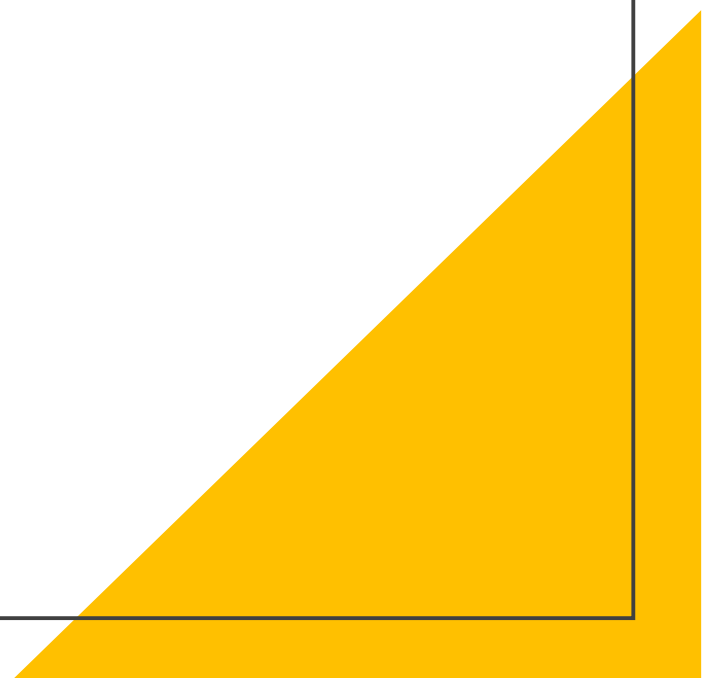
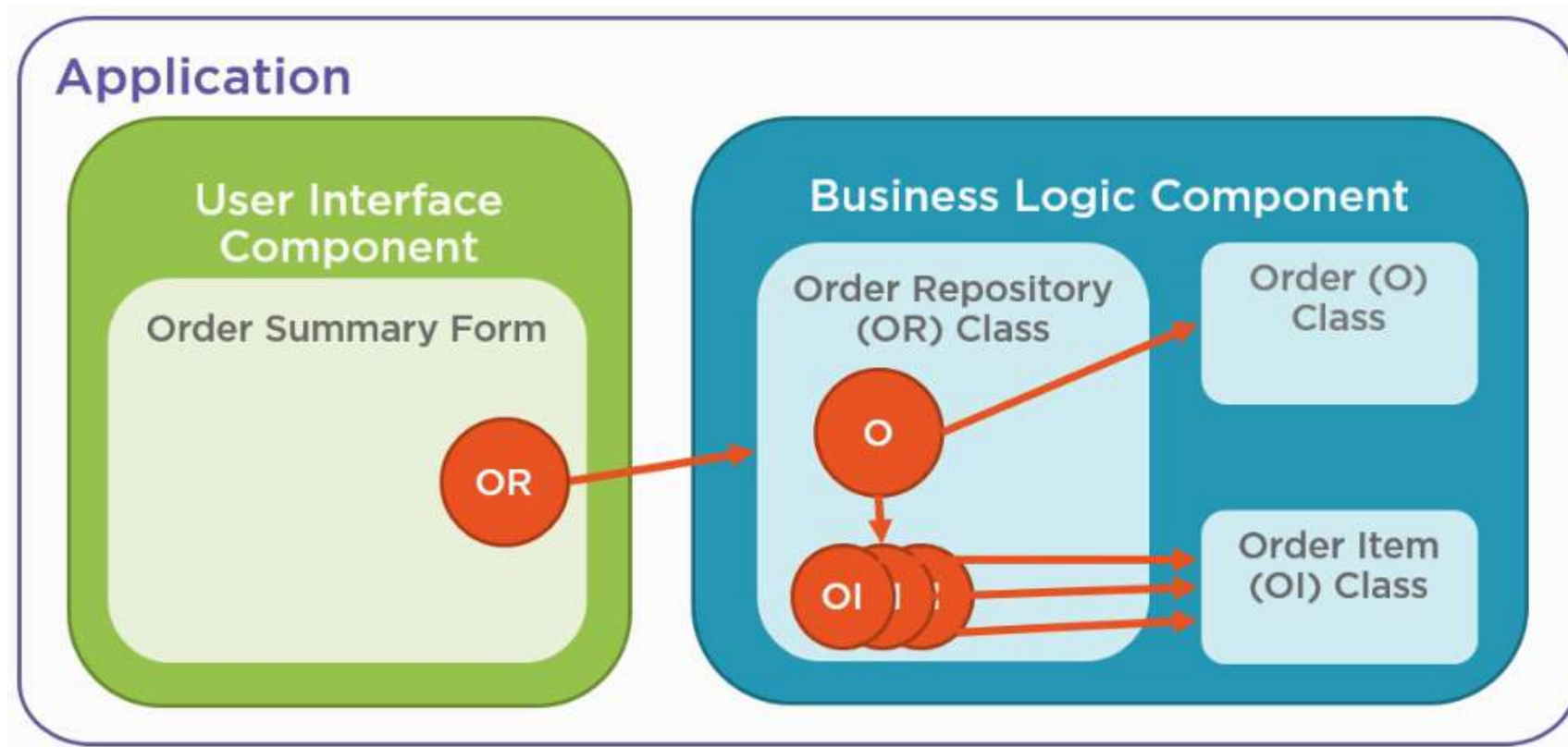


