Programming Simplicity - Observations - Sampler

Paul Tarvydas

Programming Simplicity - Observations - Sampler

Paul Tarvydas

This book is for sale at
http://leanpub.com/programmingsimplicity-observations-sampler

This version was published on 2023-04-28

Table of Contents

The ground truth is *programming*, not *coding*.

The goal of programming is to control a machine.

A secondary goal of programming is to make it possible to change a program easily, or, to steal parts of a program and use them in other programs.

A tertiary goal of programming is to buff and polish specific notations, like textual programming languages.

Buffing and polishing code notations is a sub-goal of programming, and is, therefore, not as important as the main goal: controlling a machine. Coding - writing textual scripts in a programming language - is but a subset of the main goal.

What is the point of programming?

To control a machine.

To accurately break down an action in steps so small that even a machine can perform the steps.

Current electronic machines provide us with a set of steps called “opcodes”.

Call Return Spaghetti

Introduction

In this essay, I show that a diagram of a Call/Return system makes less sense than a diagram of a concurrent system.

I show the fundamental operation of a concurrent system and argue that it is inherently simpler than a system based on Call/Return.

Simple System

Fig. 1 contains a diagram of a simple system.

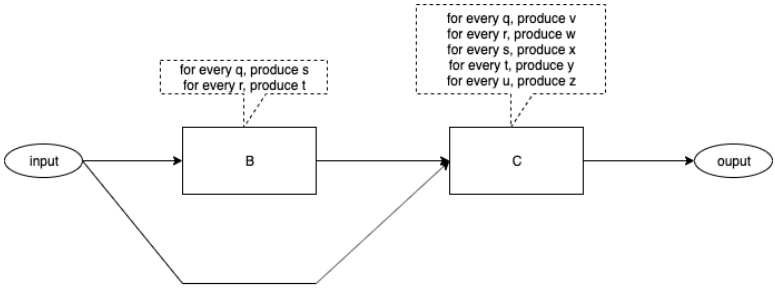


Fig. 1 A Simple System

The diagram contains one input[1] port and one output[2] port.

The diagram contains two components[3]. The algorithms for the boxes are straight-forward[4]. The algorithms are stated in terms of what each box outputs when inputs arrive at that box.

[1]The oval labelled "input".
[2]The oval labelled "output".
[3]Boxes labelled B and C.
[4]See the dashed callouts pointing to each box.

The flow of data within the diagram is shown by arrows.[5]

It appears that we have plugged two software components together to form a system.

What Happens?

What Happens When Events Arrive?

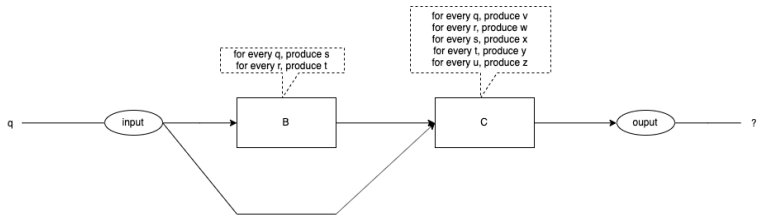


Fig. 2 An Event Arrives

What happens when an event arrives at the input? See Fig. 2.

The event, “q”, is injected into the input.

The algorithms specify exactly what each box does for any given input.

What is the expected output?

Do we see the expected output every time?

Do we see the expected output for every coding of the diagram?

Current State of the Art

The Code for Components B and C

We can implement the diagram in pseudo-code.

[5]The input flows to B and to C. B’s output flows to C. C’s output flows to the output.

Function B

```
function B(in) {
  if (in == q) {
    call C(s)
} else if (in == r) {
    call C(t)
} else {
  FatalError()
}
```

Fig. 3 Function B

Function C

```
function C(in) {
  if (in == q) {
    output <- v
 } else if (in == r) {
    output <- w
 } else if (in == s) {
    output <- x
 } else if (in == t) {
    output <- y
 } else if (in == u) {
    output <- z
} else {
   FatalError ()
}
```

Fig. 4 Function C

Code Version 1

Version 1 of the code might call component B first:

```
main () {
  call B(q)
  call C(q)
}
```

Fig. 1 Code Version 1

Code Version 2

Version 2 of the code might call C first:

```
main () {
  call C(q)
  call B(q)
 }
```

Fig. 1 Code Version 2

Final Output

Final Output The final output of the preceding routines depends on which version of the code we use.

Version 1 results in the following code path:

```
main {
 call B(q)
   B calls C(s)
     output <-- x
   C returns to B
  B returns to main
 call C(q)
    output <-- v
    C returns to main
}
main done
```

Fig. 1 Final Output for Version 1

The final output for Version 1 is x,v[6].

While version 2 results in the following code path:

```
main () {
  call C(q)
    output <-- v
  call B(q)
    B calls C(s)
      output <- x
      C returns to B
    B returns to main
}
main done
```

Fig. 1 Final Output for Version 2

The final output for Version 2 if v,x.

Version 1 and version 2 create different results.

[6] in left to right order

Control Flow

Fig. 5 shows the control flows for code versions 1 and 2.

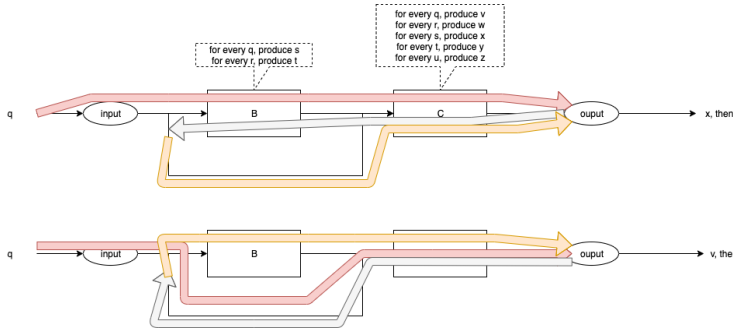


Fig. 5 Control Flow for Versions 1 & 2

The Desired Outcome

We want to plug software components together.[7]

We want the diagram(s) to mean exactly one thing.

We want the diagram(s) to mean the same thing every time.

This is possible.

I will show the event flow that we desire, in a series of diagrams, then, I will discuss how this flow can be achieved[8].

[7] After all, computer (digital) hardware is plugged together.
[8] Even on synchronous operating systems.

Event Delivery 1

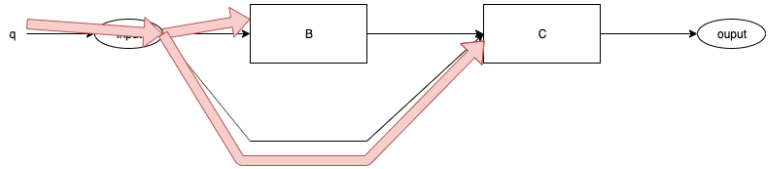


Fig. 5 Event q Delivered

Fig. 5 shows event “q” being delivered to B and C.

Nothing else happens, no routines are called.

After Event Delivery 1

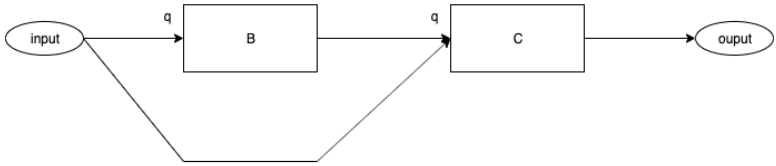


Fig. 5 After Event Delivery 1

Fig. 5 shows what the system looks like after Event Delivery 1 has occurred.

Both, Components B and C have an event “q” at their inputs.

(Neither Component has acted yet).

Two Possible Control Flow Paths

At this point, two control flow paths are possible:

1. Component B runs first.
2. Component C runs first.

I will draw a sequence of diagrams for each path.

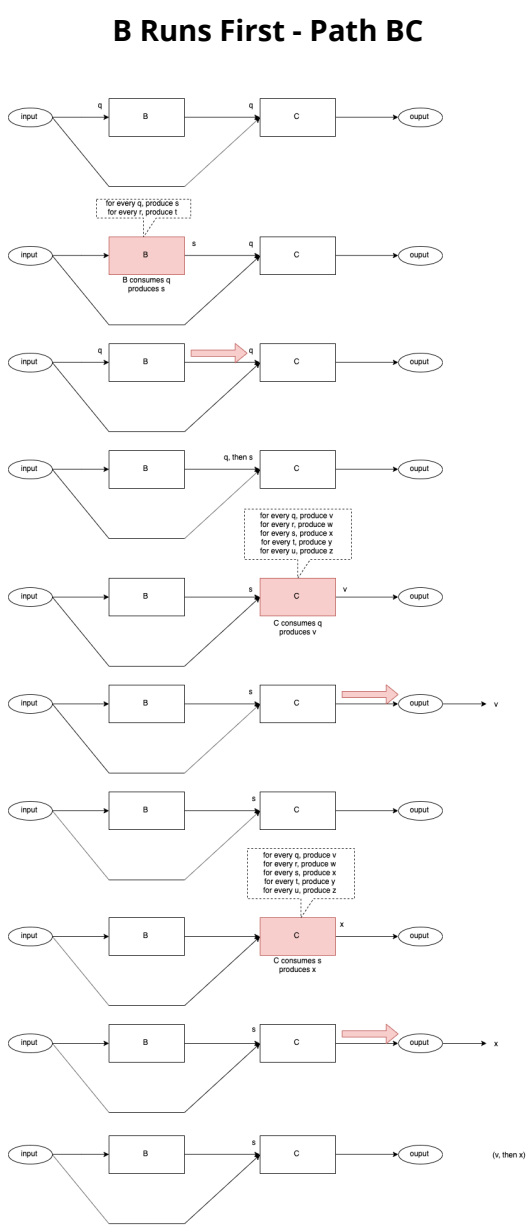


Fig. 6 Control Flow BC

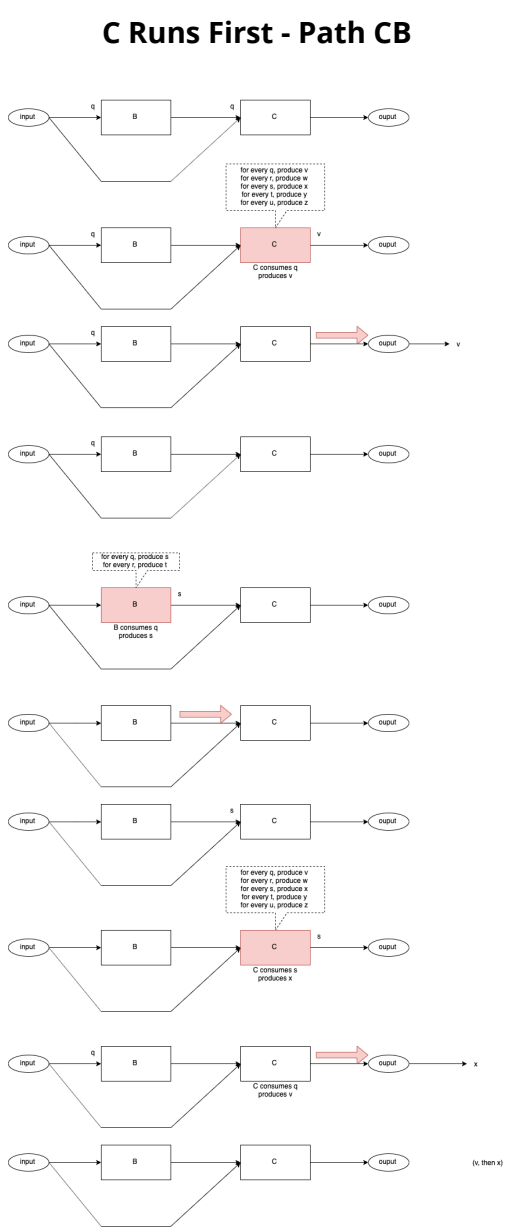


Fig. 7 Control Flow CB

Final Result

In both cases, Path BC and Path CB, the final result is the same - v is output first, then x is output.

