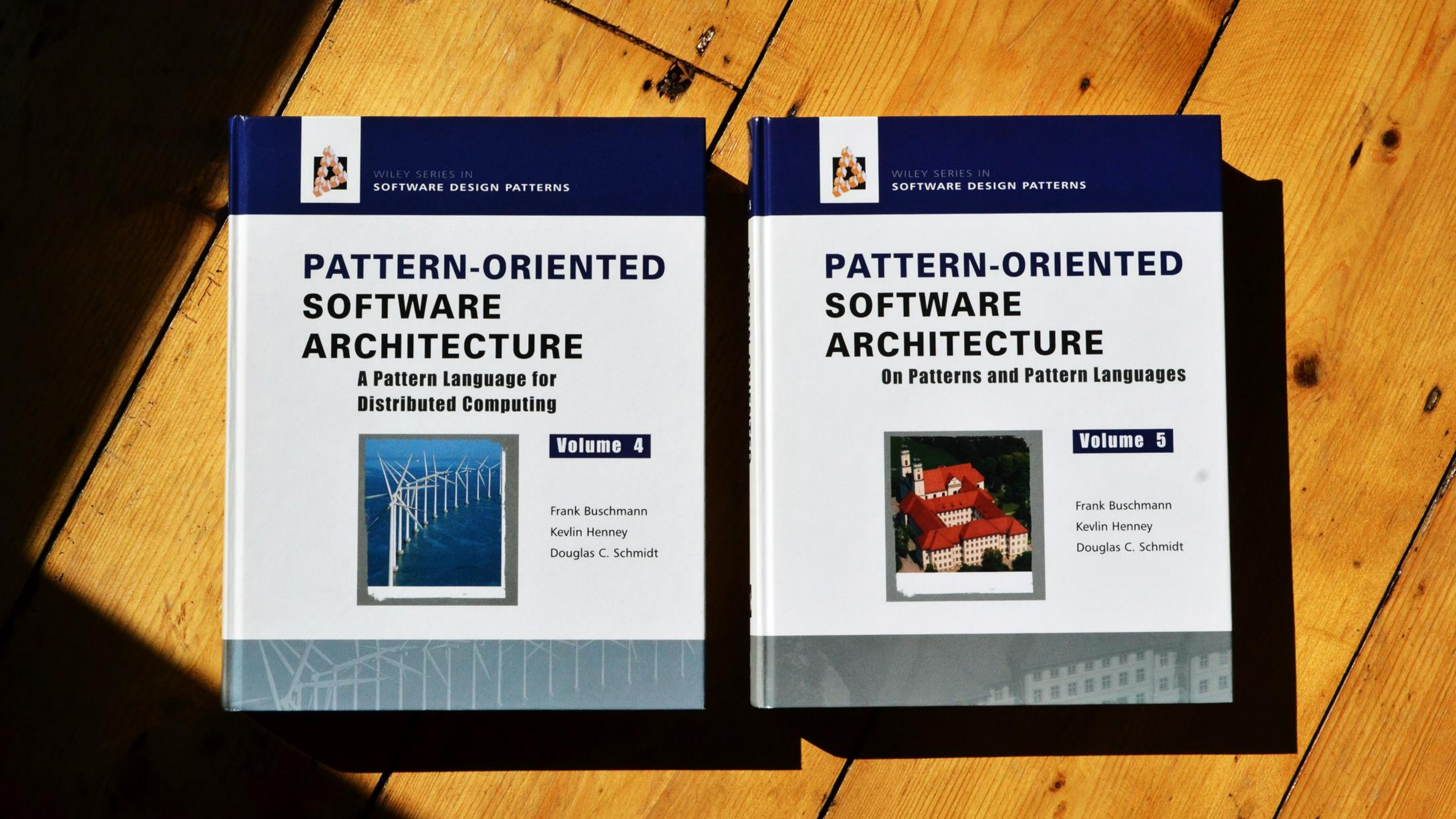
Thinking Outside the Synchronisation Quadrant

@KevlinHenney

97
Collective Wisdom
from the Experts
97 Things Every
Programmer
Should Know
O'REILLY®
Edited by Kevlin Henney
知るべき
97 Things Every Prog
O'REILLY®
オライリー・ジャパン
Kevlin Henney 编
李军 译　吕骏 审校
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn
件事
ILLY®
97
E N T G V P E G R R M S A H K N U O L W O
97件事
ДОВ
ММИСТ



WILEY SERIES IN
SOFTWARE DESIGN PATTERNS

PATTERN-ORIENTED
SOFTWARE
ARCHITECTURE

A Pattern Language for
Distributed Computing

Volume 4

Frank Buschmann
Kevlin Henney
Douglas C. Schmidt

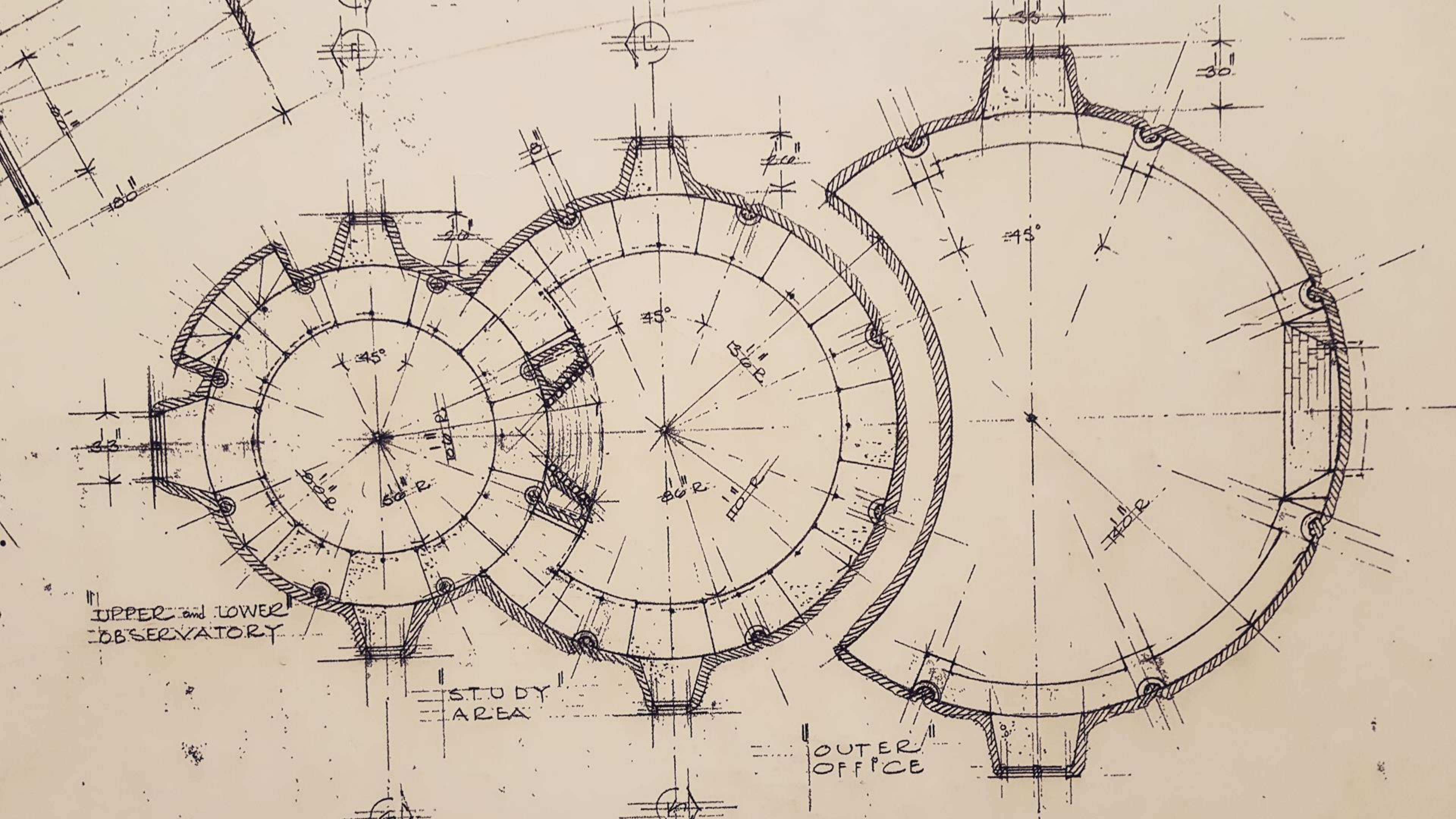
WILEY SERIES IN
SOFTWARE DESIGN PATTERNS

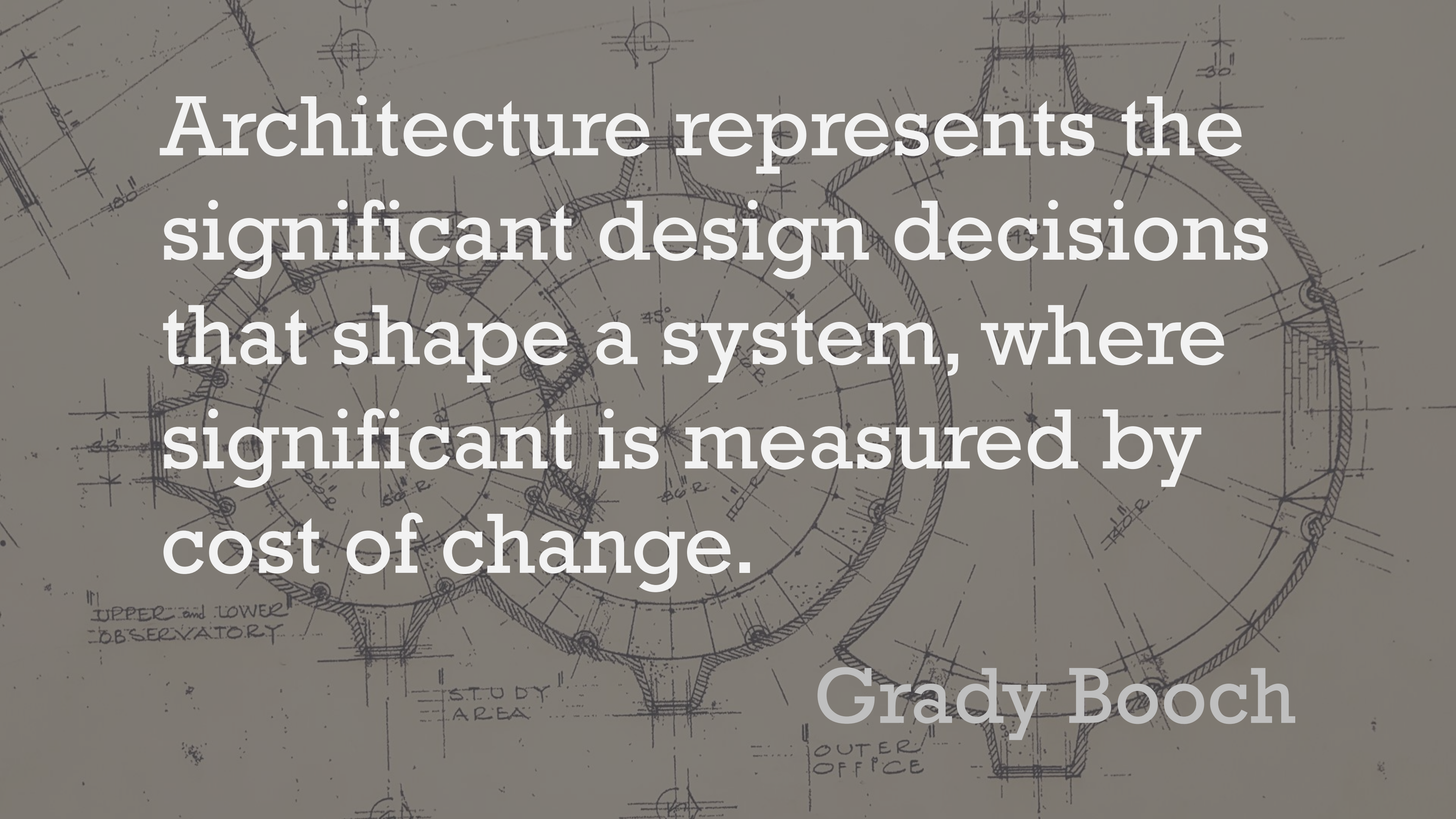
PATTERN-ORIENTED
SOFTWARE
ARCHITECTURE

On Patterns and Pattern Languages

Volume 5

Frank Buschmann
Kevlin Henney
Douglas C. Schmidt


UPPER and LOWER OBSERVATORY
STUDY AREA
OUTER OFFICE
45°
45°
45°
33"
33"
30"
20"
8"
180"
86"R
140"R



> Architecture represents the significant design decisions that shape a system, where significant is measured by cost of change.

Grady Booch

Concurrency

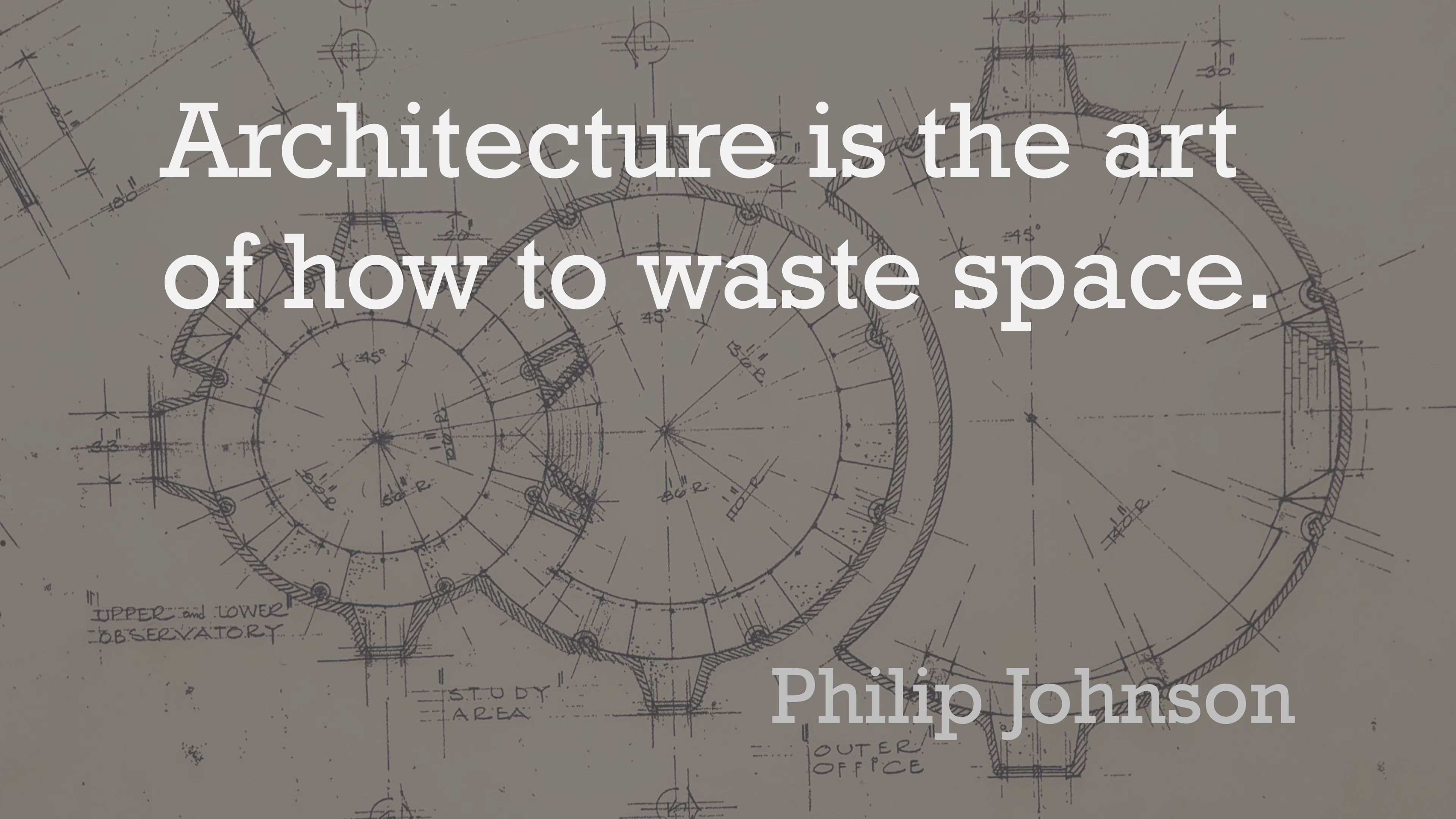
Concurrency

Threads

Concurrency

Threads

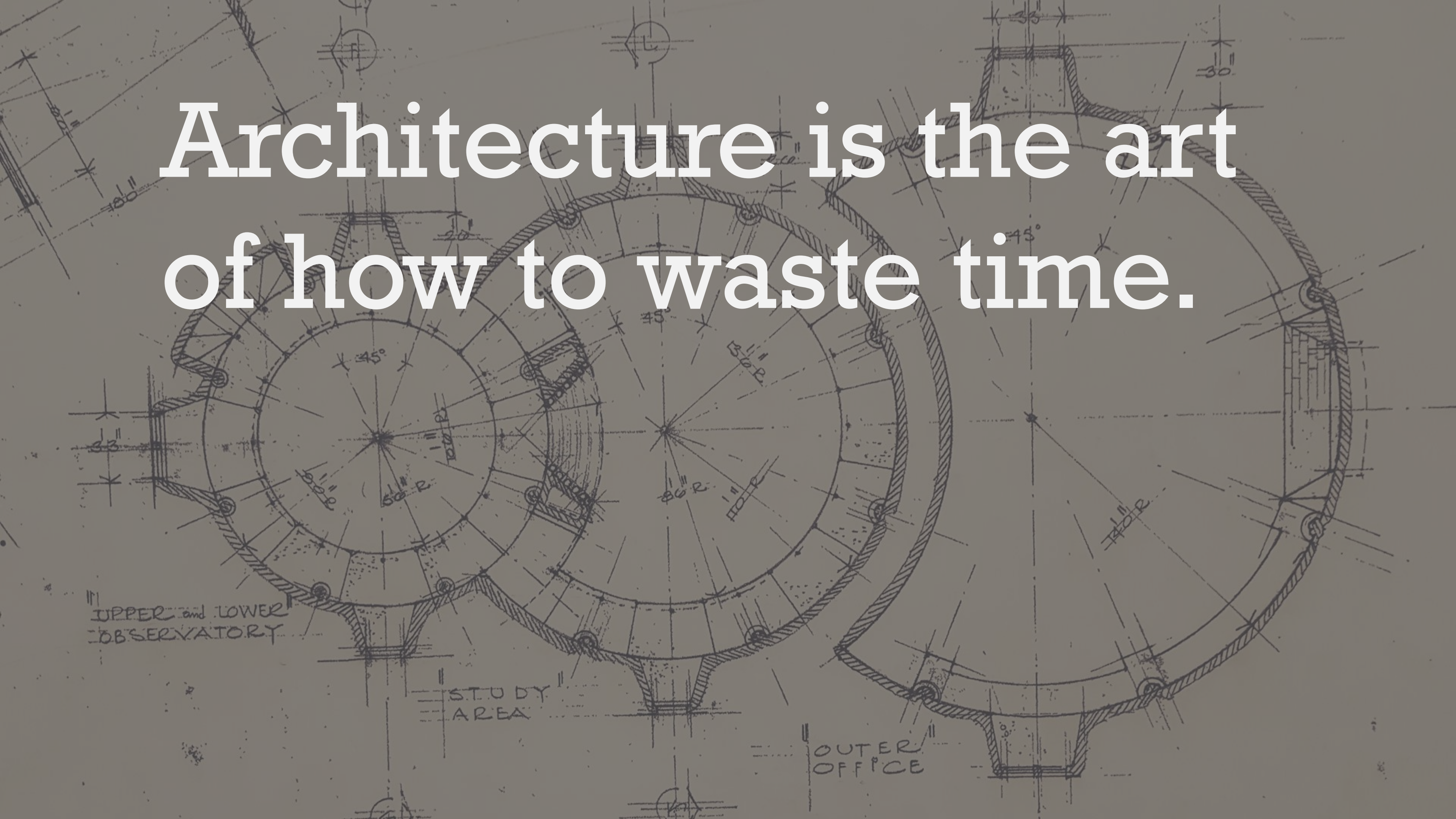
Locks

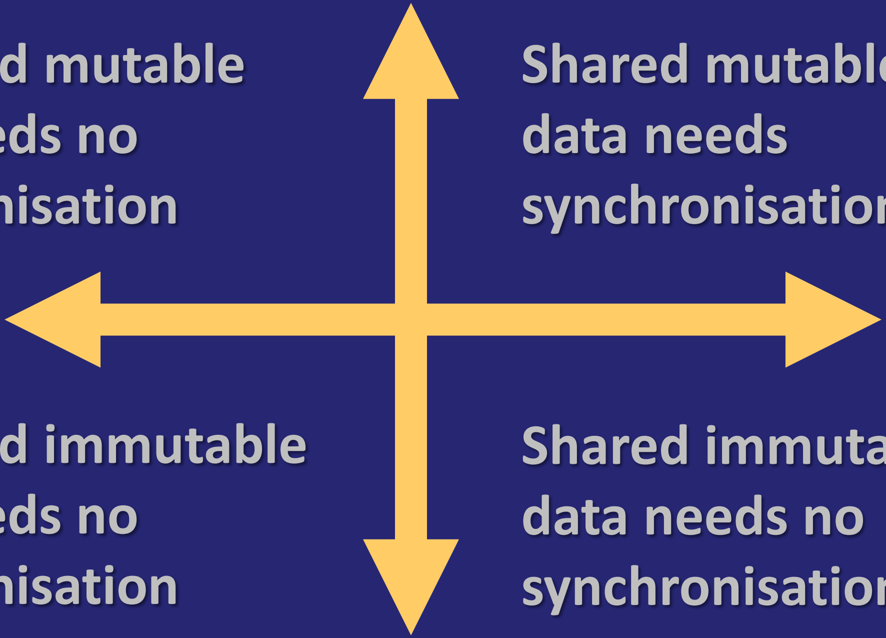


Architecture is the art of how to waste space.

Philip Johnson

Architecture is the art of how to waste time.





Mutable

Unshared mutable data needs no synchronisation

Shared mutable data needs synchronisation

Unshared

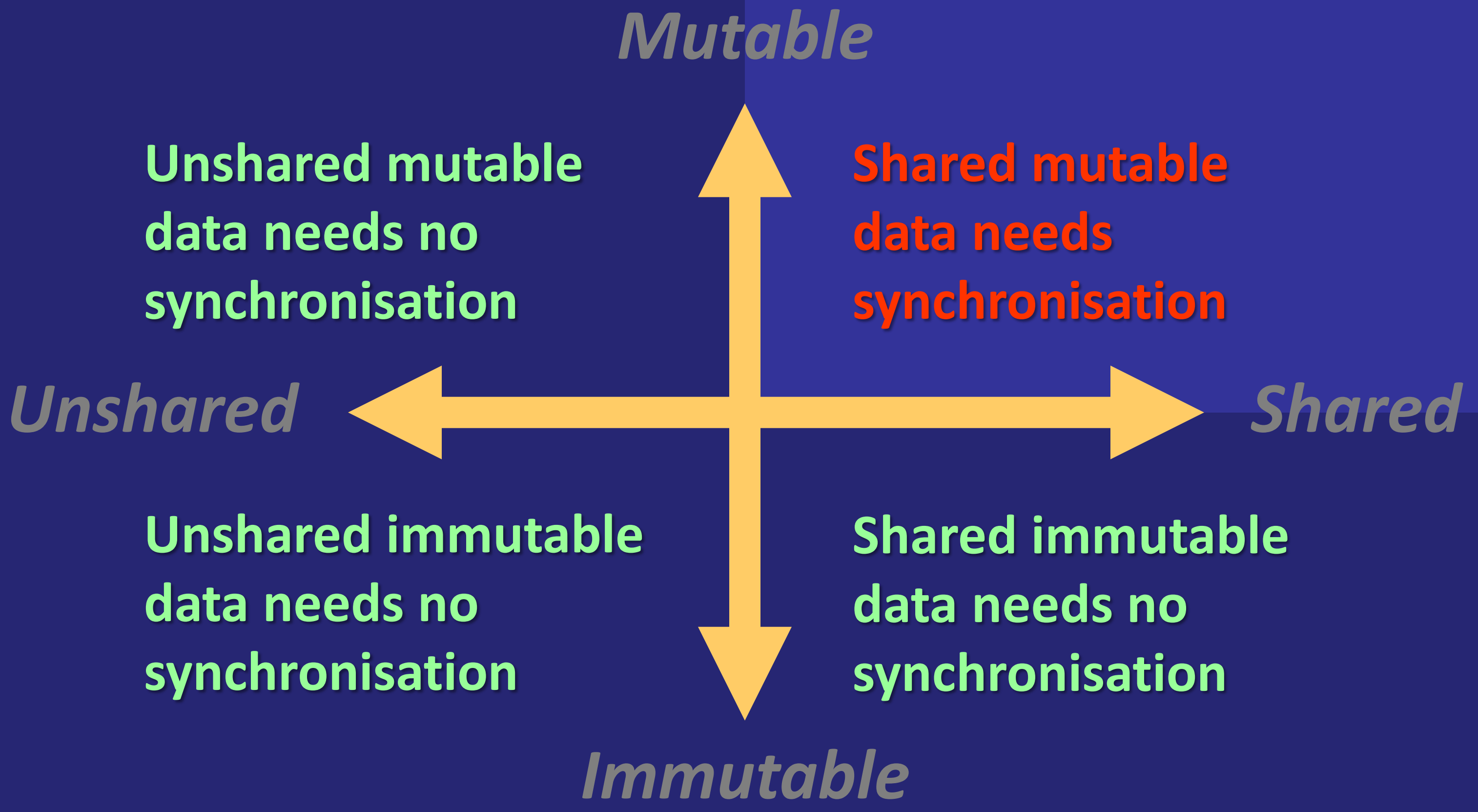
Shared

Unshared immutable data needs no synchronisation

Shared immutable data needs no synchronisation

Immutable

The Synchronisation Quadrant



Mutable

Unshared mutable
data needs no
synchronisation

Shared mutable
data needs
synchronisation

Unshared

Shared

Unshared immutable
data needs no
synchronisation

Shared immutable
data needs no
synchronisation

Immutable

Threads

Article development led by ACM Queue
queue.acm.org

DOI:10.1145/2844112

We need it, we can afford it, and the time is now.

