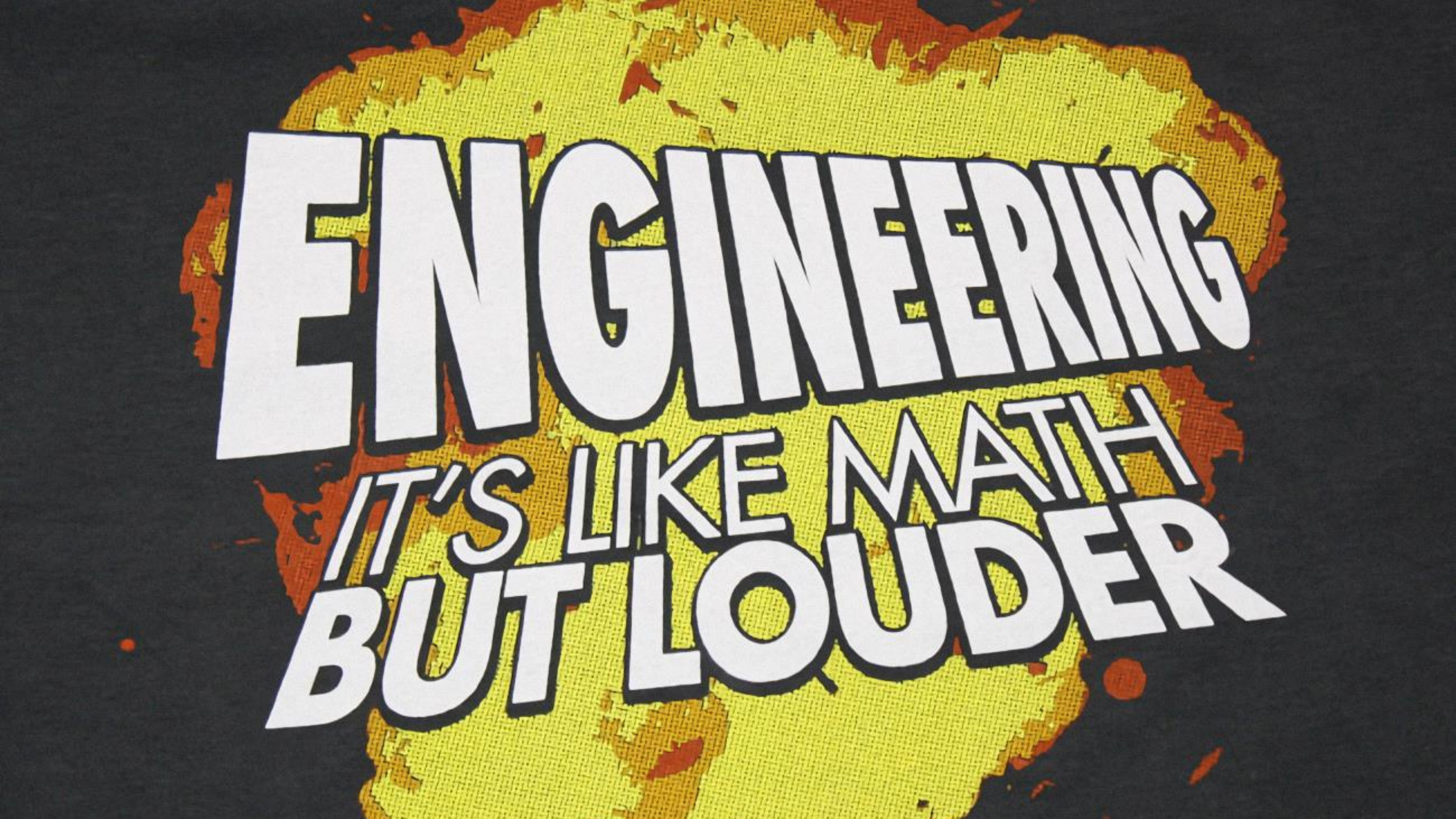

ENGINEERING
IT'S LIKE MATH
BUT LOUDER

Have you tried turning it off and on again

Defining success and failure

The reasons things go wrong

- The expected reasons: External dependencies
  - Network (can't connect)
  - Files (don't exist)
  - Memory (not enough)
  - Handled with: exceptions, error returns, etc
- The unexpected reasons: Bugs
  - "Algorithm" bugs (implementation was incorrect)
  - "Specification" bugs (implementation didn't exist)
  - Handled with...?
    - Testing? Too late for that now!

The three observers of our error

- The developer
  - Wants to fix bugs
  - Wants fallback strategies when external dependencies fail
- The user
  - Wants an error message
- The program
  - Wants to continue
  - Wants to ignore errors or self-heal

Our mechanisms should satisfy **all** of these… ideally

When things go worng

- We want unification:
  - Detecting all things that go wrong
  - Handling/recovering from all things that went wrong
- We want little code:
  - "Happy path" code should not be swamped by error code
  - Don't burden the programmer too much
    - More likely to be written in the first place
    - More likely to be correct
    - Less mental overhead when programming the happy stuff

Identifying the unacceptable

- An operation has two outcomes
  - It works properly
  - It doesn't work properly
- Our detection mechanism has two outcomes
  - We accept the operation
  - We reject the operation

- Let's make the obvious figure...

