Metoda: WBTestingCancelOrder(OrdersController)

Testna klasa: WBCancelOrderTests

Radila: Lejla Heleg 19197

1. Obuhvat iskaza/linija *(Statement/Line coverage)*

Za obuhvaćenost svih iskaza prilikom testiranja bila su potrebna četiri testa:

```
// Test1
 [TestMethod]
 public async Task
WBTestingCancelOrder_DeliveryDateCausesTrue_ShouldSetSuccessMessage()
 {

     var controller = new Mock<OrdersController>(dbContextMock.Object)
     {
         CallBase = true
     };

     userMock = new Mock<ClaimsPrincipal>();

     controller.Object.ControllerContext = new ControllerContext
     {
         HttpContext = new DefaultHttpContext { User = userMock.Object }
     };

     userMock.Setup(u => u.FindFirst(ClaimTypes.NameIdentifier)).Returns(new
Claim(ClaimTypes.NameIdentifier, "userId"));

     var tempData = new TempDataDictionary(new DefaultHttpContext(),
Mock.Of<ITempDataProvider>());
     controller.Object.TempData = tempData;

     await controller.Object.WBTestingCancelOrder(new Order { OrderID = 7,
DeliveryDate = deliveryDateTime, PaymentID = 1, purchaseDate = purchaseDateTime });

     Assert.AreEqual("Order successfully canceled.",
controller.Object.TempData["SuccessMessage"]);
 }

 // Test2
 [TestMethod]
 public async Task WBTestingCancelOrder_OrderDoesNotExist_ShouldReturnNotFound()
 {
     var nonExistingOrder = new Order { OrderID = 112001 };

     userMock.Setup(u => u.FindFirst(ClaimTypes.NameIdentifier)).Returns(new
Claim(ClaimTypes.NameIdentifier, "other"));

     var result = await controller.WBTestingCancelOrder(nonExistingOrder);

     Assert.IsInstanceOfType(result, typeof(NotFoundResult));
 }
```

```
 // Test3
 [TestMethod]
 [ExpectedException(typeof(ArgumentNullException))]
 public async Task
WBTestingCancelOrder_WhenUserIdIsNull_ThrowsArgumentNullException()
 {
     var order = new Order { OrderID = 3 };
     var mockHttpContextAccessor = new Mock<IHttpContextAccessor>();

     mockHttpContextAccessor.Setup(a =>
a.HttpContext.User.FindFirst(ClaimTypes.NameIdentifier)).Returns((Claim)null);

     controller.ControllerContext = new ControllerContext
     {
         HttpContext = new DefaultHttpContext { User = new ClaimsPrincipal() }
     };

     await controller.WBTestingCancelOrder(order);

 }

 // Test4
 [TestMethod]
 public async Task
WBTestingCancelOrder_DeliveryDateCausesFalse_ShouldSetErrorMessage()
 {
     var orderToDelete = new Order { OrderID = 3, DeliveryDate =
DateTime.Now.AddDays(2) };

     userMock.Setup(u => u.FindFirst(ClaimTypes.NameIdentifier)).Returns(new
Claim(ClaimTypes.NameIdentifier, "userId"));

     var tempData = new TempDataDictionary(new DefaultHttpContext(),
Mock.Of<ITempDataProvider>());
     controller.TempData = tempData;

     await controller.WBTestingCancelOrder(orderToDelete);

     Assert.AreEqual("You cannot cancel an order scheduled for delivery within the
next 3 days.", controller.TempData["ErrorMessage"]);
 }
```

Pokrivenost koda nakon pokretanja testova:

```
        // method added for WB testing
        [HttpPost]
        [ValidateAntiForgeryToken]
        4 references | Please sign-in to New Relic CodeStream to see Code Level Metrics | 4/4 passing
        public async Task<IActionResult> WBTestingCancelOrder([Bind("OrderID")] Order order)
        {
            // Get the ID of the currently logged-in user
            string userId = User.FindFirst(ClaimTypes.NameIdentifier)?.Value;
            if (userId == null)
            {
                throw new ArgumentNullException(nameof(userId));
            }

            // Get the order that needs to be canceled
            var orderToDelete = await _context.Orders.FindAsync(order.OrderID);
            if (orderToDelete == null)
            {
                return NotFound();
            }

            // Check if the delivery date is less than 3 days before today
            if ((orderToDelete.DeliveryDate - DateTime.Today).TotalDays < 3)
            {

                TempData["ErrorMessage"] = "You cannot cancel an order scheduled for delivery within the next 3 days.";
            }
            else
            {
                // Retrieve productOrders based on the OrderId
                List<ProductOrder> productOrderToDelete = _context.ProductOrders
                    .Where(o => o.OrderID == orderToDelete.OrderID)
                    .ToList();

                // Retrieve payment based on the PaymentId
                Payment paymentToDelete = _context.Payments
                    .FirstOrDefault();


                foreach (var productOrder in productOrderToDelete)
                {
                    List<ProductSales> productSalesToDelete = _context.ProductSales
                   .Where(o => o.ProductID == productOrder.ProductID && o.SalesDate == orderToDelete.purchaseDate)
                   .Take((int)productOrder.ProductQuantity)
                   .ToList();

                    foreach (var productSales in productSalesToDelete)
                    {
                        _context.Remove(productSales);
                    }

                }

                foreach (var productOrder in productOrderToDelete)
                {
                    _context.Remove(productOrder);
                }

                _context.Remove(paymentToDelete);

                _context.Remove(orderToDelete);

                TempData["SuccessMessage"] = "Order successfully canceled.";

            }

            // Reload the user's orders after cancellation
            ViewBag.UserOrders = GetUserOrders(userId);

            return Redirect("ActiveOrders");
        }
    }
}
```