Achieving the Desired Result

Requirements

- All Components have an input queue, onto which incoming events are placed.[9]
- Components cannot call one another.
- Components are asynchronous.
- A Dispatcher routine decides which Component will run and in what order[10].
- Components are ready if they have events in their input queues.
- Components consume one input event and produce as many output events[11] as necessary in reaction to the input event.
- Components perform a co-routine[12] dance with the Dispatcher. When a Component has processed a single event to completion, the the Component yields to the Dispatcher. The Dispatcher decides which Component will run next.[13] Components do not decide on the order of dispatching (as with call-return based code) nor can Components rely on any certain dispatching order.[14]

[9] In a production version, Components also have output queues. That requirement is a fine point, discussed elsewhere.

[10] The order is arbitrary. Components are fully asynchronous.

[11] 0 or more. In this example, each component produces exactly 1 output for each input, but this is not a requirement.

[12] FYI - This is easy to accomplish using closures and state-machine mentality. Discussed later. It is also easy to accomplish using threads, albeit this is overkill.

[13] There are many scheduling possibilities. For example, the Dispatcher may invoke a Component repeatedly until the Component's input queue is empty. Or, the Dispatcher may choose to work in a round-robin fashion. Fairness is not an issue (since a Component will eventually go idle when its input queue is empty).

[14] Components are truly asynchronous and must survive through any dispatching order.

- Events and data contained in events, are immutable.
- If a Component sends one event to multiple receivers, it must lock the routing wire.[15]

In addition,

- Components have no parameters, Send()[16] is used instead.
- Components have no return values. Send() is used instead.
- There is no syntax for exceptions[17]. Send() is used instead.

Using Threads

Operating system threads[18] can be used to trivially implement components.

Each Component has a mailbox[19] and it sends messages to it parent[20]. The parent routes the messages to the mailboxes of appropriate receivers[21].

Note - using threads is overkill. An operating system based thread involves hardware MMUs[22] and separate stacks. Operating system threads implement the out-dated notion of time-sharing. None of these are actually required to make this system work.

[15] In practical systems, this is not an issue. It becomes an issue for bare metal systems (no operating system) or systems where Components are distributed along "very slow" connections. I leave this "problem" to the Architect to solve in a manner suitable for the application. I simply want to give the Architect the tools to work with to build reliable systems. The Architect makes guarantees of reliability. This system provides only the bare minimum tool set.

[16] SEND() is the only mechanism for transferring data. Data can be transferred to any number of receivers up and down the line, using SEND(). There is no need for a specialized RETURN() expression.

[17] SEND() is the only mechanism for transferring data. Exceptions are simply data. Data can be transferred to any receiver using SEND().

[18] a.k.a. processes

[19] a.k.a. input queue

[20] which I call a schematic

[21] The routing information is based on the arrows on the diagram(s).

[22] Memory Management Units

Fairness

Fairness is not an issue.

Components run a single incoming event to completion, they yield only to the Dispatcher.

This system mimics, more closely, the modern ideas of distributed systems[23].

Thread Safety

Thread Safety is not an issue.

Components cannot share memory, hence, thread safety is not an issue.

Shared Memory

Shared memory is not an issue.

Components cannot share memory.

Components can only send immutable messages.

In very tightly coupled systems, we have the temptation to send pointers to large blobs of memory. The sender might mutate the blobs of memory before the message is read by the receiver.

This system makes no guarantees for such situations.

This system[24] gives the Architect all of the atomic tools necessary to create systems that work. For example, the memory-sharing issue was encountered 40+ years ago in TTL-based hardware systems. The solution was to use "double-buffering" and "overrun" flags. If a system could not switch between buffers quickly enough, then it created an overrun condition. A system which encountered overrun was deemed simply to be "too slow".

[23] e.g. IoT, multi-core and internet systems

[24] We call it Arrowgrams.

The Architect must make the calculation[25] of whether his/her design is "fast enough" for a given purpose.

Priority Inversion

Priority inversion is not an issue.

I don't use[26], nor specify priorities[27], hence, priority inversion cannot happen.[28]

Loops and Recursion

It turns out that Looping (and Recursion) is the exception, not the rule.

Components must not enter long-running loops (or deep recursion). Components must yield to the Dispatcher. Note that compilers could insert yields at the bottom of Loops to accomplish this behaviour.

The Dispatcher routine is the only routine in the system that runs a loop. It loops through a list of ready closures and, randomly, invokes a ready closure. When the closure finishes[29], the Dispatcher simply picks another ready closure to run.

Dynamic Routing

Dynamic routing is not an issue, because it's not supported.

Dynamic routing used to be called self-modifying code. Self modifying code is a bad idea.[30]

[25] Calculation is discussed elsewhere.

[26] This is not a flippant statement, regardless of how it sounds. It is based on hard-won experience with real products. Priorities can (almost) always be designed out of a system.

[27] This is system of atomic tools. Priorities are non-atomic and can be implemented using the atomic tools.

[28] If a system must act using priorities, then the Architect is referred to literature on hardware priorities1 fully documented some 40+ years ago. 1 NMI and IRQ levels.

[29] e.g. executes a RETURN

[30] Self-modifying code is especially bad from the maintenance perspective.

Using Closures

Most modern languages provide the concept of closures.[31] Closures might be called anonymous functions, or callbacks, or be embedded in concepts such as futures, etc.

Even C has a way to make closures, using function pointers.

The minimum closure required by this system requires some static, but not exported, data, and a temporary stack[32]. N.B. one stack, for the complete system, is enough[33].

OO[34] Objects and Blocks are very close, in principle, to the ideas expressed here. The difference is that I specify closures that run asynchronously whereas OO uses Objects that perform synchronous call-return and explicit naming of receivers. I recommend that closures send messages to their parents and do not refer to each other directly.

In my opinion, and experience, creating separate stacks for each closure and using MMUs is overkill. I don't wish to use time-sharing in my programs. I might use time-sharing if I were to build an operating system.[35]

If one imagines that closures contain state-machines, then, this method could be considered to be a system of communicating state machines. I think in terms of clockwork[36].

[31] Closures were explored in the 1950's using Lisp.

[32] A temporary stack is used for compiler-generated temporary values.

[33] One stack is needed for implemenationon von Neumann architectures. Maybe one stack is one too many in non-von Neumann architectures?

[34] Object Oriented

[35] I argue that we shouldn't use operating systems at all.

[36] Rendezvous techniques also form clockwork systems. Rendezvous has the drawback that it imposes synchronous operation of processes. This is anathema to concurrency. In a concurrent system, processes are asynchronous by default. Synchronization is the exception not the rule. If synchronization is needed, it must be explicitly designed (for example, see ACK/NAK protocols in networking, and, synchronization techniques used in TTL hardware of the 1980's).

Other Features

Reuse

This system emphasizes reuse of Architecture[37].

Architecture reuse is more valuable than code reuse.

Refactoring

Software Component Architectures composed in this manner can be easily refactored into other Architectures, simply by moving/deleting/adding arrows.

Components SEND() messages to their parents. Parents contain the routing tables[38]. Parents route messages between their children. This combination makes refactoring of Architectures easy[39].

Isolation

This system produces a natural hierarchical composition of Architectures.

Parents route messages between their children.

Parents act like Components in all other respects. Parents cannot SEND() messages to their peers. They can only SEND() messages upwards to their parents, and route messages of their direct children.

Global Variables

Global variables are not an issue.

[37]Reuse can be performed by cut/copy/paste. According to Paul Bassett, OO does not provide reuse. OO provides multiple-use (which leads to parameterization, which leads to accidental complexities, and so on).

[38]a.k.a. arrows

[39]Routing tables is nothing more than indirection. It also allows for reuse (as opposed to multiple use) of architectures.

Global variables cannot leak beyond the boundaries of their Components.

Global variables are not a problem, if properly encapsulated.

Global Types

Global Types in synchronous languages are just as bad as Global Variables in those languages.

Global anything is bad.

Encapsulation must be applied to every concept in software architecture.

Namespaces

A component has two external namespaces:

- The set of inputs.
- The set of outputs[40].

The internal namespace(s) of Components does not leak out.

All input names must be unique within the input namespace.

All output names must be unique within the output namespace.

The same name may appear in, both, the input and output namespaces.

Namespaces are naturally encapsulated in a hierarchical manner, due to hierarchical encapsulation of Components.

If two Components have exactly the same input namespace and the same output namespace, then the components are considered to be interchangeable, and "pin compatible".[41]

[40] I call them "input pins" and "output pins", resp., inspired by TTL hardware concepts.

[41] This is similar to referential transparency, but, without the constraint that pin-compatible components must produce the same outputs (this loosening of the same-output restriction allows upgrading).

Isolation of Control Flow

Control Flow within Components is naturally isolated by the fact that Components are truly asynchronous.

Control flow begins when a Component is invoked, and, control flow ends when the Component yields to the Dispatcher.

Control Flow does not leak beyond the boundaries of Components[42].

[42]Control Flow cannot leak because Components cannot CALL other components. Components can only SEND() messages upwards to their parents for routing. Corollary: the direction of SEND() is vertical-only - SEND() cannot be used to deliver messages sideways to peer components. Components can SEND() data upwards to their parent. A parent can route messages between children (and their own input and output pins). Parents can SEND() commands downwards to children.

Scalability

Complexity is not the problem.

There is no silver bullet. There are many silver bullets.

The main problem in software design is scalability.

We want to "plug" pieces tother like LEGO blocks.

Better scalability implies fewer dependencies.

Early hardware people got this "right". They took incredibly complicated devices (semiconductors made up of various kinds of rust) and built chips / ICs (integrated circuits).

Chips were black boxes. They had a set of input/output pins. The insides of the chips were inscrutable - encased in opaque epoxy.

Nothing leaked out of or into a chip except through the pins of the chip.

Properties of a chip were described in easily-measured terms:

- voltage on a pin
- current needed by a pin
- diagram / chart of the outputs, given a set of inputs
- timing.

Then, hardware designers "discovered" that point-to-point wiring between chips led to non-scalable designs.

They built a (small) hierarchy - chips mounted on boards plugged into backplanes.

The earliest backplanes were basically point-to-point wiring harnesses. For example, an early Wang word processor had a backplane with some 400 pins, allowing a chip on one board to send signals directly to a chip on another board.

Then, came the S100 bus. It had only 100 pins. It was well defined and documented. Certain connections were not allowed, even if they could be done more efficiently as point-to-point connections.

The idea of the Bus led to Apple computers and, ultimately, the IBM desktop computer. (There was more than one Bus definition, but the market shook those out).

Can software be built like chips? I argue Yes.

We need to build software in hierarchies.

Divide and conquer.

No leakage - of anything - between layers in a hierarchy. ("Anything" includes things like variables, types, control flow, dependencies of any kind, etc.).

Coming back to Complexity: we *don't care* how complicated a component is, as long as it is well-encapsulated and as long as we don't have to deal with any of its leakage of dependencies.

