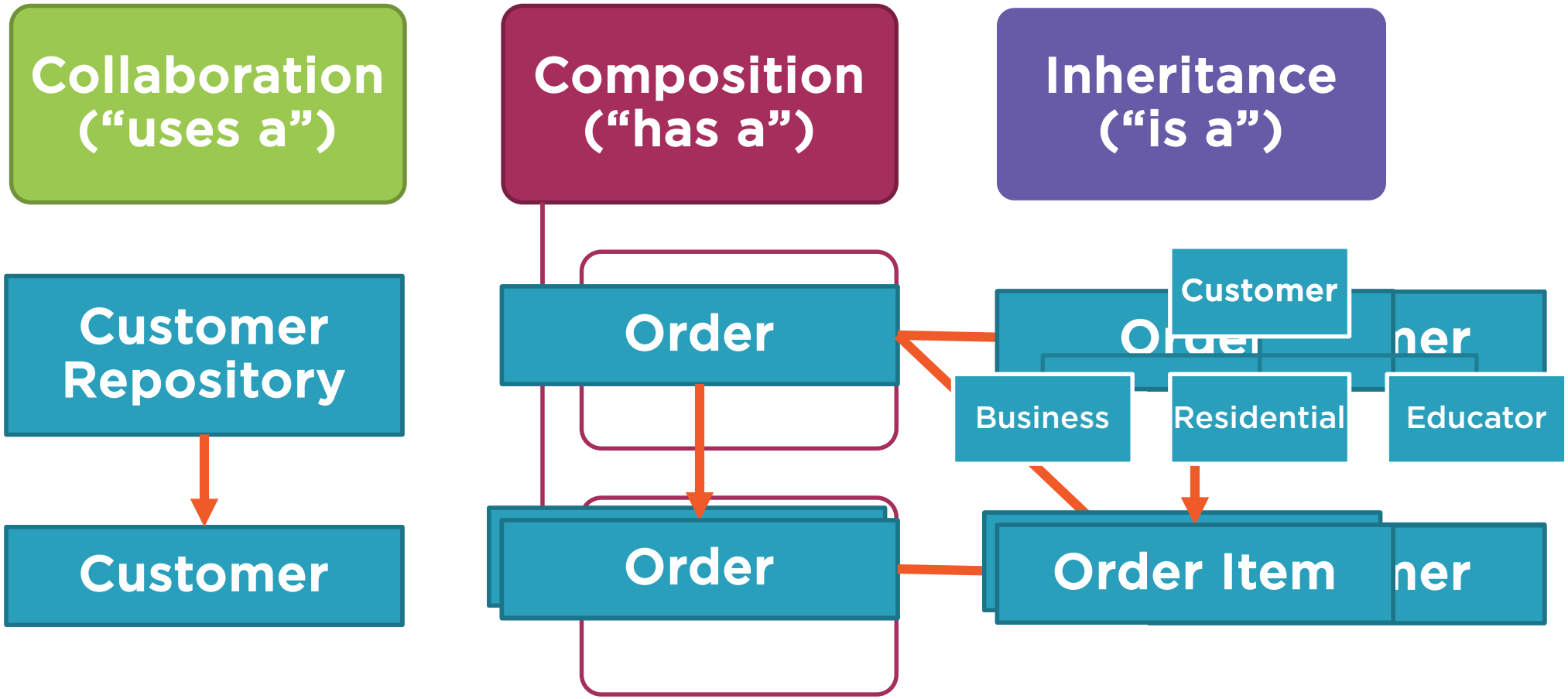
Address

Address Repository

Defining Relationships

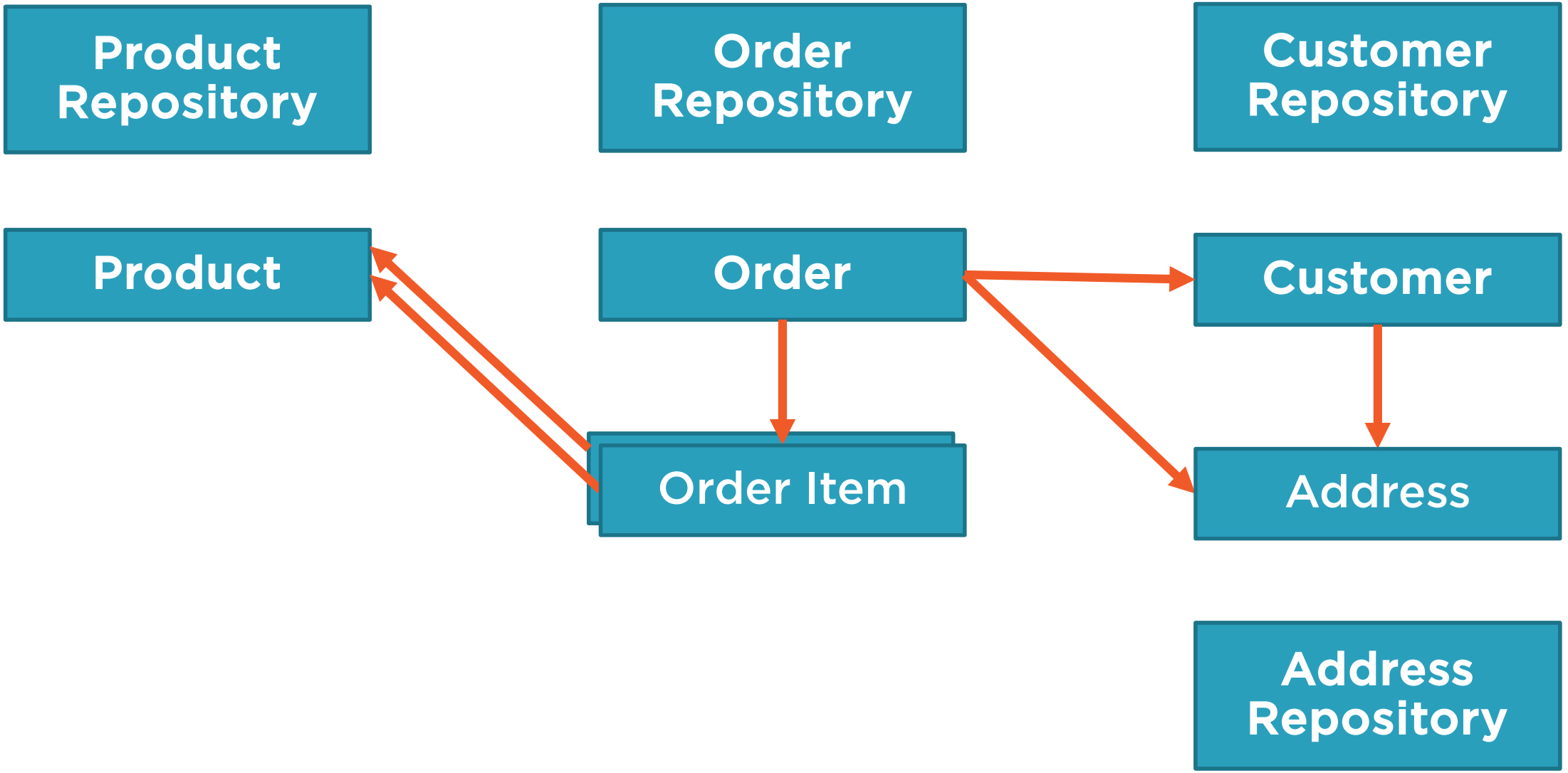


Types of Relationships



Collaboration ("uses a")

Composition ("has a")



Composition ("has a")

Product

Order

- Customer
- Order date
- Shipping addr.
- Order items
- Validate()

Order Item

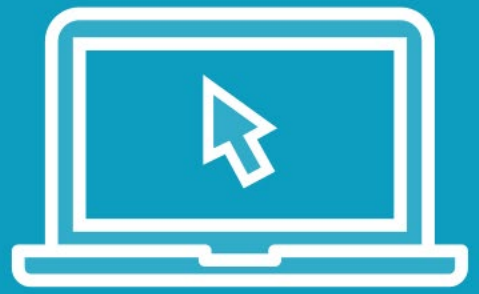
- Product
- Quantity
- Purchase price
- Validate()

Customer

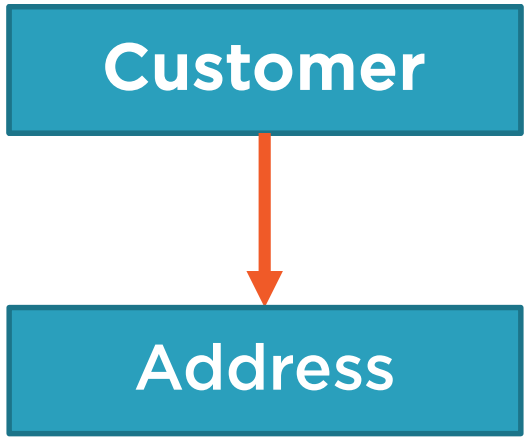
- Name
- Email address
- Home address
- Work address
- Validate()

Address

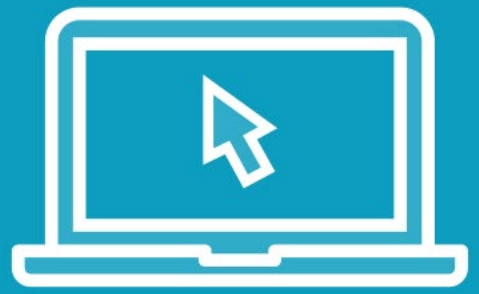
Demo



Implementing a composition relationship

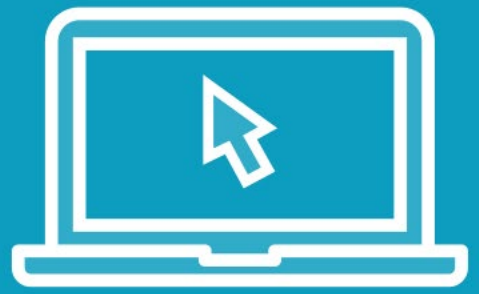


Demo



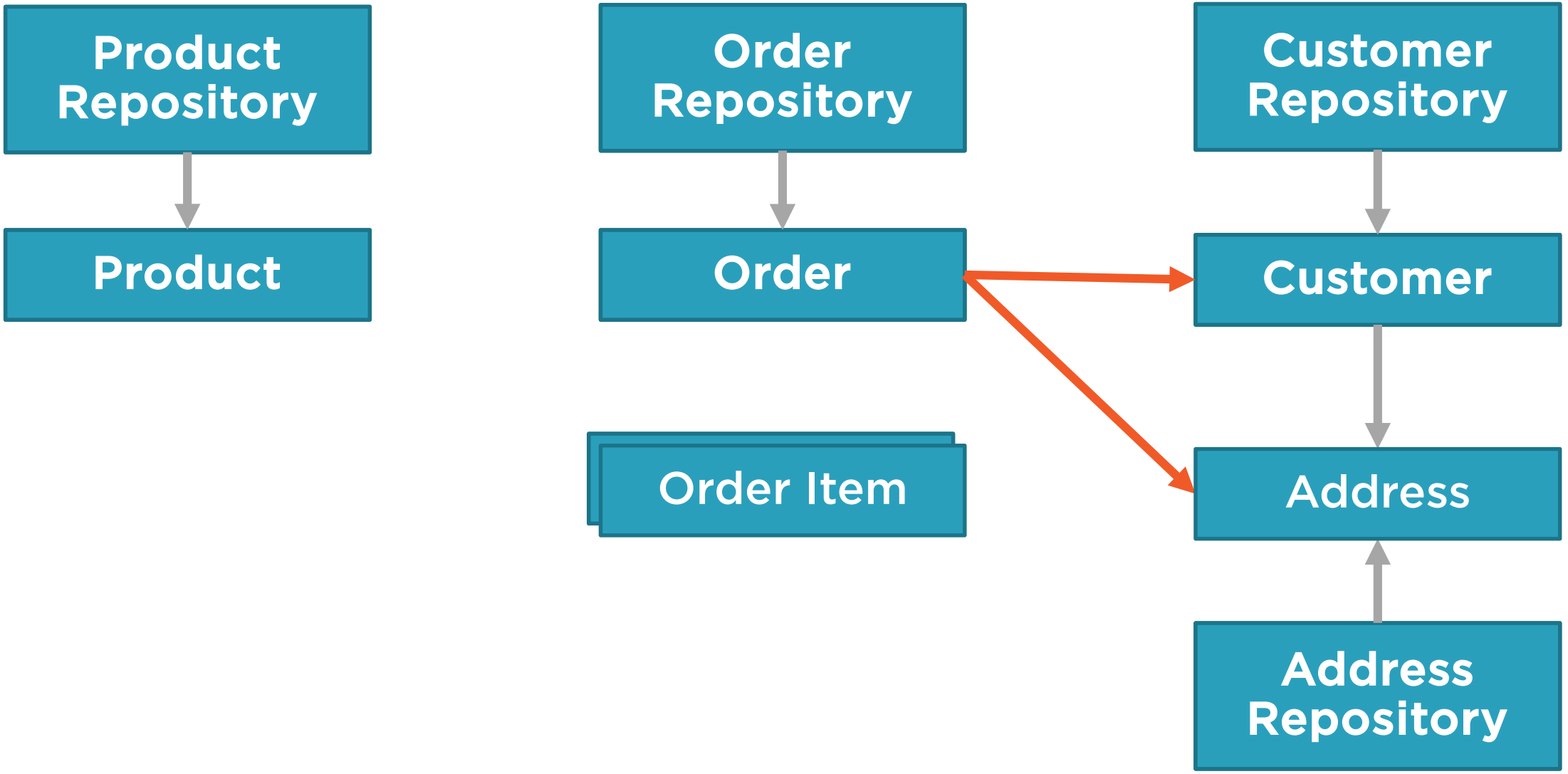
Populating the referenced object

Demo



Testing a composition relationship

Relationships



Advantages of Using IDs

Reduces coupling

Increases efficiency

Object Relationships

Collaboration
("uses a")

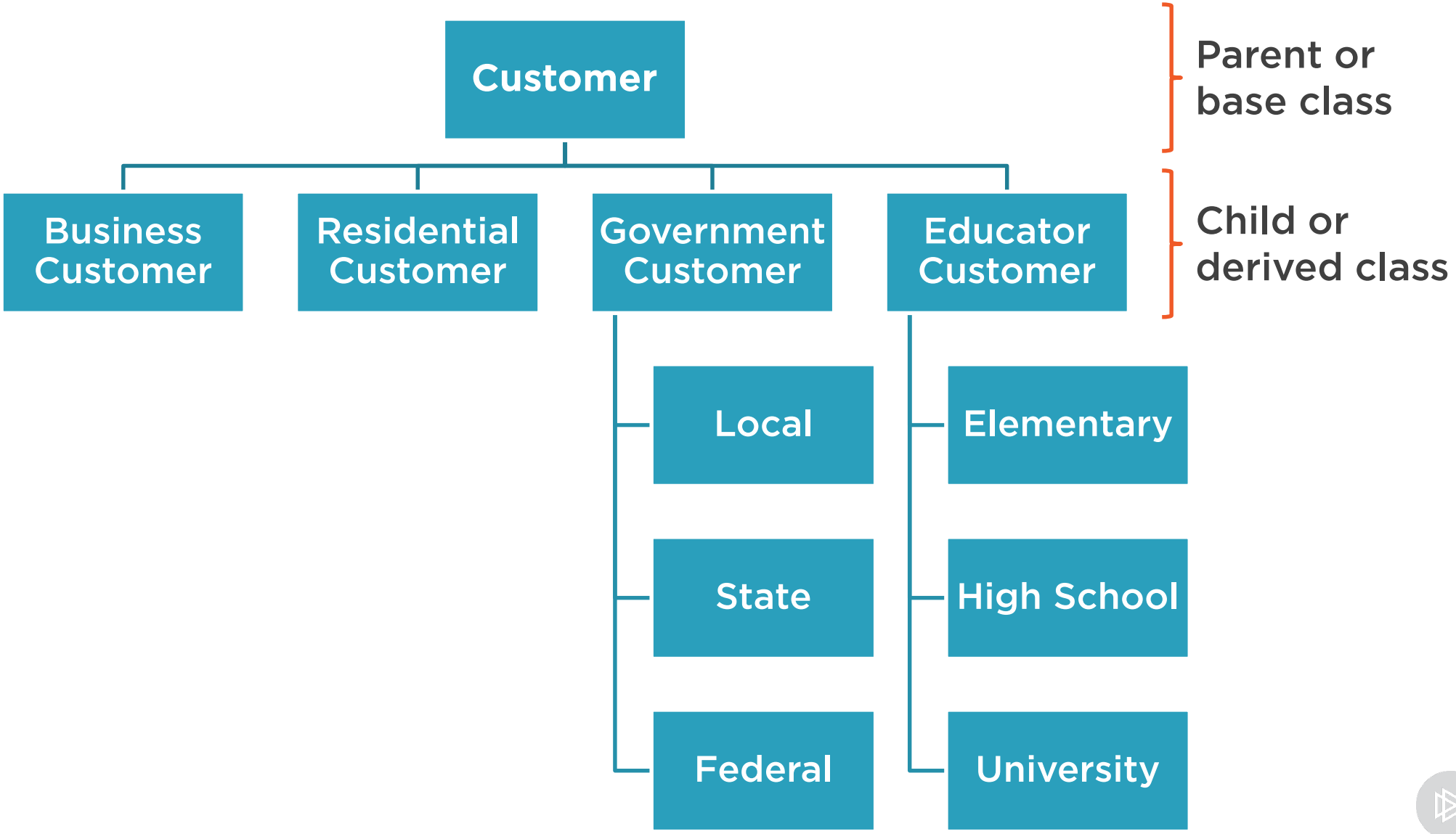
Composition
("has a")

Inheritance
("is a")

“The new system must manage business, residential, government, and educator types of customers.”