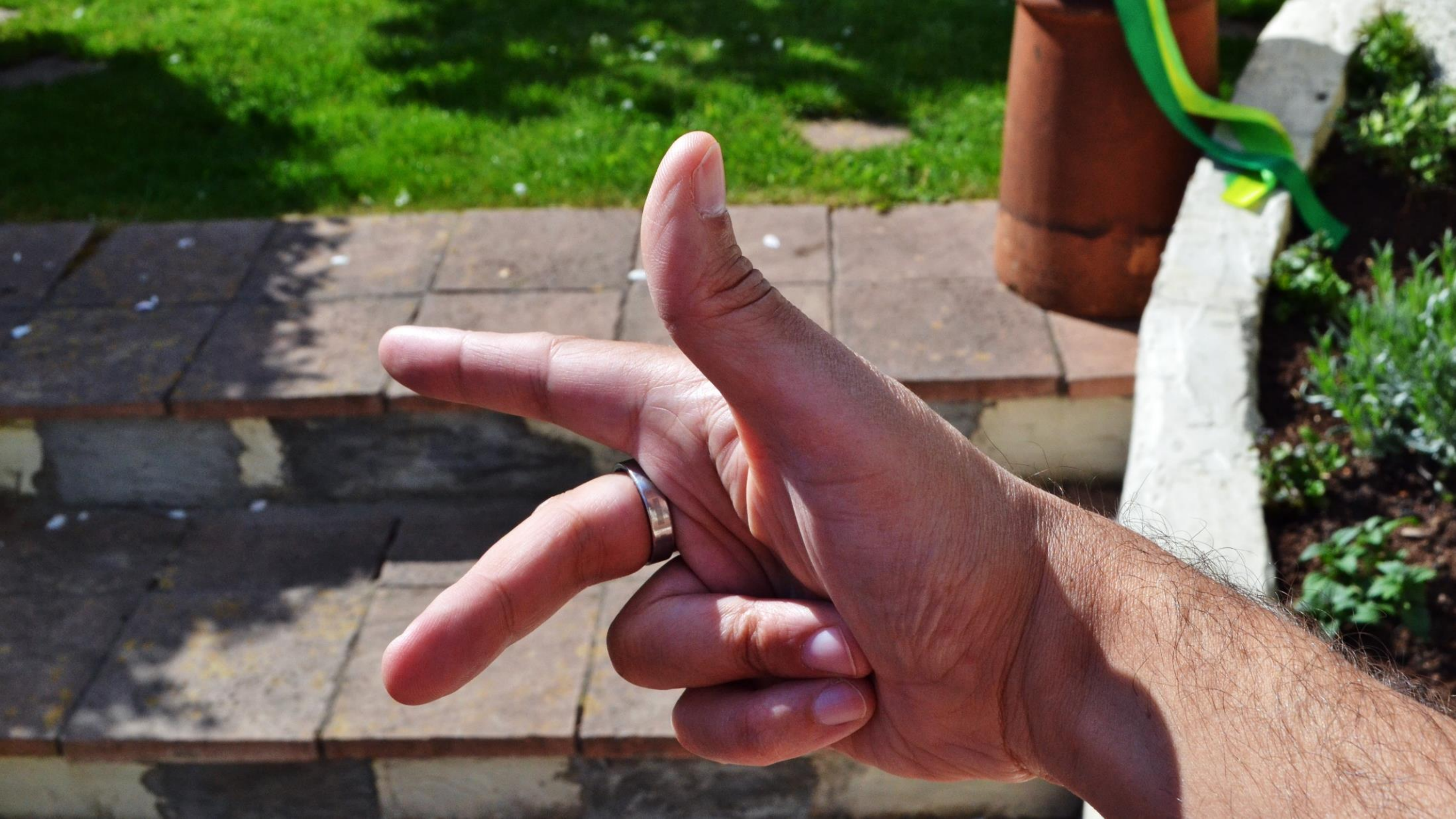
BY PAT HELLAND

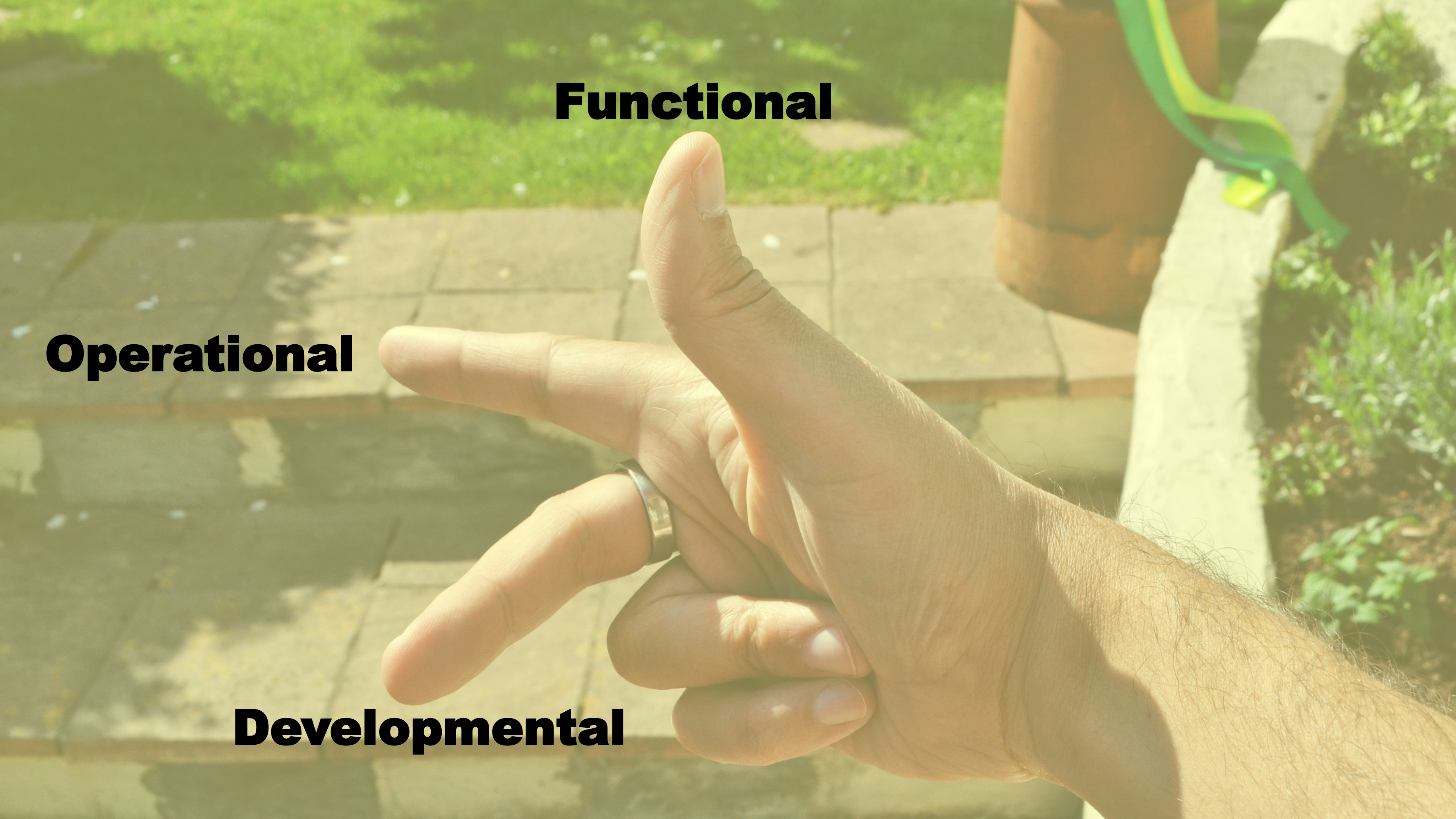
Immutability Changes Everything

latches has become harder because latch latency loses lots of instruction opportunities. Keeping immutable copies of lots of data is now affordable, and one payoff is reduced coordination challenges.

Storage is increasing as the cost per terabyte of disk keeps dropping. This means a lot of data can be kept for a long time. Distribution is increasing as more and more data and work are spread across a great distance. Data within a data center seems "far away." Data within a many-core chip may seem "far away." Ambiguity is increasing when trying to coordinate with systems that are far away—more stuff has happened since you last heard the news. Can you take action with incomplete knowledge? Can you wait for enough knowledge?

Turtles all the way down

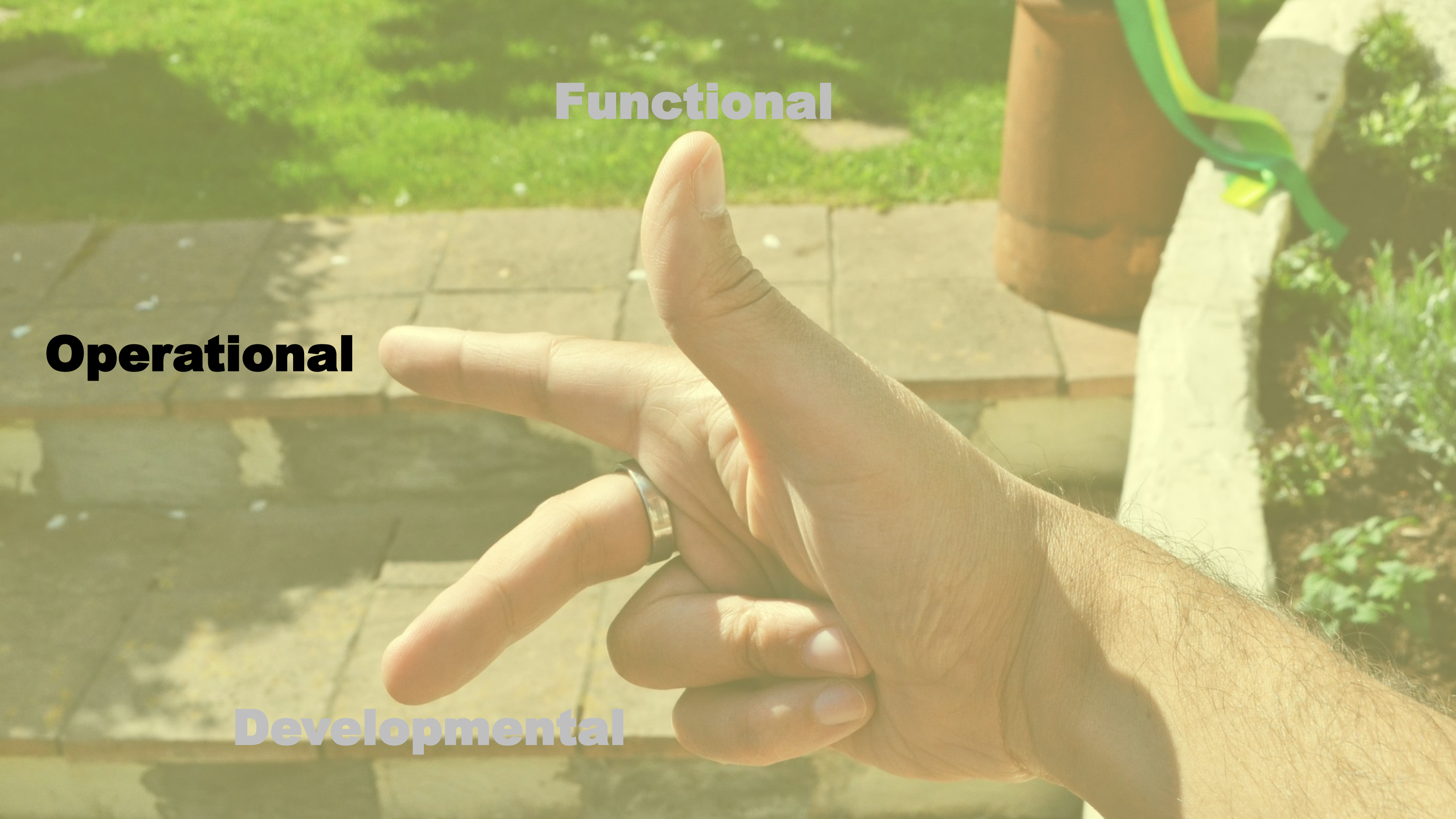




Functional

Operational

Developmental



Functional

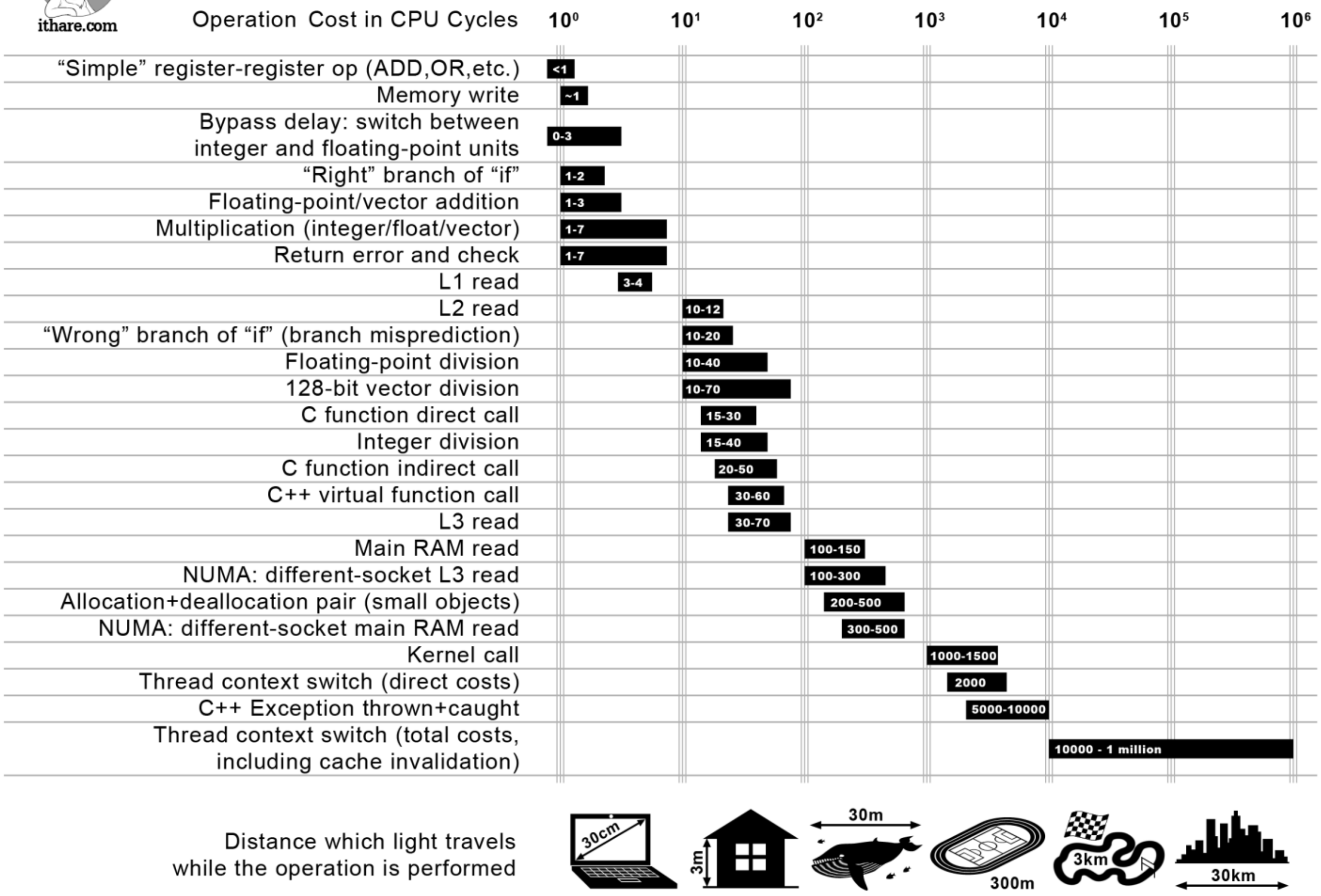
Operational

Developmental

This is the monstrosity in love,
lady, that the will is infinite,
and the execution confined;
that the desire is boundless,
and the act a slave to limit.

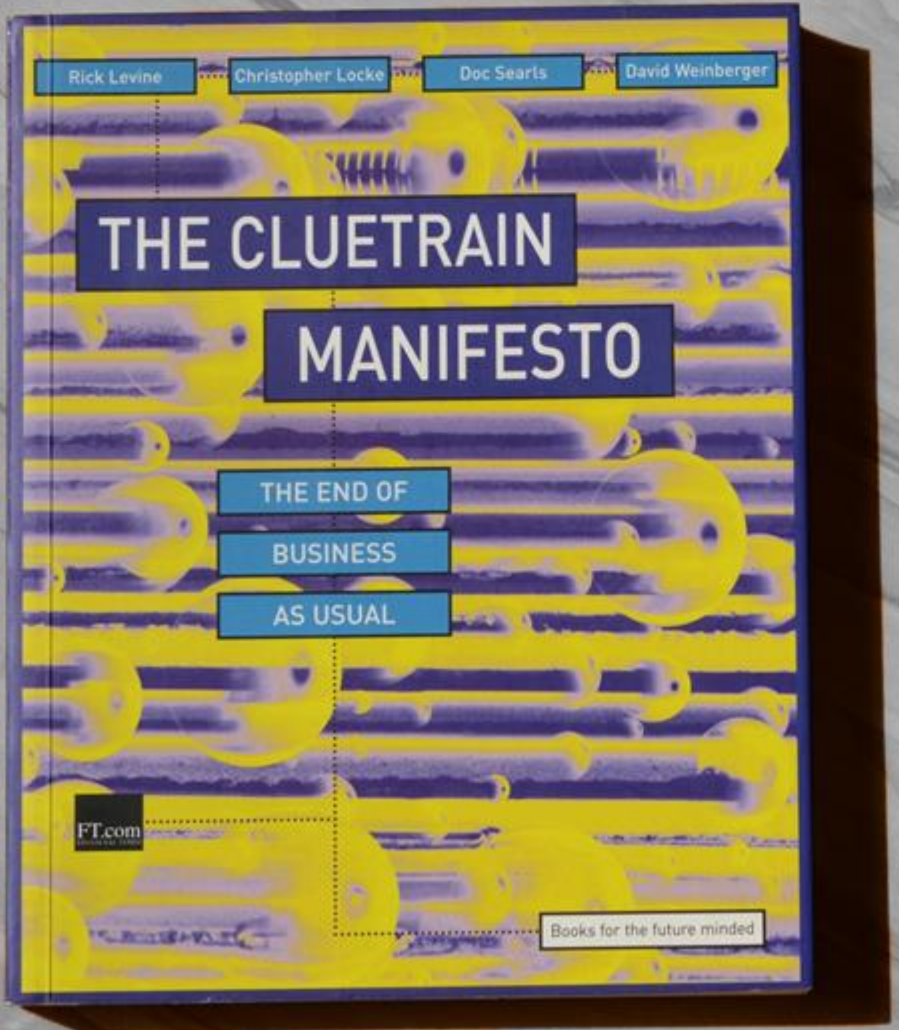
William Shakespeare
Troilus and Cressida

Not all CPU operations are created equal



Operation	Cost in CPU Cycles
"Simple" register-register op (ADD,OR,etc.)	<1
Memory write	~1
Bypass delay: switch between integer and floating-point units	0-3
"Right" branch of "if"	1-2
Floating-point/vector addition	1-3
Multiplication (integer/float/vector)	1-7
Return error and check	1-7
L1 read	3-4
L2 read	10-12
"Wrong" branch of "if" (branch misprediction)	10-20
Floating-point division	10-40
128-bit vector division	10-70
C function direct call	15-30
Integer division	15-40
C function indirect call	20-50
C++ virtual function call	30-60
L3 read	30-70
Main RAM read	100-150
NUMA: different-socket L3 read	100-300
Allocation+deallocation pair (small objects)	200-500
NUMA: different-socket main RAM read	300-500
Kernel call	1000-1500
Thread context switch (direct costs)	2000
C++ Exception thrown+caught	5000-10000
Thread context switch (total costs, including cache invalidation)	10000 - 1 million

http://ithare.com/infographics-operation-costs-in-cpu-clock-cycles/



Multitasking is really just rapid attention-switching.

And that'd be a useful skill, except it takes us a second or two to engage in a new situation we've graced with our focus.

So, the sum total of attention is actually decreased as we multitask.

Slicing your attention, in other words, is less like slicing potatoes than like slicing plums: you always lose some juice.

David Weinberger

$$t = t_1$$

completion time
for single thread

$$t = \frac{t_1}{\underset{\substack{\uparrow \\ \text{division of} \\ \text{labour}}}{n}}$$

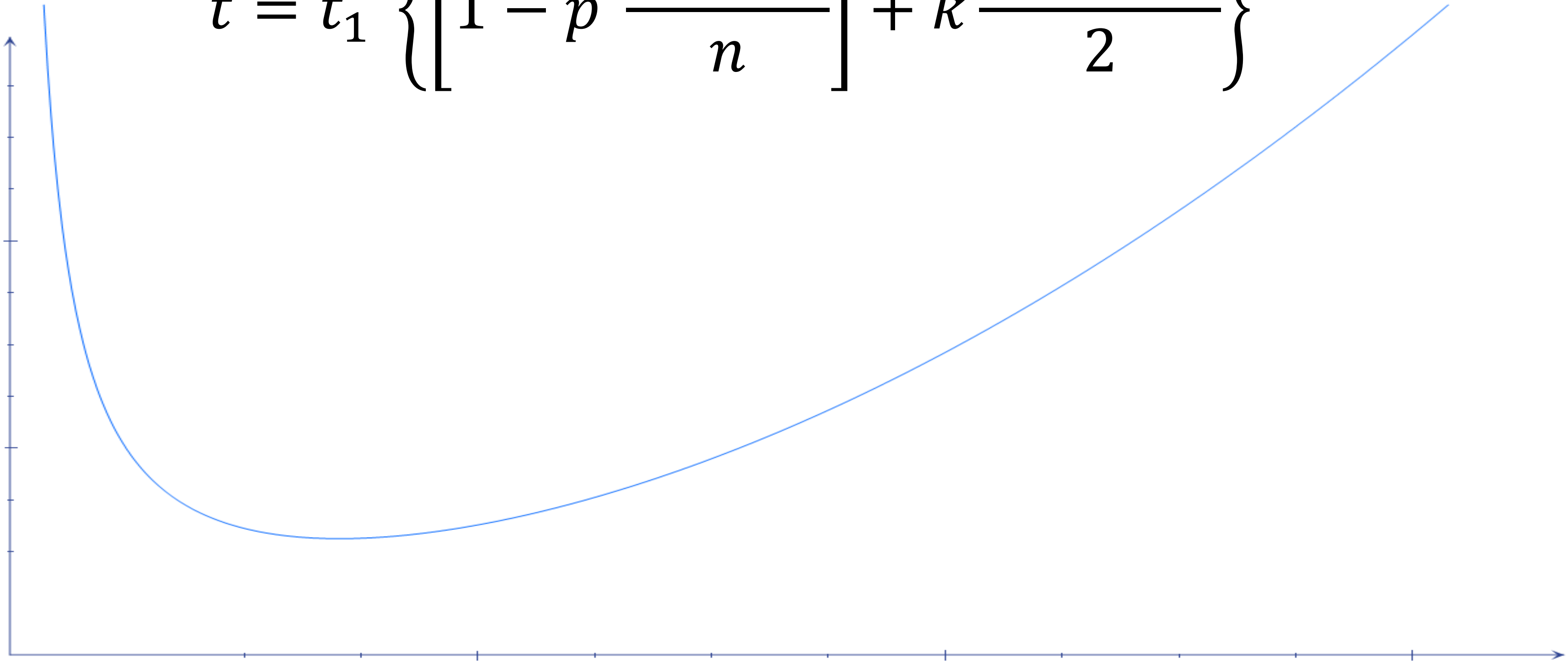
$$t = t_1 \left[1 - p\,\frac{(n-1)}{n}\right]$$

Amdahl's law

portion in
parallel

$$t = t_1 \left\{ \left[1 - p \, \frac{(n-1)}{n} \right] + \underset{\substack{\uparrow \\ \text{typical} \\ \text{communication} \\ \text{overhead}}}{k} \, \frac{n(n-1)}{\underset{\substack{\uparrow \\ \text{inter-thread} \\ \text{connections} \\ \text{(worst case)}}}{2}} \right\}$$

$$t = t_1 \left\{ \left[1 - p\,\frac{(n-1)}{n} \right] + k\,\frac{n(n-1)}{2} \right\}$$



