Project DOT

Forth Coding Practices for High Quality, Maintainable Software

The Problem

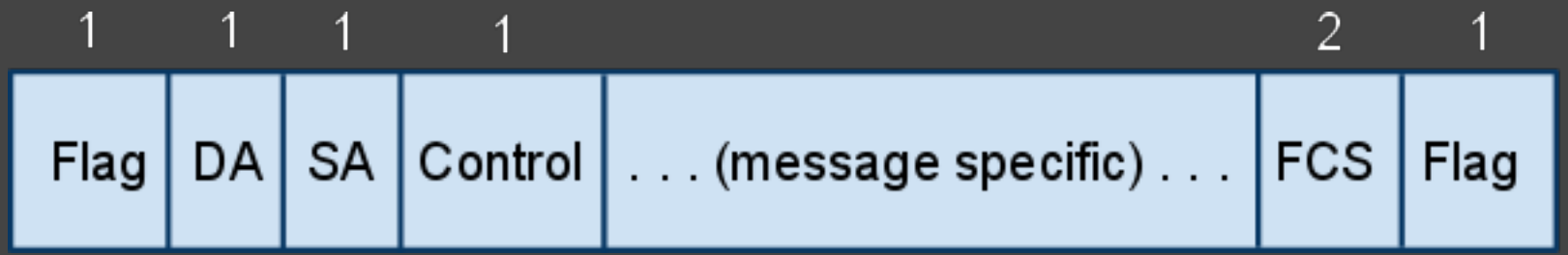
- PC/Arduino communications proved unreliable in practice.
- About one in one hundred bytes would drop due to timing inaccuracies between the Arduino and FTDI USB adapter.
- Reliable protocol between PC and Arduino thus necessary to ensure successful data exchange.

The Solution: AX.25 (sort of)

- AX.25 is a proper subset of ISO HDLC.
- Two 7-byte address fields instead of one 1-byte.
- No official support for modulo-32768, SNRM, or SARM modes of operation, but no impediments to supporting these either.
- In all other respects, AX.25 ***is*** HDLC.
- Probably the simplest to understand protocol specification to read since IP came out.

The Solution: AX.25 (sort of)

- Proven technology originating with IBM System/360 Mainframes in the form of SDLC, later standardized internationally as HDLC.
- It works in high-noise (amateur radio), high-latency (Earth orbiting satellite), low-noise (RS-422), and low-latency (SONET/SDH) environments without any significant changes.
- Sole difference: use two 1-byte addresses instead of 7-byte.