Identifying the unacceptable - 2

		Detection mechanism outcomes	
		We accept it:	We reject it:
Operation outcomes	It works:	Correctly identified successful operations	Failed to identify successful operations
	It doesn't work:	Failed to identify failed operations	Correctly identified failed operations

Identifying the unacceptable - 3

- "Traditional" error detection aims to detect failure
  - Detecting (and rejecting) expected failure is easy
  - But detecting unexpected failure (bugs) is impossible
- Acceptance tests aim to detect success
  - Detecting (and accepting) expected success is easy
  - But failing to detect unexpected success is **not** a problem

The path of failure

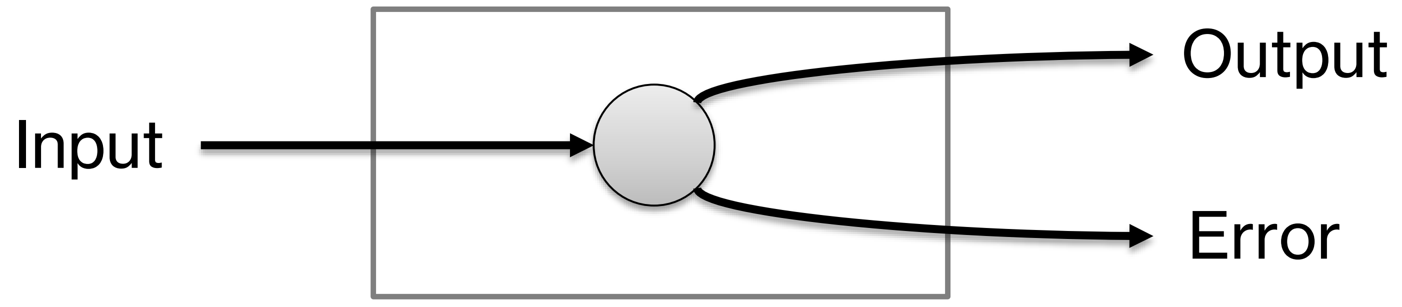
Error returns

- An example of "traditional" error detection
  - From linux source (cifs_smb3_do_mount)
    - https://github.com/torvalds/linux/blob/master/fs/cifs/cifsfs.c#L712-L810
    - Notice the appropriate use of goto
      - Error is returned as data
      - Control flow and data flow are aligned

Returns vs Exceptions

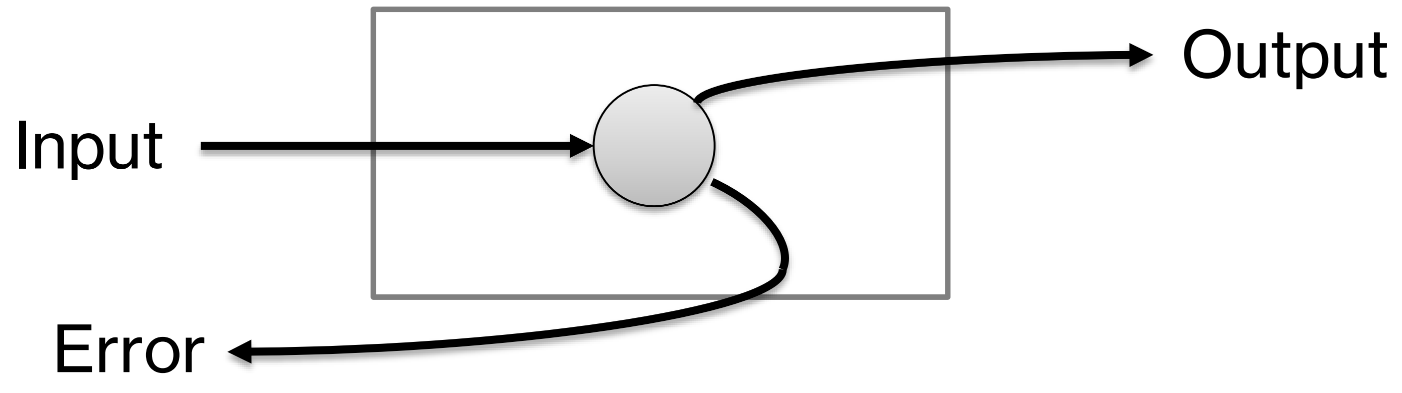
Always align data flow and control flow

Error returns:



Control flow should go forwards

Exceptions:



Control flow should go **backwards!**

Exceptions (the proper way)