Command-line tools can be 235x faster than your Hadoop cluster

Adam Drake
http://aadrake.com/command-line-tools-can-be-235x-faster-than-your-hadoop-cluster.html

```
template<
    typename Iterator,
    typename Mapping,
    typename Reduction,
    typename Value>
Value map_reduce(
    Iterator begin, Iterator end,
    Mapping mapping, Reduction reduction, Value initial)
{
    std::vector<std::thread> threads;
    for(auto to_map = begin; to_map != end; ++to_map)
        threads.push_back(std::thread(mapping, *to_map));
    for(auto & to_join : threads)
        to_join.join();
    return std::accumulate(begin, end, initial, reduction);
}
```

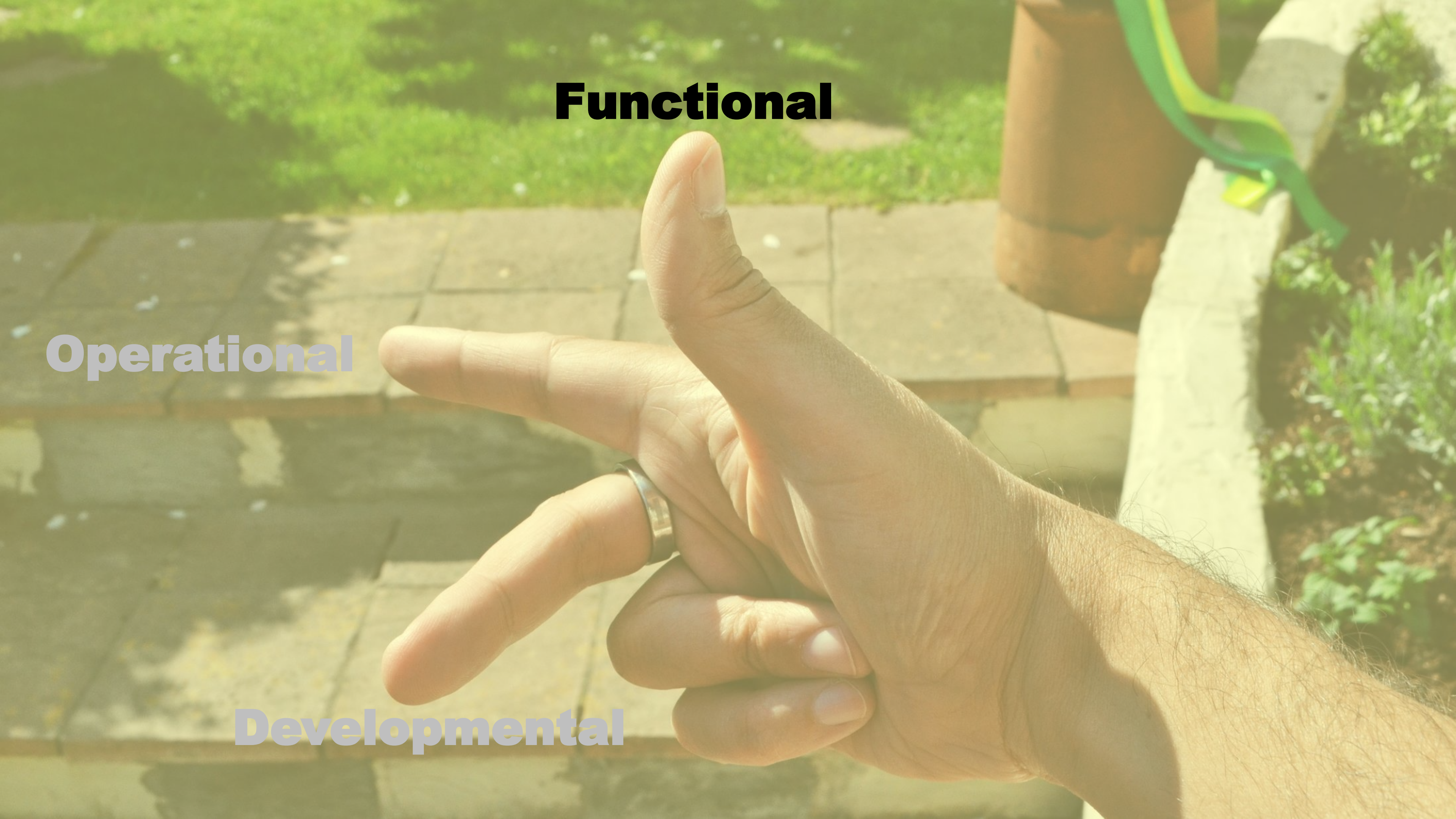
```
template<
    typename Iterator,
    typename Mapping,
    typename Reduction,
    typename Value>
auto map_reduce(
    Iterator begin, Iterator end,
    Mapping mapping, Reduction reduction, Value initial)
{
    std::vector<std::thread> threads;
    for(auto to_map = begin; to_map != end; ++to_map)
        threads.push_back(std::thread(mapping, *to_map));
    for(auto & to_join : threads)
        to_join.join();
    return std::accumulate(begin, end, initial, reduction);
}
```

```
auto map_reduce(
    auto begin, auto end, auto mapping, auto reduction, auto initial)
{
    std::vector<std::thread> threads;
    for(auto to_map = begin; to_map != end; ++to_map)
        threads.push_back(std::thread(mapping, *to_map));
    for(auto & to_join : threads)
        to_join.join();
    return std::accumulate(begin, end, initial, reduction);
}
```

```
auto map_reduce(
    auto begin, auto end, auto mapping, auto reduction, auto initial)
{
    std::for_each(std::execute::par_unseq, begin, end, mapping);
    return std::accumulate(begin, end, initial, reduction);
}
```

```
auto map_reduce(
    auto begin, auto end, auto mapping, auto reduction, auto initial)
{
    using namespace std::execute;
    std::for_each(par_unseq, begin, end, mapping);
    return std::accumulate(begin, end, initial, reduction);
}
```

```
auto map_reduce(
    auto begin, auto end, auto mapping, auto reduction, auto initial)
{
    using namespace std::execute;
    std::for_each(par_unseq, begin, end, mapping);
    return std::reduce(par_unseq, begin, end, initial, reduction);
}
```



Functional

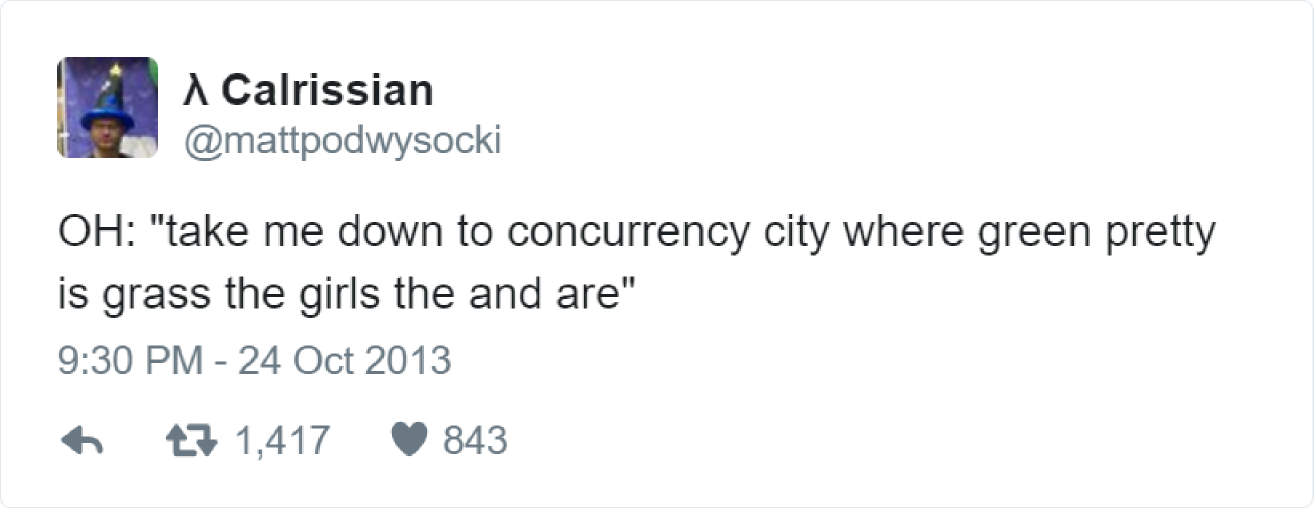
Operational

Developmental

> A large fraction of the flaws in software development are due to programmers not fully understanding all the possible states their code may execute in.
>
> In a multithreaded environment, the lack of understanding and the resulting problems are greatly amplified, almost to the point of panic if you are paying attention.

John Carmack

http://www.gamasutra.com/view/news/169296/Indepth_Functional_programming_in_C.php



λ Calrissian
@mattpodwysocki

OH: "take me down to concurrency city where green pretty is grass the girls the and are"

9:30 PM - 24 Oct 2013

↩ ⇄ 1,417 ♥ 843

https://twitter.com/mattpodwysocki/status/393474697699921921

There are several ways to address the problem of deadlock...

http://www.cs.rpi.edu/academics/courses/fall04/os/c10/index.html

Just ignore it and hope it doesn't happen.

Ostrich Algorithm

http://www.cs.rpi.edu/academics/courses/fall04/os/c10/index.html

Detection and recovery — if it happens, take action.

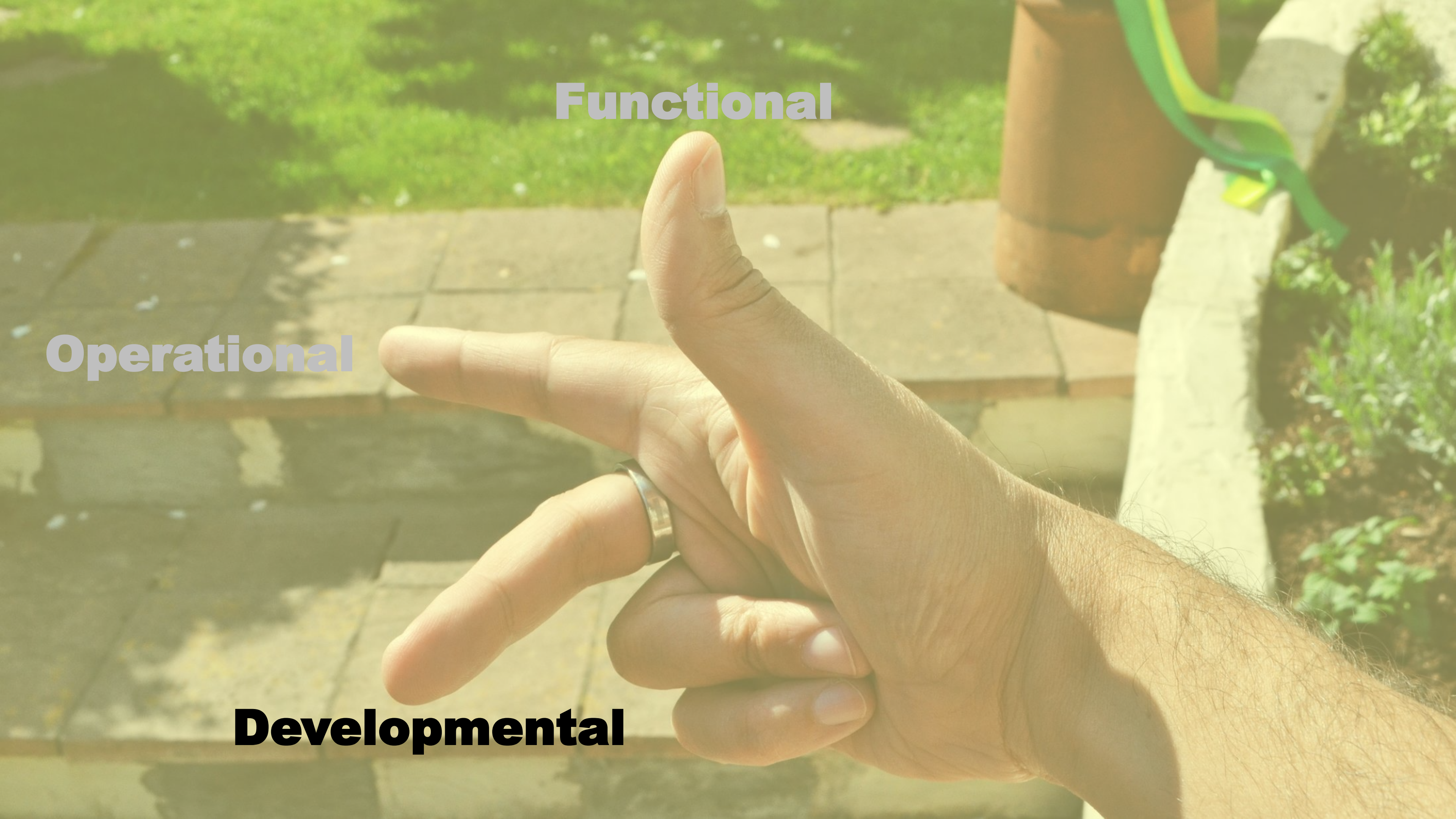
http://www.cs.rpi.edu/academics/courses/fall04/os/c10/index.html

Dynamic avoidance by careful resource allocation — check to see if a resource can be granted, and if granting it will cause deadlock, don't grant it.

http://www.cs.rpi.edu/academics/courses/fall04/os/c10/index.html

Prevention — change the rules.

http://www.cs.rpi.edu/academics/courses/fall04/os/c10/index.html

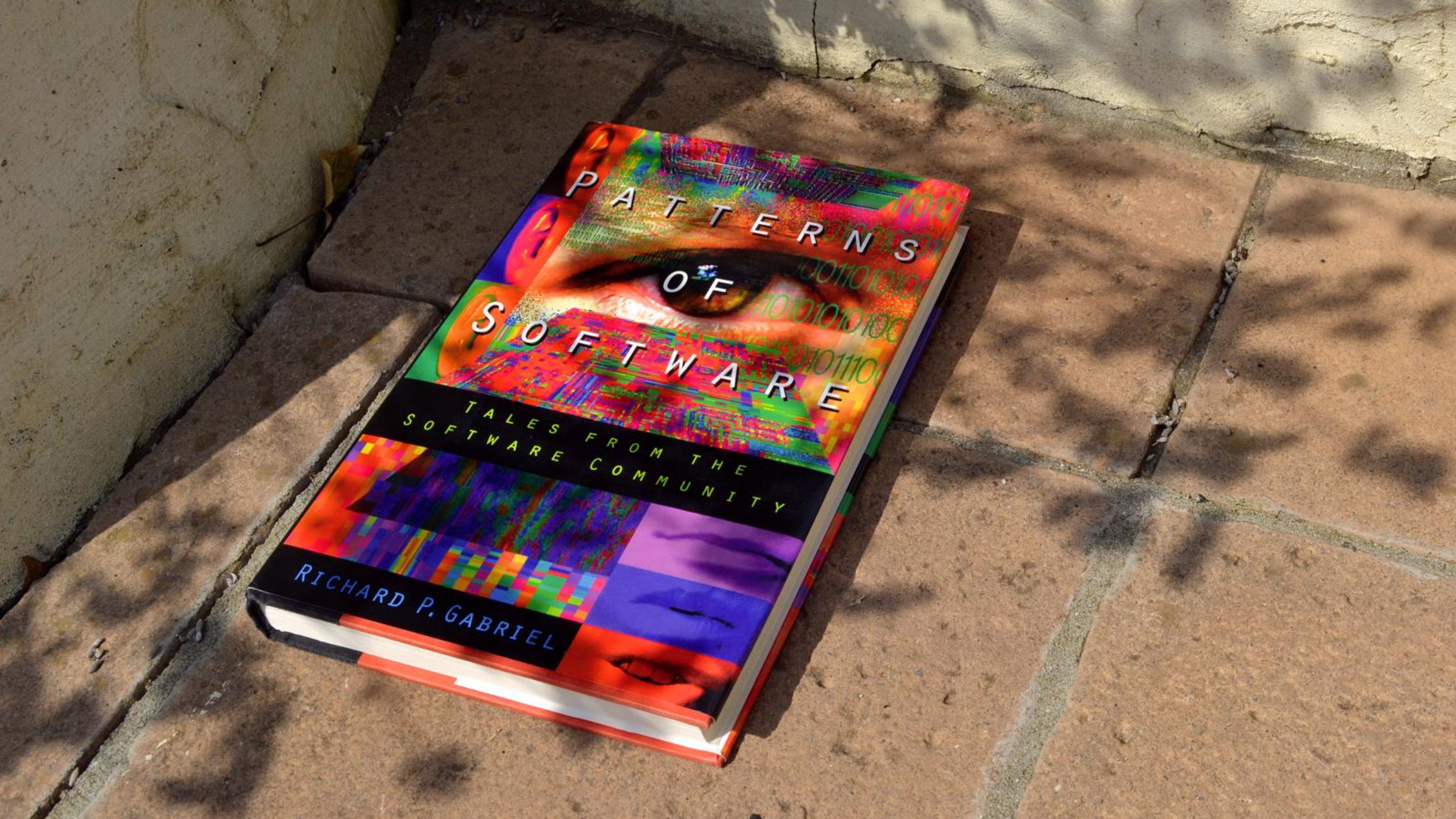


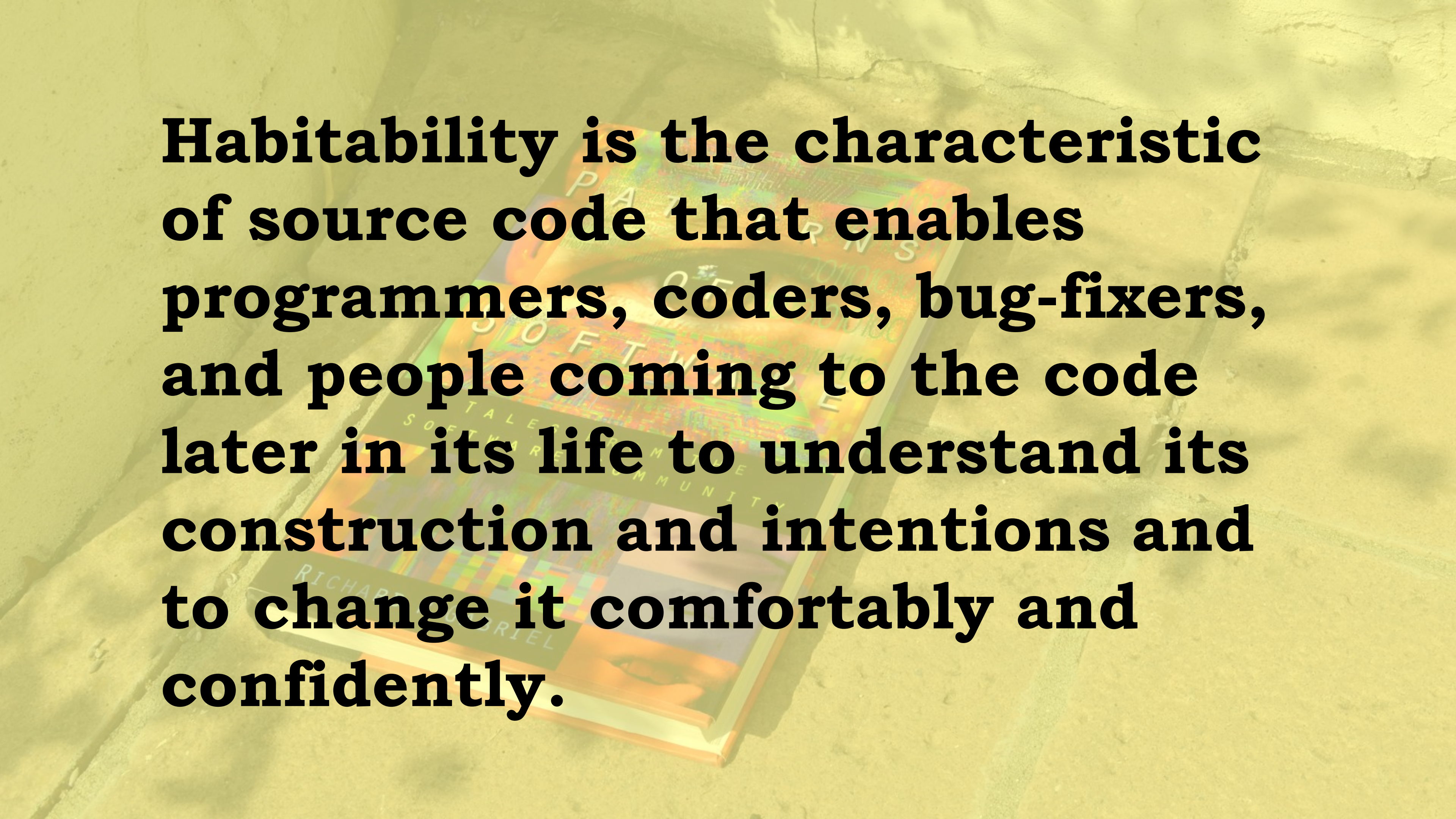
Functional

Operational

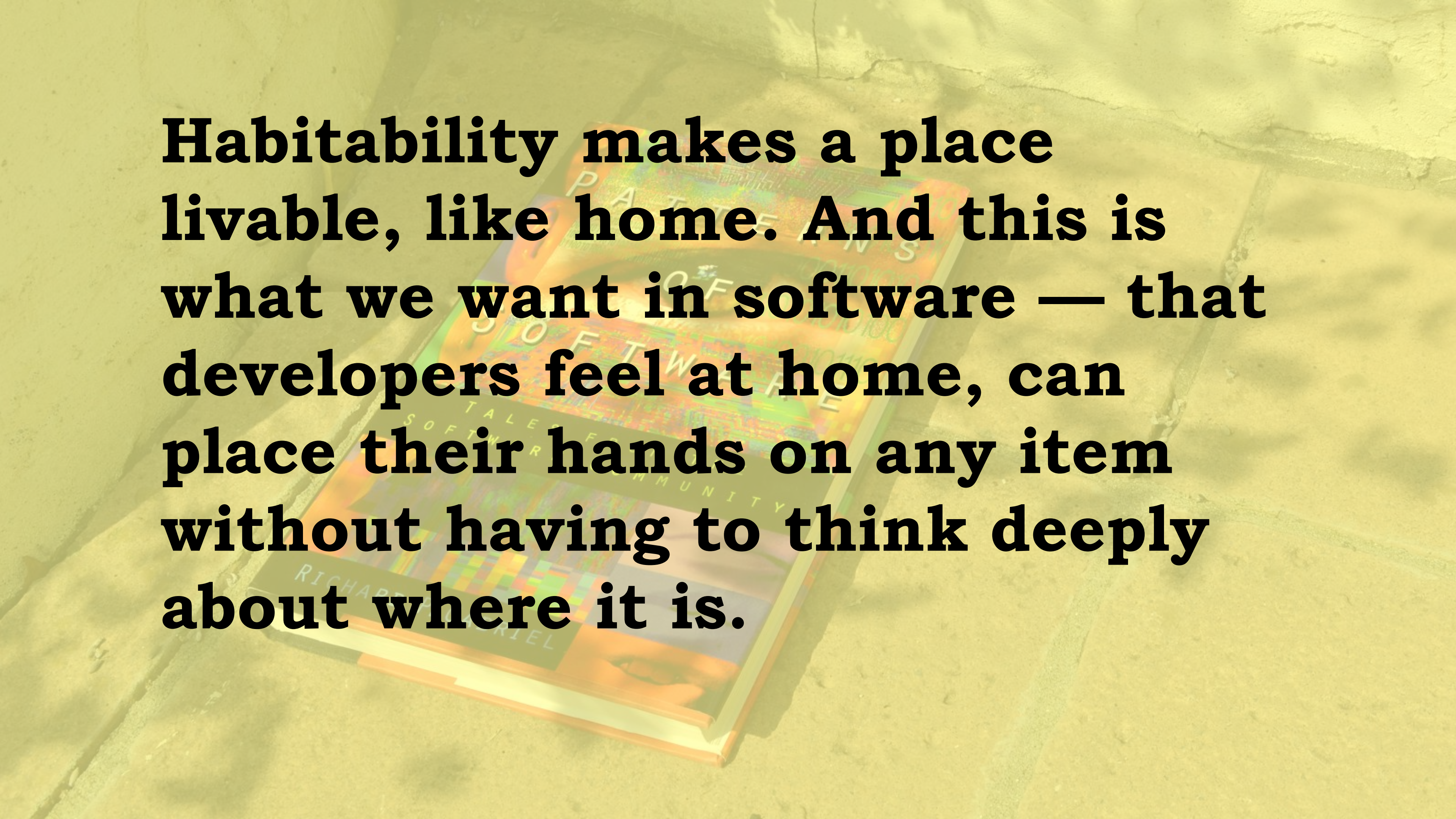
Developmental

habitable


PATTERNS OF SOFTWARE
TALES FROM THE SOFTWARE COMMUNITY
RICHARD P. GABRIEL



Habitability is the characteristic of source code that enables programmers, coders, bug-fixers, and people coming to the code later in its life to understand its construction and intentions and to change it comfortably and confidently.



Habitability makes a place livable, like home. And this is what we want in software — that developers feel at home, can place their hands on any item without having to think deeply about where it is.

testable

Simple Testing Can Prevent Most Critical Failures

An Analysis of Production Failures in Distributed Data-Intensive Systems

https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-yuan.pdf

A majority of the production failures (77%) can be reproduced by a unit test.

https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-yuan.pdf

We want our code to be unit testable.

What is a unit test?

A test is not a unit test if:

- It talks to the database
- It communicates across the network
- It touches the file system
- It can't run at the same time as any of your other unit tests
- You have to do special things to your environment (such as editing config files) to run it.

Michael Feathers

http://www.artima.com/weblogs/viewpost.jsp?thread=126923

A unit test is a test of behaviour whose success or failure is wholly determined by the correctness of the test and the correctness of the unit under test.

