Functional developer in object-oriented world

How F# helped me to write better C#

Bartosz Sokół

@bartsokol

4developers 2017

Who am I?

- Software developer with 12+ years of hands-on experience
- Used to work with many languages, including C#, F#, JavaScript, PHP, Python and more
- Charmed by functional programming, especially in ML flavour
- Occasional blogger at bart-sokol.info
- Twitter spammer @bartsokol
- Conference junkie on permanent rehab
- Hobbyist traveller and photographer

The Beginning

Object-Oriented Pain

What did we start with?

- Legacy solution with many projects in it
- Logic split into multiple projects and services
- Improperly approached microservice architecture
- Layers, layers of layers, and more layers
- Sorry, I got lost in layers
- Oh, and patterns...

Welcome to the Patternland

- Controller
- Facade
- Handler
- Provider
- Proxy
- Wrapper
- Repository

- Coordinator
- Builder
- Factory
- Decorator
- Memoize
- Grinder
- ...

Problems

- Distributed logic made the code hard to reason about
- The only thing you could expect was unexpected
- Handling errors was a pain - will it throw?
- Can it be null or not?
- Going deep vs going forward - stack overflow in my head

What to do with it?

The challenge - one feature needs to be added to our solution.

- Keep patching current code?
- Move code a bit / rewrite some parts using the same approach?
- Full rewrite using the different approach?

The alternative

Functional relief

F# Success Story

- We've created and deployed our first F# project in late 2016
- It took around 1 week to develop it
- Stateless rule engine, getting data from external services and doing calculations
- Used in production since December 2016
- 3 months passed and still no bugs or other issues raised

Lessons learned

- Explicit is better than implicit
- Implicit is better than boilerplate
- Don't fear the static
- The most important "pattern" is composition
- Functions can solve almost any issue
- Test the most important things in detail, the others on a higher level

Key takeaways

- `Result<T>` type - for handling errors
- `Option<T>` type - for representing non-existent values
- `Map` and `Bind` functions for combining calls
- FTW! |> Pipe operator -> composition
- API design - RPC-like style vs REST

Result<T> type

- It can be either a Success or a Failure
- If a function can fail, it should return Result<T>
- Result needs to be handled explicitly
- If we encounter any expected exception, we wrap it into Result (details captured in Error type)
- It's a struct with default value of Failure, so even if you take shortcuts you won't get null reference exceptions so easily

Result<T> - F#

```
type Result<'a> = Success of 'a | Failure of Error
```

Result<T> - C#

```
public struct Result<T> : IEquatable<Result<T>>
{
        public Result(T value)
        {
                Value = value;
                IsSuccess = true;
        }

        public Result(Error error)
        {
                Error = error;
        }

        public T Value { get; }
        public Error Error { get; }
        public bool IsSuccess { get; }
        public bool IsFailure => !IsSuccess;

        (...Equals and ToString overloads go here...)
}
```

Option<T> type

- It can be either Some value or None
- If a value in a data structure is optional, it is wrapped into Option<T>
- More explicit than reference and nullable types
- A bit harder to work with than nulls (no built-in language support), but it pays back quickly
- It's a struct with default value of None

Option<T> - F#

```
type Option<'a> = Some of 'a | None
```

Option<T> - C#

```
public struct Option<T> : IEquatable<Option<T>>
{
        public Option(T value)
        {
                Value = value;
                IsSome = true;
        }

        public T Value { get; }
        public bool IsSome { get; }
        public bool IsNone => !IsSome;

        (...Equals and ToString overloads go here...)
}
```

Map and Bind

Keys to composition

Map

```
Option<TOut> Map<TIn, TOut>(this Option<TIn> val, Func<TIn, TOut> mapper) =>
        val.IsSome ? Some(mapper(val.Value)) : None<TOut>();
```

- Map expects a function which returns not lifted value
- If a parameter is Some/Success, it calls mapper and wraps it in Some/Success
- If a parameter is None/Failure, it returns None/Error of the output type

Bind

```
Option<TOut> Bind<TIn, TOut>(this Option<TIn> val, Func<TIn, Option<TOut>> binder)
        val.IsSome ? binder(val.Value) : None<TOut>();
```

