Refactoring to Testable Code in F#

IMPLEMENTING A PROGRESSIVELY TESTABLE FEATURE

Objectives

- To demonstrate how to create a “real-life” application feature in F#

  Real-life meaning that it has the following:

  - “IO” dependencies (database, API calls, sending emails, etc.)
  - Business logic that make decisions on the external data
  - Works with “async” calls
- To show how to develop a “progressively testable” feature in F#

  Progressively testable - Can be easily refactored to be testable if/when needed
- To show comparable fully testable C# and F# implementations side-by-side

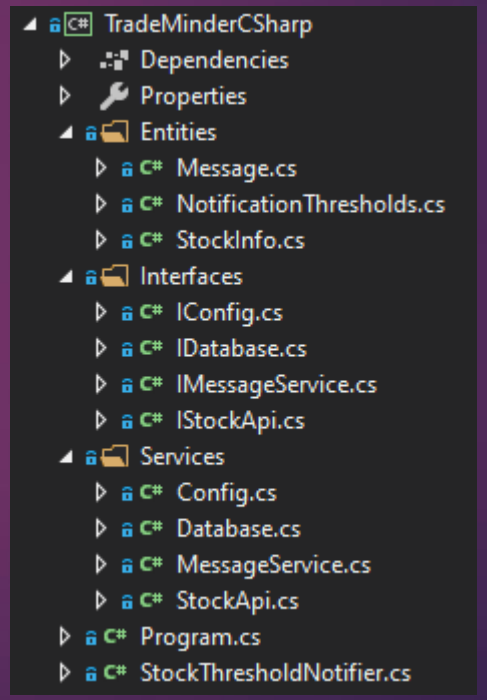
TradeMinder Demo Specs

- Create a console app that notifies a user if a given stock price is above or below a user defined threshold.
  - User defined thresholds are stored in a local Sqlite database (management of stock thresholds not covered).
  - Stock prices will be checked via a web API.
  - Messaging is stubbed out to only write to console.

C# Implementation

Solution Layout

- **StockThresholdNotifier.cs** contains the feature workflow.
- "IO" dependencies are modeled as "Services"
- Each service has an interface to facilitate testability and loose coupling.
  - See Microsoft Unit Testing Best Practices
  - Dependencies are injected into feature class constructor and can be substituted with mocks during unit testing.
- This is a fairly common architecture that has been used successfully in *many* projects



C# - Program.cs

Notes

- This is the "composition root"
- Dependencies are instantiated and then injected into the feature class.
  - *Some apps use a DI container instead,*
    like Ninject.

```
using System;
using TradeMinderCSharp.Services;

namespace TradeMinderCSharp
{
    class Program
    {
        static void Main(string[] args)
        {
            if (args.Length == 2)
            {
                var symbol = args[0];
                var email = args[1];

                var notifier = new StockThresholdNotifier(
                    new Database(new Config()),
                    new StockApi(),
                    new MessageService());

                notifier.CheckStock(symbol, email).Wait();
            }
            else
            {
                Console.WriteLine("Invalid args. Expected: 'TradeMinderCSharp.exe MSFT jmarr@microdesk.com'");
            }
        }
    }
}
```

C# - Feature

Notes

- Dependencies are c'tor injected
- CheckStock "feature" method:
  - 1) Gets input data
  - 2) Applies thresholds (business rules)
  - 3) Sends message (maybe)
- This code is fully testable, and can be written and tested before the injected IO dependencies are implemented.

```csharp
using System;
using System.Threading.Tasks;
using TradeMinderCSharp.Entities;
using TradeMinderCSharp.Interfaces;

namespace TradeMinderCSharp
{
    public class StockThresholdNotifier
    {
        private IDatabase _database;
        private IStockApi _stockApi;
        private IMessageService _messageSvc;

        public StockThresholdNotifier(IDatabase database, IStockApi stockApi, IMessageService messageSvc)
        {
            _database = database;
            _stockApi = stockApi;
            _messageSvc = messageSvc;
        }

        /// <summary>
        /// This pure function creates a notification (or not) based on the stock info and thresholds.
        /// </summary>
        public Message MaybeCreateMessage(StockInfo stock, NotificationThresholds thresholds)
        {
            if (stock.Value > thresholds.High)
                return new Message { Email = thresholds.Email, Body = $"{stock.Symbol} exceeds the maximum value of ${thresholds.High}." };
            else if (stock.Value < thresholds.Low)
                return new Message { Email = thresholds.Email, Body = $"{stock.Symbol} is less than the minimum value of ${thresholds.Low}." };
            else
                return null;
        }

        /// <summary>
        /// This function contains the logic to run the feature.
        /// </summary>
        public async Task CheckStock(string symbol, string email)
        {
            // 1) IO - Get necessary data
            var stock = await _stockApi.GetStock(symbol);
            var thresholds = await _database.GetThresholds(symbol, email);

            if (stock != null && thresholds != null)
            {
                // 2) Process business rules to create an alert (or not).
                var message = MaybeCreateMessage(stock, thresholds);

                // 3) IO - Send the message (if one exists)
                if (message != null)
                {
                    await _messageSvc.SendMessage(message);
                }
                else
                {
                    Console.WriteLine("No message was sent.");
                }
            }
            else
            {
                Console.WriteLine("Requires stock and threshold.");
            }
        }
    }
}
```