Kevlin Henney

http://www.theregister.co.uk/2007/07/28/what_are_your_units/

What do we want from unit tests?

When a unit test passes, it shows the code is correct.

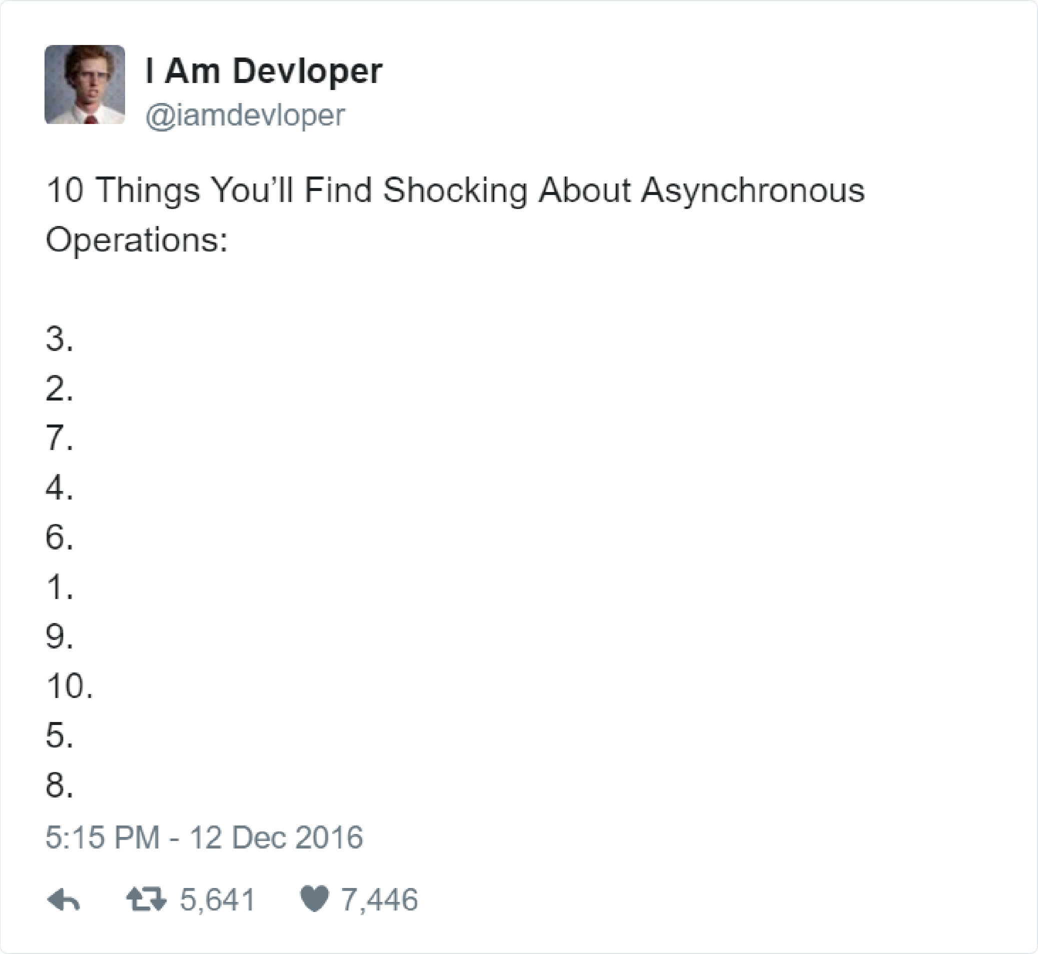
When a unit test fails, it shows the code is incorrect.

isolated

immutable

sequential

asynchronous



I Am Devloper
@iamdevloper

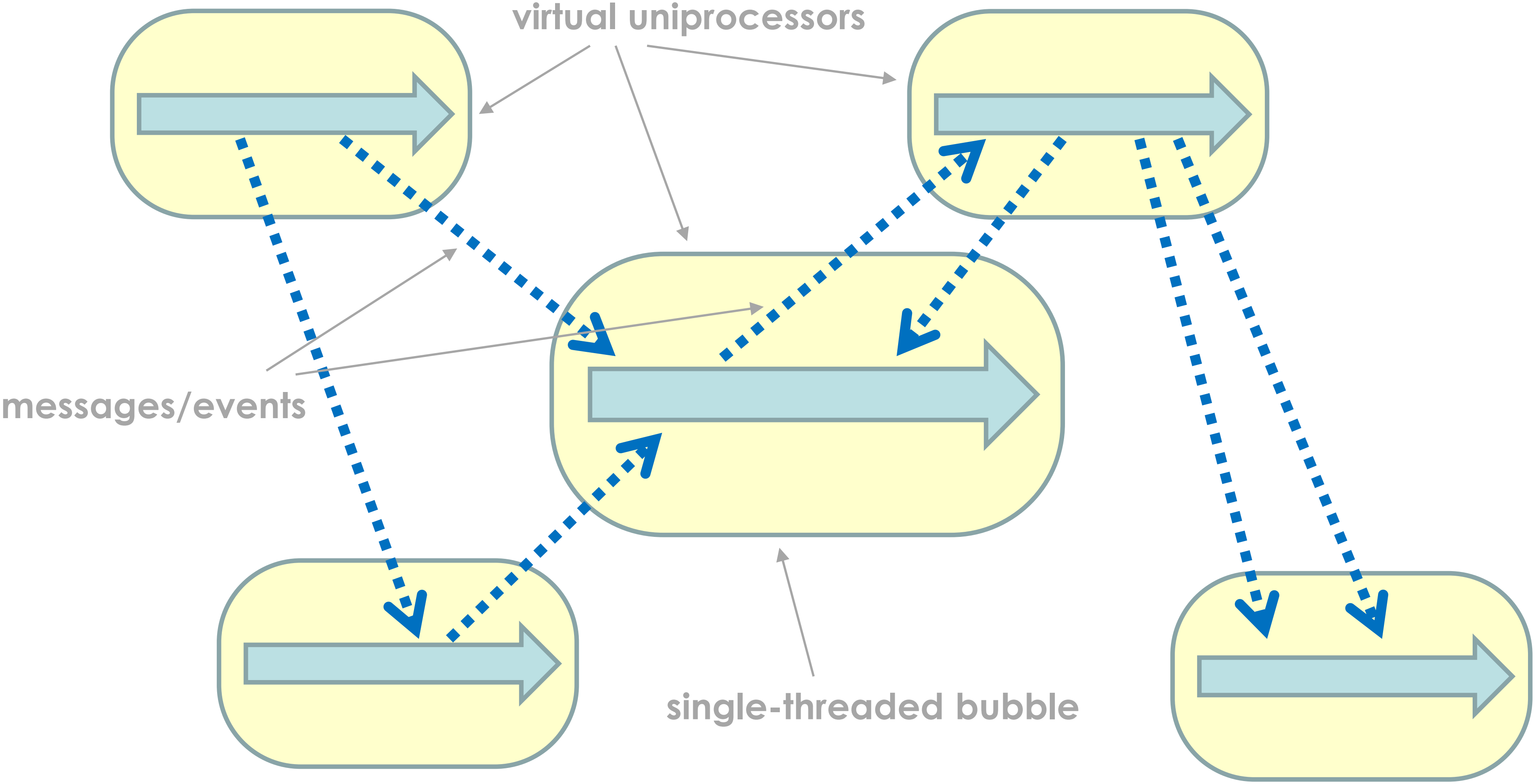
10 Things You'll Find Shocking About Asynchronous Operations:

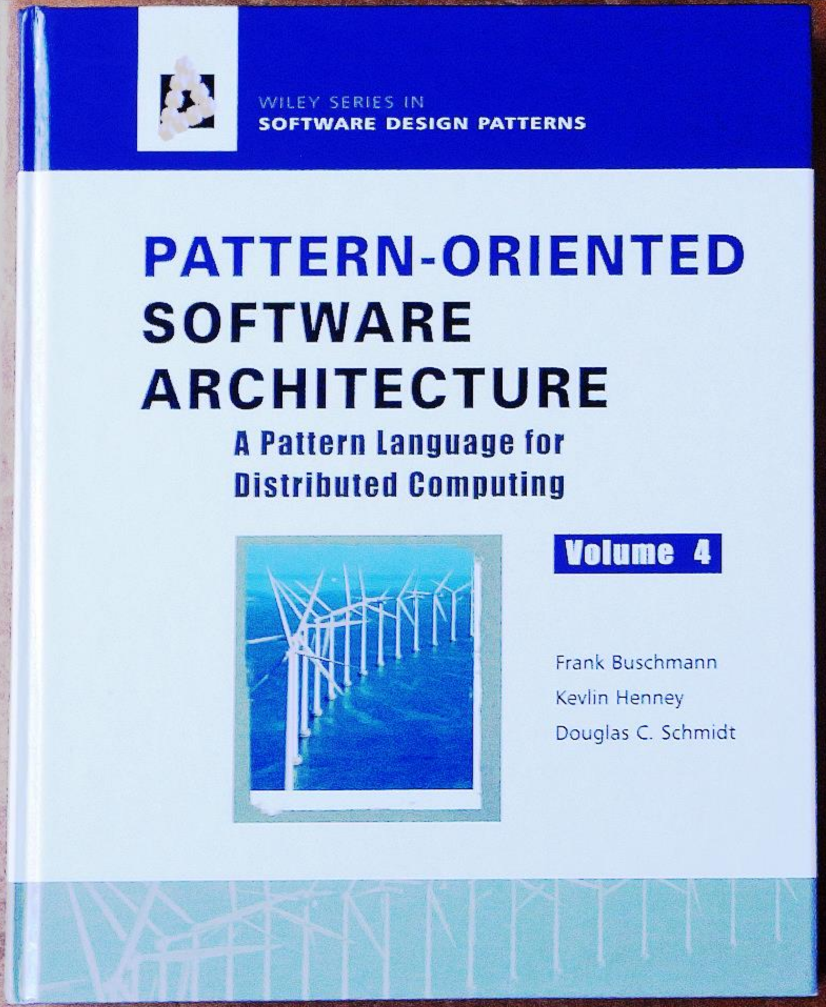
3.
2.
7.
4.
6.
1.
9.
10.
5.
8.

5:15 PM - 12 Dec 2016

↩ ⇄ 5,641 ♥ 7,446

https://twitter.com/iamdevloper/status/808344541669498881


virtual uniprocessors
messages/events
single-threaded bubble



WILEY SERIES IN
SOFTWARE DESIGN PATTERNS

PATTERN-ORIENTED SOFTWARE ARCHITECTURE

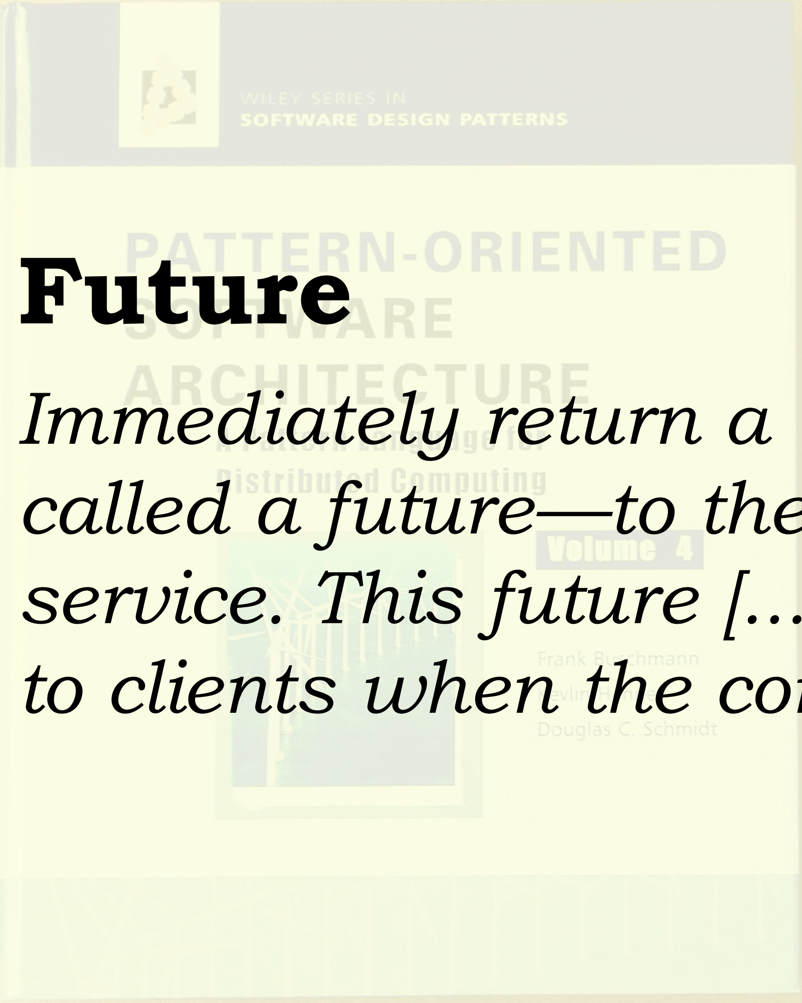
A Pattern Language for Distributed Computing

Volume 4

Frank Buschmann
Kevlin Henney
Douglas C. Schmidt

Future

Immediately return a ‘virtual’ data object—called a future—to the client when it invokes a service. This future [...] only provides a value to clients when the computation is complete.



```
ResultType result = function();
...
```

```
...
ResultType result = function();
```

```
std::future<ResultType>
    iou = std::async(function);

...

ResultType result = iou.get();
```

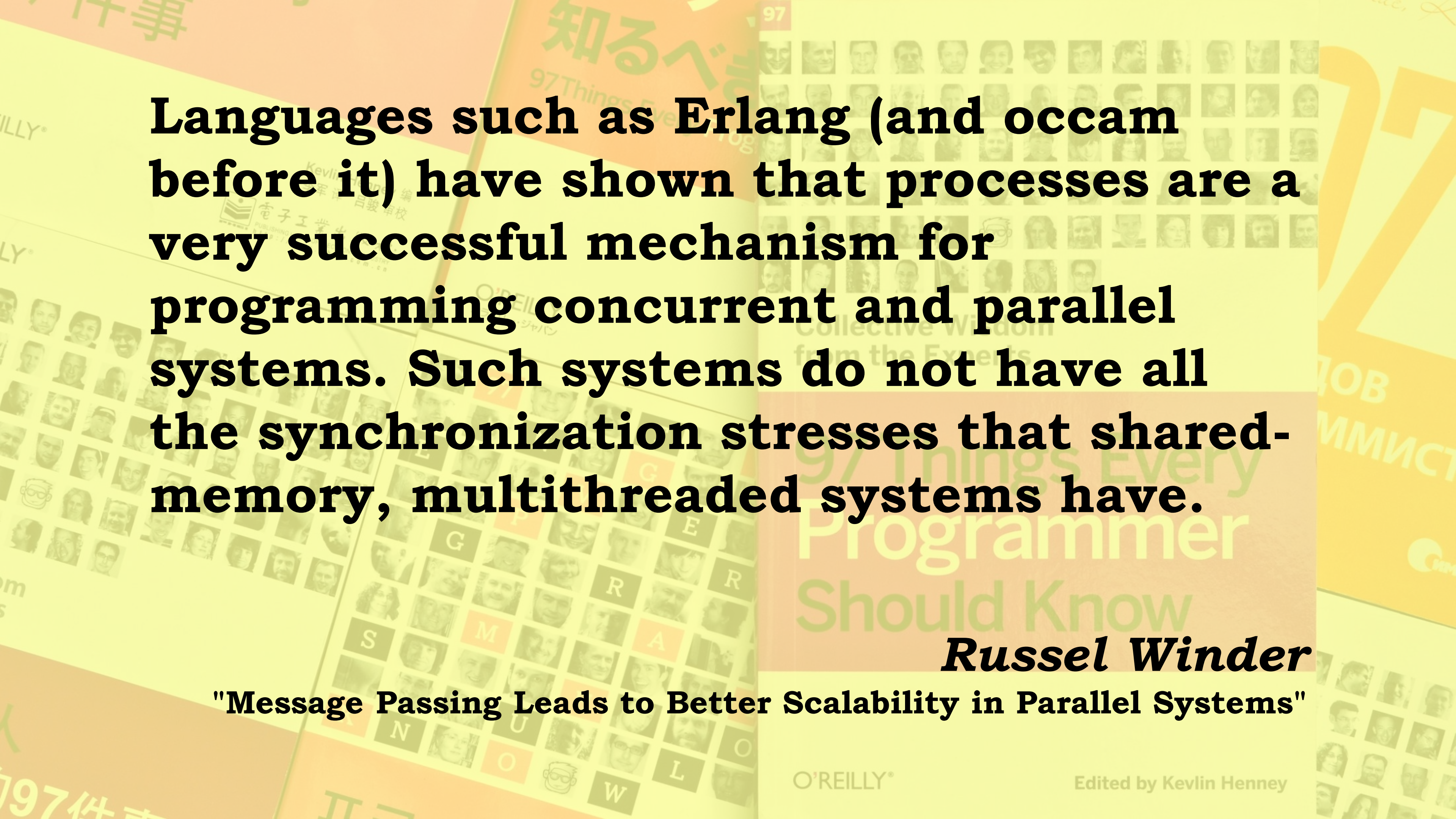
```
joiner<ResultType>
    iou = thread(function);

...

ResultType result = iou();
```

"C++ Threading", *ACCU Conference*, April 2003
"More C++ Threading", *ACCU Conference*, April 2004
"N1883: Preliminary Threading Proposal for TR2", *JTC1/SC22/WG21*, August 2005

Instead of using threads and shared memory as our programming model, we can use processes and message passing. Process here just means a protected independent state with executing code, not necessarily an operating system process.

Russel Winder

"Message Passing Leads to Better Scalability in Parallel Systems"

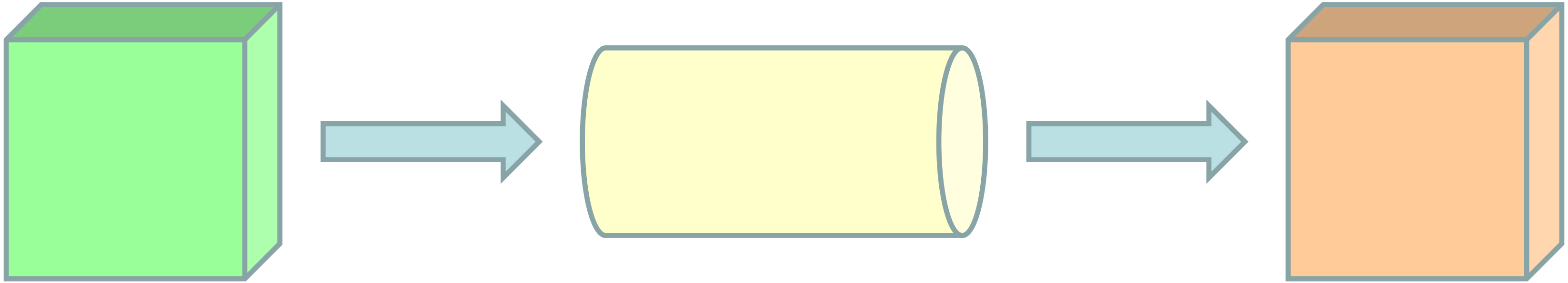

97 Things Every Programmer Should Know
Collective Wisdom from the Experts
O'REILLY
Edited by Kevlin Henney

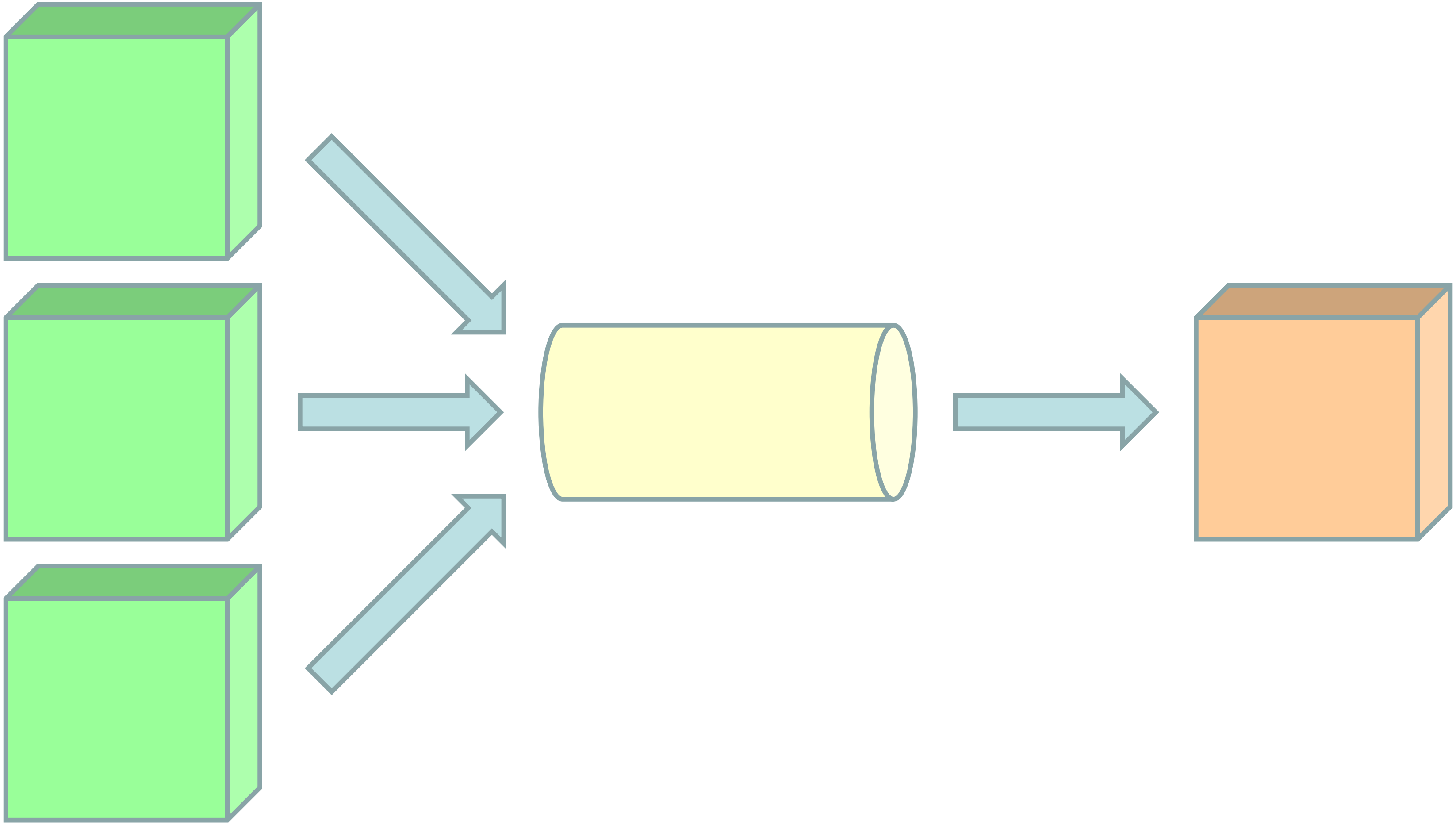
Languages such as Erlang (and occam before it) have shown that processes are a very successful mechanism for programming concurrent and parallel systems. Such systems do not have all the synchronization stresses that shared-memory, multithreaded systems have.

