# Threading is not a model

**Joe Gregorio**
Developer Relations, Google Wave

# Scope

My Opinion

# Goal

I want to annoy you.

## The path

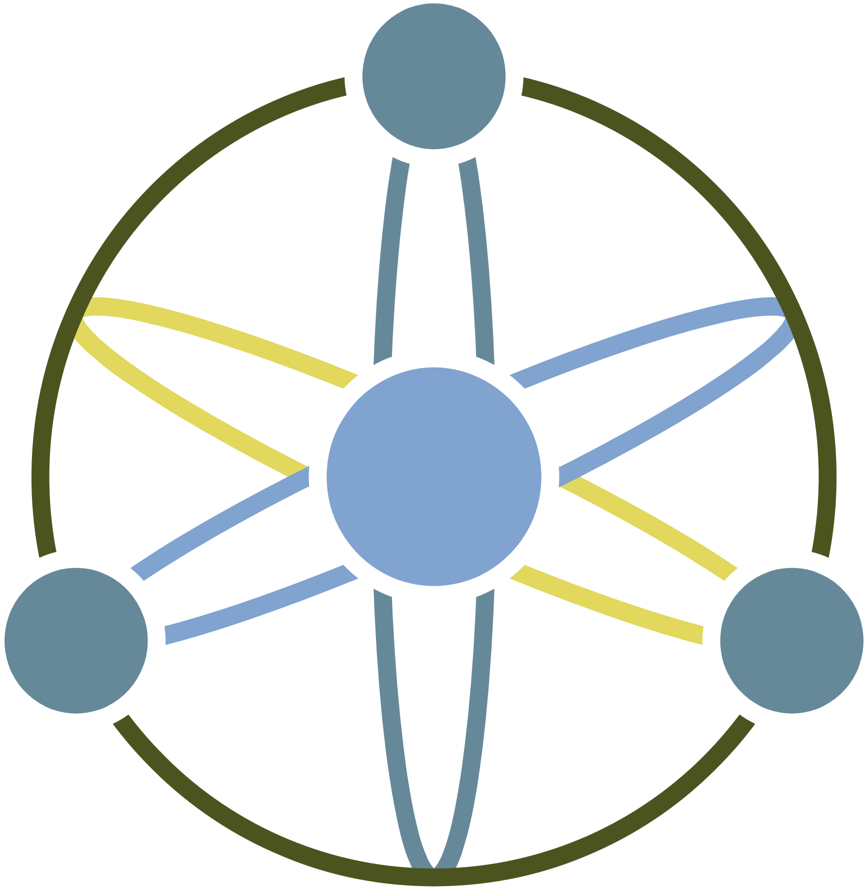
A short story, a book, design patterns, and Djikstra

# "The Short Happy Life of the Brown Oxford"

## Philip K. Dick

The short story

# The Principle of Sufficient Irritation in action

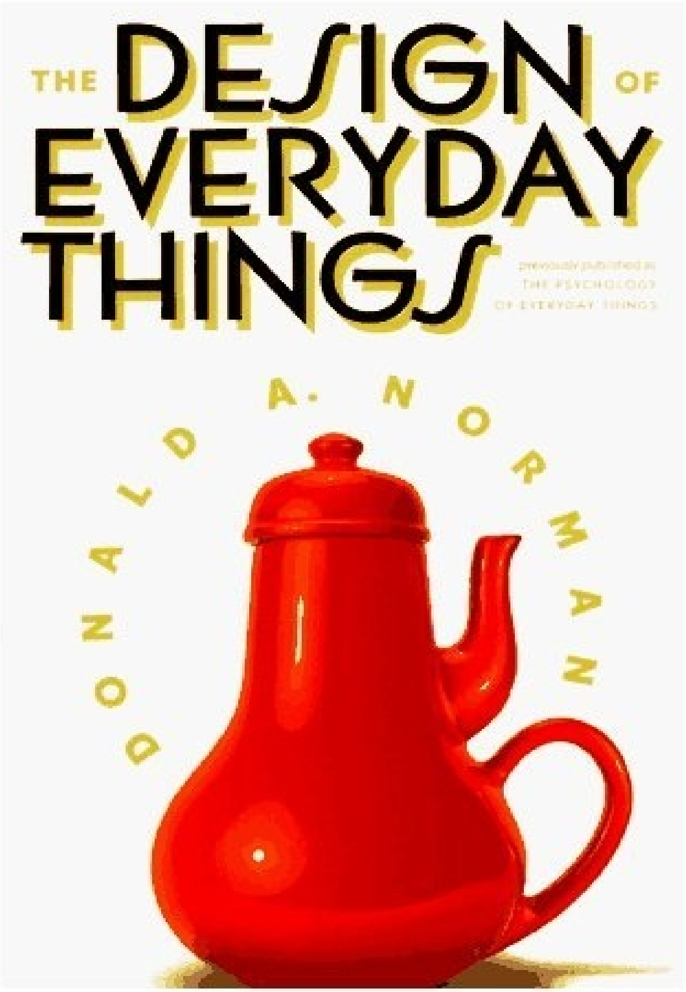


Determining the radioactive irritant is left as an exercise for the reader.