C# - Entities

Notes

- **NotificationThresholds.cs**
  User defined thresholds that are stored in Sqlite
- **StockInfo.cs**
  Represents stock price
- **Message.cs**
  A message is generated only if a stock value is higher the thresholds.High or lower than thresholds.Low.

```cs
using System;
using System.Collections.Generic;
using System.Text;

namespace TradeMinderCSharp.Entities
{
    public class StockInfo
    {
        public string Symbol { get; set; }
        public DateTime Date { get; set; }
        public decimal Value { get; set; }
    }
}
```

```cs
using System;
using System.Collections.Generic;
using System.Text;

namespace TradeMinderCSharp.Entities
{
    public class Message
    {
        public string Email { get; set; }
        public string Body { get; set; }
    }
}
```

```cs
using System;
using System.Collections.Generic;
using System.Text;

namespace TradeMinderCSharp.Entities
{
    public class NotificationThresholds
    {
        public string Symbol { get; set; }
        public decimal Low { get; set; }
        public decimal High { get; set; }
        public string Email { get; set; }
    }
}
```

C# - Interfaces

Notes

- Interfaces allow us to create "loosely coupled" services.
- A.k.a. "design by contract"

```cs
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading.Tasks;

namespace TradeMinderCSharp.Interfaces
{
    public interface IStockApi
    {
        Task<Entities.StockInfo> GetStock(string symbol);
    }
}
```

```cs
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading.Tasks;

namespace TradeMinderCSharp.Interfaces
{
    public interface IMessageService
    {
        Task SendMessage(Entities.Message message);
    }
}
```

```cs
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading.Tasks;

namespace TradeMinderCSharp.Interfaces
{
    public interface IDatabase
    {
        Task<Entities.NotificationThresholds> GetThresholds(string symbol, string email);
    }
}
```

```cs
using System;
using System.Collections.Generic;
using System.Text;

namespace TradeMinderCSharp.Interfaces
{
    public interface IConfig
    {
        string GetConnectionString();
    }
}
```

C# - Database

Notes

- Connection string info is constructor injected
- Basic ADO.NET query
- Either returns **NotificationThresholds** or **null**
  - (The seasoned dev should know to check for nulls!)

```
using Microsoft.Data.Sqlite;
using System;
using System.Collections.Generic;
using System.Data;
using System.Text;
using System.Threading.Tasks;
using TradeMinderCSharp.Entities;
using TradeMinderCSharp.Interfaces;

namespace TradeMinderCSharp.Services
{
    public class Database : Interfaces.IDatabase
    {
        private IConfig _config;

        public Database(IConfig config)
        {
            _config = config;
        }

        /// <summary>
        /// Gets data info from the database
        /// </summary>
        public async Task<NotificationThresholds> GetThresholds(string symbol, string email)
        {
            using (var conn = new SqliteConnection(_config.GetConnectionString()))
            {
                var cmd = conn.CreateCommand();
                cmd.CommandText = "SELECT * FROM Thresholds WHERE Symbol=$Symbol AND Email=$Email";
                cmd.Parameters.AddRange(new[] { new SqliteParameter("$Symbol", symbol), new SqliteParameter("$Email", email) });
                conn.Open();
                using (var rdr = await cmd.ExecuteReaderAsync(CommandBehavior.CloseConnection))
                {
                    if (rdr.Read())
                    {
                        return new NotificationThresholds
                        {
                            Symbol = rdr.GetString(rdr.GetOrdinal("Symbol")),
                            High = rdr.GetDecimal(rdr.GetOrdinal("High")),
                            Low = rdr.GetDecimal(rdr.GetOrdinal("Low")),
                            Email = rdr.GetString(rdr.GetOrdinal("Email"))
                        };
                    }
                    else
                    {
                        return null;
                    }
                }
            }
        }
    }
}
```