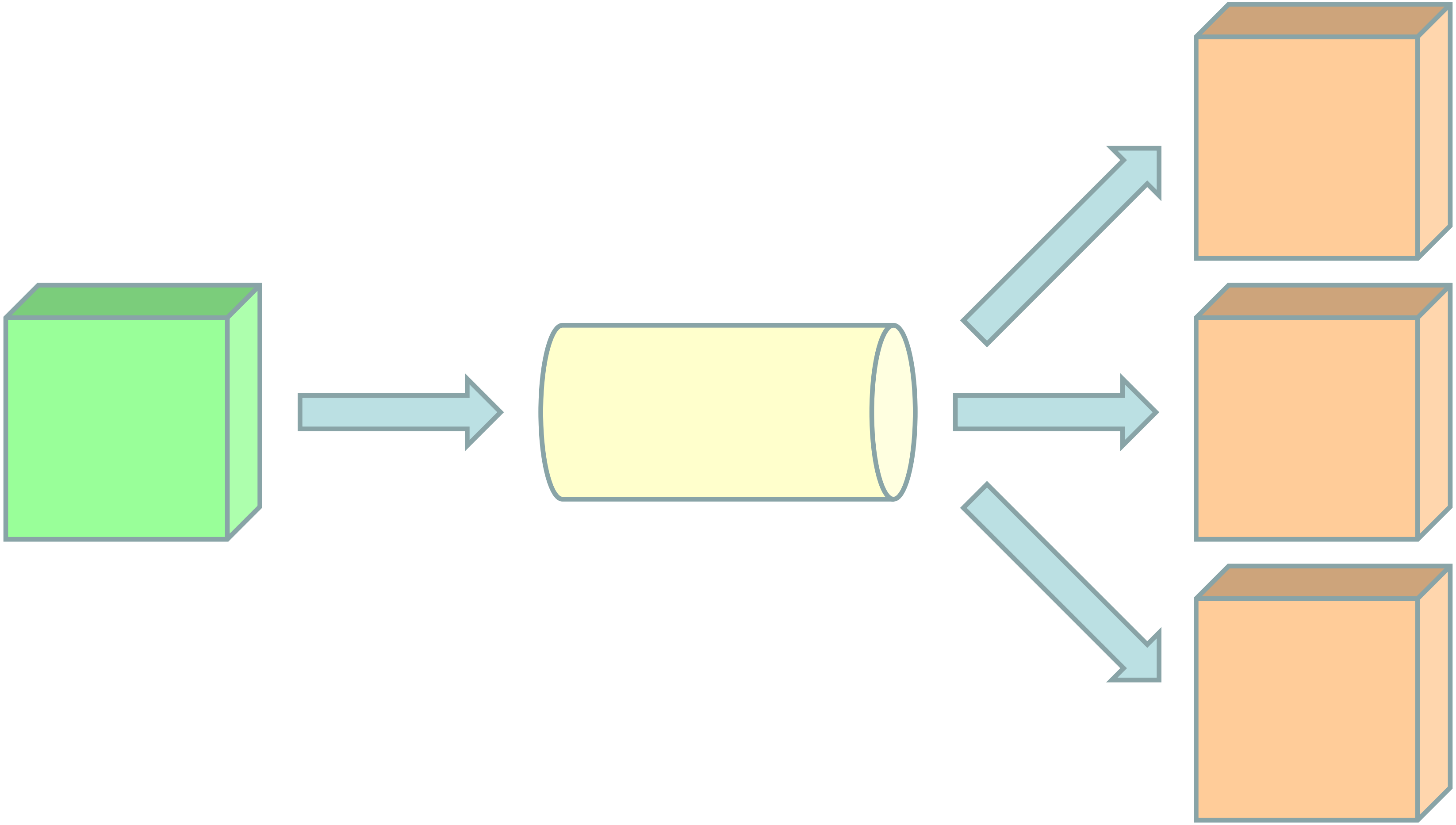
Russel Winder

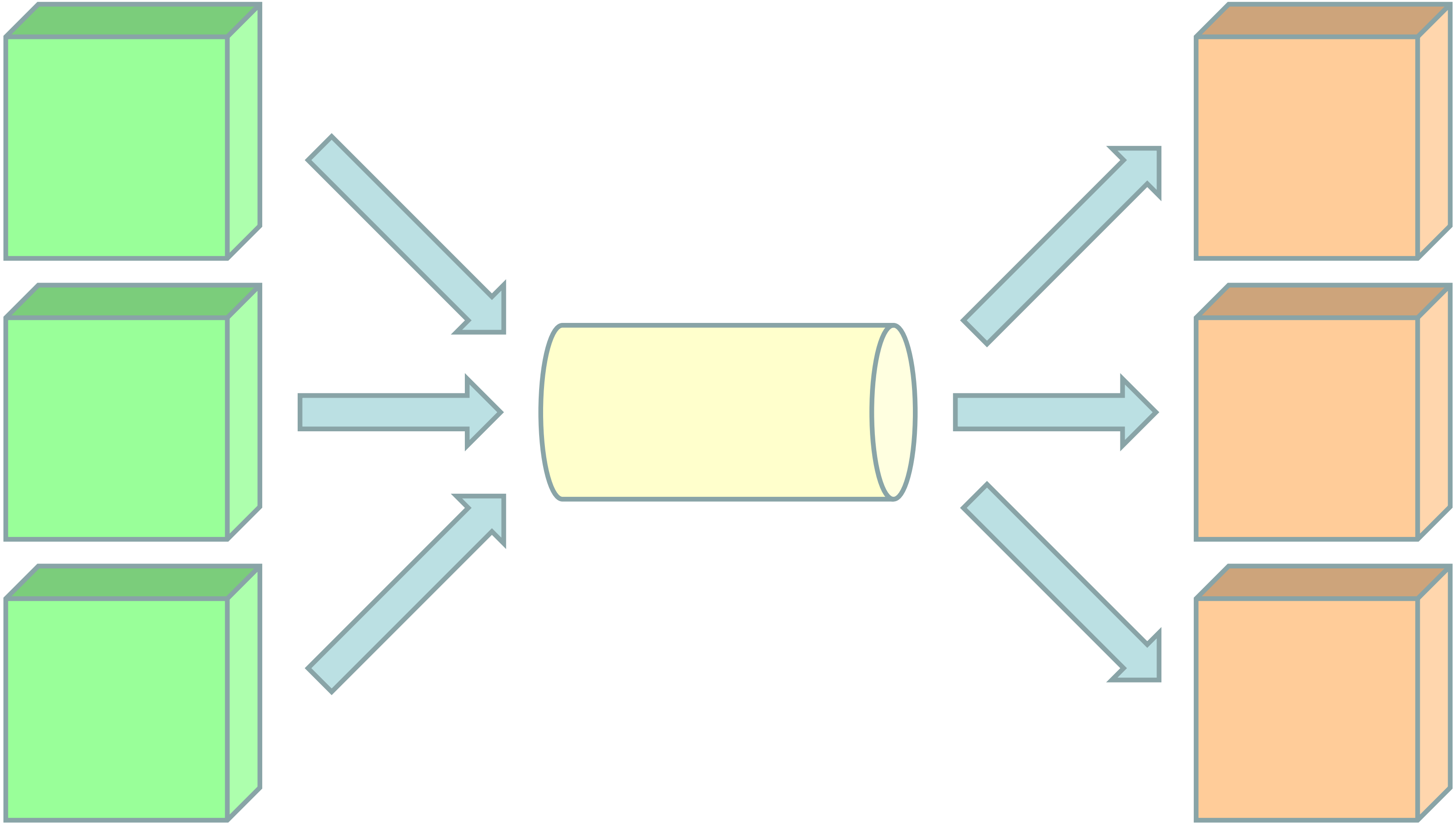
"Message Passing Leads to Better Scalability in Parallel Systems"

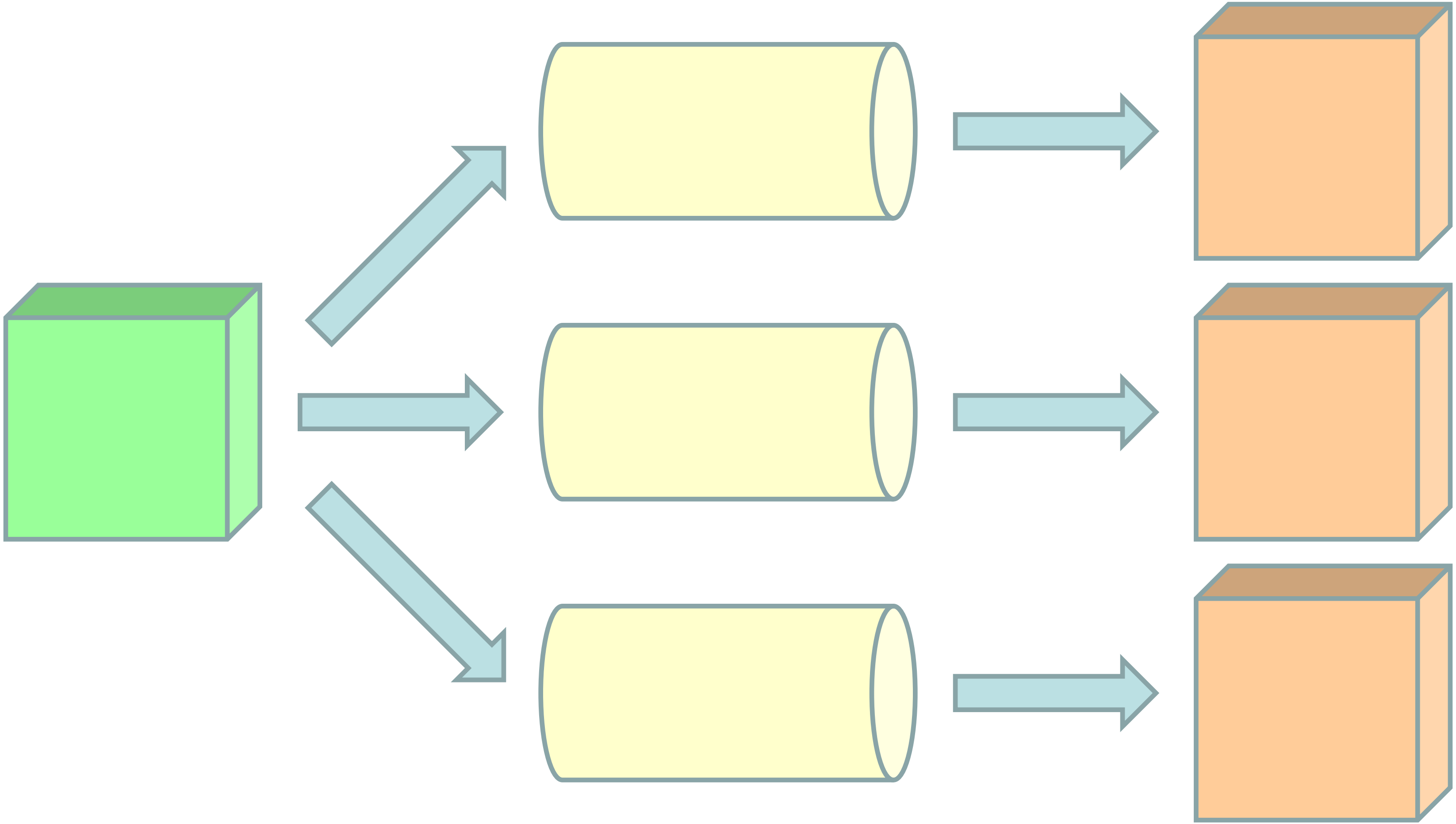
queues

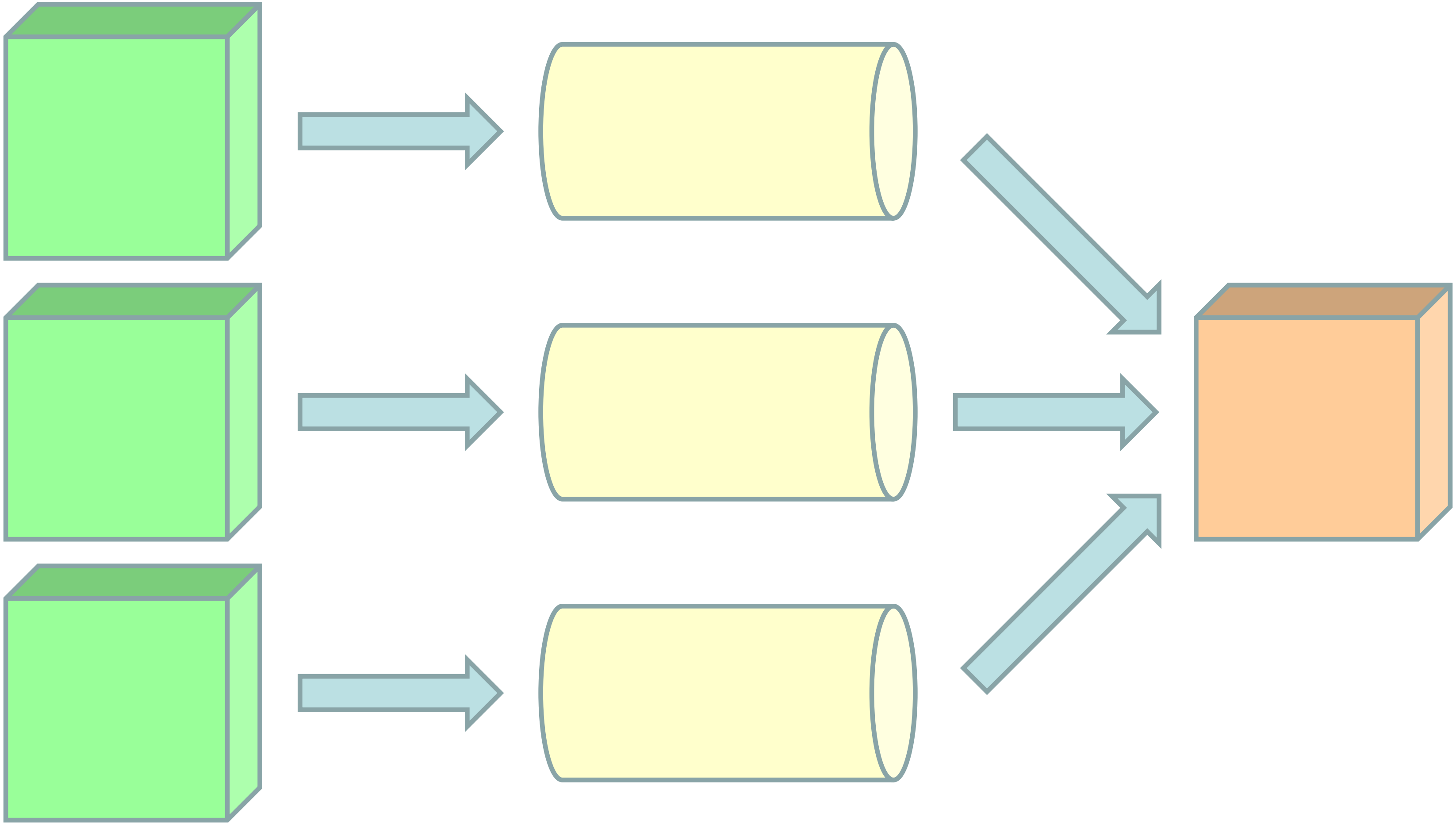


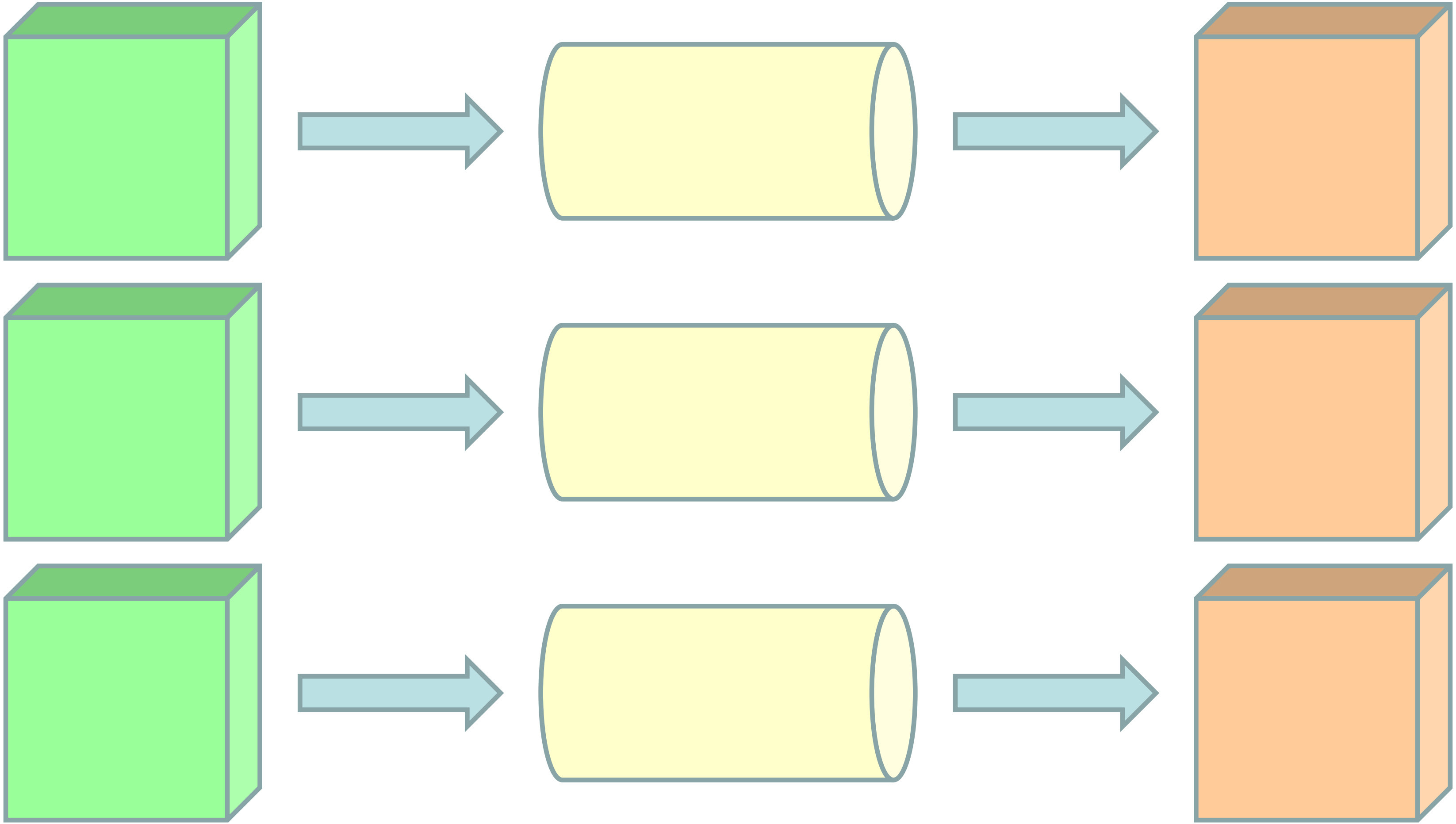












```
template<typename ValueType>
class queue
{
public:
    void send(const ValueType &);
    bool try_receive(ValueType &);
private:
    ...
};
```

```
template<typename ValueType>
class queue
{
public:
    void send(const ValueType &);
    bool try_receive(ValueType &);
private:
    std::deque<ValueType> fifo;
};
```

```
template<typename ValueType>
class queue
{
public:
    void send(const ValueType & to_send)
    {
        fifo.push_back(to_send);
    }
    ...
};
```

```
template<typename ValueType>
class queue
{
public:
    ...
    bool try_receive(ValueType & to_receive)
    {
        bool received = false;

        if (!fifo.empty())
        {
            to_receive = fifo.front();
            fifo.pop_front();
            received = true;
        }

        return received;
    }
    ...
};
```

```
template<typename ValueType>
class queue
{
public:
    void send(const ValueType &);
    bool try_receive(ValueType &);
private:
    std::mutex key;
    std::deque<ValueType> fifo;
};
```

```
void send(const ValueType & to_send)
{
    std::lock_guard<std::mutex> guard(key);
    fifo.push_back(to_send);
}
```

```
bool try_receive(ValueType & to_receive)
{
    bool received = false;

    if (key.try_lock())
    {
        std::lock_guard<std::mutex> guard(key, std::adopt_lock);

        if (!fifo.empty())
        {
            to_receive = fifo.front();
            fifo.pop_front();
            received = true;
        }
    }

    return received;
}
```

```
template<typename ValueType>
class queue
{
public:
    void send(const ValueType &);
    bool try_receive(ValueType &);
private:
    std::mutex key;
    std::deque<ValueType> fifo;
};
```

```
template<typename ValueType>
class queue
{
public:
    void send(const ValueType &);
    void receive(ValueType &);
    bool try_receive(ValueType &);
private:
    std::mutex key;
    std::condition_variable_any non_empty;
    std::deque<ValueType> fifo;
};
```

```
template<typename ValueType>
class queue
{
public:
    void send(const ValueType &);
    bool try_send(const ValueType &);
    void receive(ValueType &);
    bool try_receive(ValueType &);
    queue();
    explicit queue(std::size_t max_size);
private:
    std::mutex key;
    std::condition_variable_any non_empty, non_full;
    std::size_t max_size;
    std::deque<ValueType> fifo;
};
```

```
template<typename ValueType>
class queue
{
public:
    void send(const ValueType &);
    void receive(ValueType &);
    bool try_receive(ValueType &);
private:
    std::mutex key;
    std::condition_variable_any non_empty;
    std::deque<ValueType> fifo;
};
```

```
void send(const ValueType & to_send)
{
    std::lock_guard<std::mutex> guard(key);
    fifo.push_back(to_send);
    non_empty.notify_all();
}
```

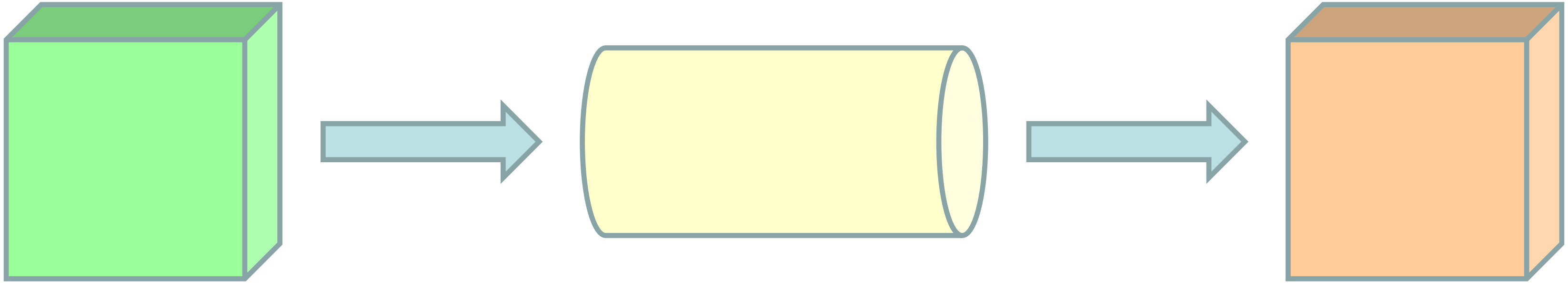
```
void receive(ValueType & to_receive)
{
    std::lock_guard<std::mutex> guard(key);
    non_empty.wait(
        key,
        [this]
        {
            return !fifo.empty();
        });
    to_receive = fifo.front();
    fifo.pop_front();
}
```

```
template<typename ValueType>
class queue
{
public:
    void send(const ValueType &);
    void receive(ValueType &);
    bool try_receive(ValueType &);
    void operator<<(const ValueType &);
    void operator>>(ValueType &);
private:
    std::mutex key;
    std::condition_variable_any non_empty;
    std::deque<ValueType> fifo;
};
```

```
template<typename ValueType>
class queue
{
public:
    void send(const ValueType &);
    void receive(ValueType &);
    bool try_receive(ValueType &);
    void operator<<(const ValueType &);
    receiving operator>>(ValueType &);
private:
    std::mutex key;
    std::condition_variable_any non_empty;
    std::deque<ValueType> fifo;
};
```

```
template<typename ValueType>
class queue
{
public:
    void send(const ValueType &);
    void receive(ValueType &);
    bool try_receive(ValueType &);

    void operator<<(const ValueType & to_send)
    {
        send(to_send);
    }
    receiving operator>>(ValueType & to_receive);
    {
        return receiving(this, to_receive);
    }
    ...
};
```

```
class receiving
{
public:
    receiving(queue * that, ValueType & to_receive)
    : that(that), to_receive(to_receive)
    {
    }
    receiving(receiving && other)
    : that(other.that), to_receive(other.to_receive)
    {
        other.that = nullptr;
    }
    operator bool()
    {
        auto from = that;
        that = nullptr;
        return from && from->try_receive(to_receive);
    }
    ~receiving()
    {
        if (that)
            that->receive(to_receive);
    }
private:
    queue * that;
    ValueType & to_receive;
};
```

