State, Concurrency & Complexity

Does programming really need to be so hard?

Objectives

- explain how *complexity* impacts software
- distinguish between *necessary* and *accidental* complexity
- identify ways we *deal with complexity*
- understand how *state* affects programs

Objectives (2)

- understand the importance of *concurrency*
- recognize ties between *state, concurrency, and complexity*
- understand how *different programming paradigms* deal with state & concurrency

Topics

1. Complexity (overview)
2. Managing complexity
3. State
4. Concurrency
5. Object oriented programming
6. Functional programming

§ Complexity

What makes programming hard?

- language & API
- code volume
- algorithmic complexity
- performance requirements
- backwards/forwards compatibility
- **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*

but what makes this interesting is that we can *partition* complexity into *different categories*

essential complexity arises from the actual problem we're trying to solve.

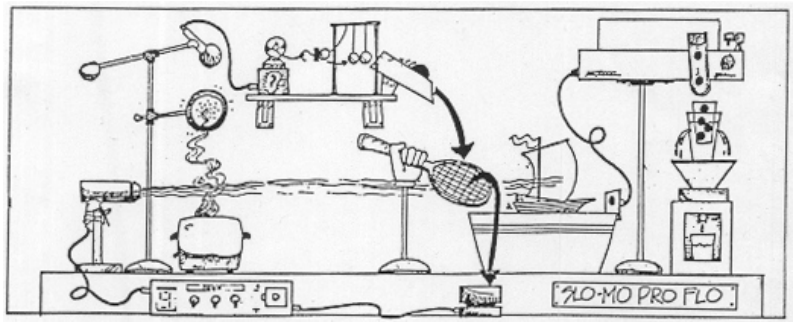
e.g., determine the next best chess move, given all prior moves and the rules of the game

in an ideal world, a sufficiently detailed
problem description can be used to
automatically generate a working solution (!)

layout of a chess board, rules of the game,
meaning of “good” outcomes (e.g., checkmate)

↓

Magic solution machine



↓

Perfect Chess Artificial Intelligence

accidental complexity arises because our programming tools aren't perfect:

- expressing logic via (imperfect) language
- user defined data types (e.g., chessboard)
- managing *derived data* (e.g., game tree)
- performance optimization

we want to *minimize* ***accidental*** *complexity*

i.e., don't make it harder than it needs to be!

§ Managing complexity

Ask any project manager:

How do we deal with complexity in a large software project?

... add more programmers!

*... 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*

Ask any project manager:

How do we deal with complexity in a large software project?

... add more programmers!

1. Do not make overly-optimistic estimates
2. Plan appropriately (reasoning)
3. Divide and conquer (modularization)
4. Test, test, test! (testing)

Reasoning:

- pencil & paper planning
- algorithmic proofs
- “hammock-driven development”

… but the brain’s working store is *very* limited!

Modularization:

- break problem into manageable pieces
- work on each piece separately
- define clear application programming interfaces (APIs) to connect them

Testing:

- ideally, start *before* implementation
  - given input, specify output/behavior
- *unit tests* for discrete program modules
- perform *continuous integration*

Testing *granularity*?

- functions
- packages
- programs
- systems

Ideally, modules being tested are *composable*.

e.g., if A is tested, and B is tested, then we know that A + B works predictably.

Modularization & Composability allow us to
use prior work & ignore their implementation

i.e., modularization & composability give us automatic ***abstraction***

(and programmers ♥ abstraction!)

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

- Alfred North Whitehouse

but … what defeats composability?


State

state |stāt|

noun

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

vs.

given a particular set of inputs, a
stateless function always returns the same result

a.k.a. *pure* function

mathematical functions are stateless

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

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

- we can always replace identical function calls with the same value
- known as *referential transparency*

referential transparency supports composability

— if we test a pure function in isolation, it will work *exactly as predicted anywhere we use it*.

```
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!”
```

however, functions in programming languages may reference and update *mutable state*, which can affect the result of its computations

```
num_times = 0

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

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

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

foo()               # => “I’m too old for this!”
```

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

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

```
num_times = 0

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

we say that the function `foo` has *side effects*
(or is a *stateful* function)

functions with side effects are harder to test — their results *are dependent on the* ***current state***

even worse: this state may be modified by other stateful functions, too!

One of the issues (that affects both testing and reasoning) is the exponential rate at which the number of possible states grows — for every single **bit** *of state that we add we* **double** *the total number of possible states.*

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

note: if an otherwise stateless function calls a stateful function, it is no longer referentially transparent

i.e., statefulness is *contagious*!

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*

reasoning about stateful programs is hard ... we need to consider *all possible paths* through a program and how changes to state affect results

but … what if more than one path were being taken through our programs *simultaneously*?

§ 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*

concurrency within a program = multiple *threads* executing functions (potentially the same ones) simultaneously

```
results = {}

def cache_result(a,b,c):
    results[(a,b,c)] = quadratic_roots(a,b,c)

# allow for 10 threads to run concurrently
pool = ThreadPool(10)

for a in range(1,5):
    for b in range(1,5):
        for c in range(1,5):
            pool.add_task(cache_result, a, b, c)
```

```
results[(1,4,4)]  # => -2

results[(1,3,2)]  # => (-1.0, -2.0)

results[(1,1,1)]  # => No real roots!
```

```
def foo(n):
    global shared
    for _ in range(n):
        shared = shared + 1

def bar(n):
    global shared
    for _ in range(n):
        shared = shared + 1

def test(n):
    global shared
    shared = 0
    pool = ThreadPool(2)
    pool.add_task(foo, n)
    pool.add_task(bar, n)
    pool.wait_completion()
    print(shared)
```