2. Obuhvat grana/odluka *(Branch/Decision Line coverage)*

Za obuhvaćenost svih grana/odluka prilikom testiranja bila su potrebna četiri testa:

```
// Test1
 [TestMethod]
 public async Task
WBTestingCancelOrder_DeliveryDateCausesTrue_ShouldSetSuccessMessage()
 {

     var controller = new Mock<OrdersController>(dbContextMock.Object)
     {
         CallBase = true
     };

     userMock = new Mock<ClaimsPrincipal>();

     controller.Object.ControllerContext = new ControllerContext
     {
         HttpContext = new DefaultHttpContext { User = userMock.Object }
     };

     userMock.Setup(u => u.FindFirst(ClaimTypes.NameIdentifier)).Returns(new
Claim(ClaimTypes.NameIdentifier, "userId"));

     var tempData = new TempDataDictionary(new DefaultHttpContext(),
Mock.Of<ITempDataProvider>());
     controller.Object.TempData = tempData;

     await controller.Object.WBTestingCancelOrder(new Order { OrderID = 7,
DeliveryDate = deliveryDateTime, PaymentID = 1, purchaseDate = purchaseDateTime });

     Assert.AreEqual("Order successfully canceled.",
controller.Object.TempData["SuccessMessage"]);
 }

 // Test2
 [TestMethod]
 public async Task WBTestingCancelOrder_OrderDoesNotExist_ShouldReturnNotFound()
 {
     var nonExistingOrder = new Order { OrderID = 112001 };

     userMock.Setup(u => u.FindFirst(ClaimTypes.NameIdentifier)).Returns(new
Claim(ClaimTypes.NameIdentifier, "other"));

     var result = await controller.WBTestingCancelOrder(nonExistingOrder);

     Assert.IsInstanceOfType(result, typeof(NotFoundResult));
 }
```

```
// Test3
 [TestMethod]
 [ExpectedException(typeof(ArgumentNullException))]
 public async Task
WBTestingCancelOrder_WhenUserIdIsNull_ThrowsArgumentNullException()
 {
     var order = new Order { OrderID = 3 };
     var mockHttpContextAccessor = new Mock<IHttpContextAccessor>();

     mockHttpContextAccessor.Setup(a =>
a.HttpContext.User.FindFirst(ClaimTypes.NameIdentifier)).Returns((Claim)null);

     controller.ControllerContext = new ControllerContext
     {
         HttpContext = new DefaultHttpContext { User = new ClaimsPrincipal() }
     };

     await controller.WBTestingCancelOrder(order);

 }

 // Test4
 [TestMethod]
 public async Task
WBTestingCancelOrder_DeliveryDateCausesFalse_ShouldSetErrorMessage()
 {
     var orderToDelete = new Order { OrderID = 3, DeliveryDate =
DateTime.Now.AddDays(2) };

     userMock.Setup(u => u.FindFirst(ClaimTypes.NameIdentifier)).Returns(new
Claim(ClaimTypes.NameIdentifier, "userId"));

     var tempData = new TempDataDictionary(new DefaultHttpContext(),
Mock.Of<ITempDataProvider>());
     controller.TempData = tempData;

     await controller.WBTestingCancelOrder(orderToDelete);

     Assert.AreEqual("You cannot cancel an order scheduled for delivery within the
next 3 days.", controller.TempData["ErrorMessage"]);
 }
```

