|  |
| --- |
|  |
| C#: Language Oriented |
| Building Objects with Fluency |
|  |
|  |
|  |

# 

# C#: LOP

## Objectives

After completing this lab, you should understand how to:

* Use extension methods to build a chaining API
* Learn about the Builder design pattern

## Overview

The Builder design pattern is a creational pattern (like a Factory) that provides an API for constructing a complex object or object graph. The builder pattern is frequently applied in unit testing projects to construct known test objects, like the “late account” you need for all the tests that exercises business logic against an account with a past due date.

var lateAccount = AccountBuilder.LateAccount().Build();

If you have many test cases against “late accounts” this bit of code will save you from duplicating code like the following everywhere.

var lateAccount = new Account()

{

DueDate = DateTime.Now.AddDays(-1),

Customer =

{

Name = "Sue",

IsVip = false,

Address =

{

City = "Baltimore",

Country = "USA"

}

}

};

The builder pattern can also be found in production code. In this lab we’ll use a language oriented approach to implementing an account builder. The lab consists of a series of unit tests – you’ll need to uncomment the tests one by one and make them pass.

## AccountBuilder

1. Open the solution file in the before\Builder folder.
2. In the Builder project, examine the Account, Customer, and Address classes. These are the objects we’ll need to construct.
3. Open the AccountBuilderTests.cs file.

There are tests inside this C# file, but they are all commented out. The goal is to uncomment the tests one at a time and make each one pass (a simulation of TDD). If you wish, you can disregard this lab manual and let the unit tests guide your work.

1. Uncomment the Step1 test. Implement everything you need to make the test pass.

Although the test looks simple, this first step might be the trickiest one. You’ll need to figure out the right combination of static methods, constructors, and instance methods to make the syntax work (but don’t forget to do the simplest thing that could possible work!).

A working implementation might look like the following (but try it on your own, first!).

public class AccountBuilder

{

public static AccountBuilder DefaultAccount()

{

return new AccountBuilder();

}

public AccountBuilder()

{

\_account = new Account();

}

public Account Build()

{

return \_account;

}

private Account \_account;

}

1. Once Step1 is passing, uncomment Step2. This step forces you to create some default values in all members of the Account object graph.

The solution shown below adds additional code to the AccountBuilder constructor

public AccountBuilder()

{

\_account = new Account

{

Balance = 10000,

DueDate = DateTime.Now.AddDays(1),

Customer = new Customer

{

Name = "Michele",

Address = new Address

{

City = "Washington D.C.",

Country = "USA"

}

}

};

}

1. Move to Step 3. This requires an additional method to give an account a past due status.

*You’ll need to add another instance method to the AccountBuilder class.*

public AccountBuilder WithLatePaymentStatus()

{

\_account.DueDate = DateTime.Now.AddDays(-1);

return this;

}

1. In Step 4 you’ll need another method to create a VIP customer.

public AccountBuilder WithVipCustomer()

{

\_account.Customer.IsVip = true;

return this;

}

1. Finally, in Step 5 you’ll need something to build out the address.

*The approach this test is going to take is that you need an “AddressBuilder” for different address customizations. The way you shape your builders depends on how you need to build things, but in this case we are assuming that building an address is complicated enough to require it’s own builder (we could have added a builder for a Customer, too). In order to get this test to pass, you’ll have to think of how to communicate and toggle between AddressBuilder and AccountBuilder.*

*One approach is to use a nested class. A nested class has access to private state in the outer class. The final solution that will make all tests pass is shown below.*

using System;

namespace Builder

{

public class AccountBuilder

{

public static AccountBuilder DefaultAccount()

{

return new AccountBuilder();

}

public AccountBuilder()

{

\_account = new Account

{

Balance = 10000,

DueDate = DateTime.Now.AddDays(1),

Customer = new Customer

{

Name = "Michele",

Address = new Address

{

City = "Washington D.C.",

Country = "USA"

}

}

};

}

public AccountBuilder WithLatePaymentStatus()

{

\_account.DueDate = DateTime.Now.AddDays(-1);

return this;

}

public AccountBuilder WithVipCustomer()

{

\_account.Customer.IsVip = true;

return this;

}

public AddressBuilder WithAddress()

{

return new AddressBuilder(this);

}

public Account Build()

{

return \_account;

}

private Account \_account;

public class AddressBuilder

{

public AddressBuilder(AccountBuilder accountBuilder)

{

\_accountBuilder = accountBuilder;

}

public AddressBuilder InLondon()

{

\_accountBuilder.\_account.Customer.Address.City =

"London";

\_accountBuilder.\_account.Customer.Address.Country =

"UK";

return this;

}

public AccountBuilder Build()

{

return \_accountBuilder;

}

private readonly AccountBuilder \_accountBuilder;

}

}

}