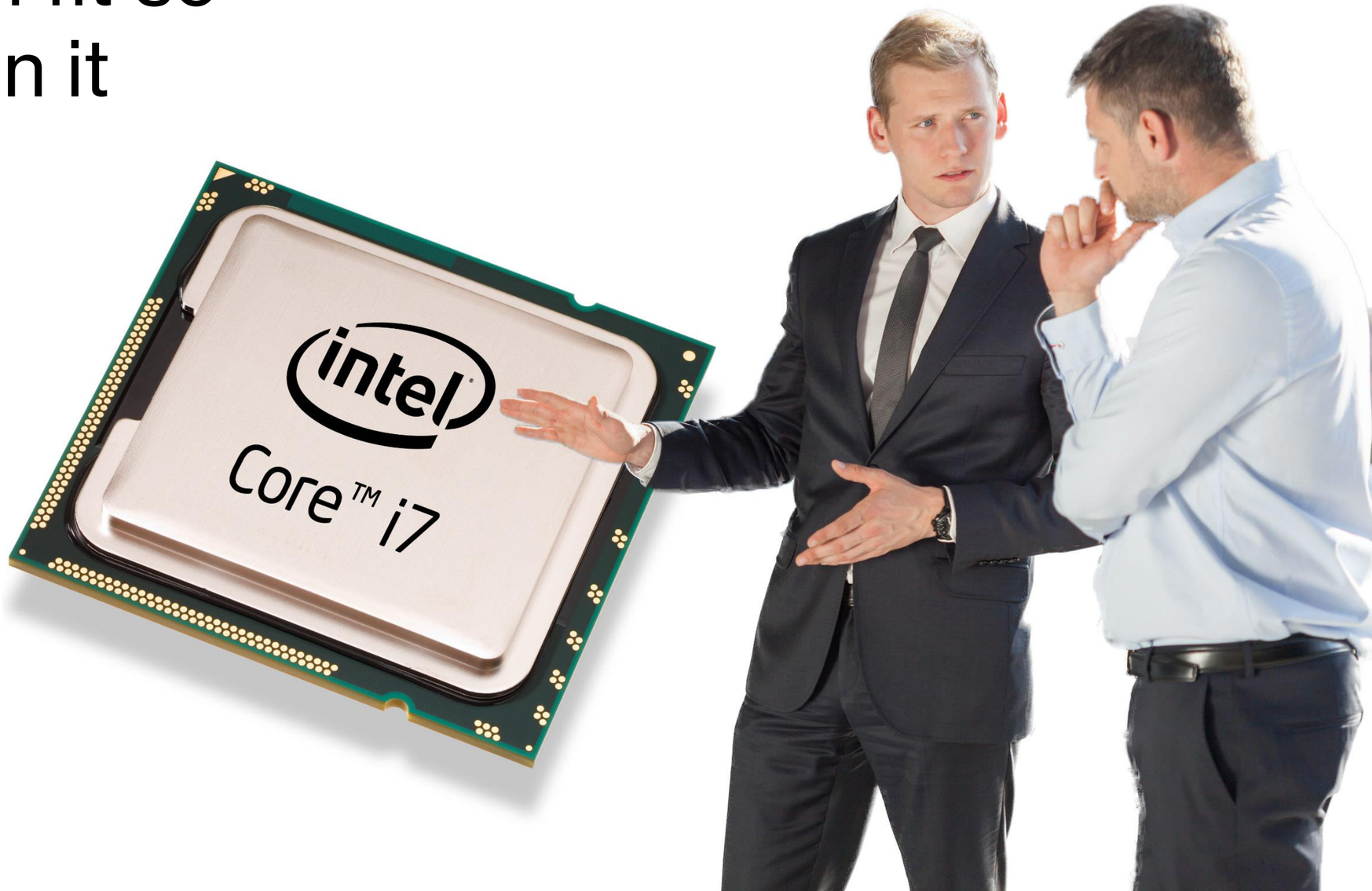
Slaps roof of processor

This bad boy can fit so much spaghetti in it



Structuring for testability

Computational functionality hierarchy

- Compiler/language
  - Type system
  - Formal verification
- Development testing
  - Unit tests
    - Testing single functions
  - Integration tests
    - Testing sets of functions with "real-world" data
- Runtime testing
  - Generic error handling
    - Deals with a single error
  - Acceptance tests
    - Detects entire classes of errors
  - Process pairs/supervisors
    - Handles complete or partial program restart
- Redundant computation
  - N-version programming