Programação

OO

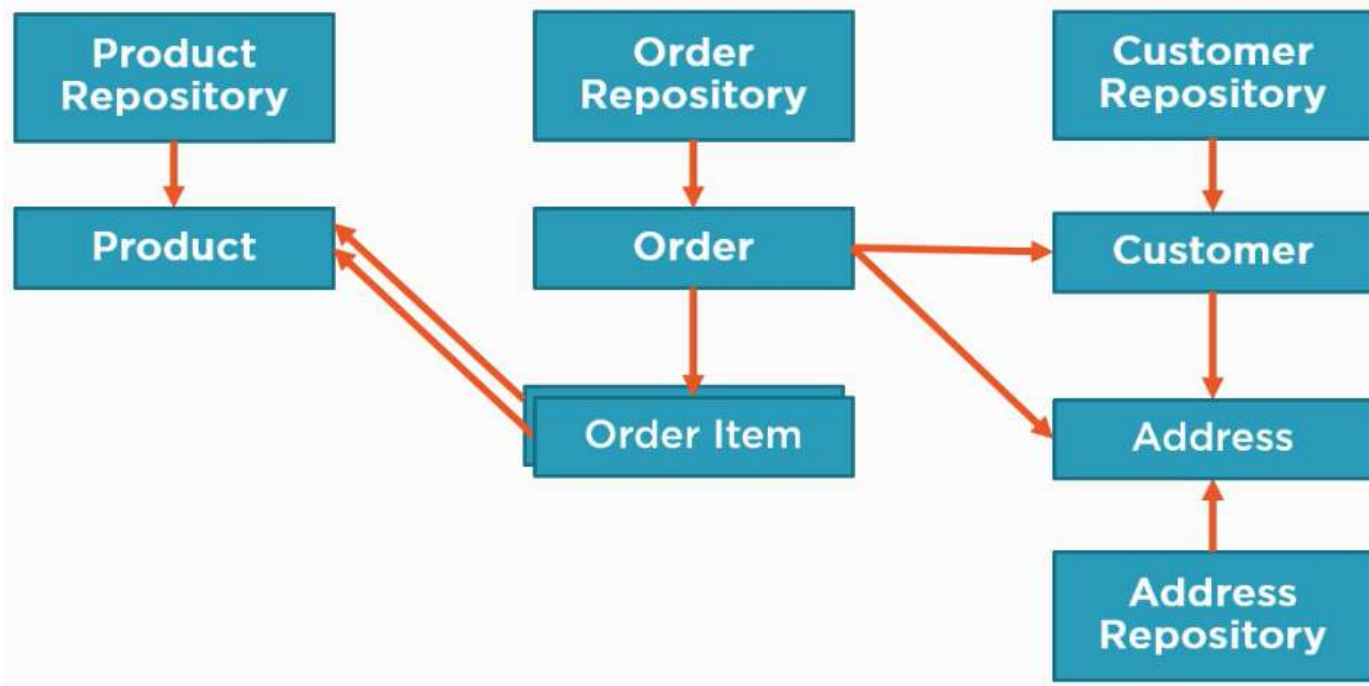
Estabelecendo Relacionamentos



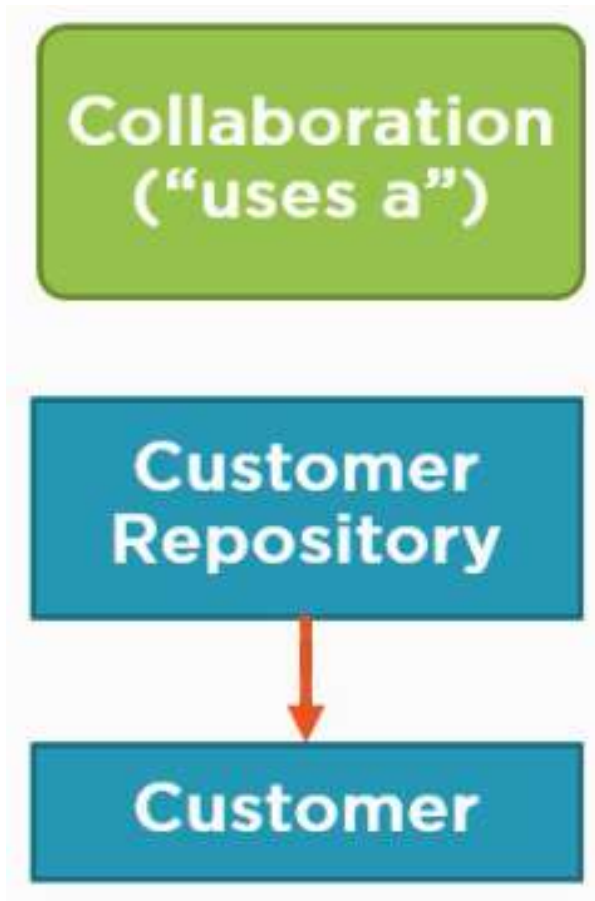


Trabalho em equipe

- O relacionamento define como o objeto interage e trabalha para realização as operações na aplicação;