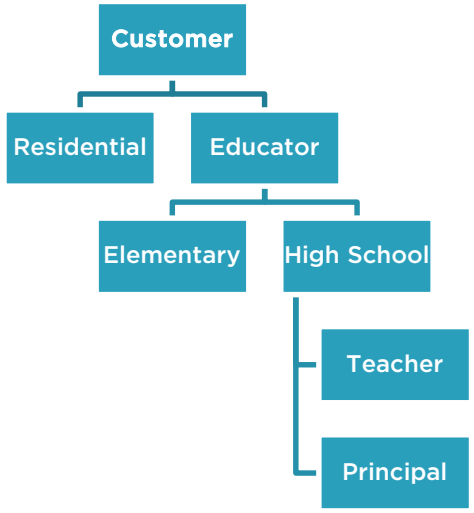
From the requirements

Inheritance ("is a")



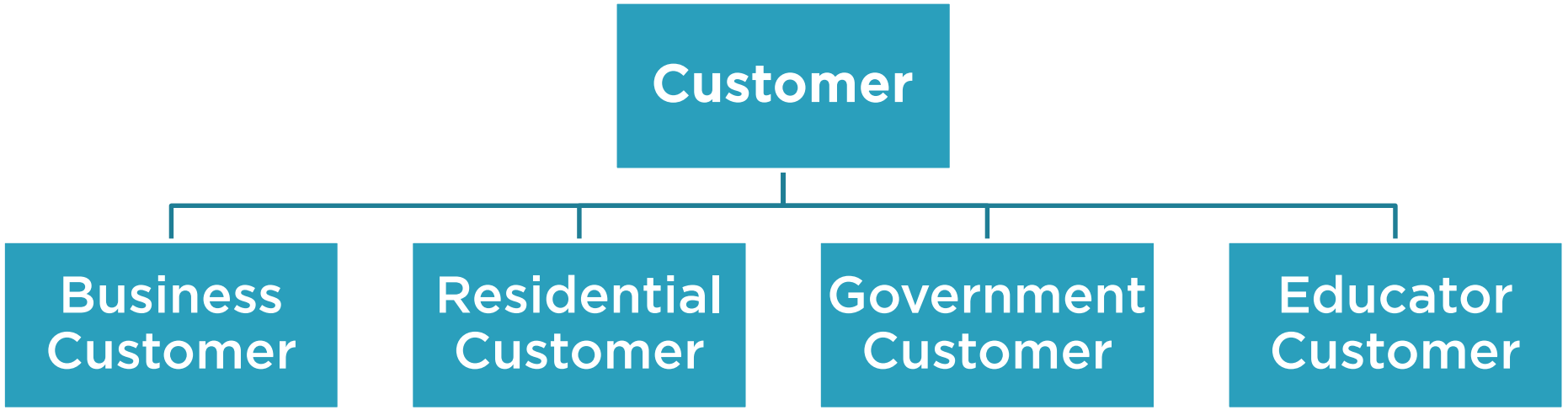
Inheritance in C#

A class can only have one parent class

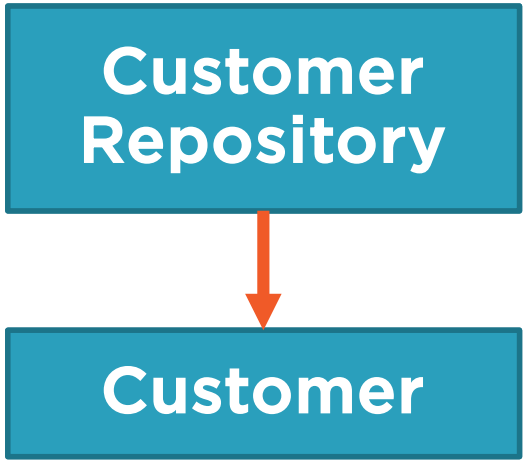


There can be any number of inheritance levels

Inheritance



Collaboration ("uses a")

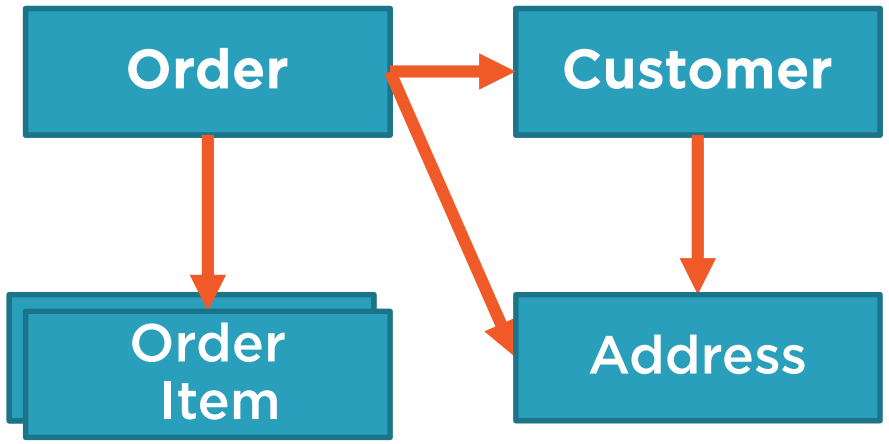


A class uses another class that is otherwise unrelated

Customer Repository "uses a" Customer instance to populate data

```
Customer customer = new Customer(customerId)
{
  EmailAddress = "fbaggins@hobbiton.me",
  FirstName = "Frodo",
  LastName = "Baggins"
};
```

Composition ("has a")



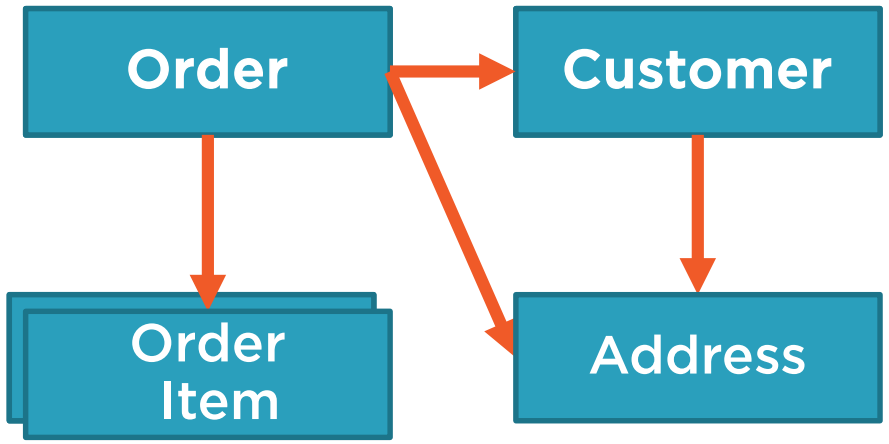
A class is made up of parts from other classes

Order "has a" customer

Order "has a" shipping address

Order "has a" set of order items

Composition ("has a")



Implement as a property reference

```
public List<OrderItem> orderItems { get; set; }
```

Or as an Id

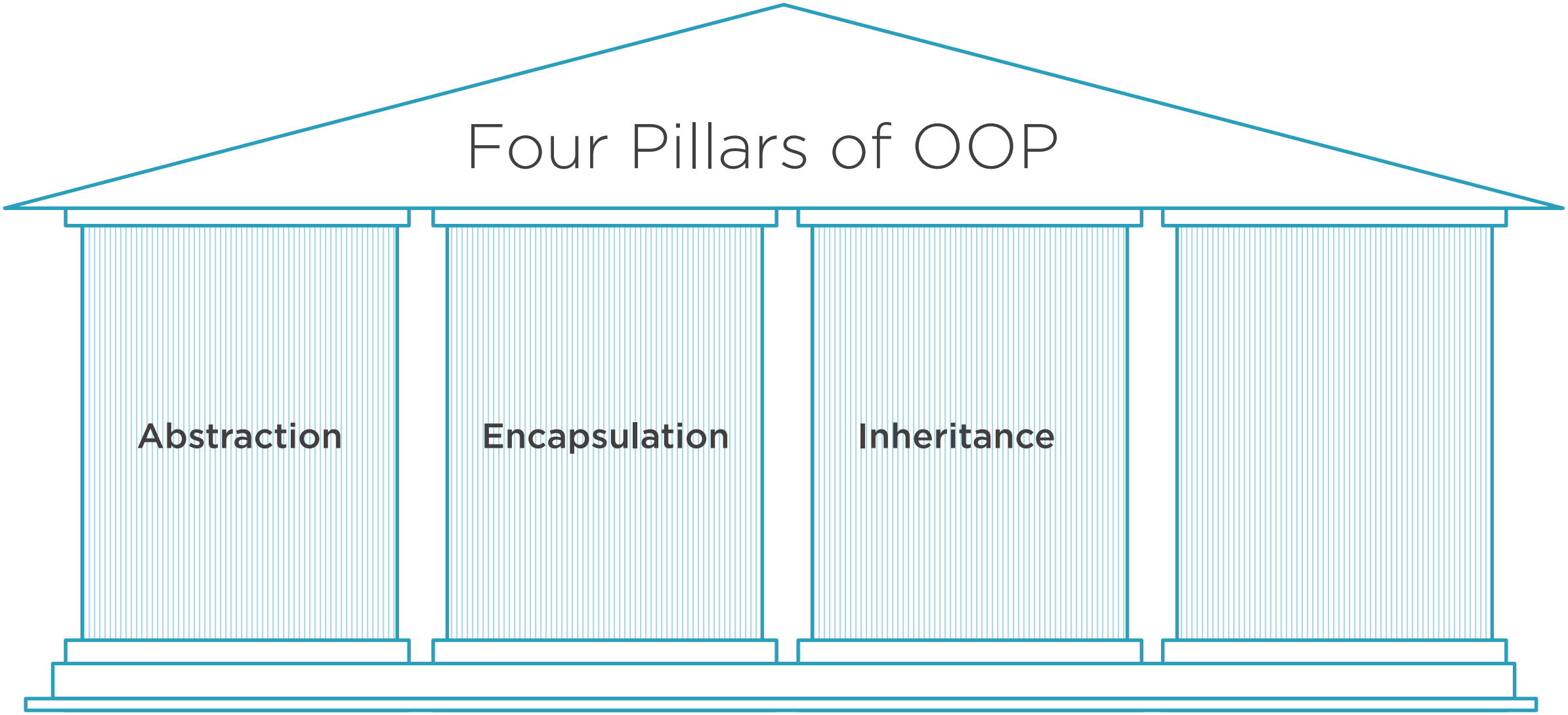
```
public int CustomerId { get; set; }
```

Inheritance ("is a")

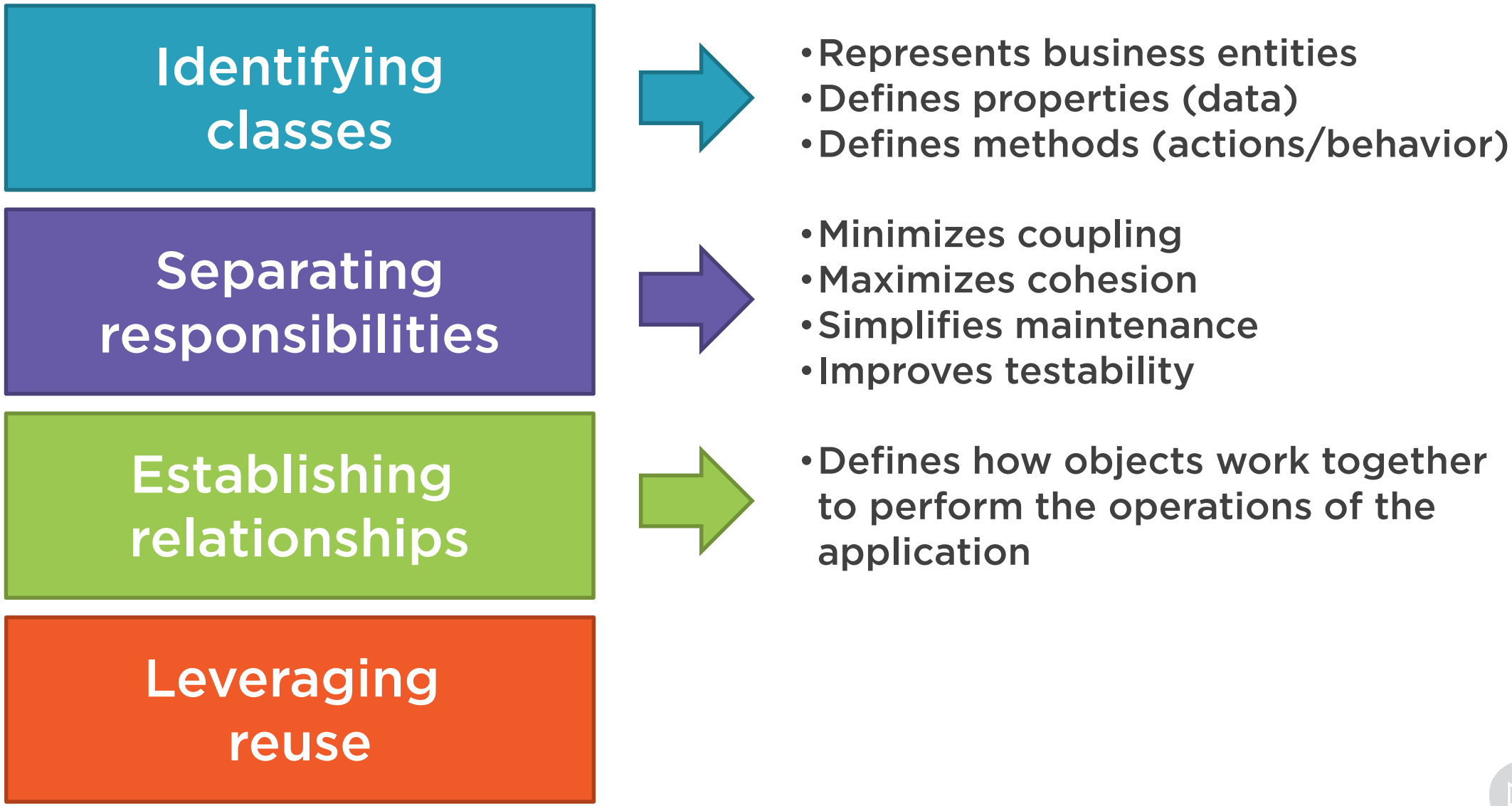
Define classes that are a more specialized version of another class

Business Customer, Residential Customer, Educator Customer

Only implement an inheritance relationship if the specific class type adds unique code


Four Pillars of OOP
Abstraction
Encapsulation
Inheritance

Object-Oriented Programming (OOP)

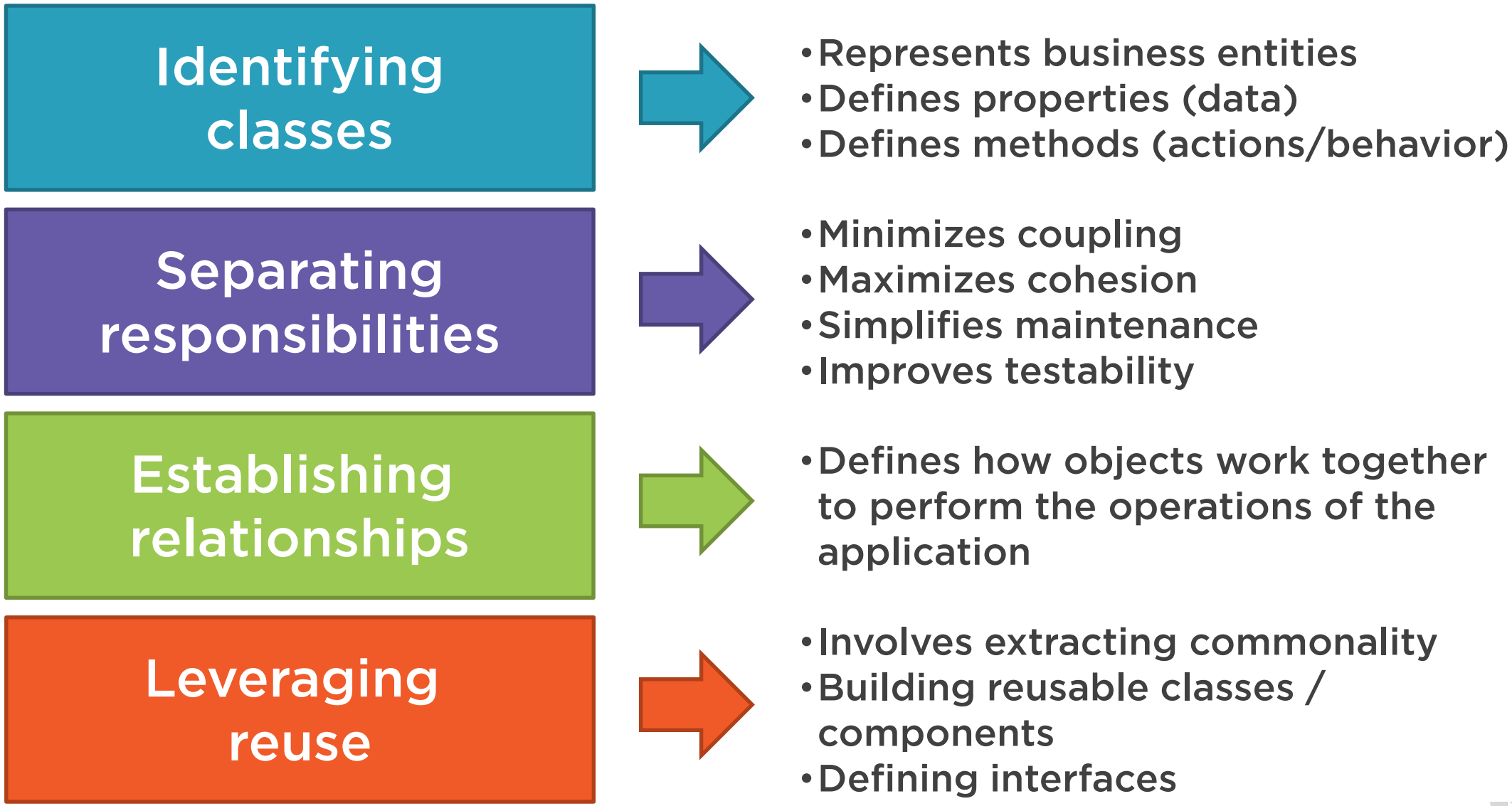


Leveraging Reuse through Inheritance

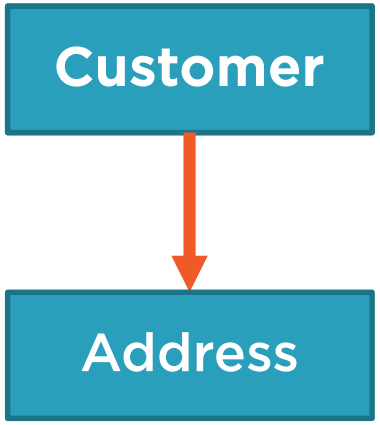
Deborah Kurata
CONSULTANT | SPEAKER | AUTHOR | MVP | GDE