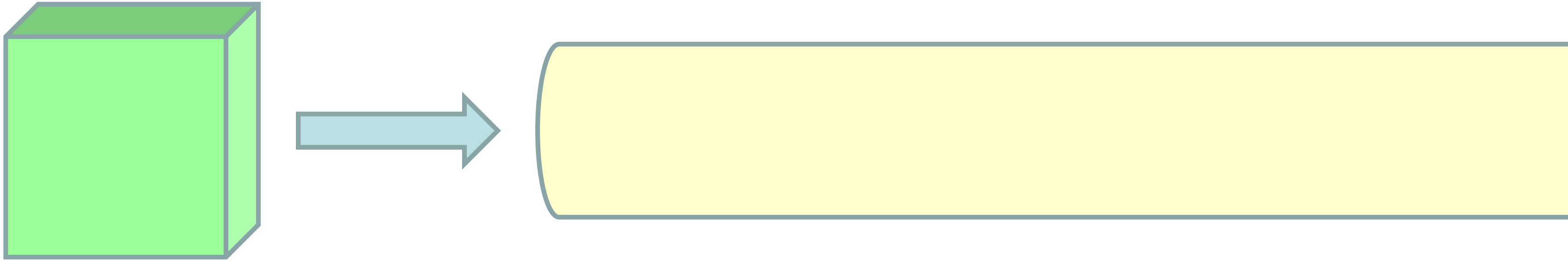


N

buffered

bounded

asynchronous



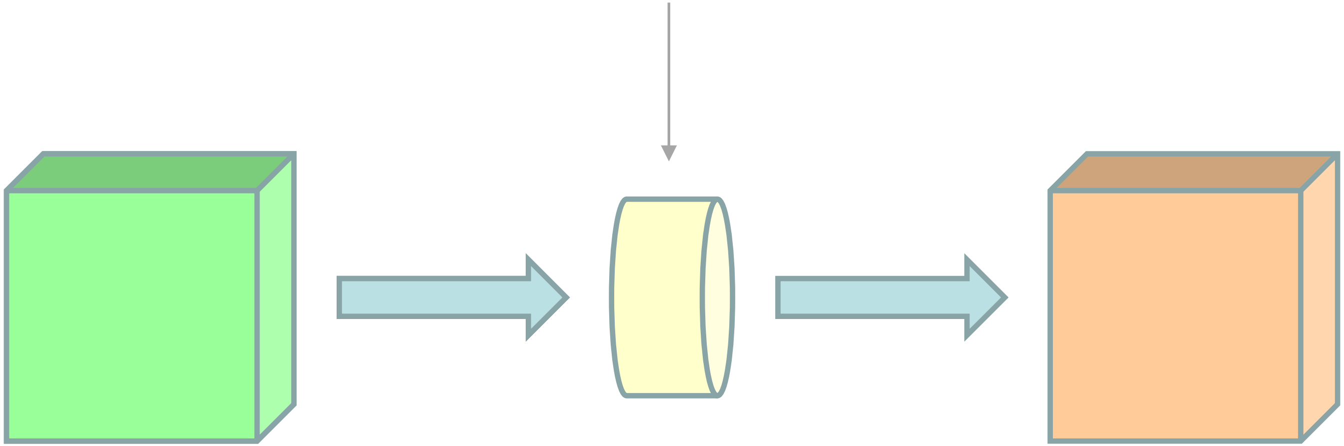
N = ∞

buffered

unbounded

asynchronous

future

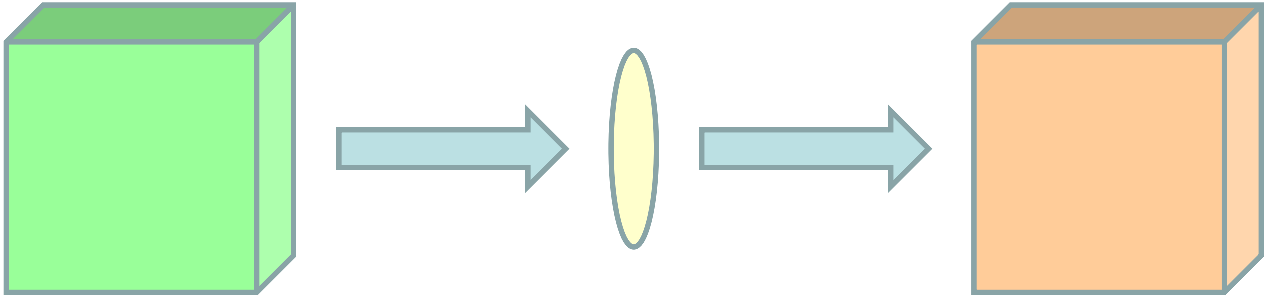


N = 1

buffered

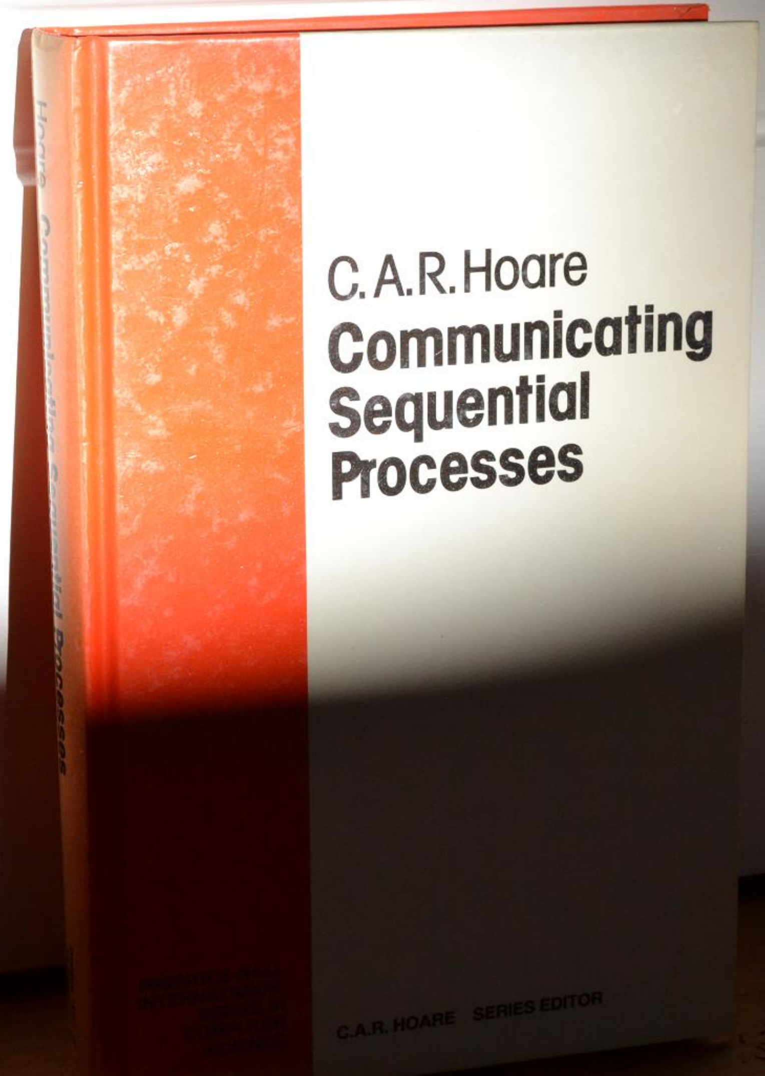
bounded

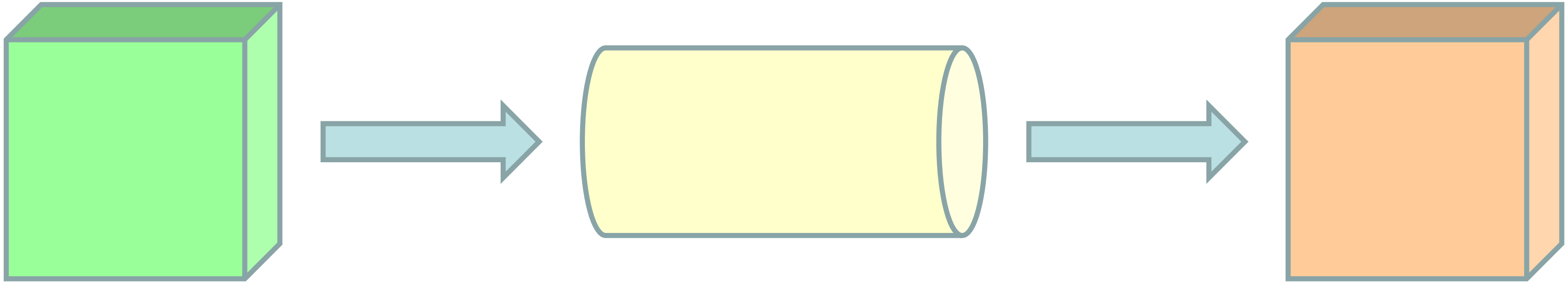
asynchronous

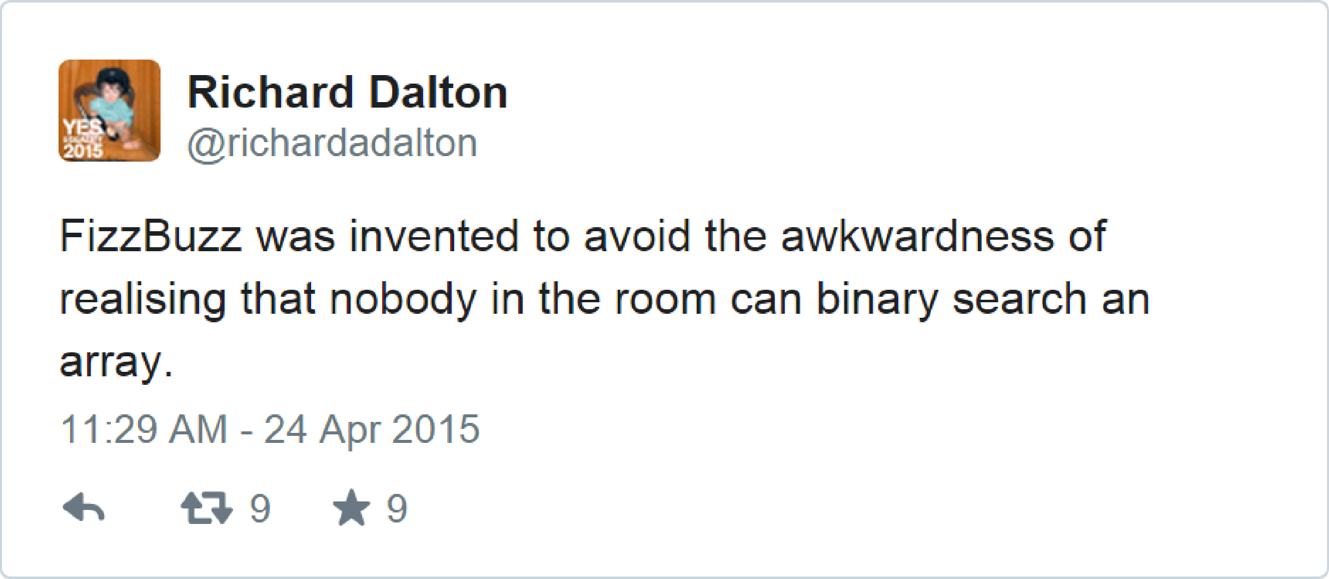


N = 0
unbuffered
bounded
synchronous

channels


C.A.R. Hoare
Communicating Sequential Processes
C.A.R. HOARE SERIES EDITOR
Hoare Communicating Sequential Processes
PRENTICE-HALL INTERNATIONAL SERIES IN COMPUTER SCIENCE





Richard Dalton
@richardadalton

FizzBuzz was invented to avoid the awkwardness of realising that nobody in the room can binary search an array.

11:29 AM - 24 Apr 2015

https://twitter.com/richardadalton/status/591534529086693376

```go
func fizzbuzz(n int) string {
      result := ""
      if n % 3 == 0 {
            result += "Fizz"
      }
      if n % 5 == 0 {
            result += "Buzz"
      }
      if result == "" {
            result = strconv.Itoa(n)
      }
      return result
}
```

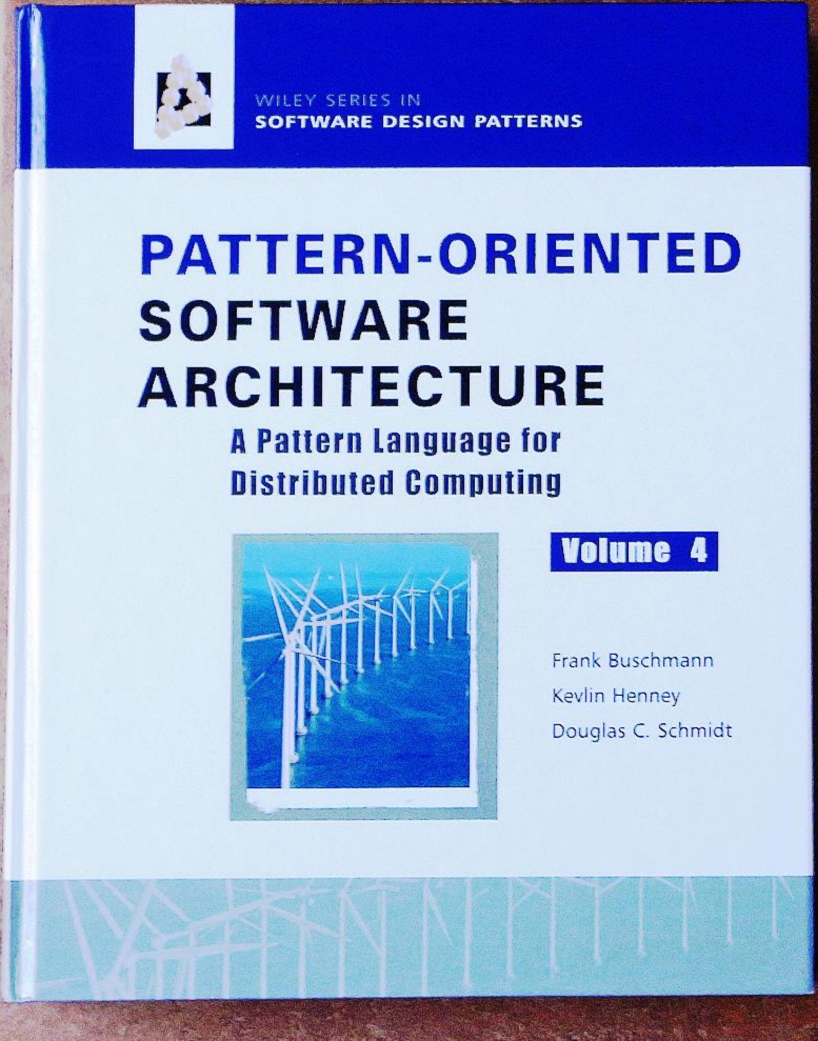
```
func fizzbuzzer(in <-chan int, out chan<- string) {
      for n := range in {
            out<-fizzbuzz(n)
      }
}
```

```go
func main() {
	request := make(chan int)
	response := make(chan string)

	go fizzbuzzer(request, response)

	for i := 1; i <= 100; i++ {
		request<-i
		fmt.Println(<-response)
	}
}
```

```
variable := expression
```

```
PAR
  channel ! expression
  channel ? variable
```

pipes & filters



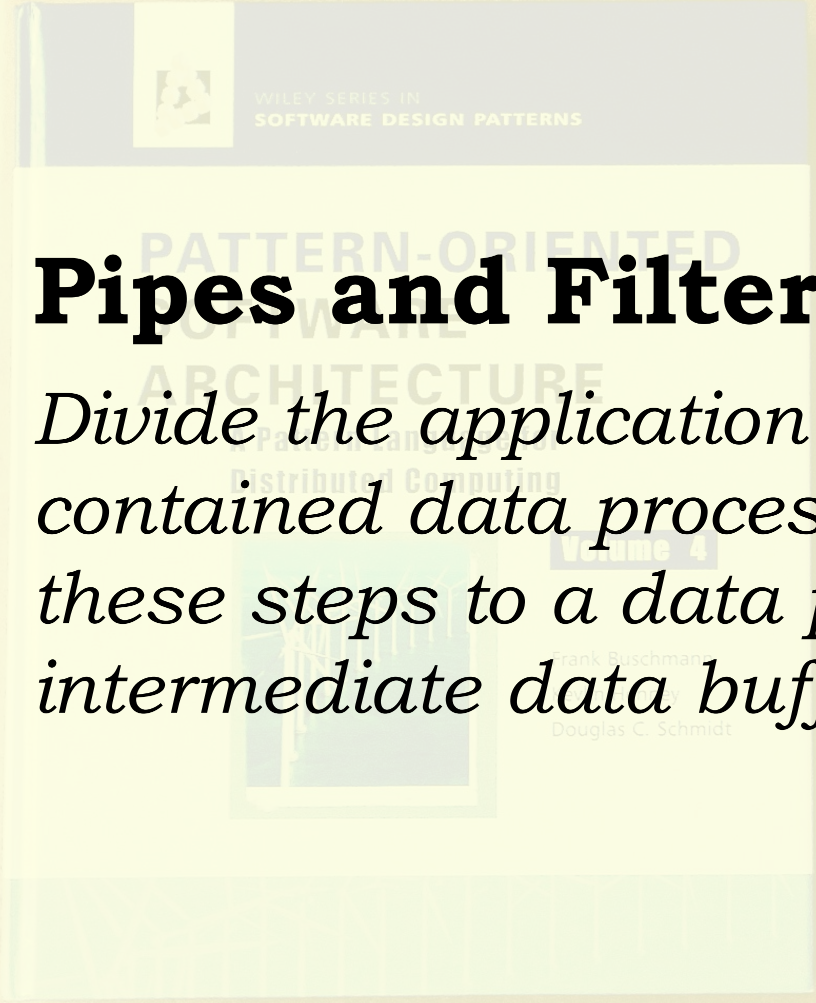
WILEY SERIES IN
SOFTWARE DESIGN PATTERNS

PATTERN-ORIENTED SOFTWARE ARCHITECTURE

A Pattern Language for Distributed Computing

Volume 4

Frank Buschmann
Kevlin Henney
Douglas C. Schmidt



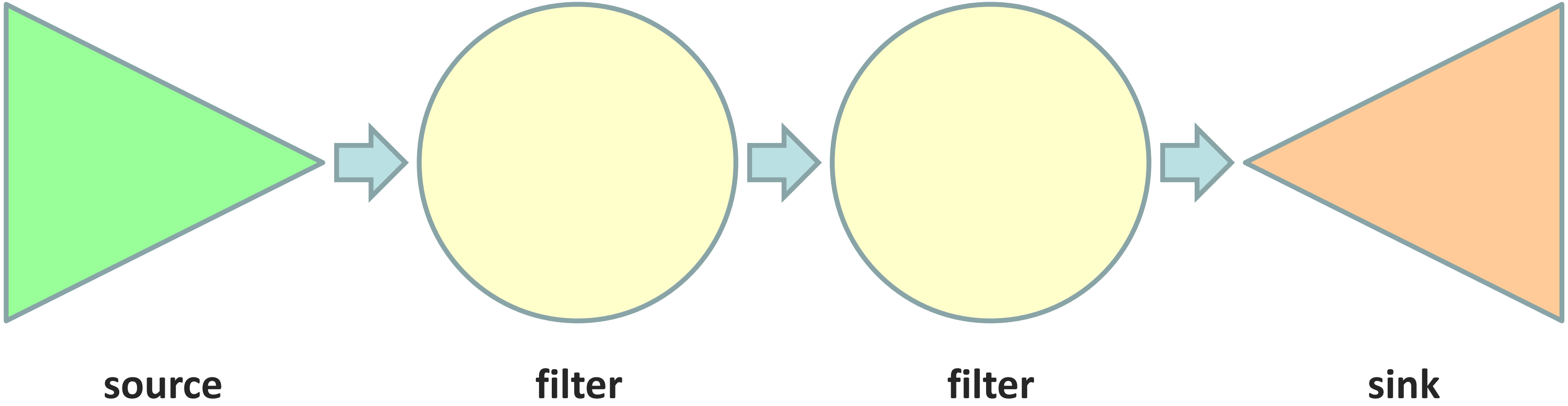
Pipes and Filters

Divide the application's task into several self-contained data processing steps and connect these steps to a data processing pipeline via intermediate data buffers.

Concatenative programming is so called because **it uses function *composition* instead of function *application***—a non-concatenative language is thus called *applicative*.

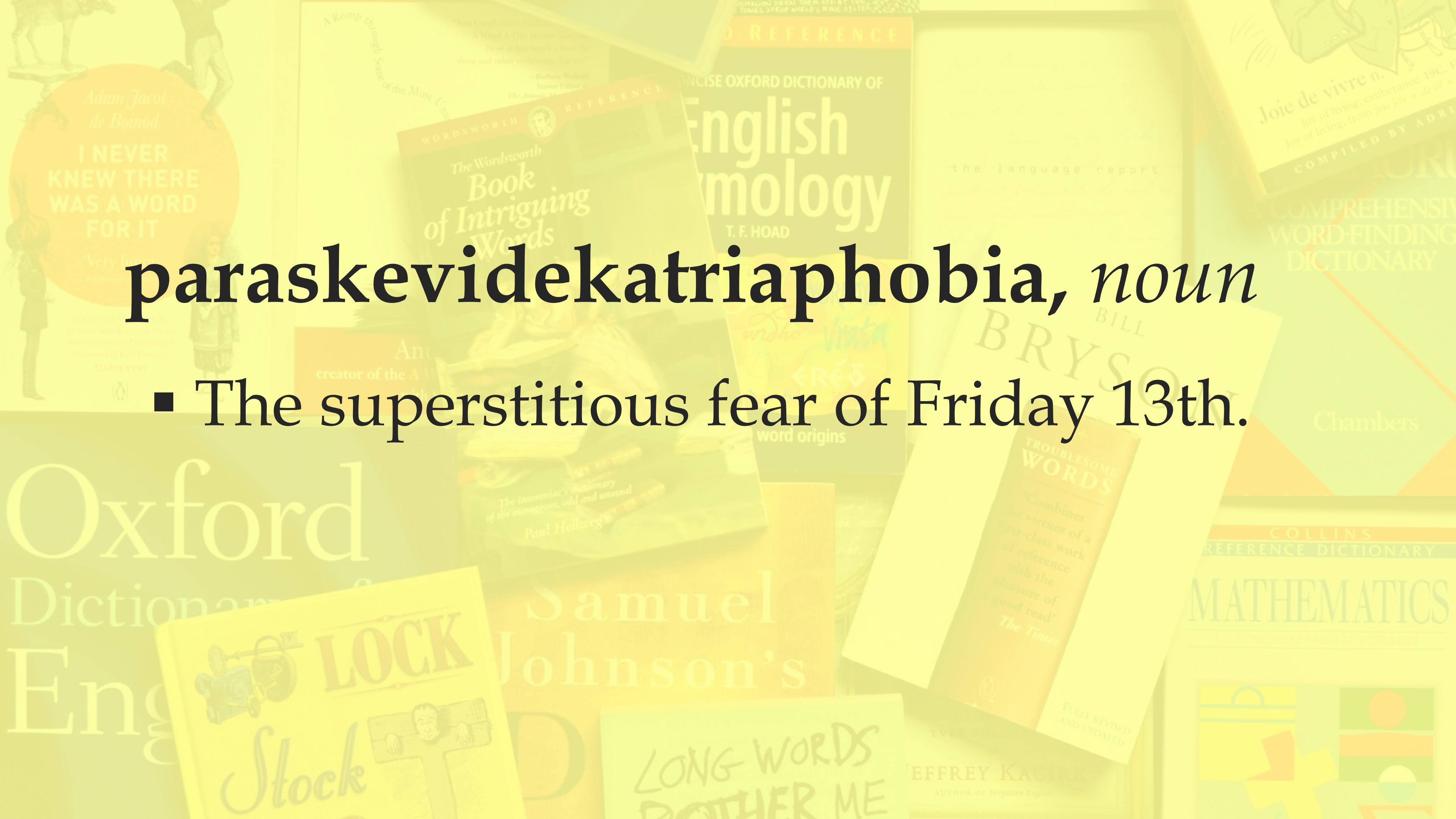
This is the basic reason Unix pipes are so powerful: they form a rudimentary string-based concatenative programming language.

Jon Purdy
http://evincarofautumn.blogspot.com/2012/02/why-concatenative-programming-matters.html


source
filter
filter
sink

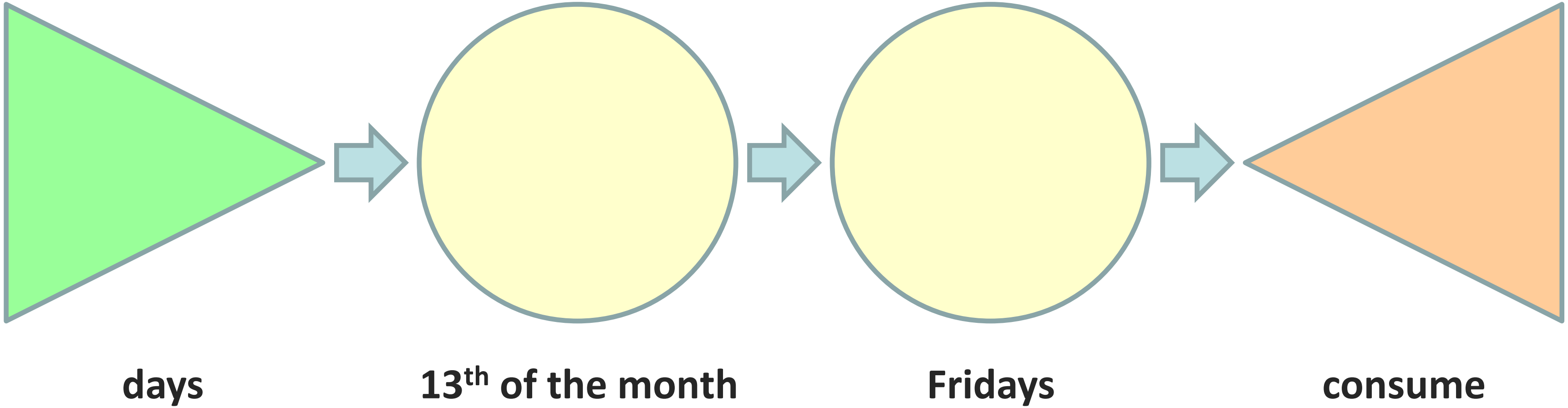
Adam Jacot de Boinod
I NEVER KNEW THERE WAS A WORD FOR IT
'Very funny' Independent on Sunday
STEPHEN FRY
A Romp through Some of the Most Unus
creator of the A W
WORDSWORTH REFERENCE
The Wordsworth Book of Intriguing Words
The insomniac's dictionary of the outrageous, odd and unusual
Paul Hellweg
REFERENCE
CONCISE OXFORD DICTIONARY OF
English Etymology
T. F. HOAD
word origins
the language report
Joie de vivre n.
COMPILED BY ADRI
A COMPREHENSIVE WORD-FINDING DICTIONARY
Chambers
BILL BRYSON
TROUBLESOME WORDS
'Combines the virtues of a first-class work of reference with the pleasure of a good read' The Times
FULLY REVISED AND UPDATED
Oxford Dictionary of English
LOCK Stock
Samuel Johnson's
LONG WORDS BOTHER ME
JEFFREY KACIRK
COLLINS REFERENCE DICTIONARY
MATHEMATICS
E.J. BOROWSKI AND J.M. BORWEIN

f / WordFriday



paraskevidekatriaphobia, *noun*

- The superstitious fear of Friday 13th.

