"Like a selection pack of teas so you can find which ones you like" (Zoe dec 2017)

(To insert highlighted text from visual studio in word, try insert->object->"open document text" then paste, according to <http://www.c-sharpcorner.com/UploadFile/efa3cf/syntax-highlighting-in-ms-word-for-C-Sharp-sql-makes-code-very/>will stop spelling highlighting etc)

Law of primacy example and counter - teach scuba divers in training pool that if oxygen stops, put your feet down and stand up. Training safer, later not. Cf stall/spin training - training more dangerous (and there will always be more trainees than successfully trained).

This isnt safety-related training!

Possible examples for code:

* doctor/patient/operation
* ***Company personnel***
  + "Culture,  loveliness and people department" (CLaP aka HR), gonna have to get a "claptrap" in there somewhere!
  + add directors type, forget to add them to bonus code for sum type/case/switch/match example!
  + Code to predict if they will stay, replay actions with different pay/bonus rates for event driven example
  + ...or instead calc tax bill based on various actions throughout the year - current state would not be enough.
  + To allow e.g whacky calcs for multithreading example, make it Santa/Easter Bunny HQ ala AoC

Chapters

* Start procedure - procedural solution
* Object orientated
  + inheritance for better or worse, problem of constructor ordering and multiple inheritence
  + Constructor parameters enforcing constraints
* Fluid interfaces/method chaining person.setuppayroll().issuebadge().assigndesk() etc
  + you've used it before - i++, sb.Append().Append()
  + mention forward to linq
  + if not cloning each time these are ALL side effects
  + but they are possibly also monads (mention forwards to monads)!
* Interfaces - multi inheritence fix
  + use for transmitting data over apis
  + vs mapping and DTOs (PK refs vs actual links to other objects - hydrate/dehydrate)
* Dependency injection
  + Now getting SOLID
  + <https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)>
  + no constructor params
* Unit testing
* Mocks - when actions aren't immutable and/or there is scattered data access, mock often has to hold state and be nearly as complex as original code!
* Repository/load up front vs scattered data access
* Back to the future - static/extension methods
* No side effects, no mutation (externally visible or state holding at least) - https://en.wikipedia.org/wiki/Pure\_function
  + void is a code smell,
  + "One-way side effects"(tm) - e.g logging;how is persistent storage not a sideeffect? (see I/O section in https://en.wikipedia.org/wiki/Pure\_function
  + elm (& haskell) : <https://en.wikipedia.org/wiki/Monad_(functional_programming)#The_I.2FO_monad> approach using events/actions/promises?
* Attack of the clones - Immutable programming
  + you do it all the time! -  x=x+7-3; msg="hello"+" " + "world";
  + ref is a horrible immutability hack
  + tuple return and unpacking make out params more natural
  + f# for fun and profit
  + types of immutability - Jon Skeet video "The changing state of immutability" https://www.youtube.com/watch?v=O89-zG84QK4 & Eric Lippert's list including "popsicle immutability" https://blogs.msdn.microsoft.com/ericlippert/2007/11/13/immutability-in-c-part-one-kinds-of-immutability/
  + example of multithreading being enabled by this approach
* Functional (ish) programming – Linq
* Functional principles - Julian’s list of functional concepts (see <http://www.tomharding.me/>) :
  + Setoid: things with a notion of equivalence
    - A ~~collection of things~~ type which understand which items are “equivalent” (where “equivalent” is not necessarily “equals”) e.g. users identified by credit card number, irrespective of what name/spelling or address they gave at time of order. See also fruit union example at <https://hackage.haskell.org/package/setoid-0.1.0.0/docs/Data-Setoid.html>
    - Any reference type object in c# is a setoid as the base Object type has an Equals() method.
    - What it’s not : A function/delegate/method is not a setoid. There can be a collection of them, but a function doesnt have understanding of what it is equivalent too - apart from exhaustive trial and error, how could it determine that f(x) => x² is equivalent to f(x) => x \* x ?
    - c# mashes this because a function/delegate does have an Equals() method but it’s reference equals only.
    - Explain the name : setoid - things that can be in a “set” or collection (lets not get too strict about what a “set” is)
  + Semigroup: things that can be squashed together (“hello” ++ “ world”)
    - A collection of setoids together with an associative operation. Associative means (x.y).z==x.(y.z) but not necessarily x.y==y.x (that’s commutative).
    - <https://en.wikipedia.org/wiki/Semigroup>
    - <http://blog.ploeh.dk/2017/11/27/semigroups/>
    - (Note – floating point addition is NOT associative - <https://www.quora.com/Is-floating-point-addition-commutative-and-associative>)