I see software as a hierarchy of black boxes. The Architect for each box chooses the best way to describe the design intent of a black box. The Engineer figures out how to dot the I's and cross the T's. The Production Engineer figures out how to make the black box "more efficient" and the Coder lays the bricks to implement the black box.

A good Architect will have a tool-belt full of Silver Bullets. Maybe a problem is best described in Relational terms, maybe a problem is best described as a State Machine (as a diagram, yet), maybe a problem can be broken down in a synchronous manner, etc., etc.

Insidious Form of Dependency

An insidious form of dependency that is overlooked is the "dynamic call chain" created by using a (usually hidden) stack to store state

and return addresses between function invocations. This dependency must be broken if software is to scale to new heights. Breaking this kind of dependency requires Concurrency. Concurrency, currently, has a bad name (i.e. it is thought to be a "hard" problem) because it has been tangled up with Time Sharing and Operating Systems. Most applications don't need Time Sharing and all applications, except Linux, MacOSX, Windows, etc., don't need to implement operating systems. Concurrency is *much* easier when Time-Sharing is removed.## Old Reality vs. New Reality Old reality - The old reality was: limited memory and one CPU (hence, the name "Central Processing Unit"). In this reality, it was reasonable to simulate concurrency and have one stack per process.

New Reality vs. Old Reality

New reality - Huge amounts of memory and many processors (none of those being "Central"). In this reality we can afford to have multiple stacks (e.g. one for each type) and use SEND() for every kind of data movement (displacing function parameters, return values, exceptions, all of which came about due to the Old Reality).

There is nothing "new" in the above ideas. Humanity has dealt with issues of isolation and timing many times before.

For example, businesses are built on the notion of hierarchy.

For example, music scores deal with real-time issues and sequencing.

Measuring Isolation

- No need for make
- No need for any package managers
- No need for tools like AutoConfig.

- Manuals that are only one page long.

Other measures:

- Using a hierarchy of DSLs to solve a given problem.
- Breaking out of a text-only syntax mentality, using DaS (Diagrams as Syntax).

5 Whys of Software Components

Q1: Why can't we plug software modules together like LEGO® blocks?

A: Because software is too complicated.

Q2: Why is software too complicated?

A: Because we can't plug software modules together like LEGO® blocks.

Q3: Why can't we plug software modules together?

A: Because the interfaces are way too complicated.

Q4: Why are interfaces too complicated?

A: Because libraries and APIs have hidden dependencies.

Q5: Why do libraries and APIs have hidden dependencies.

A: Because we use languages that hide dependencies.

Q6: What kind of dependencies are hidden by languages?

A: For one example, CALL / RETURN leaves breadcrumbs on stacks, forming dynamic dependency chains.

Q7: Why does CALL/RETURN leave breadcrumbs on the stack?

A: RETURN needs to follow the breadcrumbs back to the caller.

Q8: What happens when a CALLed routine calls another routine?

A: The callee becomes the caller and leaves another breadcrumb on the stack.

Q9: Why is a stack used? Why not use a "register"?

A: Stacks are used so that previously laid breadcrumbs are not overwritten. If we didn't have an automatic stack mechanism to save our breadcrumbs, we'd have to save them manually in some sort of list.

Q10: Is the stack of breadcrumbs a dynamic data structure?

A: Yes.

Q11: Is the stack of breadcrumbs a dependency chain?

A: Yes:

Q12: Is the stack not just a list?

A: Yes, but it is an optimized list.

Q13: Why is it necessary to optimize lists to make stacks?

A: Because we need to optimize memory usage.

Q14: Why do we need to optimize memory usage?

A: Because memory is expensive

Q15: Why is memory expensive?

A: Memory used to be expensive in the 1950's.

Q16: Is memory still expensive in 2020?

A: No.

Q17: Why do we use techniques to optimize memory usage, when memory is no longer expensive?

A: Uh, because we've always done it that way.

Q18: Did we notice that the ground truth has shifted? Memory used to be expensive, but is no longer expensive?

A: Uh, no, we didn't notice.

Q19: Why didn't we notice?

A: Uh, because we believe in building on the shoulders of others.

Q20: Are we in the weeds?

A: Yes.

Acknowledgement

5 Whys suggested by Daniel Pink in Masterclass: https://www.youtube.com/watch?v=My7hjBp4wH0

Git Could Do More

Github, Git, Diff, etc.

I consider "github" to be a whole gamut of technologies based on "diff".

- githhub
  - git
    * diff (a UNIX tool)

"Github" is a production version of "diff".

Automated DRY

Github could use "diff" in a bigger way. Instead of diff'ing lines of code, it should diff layers of design. (Which might be blocks of code).

Our PLs[1] are just glorified text editors.

A lot of what we think about and build into PLs is DRY[2].

RY[3] is what we want, but we waste brain power on DRY.

Machines could detect DRY for us. And refactor the code/design/etc. for us automagically.

Github could include technology to figure out DRY.

[1] PL means Programming Language
[2] DRY means Don't Repeat Yourself
[3] RY means Repeat Yourself

Git-based Editors

Our editors could figure out repeated lumps of text and make a golden copy for us.

The editor could show repeated lumps of source in gray and automagically update the golden copy when we edit the gray parts.

DRY vs. Component-Based Programming

One of the thrusts of Component Based Programming and of FDD is:

- Replace boiler-plate parts of your workflow with automation. Get the machine to do as much of your work as possible.

DRY (Don't Repeat Yourself) is a central lesson of Computer Science. But, it only applies to human-written code.

It's OK if the machine does RY, but if a human does RY, trouble looms (e.g. forgetting to update every instance of every piece of code).

Component-based programming allows copious RY. "Referential transparency" in FP is an RY-enabling feature. HyperCard was kinda component-based programming. Spreadsheets allow potzers to use RY more before they get into scalability issues.

Solutions that I know of:

1. Components. Wrap code into bundles and give each bundle an input API -AND- an output API. You can copy/paste and/or replace bundles that have exactly the same input and output APIs. All code goes into separate bundles. One can copy/paste bundled code without futzing with the code itself. When you change code inside a component, all other uses of that component see the same fix. Programmers believe in the

fiction that Libraries already do this, but, they don't, due to hidden dependencies caused by Call/Return.

2. If one utterly insists on using text, one might check out the research in "clone detection". I haven't checked this out, but I see that Cordy and Roy invented something called NiCad. Cordy is a simplicity hero. Cordy was mentored by Ric Holt. I learned a lot from Holt and Cordy.

Factbases

The trick to automating anything in software is to find a way to normalize the information.

We discovered how to normalize code very early on - we used a notation called assembler.

Assembler is code represented as triples - relation, subject, object.

For example

```
MOV R0, R1
```

is a triple. The relation is MOV. The subject is R0. The object is R1.

If the idea of triples sounds familiar, it's because you've already heard of it.

XML is triples[1]. The Semantic Web is triples.

So, what is the normalized format of data?

Triples!

We want to put data into the form: relation, subject, object.

Conveniently, this already exists. It is a function of two parameters:

```
relation(subject,object);
```

or,

```
fn(id,x);
```

Why is this a Good Thing?

Well, because we impose no structure on the data.

If the data has no structure, then it is easy to parse (in an automatic manner).

[1] XML has too much syntax to be easily readable.

If the data has no structure, then it can be used for other kinds of things. Things that the original programmer never thought of.

Note that organizing data into a data structure at compile time is just an optimization. In the 1950's, it seemed like a good idea to preserve CPU power by pre-compiling data into data structures.

Today, CPUs are cheap and abundant, we can waste CPU time building data structures at runtime. We don't have to worry about pre-compiling things.

How shall we waste CPU time?

By performing exhaustive search.

By using backtracking. Backtracking was verbotten in the 1900's, but no more.

I know, from experience, that I will be using PROLOG for doing exhaustive search. So, I will put a period ('.') at the end of each function and call this a fact.

```
fn(id,x).
```

Caution: use the principles of Shuhari here.

Shu: don't expand the definition of "x" in the above. It is one thing and one thing only.

For example, imagine that I have a rectangle R, with top-left (x,y) and a width, w, and a height, h.

In factbase notation, this becomes

```
1 rectangle(R,nil).
2 top_x(R,x).
3 top_y(R,y).
4 width(R,w).
5 height(R,h).
```

Resist the urge to do something like:

```
rectangle(R,[x,y,w,h]).
```

or

```
1 rectangle(R,nil).
2 top(R,[x,y]).
3 wh(R,[w,h]).
```

We will use CPU power[2] to glean various relationships about the data. In fact, one of the first relationships will be to create a bounding box for each rectangle (we will see this later).

CPU power is cheap.

Memory is cheap.

Don't waste brain power.

Don't try to predict the various ways in which data will be structured.[3]

PROLOG already knows how to deal with triples of the above form.

PROLOG knows how to search triples.

You know how to search triples (loops within loops within ...).

It gets boring after a while.

When it gets boring, automate.

Automation done before you know what you're doing is called premature optimization.

You can't know what you're doing unless you've done it manually several times[4].

Corollary: premature optimization happens in anything that is built before the 3rd version.

[2] exhaustive search

[3] Only tyrants tell you what to think. Be free. Let others be free.

[4] Fred Brooks says that you need to build something 2 times (and throw the results away). Then, knowing what you are doing, you build it a 3rd time.

Code is cheap, thinking is hard.

Code is cheap, experience is hard.

It is best to grow the factbase with new facts, rather than culling facts from the factbase. This allows maximum flexibility during design. Culling is just an optimization and should be left to Optimization Engineering.

Recap:

Code is normalized into triples, e.g. `MOV R0,R1`.

Data is normalized into triples, e.g. `top_x(R,x)`.

Factbases are the normalized (triple-ified) form of data.

Further Reading: see the Appendix for tools like gprolog, JS prolog, miniKanren, AWK.

The Universal Datatype

The Universal Datatype is a *relation*, e.g.

`relation(subject,object)`

Triples

Relations are also called a triples.

Assembler

`MOV R0,R1` is a triple.

Relation, Subject, Object.

Normalization

Data / Code represented as relations is normalized.

Normalization is the most-atomic form of representation.

Example:

`rectangle (5, 10, 20, 30)`

can be further atomized — normalized — as

```
rectangle (R1)
x (R1, 5)
y (R1, 10)
width (R1, 20)
height (R1, 30)
```

[Yes, normalization wastes space and CPU power, but, we have lots of each today.]

Factbase

I often use the term fact and put facts into a factbase.

Compilers

Compilers like triples, e.g. MOV R0,R1.

Optimization

Optimization is easier when target code/data has been normalized.

Peephole optimization is easy to do with even simple tools like awk when code / data has been normalized into triples.

Fraser/Davidson wrote a landmark paper[1] on peepholing which formed the basis of Gnu's gcc.

Normalizing code and optimizing it is not just for compilers. The techniques could be ratcheted up a notch to cover higher levels of software Architectures.

[1] https://dl.acm.org/doi/10.1145/357094.357098

Anecdote - Y2K and COBOL

We analyzed banking source code for Y2K problems.

We used TXL to convert all source code into factbases, then ran backtracking pattern-matching rules over the normalized code.

Pattern Matching Factbases

Backtracking

Exhaustive matching can be done with simple algorithms — backtracking.

Backtracking is easier when the data is normalized.

[That's why compiler writers aim at assembler when writing optimizers. Today, trees are used, but trees get in the way.]

PROLOG

PROLOG is one of the earliest attempts at backtracking.