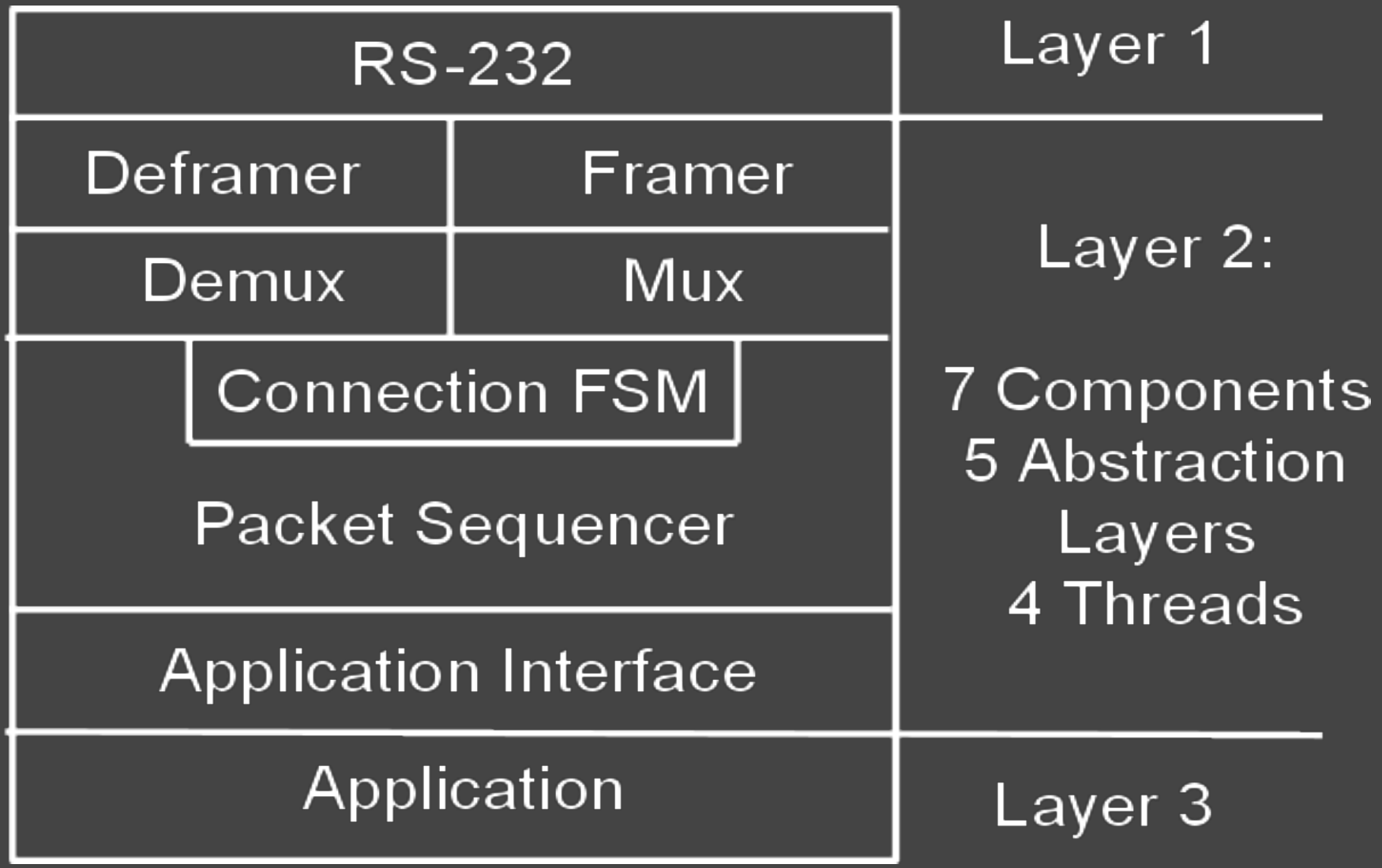


The Solution: AX.25 (sort of)

But, why AX.25/HDLC and not TCP/IP?!

- TCP has more complex state to maintain.
- Telemetry inherently message-oriented; TCP is byte-oriented.
- TCP requires explicit framing of data to delineate messages.
- Separation of TCP and IP functionality
  - I have to write more code.
  - Greater opportunity for mistakes.
  - ***FAR*** greater overhead (*40 to 60 bytes* versus only 5)
  - Checksums instead of CRCs