C# interactive window in visual studio:

> 0.001f+0.001f

0.002

> 0.002f+0.003f

0.005

> 0.001f+0.003f

0.004

> 0.001f+0.004f

0.00500000035

> (0.001f+0.001f)+0.003f==0.001f+(0.001f+0.003f)

false

>

* + - An example is positive integers (1,2,3….n) together with addition (this example is also commutative).
    - Another is string concatenation (NOT commutative) :(“hello”+”there”)+”world”==”hello”+(“there”+”world”) but “hello”+”world”!=”world”+”hello”
    - Another example is (0) and multiplication, addition or subtraction!
    - What it’s not: The collection must be “closed” for the operation - performing the operation on 2 items from the collection must have a result which is also in the collection - {1,7,8,12} together with addition is not a semigroup. 7+8=15, which is not in the collection. But changing 12 to 15 doesn’t help because 8+15=23 which isn’t in the collection
    - Positive integers and subtraction is not - not closed
    - All positive and negative integers and subtraction is not - not associative
    - Explain the name: it’s not as strict as a “group”!
    - Deep dive : A “group” en.m.wikipedia.org/wiki/Group\_(mathematics) has more rules to be followed, but a semigroup has more rules than a “magma” https://en.m.wikipedia.org/wiki/Magma\_(algebra) (no need for associativity)
  + Monoid: things that are semigroups but also have a special “empty” value
    - An example is non-negative integers (0,1,2…n) together with addition – 0 is the identity value for addition. Another example is positive integers (1,2,3..n) together with multiplication – 1 is the identity element for multiplication
  + Functor: things that you can map (map accepts a function that can operate on the functor’s “content” in some way that conforms to the functor laws).
    - <http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html>, links to <http://learnyouahaskell.com/a-fistful-of-monads>
    - <http://www.codefugue.com/haskell-in-c-sharp-functors/>
    - is new c# ?. operator a functor fmap???
    - Nullable is NOT - you have to know to use hasvalue and do your own protection, so it is not “maybe” type.
  + Applicative: things that have an apply function which allows you to apply a function stored inside the structure, to a value stored inside the same structure. Honestly this is useful.
  + Monad: has a bind function that allows you to effectively chain monadic operations. This is the superpower that applicative lacks. Bind accepts a Monad x and a (x -> Monad y) function.
    - <https://mikhail.io/2016/01/monads-explained-in-csharp/>
    - <http://mikehadlow.blogspot.co.uk/2011/01/monads-in-c1-introduction.html>
    - <https://importantshock.wordpress.com/2009/01/18/jquery-is-a-monad/>
* Functional languages - f#, elm, clojure, haskel
* Simply the best?
  + Julian's AoC 2017 8 lines functional, immutable vs more lines, mutable c#?
  + There's no black or white, embrace the grey!