[I have built a PROLOG in JavaScript. See https://guitarvydas.github.io/]

TXL

TXL is a functional, backtracking, parser language.

http://www.txl.ca/

MiniKanren

MiniKanren appears to be the successor to PROLOG-like languages.

MiniKanren can do seemingly-magical things https://www.youtube.com/watch?v=elLvkklsk (and https://github.com/webyrd/Barliman).

One has to wonder what the child of MiniKanren and AI might turn out like.

Programming Language Design

Imagine if all code were normalized to triples.

We'd be programming in assembler, or in the mostly-syntaxless Lisp.

Successive programming language designs have tried to remedy the problems of working in triples, for human consumption.

Programming languages have taken years to design and to perfect.

Now, using PEG parsers, we can build languages in a day[2].

We can tune a language for a specific problem.

I call these SCLs — Solution Centric Languages.

PEG vs. YACC

YACC embodies LR(k) theory.

YACC builds languages from the ground up.

PEG builds parsers.

Programmers can understand PEG. Compiler-writers understand (mostly) YACC.

[2]Especially if we cheat.

PEGs are easier to use than YACC.

YACC needs a scanner, e.g. LEX.

PEGs are all-in-one - scanner and parser, utilizing familiar REGEXP-like syntax.

PEG vs. REGEXP

PEG is like REGEXP, only better.

If you use REGEXPs, stop.

Use PEGs instead.

PEGs make it easy to match sequences that REGEXPs have a hard time with.[3]

Automation

Normalization leads to automation.

First, make it repetitive and boring.

Then automate.

Programming

Programming consists of two basic activities:

1. breathe in — pattern match
2. breathe out — rearrange and emit.[4]

[3] The difference lies in the fact that PEGs use a stack and allow you to easily write pattern-matching subroutines.

[4] (2) might also involve actions

If (2) occurs before (1) is finished, we get problems. FP is an attempt to fix such problems by throwing the baby out with the bathwater. State is not the problem — unscoped use of State is the problem].[5]

[5]https://guitarvydas.github.io/2020/12/09/Isolation.html. See, also, Structured Programming, StateCharts, etc.

Triples

Triples are everywhere, albeit optimized into oblivion.

The major simplicity that I have known for years - and unable to express succinctly - is that compiler-people like triples.

I, also, call this “divide and conquer”. [If you think of a better way to say that, let me know!].

XML

AFAIK, xml started out life as triples. (This doesn’t mean that I am right, it is only what I believe)

Example of RDF triples: https://www.iro.umontreal.ca/~lapalme/ForestInsteadOfTh

Example

An example is given here

(from [1])

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<note>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
</note>
```

[1] https://www.javatpoint.com/xml-example

As Triples

note id234 null from id234 “Jani” to id234 “Tove” heading id234 “Reminder” body id234 “Don’t forget me this weekend!”

Computer Science

All of CS breaks down into triples - something I call “simple”.

All of CS has been about trying to optimize triples to save space and CPU.

Very 1950’s.

In 202x’s, memory is cheap, CPU’s are cheap.

Time to rethink.

Human-time is still expensive.

Waste computer-time, not human-time.

Data Structures

Data structures are just optimizations of triples

Data structures at compile-time are

- 1950’s notions of how to optimize for CPU usage
- Design Intent (more human-readable, but hard to automatically optimize)

Curried Functions

Even curried functions are triples