C# - StockApi

Notes

- Uses YahooFinanceApi to check stock price
- Returns **StockInfo** or **null**

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading.Tasks;
using TradeMinderCSharp.Entities;
using YahooFinanceApi;

namespace TradeMinderCSharp.Services
{
    public class StockApi : Interfaces.IStockApi
    {
        /// <summary>
        /// Gets the latest stock info for the given symbol.
        /// </summary>
        public async Task<StockInfo> GetStock(string symbol)
        {
            var securities = await Yahoo.Symbols(symbol).Fields(Field.Symbol, Field.RegularMarketPrice).QueryAsync();
            var stock = securities[symbol];

            if (stock != null)
            {
                return new StockInfo { Symbol = symbol,
                                       Date = DateTime.Now,
                                       Value = Convert.ToDecimal(stock.RegularMarketPrice) };
            }
            else
            {
                return null;
            }
        }
    }
}
```

C# - Unit Tests

Notes

- Expected IO data is prepared
- IO dependencies are mocked using "Moq".
- "Setup" asserts that a method is called
- "Returns" specifies a value
- "Verify" asserts the value of an argument passed to the dependency interface.
  - "Verify" must happen *after* the feature is tested or else it won't work properly! ☹

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Moq;
using System;
using System.Threading.Tasks;
using TradeMinderCSharp.Entities;
using TradeMinderCSharp.Interfaces;

namespace TradeMinderCSharp.UnitTests
{
    [TestClass]
    public class FullTests
    {
        [TestMethod]
        public void FullTest_WhenStockIsWithinThresholds_ShouldNotCreateMessage()
        {
            // Prepare return values
            var stock = new StockInfo { Symbol = "MSFT", Date = DateTime.Now, Value = 10.0M };
            var thresholds = new NotificationThresholds { Symbol = "MSFT", Email = "jmarr@microdesk.com", High = 15, Low = 4 };

            // Stub the StockApi
            var stockApi = new Mock<IStockApi>();
            stockApi
                .Setup(s => s.GetStock(It.IsAny<string>()))
                .Returns(Task.FromResult(stock))
                .Verifiable();

            // Stub the Database
            var database = new Mock<IDatabase>();
            database
                .Setup(d => d.GetThresholds(It.IsAny<string>(), It.IsAny<string>()))
                .Returns(Task.FromResult(thresholds));

            // Stub the MessageService
            var messageSvc = new Mock<IMessageService>();
            messageSvc
                .Setup(m => m.SendMessage(It.IsAny<Message>()));

            // Build up feature class
            var notifier = new StockThresholdNotifier(database.Object, stockApi.Object, messageSvc.Object);

            // Run
            notifier.CheckStock("MSFT", "jmarr@microdesk.com")
                .Wait();

            // Assert expected argument value
            stockApi.Verify(s => s.GetStock("MSFT"));

            // Assert expected argument values
            database.Verify(d => d.GetThresholds("MSFT", "jmarr@microdesk.com"));
        }
    }
}
```

F# - “Refactor to Testability”

Goals

- To demonstrate *how* to design the same feature in F#
- To show a repeatable “recipe” for refactoring an F# feature to be more testable
- To demonstrate how to architect in F# using functions vs. OOP class based design
- To show how easy and flexible it is to develop a general purpose app “line-of-business” feature in F#

F# - Progressively Testable Design

Phases

- Phase 1: Untestable
  - IO Dependencies will be called directly from feature (tightly coupled)
  - A.k.a. "*Git 'er done*"
- Phase 2: Partially testable
  - Only the core business logic will be extracted and tested
  - A.k.a. "*Best bang for the buck*"
- Phase 3: Fully testable
  - IO Dependencies will be injected
  - Should be as testable as the C# implementation
  - A.k.a. "*Overkill*" ;)

Core Concept: IO and Pure

New Terms

- **Feature** Function – A "transaction script" function that implements the feature.
- **IO** Function – A functions that represent external dependencies
  - i.e. Databases, APIs, file system, etc.
- **Pure** Function – A pure logic function that doesn't make any IO calls
  - Must be passed everything it needs to do its work
  - Must always return the same output for a given input
  - These are very easy to test

Core Concept: *IO -> pure -> IO*

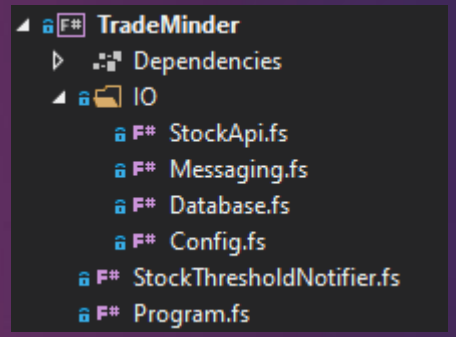
Architectural Concept

- Feature = "IO -> pure -> IO"
  - A.k.a. "IO Pure Sandwich"
  - A.k.a. "Functional Core, Imperative Shell"
- You should always start by identifying which operations in your feature are "IO" and which can be "pure" (calculations / transformations on data).
- Step 1) Get the IO data required
- Step 2) Pass it into the pure functions
- Step 3) Do any final IO (saving to database, file system, messaging, etc).

