Complexity, State, and Concurrency

Science

Computer Science

CS 100: Introduction to the Profession

Matthew Bauer & Michael Lee

ILLINOIS INSTITUTE OF TECHNOLOGY
College of Science

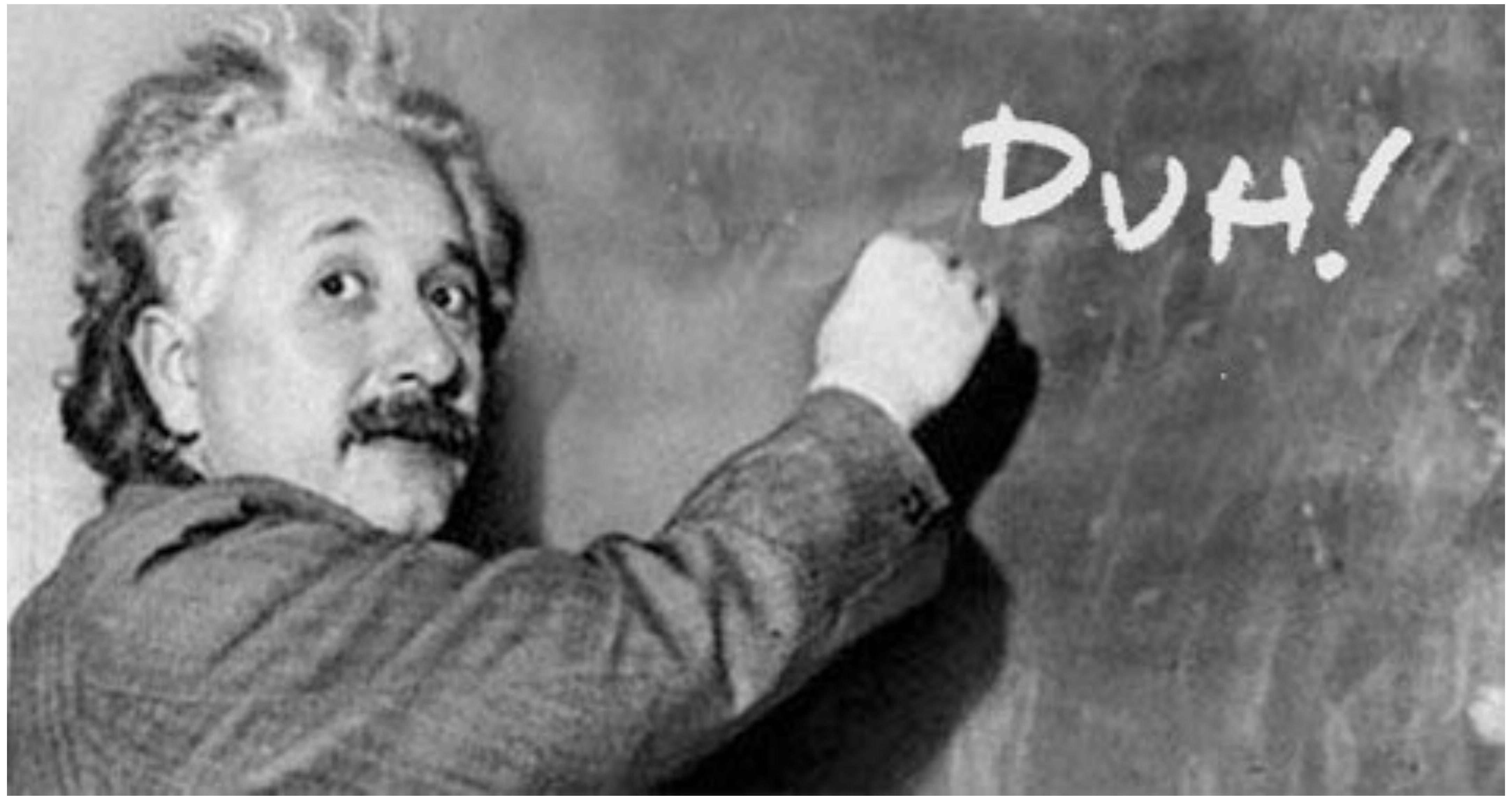
Q: What makes programming hard?

- Language (so many!)
- Code volume (e.g., millions of lines of code)
- Huge libraries (platforms/APIs)
- Algorithmic complexity
- Backwards compatibility / Standards / Compliance
- Performance/Efficiency concerns
- Scaling requirements

§ Complexity

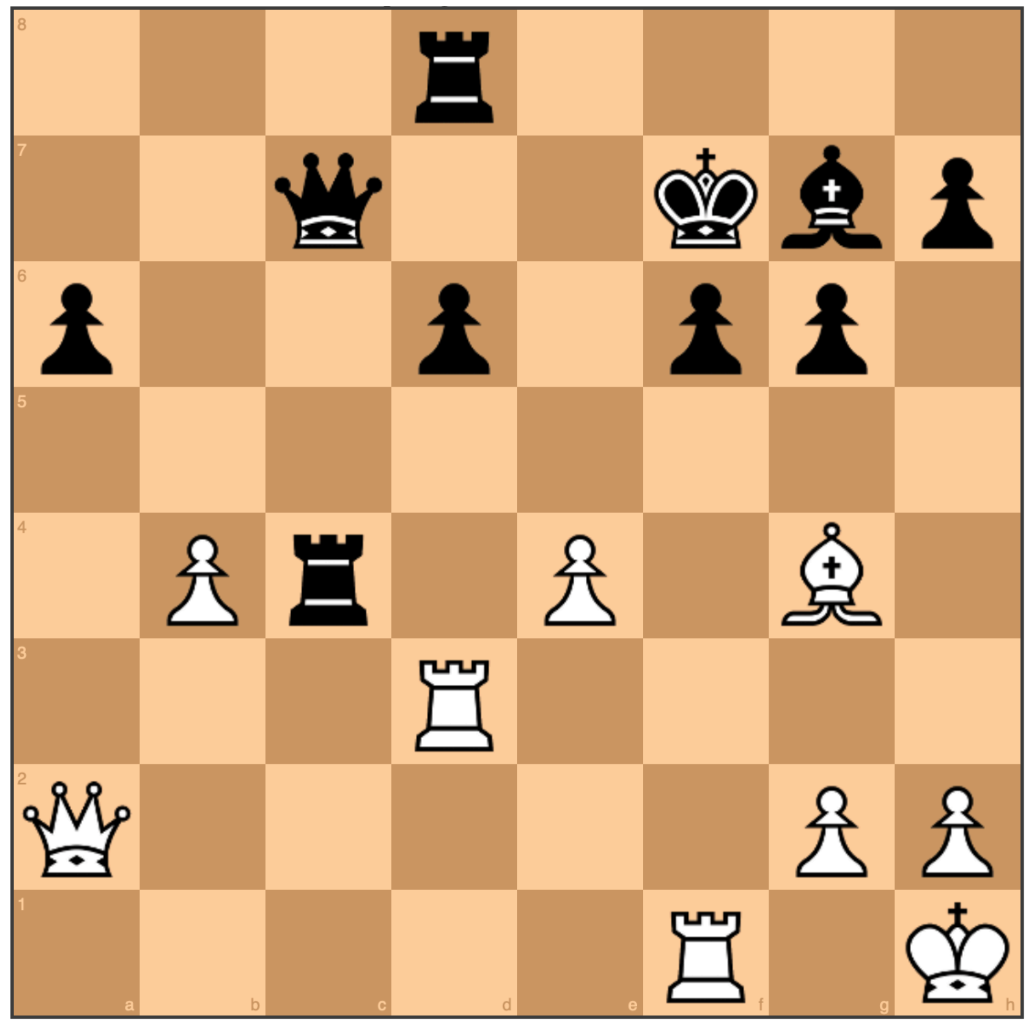
Complexity is the root cause of the vast majority of problems with software today. Unreliability, late delivery, lack of security — often even poor performance in large-scale systems can all be seen as deriving ultimately from **unmanageable complexity**.