Pokrivenost koda nakon pokretanja testova:

```
        // method added for WB testing
        [HttpPost]
        [ValidateAntiForgeryToken]
        4 references | Please sign-in to New Relic CodeStream to see Code Level Metrics | 4/4 passing
        public async Task<IActionResult> WBTestingCancelOrder([Bind("OrderID")] Order order)
        {
            // Get the ID of the currently logged-in user
            string userId = User.FindFirst(ClaimTypes.NameIdentifier)?.Value;
            if (userId == null)
            {
                throw new ArgumentNullException(nameof(userId));
            }

            // Get the order that needs to be canceled
            var orderToDelete = await _context.Orders.FindAsync(order.OrderID);
            if (orderToDelete == null)
            {
                return NotFound();
            }

            // Check if the delivery date is less than 3 days before today
            if ((orderToDelete.DeliveryDate - DateTime.Today).TotalDays < 3)
            {

                TempData["ErrorMessage"] = "You cannot cancel an order scheduled for delivery within the next 3 days.";
            }
            else
            {
                // Retrieve productOrders based on the OrderId
                List<ProductOrder> productOrderToDelete = _context.ProductOrders
                    .Where(o => o.OrderID == orderToDelete.OrderID)
                    .ToList();

                // Retrieve payment based on the PaymentId
                Payment paymentToDelete = _context.Payments
                    .FirstOrDefault();


                foreach (var productOrder in productOrderToDelete)
                {
                    List<ProductSales> productSalesToDelete = _context.ProductSales
                   .Where(o => o.ProductID == productOrder.ProductID && o.SalesDate == orderToDelete.purchaseDate)
                   .Take((int)productOrder.ProductQuantity)
                   .ToList();

                    foreach (var productSales in productSalesToDelete)
                    {
                        _context.Remove(productSales);
                    }

                }

                foreach (var productOrder in productOrderToDelete)
                {
                    _context.Remove(productOrder);
                }

                _context.Remove(paymentToDelete);

                _context.Remove(orderToDelete);

                TempData["SuccessMessage"] = "Order successfully canceled.";

            }

            // Reload the user's orders after cancellation
            ViewBag.UserOrders = GetUserOrders(userId);

            return Redirect("ActiveOrders");
        }
    }
}
```

3. Obuhvat uslova *(Conditional coverage)*

Za obuhvaćenost grana/odluka prilikom testiranja bila su potrebna četiri testa:

```
// Test1
 [TestMethod]
 public async Task
WBTestingCancelOrder_DeliveryDateCausesTrue_ShouldSetSuccessMessage()
 {

     var controller = new Mock<OrdersController>(dbContextMock.Object)
     {
         CallBase = true
     };

     userMock = new Mock<ClaimsPrincipal>();

     controller.Object.ControllerContext = new ControllerContext
     {
         HttpContext = new DefaultHttpContext { User = userMock.Object }
     };

     userMock.Setup(u => u.FindFirst(ClaimTypes.NameIdentifier)).Returns(new
Claim(ClaimTypes.NameIdentifier, "userId"));

     var tempData = new TempDataDictionary(new DefaultHttpContext(),
Mock.Of<ITempDataProvider>());
     controller.Object.TempData = tempData;

     await controller.Object.WBTestingCancelOrder(new Order { OrderID = 7,
DeliveryDate = deliveryDateTime, PaymentID = 1, purchaseDate = purchaseDateTime });

     Assert.AreEqual("Order successfully canceled.",
controller.Object.TempData["SuccessMessage"]);
 }

 // Test2
 [TestMethod]
 public async Task WBTestingCancelOrder_OrderDoesNotExist_ShouldReturnNotFound()
 {
     var nonExistingOrder = new Order { OrderID = 112001 };

     userMock.Setup(u => u.FindFirst(ClaimTypes.NameIdentifier)).Returns(new
Claim(ClaimTypes.NameIdentifier, "other"));

     var result = await controller.WBTestingCancelOrder(nonExistingOrder);

     Assert.IsInstanceOfType(result, typeof(NotFoundResult));
 }
```

```
// Test3
 [TestMethod]
 [ExpectedException(typeof(ArgumentNullException))]
 public async Task
WBTestingCancelOrder_WhenUserIdIsNull_ThrowsArgumentNullException()
 {
     var order = new Order { OrderID = 3 };
     var mockHttpContextAccessor = new Mock<IHttpContextAccessor>();

     mockHttpContextAccessor.Setup(a =>
a.HttpContext.User.FindFirst(ClaimTypes.NameIdentifier)).Returns((Claim)null);

     controller.ControllerContext = new ControllerContext
     {
         HttpContext = new DefaultHttpContext { User = new ClaimsPrincipal() }
     };

     await controller.WBTestingCancelOrder(order);

 }

 // Test4
 [TestMethod]
 public async Task
WBTestingCancelOrder_DeliveryDateCausesFalse_ShouldSetErrorMessage()
 {
     var orderToDelete = new Order { OrderID = 3, DeliveryDate =
DateTime.Now.AddDays(2) };

     userMock.Setup(u => u.FindFirst(ClaimTypes.NameIdentifier)).Returns(new
Claim(ClaimTypes.NameIdentifier, "userId"));

     var tempData = new TempDataDictionary(new DefaultHttpContext(),
Mock.Of<ITempDataProvider>());
     controller.TempData = tempData;

     await controller.WBTestingCancelOrder(orderToDelete);

     Assert.AreEqual("You cannot cancel an order scheduled for delivery within the
next 3 days.", controller.TempData["ErrorMessage"]);
 }
```