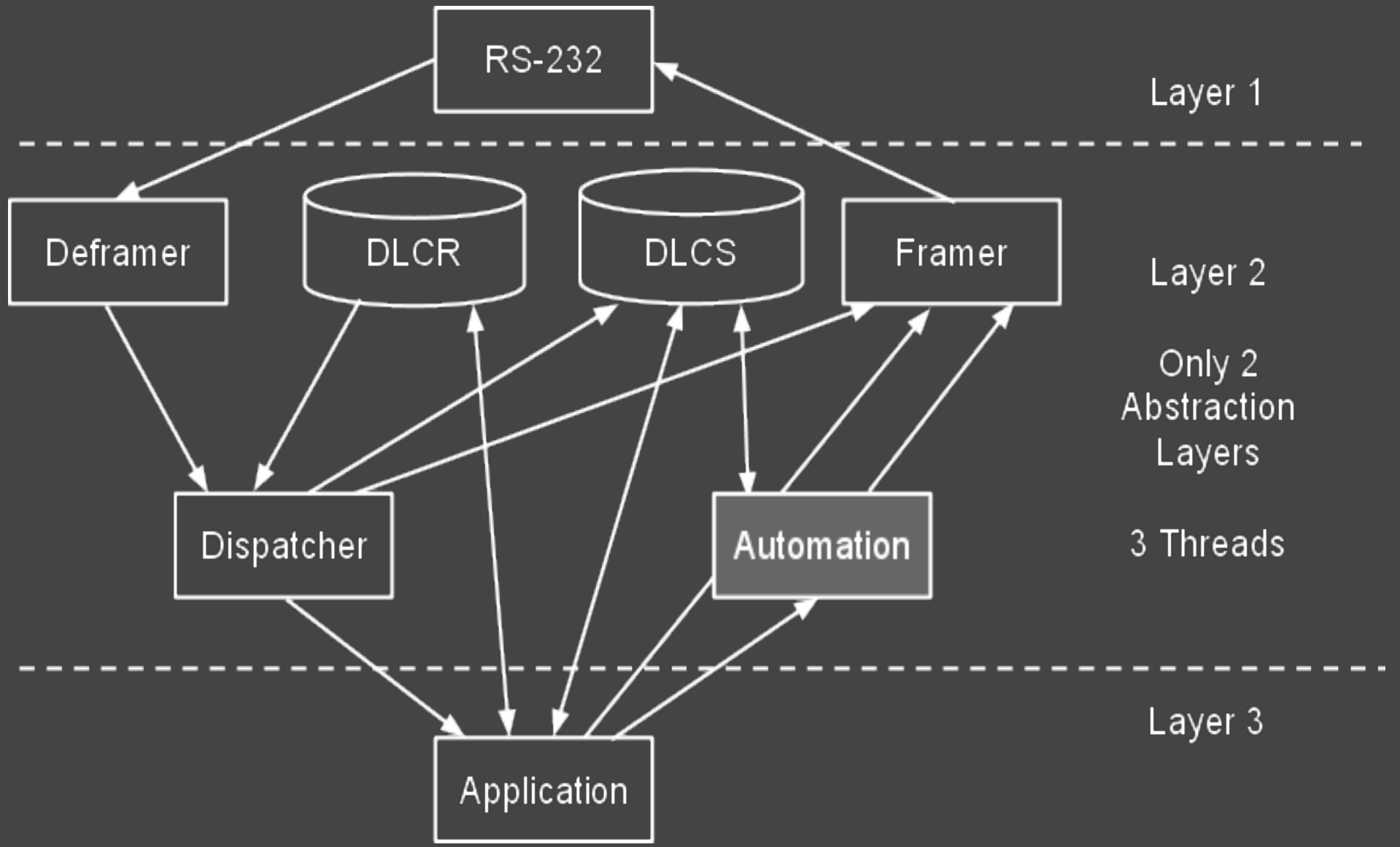
The First Cut



The First Cut: Why?

- Layers impose constraints on who can call what and when.
- Data, however, transcends layers!
- Structure-Centric Code (e.g., apps hold "handles" to "connections").
- Poor mapping to how HDLC *actually* works.
- Threat of race conditions between threads:
  - (D)MUX runs in (De)framer context.
  - Connection and packet sequencer needs another thread.
  - Application Interface runs in application context.

The Second Cut



The Second Cut: Why?

DLCR and DLCS are *Relational Variables* ("tables") equipped with a rather thin procedural abstraction on top.

Almost as a consequence of using relational algebra, code tends to *declare* (new) truth. As a rule, it doesn't have to command or ask politely.

Code is database-centric, not structure-centric. No need to pass pointers to parameter blocks everywhere!

Reduced constraints on module relationships.

The Second Cut: Why?

Dispatcher better matches how HDLC works best.

Note the distinct lack of a demultiplexor!!

Databases serve as a common interface between the different threading domains. Hence, synchronization between threads best kept factored in database access veneers.

Forth Design Patterns?!

Thinking Forth perhaps the first attempt at documenting Forth guidelines. But, it doesn't go far enough to *formalize* individual patterns.

I've detected and attempted to formalize six, without which the HDLC implementation would be substantially harder to write and maintain.

Upon reflection, these same patterns also appear in my Forth blog software!

Forth Design Patterns?!

