Recursive Iterative Design By Example

Paul Tarvydas, April 2021

Problem Overview

- I need to do my Y/E accounting, and,
- I need to do my personal taxes
- Business year is Aug-July, personal year is Jan-Dec
  - Two different time spans, same data
- Multiple transactions in one bank statement, with mostly same month & year, but day is different in each transaction (some transactions occur on the same day)

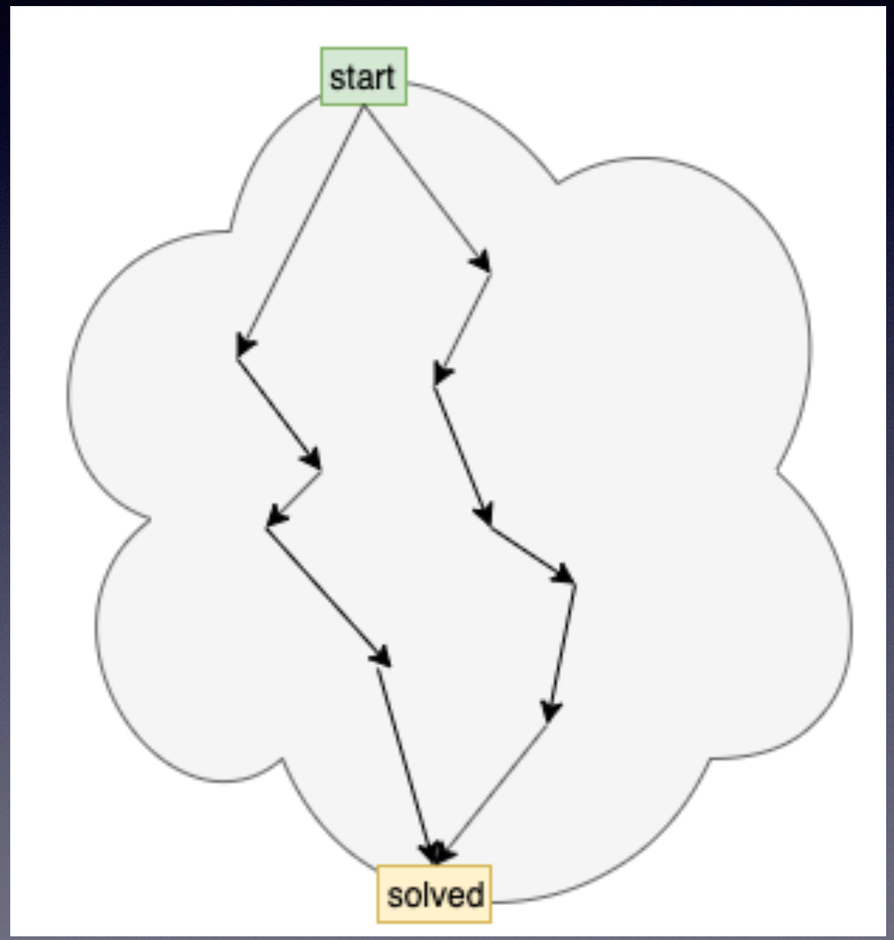
Top Down, Bottom Up

- Start at Top and work down, but be ready to dive into Bottom and work up
- Break all parts of the problem down into chunks
- Rule of thumb: fail fast - attack known/unknowns first and recast them as sets of known/knowns, or fail and re-cast the solution
- Cannot attack unknown/unknowns, because I don’t know what they are, but, I can “plan” for them
  - I can plan for changes, but I don’t know what the changes will be

Top Down Bottom Up (con’t)

- The problem space is big
- A *solution* is a traversal from one end of the space to the other end (also called “Drunkard’s Walk” or “Brownian Motion”)
- Usually, there are many solutions for the same problem
  - pick one solution, by meandering through the space