Pokrivenost koda nakon pokretanja testova:

```
        // method added for WB testing
        [HttpPost]
        [ValidateAntiForgeryToken]
        4 references | Please sign-in to New Relic CodeStream to see Code Level Metrics | 4/4 passing
        public async Task<IActionResult> WBTestingCancelOrder([Bind("OrderID")] Order order)
        {
            // Get the ID of the currently logged-in user
            string userId = User.FindFirst(ClaimTypes.NameIdentifier)?.Value;
            if (userId == null)
            {
                throw new ArgumentNullException(nameof(userId));
            }

            // Get the order that needs to be canceled
            var orderToDelete = await _context.Orders.FindAsync(order.OrderID);
            if (orderToDelete == null)
            {
                return NotFound();
            }

            // Check if the delivery date is less than 3 days before today
            if ((orderToDelete.DeliveryDate - DateTime.Today).TotalDays < 3)
            {

                TempData["ErrorMessage"] = "You cannot cancel an order scheduled for delivery within the next 3 days.";
            }
            else
            {
                // Retrieve productOrders based on the OrderId
                List<ProductOrder> productOrderToDelete = _context.ProductOrders
                    .Where(o => o.OrderID == orderToDelete.OrderID)
                    .ToList();

                // Retrieve payment based on the PaymentId
                Payment paymentToDelete = _context.Payments
                    .FirstOrDefault();


                foreach (var productOrder in productOrderToDelete)
                {
                    List<ProductSales> productSalesToDelete = _context.ProductSales
                   .Where(o => o.ProductID == productOrder.ProductID && o.SalesDate == orderToDelete.purchaseDate)
                   .Take((int)productOrder.ProductQuantity)
                   .ToList();

                    foreach (var productSales in productSalesToDelete)
                    {
                        _context.Remove(productSales);
                    }

                }

                foreach (var productOrder in productOrderToDelete)
                {
                    _context.Remove(productOrder);
                }

                _context.Remove(paymentToDelete);

                _context.Remove(orderToDelete);

                TempData["SuccessMessage"] = "Order successfully canceled.";

            }

            // Reload the user's orders after cancellation
            ViewBag.UserOrders = GetUserOrders(userId);

            return Redirect("ActiveOrders");
        }
    }
}
```

4. **Modifikovani uslov/odluka obuhvat** ***(Modified condition/decision coverage MCDC)***