1. Declarative, Imperative, then Inquisitive
2. Aggressive Handling
3. Partial Continuation
4. Ascetic Programming
5. Factor Indices Out
6. Demultiplex by Request

Declarative, Imperative, then Inquisitive

Intent

- Ease writing of software in such a way that it simultaneously facilitates easier reading and verification.

Motivation

- Forth's lack of static type safety eliminates compile-time sanity checking of your program.
- Only edit-time and run-time error discovery options remain.
- Problems found sooner makes coding easier and cheaper.
- Therefore, use edit-time conventions to eliminate errors.

Declarative, Imperative, then Inquisitive

Applicability

- Use DItI when you want to ensure the **highest possible quality**, greatest legibility, or both.

Structure

```
: foo   sensitive behavior ;
```

Declarative, Imperative, then Inquisitive

Applicability

- Use DItI when you want to ensure the **highest possible quality**, greatest legibility, or both.

Structure

```
: safe2do   safe? IF EXIT THEN error ;
: foo   safe2do sensitive behavior ;
```

Declarative, Imperative, then Inquisitive

Applicability

- Use DItI when you want to ensure the highest possible quality, **greatest legibility**, or both.

Structure

```
variable s
create stack   MAX cells allot
: pushed   st<=s<st+MAX   s @ !   1 cells s +! ;
```

Declarative, Imperative, then Inquisitive

Applicability

- Use DItl when you want to ensure the highest possible quality, **greatest legibility**, or both.

Structure

```
variable s
create stack   MAX cells allot
: pushed   st<=s<st+MAX   s @ !   1 cells s +! ;
: pushed?   -n.in.st  drop 0 ;
```

Declarative, Imperative, then Inquisitive

Collaborations

- 15% declarative words for public interface
- 70% imperative words for *internal implementation*
- 15% inquisitive words for public interfac

```
: -match   2dup st + @ = IF DROP R> R> 2DROP -1 THEN ;
: -n.in.st   0 begin dup MAX < while -match cell+ repeat drop ;
: pushed?   -n.in.st  drop 0 ;
```

Declarative, Imperative, then Inquisitive

Consequences

- Slower to execute, but easier to debug. Compare against the straight-ahead implementation:

```
: pushed?
    0 begin
        dup MAX >= if drop 0 exit then
        2dup st + @ = if drop -1 exit then
        cell+
    again ;
```

Declarative, Imperative, then Inquisitive

Consequences

- Many operations performed by a module share common preconditions, often with similar error-handling procedures for violations. DItI permits defining preconditions once, and re-using them multiple times elsewhere.
- Programs read more like specifications or conversations.

Declarative, Imperative, then Inquisitive

Sample Code

```
: SABM   +sabm +connectable +authorized  ( declarations )
         dup replyUA                      ( imperative )
         dup stations connected drop       ( declaration )
         reusable r> drop ;                ( declaration )

: U-frame   ( one of ) UA DM SABM ( else ) reusable ;
```

Declarative, Imperative, then Inquisitive

Related Patterns

- Partial Continuation
- Aggressive Handling

Aggressive Handling

Intent

- Handle an exceptional condition as close to the origin of the exception as possible.

Motivation

- High-level code excels at dictating policy.
- Low-level code excels at specific know-how.
- Procedural code relies on task-oriented design, and so high- and low-level code agree on intended task.
- Therefore, let the low-level code do its job in the context of the current task.

Aggressive Handling

Applicability

Use aggressive handling when:

- you can express how to handle an exceptional condition as a generic algorithm.
- you need errors dealt with *fast* .

Aggressive Handling

Structure

```
: guard   predicate IF EXIT THEN   handle exception ;
: guard   predicate IF . . . ELSE handle exception THEN ;
```

Collaborators

- The guard must deal with the exceptional condition upon detection.
- The predicate determines if the current situation is exceptional.

Aggressive Handling

Consequences

- Error handling is *very* fast; no need for dynamically dispatching or unwinding the stacks.
- Error handling policy is firmly set by the algorithm used in the handler. While specific elements of the handler's behavior might be satisfied through DEFER'ed words or other forms of generic programming in Forth, the over-arching algorithm can only be changed by a recompile of the guard.
- The code is substantially easier to read, for the conditional, consequent, and alternate appear together, readily available for static review.