What makes code testable?

```
// Example 1
int fn(int a, int b, int c){
    return a + b - c;
}
```

```
// Example 2
static int c = 0;
int fn(int a){
    static int b = 0;
    return a + b - c;
}
```

```
// Example 3
static int c = 0;

int fn(int a){
    return a + b() - c;
}

int b(void){
    return c++;
}
```

What makes testing code **hard**

- Not knowing where values "come from"
  - Not knowing what internal values could be
  - Not knowing what external values could be
  - Implicit dependencies vs explicit parametrization
- Not knowing what is saved for next time
  - Needing to call functions in some order to test some scenario
  - It's "ordering in time" time again...
- Interaction with other components
  - Implicit interactions, or stateful interactions

What makes testing code **easy**

Functions that only have inputs and outputs

- No internal or external state
- Pure functions

Functions that are disassociated with data structure

- Isolated from other components

We are testing code *with* data, not code *and* data

Test doubles

- Components interact, and we need to test that
  - Replace other components with doubles?
    - Proxies, mocks, stubs, etc
  - Benefits:
    - Gives us the isolation we wanted
    - Allows for more testing
    - Tools and type systems for automated testing
  - Drawback:
    - We're not testing with the "real thing"
      - Proxy objects may go out "out of sync"
  - Why not just test the whole thing?
    - It's combinatorial explosion time

The boundary problem

The other components have an interaction surface

(Aka the "interface")

This must be "real stuff", not fakes

(And without the need to connect everything)

It must not be stateful

No destructive calls from/into other components

This boundary must be just values

Just data, which is immutable

"Pass data to other components"

NOT "invoke behavior in other components"

The immutability problem

Pure functions are easy to test, but can't change anything

Immutable values are easy to construct, but can't be changed

We can't make a program like this...

We want to introduce mutation, state, time, etc

But in a targeted and minimal way

Where should this stateful data live?

Who should mutate this data (and when)?

Hierarchies of parametrization

- Implicit dependencies were bad
  - Made them explicit with parametrization
    - (Global or static variables put into parameter lists)
    - We "lifted" them to the caller
      - This also shifts responsibility

- The parameters are the program state
  - How high do we lift the parameters?
  - How do we connect parts that have separate state?