Za obuhvaćenost modifikiovanih uslova/odluka prilikom testiranja bilo je potrebno šest testova:

```
// Test1
 [TestMethod]
 public async Task
WBTestingCancelOrder_DeliveryDateCausesTrue_ShouldSetSuccessMessage()
 {

     var controller = new Mock<OrdersController>(dbContextMock.Object)
     {
         CallBase = true
     };

     userMock = new Mock<ClaimsPrincipal>();

     controller.Object.ControllerContext = new ControllerContext
     {
         HttpContext = new DefaultHttpContext { User = userMock.Object }
     };

     userMock.Setup(u => u.FindFirst(ClaimTypes.NameIdentifier)).Returns(new
Claim(ClaimTypes.NameIdentifier, "userId"));

     var tempData = new TempDataDictionary(new DefaultHttpContext(),
Mock.Of<ITempDataProvider>());
     controller.Object.TempData = tempData;

     await controller.Object.WBTestingCancelOrder(new Order { OrderID = 7,
DeliveryDate = deliveryDateTime, PaymentID = 1, purchaseDate = purchaseDateTime });

     Assert.AreEqual("Order successfully canceled.",
controller.Object.TempData["SuccessMessage"]);
 }

 // Test2
 [TestMethod]
 public async Task WBTestingCancelOrder_OrderDoesNotExist_ShouldReturnNotFound()
 {
     var nonExistingOrder = new Order { OrderID = 112001 };

     userMock.Setup(u => u.FindFirst(ClaimTypes.NameIdentifier)).Returns(new
Claim(ClaimTypes.NameIdentifier, "other"));

     var result = await controller.WBTestingCancelOrder(nonExistingOrder);

     Assert.IsInstanceOfType(result, typeof(NotFoundResult));
 }

 // Test3
 [TestMethod]
 [ExpectedException(typeof(ArgumentNullException))]
 public async Task
WBTestingCancelOrder_WhenUserIdIsNull_ThrowsArgumentNullException()
```

```
 {
     var order = new Order { OrderID = 3 };
     var mockHttpContextAccessor = new Mock<IHttpContextAccessor>();

     mockHttpContextAccessor.Setup(a =>
a.HttpContext.User.FindFirst(ClaimTypes.NameIdentifier)).Returns((Claim)null);

     controller.ControllerContext = new ControllerContext
     {
         HttpContext = new DefaultHttpContext { User = new ClaimsPrincipal() }
     };

     await controller.WBTestingCancelOrder(order);

 }

 // Test4
 [TestMethod]
 public async Task
WBTestingCancelOrder_DeliveryDateCausesFalse_ShouldSetErrorMessage()
 {
     var orderToDelete = new Order { OrderID = 3, DeliveryDate =
DateTime.Now.AddDays(2) };

     userMock.Setup(u => u.FindFirst(ClaimTypes.NameIdentifier)).Returns(new
Claim(ClaimTypes.NameIdentifier, "userId"));

     var tempData = new TempDataDictionary(new DefaultHttpContext(),
Mock.Of<ITempDataProvider>());
     controller.TempData = tempData;

     await controller.WBTestingCancelOrder(orderToDelete);

     Assert.AreEqual("You cannot cancel an order scheduled for delivery within the
next 3 days.", controller.TempData["ErrorMessage"]);
 }

// Test5
[TestMethod]
public async Task WBTestingCancelOrder_TodayCausesTrue_ShouldSetSuccessMessage()
{

    var controller = new Mock<OrdersController>(dbContextMock.Object)
    {
        CallBase = true
    };

    userMock = new Mock<ClaimsPrincipal>();

    controller.Object.ControllerContext = new ControllerContext
    {
        HttpContext = new DefaultHttpContext { User = userMock.Object }
    };

    userMock.Setup(u => u.FindFirst(ClaimTypes.NameIdentifier)).Returns(new
Claim(ClaimTypes.NameIdentifier, "userId"));
```

```
    var tempData = new TempDataDictionary(new DefaultHttpContext(),
Mock.Of<ITempDataProvider>());
    controller.Object.TempData = tempData;

    await controller.Object.WBTestingCancelOrder(new Order { OrderID = 7,
DeliveryDate = DateTime.Now.AddDays(4), PaymentID = 1, purchaseDate =
purchaseDateTime });

    Assert.AreEqual("Order successfully canceled.",
controller.Object.TempData["SuccessMessage"]);
}

// Test6

[TestMethod]
public async Task WBTestingCancelOrder_TodayCausesFalse_ShouldSetErrorMessage()
{
    controller.today = DateTime.Now.AddDays(2);
    var orderToDelete = new Order { OrderID = 3, DeliveryDate =
DateTime.Now.AddDays(4)};

    userMock.Setup(u => u.FindFirst(ClaimTypes.NameIdentifier)).Returns(new
Claim(ClaimTypes.NameIdentifier, "userId"));

    var tempData = new TempDataDictionary(new DefaultHttpContext(),
Mock.Of<ITempDataProvider>());
    controller.TempData = tempData;

    await controller.WBTestingCancelOrder(orderToDelete);

    Assert.AreEqual("You cannot cancel an order scheduled for delivery within the
next 3 days.", controller.TempData["ErrorMessage"]);
}
```

S obzirom na to da nigdje nema naredbe if koja ima više uslova koji zajedno daju true ili false, tabela nije formirana. Umjesto toga su se gledali pojedinačni slučajevi kada promjenom parametra datuma isporuke (DeliveryDate) i referentnog dana (today) dolazi i do promjene ishoda naredbe if.