Aggressive Handling

Sample Code

```
: .error    z" phy-rs232.f" perror abort ;
: 0<error   0< if .error then ;
: h!        dup 0<error 'h ! ;
: (serial   serial-port O_RDWR O_NONBLOCK open h! ;
```

If the HDLC stack cannot open the RS-232 device in Linux, then what's the point in continuing further? When such an event happens, we print the system-defined error message and quit the program outright.

Aggressive Handling

Sample Code

```
: |fbuf|-1>=0   dup |fbuf| 1- 0<
                abort" Attempt to read byte from empty buffer" ;

: -head         |fbuf|-1>=0 dup headc@ swap poph ;
```

We cannot read from a buffer if its length is zero. Therefore, any attempt to do so is a critical error in the system. Here, too, we ABORT, with the reason why.

Aggressive Handling

Related

- Partial Continuation
- Declarative, Imperative, then Inquisitive

Partial Continuation

Intent

- Exit from a deeply-nested control flow without having to transcend outer layers of software, each of which equipped with redundant exit- or error-code checking.

Motivation

- Words with complex logic have no easy way abort without dumping the user to the OK prompt.
- Sometimes, aggressive error handling proves insufficient, and customized or policy-based handling is desired.

Partial Continuation

Applicability

Use partial continuations when

- you want to escape from a deeply nested call stack to a specific caller higher up in the stack.
- you want to implement back-tracking algorithms, filters, iterators, generators, or other co-routine-like entities.
- you want to refactor common looping constructs to eliminate redundancy.

Partial Continuation

Structure

Use of partial continuations can take many forms, depending on the purpose.

```
: -bar   bar? if drop 0 r> drop then ;
: foo?   -bar drop -1 ;
```

By dropping the return address for -bar, which happens to be foo? itself, we ensure that we *return directly to whoever called foo?* .

Partial Continuation

Structure

```
: callback    >r ;
: evens:   r> -rot do i -odd i over callback loop drop ;
: .ev   evens: . ." is an even number." cr ;
100 0 .ev
```

Here we see evens: return multiple times for the one call-site it's used. It will print out all the even numbers in the range provided. This use of partial continuations permits a different kind of declarative programming style: *implicit loops.*

Partial Continuation

Collaborators

- The caller may never complete its operation.
- The caller may complete its operation *many times* , often while iterating over a sequence of things.
- The application who invokes the caller, in conjunction with the caller, are responsible for ensuring correct return-stack configuration prior to invoking the callee.

Partial Continuation

Consequences

- Partial continuations can lead to spaghetti code; inasmuch, treat them with respect, or risk making your software unmaintainable.
- Since the return stack is, in effect, used to pass parameters as well, you may find that additional "stack noise" is needed to help balance the stacks.
- Unlike CATCH/THROW in ANSI, no result codes are necessary for inter-layer communications. Aggressive handlers are free to return results directly to the application which invoked the caller if desired.

Partial Continuation

Sample Code

```
: append          buffer @ +tail ;
: -flag          dup $7E = if drop boundary r> drop then ;
: +buffer         buffer @ 0= if drop r> drop then ;
: character       -flag +buffer append ;
```

Related Patterns

- Declarative, Imperative, then Inquisitive
- Aggressive Handler

Ascetic Programming

Intent

- Enhance code re-usability through expression of algorithms at the most abstract level, without any regard to objects.

Motivation

- Code operating on data in a structure tends to get caught up with structures. Like `const` in C, pointers are contagious.
- Calling words in Forth becomes problematic due to:
  - Mixing pointers and intelligence on the data stack.
  - Invoking multiple operations requires multiple uses of a pointer, which creates stack noise and obscures intent.

Ascetic Programming

Applicability

Use ascetic programming if you

- maintain a collection of like entities.
- find passing pointers to procedures obscures the readability of your code.
- code in a language where arrays are the only aggregate type.
- work with the same data across several layers of abstraction.