F# Implementation

Solution Layout

- IO dependencies will go in the "IO" folder
- The "StockThresholdNotifier" module will contain the **Feature** function itself (a transaction script)
- The entities are all declared within their respective modules
  - Alternatively, you could also declare them all together in one file at the top, i.e. "Entities.fs".
- Program.fs contains the "main" launch function.
- F# files are order dependent, so Program.fs is always last, despite the fact that "P" comes before "S".



F# - Program.fs

Notes

- Pattern match to destructure args array
- Async.RunSynchronously is like C# Task ".Wait()"
- F# has it's own built-in async library that can be used with C# TPL Tasks.

```
1   [<EntryPoint>]
2   let main argv =
3
4       match argv with
5       | [| symbol; email |] ->
6           Phase3.StockThresholdNotifier.checkStock symbol email
7           |> Async.RunSynchronously
8
9       | _ ->
10          printf "Invalid args.  Expected: 'TradeMinder.exe MSFT jmarr@microdesk.com'"
11
12      0
```

F# - Phase 1: StockThresholdNotifier.fs

Notes

- "checkStock" models the feature as a function.
- Not unit testable
  - Sometimes that's okay!
- Tightly coupled IO dependencies
- async block (C.E.)
  - let! ("let bang") is like C# "await"
- Note: Code is factored into "IO -> pure -> IO"
  - This will make it easier to "refactor to testability"

```fsharp
module Phase1.StockThresholdNotifier
open StockApi
open Messaging

/// This function contains the logic to run the feature.
let checkStock (symbol: string) (email: string) =
    async {
        // 1) Get data required
        let! stock = StockApi.getStock symbol
        let! thresholds = Database.getThresholds (Config.getConnectionString()) symbol email

        match stock, thresholds with
        | Some stock, Some thresh ->
            // 2) Process business rules to create an alert (or not).
            let message =
                match stock.Value with
                | value when value > thresh.High ->
                    Some ({ Email = thresh.Email
                            Body = sprintf "'%s' value $%M exceeds max: %M." stock.Symbol value thresh.High })
                | value when value < thresh.Low ->
                    Some ({ Email = thresh.Email
                            Body = sprintf "'%s' value $%M is less than min %M." stock.Symbol value thresh.Low })
                | _ -> None

            // 3) Send the message (if one exists)
            match message with
            | Some msg -> do! Messaging.sendMessage msg
            | None -> printfn "No message was sent."

        | _ ->
            printfn "Requires stock and threshold."
    }
```

F# - IO: StockApi.fs

Notes

- Entities are commonly modeled alongside related functions.
- Uses YahooFinanceApi
- Returns **Some StockInfo** or **None**
- Uses "async" expression
  - Unlike regular functions, Computation Expressions use the "return" keyword.
  - Async.await converts TPL Task to F# Async
  - *There is also a "task" C.E.*

```
module StockApi
open System
open YahooFinanceApi

type StockInfo = { Symbol: string;  Date: DateTime; Value: decimal; }

/// Gets the latest stock info for the given symbol.
let getStock(symbol: string) =
    async {
        let! securities = Yahoo.Symbols(symbol).Fields(Field.Symbol, Field.RegularMarketPrice).QueryAsync() |> Async.AwaitTask
        let stock = securities.[symbol]

        return
            if stock <> null
            then Some { Symbol = symbol
                        Date = DateTime.Now
                        Value = stock.RegularMarketPrice |> decimal }
            else None
    }
```