Pokrivenost koda nakon pokretanja testova:

```
        // method added for WB testing
        [HttpPost]
        [ValidateAntiForgeryToken]
        4 references | Please sign-in to New Relic CodeStream to see Code Level Metrics | 4/4 passing
        public async Task<IActionResult> WBTestingCancelOrder([Bind("OrderID")] Order order)
        {
            // Get the ID of the currently logged-in user
            string userId = User.FindFirst(ClaimTypes.NameIdentifier)?.Value;
            if (userId == null)
            {
                throw new ArgumentNullException(nameof(userId));
            }

            // Get the order that needs to be canceled
            var orderToDelete = await _context.Orders.FindAsync(order.OrderID);
            if (orderToDelete == null)
            {
                return NotFound();
            }

            // Check if the delivery date is less than 3 days before today
            if ((orderToDelete.DeliveryDate - DateTime.Today).TotalDays < 3)
            {

                TempData["ErrorMessage"] = "You cannot cancel an order scheduled for delivery within the next 3 days.";
            }
            else
            {
                // Retrieve productOrders based on the OrderId
                List<ProductOrder> productOrderToDelete = _context.ProductOrders
                    .Where(o => o.OrderID == orderToDelete.OrderID)
                    .ToList();

                // Retrieve payment based on the PaymentId
                Payment paymentToDelete = _context.Payments
                    .FirstOrDefault();


                foreach (var productOrder in productOrderToDelete)
                {
                    List<ProductSales> productSalesToDelete = _context.ProductSales
                   .Where(o => o.ProductID == productOrder.ProductID && o.SalesDate == orderToDelete.purchaseDate)
                   .Take((int)productOrder.ProductQuantity)
                   .ToList();

                    foreach (var productSales in productSalesToDelete)
                    {
                        _context.Remove(productSales);
                    }

                }

                foreach (var productOrder in productOrderToDelete)
                {
                    _context.Remove(productOrder);
                }

                _context.Remove(paymentToDelete);

                _context.Remove(orderToDelete);

                TempData["SuccessMessage"] = "Order successfully canceled.";

            }

            // Reload the user's orders after cancellation
            ViewBag.UserOrders = GetUserOrders(userId);

            return Redirect("ActiveOrders");
        }
    }
}
```