```
fn x y -> (fn x) y
```

`(fn x) y` - looks like a double, but is really a triple - relation=fn, subject=x, object=y.

`(fn x)` is a double.

`(fn x) y` is a triple.

`(fn x)` is an unresolved triple.

PROLOG

PROLOG allows you to go into feature-itis, creating quadruples, quituples, etc., but they are just optimizations of things that are fundamentally triples.

Anything in PROLOG that is "more than" a triple is a layer.

PROLOG is (can be):

- describe problem as relations (triples)
- waste CPU power rebuilding data structures (better than wasting human intellectual power).

Human Readability

Humans shouldn't have to read triples.

But, if you deconstruct everything into triples, then it becomes easier to write DSLs and the like.

Agile TakeAways

Goal of Agile

The goal of Agile is to deal with the uncertainty of the Design process.

Agile deals with design uncertainty by involving the stake holder(s) multiple times during the design process.

Most customers, stake-holders, cannot specify Requirements with enough grueling detail to allow implementation. Most customers, stake-holders, do not want to deal with all of the details.

Agile is just a stop-gap measure applied to Design, until we find a better way to perform Design with less uncertainty.

Religion of Agile

Agile has become dogmatic - a religion.

Many programmers focus on the rituals of Agile, without addressing the goals of Agile.

Takeaways from Agile

- Stake-holders cannot (will not) specify all of the details of a problem.

Anti-Takeways from Agile

Sprints Are Too Long

2/3-week sprints are too long.

We should be striving for continuous improvement in the process of gathering requirements.

2 Hour Sprints

We should be striving for 2-hour sprints.

Redefining Sprints

We should be striving to change the concept of sprints into something even more productive.

Flexibility

We want:

- flexibility in the Design
- optimal behavior in the field.

This suggests that we should break programming down into two major categories:

1. Flexible Design
2. Optimization for Delivery.

The categories have different goals.

We should treat each category differently.

We should not conflate both categories into the same activity, nor the same PL[1].

Inflexibility

Optimization removes flexibility to achieve better performance.

Premature optimization is pervasive, yet hidden, in most apps written with current PLs.

Tell: Choosing Programming Language First

Choosing a PL before fully understanding a problem leads to later inflexibility.

The only "good" reason to hire programmers experienced in a certain language is to maintain and upgrade existing code.

Tell: Choosing Only One Programming Language

If the programming language does not fit the problem, aspects of the problem-solution must be force-fit into the programming language, e.g. Relational Programming languages do not fit output formattting (well).

The use of only one programming language is a tell.

All aspects of the solution might not suit themselves to a single, chosen language.

(Sometimes, all aspects of the solution do fit a given language. This is an exception, not the rule).

[1]PL means Programming Language

The use of a single programming language - for design - is indicative of the "programmers are interchangeable" mind-set, instead of the "best-of" mind-set. For example, Apple Computers lets the cost of their computers be determined by a "best-of features" mind-set, instead of allowing cost to drive which features are included.

Note that I argue that PLs are cheap to build[2], today.

Indirection

Indirection retains flexibility at the expense of optimization.

Multiple Syntaxes

Designs often have multiple facets.

Each facet might be best expressed in a different syntax.

Hence, an app should employ multiple syntaxes.

[I argue that SCNs - Solution Centric Notations - should be used. SCNs are cheap to build (hours instead of years). Multiple SCNs can be tuned for a specific problem. SCNs are so cheap, that one can build SCNs for a specific problem instead of resorting to one off-the-shelf PL].

Diagrammatic Syntaxes

There is no reason to use only text-based notations for a given problem.

Diagrams can be transpiled to executable code. I detail how to do this in other essays.

[2] "Cheap to build" means hours, not years, for implementation.

Architects' toolbelts[3] should include many textual and diagrammatic PLs.

Whiteboards

Most stake-holders and most CTOs use whiteboards to present their ideas to implementors.

Q: Can we capture the whiteboard drawings?

A: Yes, using photos and drawing editors.

Q: Can we manually transpile drawings into working code?

A: Yes, I document how to do this in other essays.

Q: Can we automatically transpile drawings into working code? (I.E. can we build compilers for diagrammatic languages?)

A: Yes, I document how to do this in other essays[4].

Reuse In The Large

It is not enough to reuse code.

It is not enough to reuse Architecture.

We should reuse organizational principles from other professions.

For example, Architecture/Engineering/Construction organizational structure could be borrowed from the construction industry.

Most software organizations don't yet understand that Engineering is not Coding.

[3] Paradigms are more important than syntax

[4] Modern hardware can run exhaustive-search languages, e.g. PROLOG and Relational Programming, which makes transpiling diagrams-to-code simple.

Code is cheap.

Code is cheap.

Thinking is hard.

Software Development Roles

See Software Development Roles[5].

[5] https://guitarvydas.github.io/2020/12/10/Software-Development-Roles.html

Compilers Are Too Slow

Oh, the irony!

The premature-optimization clique convinced programmers that interpreters were "too slow" and that compilers should be used instead.

Now, 50+ years down the road, we find that premature-optimization has resulted in:

- workflows that are too slow
- Mb of bloatware
- techniques that are so complicated that they can't be explained to the majority of professional programmers (think website makers, game makers, JS crafters, etc.)
- acceptance of buggy products - programmers just ship and update weekly, forgetting Q/A

The implication is obvious:

- don't use compilers until you have designs fleshed out and fully working[1][2]

Use compilers only when you need to Production Engineer a Design.

Failing that, buy a faster development system and keep complaining...

[1] It is OK to use dynamic languages to experiment with and develop designs. The idea that you know everything about a design without experimentation is called The Waterfall Method. This, The Waterfall Method, is what is considered to be "programming" currently. Programmers dive into a project with the assumption that they will succeed. The infrastructure for intermediate failures is not built up, programmers jump directly to optimizing designs (which are, at first, untested designs).

[2] PL means Programming Language

Efficiency

"Efficiency" is in the eye of the beholder.

1. If you are Designing a piece of software, then "efficiency" equates to turn-around time based on using a souped-up development machine.
2. If you are Production Engineering a piece of software to hand to clients, "efficiency" means making the result runnable on low-cost hardware (or lowering the cost-to-the-user)

Sector Lisp, FP in < 512 Bytes

FP (Functional Programming) can be small and beautiful.

Sector Lisp is smaller than 512 bytes ([sic], not K, not M, not G, but bytes) on current hardware.

Explain why your favourite language *needs* to be larger and slower than this.

Compiler-writer addage: the only good optimization is one that makes the overall code smaller, when the code for the optimization is included in the code base.

Forgotten

Programming is about making (electronic) machines do repetitive tasks for humans, in any way possible.

There is no rule that says that programmers *must* use characters and text to write programs.

CEOs and visionaries use white-boards and napkins.

We would *prefer* that our machines would tighten-up the process of Design, at least a bit.

We can use manual methods until automated transpilers appear.

Notation Worship

Programming language design has become a cult of notation-worship, where, for no good reason, it is presumed that programming notation must consist of grids of non-overlapping sequential small bitmaps, as invented in the mid-1900s.

This technique hasn't been updated to take into account 202x hardware advances.

Error Checking - Silly Mistakes

Yes, we would prefer that machines check our work for silly mistakes.

We already have syntax checkers.

Types are more-of-the-same kind of checking, but, currently, result in bloatware.

In developing projects in JavaScript, I found that a large majority of my bugs could have been caught - rapidly, using few CPU resources - if the system simply checked for simple type violations, like number-of-parameters to functions.

I didn't need full-blown type-checking, I didn't need TypeScript. I just needed something that would have caught my typos and alerted me to them (e.g. "function defined to take 3 args, but only 2 args were given in line XXX").

Most programming languages already do this. JavaScript doesn't[3].

[3] I am purposefully ignoring *strict mode* and *typescript*.

JavaScript provides a default value for missing parameters (i.e. “undefined”).

5-Line Programs

Programs that are only 5 lines long don’t need type-checking and local variables.

The goal shouldn’t be to appease the act of building 6-line programs, but, to ensure that programs do not grow beyond 5 lines in length.

Silly? Q: What is the average length of a TensorFlow script written in Python?

[5 is just a ball-park number meant for emphasis. Font size and window size play roles in locality-of-reference.]

[[Essays/5 Line Programs]]

Why Don’t We Use Diagrams For Programming?

Why don’t we use diagrams for writing programs?

A: because mid-1900s hardware did not support inexpensive overlapping bitmaps.

Dependencies

Why do we have *any* dependencies between software components?

Our programming languages, supposedly higher-level than assembler, should inhibit, not encourage, the creation of dependencies.

Depedencies come in several flavours, e.g.

- hard-wired names of called functions
- synchronocity-under-the-hood
- CALL/RETURN.

Programmers can make it their goal to expunge all dependencies.

Eating dogfood: expunge the use of *make*, *npm*, *packages*, *package managers*, using CALL/RETURN to invoke code libraries.

Gedanken Exercises

1. You have an apple and a ball on the table. You cut the apple in half. What happens to the ball? Does the ball depend on the apple?
2. You have a 2-story castle built out of LEGO pieces. You carefully remove the top-most piece and cut it in half. What happens to the castle?
3. You have a main piece of software that calls a library. You make a change to the library. What happens to the main piece of software? Can you *guarantee* that the main piece does not change?

Humans understand the notion of independence and free will. Programming does not, easily, support independence and defies human intuition. I argue that this is a main reason as to why programming is considered difficult - programming defies our intuition.

Simplicity - How Do You Build A Light Airplane?

How do you build a light airplane?

1. Build an airplane,
2. Add lightness.

L. J. (Ted) Rootham.

I.E. If you want something simple, start out with something simple. Adding baubles will not make something simpler.

The things that we call *programming languages* are simply notations for viewing and describing aspects of programming. As soon as a notation appears "difficult", programmers should switch to another notation instead of trying to force-fit all views of a problem into one notation. As already mentioned, Sector Lisp is an example of how simple a notation can be and how complex that same notation can become when force-fitted into situations that it does not describe well (e.g. functional programming vs. mutation and heaps and distributed programming and etc.).

What else is considered "difficult" today and could use a different notation? Multi-tasking, for one. The presence of dependencies, for another.

Suggestion: Type Checking Design Rules

Another suggestion: Do what compiler-writers did when they only wanted YACC to help, but didn't want all of the niggly details that came with using YACC. They used YACC to error-check their grammars, then they rewrote the grammars in another programming language.

Likewise, if ultra-type-checking helps you create a Design, use a language that gives you ultra-type-checking (e.g. Haskell) to check the consistency of a Design, then rewrite the code in some other language (e.g. JavaScript). You are responsible to not hack on the

rewritten code, if you ever want to change the Design and have it re-checked. This is not a huge problem. If programmers were to use Haskell->other-language-transpilation enough, someone would automate the process. Better yet, someone will find a better way to express type-checking than using full-blown Haskell.

Yes, it would be nice to have type-checking integrated into the Design loop, separated from compilation, but we're not there yet.

Still, some kind of checkable design rules might might be more useful than ad-hoc white-boarding.

ATM, I draw my designs in draw.io, then hand-compile to JS/Python. With Ohm-JS to help me, this is so easy that I haven't bothered to close the loop to use diagram-parsing technology to automate these steps.

Warping Programming Languages To Allow Compilation

Don't let the premature-optimization clique tell you what techniques can and cannot be used (e.g. REPLs) for programming.

Ask them them to build lint-ers instead of arguing that programming languages need to be altered for their whims.

Ohm-JS, PEG, TXL make the process of writing syntax-driven linters easier.

Appendix References

Sector Lisp[4]

[4] https://justine.lol/sectorlisp/

Why Do We Use Text For Programming Languages?

The only reason that we use text and not diagrams for programs is that in the 1950s, computer hardware could handle text, but couldn't handle vector graphics. We developed more modern technologies for non-programmers, but, programmers themselves are still stuck in the 1950s.

Worse yet, 1950s computers could only handle text that was arranged on grids of non-overlapping, fixed-size bitmaps.

Well, OK, the other reason for using textual programming languages is that we are used to using writing technologies based on clay tablets. After 2,000 years, we've progressed to using graphite and rubber on paper. We didn't imagine that computers were something other than "better clay tablets". Our continued use of 2D writing technologies has severely limited the way that we think about things. We write equations in 2D to express simple physical concepts, like sound, that are 4D phenomena (x/y/z/t). Mathematics seems complicated to many people, because it is not natural and is a round-about way of describing multi-dimensional phenomena that we experience daily. Complexity comes from the fact that we try to describe 4D effects by collapsing them down to 2D. For example, Mechanical Engineers are taught to draw multiple 2D views of simple 4D objects (top, front, side, simply ignoring time).

Computers allow us - for the first time? - to express and explore at least 4 dimensions of our reality. We can shift the camera perspective on a 3D model and "walk around it" - and - we can watch models evolve over time. We couldn't do this easily with

paper and other clay-tablet technologies and mathematics. They key word here is "easily". Mathematicians and Physicists are able to build 2D "models" of real phenomena, but only a select few can understand what's going on (usually after several years of University training). It seems that our notions about programming are based on the idea that "if it's good enough for mathematicians, it must be good for programming, too".

Further: Note that it is just plain hard to draw a sensible diagram using a synchronous (function-based) language. It becomes laughingly simple to draw diagrams if you use 0D components.

Rhetorical question: why can we draw diagrams of computer networks? Because each node on the network is - essentially - a 0D component.

Why don't we do everything this way?

Uh, because of our built-in fear-bias regarding "efficiency". We are preconditioned to worry about saving bytes instead of saving development time. Without handcuffing ourselves with clay-tablet 2D technologies, we *could* trickle-down better apps to end-users, but, we don't.

Further further: computers in 2022++ are *fundamentally different* from computers in the 1950s. In the 1950s, each computer had only 1 CPU and limited memory. In 2022++ we want to build distributed systems such as blockchain, IoT, robotics, internet, etc. Ideas from 1950s don't map very well to 2022++. But, we continue to force ourselves to use ideas from the 1950s. From a product-design perspective, modern CPUs are little more than fancy FPGAs. Modern CPUs don't deserve all of the adulation we heap upon them in the form of creating Sciences for building 1950s UIs (aka "programmming languages") and 1960s IDEs (aka "operating systems"). Our end-users suffer from our tools' deficiencies - we insist on having end-users pay premium prices for bloated operating systems and we insist on having end-users accept the fact that whatever

they buy will be buggy without recourse in Law[1].

Sigh.

[1]Real Engineers are forced to take courses in Tort Law and are reminded of their obligation to produce bug-free products lest they be sued. For example, in Canada, Engineers are given "iron rings", on graduation, to remind them of a buggy bridge design that turned out to be fatal.

FDD - Failure Driven Design

Slides

Failure is the Best Way to Learn

Two Ways of Looking at Development

1. It's going succeed
2. It's going to fail

Outlook Determines Workflow

- How do you write software if you are convinced that it will work the first time?
- How do you write software if you are convinced that it will fail?

FDD vs Regular Design

- FDD
  - How to build-in easy recovery from changes (failures)
  - meta-design
- Regular Design
  - How to build it so that it works first time, ignoring possibility of changes / failure
  - A to B design, straight ahead

Assuming Success

- Waterfall workflow
  - one-way, one direction (e.g. top to bottom)
  - no plan to iterate
  - failure comes as a "surprise"
    - hard to recover from failure

Waterfall

- antithesis of FDD == overconfidence == Waterfall design
- Waterfall: one direction: design->implement
- Waterfall: assumed that the design will succeed the first time
- Waterfall: early attempts / requirements fail to completely solve the problem, but, no recovery from failure is built into the workflow

---

Mythical Man Month

- Fred Brooks - fail, fail, succeed

Assuming Failure

- how to fail fast?
- how to recover quickly?

Failure

- The first several attempts at solving a problem will fail.

Failure vs. Success

- Development:
- When software works, we “abandon” it (ship it)
- When software fails, we continue working on it
- Most of the time, we work on failing / failed code

What Can Fail?

- Requirements
- Design
- Architecture
- Engineering
- Implementation
- Testability
  - for example: testability can drive a change back into Architecture, etc.

Learn by Failing

- Failure is said to be the best way to learn.
- What do we need to learn?
  - what the requirements are
  - all aspects of the problem space (the gotchas)

FDD - Strategies to Make Failure Less Painful / Bothersome

- iteration
- recursion / divide-and-conquer
- automation - rearrange, then push a button to rebuild everything

- layering design (see "Recursive Design, Iterative Design By Example (2), section "Bug 2" and section "Layering Solutions")
  - https://guitarvydas.github.io/2021/04/20/Recursive-Design,-Iterative-Design-By-Example-(2).html
- indirection
- create a notation, SCN (low-cost)
  - punt to toolbox languages
  - punt to foreign functions (DI & Details Kill)
- asking Why?

FDD

- Failure-Driven Development
- most of the time, the requirements will change
- most of the time, a design will have flaws in it
- most of the time, the implementation will need to be debugged and need repairs
- the number of failures >> the number of successes
- plan for failure, since failure happens more often than success

---

FDD How?

- fail fast
- build in backtracking => automation
- look to compiler technologies, transpilation
- Notations, not Languages nor DSLs

Notations, Not Languages

- Notation is a lightweight DSL
- Programming Language: heavyweight, high cost to build
- DSL: heavyweight, high cost to build
- Specialize Notation to Problem Space only
  - avoid generalizing
- YAGNI - You Aren't Going to Need It

Fail Fast

- divide problem, choose greatest risk, greatest unknown
- experiment with / implement unknown
- if unknown becomes known, defer it and choose next greatest risk (which is, now, the greatest risk)
- if unknown is "impossible", then fail and backtrack
  - redefine the problem / solution
- Testing cannot prove that a device works, but testing can prove that a device does not meet its specifications

---

Scientific Method is a Fail Fast Methodology

- A scientific theory is one which is falsifiable
- one can't prove a theory to be correct - 1 data point can only support a theory, but cannot prove it
- one can only disprove a theory - 1 data point can kill a theory

---

FDD How? Backtracking

- script everything, push a button to rebuild
- when a design fails,
  - repair requirements
  - repair the design
  - re-generate
  - try again

FDD How? Compiler Technology

- compilers pioneered automated transforms
- compilers pioneered portability

FDD How?

- don't write code
- write code that generates code

Failure : Automation

- If you assume that you will fail, you are encouraged to use automation and backtracking
- FDD workflow is: repair, push button and regenerate, try again
- Waterfall workflow is: confidence that it will work, design, then implement

  - waterfall: no assumption that attempts will fail most of the time,
  - success is assumed

Automation : Factbases

- To automate, use compiler technology
- MOV R1,R0 is a triple
- everything is a triple => easier to automate
- RTL, OCG, portability ... <= normalization
- Projectional Editing <= normalization

What is the LCD for Automation?

- Q: What is the LCD - Lowest Common Denominator?
- A: triples
- triple = `relation(subject,object)`
- curried function is `relation(subject)`, later applied to `object`
  - i.e. double X single => triple

Manual vs. Automated

- Manual work resists change.
- The time spent is not recoverable.
- Automated work accommodates change.

Are You Ever Finished?

- No, but you reach a point where a product can be shipped
- analogy: songwriting -
  - songwriter continuously tinkers with a song
  - but, making a recording draws a line
  - "Aqualung" live is, now, almost unrecognizable (jazzy beginning)
    * audience member yelled "play Aqualung" during the new intro
  - songs continue to evolve

Appendix

Appendix - DI

Design Intent

https://guitarvydas.github.io/2021/04/11/DI.html

https://guitarvydas.github.io/2020/12/09/DI-Design-Intent.html

---

Appendix - Recursive Design

https://guitarvydas.github.io/2020/12/09/Divide-and-Conquer-is-Recursive-Design.html

https://guitarvydas.github.io/2021/04/12/Recursive-Iterative-Design-By-Example.html

https://guitarvydas.github.io/2021/04/20/Recursive-Design,-Iterative-Design-By-Example-(2).html

https://guitarvydas.github.io/2021/03/18/Divide-and-Conquer-in-PLs.html

https://guitarvydas.github.io/2021/03/06/Divide-and-Conquer-YAGNI.html

https://guitarvydas.github.io/2020/12/09/Divide-and-Conquer.html

---

Appendix - Factbases

https://guitarvydas.github.io/2021/01/17/Factbases.html

https://guitarvydas.github.io/2021/03/16/Triples.html

Appendix - SCN - Notations

Solution Centric Notations

https://guitarvydas.github.io/2021/04/10/SCN.html

Appendix - Indirection

https://guitarvydas.github.io/2021/03/16/Indirect-Calls.html

Appendix - Toolbox Languages

https://guitarvydas.github.io/2021/03/16/Toolbox-Languages.html

---

Appendix - Why?

I watched Daniel Pink's Masterclass

Pink suggests asking "why?" repetitively, some 5 times to understand the problem more deeply

---

Appendix - 5 Whys of...

5 Whys of Multiprocessing: https://guitarvydas.github.io/2020/12/10/5-Whys-of-Multiprocessing.html

5 Whys of Full Preemption: https://guitarvydas.github.io/2020/12/10/5-Whys-of-Full-Preemption.html

5 Whys of Software Components: https://guitarvydas.github.io/2020/12/10/5-Whys-of-Software-Components.html

Appendix - Incremental Learning

A debugger can be used to observe the operation of someone else's code (or your own code).

- Stepping through code and interactively examining data structures is one way to understand the intended architecture.

- Fixing other peoples' mistakes can force you to think deeply about the code and data structure details. Incrementally, not in one big gulp.

Appendix - Details Kill

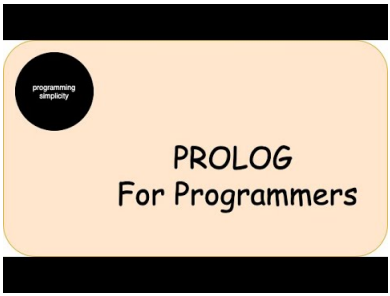
https://guitarvydas.github.io/2021/03/17/Details-Kill.html

elide details

- don't delete details, suppress them
- KISS
  - simplicity is the "lack of nuance"
  - complexity is the inclusion of too many details (in any one layer)

PROLOG for Programmers Introduction (in PROLOG)

video



PROLOG for Programmers[1]

Slides

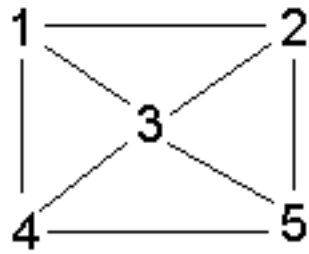
using: https://www.cpp.edu/~jrfisher/www/prolog_tutorial/2_-15.html

---

Declarative <== Relational

PROLOG == Relational

miniKanren == Relational



figure

factbase