Hierarchies of parametrization - 2

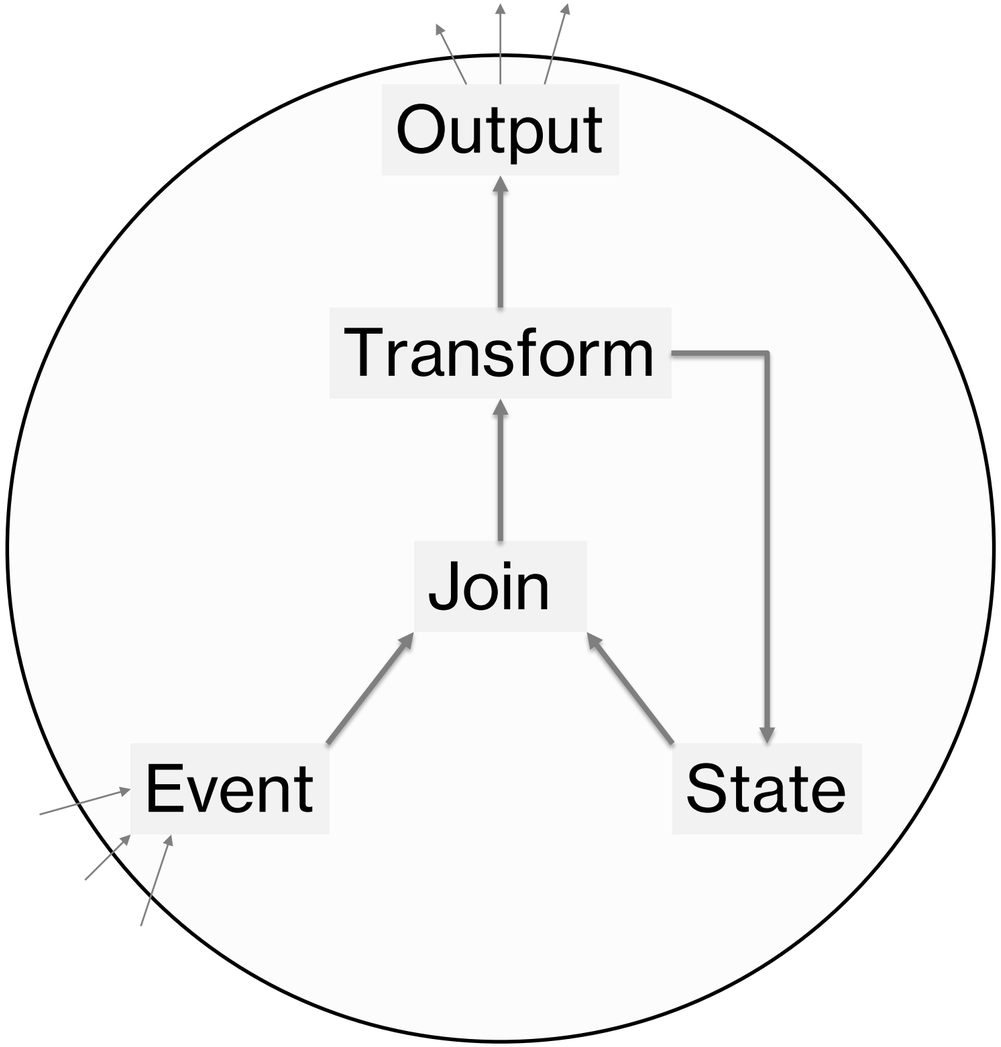
- One extreme: God object
  - All data centralized in one place
    - The "database"
  - Need to have a full understanding of all data
    - Cognitive overload
- The other extreme: Spread all over
  - All the data is separated as much as possible
  - Often need to modify multiple pieces of data simultaneously
    - This means reaching into other components and modifying their state
    - One-to-many and many-to-one code-data relations

Hierarchies of parametrization - 3

The middle road: Data divided by the mutator (writer)

- The data is useful because it is being mutated
  - Find the least amount of data that needs to be mutated at one time
    - This gives us a one-to-a-few code-data relation
  - This is the state for one "stateful process"

- Want good properties like
  - Uniqueness of ownership
  - Single responsibility
  - Uniform abstraction
  - -> "Things that belong together"