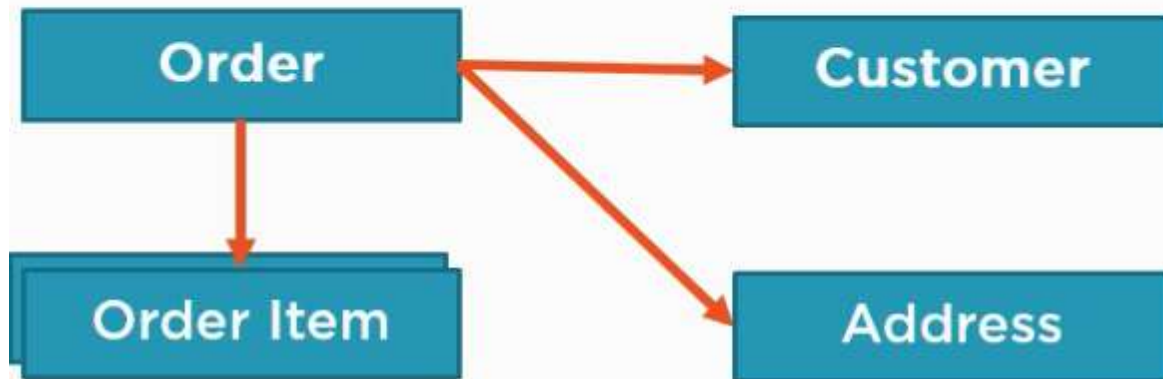
Definindo
Relacionamentos



Tipos de Relacionamento: Colaboração

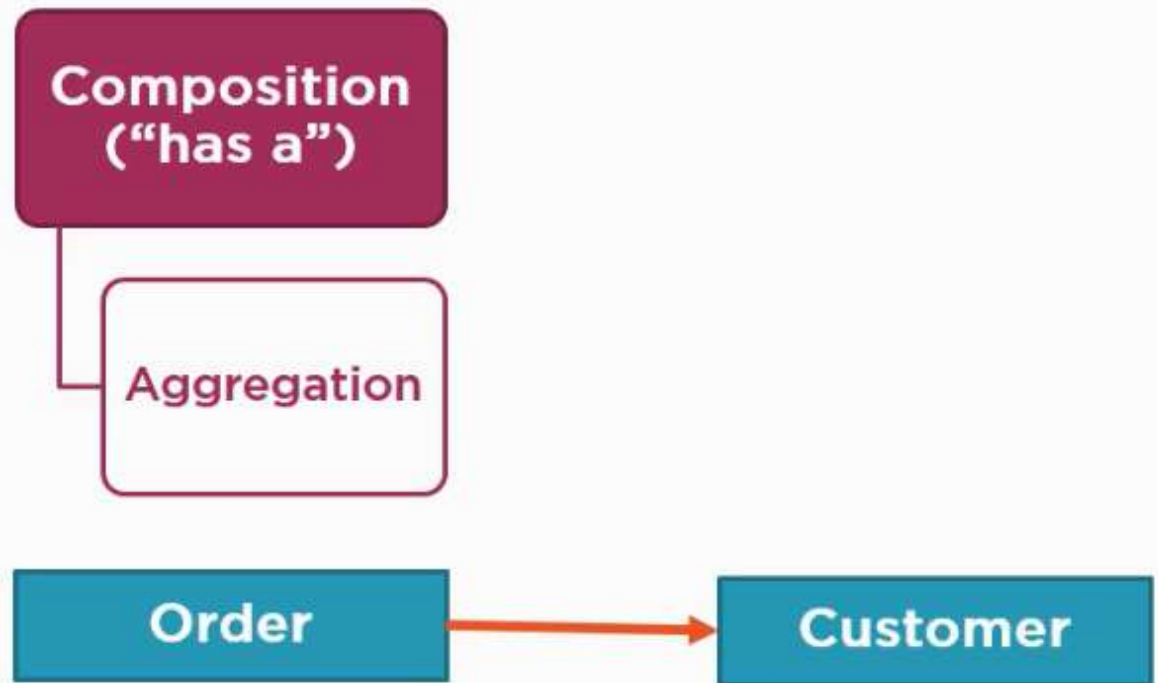
- "Usa Um" - Define um relacionamento em que um objeto colabora com, ou utiliza algum outro objeto que não está necessariamente relacionado;

Composition ("has a")



Tipos de Relacionamento: Composição

"Tem Um" - Define o relacionamento em que um objeto é composto por outro objeto.