```
edge(1,2).
edge(1,4).
edge(1,3).
edge(2,3).
edge(2,5).
edge(3,4).
edge(3,5).
edge(4,5).
```

basic relation

```
connected(X,Y) :- edge(X,Y).
connected(X,Y) :- edge(Y,X).
```

alternate basic relation

```
connected(X,Y) :- edge(X,Y) ; edge(Y,X).
```

---

inferring connections

Divide and Conquer

---

do-it style

---

top level

```
path(A,B,Path) :- BasicPath = [A], inferPath(A,B,BasicPat\
h,Path).
```

---

base case

```
inferPath(A,B,P,ResultPath) :- connected(A,B), ResultPath\
 = [B|P].
```

---

recursive case

```
inferPath(A,B,PriorPath,ResultPath) :-
       connected(A,C),
       C \== B,
       \+member(C,PriorPath),
       NewPath = [ C | PriorPath ],
       inferPath(C,B,NewPath,ResultPath).
```

PROLOGify - Better Relations

---

top level

```
path(A,B,Path) :- BasicPath = [A], inferPath(A,B,BasicPat\
h,Path).
```

->

```
path(A,B,Path) :- inferPath(A,B,[A],Path).
```

---

base case

```
inferPath(A,B,P,ResultPath) :- connected(A,B), ResultPath\
 = [B|P].
```

->

```
inferPath(A,B,P,[B|P]) :- connected(A,B).
```

---

recursive case

```
inferPath(A,B,PriorPath,ResultPath) :-
       connected(A,C),
       C \== B,
       \+member(C,PriorPath),
       NewPath = [ C | PriorPath ],
       inferPath(C,B,NewPath,ResultPath).
```

->

```
inferPath(A,B,Visited,Path) :-
       connected(A,C),
       C \== B,
       \+member(C,Visited),
       inferPath(C,B,[C|Visited],Path).
```

SWIPL at command line

```
swipl
  ?- consult(path).
  ?- path(5,1,R).
  ?- halt.
```

SWIPL in Bash script

```
#!/bin/bash
swipl -q \
    -g 'consult(path)' \
    -g 'use_module(library(http/json))' \
    -g 'bagof(R,path(5,1,R),B),write(B),nl.' \
    -g 'halt'
```

SWIPL with JSON