@deborahkurata | blogs.msmvps.com/deborahk/

Object-Oriented Programming (OOP)

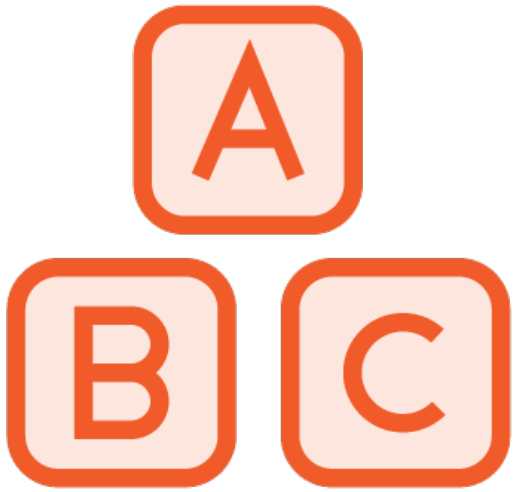


Techniques for Leveraging Reuse

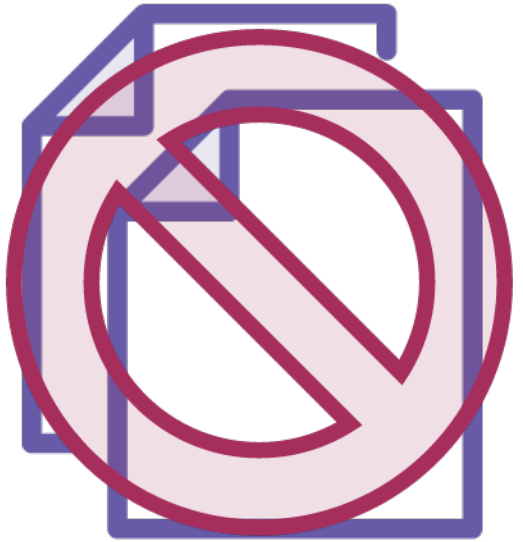


Collaboration / Composition

Inheritance

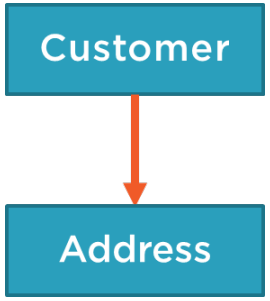


Components



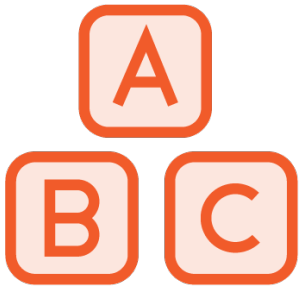
Copy/Paste

Techniques for Leveraging Reuse

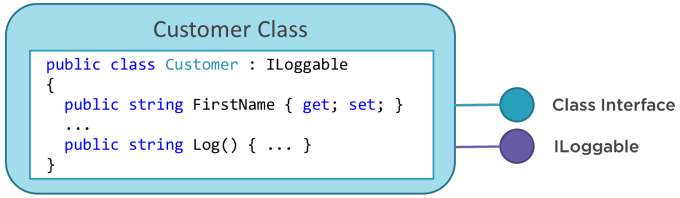


Collaboration / Composition

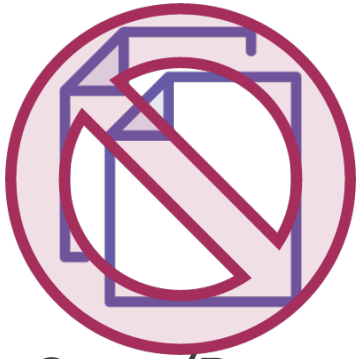
Inheritance



Components



Interfaces



Copy/Paste

Module Outline

The .NET Object class

Overriding base class functionality

Polymorphism

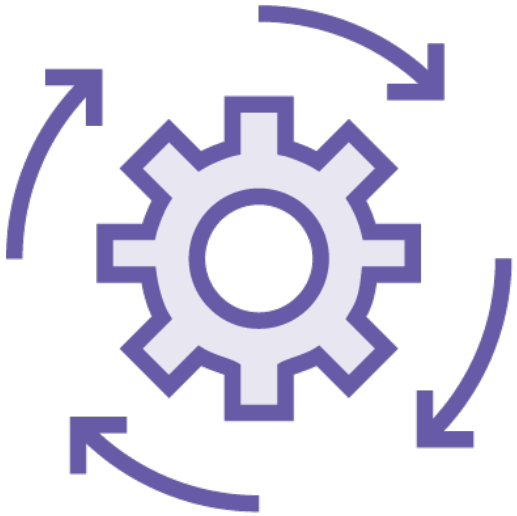
Building a base class

Secrets of Reuse

Build it once

Test it

Reuse it!



Update it in one place

Advantages of Reuse

Reduces amount of code

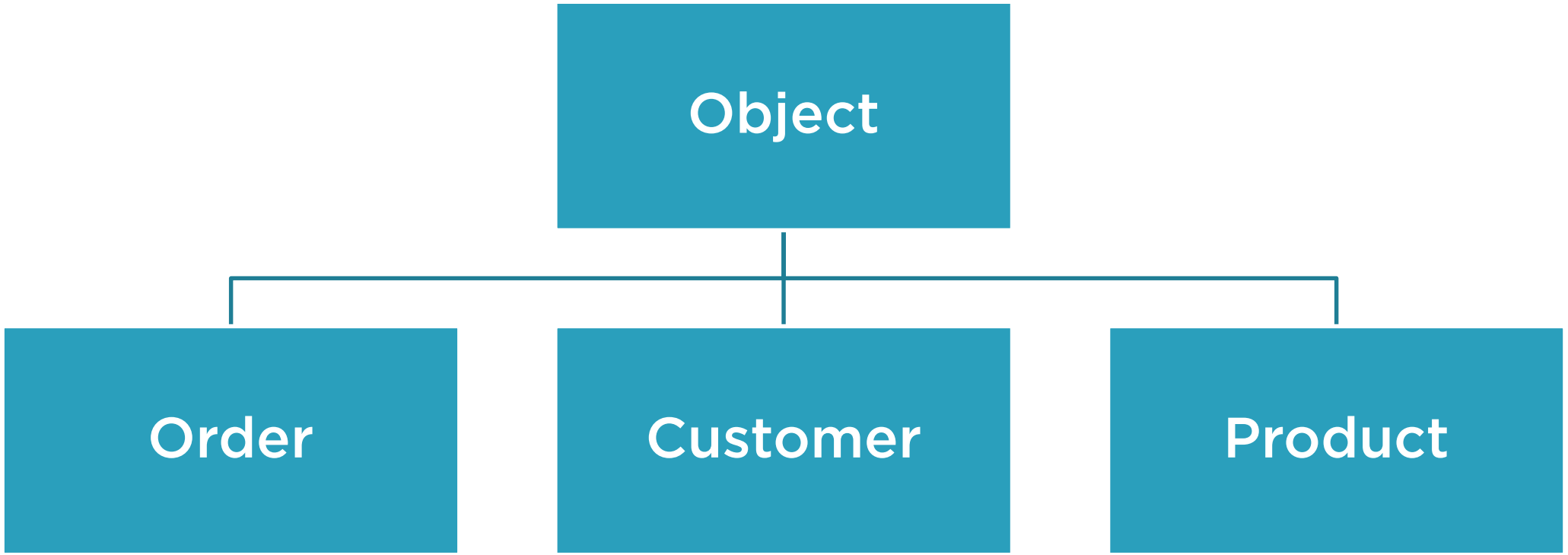
Reduces development time

Reduces costs

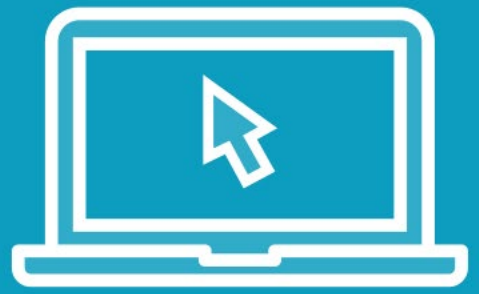
Reduces bugs

Inheritance

.NET Object Class

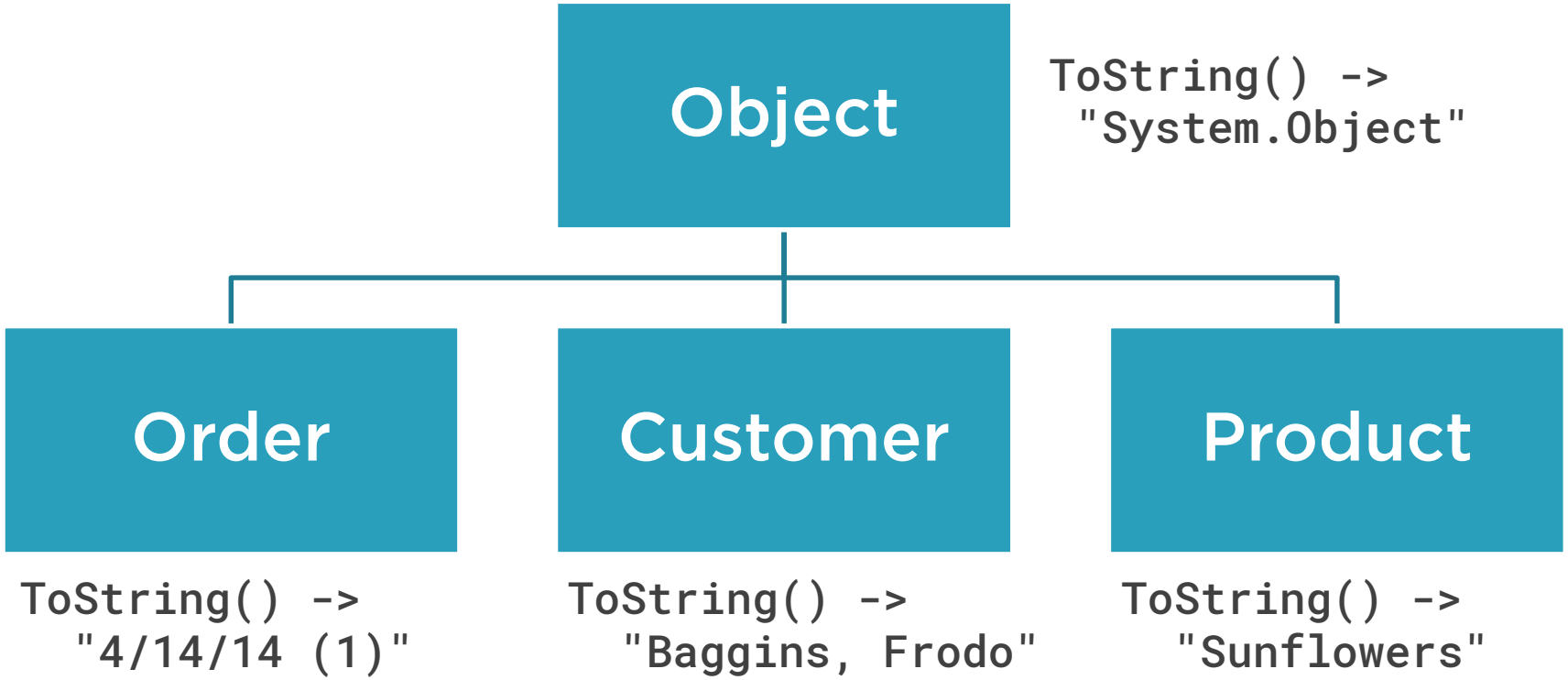


Demo



Overriding base class functionality

Polymorphism



Saving

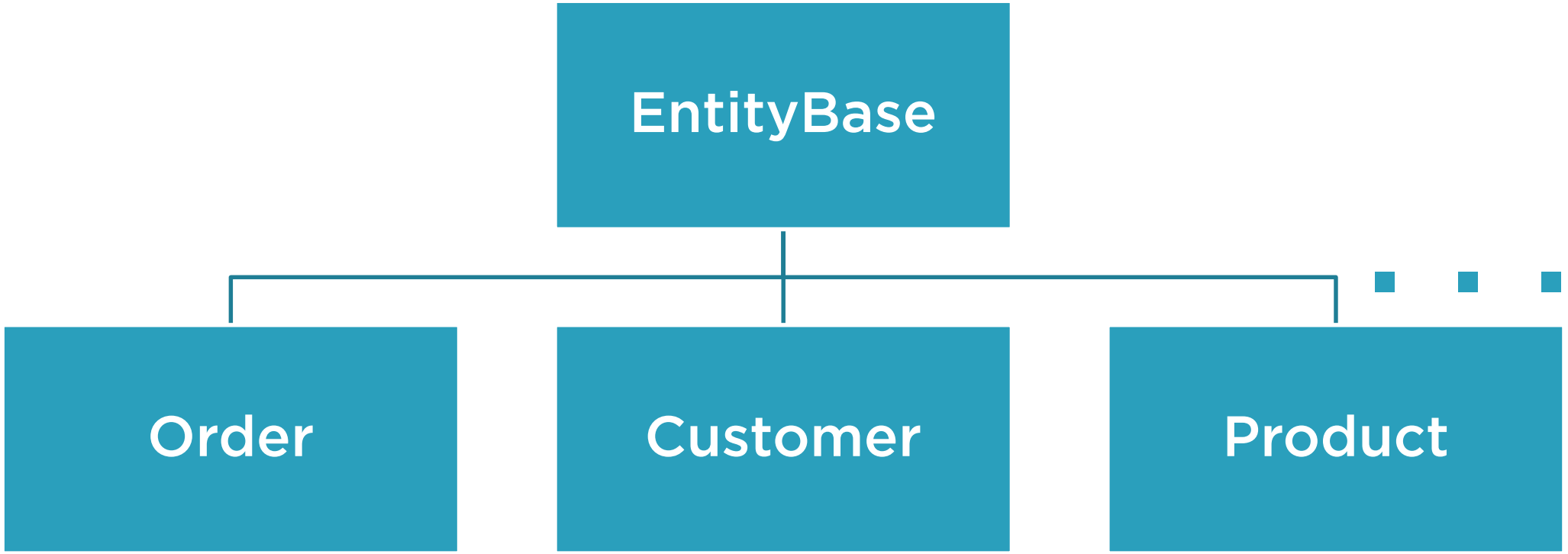
```
public bool IsNew { get; private set; }
```

```
public bool HasChanges { get; set; }
```

```
public bool IsValid => Validate();
```

```
public EntityStateOption EntityState { get; set; }
```

Base Class



Building a Base Class

Abstract Class

Incomplete, with at least one property or method not implemented

Cannot be instantiated

Intended for use as a base class

```
public abstract class EntityBase
{

}
```

Concrete Class

Normal class

Can be instantiated

Can be used as a base class

```
public class EntityBase
{

}
```

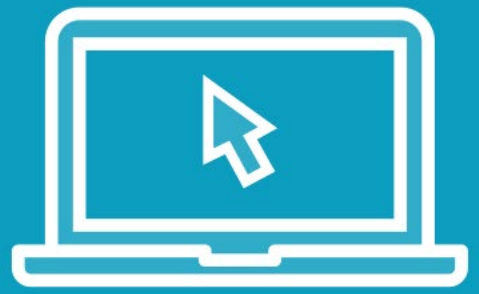
Sealed Class

Cannot be extended through inheritance

Sealed using the `sealed` keyword

```
public sealed class Customer
{

}
```

Demo



Building a base class

```
public bool IsNew { get; private set; }
```

```
public bool HasChanges { get; set; }
```

```
public bool IsValid => Validate();
```

```
public EntityStateOption EntityState { get; set; }
```