# The path

A short story, a book, design patterns, and Djikstra

# The Design of Everyday Things


THE DESIGN OF EVERYDAY THINGS
previously published as THE PSYCHOLOGY OF EVERYDAY THINGS
DONALD A. NORMAN

The book

# The path

A short story, a book, design patterns, and Djikstra

## Setting the record straight

# Let's talk about Design Patterns

*I **did not** say that patterns are bad.*
*I **did** say that using them may be a sign of weakness in a language.*

# A Blog Post

Python isn't Java without the compile

*Design Patterns in Dynamic Programming – Peter Norvig*
*Beyond Java – Bruce Tate*

## Language

Not talking just about Python

# Language

## Aren't patterns good?

*Yes, but also a sign of weakness*

## There is a lack of patterns in Python

1. Define 'lack of patterns'
2. Demonstrate that lack
3. Explain why

# Hard numbers

comp.lang.python

*100,000+ messages*

## Hard numbers

“factory method pattern” - 0

“abstract-factory pattern” - 0

“flyweight pattern” - 3

“state pattern” - 10

“strategy pattern” - 25

“flyweight” - 36

“visitor pattern” - 60

## For your comparison

# “dark matter” - 2

# For your comparison

“dark matter” - 2

“the pope” - 16

# For your comparison

“dark matter” - 2

“the pope” - 16

“sausage” - 66

*Presuming there is no overlap among these messages*

## There is a lack of patterns in Python

1. ~~Define 'lack of patterns'~~
2. ~~Demonstrate that lack~~
3. Explain why

# Explain Why

The patterns are built in.

*No one talks about the 'structured programming' pattern or the 'object-oriented' pattern any more.*

# Strategy Pattern on comp.lang.python

```
class Bisection (FindMinima):
   def algorithm(self,line):
      return (5.5,6.6)

class ConjugateGradient (FindMinima):
   def algorithm(self,line):
      return (3.3,4.4)

class MinimaSolver: # context class
   strategy=''
   def __init__ (self,strategy):
      self.strategy=strategy
   def minima(self,line):
      return self.strategy.algorithm(line)
   def changeAlgorithm(self,newAlgorithm):
      self.strategy = newAlgorithm

solver=MinimaSolver(ConjugateGradient())
print solver.minima((5.5,5.5))
solver.changeAlgorithm(Bisection())
print solver.minima((5.5,5.5))
```

# Strategy Pattern

> “When most of your code does nothing in a pompous way that is a sure sign that you are heading in the wrong direction. Here's a translation into python”
>
> - Peter Otten

# Strategy Pattern on comp.lang.python

```
def bisection(line):
   return 5.5, 6.6

def conjugate_gradient(line):
   return 3.3, 4.4

solver = conjugate_gradient
print solver((5.5,5.5))
solver = bisection
print solver((5.5,5.5))
```

# Proof by Wikipedia

"This pattern is invisible in languages with first-class functions."

http://en.wikipedia.org/wiki/Strategy_pattern

*What other language features are there, and what patterns do they make invisible?*

# Catalog of Language Features

First-class functions

Meta-programming

Iterators

Closures

# Proof by Wikipedia

In object-oriented programming, the Iterator pattern is a design pattern in which iterators are used to access the elements of an aggregate object sequentially without exposing its underlying representation.

http://en.wikipedia.org/wiki/Iterator_pattern

*The definition of low-hanging fruit.*

# Iterators

```
for element in [1, 2, 3]:
   print element

for element in (1, 2, 3):
   print element

for key in {'one':1, 'two':2}:
   print key

for char in "123":
   print char

for line in open("myfile.txt"):
   print line
```

## There is a lack of patterns in Python

1. ~~Define 'lack of patterns'~~
2. ~~Demonstrate that lack~~
3. ~~Explain why~~

# The path

A short story, a book, design patterns, and Djikstra

## Structured Programming

# "Go to statement considered harmful”

# Edsger W. Dijkstra,1968

*Letter to the editor, Communications of the ACM , Volume 11, Issue 3 (March 1968)*

## Structured Programming

We are talking about Routines!

(or procedures, or functions, or

methods) being controversial.