F# - IO: Database.fs

Notes

- NotificationThresholds entity declared in module alongside "getThresholds" function
- ConnectionString is just type "alias" for string
- "connStr" is a dependency this function needs.
  - Dependency args should always be passed in first! (you'll see why later).
- Returns **Some Thresholds** or **None**
  - This means that compiler will force caller to "unwrap" optional value and handle both cases

```fsharp
module Database
open Microsoft.Data.Sqlite
open System.Data

/// A user defined set of stock notification thresholds.
type NotificationThresholds = {
    Symbol: string
    Low: decimal
    High: decimal
    Email: string
}

type ConnectionString = string

/// Gets data info from the database
let getThresholds (connStr: ConnectionString) (symbol: string) (email: string) =
    async {
        use conn = new SqliteConnection(connStr)
        use cmd = conn.CreateCommand()
        cmd.CommandText <- "SELECT * FROM Thresholds WHERE Symbol=$Symbol AND Email=$Email"
        cmd.Parameters.AddRange [ SqliteParameter("$Symbol", symbol); SqliteParameter("$Email", email) ]
        conn.Open()
        use! rdr = cmd.ExecuteReaderAsync(CommandBehavior.CloseConnection) |> Async.AwaitTask

        return
            if (rdr.Read())
            then Some { Symbol = rdr.GetString(rdr.GetOrdinal("Symbol"))
                        High = rdr.GetDecimal(rdr.GetOrdinal("High"))
                        Low = rdr.GetDecimal(rdr.GetOrdinal("Low"))
                        Email = rdr.GetString(rdr.GetOrdinal("Email")) }
            else None
    }
```

F# - Phase 2

Notes

- Refactor to testability:
  Extract processing logic into a "pure" (testable) function.
- IO functions are still tightly coupled, but *do we really need to test those?*
  - I say **no**, because there is no branching logic around the IO (low "cyclomatic complexity").
- Benefit:
  - Business logic is easily tested
  - Code remains simple
  - *No interfaces or mocking required*
  - IO -> pure -> IO makes for easy testing of pure logic
  - "Best bang for the buck"

```fsharp
module Phase2.StockThresholdNotifier
open StockApi
open Messaging

/// Extract a pure function with all necessary data passed in as args.
let maybeCreateMessage (stock: StockInfo) (thresh: Database.NotificationThresholds) =
    match stock.Value with
    | value when value > thresh.High ->
        Some ({ Email = thresh.Email
                Body = sprintf "'%s' value $%M exceeds max: %M." stock.Symbol value thresh.High })
    | value when value < thresh.Low ->
        Some ({ Email = thresh.Email
                Body = sprintf "'%s' value $%M is less than min %M." stock.Symbol value thresh.Low })
    | _ -> None

/// This function contains the logic to run the feature.
let checkStock (symbol: string) (email: string) =
    async {
        // 1) IO - Get data required
        let! stock = StockApi.getStock symbol
        let! thresholds = Database.getThresholds (Config.getConnectionString()) symbol email

        match stock, thresholds with
        | Some stock, Some thresholds ->
            // 2) Pure - Process business rules to create an alert (or not).
            let message = maybeCreateMessage stock thresholds

            // 3) IO -> Send the message (if one exists)
            match message with
            | Some msg -> do! Messaging.sendMessage msg
            | None -> printfn "No message was sent."

        | _ ->
            printfn "Requires stock and threshold."
    }
```

F# - Phase 2: Testing a Pure Function

Notes

- Double ticks make for nice test names
- No mocking/stubbing required because pure functions are passed the data they need.

```fsharp
module TradeMinder.UnitTests.PartialTests
open NUnit.Framework
open StockApi
open Database
open System
open FsUnit
open Phase2

[<Test>]
let ``When stock is within thresholds, should not create a message.``() =
    let stock = { StockInfo.Symbol = "MSFT"; Date = DateTime.Now; Value = 10.0M }
    let thresholds = { NotificationThresholds.Symbol = "MSFT"; Email = "jmarr@microdesk.com"; High = 15.0M; Low = 4.0M }
    let result = StockThresholdNotifier.maybeCreateMessage stock thresholds
    result |> should equal None

[<Test>]
let ``When stock is greater than thresholds, should create a message.``() =
    let stock = { StockInfo.Symbol = "MSFT"; Date = DateTime.Now; Value = 20.0M }
    let thresholds = { NotificationThresholds.Symbol = "MSFT"; Email = "jmarr@microdesk.com"; High = 15.0M; Low = 4.0M }
    let result = StockThresholdNotifier.maybeCreateMessage stock thresholds

    match result with
    | Some msg -> Assert.IsTrue(msg.Email = "jmarr@microdesk.com")
    | None -> failwith "Expected a message."
```

F# - Phase 3

Make it fully testable!

- In Phase 2, we extracted only the business logic to make it testable.
- In Phase 3, we want our feature function to no longer be "tightly coupled" to the IO implementations.
  - Instead, we want to inject the IO functions to allow us to pass in mocked implementations, like the fully testable C# example.