Sealed Members

By default, class members are sealed and cannot be overridden

Expose members using

- Abstract
- Virtual

Preparing Overridable Base Class Members

Abstract

Method signature as place holder with no implementation

Only use in abstract classes

Must be overridden by derived class

```
public abstract bool Validate();
```

Virtual

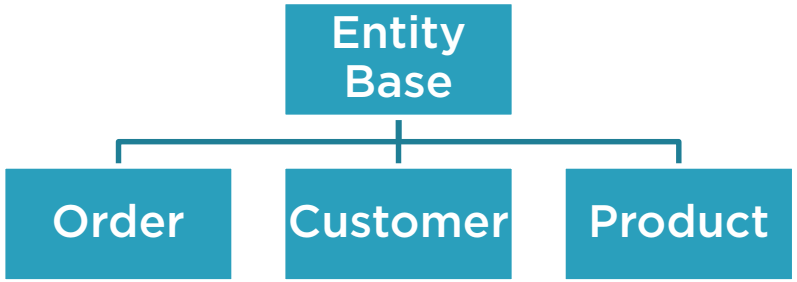
Method with default implementation

Use in abstract or concrete classes

Optionally overridden by derived class

```
public virtual bool Validate()
{
    ...
}
```

Inheritance



Define a base class with common functionality

```
public class EntityBase
{
    public bool HasChanges { get; set; }

}
```

Inherit from that class to reuse its functionality

```
public class Product : EntityBase
{
}
```

Abstract Class

An incomplete class with one or more members that are not implemented

An abstract class cannot be instantiated

Intended for use as a base class

```
public abstract class EntityBase
{

}
```

Sealed Class

A concrete class that cannot be extended through inheritance

Use it to prevent overriding the class functionality

```
public sealed class Customer
{

}
```

Abstract vs. Virtual Methods

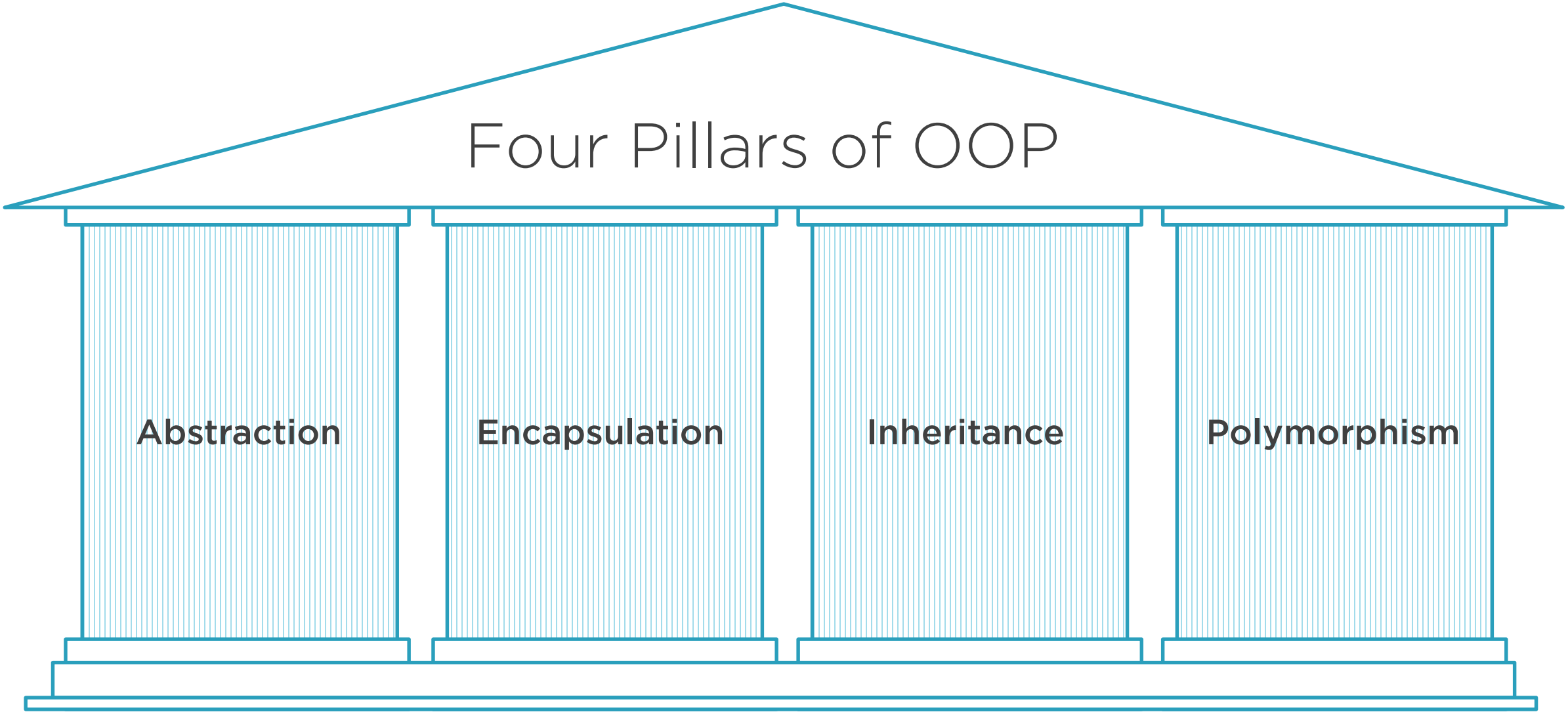
Abstract method is a placeholder, no implementation, that must be overridden

```
public abstract bool Validate();
```

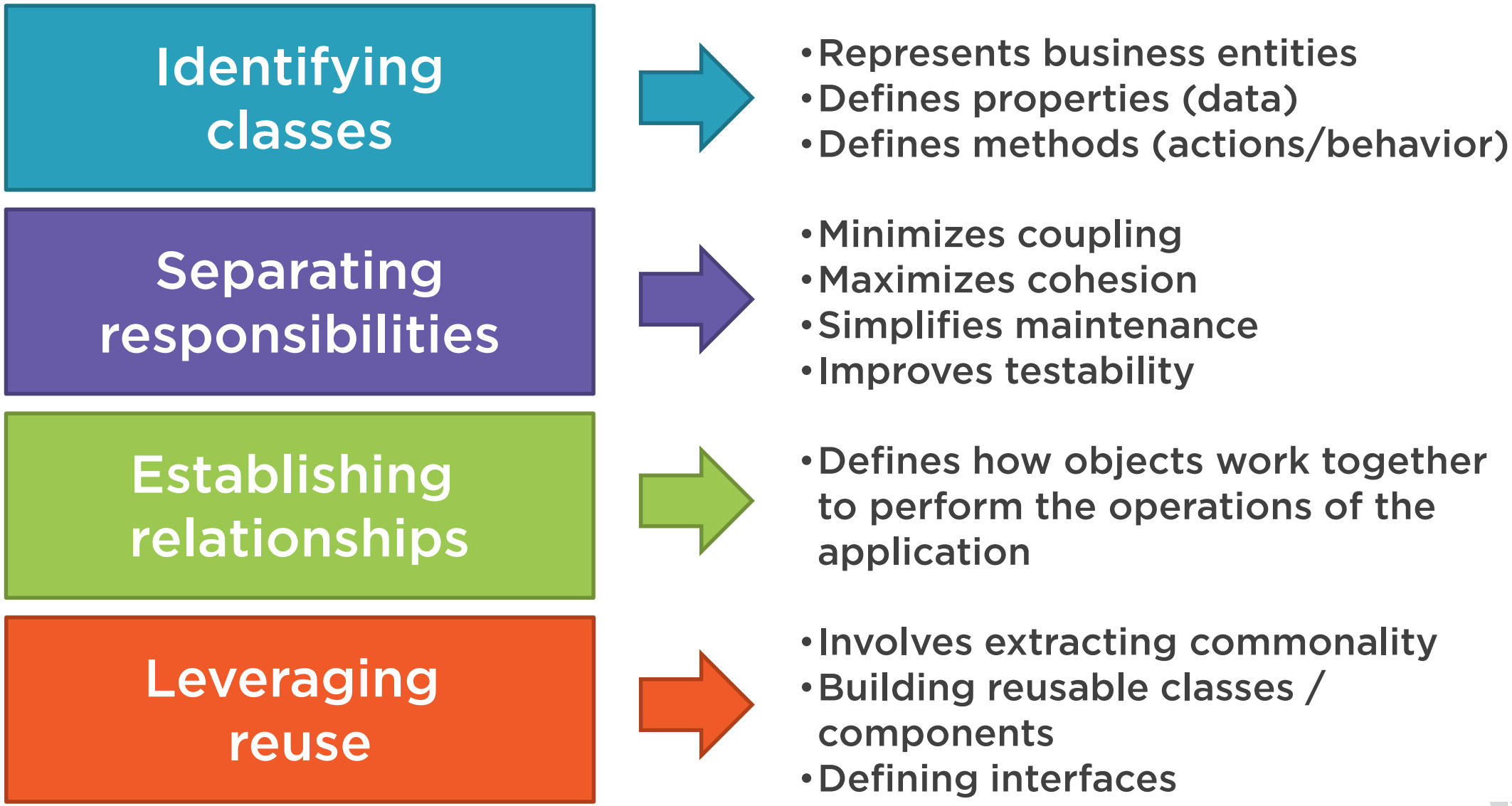
Virtual method is a method with a default implementation that can be overridden

```
public virtual bool Validate()
{
    // Default implementation
}
```

Four Pillars of OOP



Object-Oriented Programming (OOP)

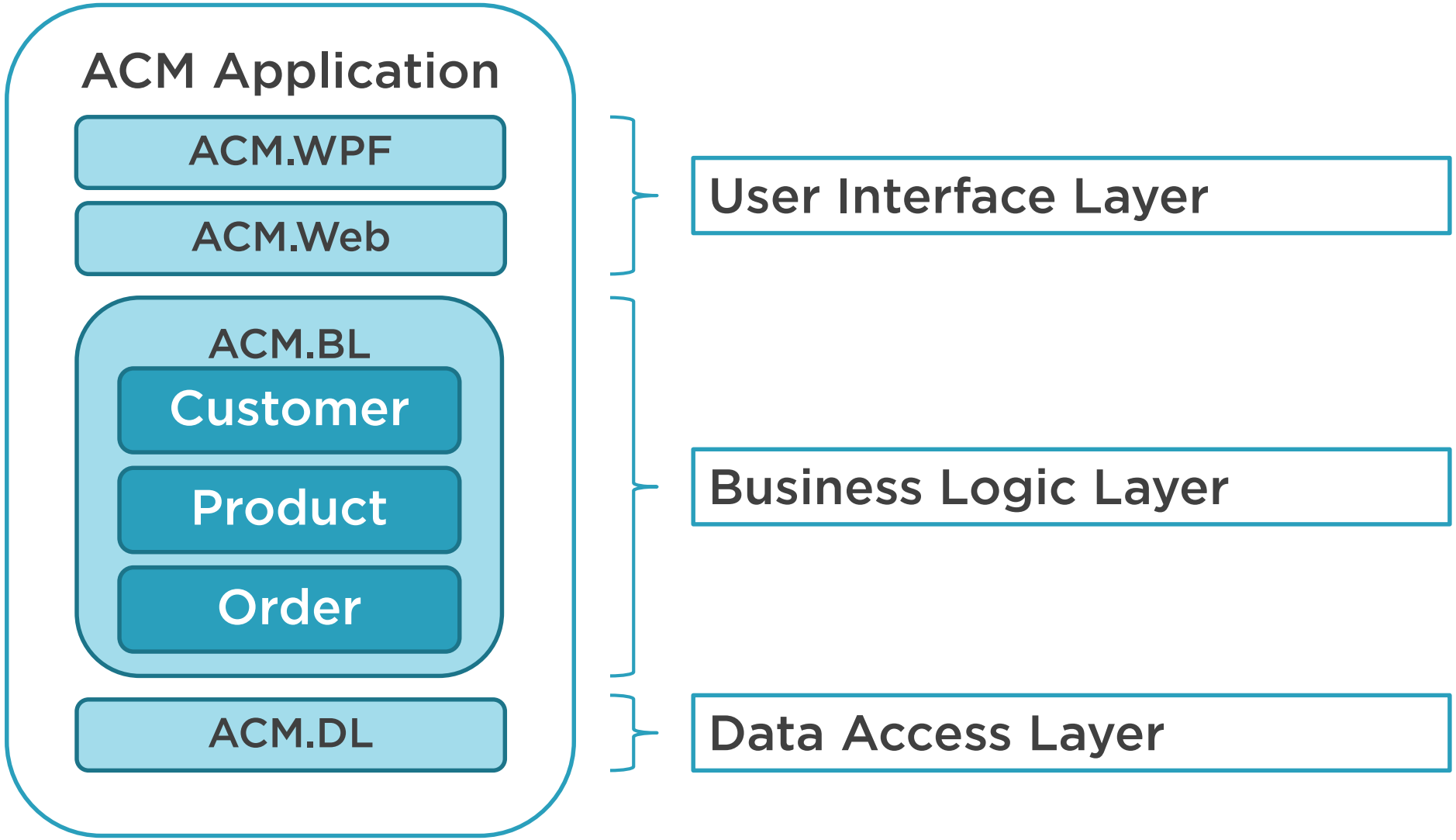


Building Reusable Components

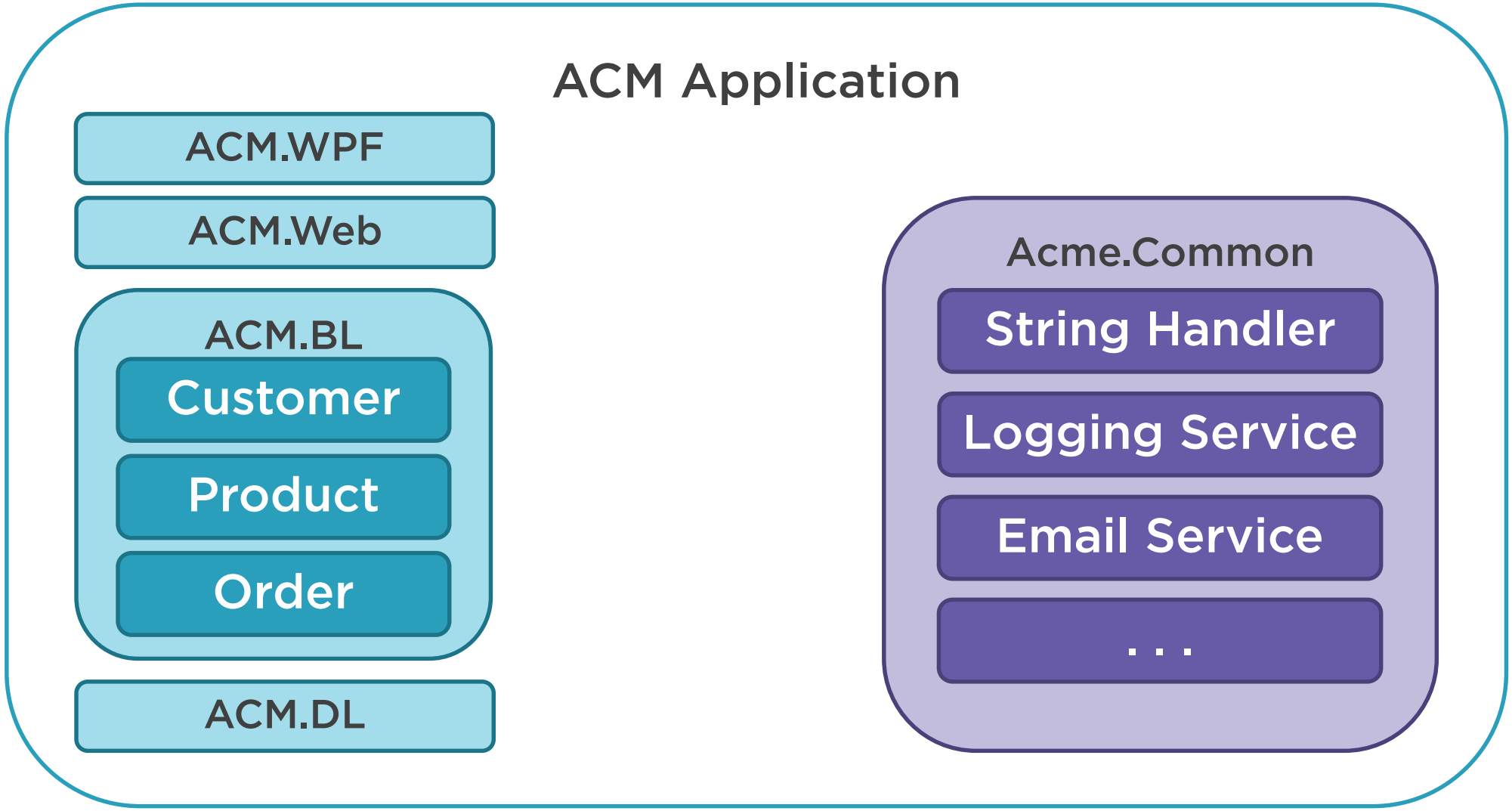
Deborah Kurata
CONSULTANT | SPEAKER | AUTHOR | MVP | GDE

@deborahkurata | blogs.msmvps.com/deborahk/

Components


ACM Application
ACM.WPF
ACM.Web
ACM.BL
Customer
Product
Order
ACM.DL
User Interface Layer
Business Logic Layer
Data Access Layer