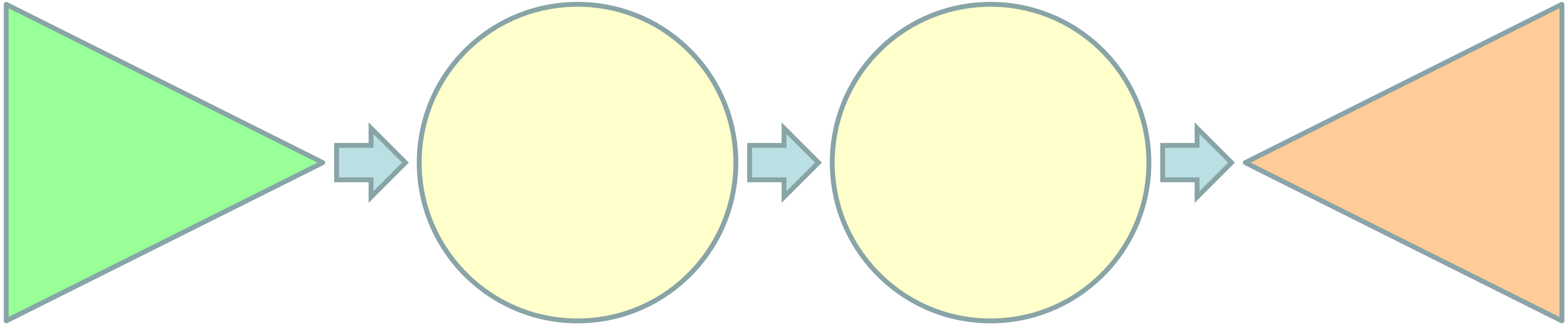


days

13th of the month

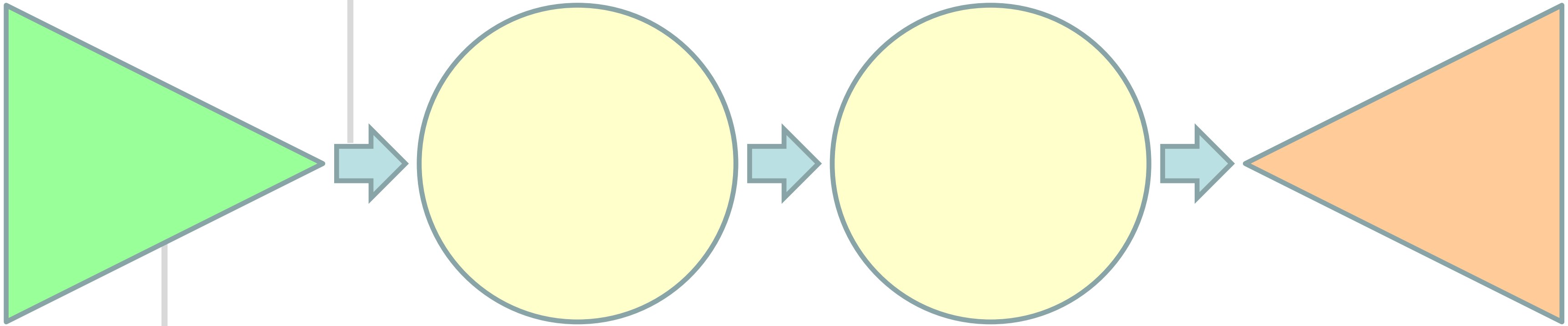
Fridays

consume



```
1..$max | %{$start.AddDays($_)} | ?{$_.Day -eq 13} | ?{$_.DayOfWeek -eq [DayOfWeek]::Friday}
```

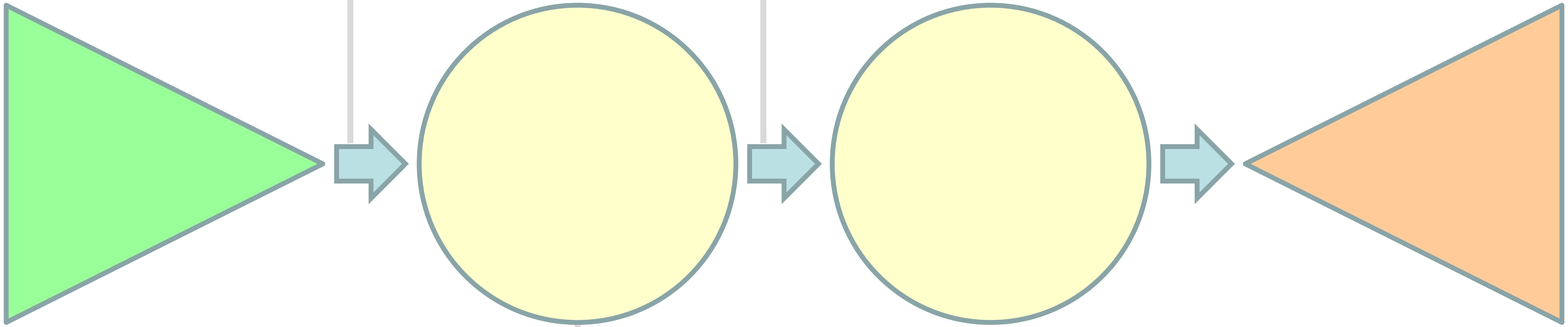
```
channel<std::tm> all_days;
```



```
void days_from(std::tm start, channel<std::tm> & days)
{
    const auto day = 24 * 60 * 60;
    for (auto seconds = std::mktime(&start);;)
    {
        seconds += day;
        days << *std::localtime(&seconds);
    }
}
```

```
channel<std::tm> all_days;
```

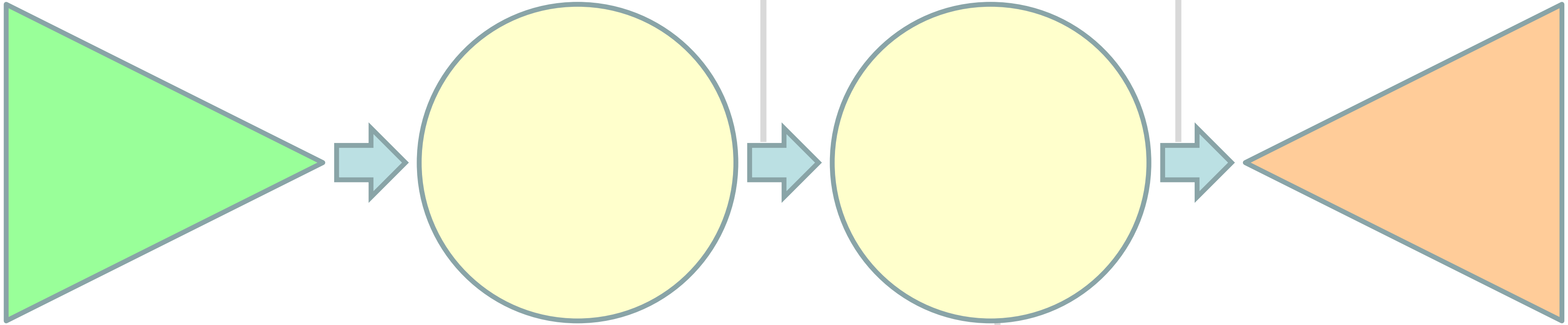
```
channel<std::tm> only_13ths;
```



```
void select_13th(channel<std::tm> & in, channel<std::tm> & out)
{
    for (std::tm day;;)
    {
        in >> day;
        if (day.tm_mday == 13)
            out << day;
    }
}
```

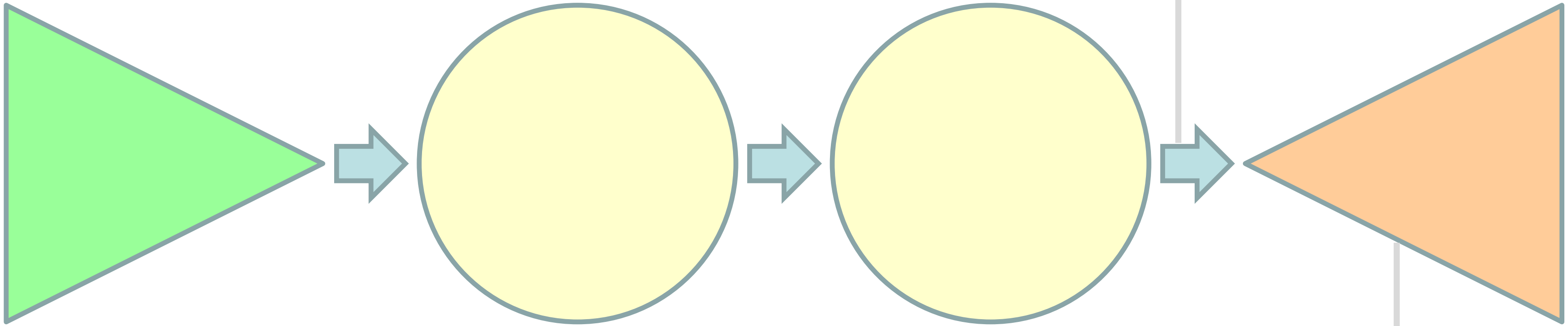
```
channel<std::tm> only_13ths;
```

```
channel<std::tm> only_friday_13ths;
```



```
void select_friday(channel<std::tm> & in, channel<std::tm> & out)
{
    for (std::tm day;;)
    {
        in >> day;
        if (day.tm_wday == 5)
            out << day;
    }
}
```

```
channel<std::tm> only_friday_13ths;
```



```
void display(channel<std::tm> & results)
{
    for (std::tm day;;)
    {
        results >> day;
        ...
    }
}
```

Simple filters that can be arbitrarily chained are more easily re-used, and more robust, than almost any other kind of code.

Brandon Rhodes
http://rhodesmill.org/brandon/slides/2012-11-pyconca/

```go
func Generate(ch chan<- int) {
    for i := 2; ; i++ {
        ch <- i
    }
}
```

http://mrg.doc.ic.ac.uk/publications/fencing-off-go-liveness-and-safety-for-channel-based-programming/

```
func Generate(ch chan<- int) {
    for i := 2; ; i++ {
}
```

```
func Filter(in <-chan int, out chan<- int, prime int)
{
    for {
        i := <-in
        if i % prime != 0 {
            out <- i
        }
    }
}
```

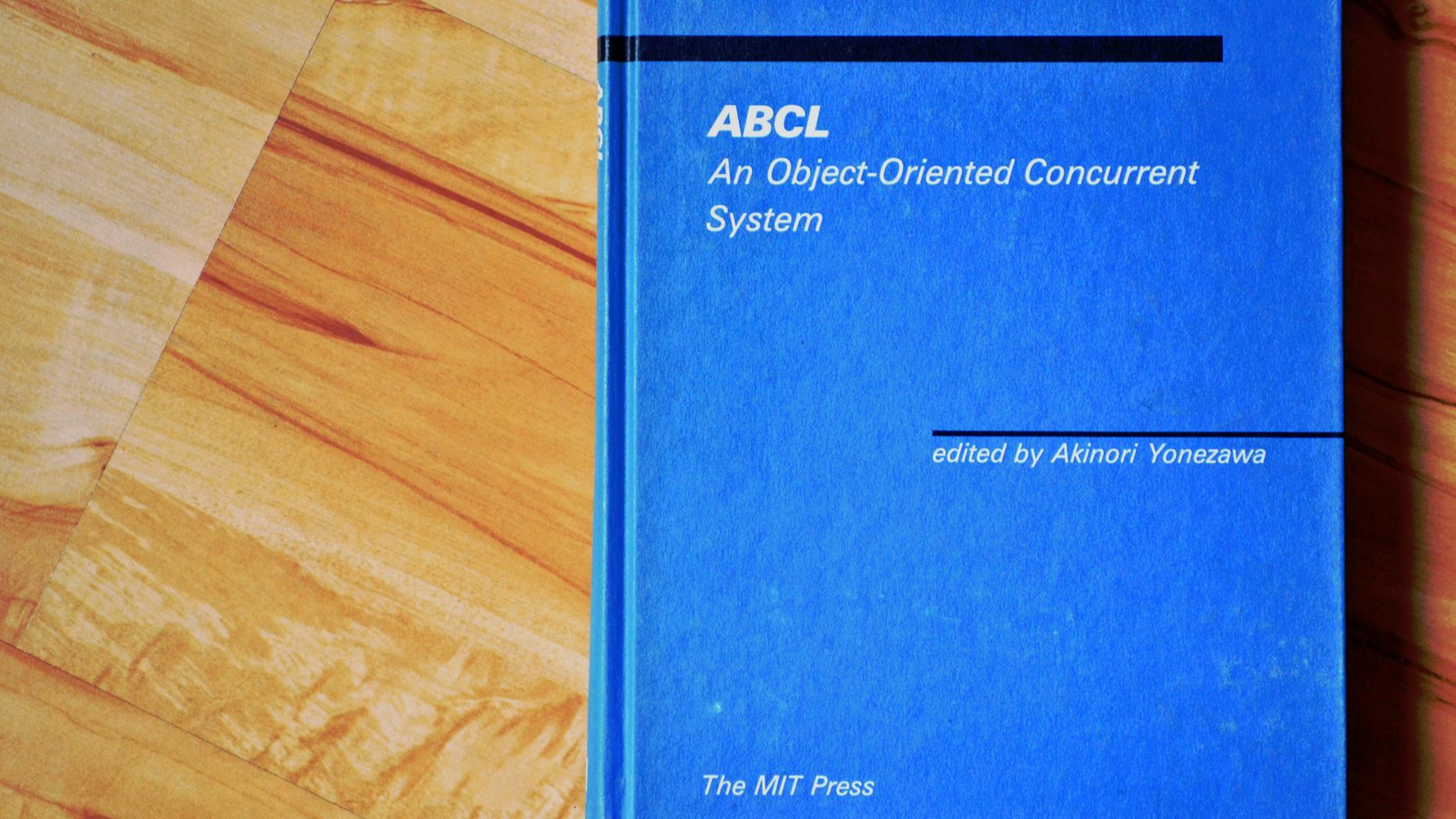
http://mrg.doc.ic.ac.uk/publications/fencing-off-go-liveness-and-safety-for-channel-based-programming/

```go
func Generate(ch chan<- int) {
    for i := 2; ; i++ {

}
```

```go
func Filter(in <-chan int, out chan<- int, prime int)
{

}
```

```go
func main() {
    ch := make(chan int)
    go Generate(ch)
    for i := 0; ; i++ {
        prime := <-ch
        ch1 := make(chan int)
        go Filter(ch, ch1, prime)
        ch = ch1
    }
}
```

http://mrg.doc.ic.ac.uk/publications/fencing-off-go-liveness-and-safety-for-channel-based-programming/



ABCL

An Object-Oriented Concurrent System

edited by Akinori Yonezawa

The MIT Press

Multithreading is just one damn thing after, before, or simultaneous with another.

Andrei Alexandrescu

Actor-based concurrency is just one damn message after another.

monitor objects

```
class phone_book
{
public:
    void update(const std::string & name, const std::string & number);
    void drop(const std::string & name);
    std::optional<std::string> find(const std::string & name) const;
private:
    mutable std::mutex key;
    std::map<std::string, std::string> entries;
};
```

```cpp
void phone_book::update(const std::string & name, const std::string & number)
{
    std::lock_guard<std::mutex> guard(key);
    entries[name] = number;
}

void phone_book::drop(const std::string & name)
{
    std::lock_guard<std::mutex> guard(key);
    entries.erase(name);
}

std::optional<std::string> phone_book::find(const std::string & name) const
{
    std::lock_guard<std::mutex> guard(key);
    auto found = entries.find(name);

    if (found == entries.end())
        return {};
    else
        return found->second;
}
```

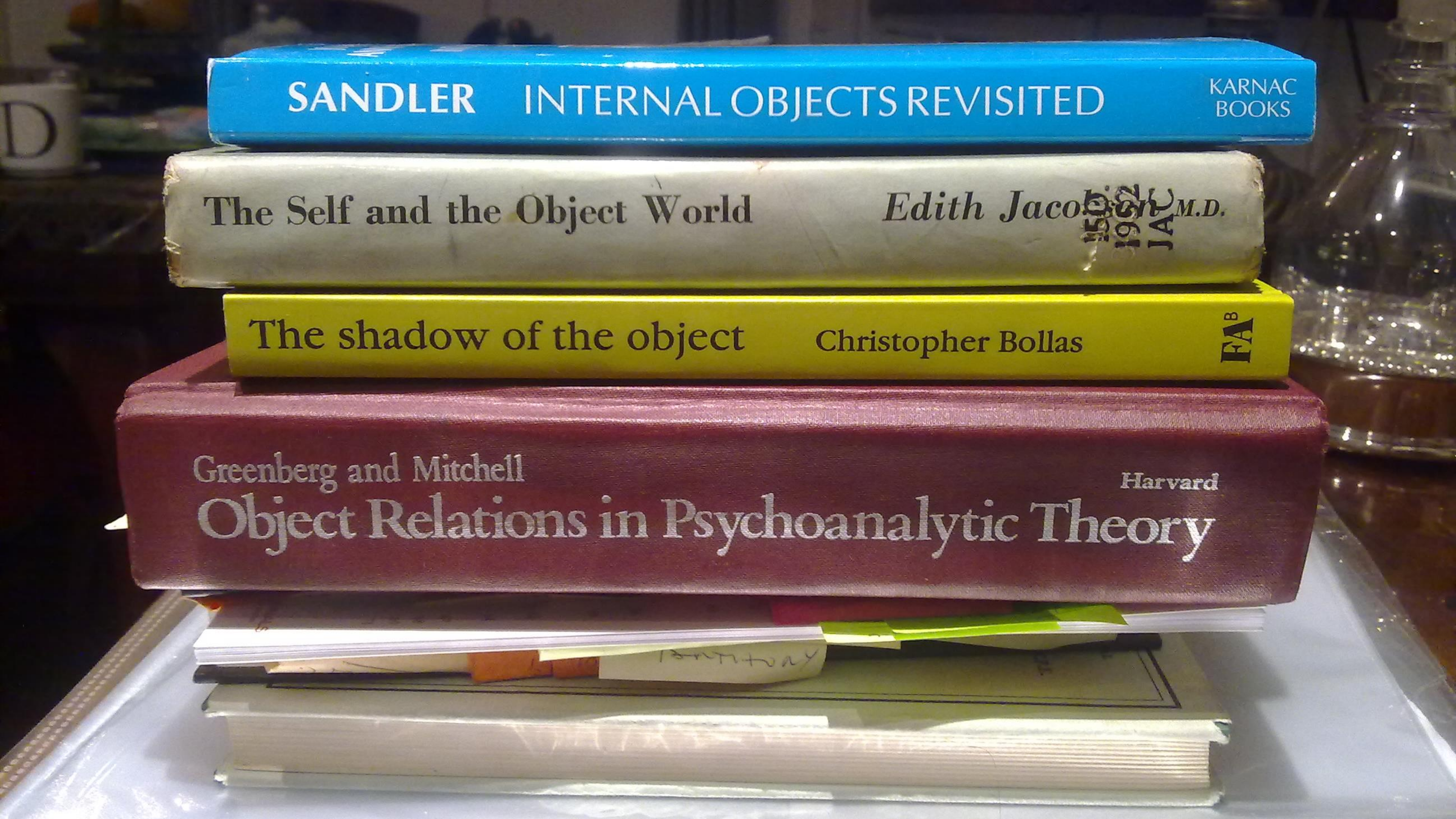
```
phone_book directory;

                         auto unfound = directory.find("Thomas Anderson");

             directory.update("Thomas Anderson", "1");

                         auto found = directory.find("Thomas Anderson");
                         unfound = directory.find("Neo");

             directory.update("Trinity", "3");
             directory.update("Morpheus", "42");
             directory.drop("Thomas Anderson");
             directory.update("Neo", "1");

                         unfound = directory.find("Thomas Anderson");
                         found = directory.find("Neo");
```

active objects

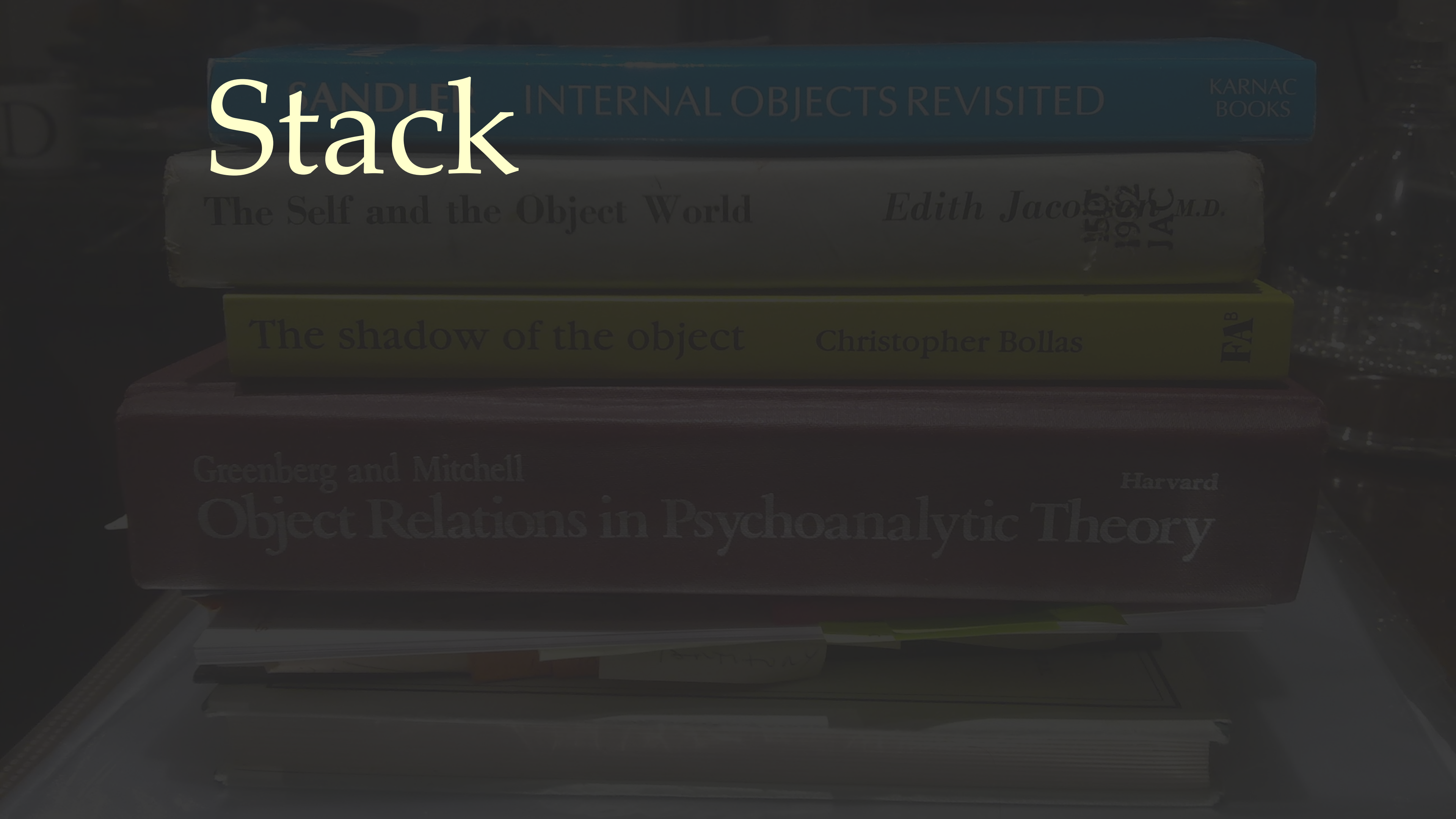
```
class phone_book
{
public:
    void operator()();
    void update(const std::string & name, const std::string & number);
    void drop(const std::string & name);
    std::future<std::optional<std::string>>
        find(const std::string & name) const;
private:
    std::thread self;
    std::queue<std::function<void()>> calls;
    std::map<std::string, std::string> entries;
};
```