Ben Moseley and Peter Marks, *Out of the Tar Pit*


DUH!

But is all complexity the same?

E.g., building an unbeatable chess AI



First steps:

- illustrate the chess board layout
- explain the rules of the game
- describe the desired outcome (e.g., checkmate)

To build a computer AI, we would also typically:

- define *domain-specific types*
- create a *game tree* (for searching ahead / weighing options)
- build supporting *algorithms* and tools (e.g., neural network for deep-learning, feedback mechanisms, UI)

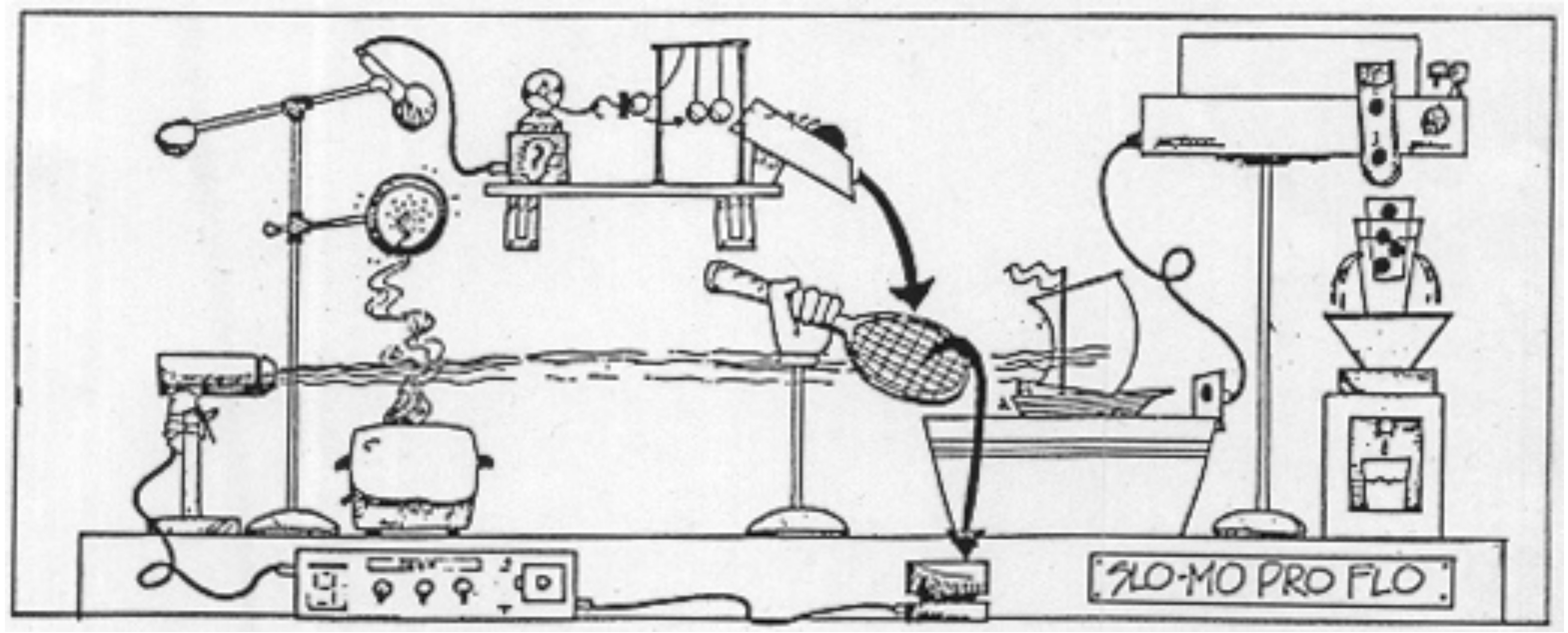
Lots of choices and issues along the way:

- language/framework/other prior work
- performance (how long is AI allowed to “think”?)
- brute force vs. expert system vs. self-learning vs. ?
- how to best accommodate updates and improvements?

Lots of complexity!

Which steps are truly necessary, and which steps are due to limitations of / problems with a particular approach?

In an ideal world, we can simply feed the critical specifications into a machine, and out pops a working solution

Chess rules →  → Perfect AI

Magic Solution Machine ™

In the real world, we should be careful to distinguish between **necessary complexity** and **accidental complexity**

I.e., which problems are *intrinsic* to the problem, and which are simply a product of our *imperfect tools*?

Seek to **minimize accidental complexity**.

Don't make programming harder than it needs to be!

§ Managing Complexity

Q: You're a project manager on a software team. The next deliverable is in a month and you're way behind schedule.

You currently have 5 programmers on the job, and they're already churning out code as fast as they can.

What do you do? (You have plenty of cash.)

~~Add more programmers!~~

ILLINOIS INSTITUTE OF TECHNOLOGY
College of Science

… our estimating techniques *fallaciously confuse effort with progress*, hiding the assumption that men and months are interchangeable.

Adding manpower to a late software project makes it later.

Frederick P. Brooks, *The Mythical Man-Month*

Techniques for managing complexity:

- planning and reasoning
- abstraction and modularization
- testing, testing, and more testing

Planning and reasoning

- white board / pen-and-paper design
- high-level software architecture decisions
- be conservative and pessimistic: things will go wrong!

Abstraction and modularization

- break software into pieces to be designed, implemented, and tested separately
- build to API specifications instead of implementations
- “black box” integration