Reusable Library Component



Module Outline

Building a reusable component

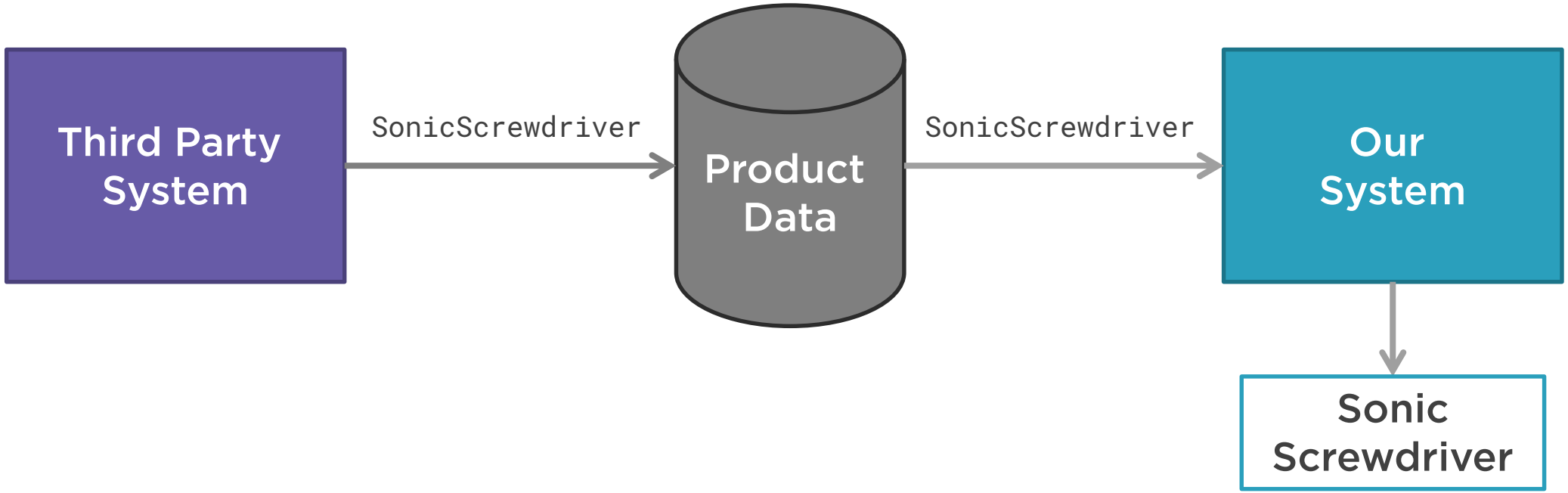
Testing a reusable component

Using the reusable component

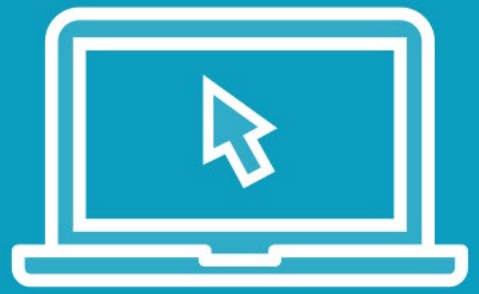
Static classes

Extension methods

Scenario

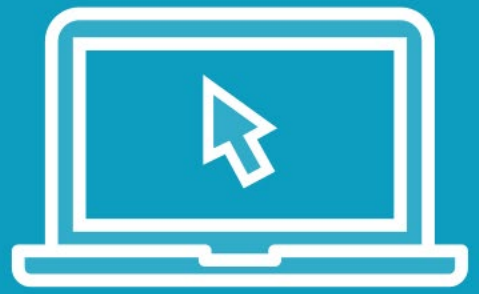

Third Party System
SonicScrewdriver
Product Data
SonicScrewdriver
Our System
Sonic Screwdriver

Demo



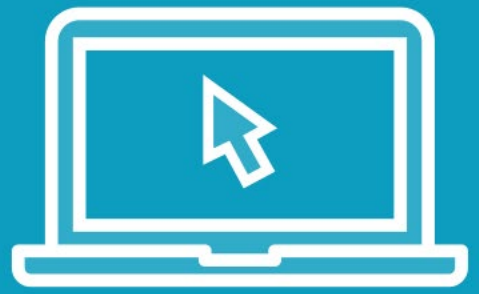
Building a reusable component

Demo



Testing a reusable component

Demo



Using a reusable component

Normal Class

Creating the method

```
public class StringHandler
{
  public string InsertSpaces(string source)
  { ...
  }
}
```

Using the method

```
var stringHandler = new StringHandler();

return stringHandler.InsertSpaces(productName);
```

Static Class

A class that cannot be instantiated

Access members using the class name

Members must also be static

Often used as a container for utility methods

Normal vs. Static Class

Creating the method

```
public class StringHandler
{
  public string InsertSpaces(string source)
  { ...
  }
}
```

Using the method

```
var stringHandler = new StringHandler();

return stringHandler.InsertSpaces(productName);
```

Creating the method

```
public static class StringHandler
{
  public static string InsertSpaces(string source)
  { ...
  }
}
```

Using the method

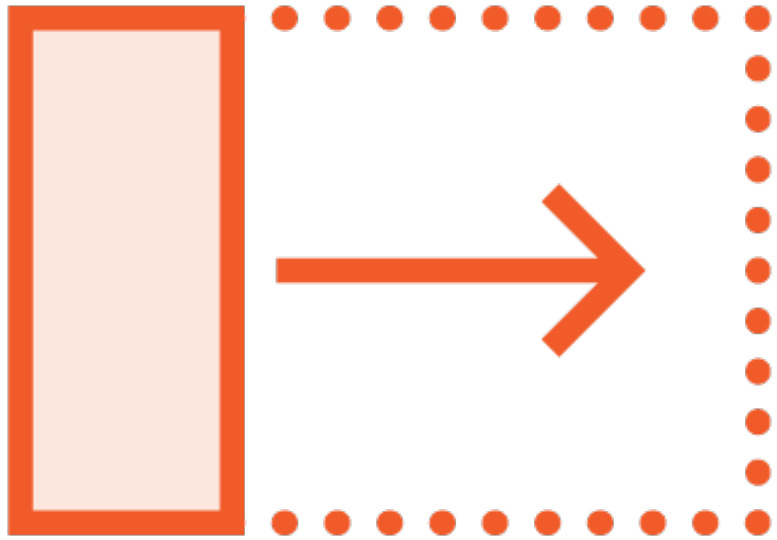
```
return StringHandler.InsertSpaces(productName);
```

Instances => Not Static

Joe Smith
Joe@aol.com
123 Main St.

Jessica Jones
Jessica@aol.com
123 First St.

Extension Method



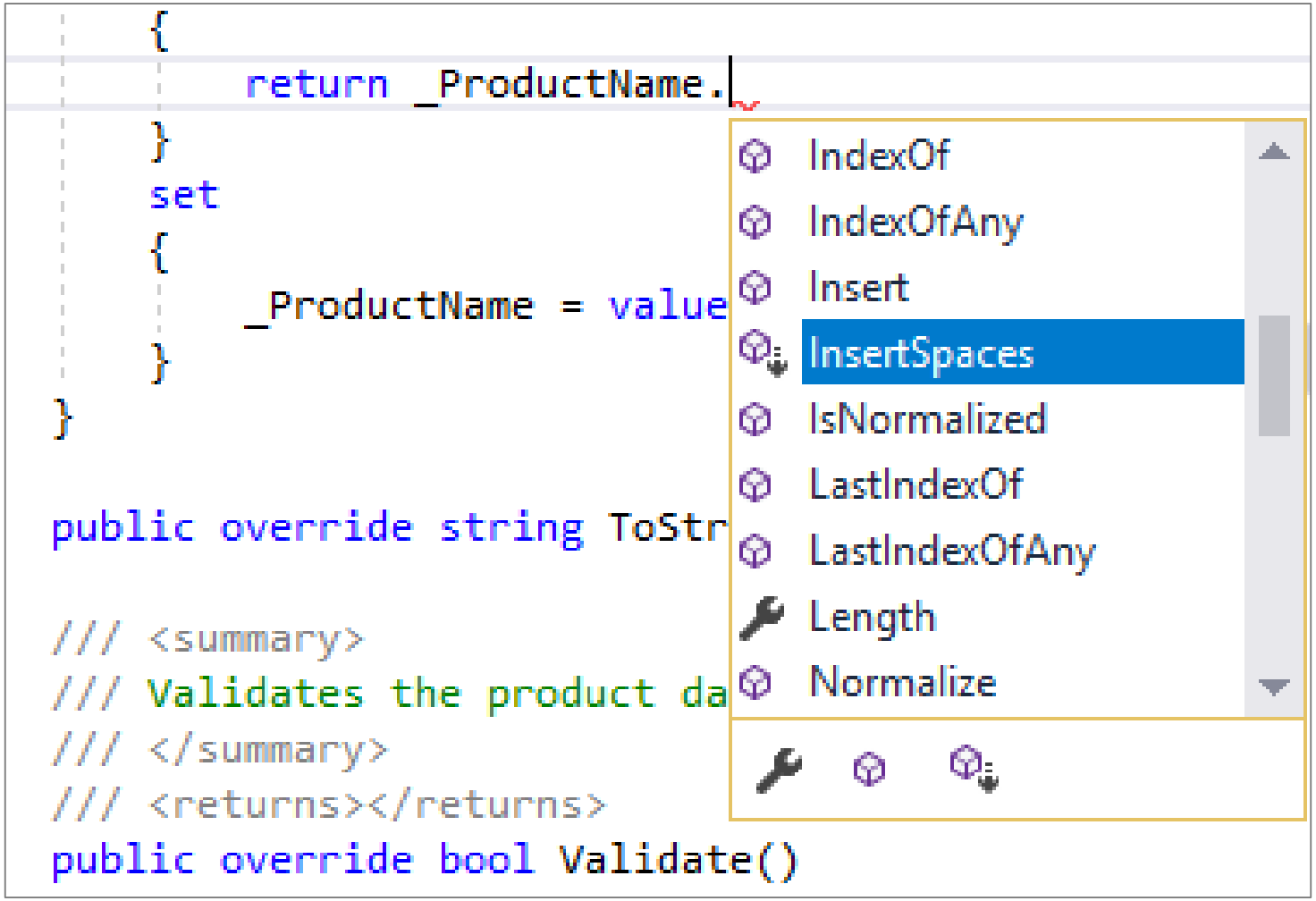
Add methods to an existing type without modifying the original type

Great for adding methods to .NET types

Extension methods appear in IntelliSense

Must be a static method in a static class

Extension Method



Extension Method

Creating the method

```
public static class StringHandler
{
  public static string InsertSpaces(string source)
  { ...
  }
}
```

Using the method

```
return StringHandler.InsertSpaces(productName);
```

Creating the method

```
public static class StringHandler
{
  public static string InsertSpaces(this string source)
  { ...
  }
}
```

Using the method

```
return _productName.InsertSpaces();
```

Static vs. Extension Method

```
public static string InsertSpaces(this string source)
```

Is the primary parameter an instance?

Does the method logically operate on that instance?

Is it desirable for the method to appear in IntelliSense for that type?

Build Reusable Components

Create a separate project for each reusable component

Build a library of general-purpose methods in the component, grouped into classes

Reuse the component

Static Class

Cannot be instantiated

Is sealed

Defined with the `static` keyword

```
public static class StringHandler
{

}
```

Use to organize utility methods

Static Method

Every method in a static class must be static

Define with the `static` keyword

```
public static string InsertSpaces(string source)
```

Access a static member with the class name

```
return StringHandler.InsertSpaces(productName);
```

Use to create reusable utility methods

Extension Method

Adds a method to an existing type without modifying the original type

Method must be static

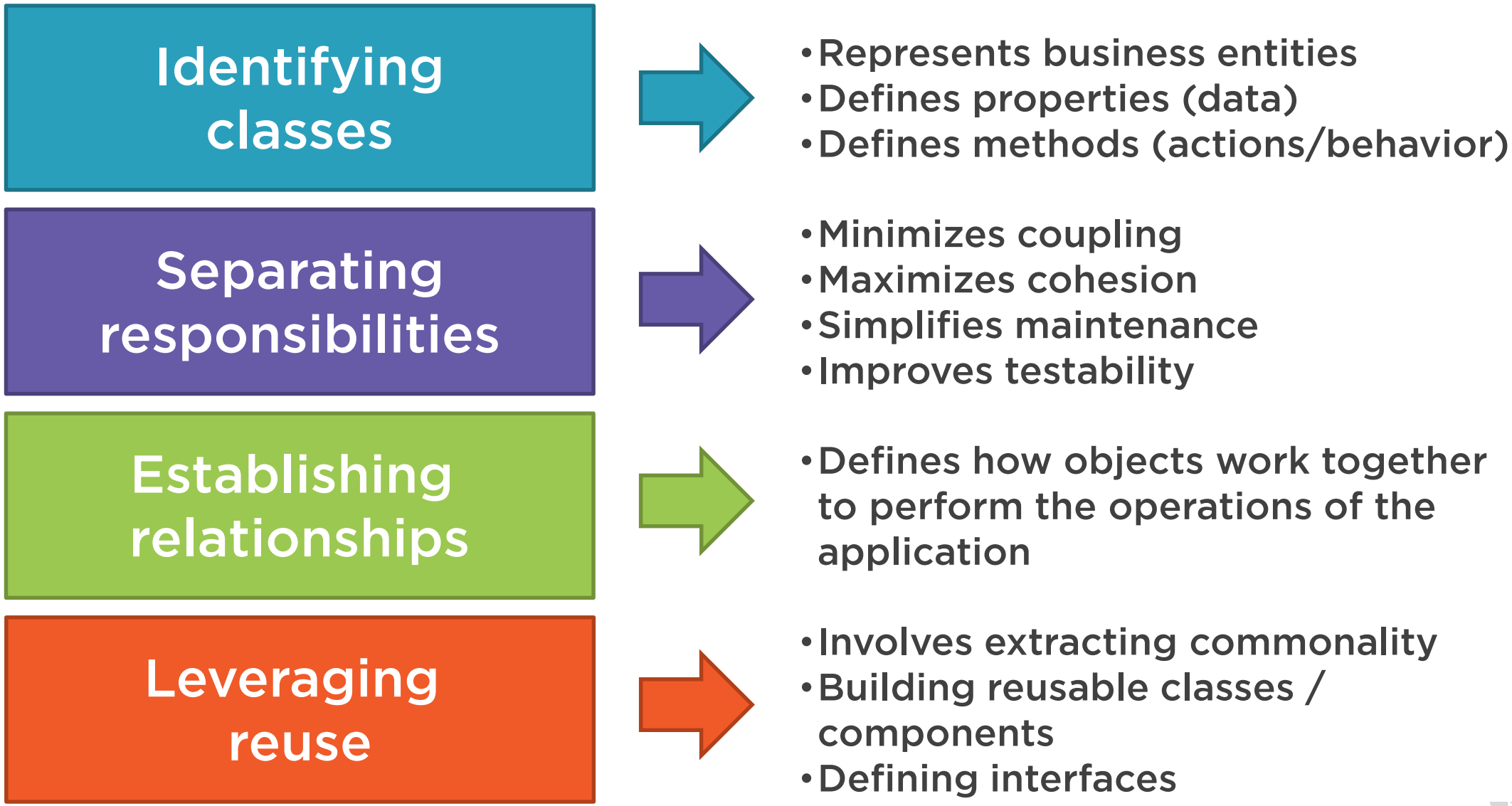
Define by adding `this` to the first parameter

```
public static string InsertSpaces(this string source)
```

Access an extension method using the extended class instance

```
return _productName.InsertSpaces();
```

Object-Oriented Programming (OOP)



Understanding Interfaces

Deborah Kurata
CONSULTANT | SPEAKER | AUTHOR | MVP | GDE

@deborahkurata | blogs.msmvps.com/deborahk/

"In computing, an **interface** is a shared boundary across which two or more separate components of a computer system exchange information.

The exchange can be between software, computer hardware, peripheral devices, humans and combinations of these."