F# - Phase 2 –> Phase 3 Refactor

- 1) Create a *Template* "higher order function"
- 2) Pass in IO functions as dependencies (using F# Type Inference)
- 3) Replace *tightly coupled* IO calls with *loosely coupled* injected functions
- 4) Recreate the feature function as a **partially applied** version of the template

Phase 2

```fsharp
/// This function contains the logic to run the feature.
let checkStock (symbol: string) (email: string) =
    async {
        // 1) IO - Get data required
        let! stock = StockApi.getStock symbol
        let! thresholds = Database.getThresholds (Config.getConnectionString()) symbol email

        match stock, thresholds with
        | Some stock, Some thresholds ->
            // 2) Pure - Process business rules to create an alert (or not).
            let message = maybeCreateMessage stock thresholds

            // 3) IO -> Send the message (if one exists)
            match message with
            | Some msg -> do! Messaging.sendMessage msg
            | None -> printfn "No message was sent."

        | _ ->
            printfn "Requires stock and threshold."
    }
```

Phase 3 (Extract to Template)

```fsharp
/// This "template" function contains the fully testable logic to run the feature.
let checkStockTemplate getStock getThresholds sendMessage (symbol: string) (email: string) =
    async {
        // 1) IO - Get necessary data
        let! stock = getStock symbol
        let! thresholds = getThresholds symbol email

        match stock, thresholds with
        | Some stock, Some thresholds ->
            // 2) Pure - Process business rules to create an alert (or not).
            let message = maybeCreateMessage stock thresholds

            // 3) IO -> Send the message (if one exists)
            match message with
            | Some msg -> do! sendMessage msg
            | None -> printfn "No message was sent."

        | _ ->
            printfn "Requires stock and threshold."
    }

/// Build up run function by partially applying dependencies.
let checkStock =
    checkStockTemplate
        StockApi.getStock
        (Database.getThresholds (Config.getConnectionString()))
        Messaging.sendMessage
```

F# - Phase 3

Notes

- I call it a template because it's like the GoF "Template" pattern.
  - Abstract class with a series of steps as abstract methods that are chained together in an implementation method.
  - Concrete classes simply implement the steps.
- **checkStockTemplate** is a "higher order function" that takes IO dependency functions as args (along with the original args).
  - *Dependency arguments (i.e. things that would be c'tor injected in C#) should always be modeled first.*
- The new **checkStock** function "builds up" the template function by *partially* applying the IO dependencies.
- IO Dependencies in the template can be substituted in unit tests.
- Fully Testable! (equivalent to C# sample)

```fsharp
module Phase3.StockThresholdNotifier
open StockApi
open Messaging

/// This pure function creates a notification (or not) based on the stock info and thresholds.
let maybeCreateMessage (stock: StockInfo) (thresh: Database.NotificationThresholds) =
    match stock.Value with
    | value when value > thresh.High ->
        Some ({ Email = thresh.Email
                Body = sprintf "'%s' value $%M exceeds max: %M." stock.Symbol value thresh.High })
    | value when value < thresh.Low ->
        Some ({ Email = thresh.Email
                Body = sprintf "'%s' value $%M is less than min %M." stock.Symbol value thresh.Low })
    | _ -> None

/// This "template" function contains the fully testable logic to run the feature.
let checkStockTemplate getStock getThresholds sendMessage (symbol: string) (email: string) =
    async {
        // 1) IO - Get necessary data
        let! stock = getStock symbol
        let! thresholds = getThresholds symbol email

        match stock, thresholds with
        | Some stock, Some thresholds ->

            // 2) Pure - Process business rules to create an alert (or not).
            let message = maybeCreateMessage stock thresholds

            // 3) IO -> Send the message (if one exists)
            match message with
            | Some msg -> do! sendMessage msg
            | None -> printfn "No message was sent."

        | _ ->
            printfn "Requires stock and threshold."
    }

/// Build up run function by partially applying dependencies.
let checkStock =
    checkStockTemplate
        StockApi.getStock
        (Database.getThresholds (Config.getConnectionString()))
        Messaging.sendMessage
```