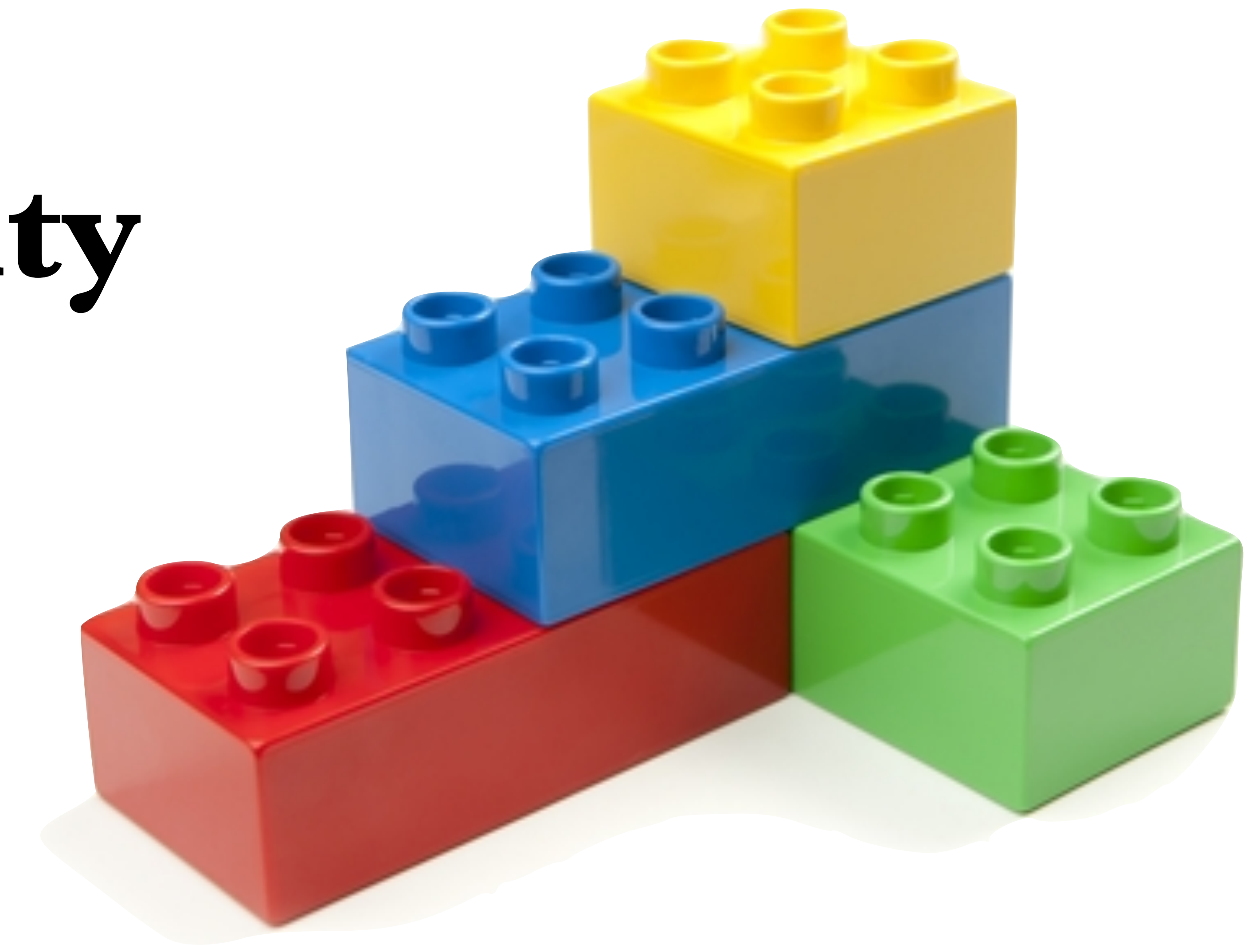
Testing, testing, and more testing

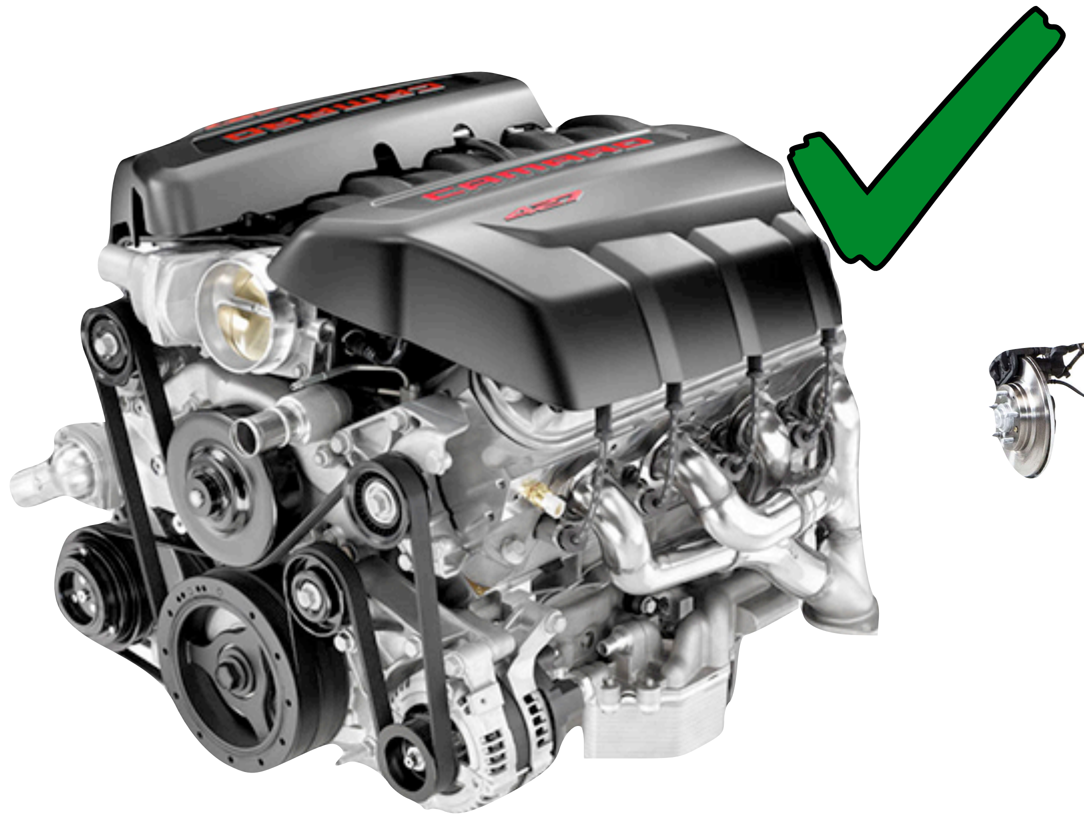
- even before development begins, specify the *expected output for every combination of input* for every module