Multiple Possible Solutions



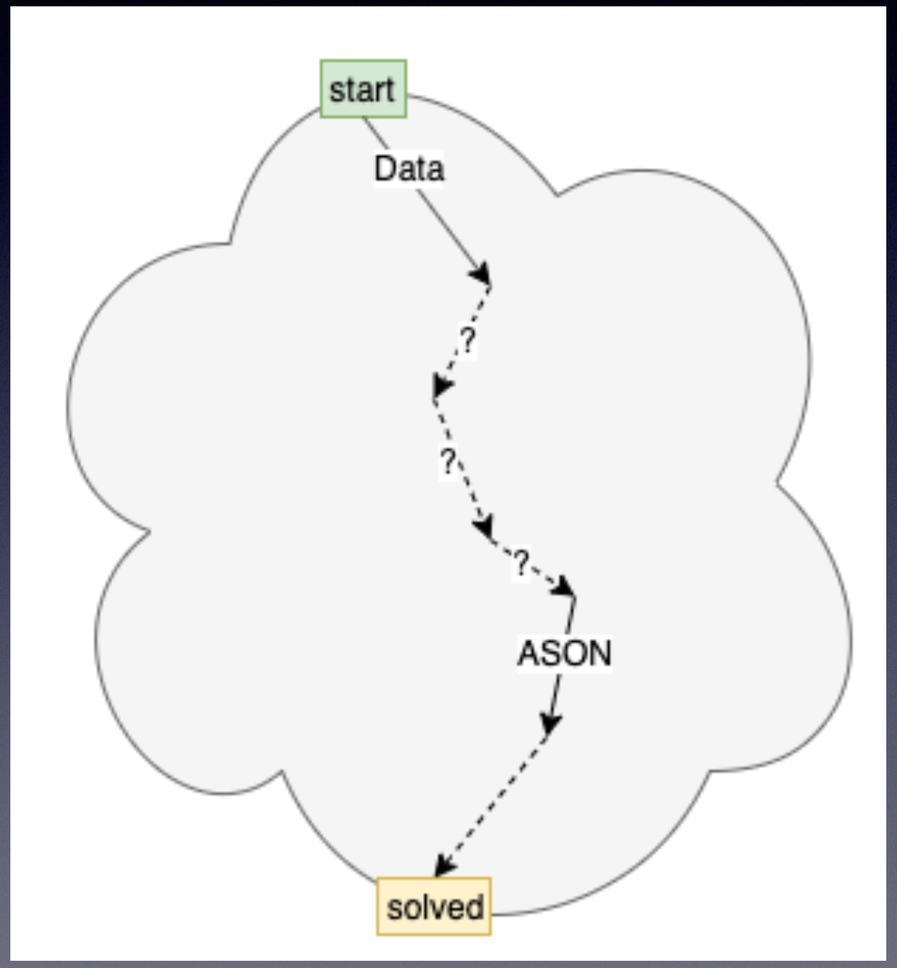
Unknown Unknowns

- Approach - Assume that I will discover something new about the problem as I progress
- I.E. I don’t know everything, yet
- Approach - do *something* and rely on the Power of Procrastination - new aspects of the problem will arise as I progress
- Approach - *assume* that I’m going to throw away my first few attempts, or, modify them
- Automate as much as possible, to mitigate effects of new sub-problems
  - When I learn something new about the problem, I can address it,
    - then, update *everything* incorporating the new information by re-running scripts (automation)

Abstraction vs. Top Down

- I can’t deal with too much abstraction, so,
  - pin something down
  - flip to bottom-up design
  - flip-flop between Top-Down and Bottom-Up
- bottom up design will not solve the problem, but will give me something to sink my teeth into
- first guess: I will use ASON as the low-level format for my data

Flip-flopping Between Top Down and Bottom Up



ASON

- ASON was designed by Carl Sassenrath, he developed the Amiga O/S and REBOL
- REBOL influenced the design of JSON
- ASON is like a new-and-improved JSON
- I have built an ASON parser (see appendix), so I know that it, minimally, can be parsed
    - ASON is a “known” to me

DI

- Other DIers may choose to solve the problem differently
  - depending on their own set of knowns
- DI means Design Intent
- It Is The Architect's Responsibility To Make A Design Clear To The Reader

Top and Bottom

- The “top” is my data
- The “bottom” is ASON
- I don’t yet know what the middle is, but it will come as I proceed
  - Q: Is this just accumulated experience?
  - A: Yes, the *belief (knowledge, reliance upon)* is experience-based. It has happened many times. I know that I can rely on being able to solve a problem by deferring the problem and “not forcing” a solution.

Experience

- My experience says that I can rely on Brainstorming
- My experience says that I can rely on the Power of Procrastination
  - I will continue to roll the problem around in my brain until some way to solve it - simply - occurs to me
  - This may take hours/days/weeks, I don’t know in advance and can’t schedule it.