Exceptions are "hard" because try-catch flow goes forward

Use scope/defer instead

```
try {
    doA();
    try {
        doB();
    } catch(Exception e){
        undoB();
        throw e;
    }
} catch(Exception e){
    undoA();
}
```

```
scope(failure) undoA();
doA();

scope(failure) undoB();
doB();
```

Exceptions (the wrong way)

- From "Cleaner, more elegant, and wrong"
  - https://blogs.msdn.microsoft.com/oldnewthing/20040422-00/?p=39683
  - Spot all the things that are wrong!
    - Uses try-catch
      - Wrong program flow for the data flow - not the fault of exceptions
    - Applies state transformation in incremental steps
      - Multiple sources of errors but only one handler - again, not the fault of exceptions
      - Doesn't split into core/shell
      - Creates some untestable unfathomable Object Oriented abortion
    - Duplicates data
      - Either let the inconsistency be ok
      - Or use transactions you fucking bot
      - Again: Not the fault of exceptions

The code paths of failure

The "happy path" uses standard control structures

`if` / `for` / `switch` / etc

Errors are when we can't follow the happy path

Use non-standard control structures!

`goto` is not the enemy

Finding success in failure

Writing acceptance tests

- Just use `assert()` a lot
  - (Or `if()` if you want forwards-flow)
  - Easy to write
    - Figuring out the logic for them is rarely hard
  - Low effort
    - No boilerplate
    - No extra `err` or `Exception` variables
  - Doesn't look like dogshit
    - No extra nesting and curly braces

Acceptance tests example

From the simulator

```
invariant {
    assert(-1 <= currFloor  && currFloor < numFloors,
        "currFloor is not between -1..numFloors");
    assert(0 <= prevFloor  && prevFloor < numFloors,
        "prevFloor is not between 0..numFloors");
    assert(departDirn != Dirn.Stop,
        "departDirn is Dirn.Stop");
}

(FloorArrival f){
    assert(state.currDirn != Dirn.Stop,
        "Elevator arrived at a floor with currDirn == Dirn.Stop");
    assert(0 <= f   &&  f <= state.numFloors,
        "Elevator \"arrived\" at a non-existent floor\n");
    assert(
        (state.currDirn == Dirn.Up   && f >= state.prevFloor) ||
        (state.currDirn == Dirn.Down && f <= state.prevFloor),
        "Elevator arrived at a floor in the opposite direction of travel\n");
    assert(abs(f - state.prevFloor) <= 1,
        "Elevator skipped a floor");
    .....
```

Handling the unacceptable

- Acceptance tests that fail just identify that "it didn't work"
  - The things we get:
    - Where it didn't work (file and line, useful for the developer)
    - Hopefully an error message written by the developer
  - The things that are still missing:
    - Exactly what originally caused the problem
    - A way to recover and heal
- How do we self-heal from an arbitrary unknown failure?
  - Corollary: what's the worst unknown failure?
    - Answer: The operation crashes
    - (Answer 2: The crash propagates and crashes other things too)
- Solution: Just restart it!

Writing restarting-mechanisms

Restarting execution

(Requires "I am alive" messages)

- Supervisor
  - One program is responsible for keeping the other alive
- Process pair
  - The program creates its own duplicate backup

Ensuring data integrity

(Requires that the data passes the acceptance test)

- Checkpoint-restart
  - Copy of data written to persistent storage
- Checkpoint-message
  - "I am alive" message contains data
- Persistent
  - Process operates on a "database" (both read and write)

Integrating with core/shell

- The only things that can fail in a process are
  - Writing to the outside world
  - Your own acceptance tests
- We want to preserve the integrity of the process and its outputs
  - "Robustness against restarting"
- Some guidelines:
  - Put the acceptance test in (or right after) the core functions
  - Do **all** acceptance tests before **any** state transformation or output
  - Stay together
    - Don't apply outputs that must go together in two separate places
      - You are about to fail at doing transactions
    - Don't spread data structures that must go together

Propagation in trees

Overkill

- Crashing the entire program is probably overkill
- Want to crash only a small part:
  - Want other operations to be minimally affected
    - Ongoing operations complete as normal
    - Interactions with restarting operations are delayed but not killed
  - Want restarting to be fast
    - Insert the obvious interpretation here
    - Want startup to be un-sequenced so the programmer isn't tempted to create dependency hell

Interactions

- If components interact in the "happy path"
  - Then components must interact during error handling
- Minimize error interactions by minimizing **all** interactions
  - Do **not** invoke behavior in other components

Hey look, it's communicating processes again...

Supervisors

- Processes are spawned hierarchically
- This gives us a few restarting-strategies:
  - One-for-one:
    - Restart just the child process that dies
  - One-for-all:
    - If a child dies, restart all children
  - Rest-for-one:
    - If a child dies, restart it and the younger ones
- Process death needs to be communicated ("I am alive" or similar)
  - You really want language/library support for this...