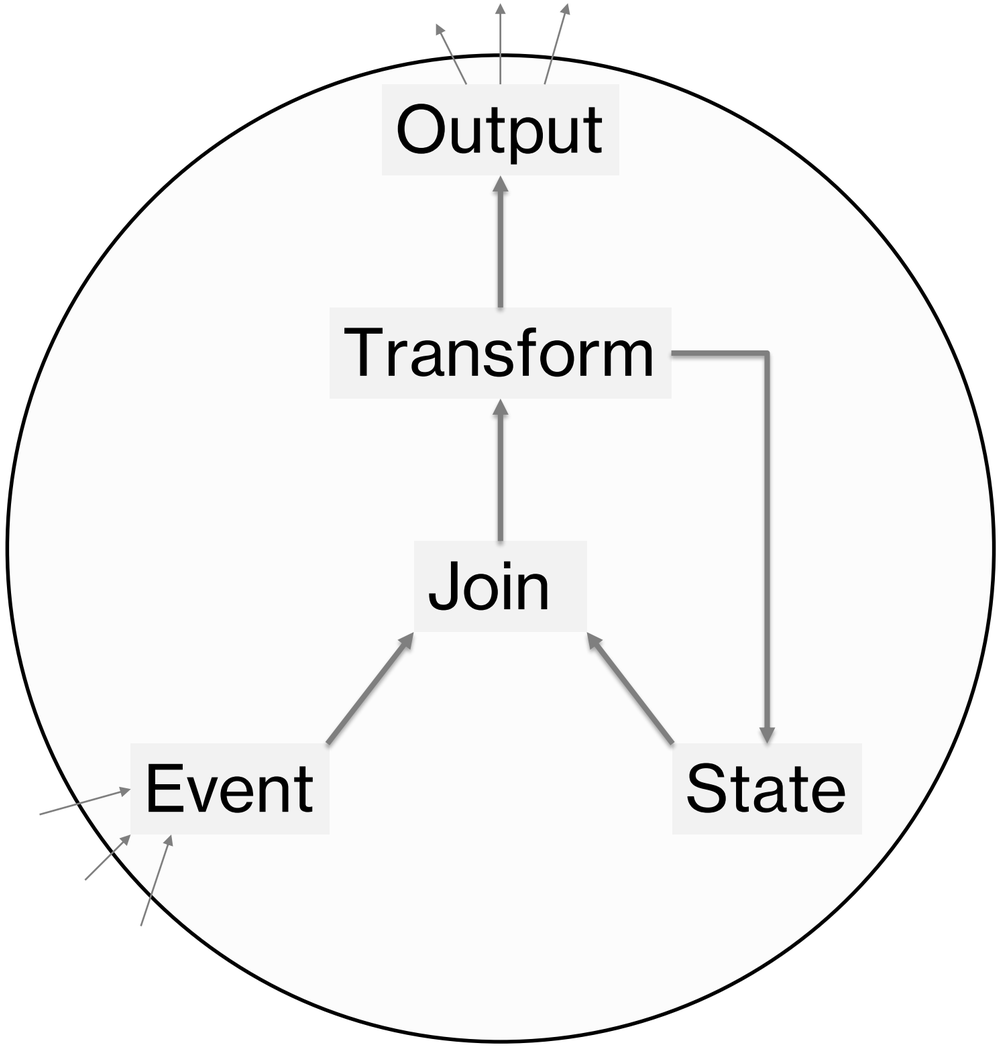


Pure functions:

```
output = fn1(input, state)
state  = fn2(input, state)
```

The combined approach

- The core:
  - Pure functions
  - Lots of paths, no dependencies
  - Perfect for unit testing
- The shell:
  - Stateful processes
  - Very few paths, lots of dependencies
  - Good for integration testing
    - But might not even need testing
- Boundaries between core functions and shells are values



Shell / Core

```
switch event {
  switch/if state {
    state  = purefn(ev, st)
    output = purefn(ev, st)
  }
}
```

Nesting order less important

- Just make as much as possible core
- Be pragmatic

The approach

1: Find the inputs and outputs

2: Find the data

(usually from what inputs and outputs need to persist)

3: Find which inputs/events mutate which data

And segment the data based on this

4: Create a persistent process for each set of data

5: Wire it together with queues or channels

And only send values over them

6: Iterate

State machine analysis

Specification

- Single elevator
  - Standard up/down-continue operation
  - Button lights, door (light)
- Stop button (not in TTK4145)
  - On press: Stop elevator, delete all requests and clear lights
    - Door should open if stop button pressed while at a floor
  - On release: Do nothing, but resume operation on new request

Inputs

These should be events

- Easier to reason about: Things only happen once
- Easier to debug: for exactly the same reason

The events:

button press	(and release?)
floor arrival	(and departure?)
stop button	(press and release)
timers	(time is always useful)

Outputs

These are the end goals of all decisions/algorithms/etc

- Shell-level functions should manipulate/write to outputs
- Core-level functions should return what manipulations should be performed

The outputs:

- Motor direction
  - stop, up & down
- door
- floor indicator
- button lights
- stop button light

Data / state

- Some data must be retained/stored between subsequent events
- Stored data/state should be minimal
  - Find the simplest way to store the least amount of data
  - Creating functions that manipulate simple representations of data is often easier
    - More complex representations can always be generated, if necessary
- The data must come from the events
  - There should be a direct mapping of {event + state -> new state}
  - This means there may be some data representing the state of the outputs
- The list of required data evolves over time
  - Have a very good rationale when adding new data
  - Try to simplify as much as possible
    - Avoid duplicates (inconsistencies waiting to happen)
    - Avoid invalid combinations (from specification)