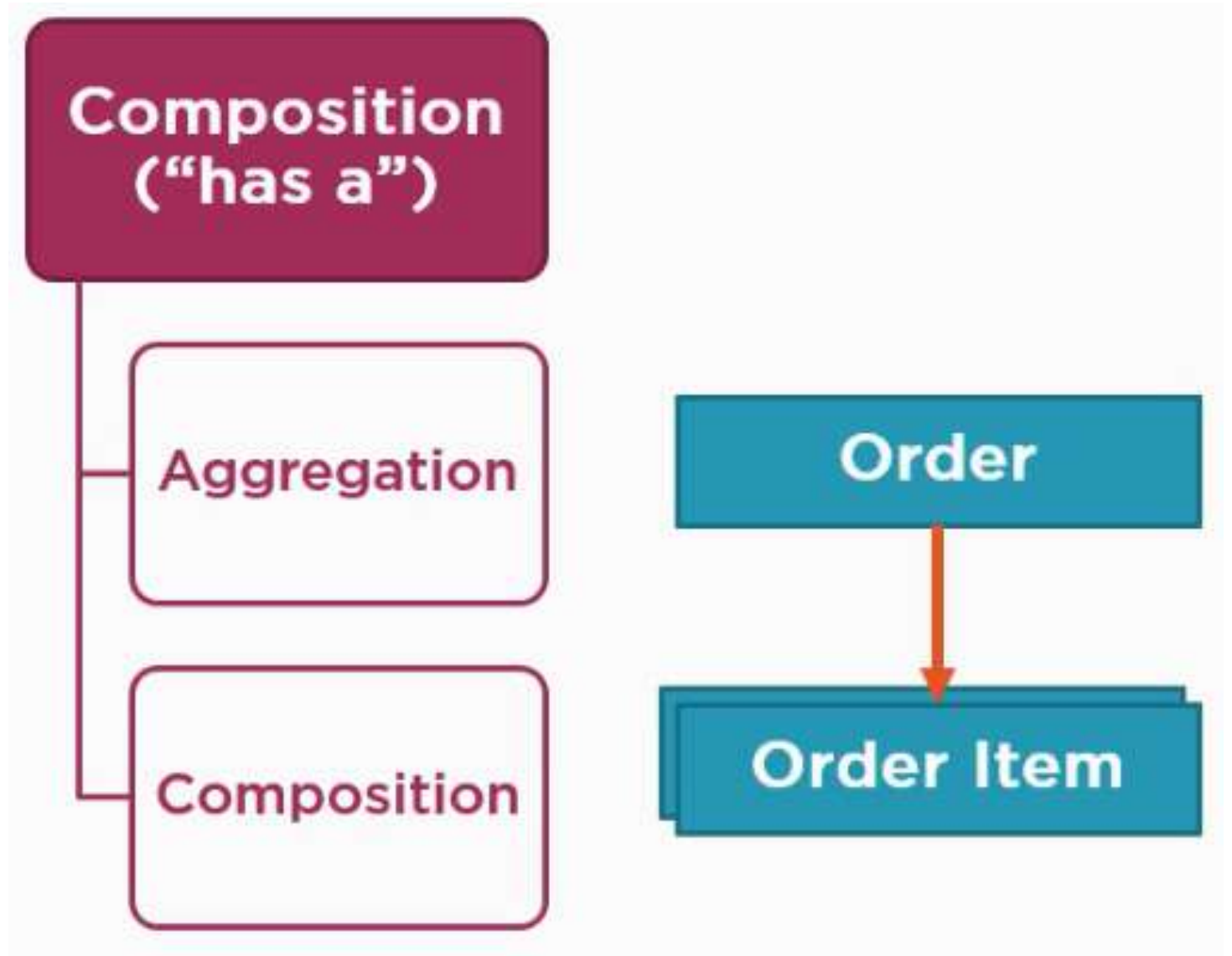


Tipos de Relacionamento: Composição por Agregação

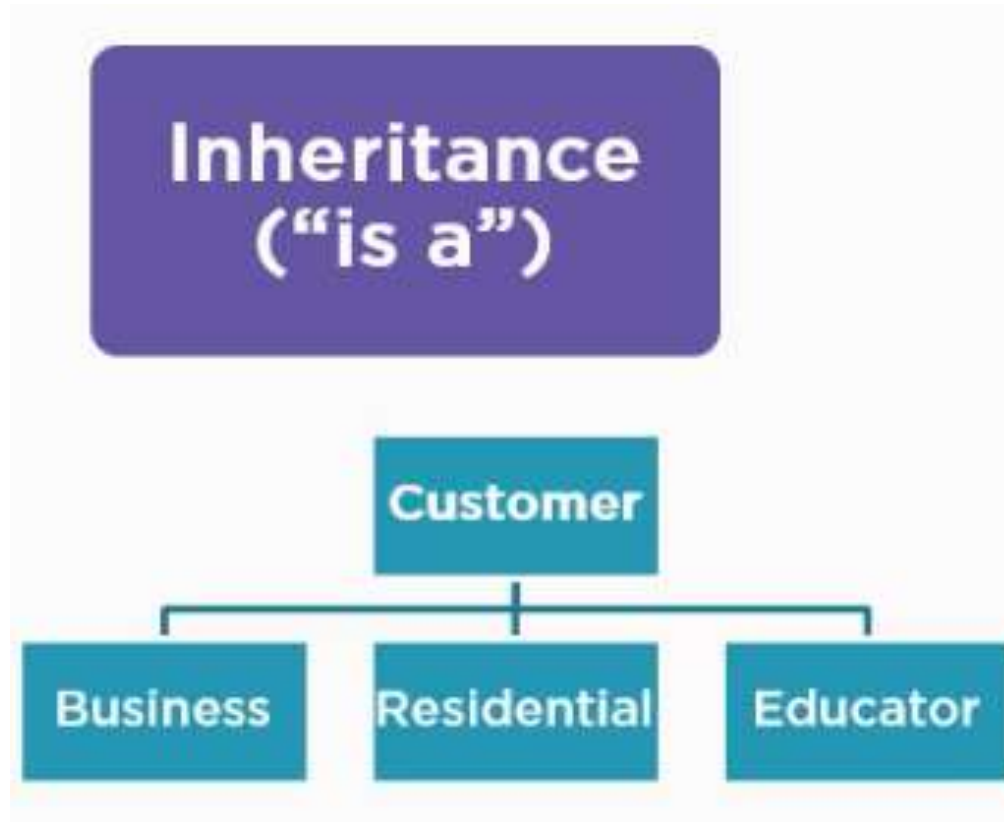
Quando um objeto é composto por múltiplos objetos que podem existir fora do relacionamento.

Tipos de Relacionamento: Composição ou por Composição

- Limita-se aos relacionamentos em que os objetos relacionados não existem sem a relação. O objeto apropria-se do relacionado e se ele for destruído, tudo que estiver relacionado a ele também será.



Tipos de Relacionamento: Herança



- "É Um" - O terceiro tipo de relacionamento, imagine que precisamos criar diferentes tipos de consumidor.

Colaboração: Usa Um