```
#!/bin/bash
  swipl -q \
      -g 'consult(path)' \
      -g 'use_module(library(http/json))' \
      -g 'bagof(R,path(5,1,R),B),json_write(user_output,B\
),nl.' \
      -g 'halt'
```

Transcript

I am gonna do a quick introduction of PROLOG for programmers using JR Fisher's tutorial. It's on this webpage here. One thing to notice is that we strive to do declarative programming. One form of declarative is relational programming. PROLOG was one of the first attempts at relational programming.

Mini kran is the current manifestation of relational program. I happen to be using eax org mode, and I'm gonna probably use a sw s w I PROLOG, although I've used G PROLOG a lot also. Let's have a look at the figure. It's Listed in that article it's basically five nodes forming a network, and there's connections between the nodes in, the various ways that are shown on this diagram.

The diagram shown can be described as a PROLOG da fact base. In PROLOG, we would say that there's an edge from one to two. The period acts like a semicolon in many languages. These are all relations. They're basically just straight out con fact constants, and if you look through the rest of it, you can get the idea of what's going on here.

If we look at the fact base, we see that there's an edge from one to two, but it doesn't say that two is connected to one. We can add a basic relation that, that shows this relationship that two nodes are connected. If there's an edge between X and Y or if there's an edge between y and.

In PROLOG, we can write this basic relation a different way, which is to use a semicolon operator. So basically we're saying connected is edge of X, Y or, or. edge of Yx in PROLOG, it is conventional to put all the or terms separated as, separate relations or in this form where, there's a semicolon between them and type relations are done with commas and we'll see that.

We can infer further connections by adding PROLOG rules, using guess what? Divide and conquer. There's the programmer, do it kind of style and the pro. Then we can prolog, gify it and turn the relations, the code into better relations.

Any attempt to solve this problem breaks down into two main problems and one envelope I call the envelope around the whole thing, the top level, there's a base case that says how we know when we're finished and the recursive case.

We'll start looking at the top level case, basically having a relation that signifies the path between A and B. And we write the path into a result called path. Fundamentally, every path starts out with square brackets in PROLOG mean list. So we the basic path always has a in it since that's where we start.

And then we infer new paths between A and B. As we go.

If we look at the base case it says that I, I. The, most basic level. If

there's something that satisfies the connection between A and B, then we're done and we just glue the result B onto the path. This is the way you write list cons and B onto the front of B, and then we're putting it into the variable called result path in the recursive case.

We look for connections between A and B. We keep a prior path around and we keep the result path. We invent a, new node and we call it c. And we look for a connection between if, A and C are connected and C is not equal to the final result, B, that we want and C is not a member of the prior path, then we cons C onto the prior path, creating a new path, and then we recur.

We try to find connections between c and b on the, and we produce a resulting path and we keep a prior path around. Now we'll look at how to write these relations in better PROLOG form. We'll look at the top level, the base case, and the recursive case separately. Now looking at the top level in PROLOG form, we take the do it style and rewrite it in PROLOG this way.

Programmers are used to writing a function signature, followed by. How to get there, how to do it. Whereas in PROLOG, everything is a relation, and that's not actually a function signature. It's just a shorthand for whatever is on the right hand side. So the left hand side really is just a, is a truism.

It says when the PROLOG run is finished, then this will be true. And here's how we, and here's how we infer that things are true in this case. What we, don't need a Temporary variable that assigns the list of containing only a into a temporary variable called Basic Path. We can just stick it directly into the relation and get rid of this.

In fact, the equal business in, PROLOG needs a little bit more of looking at. I won't dig into it, but try not to use it.

Likewise, the base case in PROLOG folds our original code, which kind of looked like this, with an equal sign into a relation that says everything in one fell swoop that that we can infer the path from

A to B and the result is b. Associate B put onto the result path, resulting path or put on, put in the resulting path is the previous path with the result.

Put B put onto it only if A and B are connected. again, this doesn't look like what programmers are used to because we're, showing things that programmers think of as do it code causing things on the left hand side. In fact, this is still a relationship that shows that infer path is true and we had a prior path and that the, the.

thing we called the resulting path is broken up into a head and a tail. A head where the head is the, new, the node and the tail is the previous path. And when we continue writing it this way, we can actually do more, more interesting things. Like we don't actually have to supply A and B.

We can ask given a. what are the po given a path and a prior path? What were the previous, what were the A and Bs that satisfied that relationship? Likewise, the recursive case of Chronification takes the the old style and writes it as a single relation that's recursive. And here we've gotten rid of.

Temporary variables like this. I've in this particular case, I took prior path and renamed it to visited which makes more sense to me in the final result. And we've dropped from five lines to four of again, this side is the abbreviation for these four lines. So if, we can show that these four lines are true, then this relation.

Is true. and PROLOG will find all of the matches for that relation. And in PROLOG, what we'll see is we can actually run some of this code in PROLOG. And we will see that it backtracks over. The code gives us one solution. If we give it a semicolon, it gives us another solution. If we give it another semi and keep doing that until it runs out of solutions, then it stops giving us solutions.

I've taken all of the code we've written so far and the facts and put them into a single file called Path pl. The the extensions, pl and pl are the, ones favored by the two PROLOGs that I use. S SWI PROLOG, and G PROLOG. Now, I'll take this. PROLOG file and

load it into PROLOG. So we'll switch to the terminal, we'll run the SW PROLOG and to load a file, I use the command called consult, and we don't need to give it a path name.

We do have to remember the dot. And that loads the.

We can qu do simple queries on the fact base. For example, we can ask to see if edge, if there's an edge between one and two. And it, and now I'm gonna, it says true. There was, I'm gonna put in a semicolon and then it'll say there are no more edges between one and. I can try an edge between two and one, and that says there isn't any.

There aren't any.

Now we can use the Most basic query that we had, the basic relation, which is connected two and one, and that returns true, you can say connected one and two, which should be true, and it is. Now we can try something more interesting. We can ask for what is the path between one five and we'll put the result in something called p.

And it says that you can get there going from five to one, to two to one, or from five to three to two to one, or five to four to three to two to one, five to four to one, five to three, to four to one, five to 3, 4, 1, 5, 3, 1, 5, 4, 3, 1, 5, 2, 3, 1. And that's,

Note that in PROLOG variables, logic variables are always capitalized. Everything else is lowercase. So if I say path one comma five, big R result, then it has no problem finding it. If I say path one, Lower case result, then it fails immediately. I can also put some of these commands into bash scripts.

I'll halt is the way to quit PROLOG. I go up here and I've I think I've created a file called G Path for running, g PROLOG. If I run that, Slash g then it'll load the, it'll consult the file path pl and then it'll, and then it'll leave me ready to do a query so I can do one five capital R, and then it'll give me the results halt.

I can do the same for SW PROLOG. I've created a bash file called