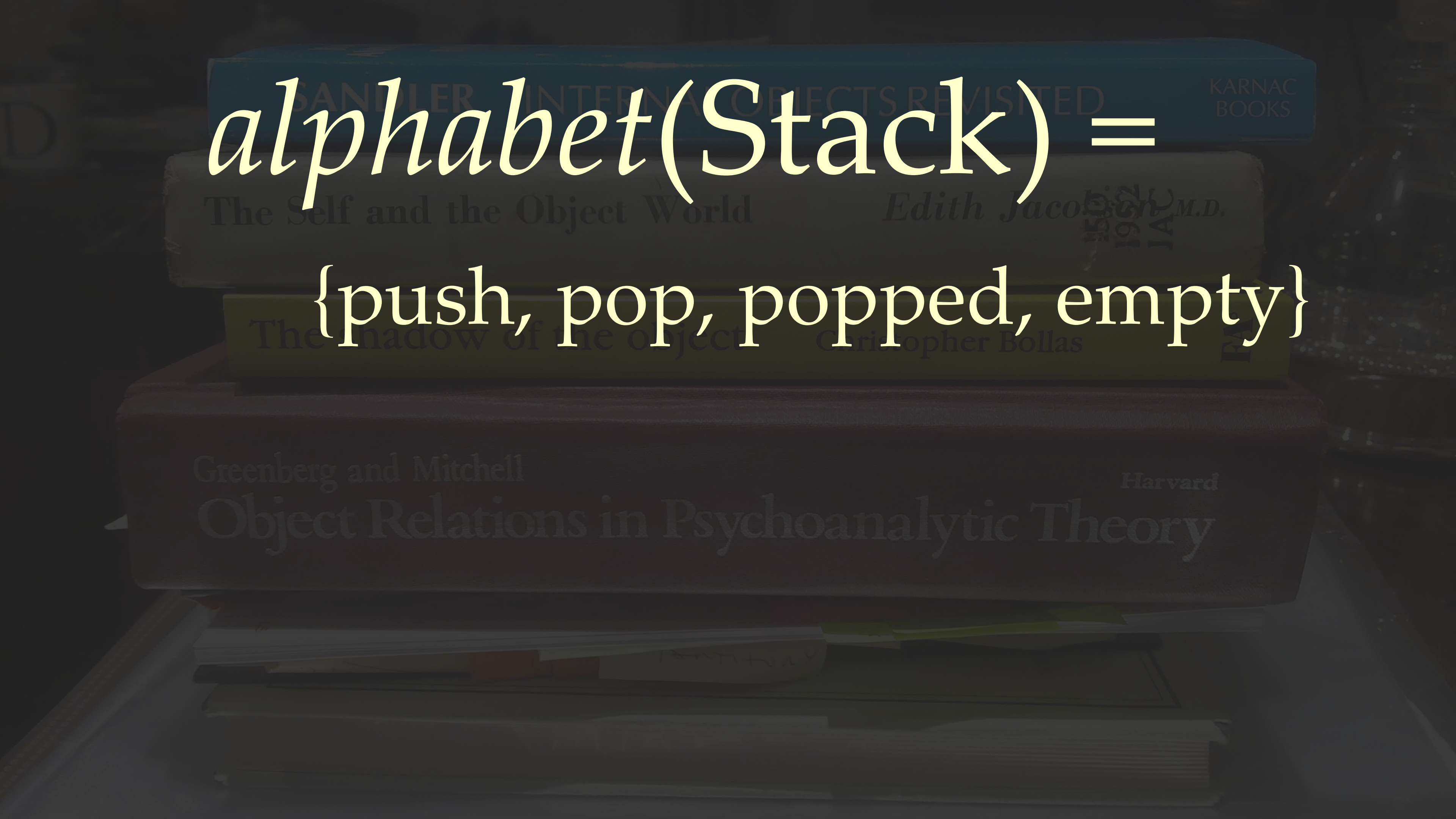
```
phone_book directory;
directory();

                         auto unfound = directory.find("Thomas Anderson").get();

             directory.update("Thomas Anderson", "1");

                         auto found = directory.find("Thomas Anderson").get();
                         unfound = directory.find("Neo").get();

             directory.update("Trinity", "3");
             directory.update("Morpheus", "42");
             directory.drop("Thomas Anderson");
             directory.update("Neo", "1");

                         unfound = directory.find("Thomas Anderson").get();
                         found = directory.find("Neo").get();
```

actors

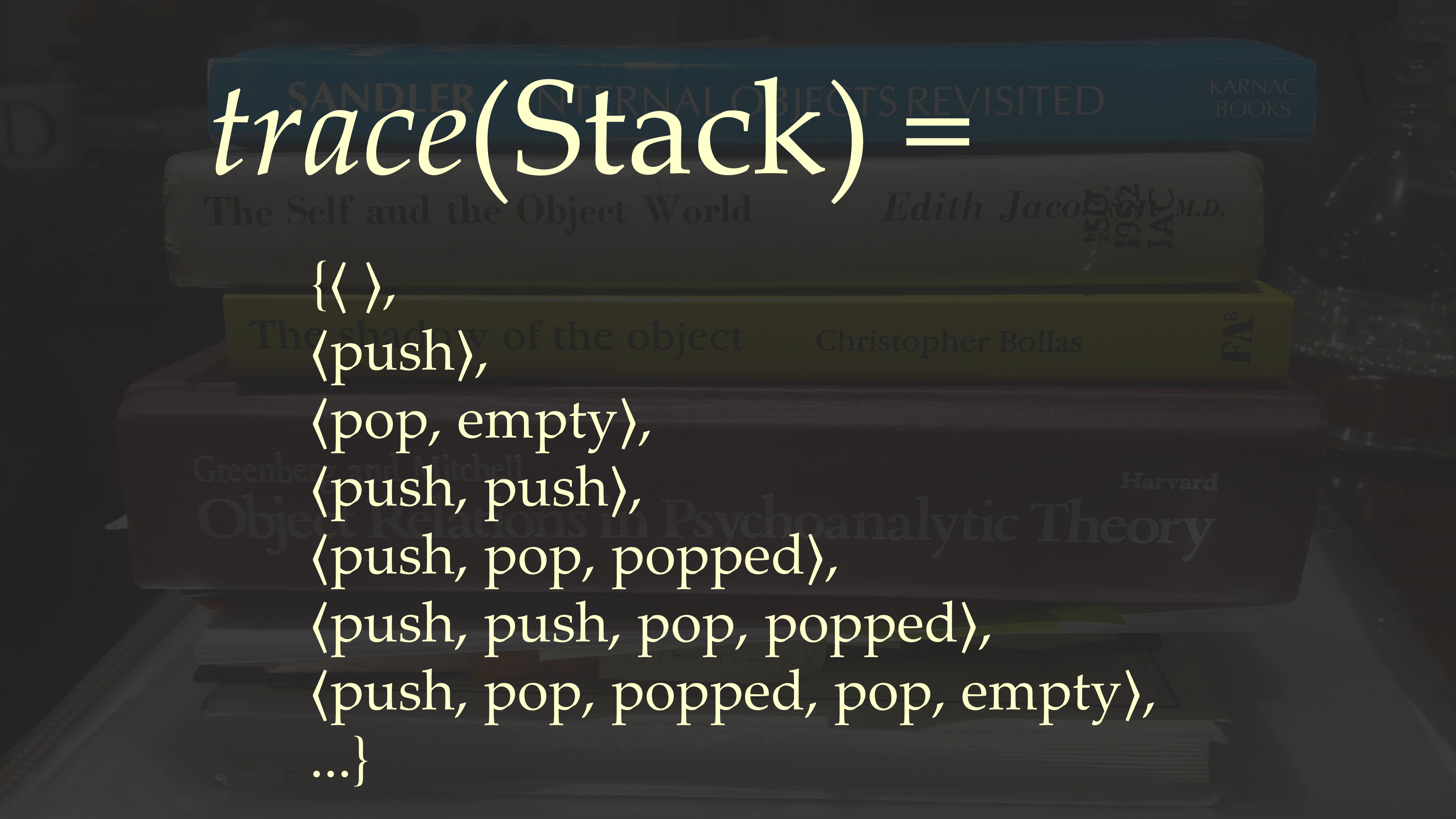

SANDLER INTERNAL OBJECTS REVISITED
KARNAC BOOKS
The Self and the Object World
Edith Jacobson M.D.
150 1952 JAC
The shadow of the object
Christopher Bollas
FAB
Greenberg and Mitchell
Object Relations in Psychoanalytic Theory
Harvard

Stack


SANDLER INTERNAL OBJECTS REVISITED
KARNAC BOOKS
The Self and the Object World
Edith Jacobson M.D.
1962
IAC
The shadow of the object
Christopher Bollas
FA
Greenberg and Mitchell
Object Relations in Psychoanalytic Theory
Harvard

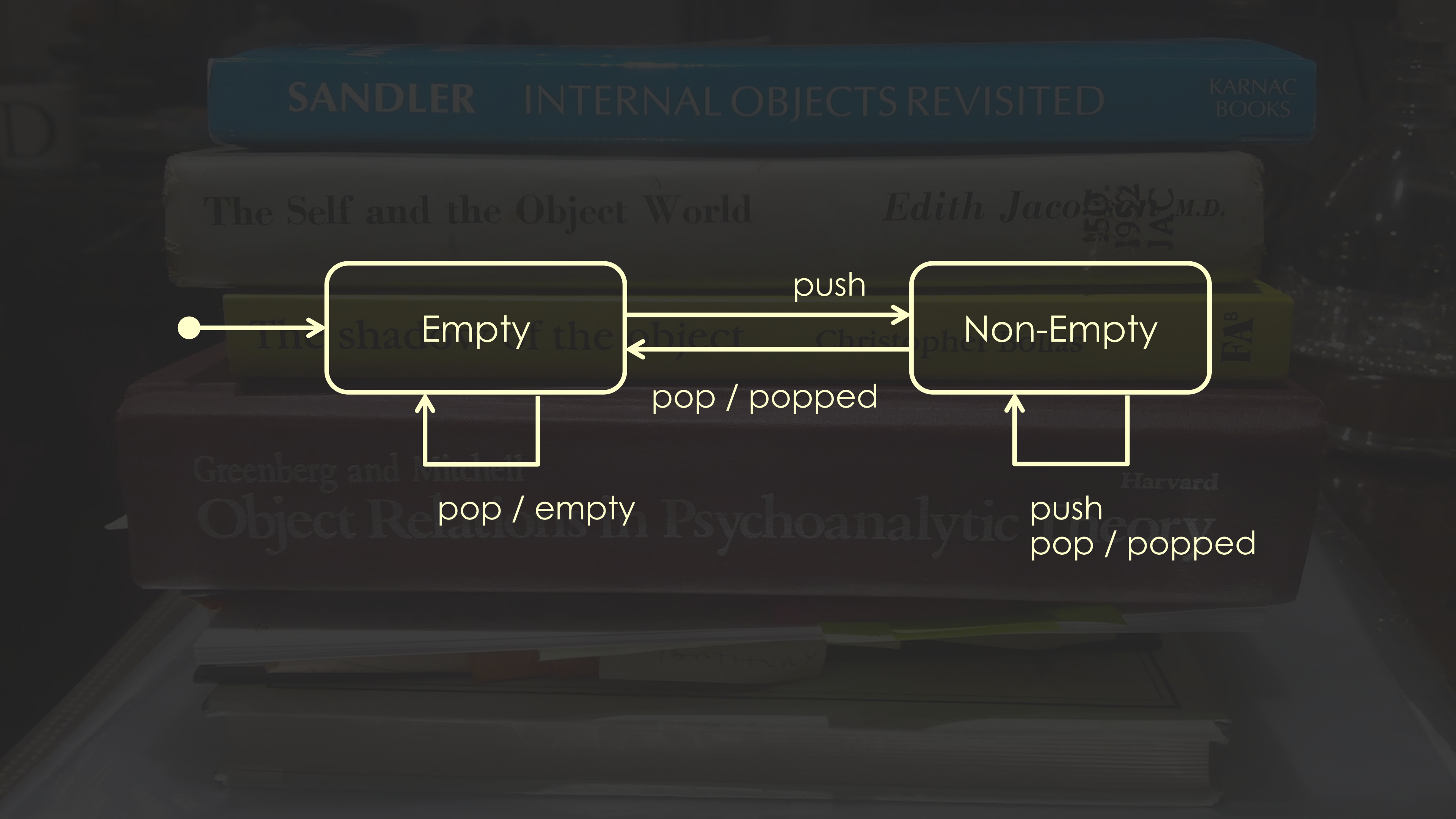

SANDLER INTERNAL OBJECTS REVISITED
KARNAC BOOKS
The Self and the Object World
Edith Jacobson M.D.
The Shadow of the Object
Christopher Bollas
Greenberg and Mitchell
Object Relations in Psychoanalytic Theory
Harvard

$$\mathit{alphabet}(\mathrm{Stack}) = \{\mathrm{push}, \mathrm{pop}, \mathrm{popped}, \mathrm{empty}\}$$



trace(Stack) =

{⟨ ⟩,
⟨push⟩,
⟨pop, empty⟩,
⟨push, push⟩,
⟨push, pop, popped⟩,
⟨push, push, pop, popped⟩,
⟨push, pop, popped, pop, empty⟩,
…}



Empty

push

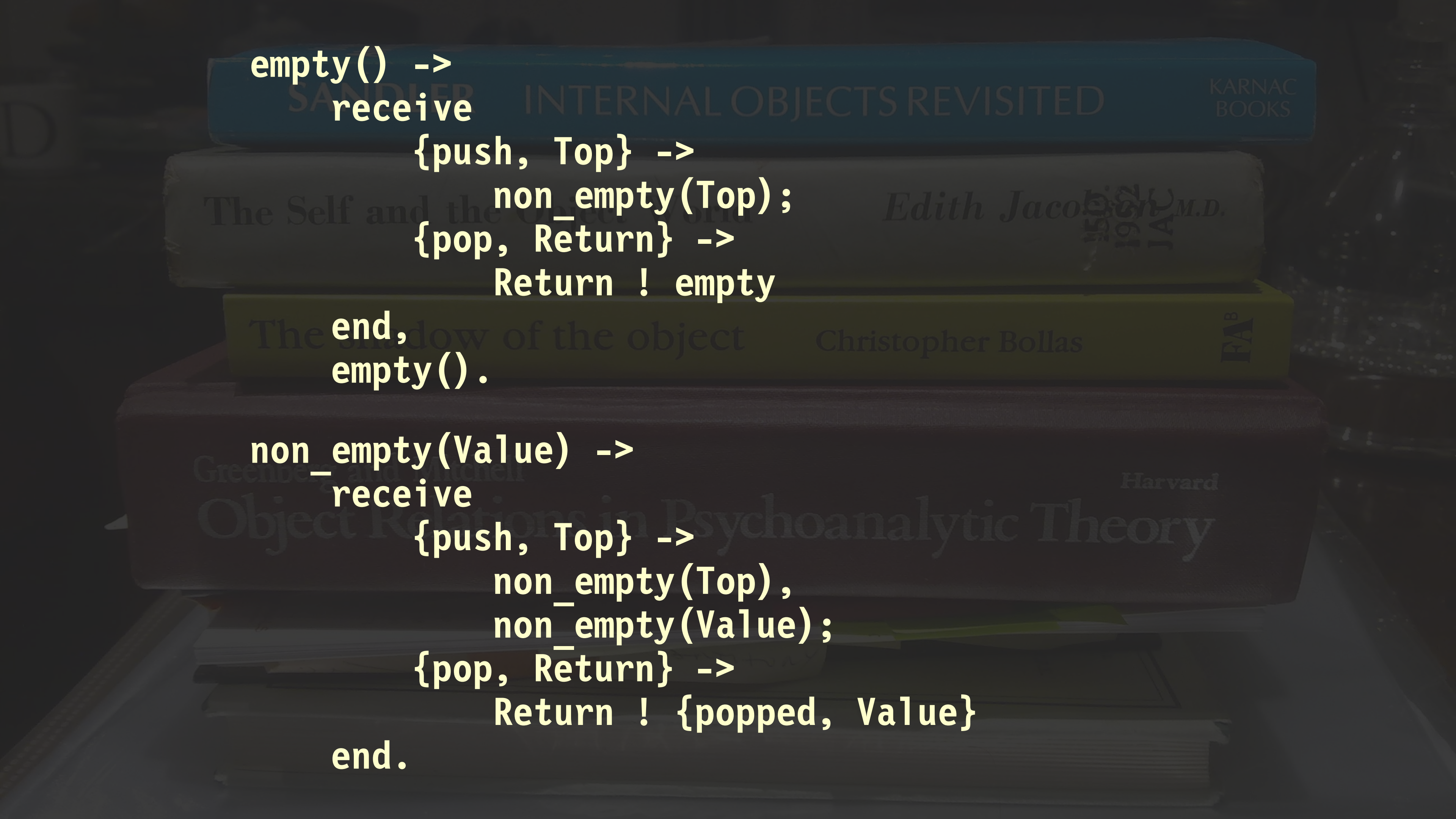
Non-Empty

pop / popped

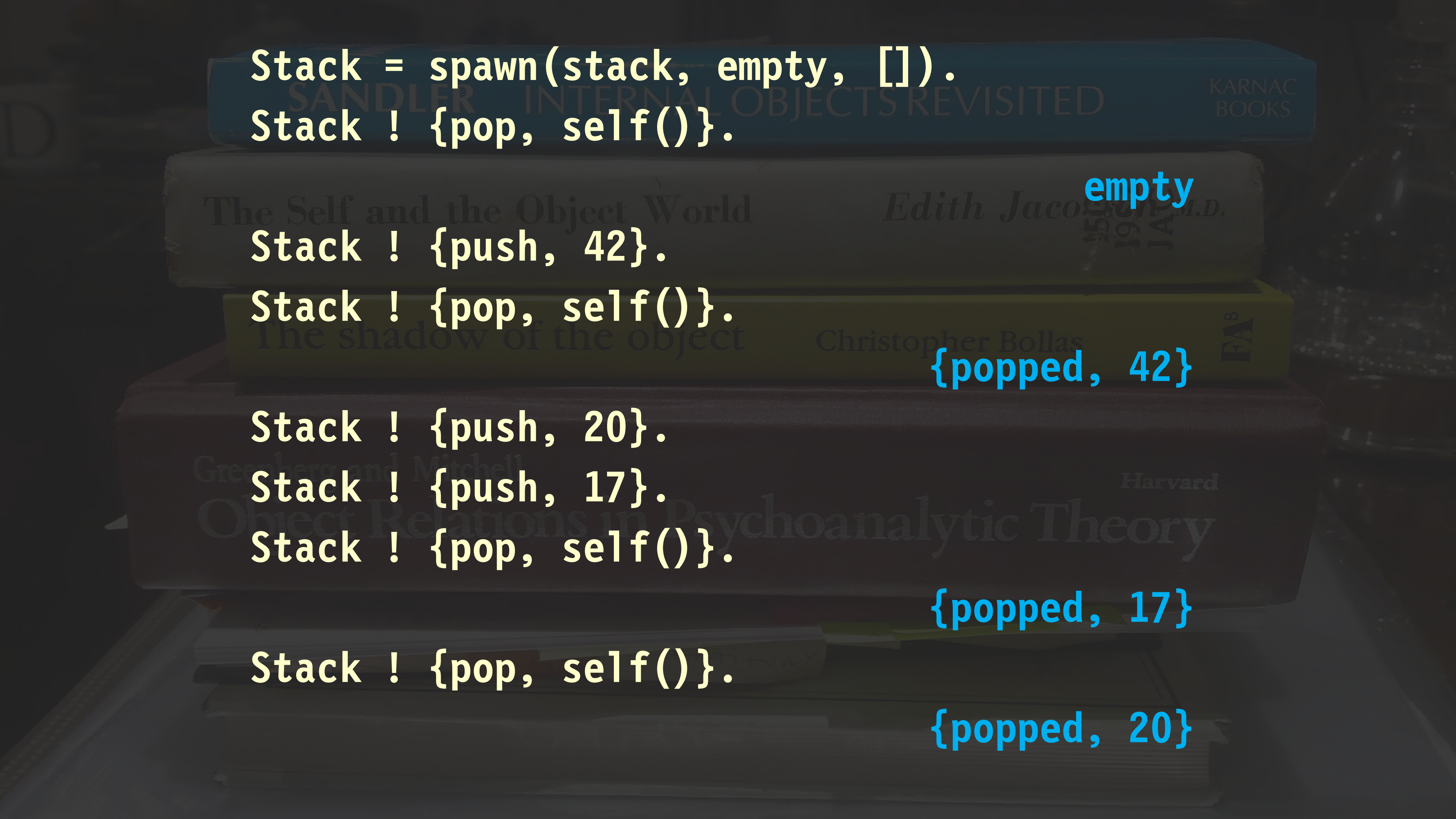
pop / empty

push
pop / popped

SANDLER INTERNAL OBJECTS REVISITED
KARNAC BOOKS
The Self and the Object World
Edith Jacobson M.D.
The shadow of the object
Christopher Bollas
FA
Greenberg and Mitchell
Object Relations in Psychoanalytic Theory
Harvard



```
empty() ->
    receive
        {push, Top} ->
            non_empty(Top);
        {pop, Return} ->
            Return ! empty
    end,
    empty().

non_empty(Value) ->
    receive
        {push, Top} ->
            non_empty(Top),
            non_empty(Value);
        {pop, Return} ->
            Return ! {popped, Value}
    end.
```



```
Stack = spawn(stack, empty, []).
Stack ! {pop, self()}.

                                           empty

Stack ! {push, 42}.
Stack ! {pop, self()}.

                                    {popped, 42}

Stack ! {push, 20}.
Stack ! {push, 17}.
Stack ! {pop, self()}.

                                    {popped, 17}

Stack ! {pop, self()}.

                                    {popped, 20}
```

```
void phone_book(queue<std::any> &);
struct entry
{
    std::string name, number;
};
struct no_entry
{
    std::string name;
};
struct find
{
    std::string name;
    queue<std::any> & there;
};
```

```
void phone_book(queue<std::any> & here)
{
    std::map<std::string, std::string> entries;
    for (std::any request;;)
    {
        here >> request;
        if (auto update = std::any_cast<entry>(&request))
            entries[update->name] = update->number;
        else if (auto drop = std::any_cast<no_entry>(&request))
            entries.erase(drop->name);
        else if (auto lookup = std::any_cast<find>(&request))
        {
            auto found = entries.find(lookup->name);
            if (found == entries.end())
                lookup->there << no_entry { lookup->name };
            else
                lookup->there << entry { found->first, found->second };
        }
    }
}
```

```
void phone_book(queue<std::any> & here)
{
    std::map<std::string, std::string> entries;
    for (std::any request;;)
    {
        here >> request;
        request
            ||  [&](entry & update) { entries[update->name] = update->number; }
            ||  [&](no_entry & drop) { entries.erase(drop->name); }
            ||  [&](find & lookup)
                {
                    auto found = entries.find(lookup->name);
                    if (found == entries.end())
                        lookup->there << no_entry { lookup->name };
                    else
                        lookup->there << entry { found->first, found->second };
                };
        }
    }
}
```

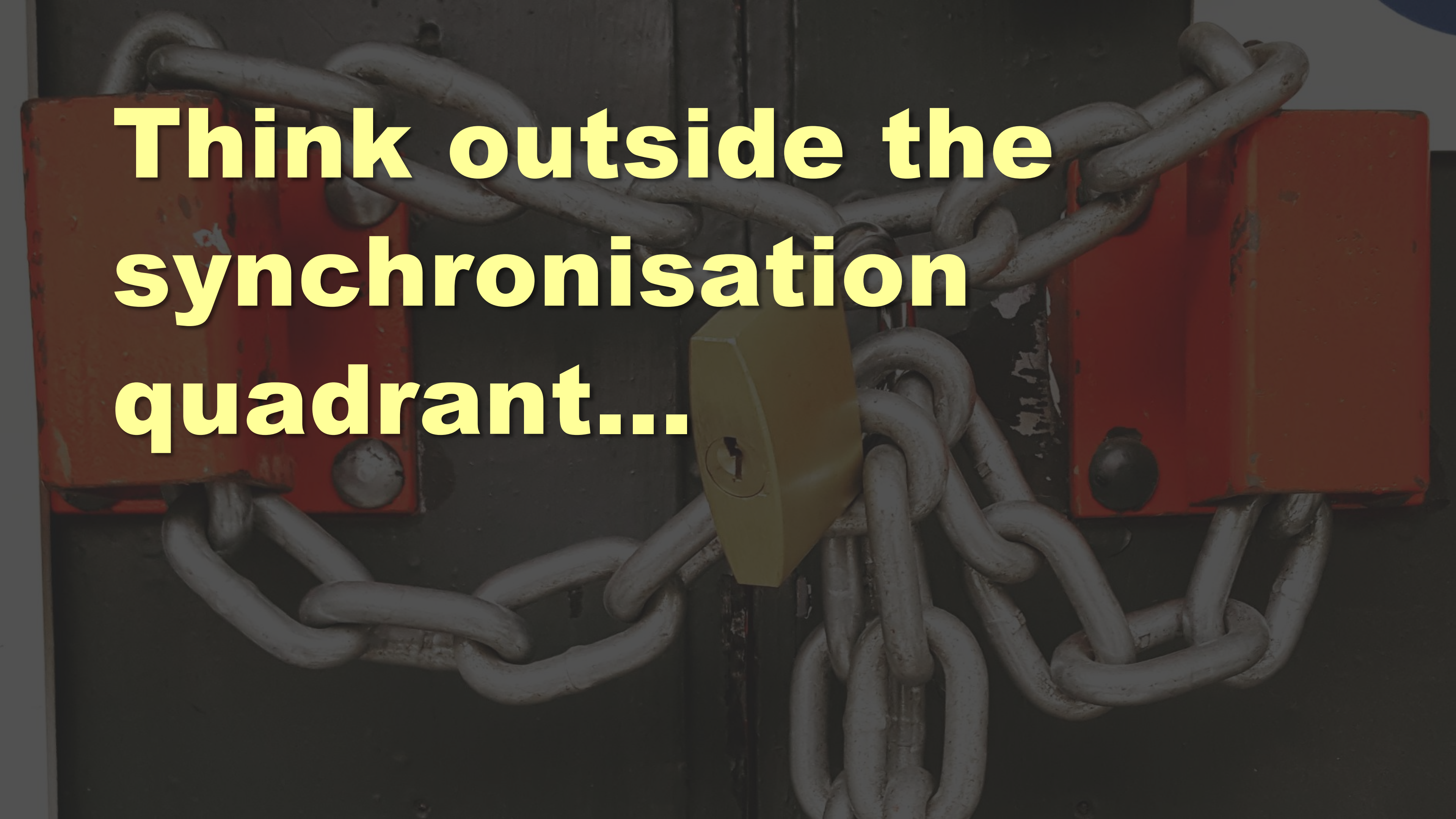
```
queue<std::any> directory;
std::thread(phone_book, std::ref(directory)).detach();

                         queue<std::any> here;
                         directory << find { "Thomas Anderson", here };
                         std::any unfound;
                         here >> unfound; // no_entry { "Thomas Anderson" }

             directory << entry { "Thomas Anderson", "1" };

                         directory << find { "Thomas Anderson", here };
                         std::any found;
                         here >> found; // entry { "Thomas Anderson", 1 }

             directory << entry { "Trinity", "3" };
             directory << entry { "Morpheus", "42" };
             directory << no_entry { "Thomas Anderson" };
             directory << entry { "Neo", "1" };

                         directory << find { "Neo", here };
                         here >> found; // entry { "Neo", 1 }
```

> Programming in a functional style makes the state presented to your code explicit, which makes it much easier to reason about, and, in a completely pure system, makes thread race conditions impossible.

John Carmack

http://www.gamasutra.com/view/news/169296/Indepth_Functional_programming_in_C.php

Think outside the synchronisation quadrant...



All computers wait at the same speed.