- CustomerRepository utiliza Customer para manipular os dados, o mesmo para OrderRepository e AddressRepository



CustomerRepository.cs

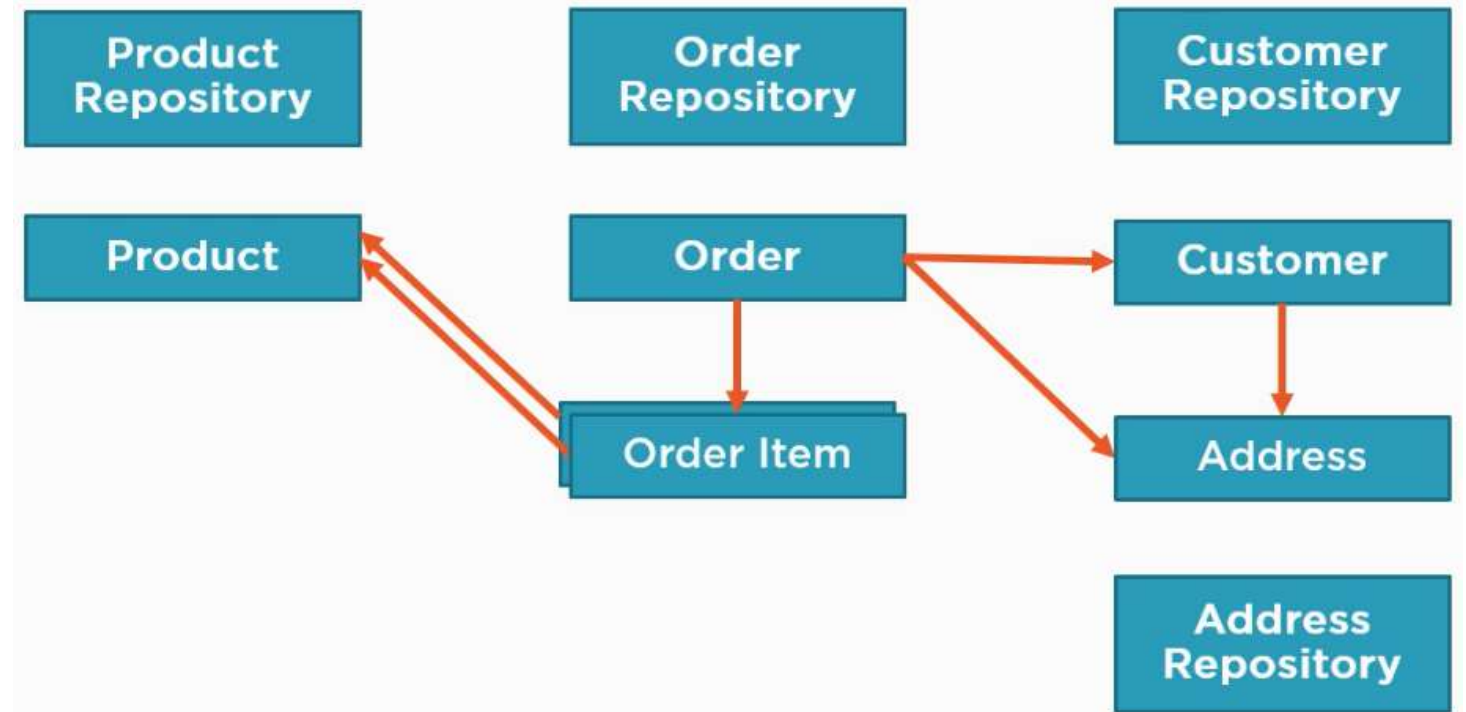
```
public Customer Retrieve(int customerId)
{
    // Create the instance of the Customer class
    // Pass in the requested id
    Customer customer = new Customer(customerId);

    // Code that retrieves the defined customer

    // Temporary hard-coded values to return
    // a populated customer
    if (customerId == 1)
    {
        customer.EmailAddress = "fbaggins@hobbiton.me";
        customer.FirstName = "Frodo";
        customer.LastName = "Baggins";
    }
    return customer;
}
```