Ascetic Programming

Structure

The caller, per DItI, wants to establish the knowledge that Sam is 35 years old, and so invokes:

```
35 S" Sam" aged
```

Ascetic Programming

Structure

The implementor of aged (the callee) maintains a *relational variable* corresponding age with username, indexing into table as needed.

```
create ages   /column allot
create names   /column 2* allot
    . . .
: inserted   +roomy  #r @ 2* names + 2dup ! cell+ nip !
                #r @ ages + !    1 cells #r +! ;
: +exists   2dup nameInserted? if exit then  inserted r> drop ;
: aged   +exists  namedRecord ages + ! ;
```

Ascetic Programming

Collaborations

- The caller typically wants to say something new, or update existing knowledge.
- The callee accepts this knowledge however the caller represents it, and performs the actions necessary to ensure relevant state updates or queries occur to satisfy the caller.

Ascetic Programming

Consequences

- As a general rule, object concepts are replaced by corresponding relational concepts.
- Substantially fewer pointers passed between procedures, greatly simplifying interfaces.
- Interfaces to modules tend to be more generic and re-usable, without the general need for templates.
- Type relationships modeled in database, not explicitly in code.
- Supports stupidly simple persistence mechanisms.
- If multithreaded, the code must deal with locking against concurrent access.
- Slower latencies than direct pointer dereferencing.

Ascetic Programming

Example Code -- Table definition

```
1 cells          constant /row
#dlcs /row *     constant /column

: column          /column allot ;
create           localA column
create           remoteA column
create           stat column
variable         nextDlc
```

Ascetic Programming

Example Code -- Basic Search and Hit Testing

```
: hit?        >r over remoteA r@ + @ =  over
              localA r@ + @ = and r> swap ;
: -match       hit? if nip nip r> r> 2drop exit then ;
: -found      0 begin dup nextDlc @ < while
              -match cell+ repeat drop ;
: row          -found  2drop 0 r> drop ;
```

Ascetic Programming

Example Code -- Basic Predicates

```
: isDlc?                row drop -1 ;
: disconnected?     isDlc? 0= ;
: is?                  -rot   row stat + @ = r> drop ;
: connecting?        DLCS_CONNECTING is? nip ;
: connected?        DLCS_CONNECTED is? nip ;
```

Ascetic Programming

Example Code -- Basic State Changes

```
: disconnected  row collapse ;
: change        >r localA r@ + ! remoteA r@ + ! stat r> + ! ;
: +absent       2dup isDlc? if 2dup row change r> drop -1 then ;
: +room         nextDlc @ /column = if 2drop drop r> drop 0 then ;
: update        -rot +absent +room
                nextDlc @ change /row nextDlc +! -1 ;
: connecting    DLCS_CONNECTING update ;
: connected     DLCS_CONNECTED update ;
```

Ascetic Programming

Example Code -- Client Use

```
1 2 connected? .   0 ok
1 2 connecting drop   ok
1 2 connected? .   0 ok
1 2 connecting? .   -1 ok

create callsignA   'K c, 'C c, '5 c, 'T c, 'J c, 'A c,
create callsignB   'K c, 'F c, '4 c, 'F c, 'S c, 'E c,

callsignA callsignB connected drop   ok
callsignA callsignB connected? .   -1 ok
```

Ascetic Programming

Related Patterns

- Declarative, Imperative, then Inquisition
- Partial Continuation
- Factor Indices

Factor Indices

Intent

- Focus on core intelligence by letting records state only facts, while concurrently maintaining rapid access to relevant records in other knowledge stores.

Motivation

- Full-table scans of relational data requires O(n) time worst-case.
- If a search criteria ("foreign key") is used often, or when accessing slow media, using a structure with $O(\log_2 n)$ time can save $k_1 n - k_2 \log_2 n$ time.

Factor Indices

Applicability

You want to factor indices out from normal data when

- absolute, break-neck performance is unnecessary, but more naive approaches to finding data proves too slow to be of value.
- storing data on rotating media, or otherwise much slower media than RAM. Keeping relevant indices in RAM can eliminate access overheads.
- you want to access data in one of several prescribed sort orders (e.g., ascending/descending, by priority, etc.) without moving records around in memory.