Experience (Con’t)

- The More You Know, The More Creative You Can Be (Pat Pattison)
- Being a *generalist* helps to *think outside the box* and to solve problems that look impossible (at first)
- I studied Physics, EE, guitar, golf, songwriting, astronomy, computer building (wire-wrapping, taping PCBs), guitar stomp-box building, speaker cabinet building, weaving sashes, silk-screening, photography and developing photographs, audiophile, slot cars, cooking, 3D printing, C, Common Lisp, Eiffel, compiler technologies, operating systems, embedded systems, RTOSs, many programming languages, …

Power of Procrastination

- aka “Shower Time”
- aka “Sleeping on It”
- I don’t need to *think* everything through before starting
- I start doing *something* and will learn new things about the problem as I go
- The hard-earned work is my thinking and my data

Power of Procrastination - Do Something / Anything

- I start doing *something* and will learn new things about the problem as I go
- the “secret” is to never allow myself to be backed into a corner
- I assume that I will throw my work away and start again
- This assumption changes my overall approach

Code is Cheap, Thinking is Hard

- The hard-earned work is: my thinking and my data
- The “easy” stuff is my code
- I use automation and expect it to re-generate my code
- I use toolbox languages instead of calcifying my thoughts in strongly-typed languages
- The design might change as I go
- The type system design might change as I go

Brainstorming

- I learned many techniques for Brainstorming while learning about songwriting
- I have come to understand that I can explore unknown territory via brainstorming
- Brainstorming will result in *something,* but I don’t know what yet (nor do I care)
- My favourite tool for brainstorming is Scapple®

Brainstorming (con’t)

- I can do *some* brainstorming “in my head” without crutches
- Some people can think more deeply than I can, but, they reach a limit
- Sometimes a solution is achieved before the thinking-limit is reached & sometimes not
- Some people use Strong Typing to perform brainstorming
  - They try to express the type system for the solution, without writing code but writing code for the type design
  - This, similarly, gets them to find new aspects of the problem-at-hand

Strong Typing - Help or Hinder?

- Strong Typing can be a form of Waterfall design
- Type design as brainstorming is good
- Forced typing by decree is waterfall (Bible thumping)
  - finalizing a type system before the design is finished assumes too much up-front knowledge —> waterfall
- recursive *iterative* design means that any aspect of the design — e.g. type system — might be re-designed several times

Strong Typing (con’t)

- Static Typing is: an interpreter that can be run earlier than the rest of the app
- At least 3 timeframes for typing - Static, Dynamic, Never
  - Dynamic typing can be Strong, but is done at runtime
- Typing was originally meant as a crutch (optimization) for compiler writers

Typing (con’t)

- Common Lisp allows deferred typing
  - types can be added later to generate better code and to tighten up the design
- ML-like languages infer typing
- Haskell-like languages attempt to do all type checking statically,
  - and, make flexibility/language trade-offs to achieve static analysis

Nesting (Scoping)

- I don’t want to discard any data, since I expect that I will change my plan
- I use a mix of .CSV files (downloaded from the bank) and manual entry (data that I forgot to download and can’t access) to create my data and check it against paper and .PDF statements
- Some data is too repetitive to be entered manually, reliably
- I will *nest* certain slowly changing information, e.g. *statement dates* (there are many transactions in each statement) and *year* and *from-account (debit/credit)*

Nesting (con’t)

- Data “inherits” attributes declared in outer scopes (nested)
  - e.g. { year : xyz [[…] […] …] } — all transactions (can) inherit the *year* declared in the outer scope

Nesting - Static vs. Dynamic

- static is “better” than dynamic, but not important enough to change a notation
- “static” means that I can do an edit-cut/copy/paste operation on the source text,
- whereas “dynamic” means that I have to wait until runtime to find out what I can copy/paste
    - dynamic —> cut/copy/paste on a data structure instead of source text

Transpilation

- I am careful to choose notation that I can easily transpile
  - for example, inheriting the year can be handled by a simple textual rewrite (1. every time we encounter a *year* variable, push it onto a stack, 2. every time we encounter a *mm-dd* date, rewrite it as yyyy-mm-dd using the top of the stack)
  - [N.B. it might be fruitful to create many stacks, one for each kind of inherited item, e.g. a year-stack, a statement-date stack, a from-account-stack, etc. Or, one can use dynamic scoping (as per early Lisps) …]
  - [N. B. stacks —> concatenative language design]