Composição: Tem Um

- Existe quando o objeto é composto por um ou mais objetos de outra classe.



Order
<ul style="list-style-type: none">• Customer• Order date• Shipping addr.• Order items• Validate()

Order Item
<ul style="list-style-type: none">• Product• Quantity• Purchase price• Validate()

Customer
<ul style="list-style-type: none">• Name• Email address• Home address• Work address• Validate()

Composição: Tem Um

Composição: Tem Um

Aqui aplicamos a
ideia da
CARDINALIDADE

Os objetos possuem
relacionamentos 1:1
– 1:N – N:N

Um pedido tem
apenas 1
consumidor;

Um pedido tem 1
ou mais itens do
pedido;

Customer.cs

- Adicionamos 2 novos atributos de endereço para o consumidor;
- Ou criamos uma lista de endereços;

```
public class Customer
{
    public Customer()
    {
    }
    public Customer(int customerId)
    {
        CustomerId = customerId;
    }
}
```

```
public Address WorkAddress { get; set; }
public Address HomeAddress { get; set; }
```

```
public class Customer
{
    public Customer()
    {
    }
    public Customer(int customerId)
    {
        CustomerId = customerId;
    }
}
```

```
public List<Address> AddressList { get; set; }
```



Atenção!

- Lembre-se que uma lista não possui valores padrão de carregamento, elas precisam ser declaradas e inicializadas, seu valor padrão é null;
- Altere o construtor de Customer para evitar exceções e aplique o ENCADEAMENTO DE CONSTRUTORES.

```
public Customer(): this(0)
{
    ...
}
public Customer(int customerId)
{
    CustomerId = customerId;
    AddressList = new List<Address>();
}
```



Populando Objetos Referenciados

- Agora que a classe Customer tem um relacionamento com Address precisaremos popular os dados do Endereço do Consumidor.
 - Sabemos que não é responsabilidade de Customer lidar com Address e por isso precisamos de um REPOSITÓRIO para Address;
 - Crie o arquivo AddressRepository.cs em sua biblioteca de classes UNOESC.BL e inclua o código no corpo da nova classe.
- 


```
public Address Retrieve(int addressId)
{
    // Create the instance of the Address class
    // Pass in the requested Id
    Address address = new Address(addressId);

    // Code that retrieves the defined address

    // Temporary hard coded values to return
    // a populated address
    if (addressId == 1)
    {
        address.AddressType = 1;
        address.StreetLine1 = "Bag End";
        address.StreetLine2 = "Bagshot row";
        address.City = "Hobbiton";
        address.State = "Shire";
        address.Country = "Middle Earth";
        address.PostalCode = "144";
    }
    return address;
}
```

```
public bool Save(Address address)
{
    // Code that saves the passed in address

    return true;
}
```