Factor Indices

Structure

Too numerous to list here. Google for B-tree, skiplists, binary trees, hash tables, and more!!
Common characteristics:

- Relational records composed predominantly of business logic.
- Index records composed almost exclusively of pointers and, if appropriate, (subsets of) candidate keys.

Factor Indices

Collaborations

- Each record in a relational variable states a fact about some relevant portion of your business.
- One or more index records may refer to any given record.
- As updates to knowledge tables occur, indices must be updated as a necessary precondition to returning to the caller.
- The callee is, except for the time measured to perform a database operation, wholesale ignorant of what kinds of indices exist, if any at all.

Factor Indices

Consequences

- Inserts of fresh data into tables slows down, as the need to update indices as an atomic operation becomes necessary.
- Updates and deletions of existing data *may* slow down, depending on the nature of the change, as search criteria might change, thus necessitating an update to all relevant indices.
- Searches become substantially faster, as unnecessary accesses to slow media or RAM are removed.
- Indices consume greater amounts of storage space, and must be provisioned like normal storage space.
- Although remaining factored, code becomes more complex.

Factor Indices

Sample Code

This example comes from my Unsuitable blog, as I have no need for indices in my HDLC codebase at this time.

First, I allocate space for the index:

```
create e0  5 cells allot
e0 5 cells -1 fill
e0 4 cells + constant en-1
en-1 cell+ constant en
variable ep
```

Factor Indices

Sample Code

Then, I use an insertion sort on the index while doing a full-table scan of articles to get the n most recent articles posted:

```
: insert     ep @ dup cell+ over en-1 swap - move ;
: nil        ep @ @ -1 = if article ep @ ! r> drop then ;
: eol        ep @ en >= if r> drop then ;
: Te<Tr    article ep @ @ article! timestamp swap article!
           timestamp < if insert article ep @ ! r> drop then ;
: Te>=Tr  [ 1 cells ] literal ep +! ;
```

(continued . . .)

Factor Indices

Sample Code

(. . . continued)

```
: sort         e0 ep ! begin eol nil Te<Tr Te>=Tr again ;
: consider  articleId -1 xor if sort then ;
: scan      articleIds dup /afields + begin 2dup < while
              over articleIds - article! consider swap cell+ swap
              repeat 2drop ;
```

Note how much harder it is to understand this code versus the HDLC code? It's because I wrote it before identifying the DItI and aggressive handling patterns. :-)

Factor Indices

Related Patterns

- Ascetic Programming

Demultiplex by Request

Intent

- Simpify state changes in one of several instances of a finite state machine.

Motivation

- Many protocols depend on destination to select a state machine instance. However, responses to events across all instances remains the same. Therefore, demultiplexing on destination address first makes things unnecessarily complex.

Demultiplex by Request

Applicability

Demultiplexing by request pays off when

- you manage a plurality of autonomous entities, all of which behave similarly to like events.
- you want to implement object-oriented semantics.

Demultiplex by Request

Structure

Message requests can be encoded through:

- integers passed as a parameter (e.g., Win32 message handlers); code dispatches via a "switch" statement.
- offsets from a jump table (e.g., C++ virtual method tables; hardware dispatches through indirect jumps
- bytecodes embedded in data structures (e.g., HDLC "control" field); software dispatches through either indirection or "switch"-like construct, as appropriate.

Demultiplex by Request

Collaborators

- The caller and callee must agree on a common protocol.
- The caller is responsible for "marshaling" messages into a format readable by the callee.
- The callee is responsible for interpreting the message, acting accordingly, and if required, providing a response.

Demultiplex by Request

Consequences

- Satisfying requests take longer because of runtime resolution of behavior.
- Invoked behavior no longer tied to request ID, and can even change over time.

Demultiplex by Request

Sample Code

```
: U-frame   ( one of ) UA DM SABM ( else ) reusable ;

create table0
    ' I-frame , ' I-frame , ' S-frame , ' U-frame ,

: handler     .control swap +c@ 3 and cells table0 + @ ;
: dispatch    dup handler execute ;
```

Demultiplex by Request

Related Patterns

- Declaration, Imperative, then Inquisitive
- Object Orientation

The End

(or is it just The Beginning?)

Thank You!