(Not sure where):

* Comments dont compile
  + explanatory method and variable names
  + add vars/methods to improve clarity
* YAGNI
  + Optimisation is a feature - speed, memory use, lines of code, 'coolness', clarity (ok maybe this is always required)
  + Recursion - implement with Queue
  + Multithreading? (even if immutability makes it easier sometimes) - see
  + <http://adventofcode.com/2017/day/18>
  + (part 2)/ https://github.com/Kolossi/AdventOfCode2017/blob/40681aeda021f7899b4a80b49ead79865226fa32/Runner/Day18.cs#L244.  Generally whatever we are writing has many instances (web pages, api servers etc).  So do \_we\_ really need to enable parallelisation at our level, no matter how cool it feels to sprinkle async liberally in the code without understanding it?
    - give code example of bad async await usage : await ThingThatTakes10Secs();
    - await ThingThatTakes10Secs(); takes 20 secs!!! - define asyncs first, then AwaitAll or no parallelism!!!
* Minimise copy/paste - code reuse
* Dependencies
  + stopping build dependencies with copy/paste interface defs doesnt prevent actual definition dependency
  + loose coupling from sensible seperation & high cohesion in design of modules, not just breaking build dependencies
  + <https://en.wikipedia.org/wiki/Cohesion_(computer_science)>
  + /
  + <https://en.wikipedia.org/wiki/Loose_coupling>
* Com/dcom/remoting/webservices/wcf/RESTful/ajax/ajaj(json)?
  + SOAP will save us - until it gets dirtied by everyone thinking they have to send xml documents over it
  + REST (https://en.wikipedia.org/wiki/Representational\_state\_transfer) vs immutable
    - Possibly REST is like shallow immutability? - no context about request is held, but underlying objects may be changed.
    - (GETs at least) "indicate cacheability" (see caching chapter), quote from Roy Fielding on wiki page (
    - <https://en.wikipedia.org/wiki/Representational_state_transfer#Architectural_properties)>
    - Won't go into REST in this book, my snoring would put you off reading
* Microservices
* MER/"the method"
  + 3 level architecture again
  + (Arrogant?) assumption that volatility can be encapsulated in just one area
* 3 level architecture
* "If" & "null" are evil/code smells
  + bring in sum type/discriminated unions/OneOf
  + equivalent of enforcing functionality with polymorphism/abstract possible case/switch/match but also function pattern matching, haskell Typeclass (roughly like an interface)/clojure multi-methods  - see Julian conversation below.
* Event based programming - rewind/play (after no side effects section)
* Caching - easier without side effects
* State machines? \_ALL\_ side effect!
* Do AoC style challenges with textual results? (maybe e.g from a pseudo code functional description to procedural?)

Julian conversation:

psweeney [13:29]

ok, so can I ask you a completely random coding question?

say I'm doing non-oo coding using union types / sum types / discriminated unions /whatever-they-are-called, and trying to get a polymorphism style of behaviour, say I have a

`type MyType = foo | bar | pee` (I'm writing this elm-style but not a language specific question)

and I want each of this type to have a behaviour `DoThing`

would you define a DoThing on MyType which then does a case and calls the DoThing which you've also defined on foo,bar and pee or would you push the case up to the point of call?

A long question I know and no-doubt no correct answer, I'm just wondering what the thinking is - e.g. in the elm code at trp?

julianjelfs [13:40]

So if I understand you correctly you want to pass an instance of MyType into a function called DoThing and have it do something different in each case?

psweeney [13:41]

I guess yes that's the right way of saying it :slightly\_smiling\_face:

julianjelfs [13:41]

so I think that \_is\_ a little bit language specific ...

haskell is nice because you can actually patterns match in the function declaration e.g.

 ```doThing :: MyType -> Whatever --definition

doThing Foo = fooversion

doThing Bar = barversion

doThing Pee = peeversion

```

and you can tell the compiler to fail if you are left with a partial function (i.e. a function that does not cover all cases)

with Elm, the pattern match would be pushed into the function body, but basically the same

psweeney [13:44]

I did wonder that, but thought about the case where one not defined but as you say compiler would pick it up - doh!

julianjelfs [13:44]

there is a related question ...

psweeney [13:44]

so for that useage, you'd never need case?

julianjelfs [13:45]

what if I want to say that there is a capability `doThing` to be provided by `ThingDoer` types

and I want a function that depends on its argument being able to `doThing`s

in this case Elm is going to fall short but not haskell

 ```needsAThingDoer :: ThingDoer a => a -> Whatever