- Bind expects a function which returns lifted value
- If a parameter is Some/Success, it returns the result of a binder call
- If a parameter is None/Failure, it returns None/Error of the output type

Introducing Flows

Making the important things explicit

Flows

- Making the important things visible at the first glance
- Flattening the class structure
- Getting rid of messy `if` statements
- Reducing the need to name things
- Making composition and code reuse much easier

Flow - Map and Bind in action

```
public Result<OperationResultDto> Execute(OperationDto dto) =>
        InputValidation.Validate(dto)
        .Map(IntentMapper.Map)
        .Bind(_contextProvider.ProvideContext)
        .Bind(_businessValidation.Validate)
        .Map(Executor.Execute)
        .Map(SummaryCalculator.Calculate)
        .Bind(_storage.Persist)
        .Map(ResponseMapper.Map);
```

Flow - let's try without Map and Bind

```
public Result<OperationResultDto> Execute(OperationDto dto)
{
        var validationResult = InputValidation.Validate(dto);
        if (validationResult.IsFailure) return ResponseMapper.Map(...magic to map
        var intent = IntentMapper.Map(validationResult.Value);
        var contextResult = _contextProvider.ProvideContext(intent);
        if (contextResult.IsFailure) return ResponseMapper.Map(...magic to map it
        var businessValidationResult = _businessValidation.Validate(contextResult
        if (businessValidationResult.IsFailure) return ResponseMapper.Map(...magi
        var result = Executor.Execute(businessValidationResult.Value);
        var summary = SummaryCalculator.Calculate(result);
        var storageResult = _storage.Persist(summary);
        return ResponseMapper.Map(storageResult);
}
```

Map and Bind - benefits

- Extremely simple
- Reduced noise
- Less code
- They enable composability
- Once learned, can be reused for different types

What else did we need?

Set of static helper functions

- Async/Task support
- Tuple support
- Collection support
- Conversion functions between Result and Option
- Conversion functions from other types (Nullable, String, Object, etc.)

Other important decisions

Small changes that can have big impact

Decouple the data from the behaviour

- The only logic in data structure is captured within constructor
- Enables extendability and composability
- Less problems with mocking
- Makes immutability a lot easier

Use immutable data structures whenever possible

- Limits the risk of incompatible behaviour between the calls
- Makes reasoning about the code much simpler
- Makes creating illegal states much harder
- Use structs for short living data structures

Dependency Injection is not a swiss army knife

- Use it only for dependencies with side effects
- Use static references for functions without side effects
- The lower you get into class structure, the less you need DI
- Less DI => less code
- Surprisingly, makes testing a lot of things much easier

Context - capturing the state

- Context captures all the things needed to execute the flow
- Makes reasoning about the code easier
- Passing the data between the steps is much simpler
- All the data required for the operation is gathered upfront and available for further steps

C# in a world of functions

The Good, The Bad and The Ugly

C# 6 (and older) - useful features

- Importing dependencies with `using static`
- Method and property expressions
- Anonymous objects
- Tuples
- Extension methods

C# 7 features we're hoping to use

- Better support for tuples
- Pattern matching - very limited but still better than none
- Local functions
- More expression bodied members

The biggest issues with C# that we couldn't overcome #1

- Immutability is a pain - no language level support for it
- Cloning immutable objects is almost impossible
- Equality comparison for reference types requires a lot of effort
- No algebraic types makes representing this-or-this-or... types cumbersome

The biggest issues with C# that we couldn't overcome #2

- No powerful pattern matching makes code much more complicated than it should be
- No partial application and currying
- Complicated function parameter syntax
- No pipe operator means we have to name too many things
- Language version is coupled to the tooling version

Inspirations

- F# (ML) language design
- Scott Wlaschin - his website fsharpforfunandprofit.com and his presentations (especially ones about Railway Oriented Programming, Dr Frankenfunctor and the Monadster, Domain Driven Design)
- React and Redux

THE END

Thank you for attention!

Bartosz Sokół

@bartsokol

4developers 2017