SW Path Bash, and it it's got a slightly different command line syntax, and we'll try running it. Change this slash. And it leaves me ready to do a query. It's, consulted the file called Path pl and it relieves me ready to do a query like that. And same thing. Halt period. Halt is the way you finish. And then you all have to remember the period.

Swipe PROLOG has the minus Q modifier, so I'll say minus q. I'll do a back slash, which is shell syntax, to continue a line and modify the file. This way. I then run it again, and there it is, ready for me to put in a query.

You can even put -l halt this again, halt. You can put the query in as a further command line argument minus G path five one big R and we can try that. Guess what? It ran the query, but it didn't show us the. Now we'll retry that with rewriting the query to have an output. So we're gonna collect all of the results are of pa of the query path R into a bag called B.

Then we're gonna write b. onto, stood out, and then we're gonna c cause a new line. And if I got that we should be able to run it. There it is. And I forgot to put the, I can put a halt at the end of that like this minus g halt. Let's try that. And that's the whole query and it finished.

Things get more interesting. If we write out the facts as JSON instead of PROLOG fact base format, I'll make the modifications to the shell script. Here we consult path.pl. We, I've added the line that uses the module called JSON. Then we do the bag of, and instead of calling we call JSON Wright.

This is user. This is stood out, user output in Prologis the bag and the N -l. We've saved it and then we try it again, and there we are. It's output is JSON.

The Holy Grail of Software Development

Video

video[1]

Transcript

The holy grail of software development. What is the holy grail? It's simple. We wanna build a piece of software and forget how it works. We just want to use it. We want to design it, bench, test it, then ship it with no further bugs in the field. This is possible. And digital hardware design, you could design, debug a design on a bench, ship it, and we would have zero bugs in the.

even though ICS that make up a electronic circuit are all asynchronous they're, what we call multitasking telecom. In the 1980s at least, had something called the four nines culture. They guaranteed a 99.99% uptime. It. It was so ubiquitous that people would call each

other during a power failure on a telephone and say, is the power out at your, where you.

And they wouldn't even notice that the phone was still working even though there was power failure. So in software, we have many bugs in the field. We have monthly, weekly, daily updates. We have something that I'm calling GitHub culture, where somebody looks at a repo and says it can't be any good if there weren't any recent pushes to the repo.

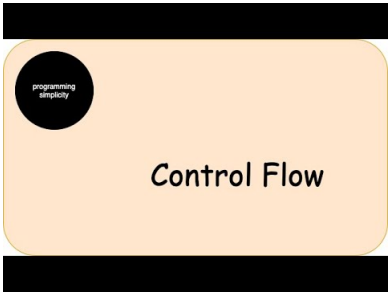
That's the wrong emphasis. Working code should be perfect. It should need no new modific. Sam Aaron, for example, teaches 10 year olds how to create multitasking programs. What's our problem? What's the difference between software and hardware development? That's a rhetorical question. I think it boils down to the use of global variables.

I think that all current languages use a hidden global variable that includes functional programming. And most CPU architectures the stack is a global variable.

Having that global variable hidden from view has been causing this accidental complexity. Call and return instructions provided by the hardware implicitly use the stack and it's a global variable. There's only one copy of it. We, talk. Things like thread safety, and we have this fiction that multitasking is hard, and all of that stuff is caused by the fact that we use a global variable called the stack.

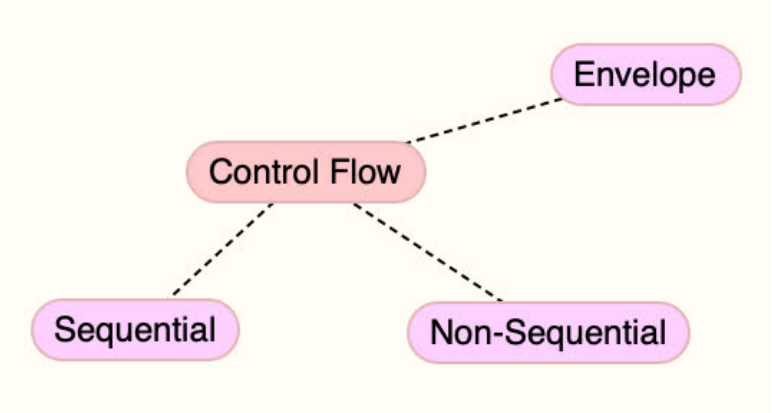
Control Flow

Video

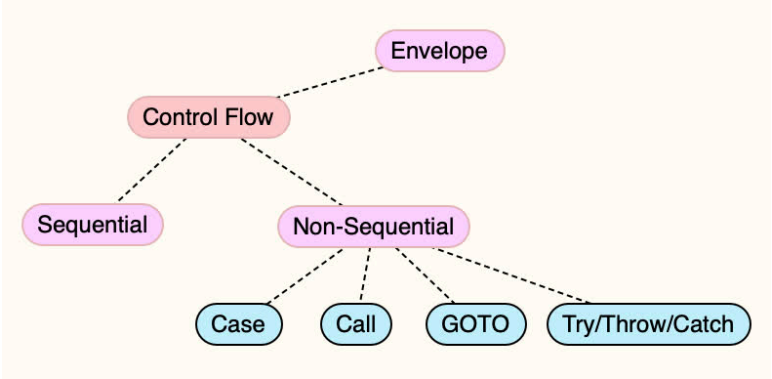


Control Flow video[1]

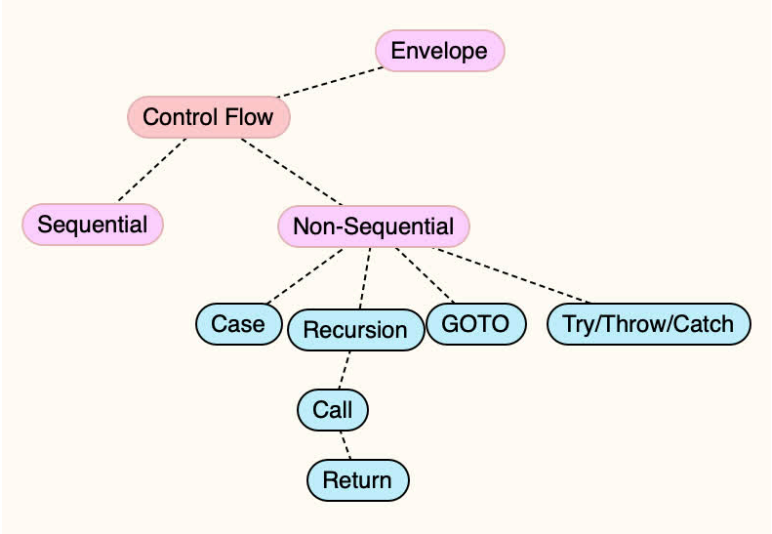
Slides



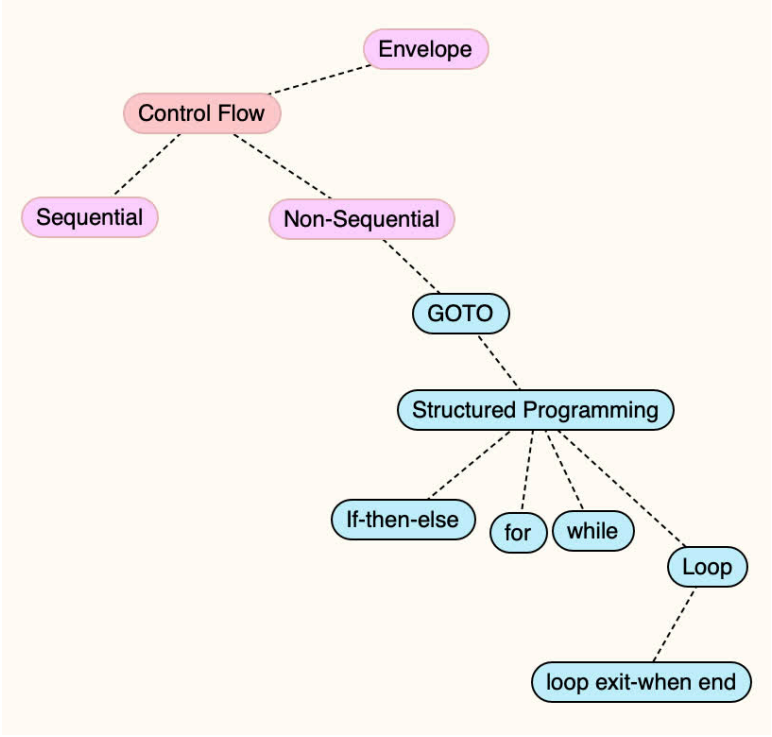
Control Flow 1



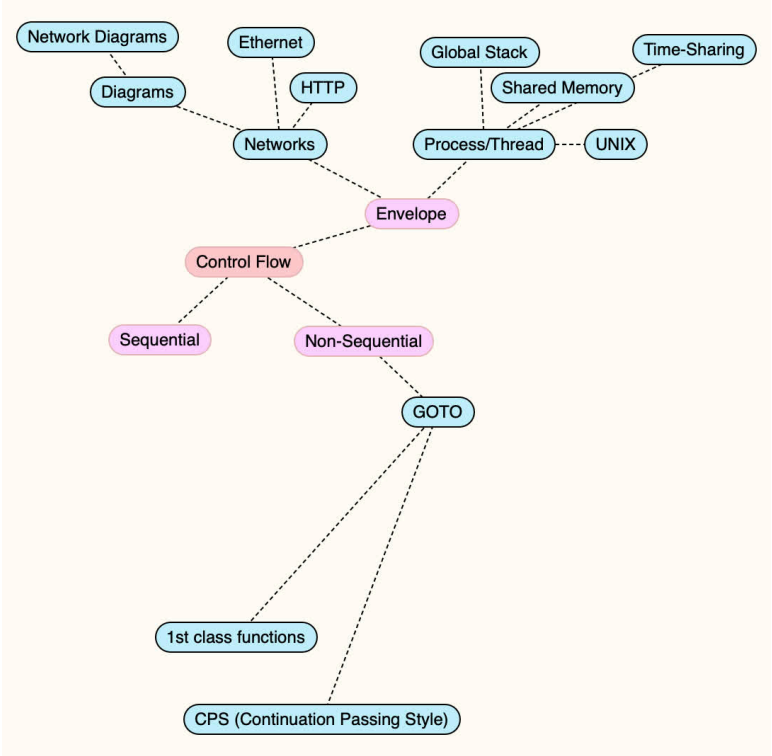
Control Flow 2



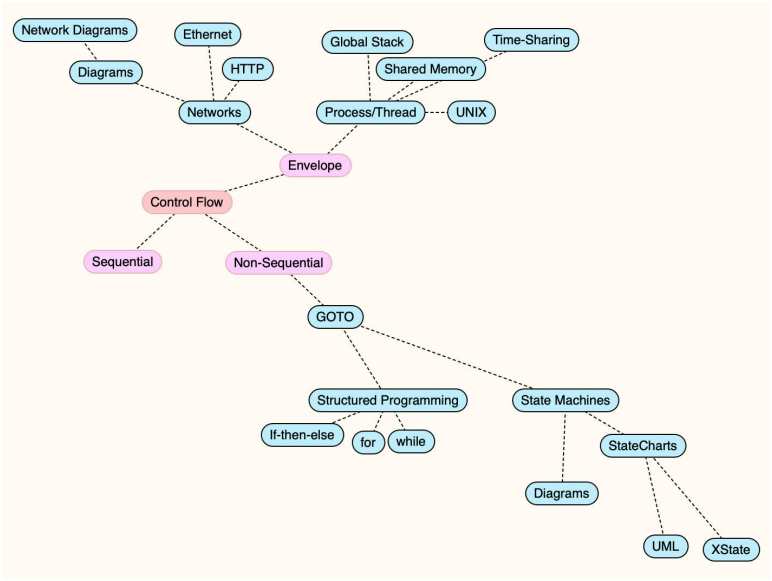
Control Flow 3



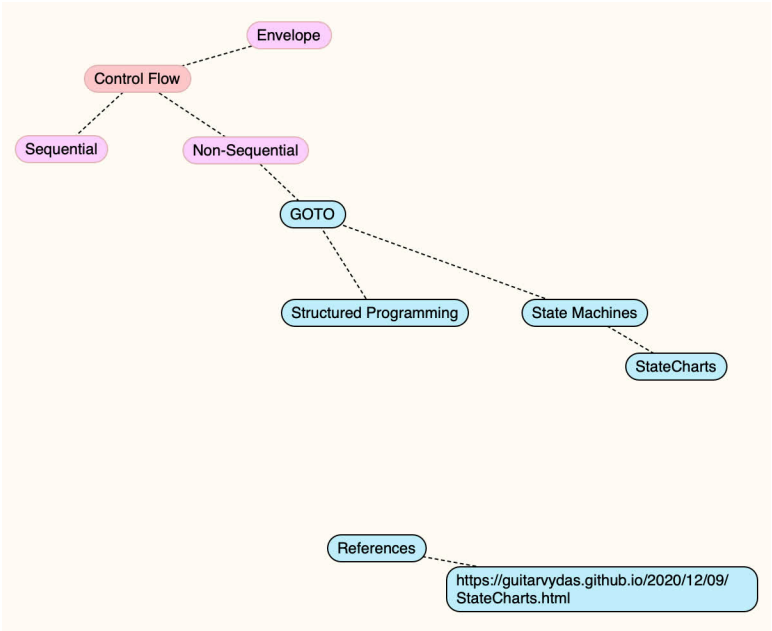
Control Flow 4



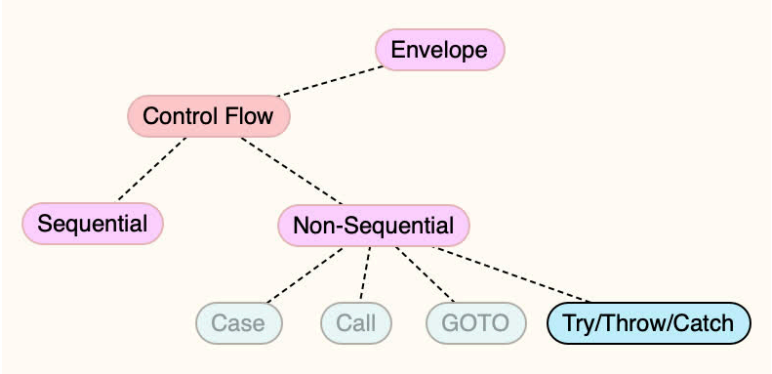
Control Flow 5



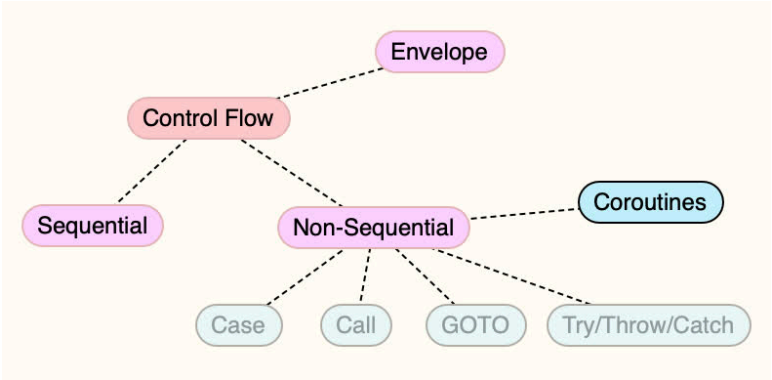
Control Flow 6



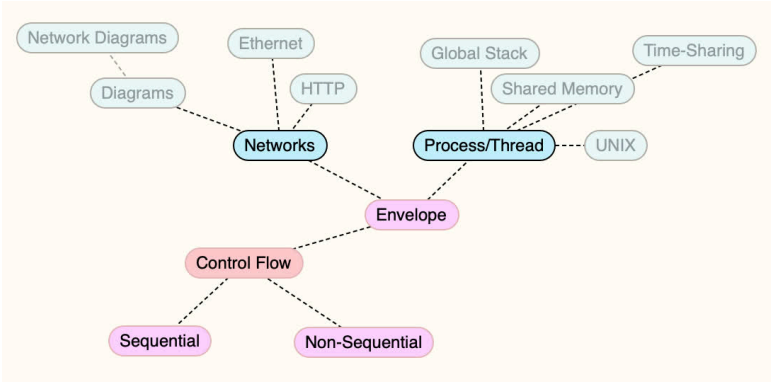
Control Flow 7



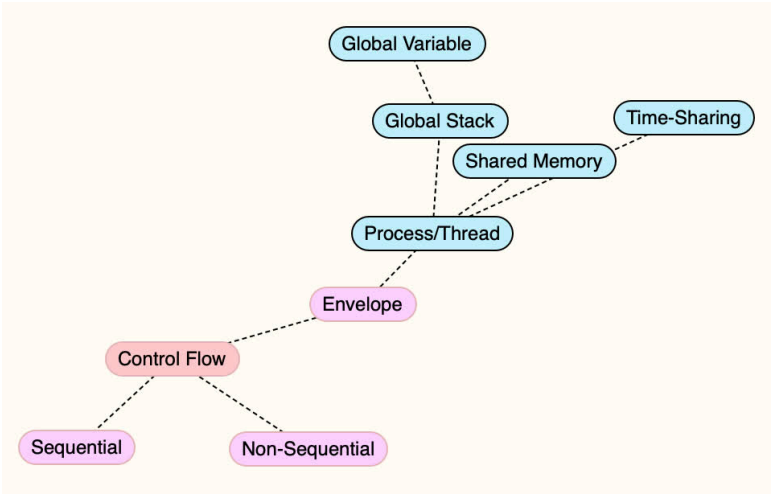
Control Flow 8



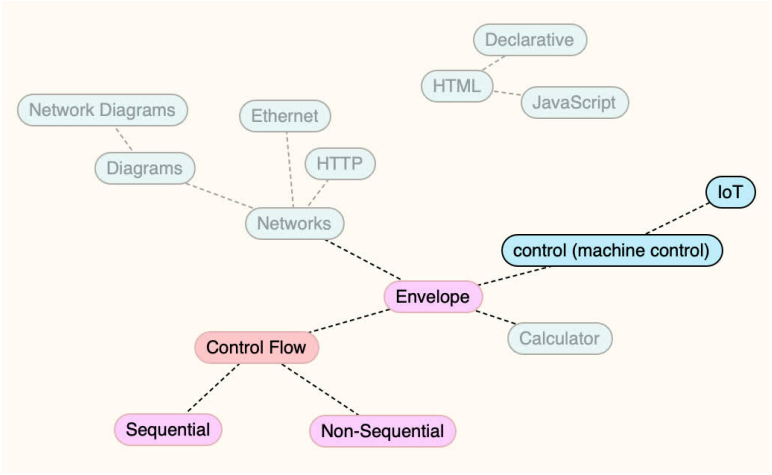
Control Flow 9



Control Flow 10



Control Flow 11



Control Flow 12

Transcript

I'm gonna talk about control flow. This is a map of where we're going. I'm gonna simplify it. I break control flow up into three main categories, sequential, non sequential, and then envelope or what we call multitasking these days.

- Sequential is where one action happens after another.
- Non-sequential control flow is where actions happen in an order that is different from one after another.
- Envelope is a non-sequential programming, but I think of it as being, higher level than non-sequential. Non-sequential breaks down into four main cases. Case call go to try throw and.

Call includes recursion and return. Some would say that, recursion is in fact the kind of, or is the, the uber parent of calling. And so I'll try to do that if I can. Here, there we go. Go to is the main way to change control flow in a, standard program. Go-to is not a problem, but unrestricted use of go-to is a problem.

Very early on we realized that, doing go-tos all over the place was a bad idea. We invented something called structured programming and that broke down into if then else for and while, and then a language called concurrent Euclid, in used something called a loop, which subs. Both for and while into one construct called the Loop Exit, when and end.

It might be noted that first class functions are really just a kind of go-to, which is even more unrestricted than the original go-to that we had. Continuation passing style comes from first class functions, and it is just the ultimate go-to that is available to us if we stop treating, go-to as something to be shunned and as a useful but very low level operation.

We can branch structured programming into other kinds of ways of, structured program. gave us only one way to constrain the use of go-tos. There are different ways, another way to constrain go-tos is by using state machines, they give much more flexible control flow const, or they give different flexible control flow constructs.

Structured programming gave us one way to constrain go-tos. There are different ways. . There are different ways to constrain go-tos than just using structured programming. One is state machines. Another one we'll see later is just, using an envelope. State machines are interesting, in that. In 1986, Harel wrote a paper called, about StateCharts.

I discuss state charts in my blog at this address. Try, throw and catch is a bag added onto the side of common programming languages to handle non-sequential, dynamic control flow chains. Co routines, albeit useful, have been left in the dust by most of our current programming languages.

Now we'll look at the, thing that I call an envelope, which breaks down into processes that are threads and networks.

looking at processes, we notice a, that they involve shared memory. They use a global variable called the stack, and they involve time sharing. All of these issues are issues based on early forms of computing. Back in the 1950s, CPUs and memory were expensive. These things have given us accidental complexities.

Now let's look at networks. This is the wave of the future of computing in my mind.

Networks break down at http and ethernet. Mostly, in networks. We also can draw sensible diagrams of things, although we haven't formalized it. Networking has, created a new breed of language. We call that language, HTML.

It is, declarative and it's not stack based, so it can, go across distributed processors. There are parts of HTML that we haven't figured out how to make declarative. So we use JavaScript and other

things, to help us, create program more interesting programs using HTML up to now we've been, trying to, shoehorn everything into a calculator style style of programming, embodied by the original use of computers, which was to, perform calculation ballistic calculations. A new form of computing, or a different form of computing is, distributed control flow or machine control. Tends to break the types of things we can do with current programming languages.

The most, distributed processing and distributed control is gonna be where IoT ends up, inform in Internet of Things. . Right now we're using, old fashioned techniques like, dropping this enormous library called Linux on top of small processors, when we could be shaving, these programs to, suit their purpose.

For example, controlling a refrigerator does not require all of Linux.