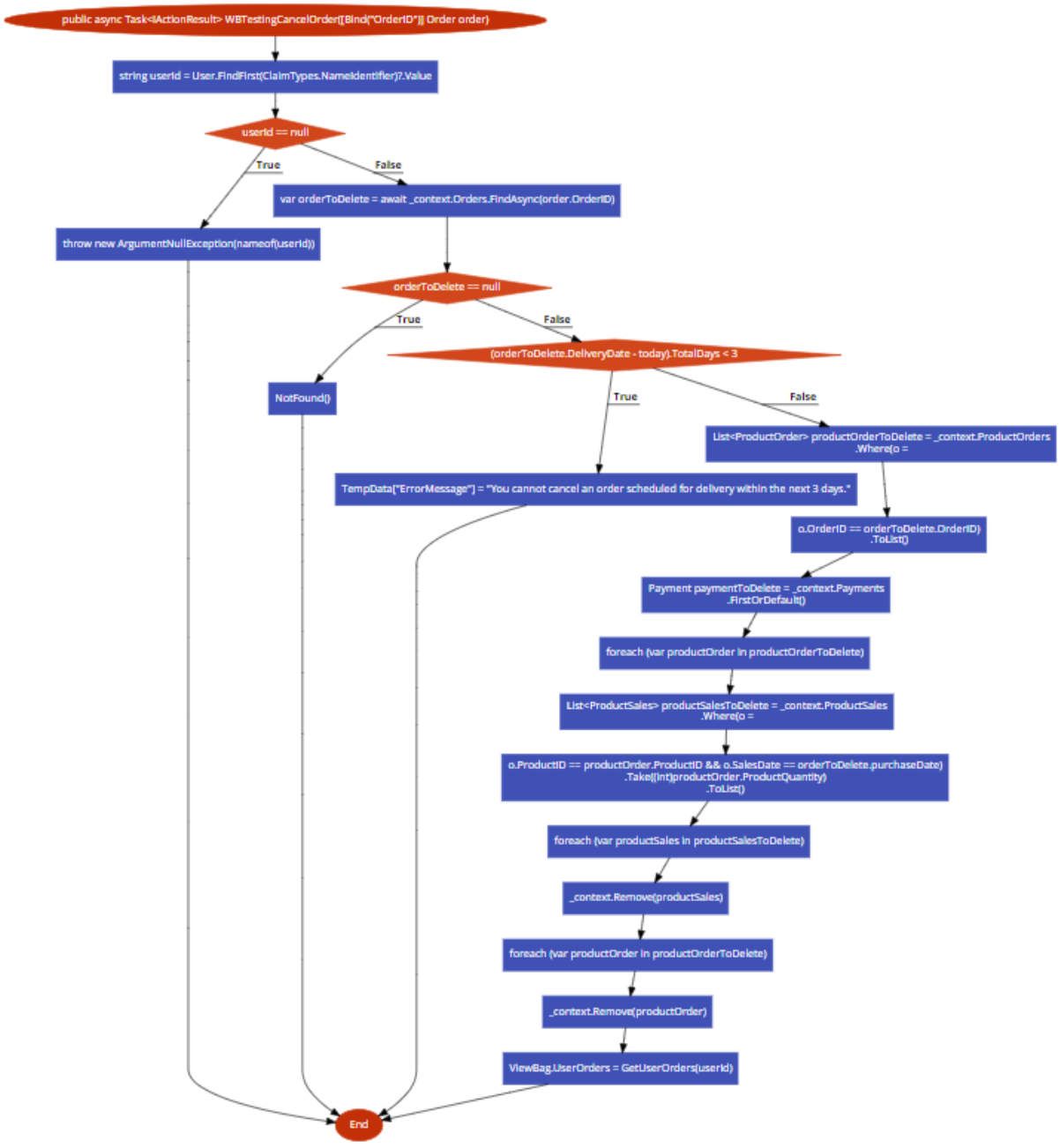
5. Obuhvat petlji *(Loop coverage)*

Kako su foreach petlje formirane za prisanje pojedinih ćelija u redu kroz koji se prolazi u istim, postoje 2 moguća iskoda: petlja nema iteracija (red ne postoji) i ima iteracija koliko ima kolona u tabeli (red postoji).

Testni slučaj:

6. Obuhvat puteva *(path coverage)*

Dijagram toka modula:



Programski graf modula:

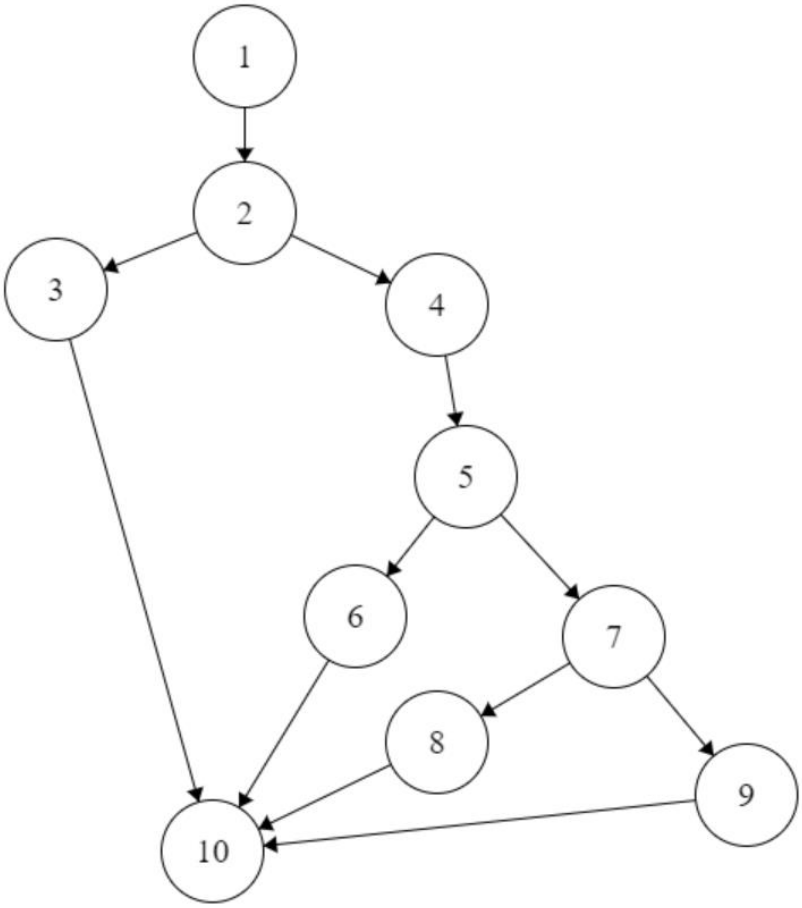


Tabela mogućih tokova:

Redni broj puta	Čvorovi koje obuhvata put
1	1,2,3,10
2	1,2,4,5,6,10
3	1,2,4,5,7,8,10
4	1,2,4,5,7,9,10