Populando Objetos Referenciados

Populando Objetos Referenciados

Adicione também, um método para retornar uma coleção de endereços do consumidor.

```
public IEnumerable<Address> RetrieveByCustomerId(int customerId)
{
    // Code that retrieves the defined addresses
    // for the customer.

    // Temporary hard-coded values to return
    // a set of addresses for a customer
    var addressList = new List<Address>();
    Address address = new Address(1)
    {
        AddressType = 1,
        StreetLine1 = "Bag End",
        StreetLine2 = "Bagshot row",
        City = "Hobbiton",
        State = "Shire",
        Country = "Middle Earth",
        PostalCode = "144"
    };
    addressList.Add(address);
}
```

```
address = new Address(2)
{
    AddressType = 2,
    StreetLine1 = "Green Dragon",
    City = "Bywater",
    State = "Shire",
    Country = "Middle Earth",
    PostalCode = "146"
};
addressList.Add(address);

return addressList;
```

Populando Objetos Referenciados

```
public class CustomerRepository
{
    public CustomerRepository()
    {
        addressRepository = new AddressRepository();
    }

    private AddressRepository addressRepository { get; set; }
}
```

- Agora, altere a classe CustomerRepository para utilizar o novo AddressRepository;
 - Adicione um atributo de ligação e o carregamento no método construtor;

Populando Objetos Referenciados

```
public Customer Retrieve(int customerId)
{
    // Create the instance of the Customer class
    // Pass in the requested id
    Customer customer = new Customer(customerId);

    // Code that retrieves the defined customer

    // Temporary hard-coded values to return
    // a populated customer
    if (customerId == 1)
    {
        customer.EmailAddress = "fbaggins@hobbiton.me";
        customer.FirstName = "Frodo";
        customer.LastName = "Baggins";
        customer.AddressList = addressRepository.RetrieveByCustomerId(customerId).
                                   ToList();
    }
    return customer;
}
```

- No método Retrieve() carregue a propriedade AddressList de customer, para que ele retorne com os dados de Address preenchidos.

Composição por IDs

```
public class Order
{
    public Order()
    {
    }
    public Order(int orderId)
    {
        OrderId = orderId;
    }
    public int CustomerId { get; set; }
    public DateTimeOffset? OrderDate { get; set; }
    public int OrderId { get; private set; }
    public int ShippingAddressId { get; set; }
}
```

- Uma forma diferente de relacionar objetos sem carregar toda sua estrutura é utilizando IDs como referência, assim, poupamos um objeto de todos os detalhes de sua relação;
 - Reduz Acoplamento;
 - Torna mais eficiente, sem carregar tantas informações na memória;
 - Observe a alteração na classe Order.cs


```
public class Order

public Order() : this(0)
{
}

public Order(int orderId)
{
    OrderId = orderId;
    OrderItems = new List<OrderItem>();
}

public int CustomerId { get; set; }
public DateTimeOffset? OrderDate { get;
public int OrderId { get; private set;
public List<OrderItem> OrderItems { get
public int ShippingAddressId { get; set
```

Composição por IDs

- Outro relacionamento importante de Order é com os Itens do Pedido, observe a inclusão do atributo OrderItems e do seu carregamento nos construtores com chamada encadeada.

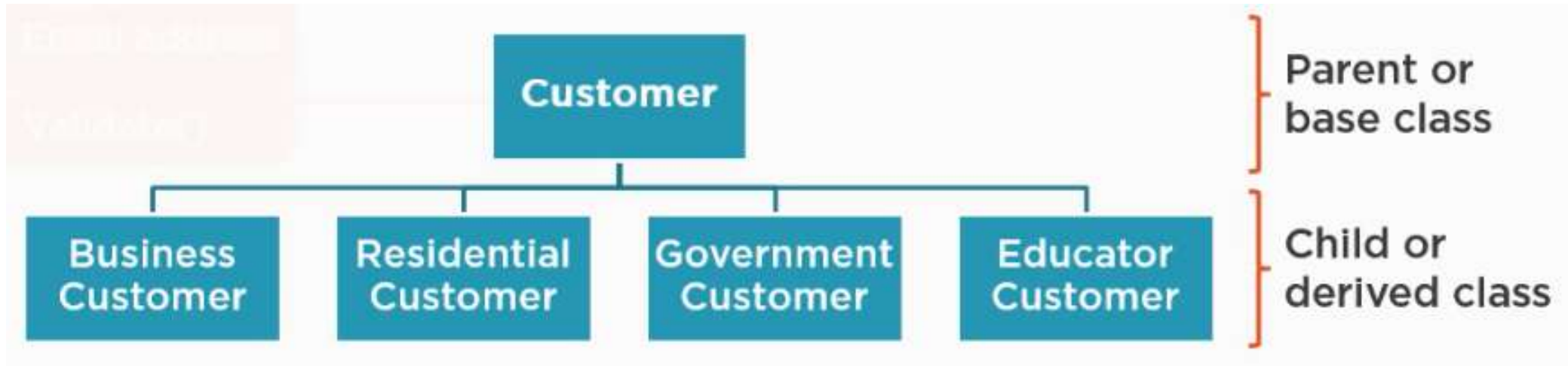
Customer.cs

- Altere a sua classe Customer para poder classificar o tipo de consumidor incluindo a propriedade CustomerType

```
public class Customer
{
    public Customer(): this(0)
    {
    }

    public Customer(int customerId)
    {
        CustomerId = customerId;
        AddressList = new List<Address>();
    }

    public List<Address> AddressList { get; set; }
    public int CustomerId { get; private set; }
    public int CustomerType { get; set; }
    public string EmailAddress { get; set; }
```