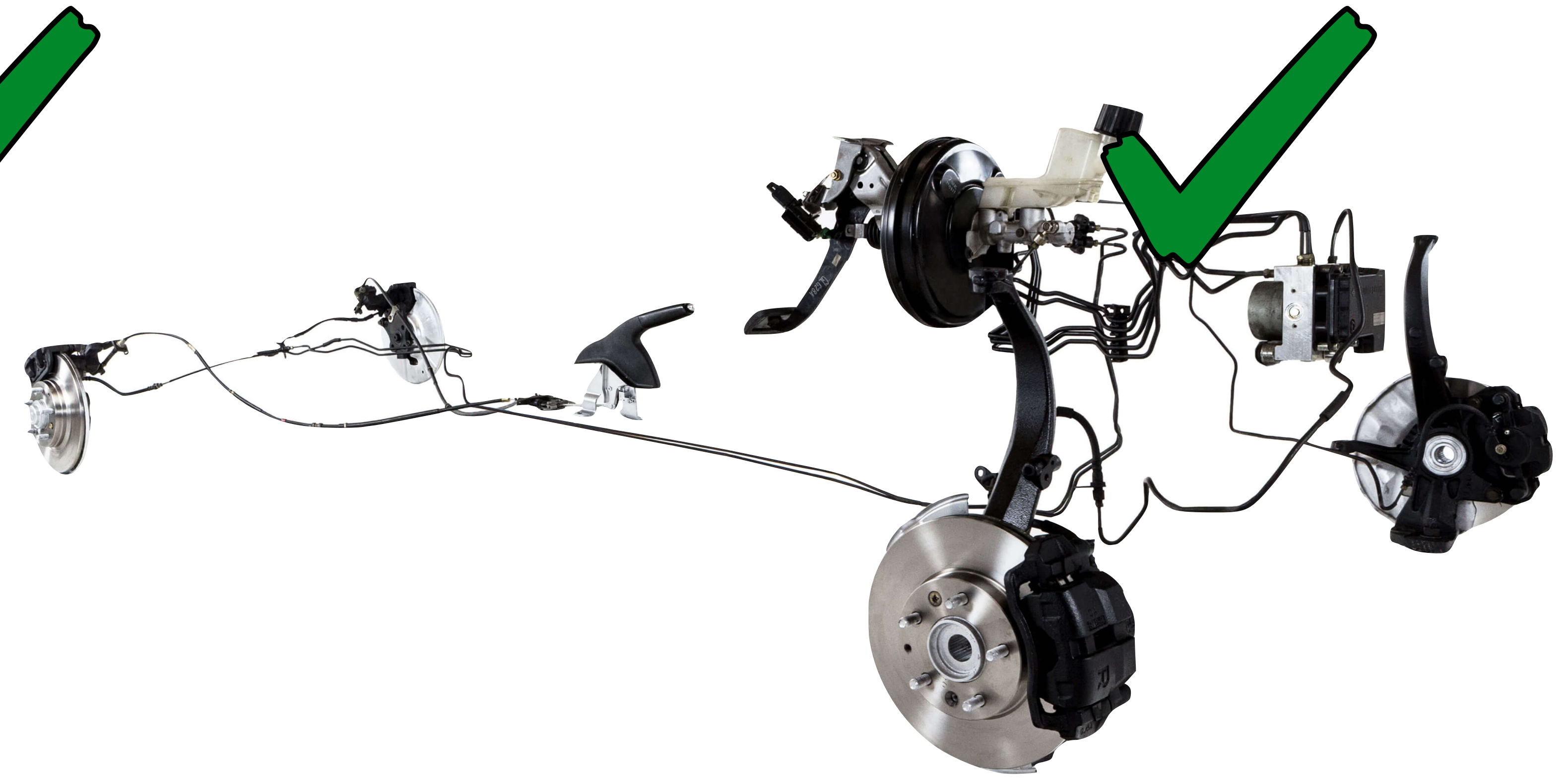
- ensure all tests pass during the development phase! (known as *continuous integration*)

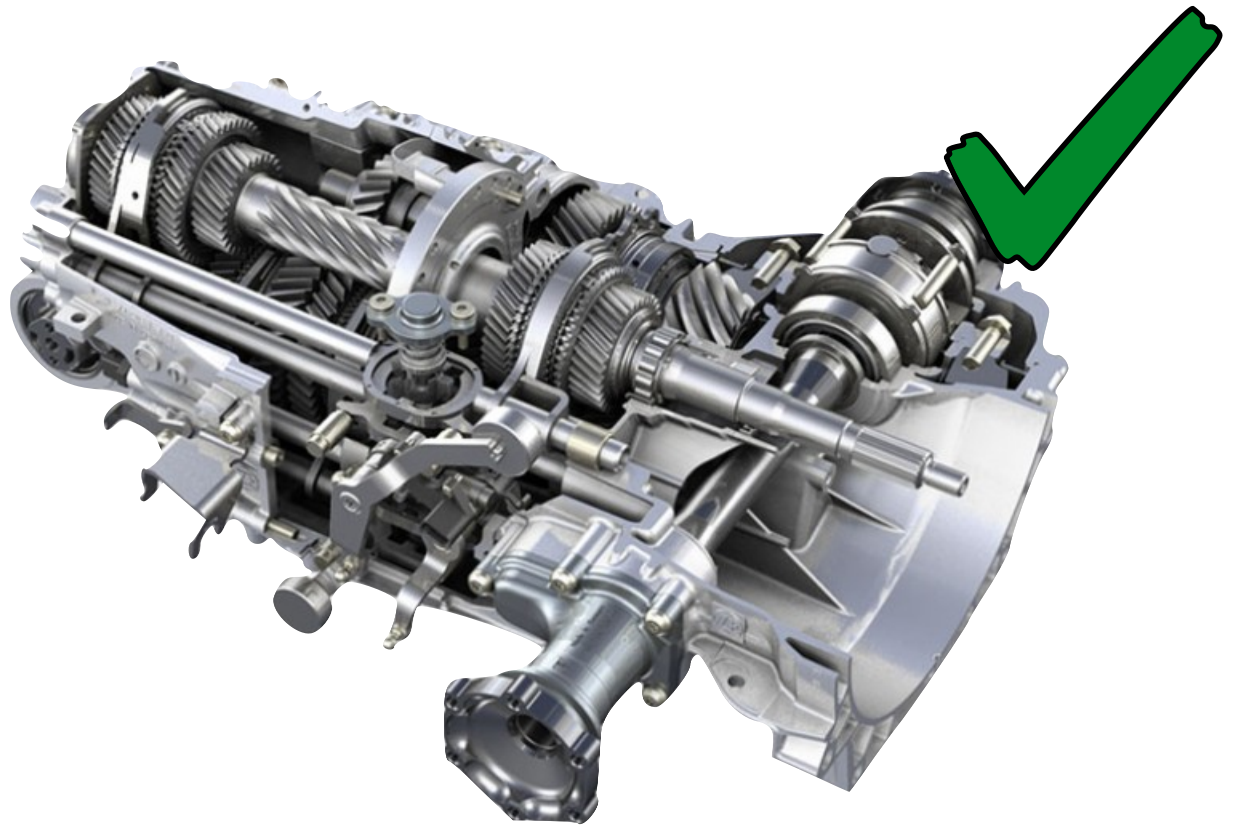
After fully testing modules in isolation we can piece them together to build bigger systems (that work predictably with little further testing)

