Design of Routines
// Enqueue byte into current position and increment transmission length.
#define _ENQUEUE_BYTE(byte) do { \
if(byte == ZephyrEncoding::DELIMITER_BYTE) { \
    buffer[pos++] = ZephyrEncoding::ESCAPE_BYTE; \
    buffer[pos++] = 'd'; \
} else if(byte == ZephyrEncoding::ESCAPE_BYTE) { \
    buffer[pos++] = ZephyrEncoding::ESCAPE_BYTE; \
    buffer[pos++] = ZephyrEncoding::ESCAPE_BYTE; \
} else { \
    buffer[pos++] = byte; \
} \
} while(0);

_ENQUEUE_BYTE(idLSB);
_ENQUEUE_BYTE(idMSB);
_ENQUEUE_BYTE((uint8_t)(msg.len));
for(int i = 0; i < msg.len; i++) {
    uint8_t b = msg.data[i];
    _ENQUEUE_BYTE(b);
}
// Add a delimiter byte (and ensure it doesn't get escaped)
buffer[pos++] = ZephyrEncoding::DELIMITER_BYTE;

return pos;
Valid reasons to Create a Routine

What is the author’s single most important reason?

A) hide sequences
B) support sub-classing
C) avoid duplicate code
D) reduce complexity
E) improve portability
It is okay to have small functions

- Self-documenting code
- Avoid duplication
  - eases maintenance
Cohesion

- **Strive for Functional Cohesion**
  - A routine that performs exactly one operation
  - Usually documentable with clear method name
- **High cohesion correlated with low fault rates / fix costs**

- **Fixing low cohesion?**
  - Break offending method into cohesive pieces
  - Call the pieces in turn.
A* example

function A*(start, goal)
    closedSet := {}
    openSet := {start}
    cameFrom := the empty map
    gScore := map with default value of Infinity
    gScore[start] := 0
    fScore := map with default value of Infinity
    fScore[start] := heuristic_cost_estimate(start, goal)

    while openSet is not empty
        current := the node in openSet having the lowest fScore[] value
        if current = goal
            return reconstruct_path(cameFrom, current)
        openSet.Remove(current)
        closedSet.Add(current)
        for each neighbor of current
            if neighbor in closedSet
                continue
            tentative_gScore := gScore[current] + dist_between(current, neighbor)
            if neighbor not in openSet // Discover a new node
                openSet.Add(neighbor)
            else if tentative_gScore >= gScore[neighbor]
                continue
            cameFrom[neighbor] := current
            gScore[neighbor] := tentative_gScore
            fScore[neighbor] := gScore[neighbor] + heuristic_cost_estimate(neighbor, goal)
        return failure
Routine length (LOC = lines of code)

- Data cited in the book indicated in general

A) Longer routines (100 – 250 LOC) had lower error rates than shorter routines (< 50 LOC).

B) Shorter routines (< 50 LOC) had lower error rates than longer routines (100-250 LOC)

C) Data was inconclusive relating error rate and routine length.
So how to choose a routine length?

- Let cohesion and complexity be your guide
  - Pull out highly cohesive routines
  - Limit the number of paths through the code
Cache-conscious Matrix Multiply code

/* Multiply n x n matrices a and b */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i += B)
        for (j = 0; j < n; j += B)
            for (k = 0; k < n; k += B)
                /* B x B mini matrix multiplications */
                for (i1 = i; i1 < (i+B); i1++)
                    for (j1 = j; j1 < (j+B); j1++)
                        for (k1 = k; k1 < (k+B); k1++)
                            c[i1*n+j1] += a[i1*n + k1] * b[k1*n + j1];
}
Parameters

- logical ordering (e.g., input-modify-output)
- consistency
- use all of the parameters
- limit number of parameters to ~7
- put status or error parameter last
- don't use input parameters as working parameters
Example from book

```c
int Sample(int inputVal) {
    int workingVal = inputVal;
    workingVal = workingVal * CurrentMultiplier(workingVal);
    workingVal = workingVal + CurrentAdder(workingVal);
    ...
    return workingVal;
}
```

```
int Sample(int inputVal) {
    inputVal = inputVal * CurrentMultiplier(inputVal);
    inputVal = inputVal + CurrentAdder(inputVal);
    ...
    return inputVal;
}
```
Just-in Time (JIT) Compilers
To Dos for Thursday

- No reading assignment!
  - We’ll talk about design patterns

- For next code review: GoFish!
  - Implement a system that plays “Go Fish”
  - Input: number of players; plays game to completion
  - On each player’s turn:
    - They pick a random opponent, o
    - They pick a random card, c, from their hand
    - Ask o if they have any cards with c’s rank
Go Fish

Each player gets five cards. If you are dealt a four of a kind, or get four of a kind during game play, those cards are removed from your hand, and you get a point.

Moving clockwise, players take turns asking a specific player for a given rank of card. If someone asks you for a rank that you have, the cards are taken from your hand. If you do not have any cards of that rank, your opponent must “go fish”, taking one new card from the pile of cards.

When it’s your turn, select a player you think might have a needed card. Pick one card from your hand of the desired rank. If the player has the desired card, he or she must pass it over. If not, you must “go fish”. If you get the card you asked for, you get to go again.

If you run out of cards and there are still cards left, you get five free cards.

Play continues until all hands are empty and there are no more cards to draw from. The winner is the player with the most points at the end of the game.