```

meaning give me any kind of `a` as long as it's a thing doer

In this case ThingDoer is a Typeclass which is like an interface

psweeney [13:47]

ok, looks good, so in c# world, we'd use an interface IThingDoer. This sounds haskell specific though?

is there a more general answer for the functional languages f#/clojure/haskel etc, or is it all language specific?

julianjelfs [13:48]

It's not exactly haskell specific. But it is something that Elm and F# notably lack. It means that Elm and F# only actually have parametric polymorphism rather than the more powerful \_constrained\_ parametric polymorphism

psweeney [13:49]

so can you use the pattern matching function declaration in f# etc?

julianjelfs [13:49]

Not 100% sure

psweeney [13:50]

k. If you can, you'd rarely need the case thang, which I thought was a big selling point of union types?

julianjelfs [14 minutes ago]

On this particular point - the selling point is being able to exhaustively pattern match whether it be via the case statement or destructuring in fn definition. (And also being able to model data more correctly)

julianjelfs [13:51]

yes it's true that I use `case` a lot less in haskell than I do in Elm

psweeney [13:51]

thanks. sorry this must all seem very random, just trying to get my head around it all and the advantages and disadvantages of using different coding approaches as a kind of side project which I may write up some time

julianjelfs [13:52]

yes it's interesting stuff. Clojure is a little bit different again, it has something called multi-methods to do this kind of dynamic dispatch. But it's dynamic so really very different.

psweeney [13:52]

triggered by your elm talks tbh :thumbsup::skin-tone-2:

i started thinking I'd write a kinda white paper, but as I work through it, it's more like a massive multi-chapter book now!

julianjelfs [13:53]

the more I study this stuff the more I think that it is most fundamentally about composition

whatever you are doing there is something like a -> b -> c etc

and in order to be able to think of that as a -> c you \_have\_ to abide by certain rules

hence the preoccupation with purity

psweeney [13:56]

the pure approach is \_obviously\_ so right when you think about it .... but more in theory than practice IMHO - as seen in AoC it's often quicker, easier/simpler(don't!) and produces more readable code to go all mutable and side-effectsy. I'd like to maximse the purity-ness though, trying to find the "best" (whatever that means) middle way

anyway I've taken a lot of your time approaching xmas hometime. The Aoc puzzles and the stuff you/tim have done on there are very thought-provoking I'm getting heavily into the code thinking though, so I may bug you again in the new year. (edited)

I hope you and all yours have a great one

julianjelfs [14:00]

My point is that \_there is no middle ground\_ unfortunately though

well there sort of is

what I mean is that as soon as the laws are broken, composition is broken and a lot of the elegance disappears

but I do agree that it isn't always easy to see how to proceed in this style

but it gets easier with practice

i.e. I don't think it is unnatural, I think it is just unfamiliar

psweeney [14:03]

lol, and \_that\_ right there, is the debate I'm having with myself. Elegance is a virtue, but isn't (necessarily) a software requirement, so is the fact the elegance disappears \_always a problem - there's obviously a lot of cases where it IS and has implications for maintainability long term etc. I take your point on familiarity, but if it isn't easy to quickly pick up, it isn't as easily maintainable long-term as teams change?

I'm always sceptical of black or white being the answer in either direction though, flexibility to situation is key- embrace the grey!

julianjelfs [14:04]

I can only suggest that you seriously try it! I'll help :slightly\_smiling\_face: I genuinely believe there is no trade-off here.

psweeney [14:06]

Thanks I fully intend to! I'm actually re-awakening my interest of development which was waining a bit by stepping up/out to thinking at a more meta level about it!

julianjelfs [14:07]

OK well good luck with the rest of AoC and try to find time for Christmas too!

psweeney [14:08]

oh yeah, I'm resigned to the fact that pezza will take me as I can't match his early starts, and I'm not going to be doing it 7am on Christmas, maybe 3pm after lunch!

All the best Julian cu next year