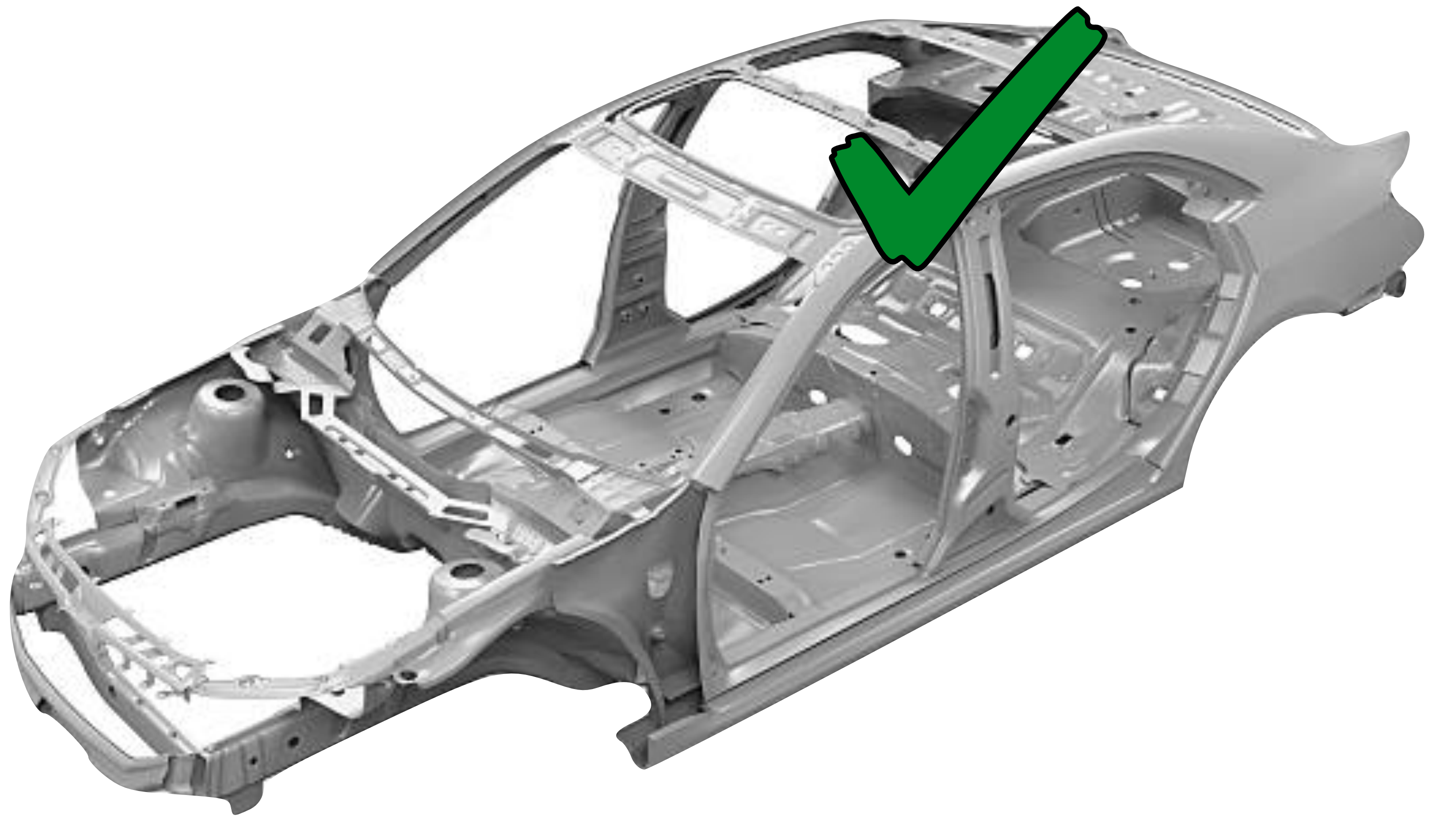
Principle of **composability**











=


Live to Rally
BUGS RALLY EXHIBITION
217
CALIFORNIA
BUG RAL

```python
def discriminant(a,b,c):
    return b*b - 4*a*c

def quadratic_roots(a,b,c):
    d = discriminant(a,b,c)
    if d == 0:
        return -b / (2*a)
    elif d > 0:
        sqrt_d = math.sqrt(d)
        return ((-b+sqrt_d)/(2*a), (-b-sqrt_d)/(2*a))
    else:
        return "No real roots!"
```

$$
\begin{cases}
\Delta = b^2 - 4ac \\
x = \frac{-b \pm \sqrt{\Delta}}{2a}
\end{cases}
$$

```
quadratic_roots(1,4,4)     => -2

quadratic_roots(1,-1,-2)   => (2.0, -1.0)

quadratic_roots(1,3,8)     => "No real roots!"
```

Civilization advances by extending the number of important operations which we can perform *without thinking*.

Alfred North Whitehouse

What are some barriers to composability?

§ State

state | stāt |

noun

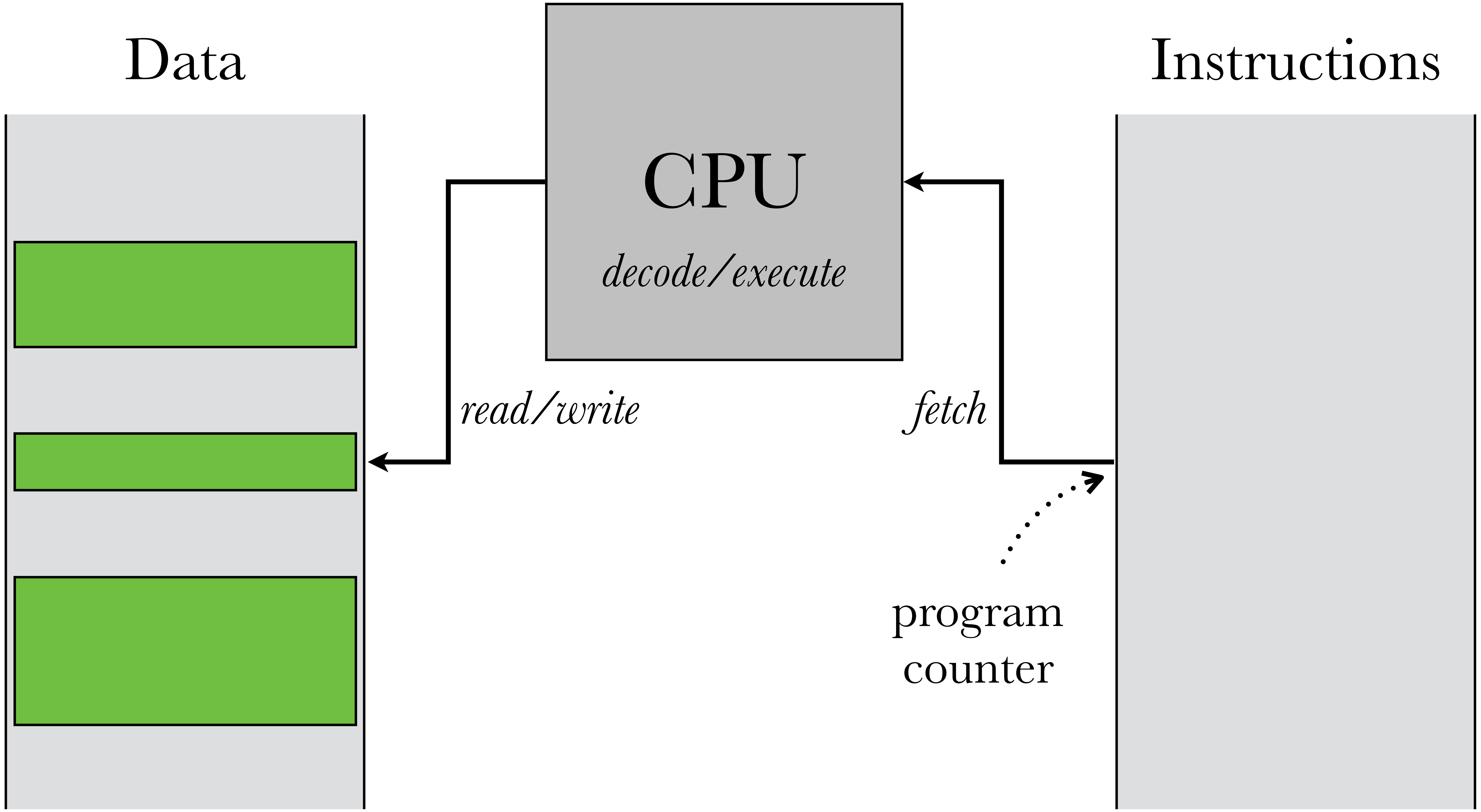
1 the particular condition that someone or something is in at a specific time

