1. Setup Console Application and Classes
   1. Console Application – UI,
   2. Classes – BLL, Data, Models, Tests
   3. Add References
      1. UI – BLL and Models
      2. BLL – Models and Data
      3. Data – Models
      4. Tests – BLL, Data, and Models
2. Setup UI
   1. Create a Menu class that prompts the user for a 1-5 selection
      1. Display Orders
      2. Add an Order
      3. Edit an Order
      4. Remove an Order
      5. Quit
   2. Create a switch statement that checks for a selection
      1. Keep prompting the user until they enter a valid selection
      2. A workflow will be set for each statement as they are completed
   3. Create a class called ConsoleIO with 2 methods
      1. One to display order information
      2. Another to display product information
3. Order Models
   1. Setup Order files in Models
      1. Order information
      2. Product information
      3. Tax information
   2. Setup Responses
   3. Setup Interface IOrderRepository
4. Manager Class
   1. Create an OrderMananger method
   2. Create a GetDate method that prompts the user for a DateTime to search
      1. Return false if order does not exist with the proper response message
      2. Return true otherwise
   3. Set a App.config reference in BLL
      1. In that file, set a key to “Mode” and value to “Test”
   4. Create an OrderMananger Factory with a Switch statement
      1. Add a “Test” case
      2. Default with an exception
5. Test Repository
   1. Create a test repository class in Data
   2. Instantiate a new Order as \_order
   3. Setup methods for saving an order
      1. SaveOrder – Set \_order equal to order
      2. SaveNewOrder – Add order to \_orders
      3. LoadOrder – Return \_order
      4. RemoveOrder – Remove an order from \_order
      5. EditOrder – Will review this later
   4. More methods to come
6. Create a OrderLookupWorkflow
   1. Prompt the user for a DateTime
      1. Ensure the input is a valid DateTime using DateTime.TryParse
      2. Return a Response set to false with a message stating the format was invalid
      3. If the TryParse was successful, set the response equal to the method that checks for order dates
      4. Have the Console.IO print the order details if there are orders from the date
      5. If no orders from the date are present, print a message saying so
7. Create a Workflow for adding orders
   1. Instantiate tax and product information
8. In the OrderManager class, create an AddOrderInformation workflow that takes in the tax and product information
9. Create 4 methods in the OrderManager class that return their respective variables. Set each part of the Order’s variables equal to the method’s return values
   1. GetName – Prompt the user for the customer name
   2. GetState – Prompt the user for their state
      1. If that state does not exist in the database, prompt the user for another state
   3. GetProductType – Prompt the user for one of the selected products
      1. Print out the product information for the user to select
   4. GetArea – Get the area of work needed
      1. Must be at least 100
      2. TryParse to make sure the information is valid
   5. Setup another method called OrderCalculations for calculating everything else required for the order. Have it take in the order, taxes, and products
      1. OrderNumber – Set to the most recent order number plus 1
      2. TaxRate – Set to the respectives state’s tax rate
      3. CostPerSquareFoot – Product’s cost per square foot
      4. LaborCostPerSquareFoot – Product’s labor cost per square foot
      5. MaterialCost – Cost per square foot time the area
      6. LaborCost – Labor cost per square foot times the area
      7. Tax – Labor cost plus the material cost. Multiply those by the tax rate divided by 100
      8. Total – LaborCost, MaterialCost, and Tax all added together
      9. Return the order
   6. All the information should be calculated. Return the order to the AddOrderWorkflow
   7. Print out the order information with the ConsoleIO
   8. Prompt the user if they want to save the order
      1. If yes, call the SaveNewOrder method from the TestRepository
      2. If no, return without doing anything
10. Removing an order
    1. Create a RemoveOrderWorkflow
    2. Just like the LookupOrderWorkflow, prompt the user for a date and TryParse it into a DateTime. The responses will be the same except we do not return if an OrderDate exists
    3. Create a method in the OrderManager that prompts the user for an OrderNumber
    4. In the OrderTestRepository, create a method that takes in the order number
       1. Check \_orders for an order number that matches the number entered
       2. Use a SingleOrDefault statement
    5. If the order exists…
       1. Display the order information
       2. Prompt the user for the deletion
          * If yes, call the method for removing the order
          * If no, return without doing anything
11. Edit an order
    1. Just like the RemoveOrderWorkflow, prompt the user for a date and order number
    2. If the order exists, display the order
    3. Create and EditOrderInformation method in the OrderManager
       1. Take in the order
       2. Take in the tax/product information
    4. Set 2 bools for the Order class in Models
       1. Edit
       2. Recalculate
    5. In the EditOrderInformation method, set Edit to true and Recalculate to false
    6. Prompt the user for the customer name, state, product type, and area
    7. Edits will need to be made to the 4 methods used for step 11f
       1. If the user presses enter without typing anything, the specified information will not change
       2. Set an if statement that checks if Edit is true and if the user input is blank
    8. For the state, productType, and area…
       1. If any of these are changed, we will need to recalculate them
       2. Set the bool recalculate to true if the user enters valid input that is not blank
    9. If Recalculate is set to true, call the OrderCalculations method to calculate the order again
    10. In the OrderCalculations method, add an if statement that checks if the Edit bool is true
        1. If true, the order number will not changes
    11. Return the order
    12. Back in the EditOrderWorkflow, print out the new order information using ConsoleIO
    13. Prompt the user to save the changes
        1. Create a SaveChanges method in OrderManager and a EditOrder method in the OrderTestRepository
        2. SaveChanges
           * Have the orderTestRepository call the EditOrder method with the edited order
        3. EditOrder
           * Set all of the order’s information changed to the edited order
        4. If the user enters yes, call the SaveChanges method to save the order
        5. If no, return without doing anything
12. Using the text files
    1. Create an OrderProdRespository in Data
    2. Copy the data from OrderTestRepository to OrderProdRepository
    3. Set up the order information, tax information, and product information to be read from their respective files
    4. In the SaveNewOrder method…
       1. Append the text from the order to the file
       2. Make sure the formatting is done properly
    5. In the RemoveOrder method…
       1. Check if the order file exists, delete if it does
       2. The first line should be the variable information (OrderDate,OrderNumber,etc)
       3. Print out the entire order data in the proper format to the text file with a foreach loop
    6. In the EditOrder method…
       1. Repeat the 3 steps from step e
       2. In the foreach loop, add an if statement that checks if the edited order number equals the current order number from \_orders
          * If true, write all of the edited order information to the text file instead of the text from \_orders
          * Use an else statement that prints out the order information if the order numbers do not match
    7. In the OrderManagerFactory, add a case for “Prod” that calls the OrderProdRepository
    8. In the App.Config, change the “Test” key to “Prod”