- Wikipedia 1/9/19

User Interface

Web API

Class Interface

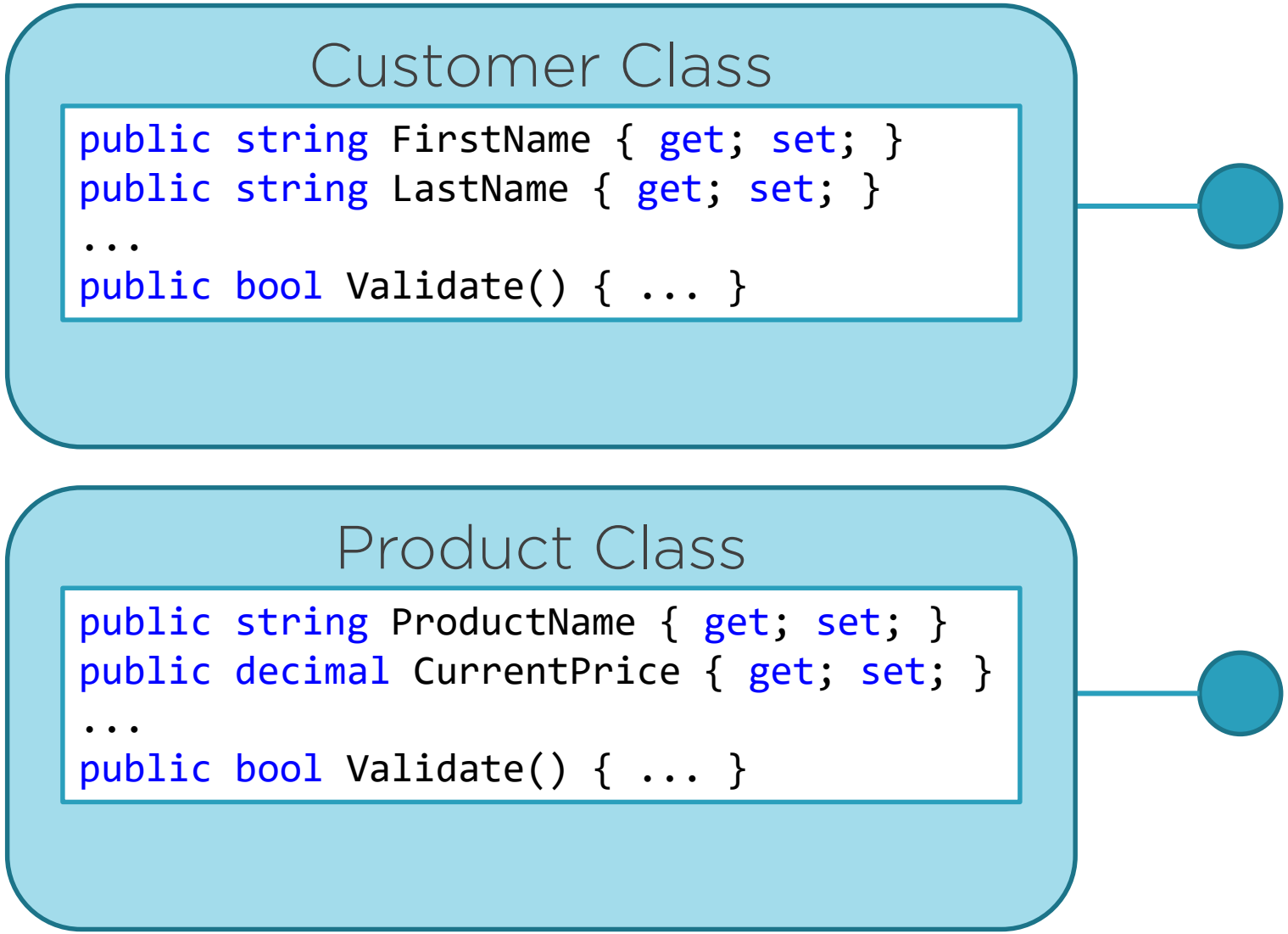
Module Outline

- Class interface
- Defining an interface
- Implementing an interface
- Interface-based polymorphism

Class Interface

```
public class Customer : EntityBase
{
    public List<Address> AddressList { get; set; }

    public int CustomerType { get; set; }

    public static int InstanceCount { get; set; }

    public string LastName...

    public string FirstName { get; set; }

    public string EmailAddress { get; set; }

    public int CustomerId { get; private set; }

    public string FullName...

    public override bool Validate()...

    public override string ToString() => FullName;
}
```

Class Interface



Interface

Customer Class

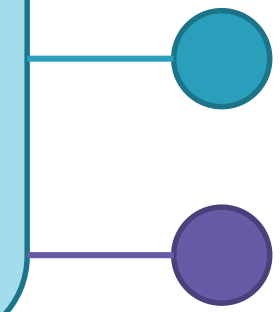
```
public string FirstName { get; set; }
public string LastName { get; set; }
...
public bool Validate() { ... }
```

```
public string Log() { ... }
```

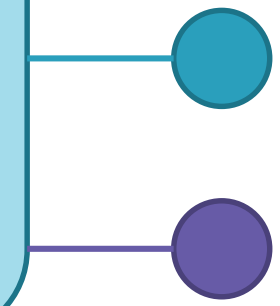
Product Class

```
public string ProductName { get; set; }
public decimal CurrentPrice { get; set; }
...
public bool Validate() { ... }
```

```
public string Log() { ... }
```



```
public interface ILoggable
{
    string Log();
}
```



Interface

Customer Class

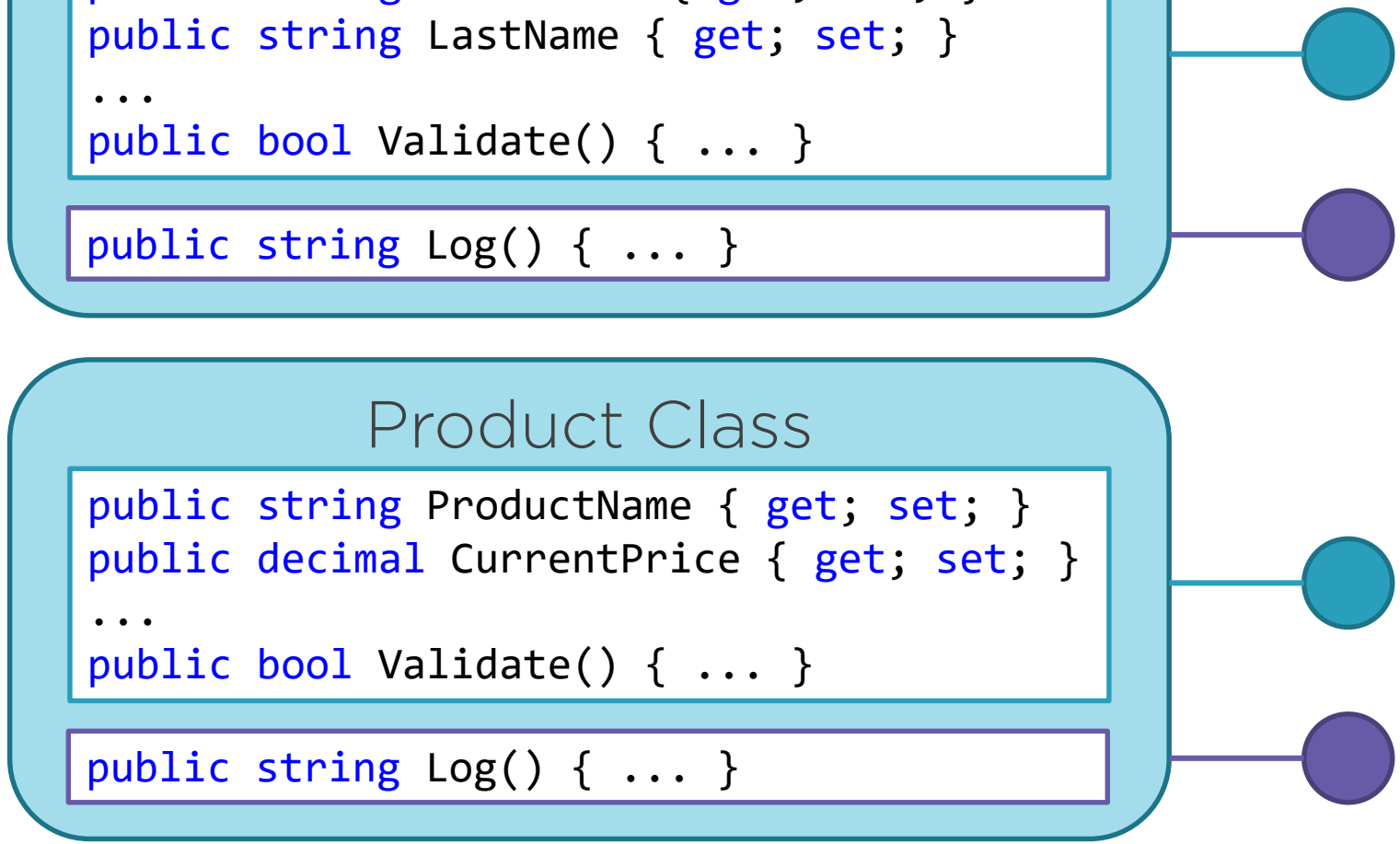
```
public string FirstName { get; set; }
public string LastName { get; set; }
...
public bool Validate() { ... }
```

```
public string Log() { ... }
```

Product Class

```
public string ProductName { get; set; }
public decimal CurrentPrice { get; set; }
...
public bool Validate() { ... }
```

```
public string Log() { ... }
```



ACM Application

- ACM.WPF
- ACM.Web
- ACM.BL
- ACM.DL
- Acme.Common

Interfaces Define "Roles"

Default person Interface

IParent

IPartner

IEmployee

Customer Class

```
public class Customer : ILoggable
{
  public string FirstName { get; set; }
  ...
  public string Log() { }
}
```

Class Interface

ILoggable

Interfaces are a "Contract"

```
public interface ILoggable
{
    string Log();
}
```

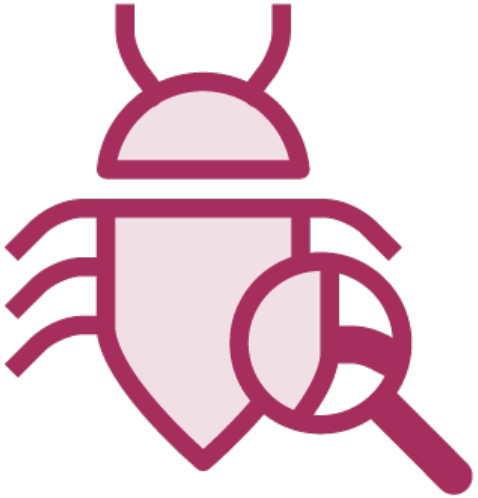
Customer Class

```
public class Customer : ILoggable
{
  public string FirstName { get; set; }
  ...
  public string Log() { ... }
}
```

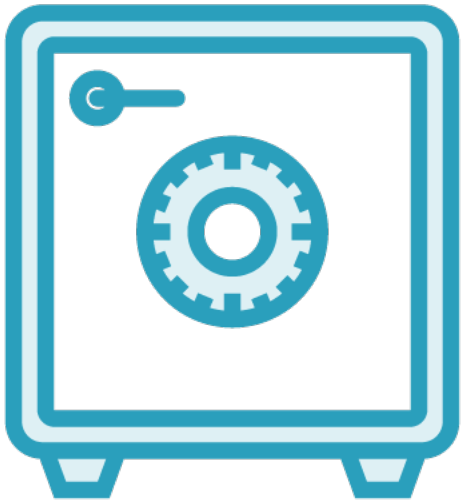
Class Interface

ILoggable

Logging



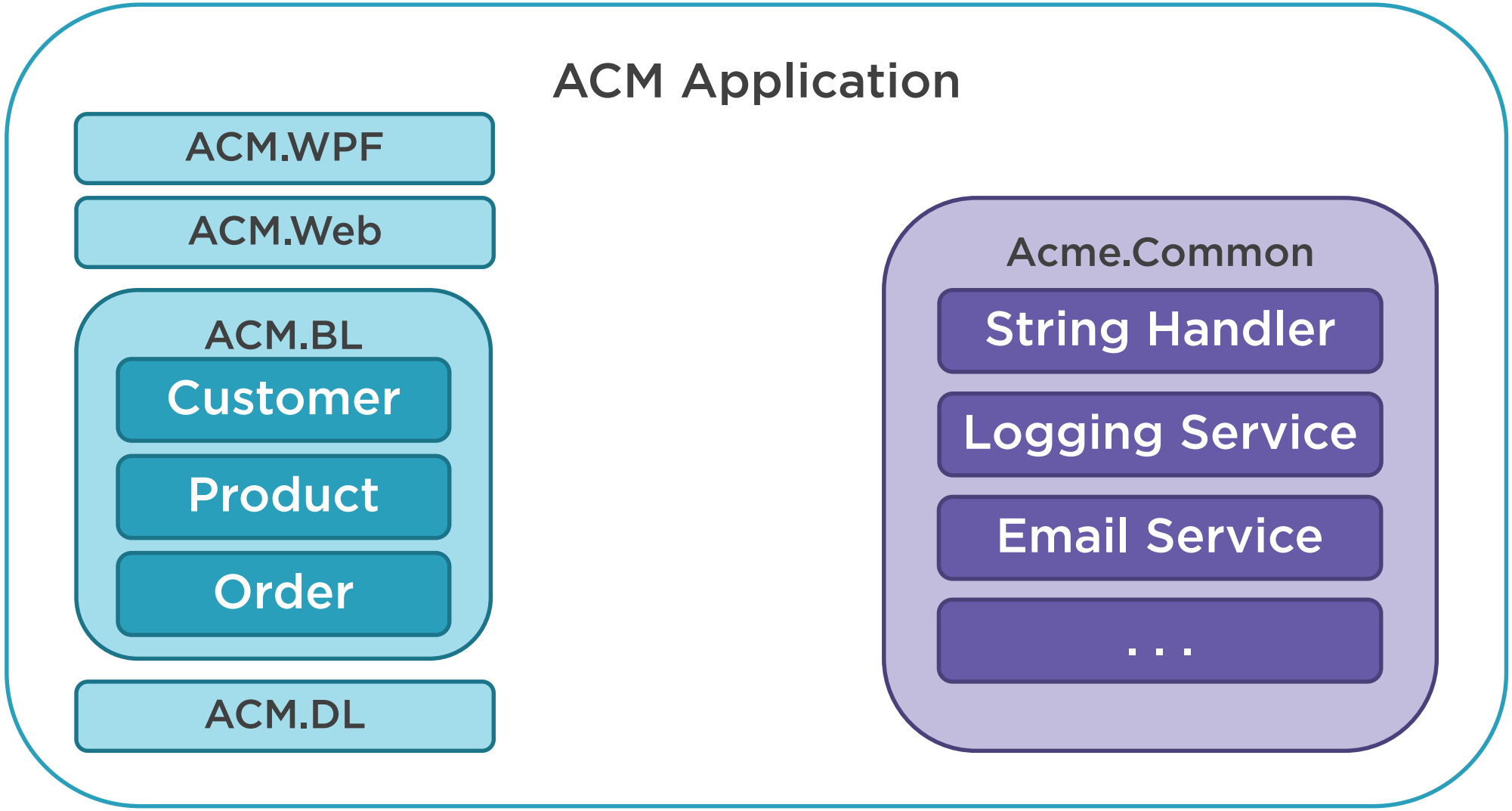
Resolving bugs



Security

Data analysis

Reusable Library Component



Interface

```
public interface ILoggable
{
    string Log();
}
```

```
public interface IEmailable
{
    bool CopyToSender { get; set; }

    string Send(string sendTo, string[] ccTo);
}
```

Defining an Interface

Add a new interface item to a project

Prefix the interface name with "I"

Specify the `public` access modifier

Add property, method, event, or indexer signatures

No need for an access modifier on the members

No implementation of the members

Implementing an Interface

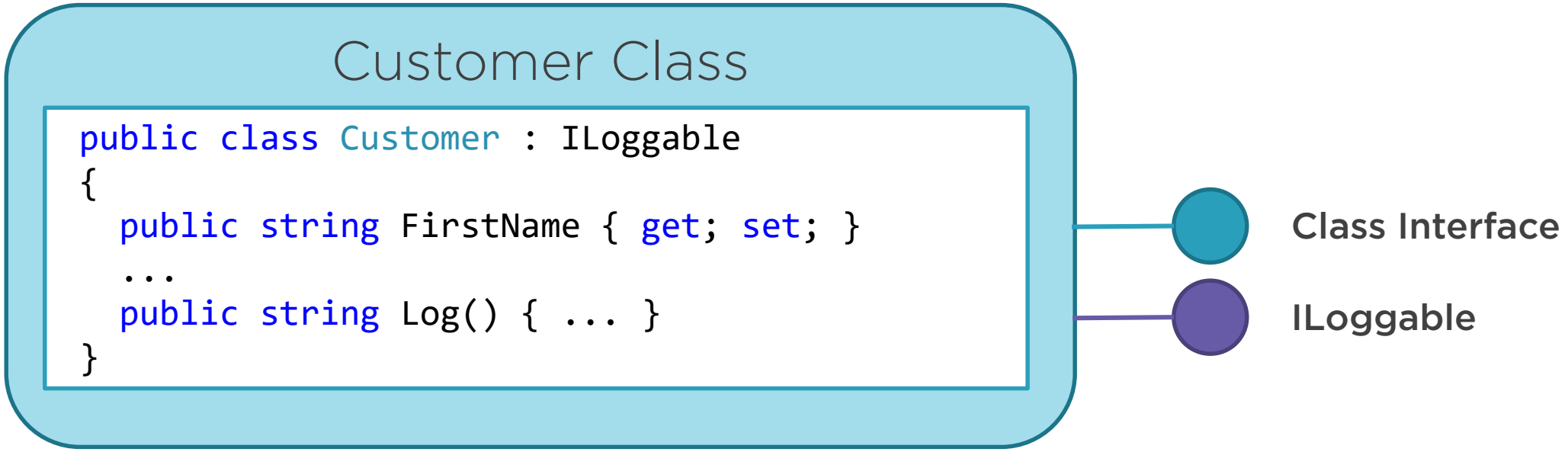
Add the interface to the class signature

```
public class Customer : ILoggable, IEmailable
```