The prevailing model of computation is a *stateful* one

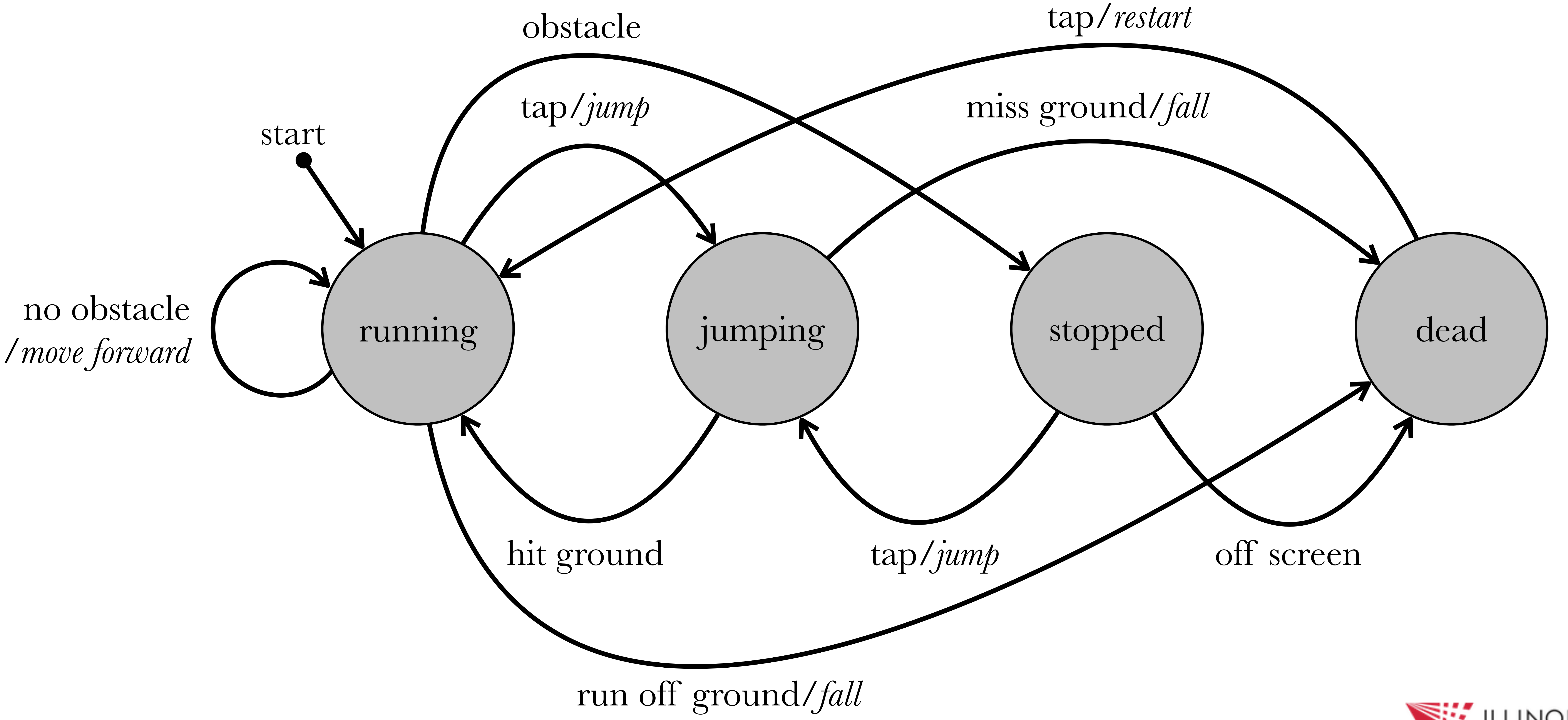
To determine what is going on in our programs, we ask:

- what line of code is being executed?
- what are the values of different variables?
- what is stored in global/local/dynamic data regions?

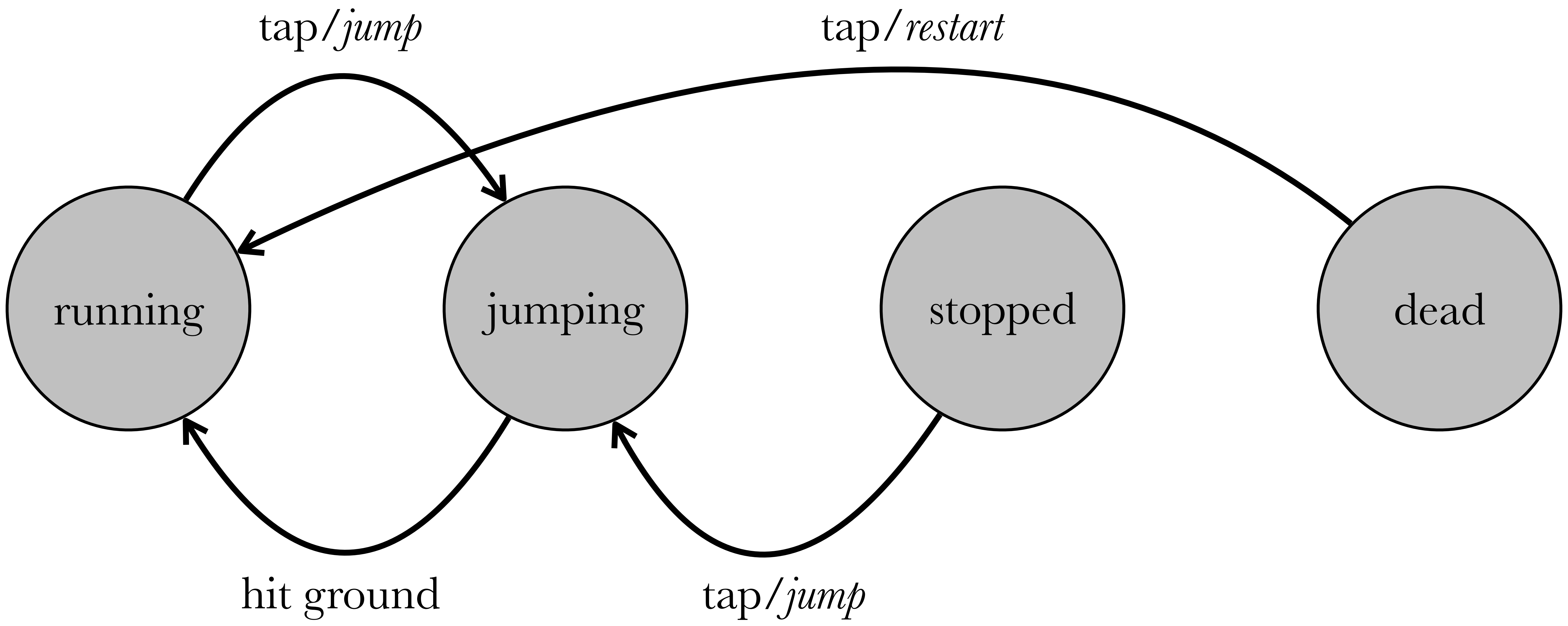
The prevailing model of computation is a *stateful* one