```
test(50)      =>      100

test(500)     =>     1000

test(5000)    =>    10000

test(50000)   =>    81443

test(500000)  =>   692171
```

results are *non-deterministic*

— caused by “race conditions”

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*

side effects + concurrency *kill composability*,
and make reasoning & testing nigh impossible!

how do our programming language paradigms deal with state & concurrency?

§ Object Oriented Programming

Classical OOP essentials:

- *nouns* in a problem are modeled using user-defined *classes*
- instances of these classes represent *objects*
- objects *encapsulate* data with behavior-defining *methods* which may *modify* said data

Classical OOP implies *stateful programming*

- the *identity* of an object is intertwined with a bundle of values (its *state*)
- an object is updated by changing part or all of its state (a.k.a. in-place "mutation")
  - while using an object, it is possible that someone else might change its state!

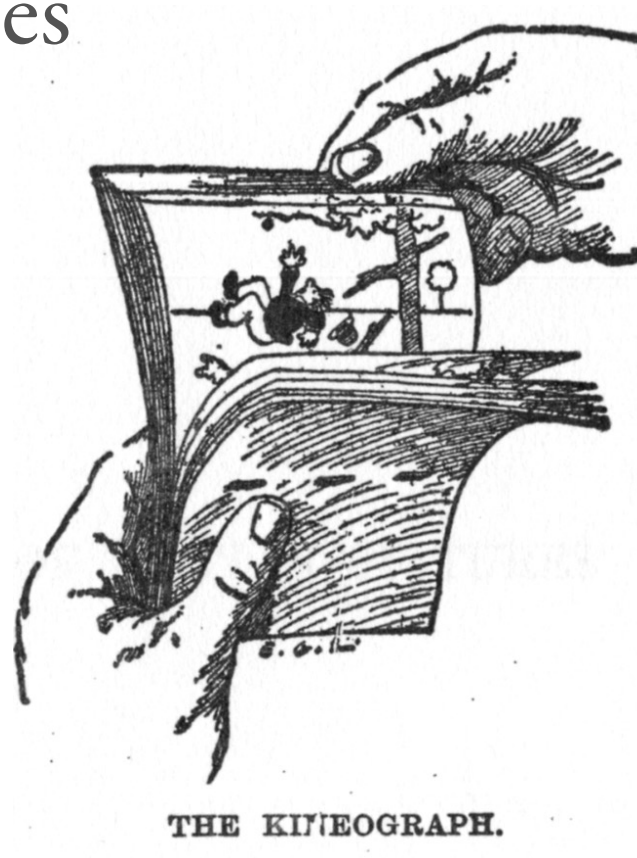
OOP typically requires special tools to deal with *concurrent, stateful changes*

- e.g., use *mutually exclusive locks* to prevent threads from concurrently accessing objects
  - i.e., "wait while I perform my (potentially costly and unpredictable) computation"
  - inefficient & *really hard* to test

OOP is ***not*** particularly suitable for writing concurrent programs!

An alternative view: objects advance through separate, *instantaneous states*

- the current "state" of an object is an immutable collection of values
- we use names (identities) to identify the *current* state
  - but a given state may never change!



§ Functional Programming

Functional programming essentials:

- eschew stateful computation
- no *statements*, only *expressions*
- prefer (or require) all functions to be *pure*
- functions are “first class” — i.e., they can be created, stored, and passed like other data

```
discriminant (a,b,c) = b**2 - 4*a*c

quadratic_roots (a,b,c) = [(-b+d)/2*a, (-b-d)/2*a]
  where d = sqrt (discriminant (a,b,c))
```

```
quadratic_roots (1,4,4)  =>  [-2.0,-2.0]
```

```
eqns = [(a,b,c) | a <- [1..4], b <- [1..4], c <- [1..4]]

map discriminant eqns  => [-3.0,-7.0,-11.0,...,-32.0,-48.0]

filter ((>= 0) . discriminant) eqns
   => [(1,2,1),(1,3,1),(1,3,2),(1,4,1),(1,4,2),(1,4,3),
       (1,4,4),(2,3,1),(2,4,1),(2,4,2),(3,4,1),(4,4,1)]

map quadratic_roots $ filter ((>= 0) . discriminant) eqns
   => [[-1.0,-1.0],[-0.38,-2.62],...,[-3.0,-9.0],[-0.5,-0.5]]
```

functional purity → referential transparency

- great for composability
- huge boon to reasoning and testing
- enables automatic performance optimizations (e.g., *memoization*)

but … not everything's a pure function!

- "search the web for 'memoization'"
- "remotely start my car"
- "fire my shrink-ray!"

key is to *minimize & isolate* state manipulation

- separate pure & impure aspects
- distinguish *identity* and *state*
- enable comprehensive testing and *high level reasoning*

many topics to explore!

- message-passing frameworks
- software transactional memory
- monads & monadic composition
- languages: Erlang, Clojure, Haskell

Conventional programming languages are growing ever more enormous, but not stronger. Inherent defects at the most basic level cause them to be both fat and weak: their primitive word-at-a-time style of programming ..., their close coupling of semantics to state transitions, their division of programming into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for building new programs from existing ones, and their lack of useful mathematical properties for reasoning about programs.

- John Backus, *Can Programming Be Liberated from the von Neumann Style?* (1978)

There is no single development, in either technology or management technique, which by itself promises even one order-of-magnitude improvement within a decade in productivity, in reliability, in simplicity.

- Frederick P. Brooks, *No Silver Bullet — Essence and Accident in Software Engineering*

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."
- Rich Hickey, "Are We There Yet?"
- John Backus, "Can Programming Be Liberated from the von Neumann Style?"