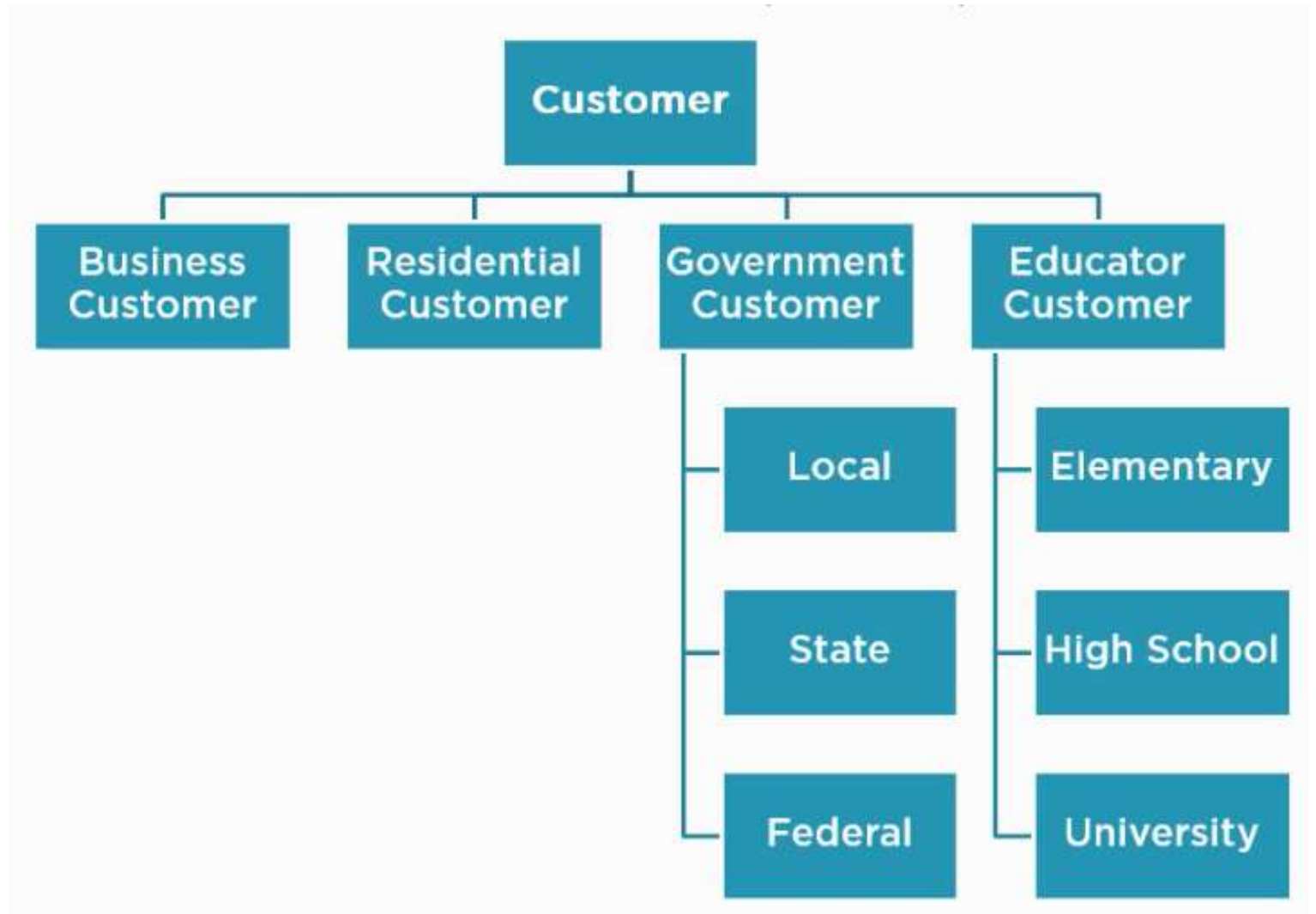


Herança - É Um

- Imagine que você recebeu um novo requisito: "O sistema deve gerenciar os tipos de consumidores negócio, residencial, governamental e educacional".
- Precisamos atualizar o modelo para lidar com estes tipos.
- Herança provê mecanismos para lidar com esta situação, podemos criar classes específicas para lidar com isto, chamamos de Classe Pai e Classe Filha, cada Classe Filha herda ou é membro de uma Classe Pai

Herança - É Um

Podemos expandir a árvore genealógica, conforme a necessidade.



Herança - É Um

Em C#, uma classe pode ter apenas uma Classe Pai;

C# não suporta múltipla herança;

Pode haver múltiplos níveis de herança;