Infinite Runner FSM



```
def process_game_event(event):
    if player_state == 'running':
        if event == 'tap':
            player_state = 'jumping'*
    elif player_state == 'jumping':
        if event == 'hit-ground':
            player_state = 'running'*
    elif player_state == 'stopped':
        if event == 'tap':
            player_state = 'jumping'*
    elif player_state == 'dead':
        if event == 'tap':
            player_state = 'running'*
            restart = True *
    ...
```



* *state mutations*

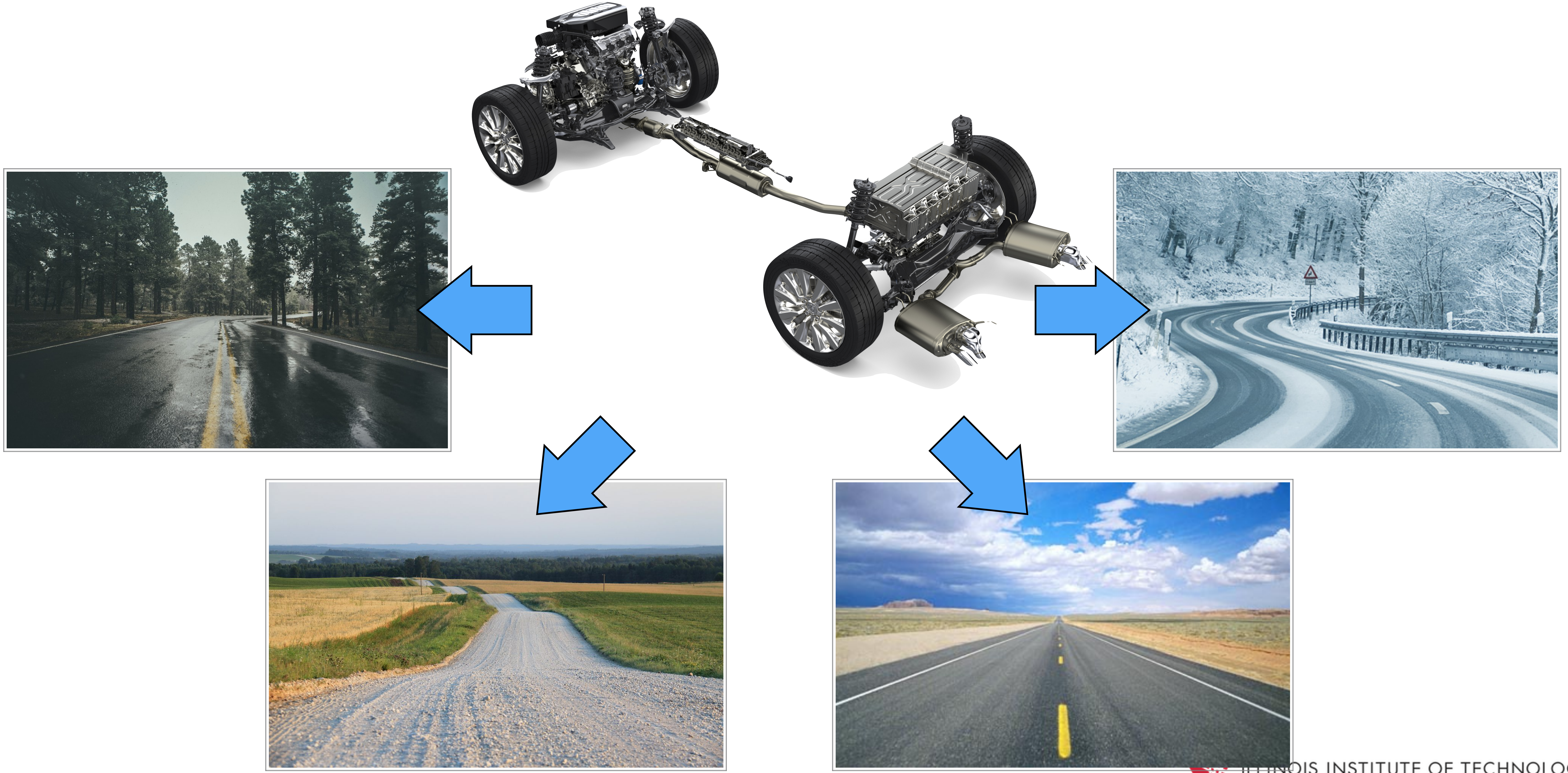
Imperative programming languages reinforce the stateful model by making the standard unit of execution the **statement**.

Statements alter state.

How do we test a stateful program?

(Is the input/output specification method sufficient?)

To properly test a stateful program, we must specify its expected behavior for *all combinations of input and starting state*



What happens when a stateful system gets itself into an unexpected state?

Its behavior is, by definition, unpredictable!

Anyone who has ever telephoned a support desk for a software system and been told to "try it again", or "reload the document", or "restart the program", or "reboot your computer" or "re-install the program" or even "re-install the operating system and then the program" has direct experience of the *problems that state causes* for writing reliable, understandable software.

Ben Moseley and Peter Marks, *Out of the Tar Pit*

```python
num_times = 0

def foo():
    num_times += 1
    if num_times < 100:
        return 10
    else:
        return "I'm too old for this!"
```

we say foo is a *stateful function*,
or that it has *side effects*

```python
foo()          # => 10
```

```python
for _ in range(99): foo()

foo()          # => "I'm too old for this!"
```

```python
# assume we don't know what went before ...

foo() + foo()       # => ?
```

N.B.: Not all systems/computations are stateful!

E.g., mathematical functions are stateless.

$$\int_0^2 x\,dx \;=\; \frac{x^2}{2}\bigg]_0^2 \;=2$$

$$\int_0^2 x\,dx \cdot \left(\int_0^2 x\,dx + 5 \cdot \int_0^2 x\,dx\right) = \,?$$

Regardless of context, they are evaluated the same way.

Useful property known as *referential transparency*.

Stateful functions are harder to test in isolation, but when different stateful functions *share state*, it gets even worse (why?)

And if an otherwise stateless function calls a stateful function, the first one becomes stateful too. I.e., *statefulness is contagious*!

How can we make this even more complicated?

§ Concurrency

The free lunch is over. We have grown used to the idea that our programs will go faster when we buy a next-generation processor, but that time has passed.

While that next-generation chip will have *more CPUs*, each individual CPU will be *no faster* than the previous year's model. *If we want our programs to run faster, we must learn to write parallel programs.*

Simon Peyton Jones, Beautiful Concurrency

The most common form of parallelism is carried out via multiple *threads* of execution that run *concurrently* within a program.

These threads may access *shared data*.

They progress through the program at different, unpredictable rates — i.e., which thread does what first is *non-deterministic*.