The data - 1

- Want as much boolean data "as possible"
  - For later reduction

- From inputs/events explicitly:
  - What buttons have been pressed
    - 2D table is simplest (type vs floor)
  - What floor we are on (previously)
    - integer
  - If the stop button is pressed
    - boolean

The data - 2

From outputs implicitly:

- If we are moving
  - boolean
- If we are currently at a floor
  - boolean
  - (Spec: These two are different: may be standing still between floors)
- What direction we are travelling in
  - Custom {down, stop, up}
  - (Spec: up and down buttons should behave differently)
  - (Algo: prefer continuing in the current travel-direction)
- If the door is open
  - boolean
- What button lights are on, floor indicator
  - Not necessary, 1-to-1 from events

Minimizing the data - 1

Find undesirable combinations from the specification

at floor	stop btn	moving	door open	
		true	true	Moving with the door open
false			true	Door open when between floors
	true	true		Stop button pressed while moving (we still have the stop button press as an event)
true		true		Moving while at a floor (we still have floor arrival events)
true	true	false	false	Emergency at a floor with the door closed

Minimizing the data - 2

From 2^4 = 16 combinations to 6 remaining

at floor	stop btn	moving	door open	
0	0	0	0	Standstill (after emergency)
0	0	1	0	Moving
0	1	0	0	Emergency in shaft
1	0	0	0	Idle
1	0	0	1	Door open
1	1	0	1	Emergency at a floor

Behaviors

- Collapse data by removing duplicates & invalid combinations
  - This variable represents a behavior:
    - The system behaves differently (reacts differently to events) depending on this value
- Collapse further with dont-care's
  - Standstill (between floors) and Idle (at floor) are quite similar
- It may be beneficial to include/"fold" other data into this variable
  - (for the sake of readability)
  - But only if there are other fundamental behaviors that must be modelled

  - Other behaviors(?):
    - Init: floor == -1
      - We can (uniquely) transition to Idle when we arrive at a floor the first time

New minimal data

Buttons

3x4 array of bool

Floor

Integer

Direction

Enum of {down, stop, up} = {-1, 0, 1}

Behavior/state

Enum of {Init, Idle, DoorOpen, Moving, EmgAtFloor, EmgInShaft}

Hey look, it's a state machine!

Splitting the pure function

One complex function:

```
{output_action, data(t+1)} = fn(event, data(t))
```

Want to split this up into maintainable chunks

Want to group these in a logical way

(which influences the splitting process)

Split 1: Call function only when event occurs

Create a function for each event

```
{output_action, data(t+1)} = fn_event_n(data(t))
```

Split 2: Create "sub-function" for each behavior/state

Most likely a switch-like structure

```
{output_action, data(t+1)} =
  fn_event_behavior(buttons(t), floor(t), direction(t))
```

Unsplitting and defactoring

- Not all event + behavior combinations need their own function
  - Find where the complexity lies

- Not all data needs to be overwritten each time
  - Sometimes only one piece is modified:
    - Button press while moving: Only changes orders, not direction and floor
    - Timeout when door closes: Only changes direction, not orders and floor

- New focus?
  - pure functions that return **one** new data update or output action

Functions

- Buttons / Floor indicator
  - Trivial implementations (directly from events, no "algorithms" needed)
- Direction
  - Several sources
    - Button press when idle
    - Door timer timeout
    - Floor arrival
  - Non-trivial implementation:
    - `fn chooseDirection`
      - Outputs: direction
      - Inputs: {buttons, direction, floor}
  - Perhaps split into a separate function for floor arrival
    - Aka `fn shouldStop`

For message passing languages

- Two kinds of functions:
  - Core: Pure functions
    - Only inputs and outputs
    - Has no persistent state
    - Performs no side effects (like communication)
      - But can return them, consult your language of choice
  - Shell: Communicating functions
    - Only communication (channels or process IDs)
    - Has persistent state, never returns
    - Can be spawned hierarchically, or flat