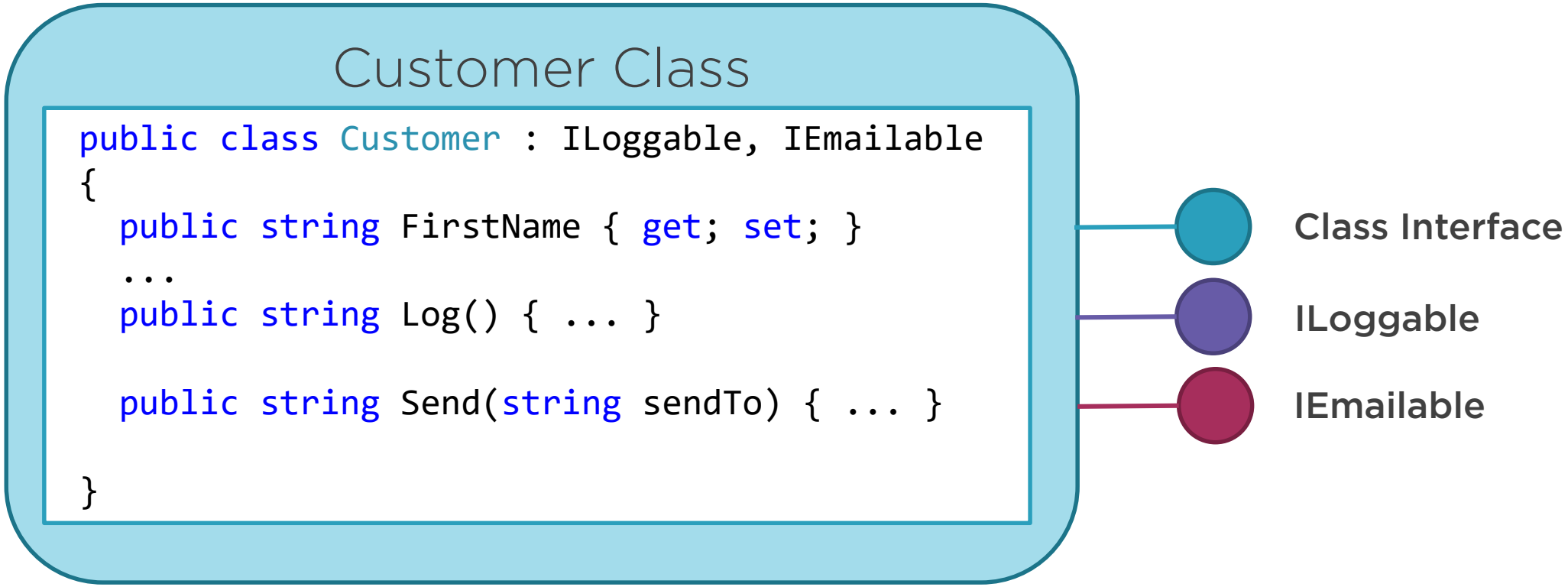
Implement each member of the interface

```
public string Log() { ... }
```

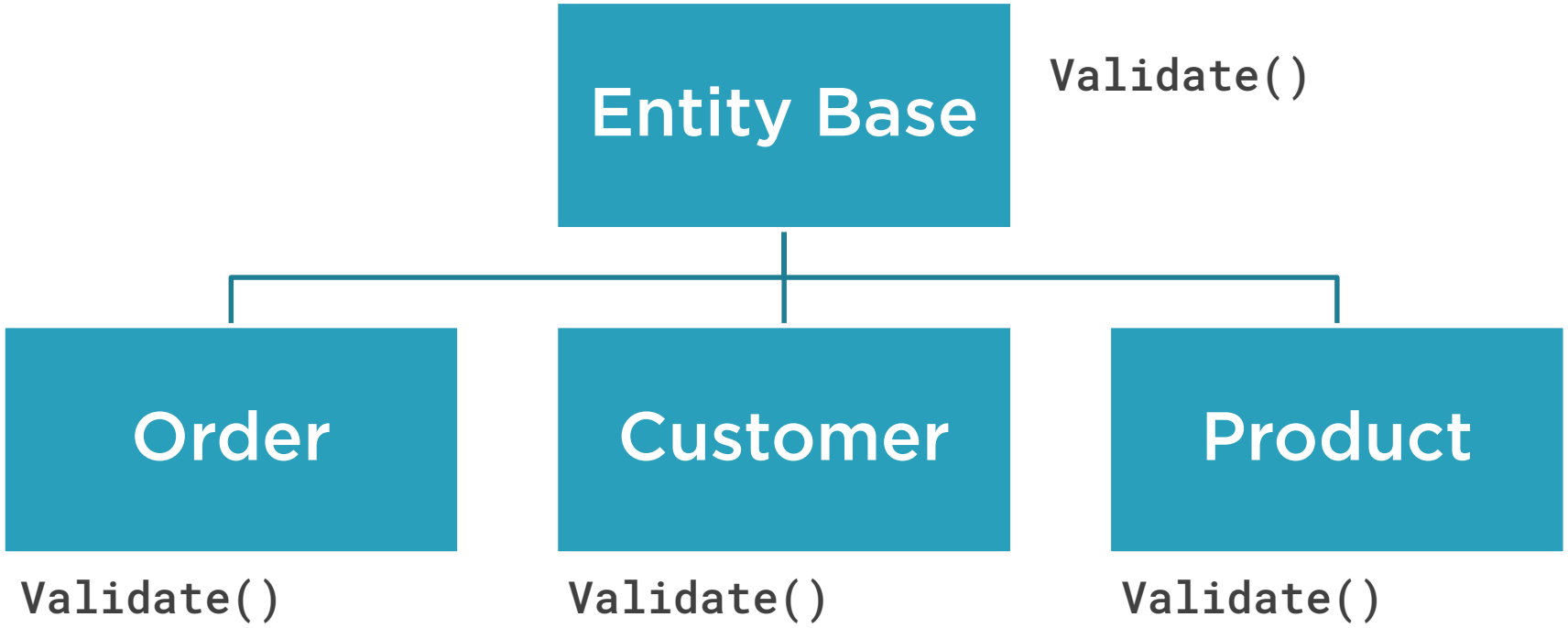
Implementing an Interface



Implementing an Interface



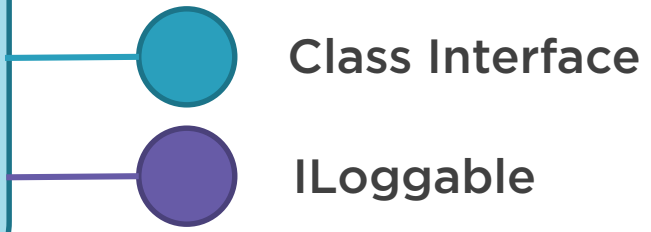
Inheritance-based Polymorphism



Interface-based Polymorphism

Customer Class

```
public class Customer : ILoggable
{
  public string FirstName { get; set; }
  ...
  public string Log() { ... }
}
```

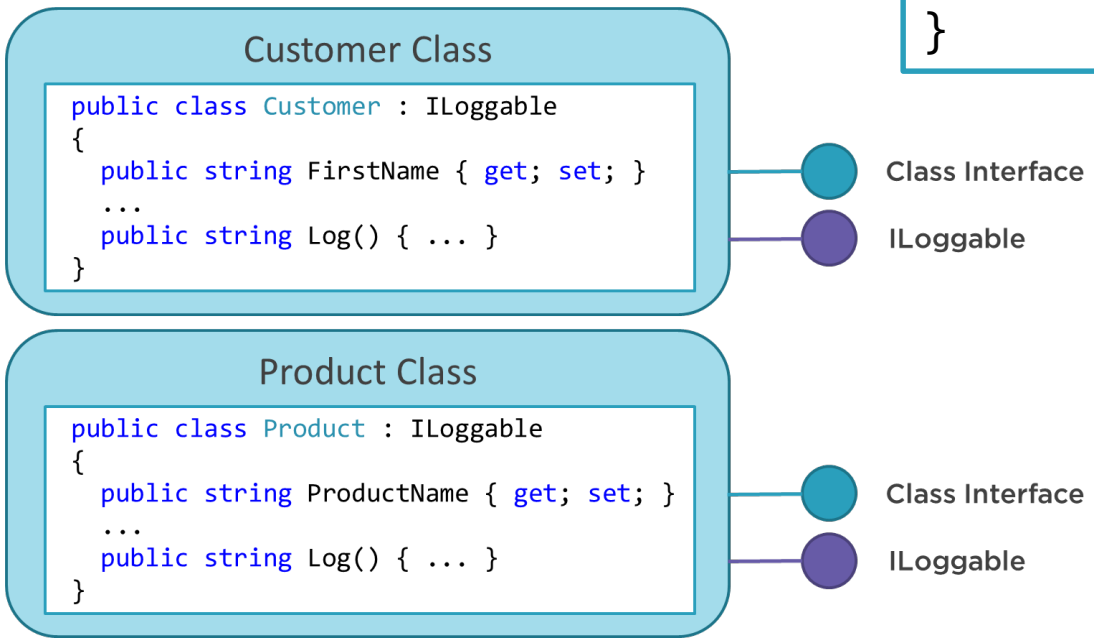


Product Class

```
public class Product : ILoggable
{
  public string ProductName { get; set; }
  ...
  public string Log() { ... }
}
```

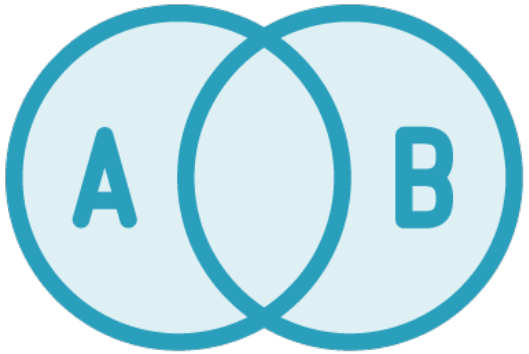
Interface-based Polymorphism

```
public void WriteToFile(List<ILoggable> itemsToLog)
{
    foreach (var item in itemsToLog)
    {
        Console.WriteLine(item.Log());
    }
}
```

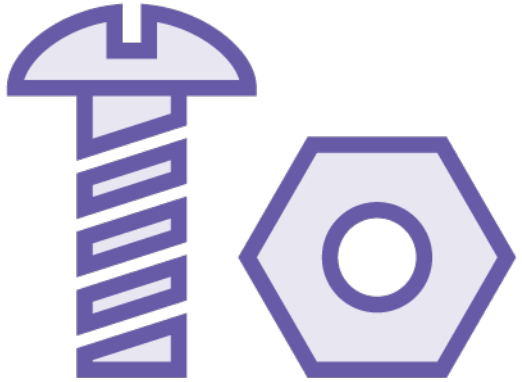


Benefits of Interfaces and Polymorphism

Strong typing



Defines commonality among unrelated classes



Aids in building generalized utility methods with class-unique functionality

Interfaces

Define a role that an object can play

Define a contract

An interface is comprised of a list of properties, methods, events and iterators

- Denoting the data and operations an object can perform

Class Interface

Every class has a basic interface

Defined by the public properties and methods of the class

The application uses this interface to work with the class in its primary role

Interface

Define any number of additional interfaces

Define an interface using the `interface` keyword

```
public interface ILoggable
{
    string Log();
}
```

Specify the signatures of the interface members with no code

Implementing an Interface

Any class can implement an interface

Implement an interface by adding it to the class signature after a colon

```
public class Customer : ILoggable, IEmailable
```

The class must then implement every property and method defined in that interface

Interface-based Polymorphism

Many shapes

One method name specified in an interface, multiple implementations of that method

One implementation in each class that implements the interface

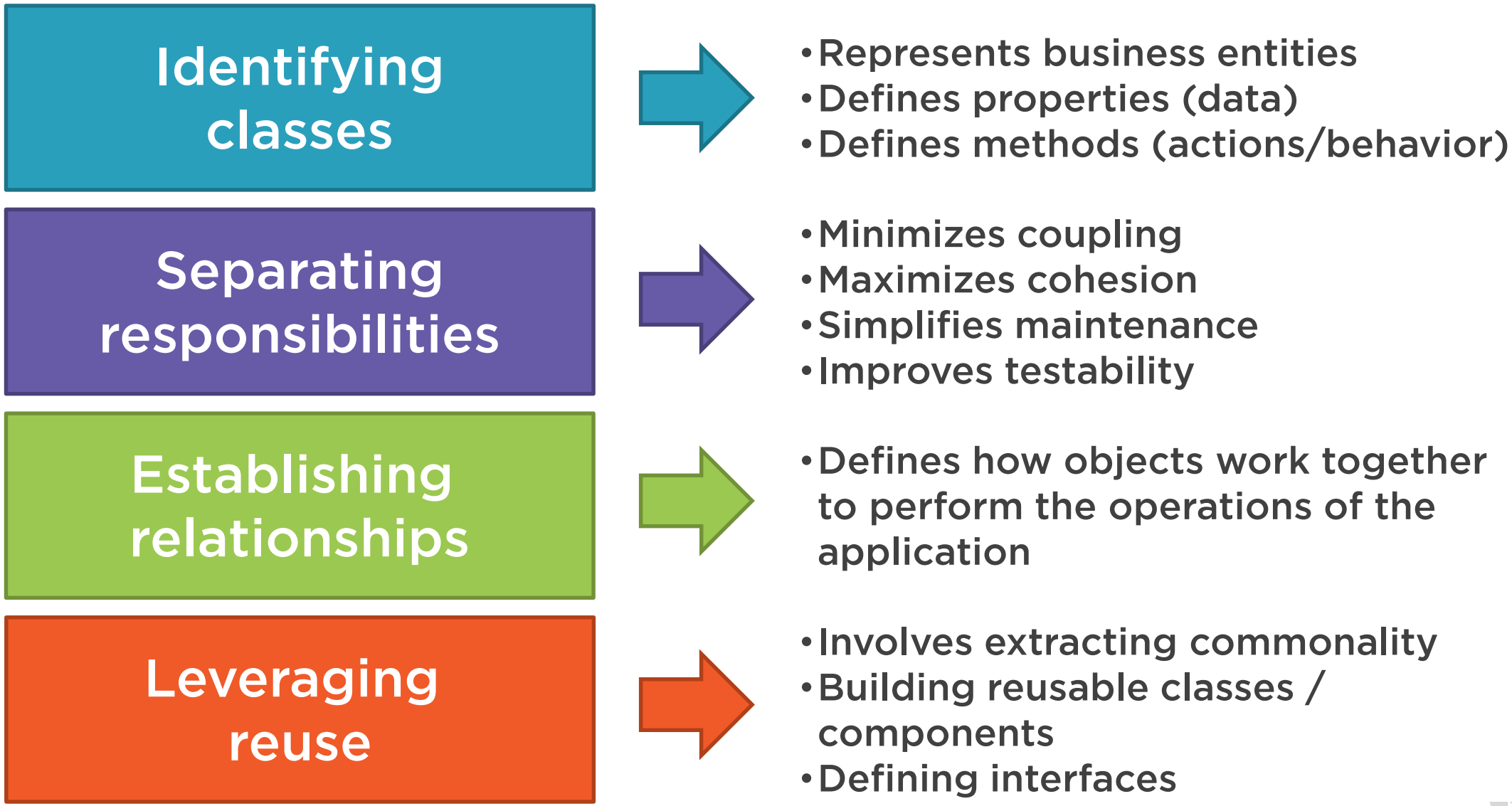
Allows us to work with otherwise unrelated classes in a generalized, reusable way

Final Words and Next Steps

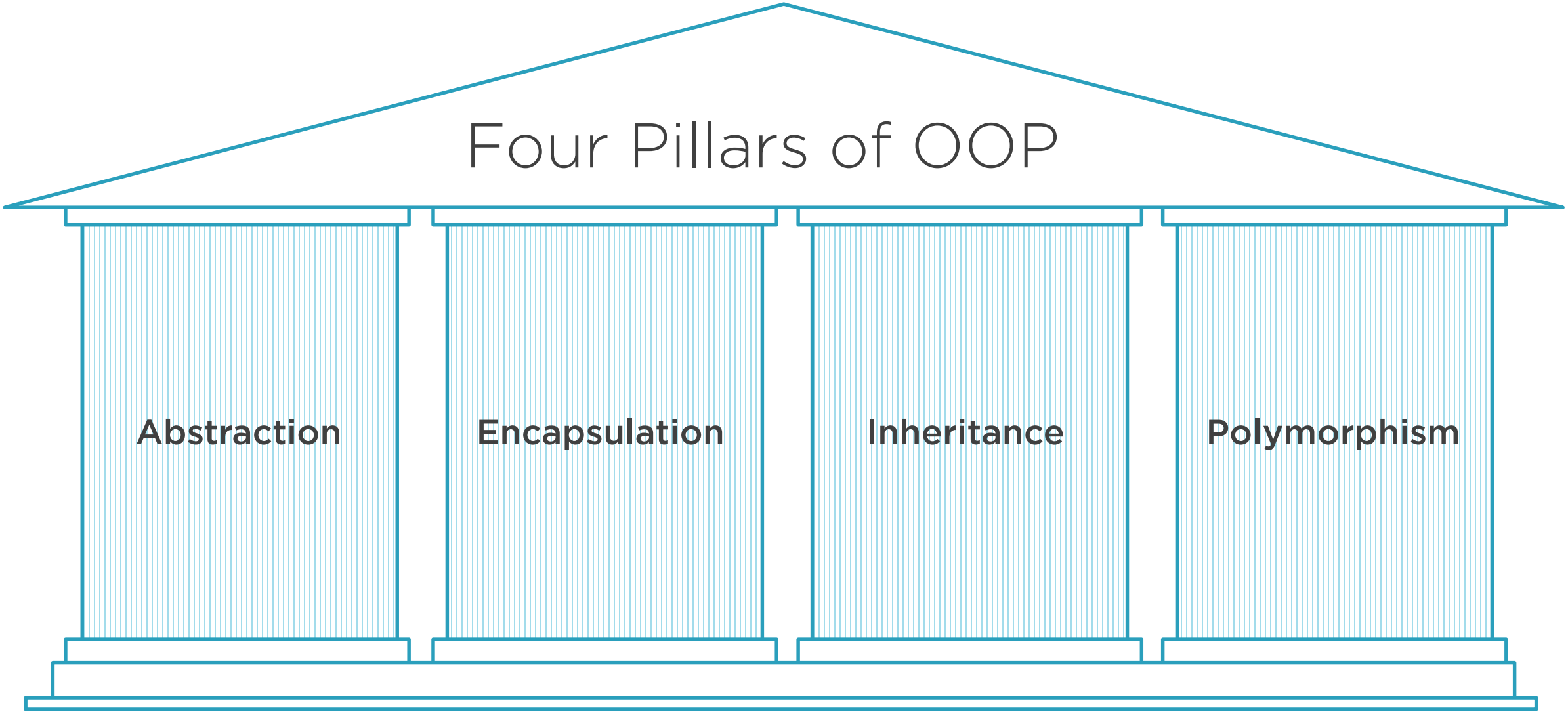
Deborah Kurata
CONSULTANT | SPEAKER | AUTHOR | MVP | GDE