Single-threaded

Multi-threaded

```python
def t1():
    for _ in range(times):
        count = count + 1

def t2():
    for _ in range(times):
        count = count + 1

def test(n):
    count   = 0
    times   = n
    thread1 = Thread(target=t1)
    thread2 = Thread(target=t2)
    thread1.start()
    thread2.start()
    thread1.join()
    thread2.join()
    print(shared)
```

```
test(50)      =>     100

test(500)     =>    1000

test(5000)    =>   10000

test(50000)   =>   81443

test(500000)  =>  692171
```

ILLINOIS INSTITUTE OF TECHNOLOGY
College of Science

```
count = count + 1
```

```
%reg  = count
%reg  = %reg + 1
count = %reg
```

count 1

regA -

regB -

```
def t1():
    for _ in range(times):
        regA = count
        regA = regA + 1
        count = regA
```

```
def t2():
    for _ in range(times):
        regB = count
        regB = regB + 1
        count = regB
```

count $\boxed{1}$

regA $\boxed{-}$

regB $\boxed{-}$

```
def t1():
    for _ in range(times):
        regA = count
        regA = regA + 1
        count = regA
```

```
def t2():
    for _ in range(times):
        regB = count
        regB = regB + 1
        count = regB
```

count 1

regA 1

regB -

```
def t1():
    for _ in range(times):
        regA = count
        regA = regA + 1
        count = regA
```

```
def t2():
    for _ in range(times):
        regB = count
        regB = regB + 1
        count = regB
```

count 1

regA 1

regB -

```
def t1():
    for _ in range(times):
        regA = count
        regA = regA + 1
        count = regA
```

```
def t2():
    for _ in range(times):
        regB = count
        regB = regB + 1
        count = regB
```

count $\boxed{1}$

regA $\boxed{2}$

regB $\boxed{-}$

```
def t1():
    for _ in range(times):
        regA = count
        regA = regA + 1
        count = regA
```

```
def t2():
    for _ in range(times):
        regB = count
        regB = regB + 1
        count = regB
```

count $\boxed{1}$

regA $\boxed{2}$

regB $\boxed{-}$

```
def t1():
    for _ in range(times):
        regA = count
        regA = regA + 1
        count = regA
```

```
def t2():
    for _ in range(times):
        regB = count
        regB = regB + 1
        count = regB
```

count $\boxed{1}$

regA $\boxed{2}$

```
def t1():
    for _ in range(times):
        regA = count
        regA = regA + 1
        count = regA
```

regB $\boxed{1}$

```
def t2():
    for _ in range(times):
        regB = count
        regB = regB + 1
        count = regB
```

count $\boxed{1}$

regA $\boxed{2}$

regB $\boxed{1}$

```
def t1():
    for _ in range(times):
        regA = count
        regA = regA + 1
        count = regA
```

```
def t2():
    for _ in range(times):
        regB = count
        regB = regB + 1
        count = regB
```

count $\boxed{2}$

regA $\boxed{2}$

regB $\boxed{1}$

```
def t1():
    for _ in range(times):
        regA = count
        regA = regA + 1
        count = regA
```

```
def t2():
    for _ in range(times):
        regB = count
        regB = regB + 1
        count = regB
```

count $\boxed{2}$

regA $\boxed{2}$

regB $\boxed{1}$

```
def t1():
    for _ in range(times):
        regA = count
        regA = regA + 1
        count = regA
```

```
def t2():
    for _ in range(times):
        regB = count
        regB = regB + 1
        count = regB
```

count $\boxed{2}$

regA $\boxed{2}$

regB $\boxed{1}$

```
def t1():
    for _ in range(times):
        regA = count
        regA = regA + 1
        count = regA
```

```
def t2():
    for _ in range(times):
        regB = count
        regB = regB + 1
        count = regB
```

count $\boxed{2}$

regA $\boxed{3}$

regB $\boxed{1}$

```
def t1():
    for _ in range(times):
        regA = count
        regA = regA + 1
        count = regA
```

```
def t2():
    for _ in range(times):
        regB = count
        regB = regB + 1
        count = regB
```

count 2

regA 3

regB 1

```
def t1():
    for _ in range(times):
        regA = count
        regA = regA + 1
        count = regA
```

```
def t2():
    for _ in range(times):
        regB = count
        regB = regB + 1
        count = regB
```

count $\boxed{3}$

regA $\boxed{3}$

regB $\boxed{1}$

```
def t1():
    for _ in range(times):
        regA = count
        regA = regA + 1
        count = regA
```

```
def t2():
    for _ in range(times):
        regB = count
        regB = regB + 1
        count = regB
```

count $\boxed{3}$

regA $\boxed{3}$

regB $\boxed{1}$

```
def t1():
    for _ in range(times):
        regA = count
        regA = regA + 1
        count = regA
```

```
def t2():
    for _ in range(times):
        regB = count
        regB = regB + 1
        count = regB
```

ILLINOIS INSTITUTE OF TECHNOLOGY
College of Science

count $\boxed{3}$

regA $\boxed{3}$

regB $\boxed{2}$

```
def t1():
    for _ in range(times):
        regA = count
        regA = regA + 1
        count = regA
```

```
def t2():
    for _ in range(times):
        regB = count
        regB = regB + 1
        count = regB
```

count $\boxed{3}$

regA $\boxed{3}$

regB $\boxed{2}$

```
def t1():
    for _ in range(times):
        regA = count
        regA = regA + 1
        count = regA
```

```
def t2():
    for _ in range(times):
        regB = count
        regB = regB + 1
        count = regB
```

count $\boxed{2}$

regA $\boxed{3}$

regB $\boxed{2}$

```
def t1():
    for _ in range(times):
        regA = count
        regA = regA + 1
        count = regA
```

```
def t2():
    for _ in range(times):
        regB = count
        regB = regB + 1
        count = regB
```

"Race conditions" in concurrent programs may lead to incorrect — and worse, unpredictable — results.

Concurrency also affects testing, for in this case, we can no longer even be assured of result consistency when repeating tests on a system — even if we somehow ensure a consistent starting state.

Running a test in the presence of concurrency with a known initial state and set of inputs *tells you nothing at all* about what will happen the next time you run that very same test with the very same inputs and the very same starting state. . . and things can't really get any worse than that.

Ben Moseley and Peter Marks, *Out of the Tar Pit*

Statefulness and concurrency can make testing near impossible, and destroy composability!

So how do we deal with this?

Approaches:

1. Outlaw modifications to shared data (i.e., no stateful code).
2. Limit concurrent execution by forcing critical shared data to be accessed in isolation, using software “locks”.
3. Delegate management of concurrency to someone else — mark which code blocks need special attention.

All these approaches have their pros/cons — concurrent programming is still very much an open research problem.

Many CS classes present different approaches to concurrency, along with problems they are intended to help solve.

References:

- Frederick P. Brooks, “No Silver Bullet.”
- Frederick P. Brooks, “The Mythical Man-Month.”
- Ben Moseley and Peter Marks, “Out of the Tar Pit.”
- Simon Peyton Jones, “Beautiful Concurrency.”
- John Backus, “Can Programming Be Liberated from the von Neumann Style?”