Objects and Bags

- I will use ASON Objects { $name_1$: … $name_2$: … …} to represent named entities
- I will use ASON Arrays as bags [… … …]
  - the concept of Array is too low-level and implementation-oriented
  - I don’t care if the items are indexed
  - The main feature I want: collection of unnamed items
- Bags contain any kind of data recursively, e.g. bags can contain constants, bags, objects
  - e.g. [1 2 3]
  - e.g. [1 {name: 2} 3]
  - e.g. [1 [2 3 {$name_1$: 4} {$name_2$: 5} 6] 7 8 9]

Dates

- In my data, the *year* changes slowly, so,
- I will implement *mm-dd* dates in addition to full ASON dates
  - saves repetitive entering of the year
    - humans are bad at repetition
    - computers are good at repetition
- I will “inherit” the *year* from outer-nesting, and prepend it to all *mm-dd* dates

First Cut

- I use ASON datatypes for my data (top)
- I use a mix of ASON objects and ASON arrays to group my data (bottom)
- Each transaction (line item on the statement) is an ASON array […]
- Each group of transactions is an ASON arrary [[…] […] …]
- I will scope and inherit *statement* (date), *year* (integer) and *from* (word)
- I will include *balance* as a named field in statements (as appropriate)

Overview of 1st Cut Data Format

{ from: *bank-name*
  { statement: *yyyy-mm-dd*
    { year: *yyyy*
      [
        [*mm-dd category $mm.nn comment*]
        [*mm-dd category $mm.nn comment*]
        …
      ]
    }
    balance: *$mm.nn*
  }
}

Overview of 1st Cut Data Format (con't)

- where *category* is a single ASON *word*
- where *comment* is a single ASON *word* or an ASON *string*
- where *bank-name* is a single ASON *word*

Overview of 1st Cut Data Format (con't some more)

- remember: this is just the first attempt, things might change
- the data is entered and is the starting point for any automation
- the data format has been chosen so that no information is lost
- I can re-format the data, using automation, as I see fit (as the design progresses)
- The raw data format is "constant" and is chosen to be machine-readable (e.g. using PEG)
- I use CPU power to rewrite the data as needed, but the original data will not mutate

Flexibility vs. Waterfall

- Having chosen this all-inclusive format, I am free to experiment with different (automated) rewritings, knowing that nothing will ever destroy the original information.
- I may waste my own time, or waste CPU power, but that is less important than having a solid footing for the starting point.
- This is part of the “plan”. This arrangement allows me to change my mind (multiple times, if needed) knowing that the hard work (entering and collecting data) will be reusable.
- Goal: I will never be boxed in.

Flexibility vs. Waterfall (con't)

- I can throw all of my code away and re-start, if I discover that that is the best choice.
- I like Lisp because it lets me be cavalier with my code.
- I dislike C because it makes me calcify my decisions early in the process
  - Calcification is a hidden form of *waterfall* design
  - I imagine that Rust has this same problem, as do most statically-typed languages
- Normalizing data achieves design flexibility (I favour triples)

WIP

- I’m not finished yet
- I hope to give periodic updates on progress
- periodic means days (not months/years)

Appendix

- ASON parser (7 layers) https://guitarvydas.github.io/2021/04/10/ASON-Notation-Pipeline.htmltriples
- factbases https://guitarvydas.github.io/2021/01/17/Factbases.html
- triples https://guitarvydas.github.io/2021/03/16/Triples.html
- knowns and unknowns https://guitarvydas.github.io/2021/04/02/Recursive-Design-Knowns-and-Unknowns.html
- DI https://guitarvydas.github.io/2021/04/11/DI.html
- Scapple https://guitarvydas.github.io/2021/04/02/PEG-Cheat-Sheet.html

Appendix PEG

- https://guitarvydas.github.io/2020/12/27/PEG.html
- https://guitarvydas.github.io/2021/04/02/PEG-Cheat-Sheet.html
- https://guitarvydas.github.io/2021/03/24/REGEX-vs-PEG.html
- https://guitarvydas.github.io/2021/03/19/Racket-PEG.html
- https://guitarvydas.github.io/2021/03/17/PEG-vs.-Other-Pattern-Matchers.html
- https://guitarvydas.github.io/2020/12/09/OhmInSmallSteps.html