F# - Phase 3 – Full Unit Testing

Notes

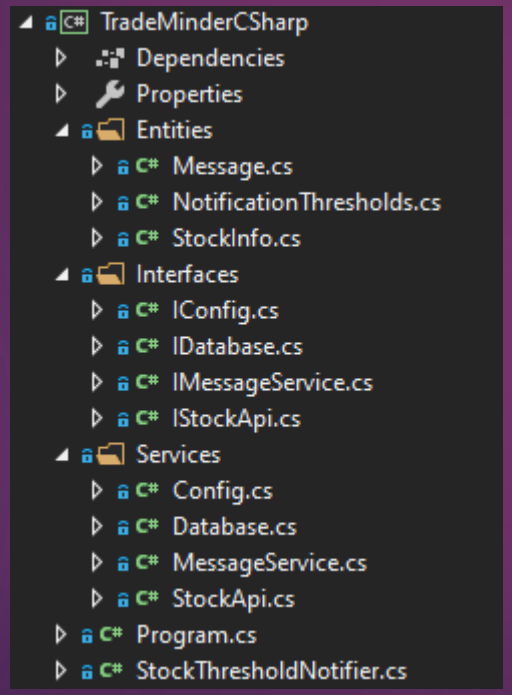
- IO dependency functions can easily be modeled as functions
  - no mocking framework required
- Testing the input args of dependencies is also very straight forward

```fsharp
module FullTests
open NUnit.Framework
open StockApi
open Database
open Messaging
open System
open FsUnit
open Phase3

[<Test>]
let ``Full test: when stock is within thresholds, should not create a message.``() =
    // Prepare return values
    let stock = { StockInfo.Symbol = "MSFT"; Date = DateTime.Now; Value = 10.0M }
    let thresholds = { NotificationThresholds.Symbol = "MSFT"; Email = "jmarr@microdesk.com"; High = 15.0M; Low = 4.0M }

    // Stub the StockApi
    let getStock symbol =
        symbol |> should equal "MSFT" // Assert expected argument value
        async { return Some stock }

    // Stub the Database
    let getThresholds symbol email =
        symbol |> should equal "MSFT"
        email |> should equal "jmarr@microdesk.com"
        async { return Some thresholds }

    // Stub Messaging
    let sendMessage (msg: Message) =
        async { return () }

    // Run
    StockThresholdNotifier.checkStockTemplate getStock getThresholds sendMessage "MSFT" "jmarr@microdesk.com"
    |> Async.RunSynchronously
```

Side-by-side: Solution Layout

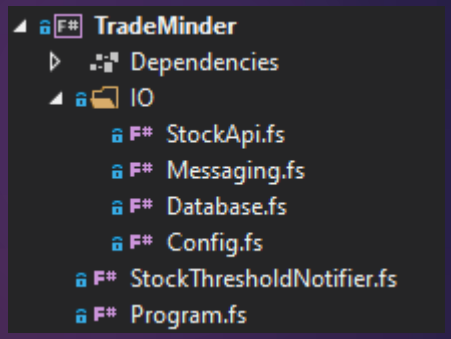
C#

- Entities are often 1-file-per-class
- Interfaces are necessary for testability; and they are often used even if no tests are written to adhere to best practices (loose coupling).
- Files listed in ABC order



F#

- Entities often live in modules alongside functions
- Interfaces are never needed for testability or loose coupling.
- Files are order dependent



Side-by-side: Database

C# - Database.cs

```csharp
using Microsoft.Data.Sqlite;
using System;
using System.Collections.Generic;
using System.Data;
using System.Text;
using System.Threading.Tasks;
using TradeMinderCSharp.Entities;
using TradeMinderCSharp.Interfaces;

namespace TradeMinderCSharp.Services
{
    public class Database : Interfaces.IDatabase
    {
        private IConfig _config;

        public Database(IConfig config)
        {
            _config = config;
        }

        /// <summary>
        /// Gets data info from the database
        /// </summary>
        public async Task<NotificationThresholds> GetThresholds(string symbol, string email)
        {
            using (var conn = new SqliteConnection(_config.GetConnectionString()))
            {
                var cmd = conn.CreateCommand();
                cmd.CommandText = "SELECT * FROM Thresholds WHERE Symbol=$Symbol AND Email=$Email";
                cmd.Parameters.AddRange(new[] { new SqliteParameter("$Symbol", symbol), new SqliteParameter("$Email", email) });
                conn.Open();
                using (var rdr = await cmd.ExecuteReaderAsync(CommandBehavior.CloseConnection))
                {
                    if (rdr.Read())
                    {
                        return new NotificationThresholds
                        {
                            Symbol = rdr.GetString(rdr.GetOrdinal("Symbol")),
                            High = rdr.GetDecimal(rdr.GetOrdinal("High")),
                            Low = rdr.GetDecimal(rdr.GetOrdinal("Low")),
                            Email = rdr.GetString(rdr.GetOrdinal("Email"))
                        };
                    }
                    else
                    {
                        return null;
                    }
                }
            }
        }
    }
}
```