@deborahkurata | blogs.msmvps.com/deborahk/

Object-Oriented Programming (OOP)



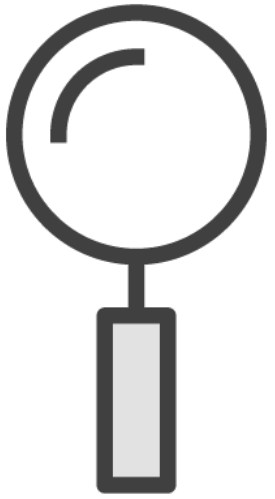
Four Pillars of OOP



Abstraction

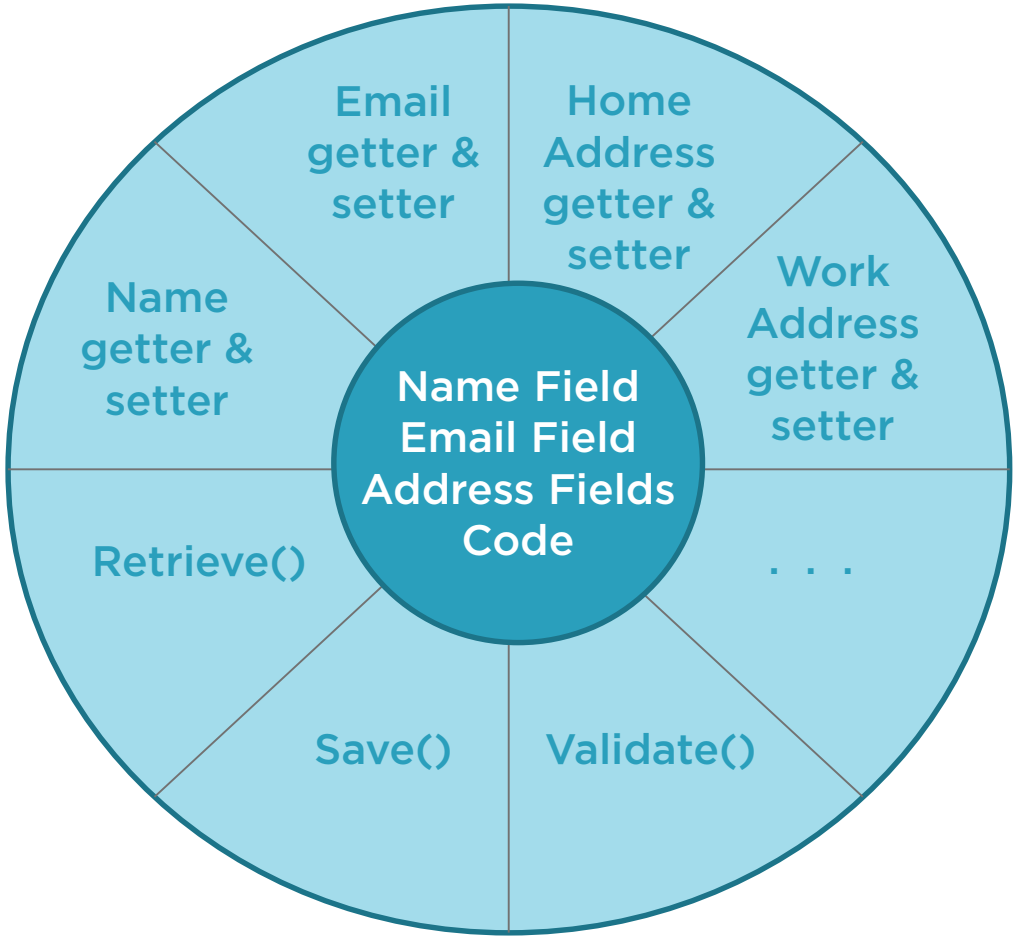
Simplifying reality

Ignoring extraneous details



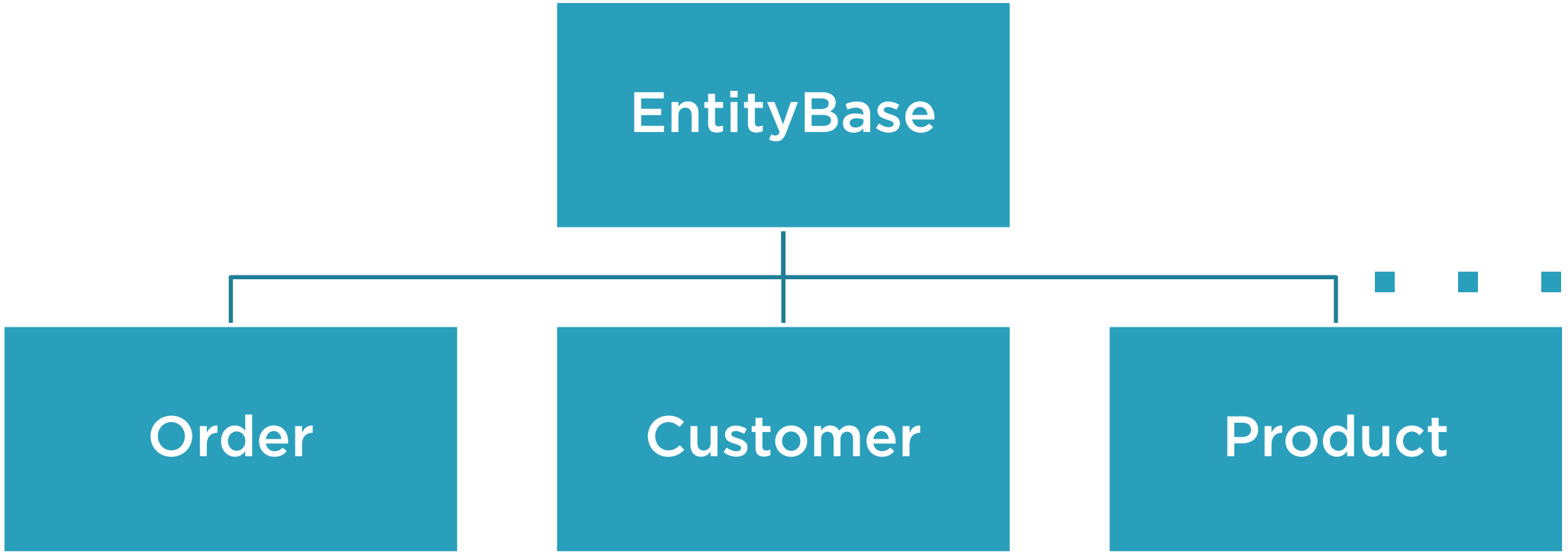
Focusing on what is important for a purpose

Encapsulation

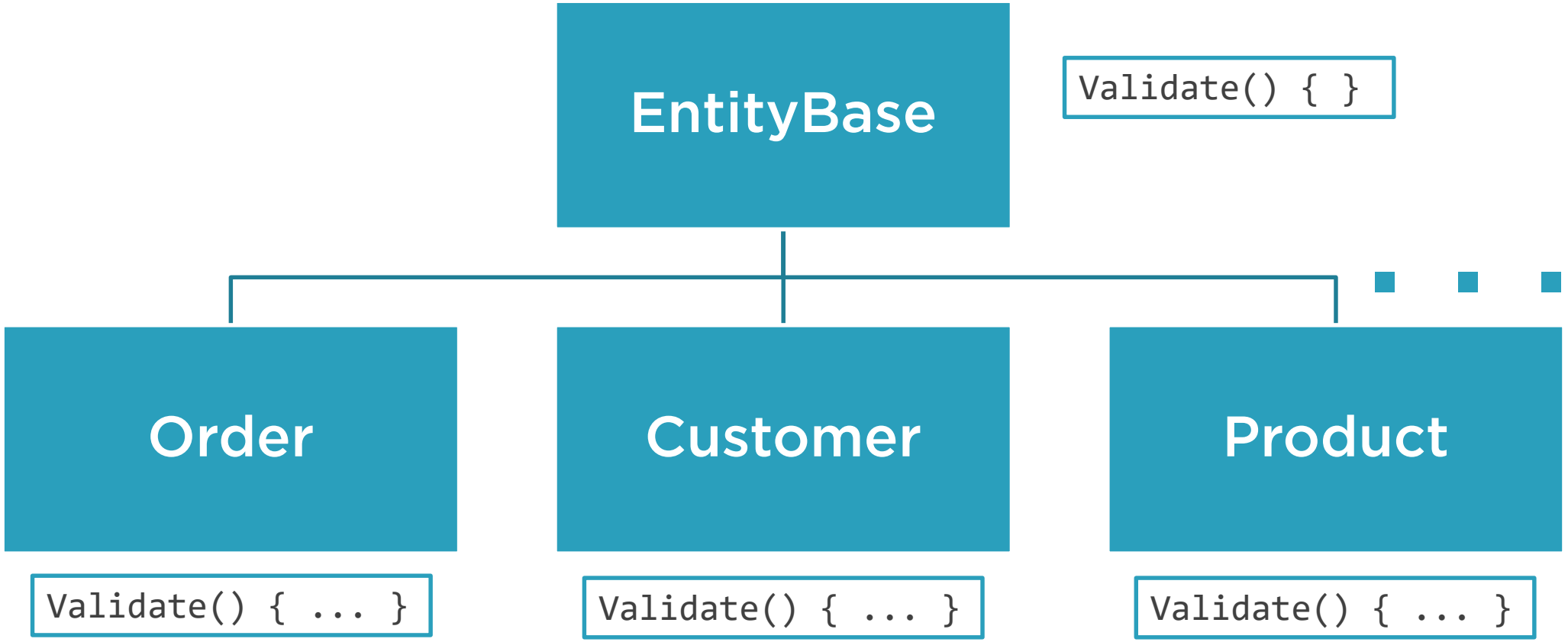


Customer Class

Inheritance

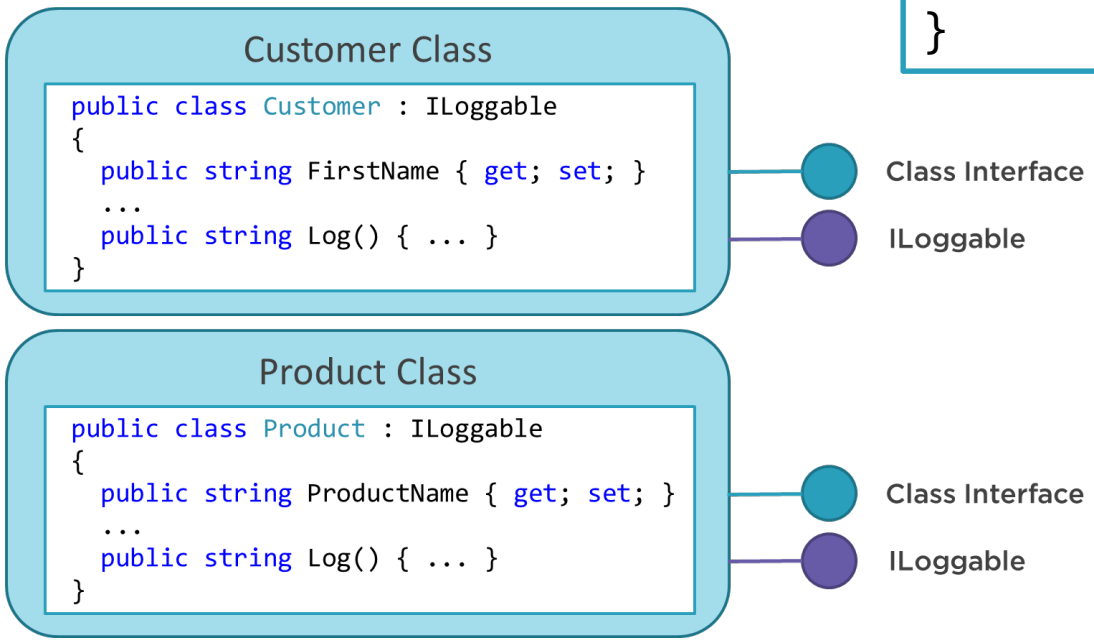


Inheritance-based Polymorphism

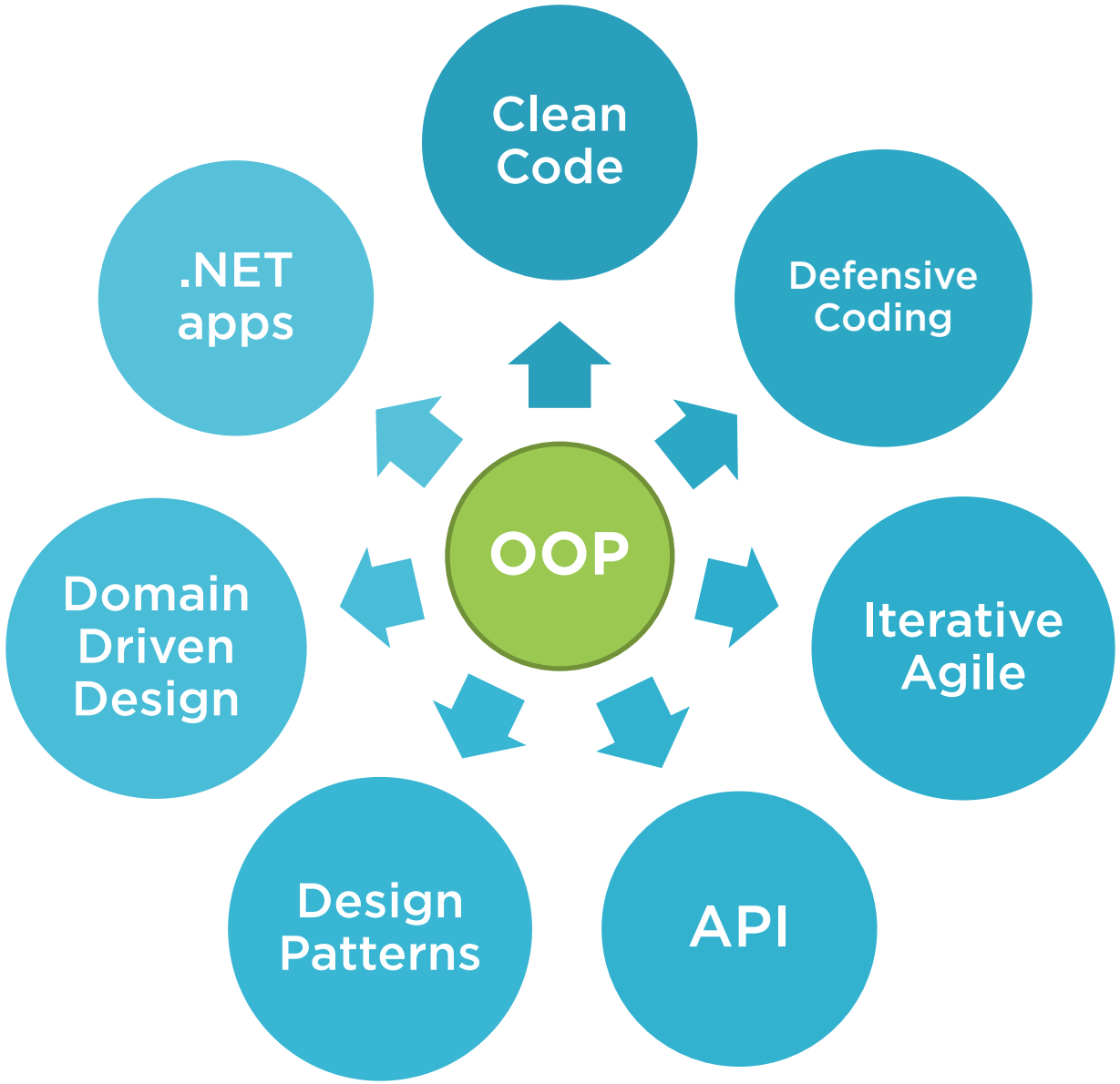


Interface-based Polymorphism

```
public void WriteToFile(List<ILoggable> itemsToLog)
{
    foreach (var item in itemsToLog)
    {
        Console.WriteLine(item.Log());
    }
}
```



OOP Is the Foundation



Next Steps

Additional Pluralsight Courses

- Defensive Coding in C#
- Clean Code: Writing Code for Humans
- C# Interfaces
- Design Patterns On-Ramp
- C# Best Practices: Improving on the Basics
- C# Best Practices: Collections and Generics

Object-Oriented Programming (OOP)