Da bi se ostvarila potpuna pokivenost potrebno je napisati četiri testa koji obuhvataju sva četiri moguća toka:

```
 // Tok 1
 [TestMethod]
 [ExpectedException(typeof(ArgumentNullException))]
 public async Task
WBTestingCancelOrder_WhenUserIdIsNull_ThrowsArgumentNullException()
 {
     var order = new Order { OrderID = 3 };
     var mockHttpContextAccessor = new Mock<IHttpContextAccessor>();

     mockHttpContextAccessor.Setup(a =>
a.HttpContext.User.FindFirst(ClaimTypes.NameIdentifier)).Returns((Claim)null);

     controller.ControllerContext = new ControllerContext
     {
         HttpContext = new DefaultHttpContext { User = new ClaimsPrincipal() }
     };

     await controller.WBTestingCancelOrder(order);

 }

// Tok 2
 [TestMethod]
 public async Task WBTestingCancelOrder_OrderDoesNotExist_ShouldReturnNotFound()
 {
     var nonExistingOrder = new Order { OrderID = 112001 };

     userMock.Setup(u => u.FindFirst(ClaimTypes.NameIdentifier)).Returns(new
Claim(ClaimTypes.NameIdentifier, "other"));

     var result = await controller.WBTestingCancelOrder(nonExistingOrder);

     Assert.IsInstanceOfType(result, typeof(NotFoundResult));
 }

 // Tok 3
 [TestMethod]
 public async Task
WBTestingCancelOrder_DeliveryDateCausesFalse_ShouldSetErrorMessage()
 {
     var orderToDelete = new Order { OrderID = 3, DeliveryDate =
DateTime.Now.AddDays(2) };

     userMock.Setup(u => u.FindFirst(ClaimTypes.NameIdentifier)).Returns(new
Claim(ClaimTypes.NameIdentifier, "userId"));

     var tempData = new TempDataDictionary(new DefaultHttpContext(),
Mock.Of<ITempDataProvider>());
     controller.TempData = tempData;

     await controller.WBTestingCancelOrder(orderToDelete);

     Assert.AreEqual("You cannot cancel an order scheduled for delivery within the
next 3 days.", controller.TempData["ErrorMessage"]);
 }
```

```
// Tok 4
 [TestMethod]
 public async Task
WBTestingCancelOrder_DeliveryDateCausesTrue_ShouldSetSuccessMessage()
 {

     var controller = new Mock<OrdersController>(dbContextMock.Object)
     {
         CallBase = true
     };

     userMock = new Mock<ClaimsPrincipal>();

     controller.Object.ControllerContext = new ControllerContext
     {
         HttpContext = new DefaultHttpContext { User = userMock.Object }
     };

     userMock.Setup(u => u.FindFirst(ClaimTypes.NameIdentifier)).Returns(new
Claim(ClaimTypes.NameIdentifier, "userId"));

     var tempData = new TempDataDictionary(new DefaultHttpContext(),
Mock.Of<ITempDataProvider>());
     controller.Object.TempData = tempData;

     await controller.Object.WBTestingCancelOrder(new Order { OrderID = 7,
DeliveryDate = deliveryDateTime, PaymentID = 1, purchaseDate = purchaseDateTime });

     Assert.AreEqual("Order successfully canceled.",
controller.Object.TempData["SuccessMessage"]);
 }
```

Pokrivenost koda nakon pokretanja testova:

```
        // method added for WB testing
        [HttpPost]
        [ValidateAntiForgeryToken]
        4 references | Please sign-in to New Relic CodeStream to see Code Level Metrics | 4/4 passing
        public async Task<IActionResult> WBTestingCancelOrder([Bind("OrderID")] Order order)
        {
            // Get the ID of the currently logged-in user
            string userId = User.FindFirst(ClaimTypes.NameIdentifier)?.Value;
            if (userId == null)
            {
                throw new ArgumentNullException(nameof(userId));
            }

            // Get the order that needs to be canceled
            var orderToDelete = await _context.Orders.FindAsync(order.OrderID);
            if (orderToDelete == null)
            {
                return NotFound();
            }

            // Check if the delivery date is less than 3 days before today
            if ((orderToDelete.DeliveryDate - DateTime.Today).TotalDays < 3)
            {

                TempData["ErrorMessage"] = "You cannot cancel an order scheduled for delivery within the next 3 days.";
            }
            else
            {
                // Retrieve productOrders based on the OrderId
                List<ProductOrder> productOrderToDelete = _context.ProductOrders
                    .Where(o => o.OrderID == orderToDelete.OrderID)
                    .ToList();

                // Retrieve payment based on the PaymentId
                Payment paymentToDelete = _context.Payments
                    .FirstOrDefault();


                foreach (var productOrder in productOrderToDelete)
                {
                    List<ProductSales> productSalesToDelete = _context.ProductSales
                   .Where(o => o.ProductID == productOrder.ProductID && o.SalesDate == orderToDelete.purchaseDate)
                   .Take((int)productOrder.ProductQuantity)
                   .ToList();

                    foreach (var productSales in productSalesToDelete)
                    {
                        _context.Remove(productSales);
                    }

                }

                foreach (var productOrder in productOrderToDelete)
                {
                    _context.Remove(productOrder);
                }

                _context.Remove(paymentToDelete);

                _context.Remove(orderToDelete);

                TempData["SuccessMessage"] = "Order successfully canceled.";

            }

            // Reload the user's orders after cancellation
            ViewBag.UserOrders = GetUserOrders(userId);

            return Redirect("ActiveOrders");
        }
    }
}
```