F# - Database.fs

```fsharp
module Database
open Microsoft.Data.Sqlite
open System.Data

/// A user defined set of stock notification thresholds.
type NotificationThresholds = {
    Symbol: string
    Low: decimal
    High: decimal
    Email: string
}

type ConnectionString = string

/// Gets data info from the database
let getThresholds (connStr: ConnectionString) (symbol: string) (email: string) =
    async {
        use conn = new SqliteConnection(connStr)
        use cmd = conn.CreateCommand()
        cmd.CommandText <- "SELECT * FROM Thresholds WHERE Symbol=$Symbol AND Email=$Email"
        cmd.Parameters.AddRange [ SqliteParameter("$Symbol", symbol); SqliteParameter("$Email", email) ]
        conn.Open()
        use! rdr = cmd.ExecuteReaderAsync(CommandBehavior.CloseConnection) |> Async.AwaitTask

        return
            if (rdr.Read())
            then Some { Symbol = rdr.GetString(rdr.GetOrdinal("Symbol"))
                        High = rdr.GetDecimal(rdr.GetOrdinal("High"))
                        Low = rdr.GetDecimal(rdr.GetOrdinal("Low"))
                        Email = rdr.GetString(rdr.GetOrdinal("Email")) }
            else None
    }
```

Side-by-side: StockApi

C#: StockApi.cs

```csharp
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading.Tasks;
using TradeMinderCSharp.Entities;
using YahooFinanceApi;

namespace TradeMinderCSharp.Services
{
    public class StockApi : Interfaces.IStockApi
    {
        /// <summary>
        /// Gets the latest stock info for the given symbol.
        /// </summary>
        public async Task<StockInfo> GetStock(string symbol)
        {
            var securities = await Yahoo.Symbols(symbol).Fields(Field.Symbol, Field.RegularMarketPrice).QueryAsync();
            var stock = securities[symbol];

            if (stock != null)
            {
                return new StockInfo { Symbol = symbol,
                                       Date = DateTime.Now,
                                       Value = Convert.ToDecimal(stock.RegularMarketPrice) };
            }
            else
            {
                return null;
            }
        }
    }
}
```

F#: StockApi.fs

```fsharp
module StockApi
open System
open YahooFinanceApi

type StockInfo = { Symbol: string;  Date: DateTime; Value: decimal; }

/// Gets the latest stock info for the given symbol.
let getStock(symbol: string) =
    async {
        let! securities = Yahoo.Symbols(symbol)
                                .Fields(Field.Symbol, Field.RegularMarketPrice)
                                .QueryAsync() |> Async.AwaitTask

        let stock = securities.[symbol]

        return
            if stock <> null
            then Some { Symbol = symbol
                        Date = DateTime.Now
                        Value = stock.RegularMarketPrice |> decimal }
            else None
    }
```

BONUS: Domain Driven Design

Skip to Phase 3 (a.k.a. DDD)

- Domain Driven Design / TDD
  - Develop domain entities and business logic first without worrying about implementation details
- "Exploratory Coding" – Great for when you just got off a call discussing a new project, and you want to flesh out the core logic.
- I like to annotate simple args (items and couponCode), and only use type inference for function signatures.
- Type inferred args are kind of like "putty" in that they conform to whatever is pressed up against them on either side!
- When IO is implemented, you can create a "checkout" function that builds up the "checkoutTemplate" using partial application.

```fsharp
module ShoppingCart

// F# Unit of Measure
[<Measure>] type USD

type CouponDiscount =
    | Discount of decimal<USD>
    | FreeOrder
    | NoDiscount

type Order = {
    Id: System.Guid
    Items: CartItem list
    Total: decimal<USD>
}

and CartItem = {
    SKU: string
    Qty: int
    Price: decimal<USD>
}

/// Calculates the total (pure / testable function)
let calculateTotal items discount =
    let subTotal = items |> List.sumBy (fun item -> item.Price * decimal item.Qty)
    // Total:
    match discount with
    | Discount amt -> subTotal - (abs amt)
    | FreeOrder -> 0.00M<USD>
    | NoDiscount -> subTotal

/// A testable checkout template - domain logic written before IO implementations
let checkoutTemplate lookupCouponCode saveOrder (items: CartItem list) (couponCode: string) =
    async {
        // 1) IO - Get info
        let discount = lookupCouponCode couponCode

        // 2) Pure logic
        let total = calculateTotal items discount

        // 3) IO - save
        do! saveOrder { Id = System.Guid.NewGuid(); Items = items; Total = total }
    }
```

</>