*Along with 'if', 'while', and 'switch' statements*

## The controversy went on for a while

"GOTO Considered Harmful"
Considered Harmful

Frank Rubin, 1987

*Communications of the ACM, Vol. 30, No. 3. (March 1987), pp. 195-196.*

# With Structured Programming

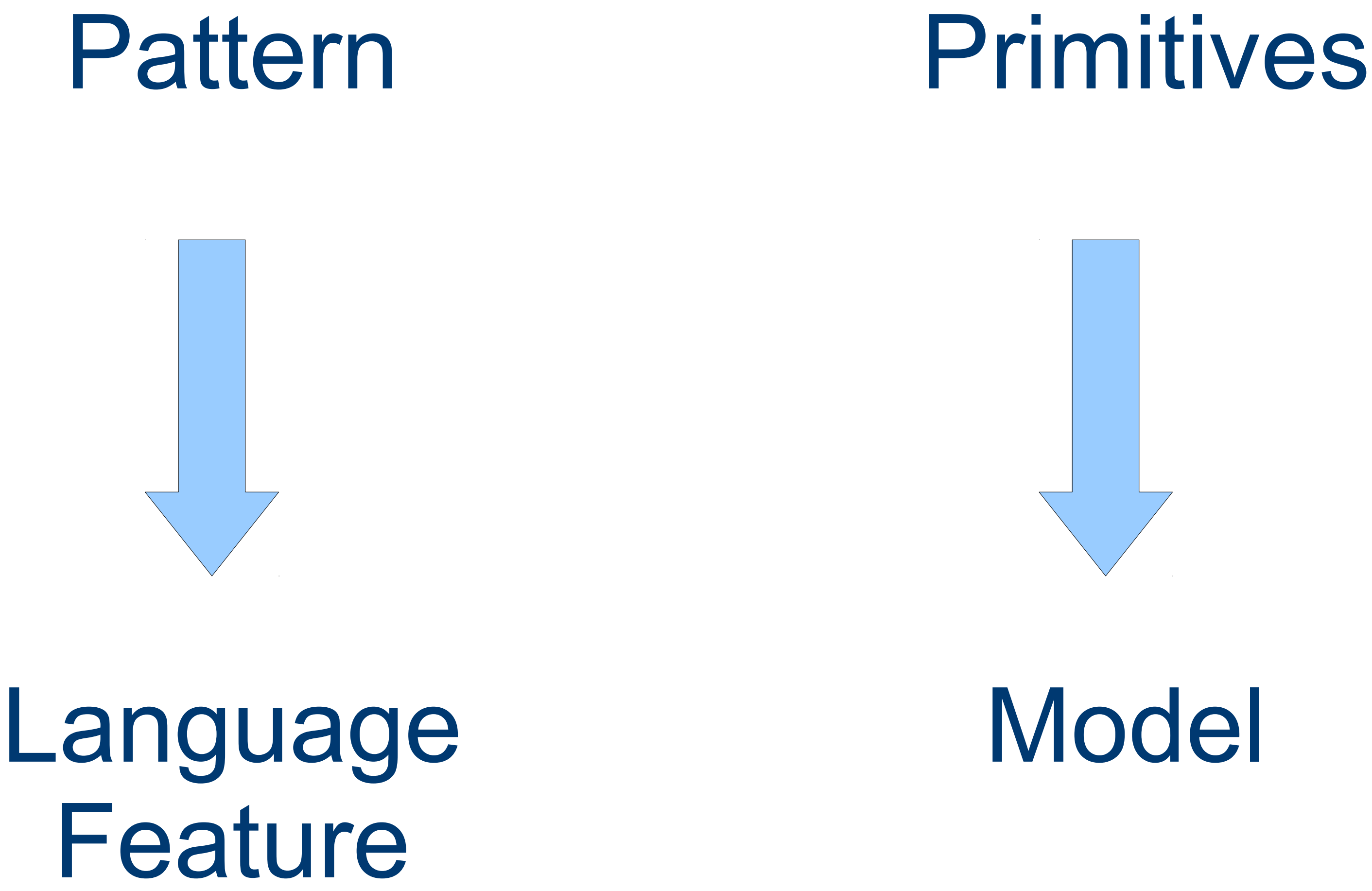
```
def hyp(x, y):
   return math.sqrt(x**2 + y**2)

>> hyp(3, 4)
5
```

# What if Structured Programming wasn't built in?

```
def hyp:
   push(pop()**2 + pop()**2)
   call math.sqrt
   return

>> push(3)
>> push(4)
>> call hyp
>> pop()
5
```

*You can do Structure Programming with our built in stack and 'call' primitives!*

# Patterns and Primitives



Pattern

Language Feature

Primitives

Model

# Some Concurrency Patterns listed on Wikipedia

Lock

Monitor Object

Reactor

Thread pool

Thread-specific storage

*These you will see on comp.lang.python*

# Some Concurrency Patterns listed on Wikipedia

Lock

Monitor Object

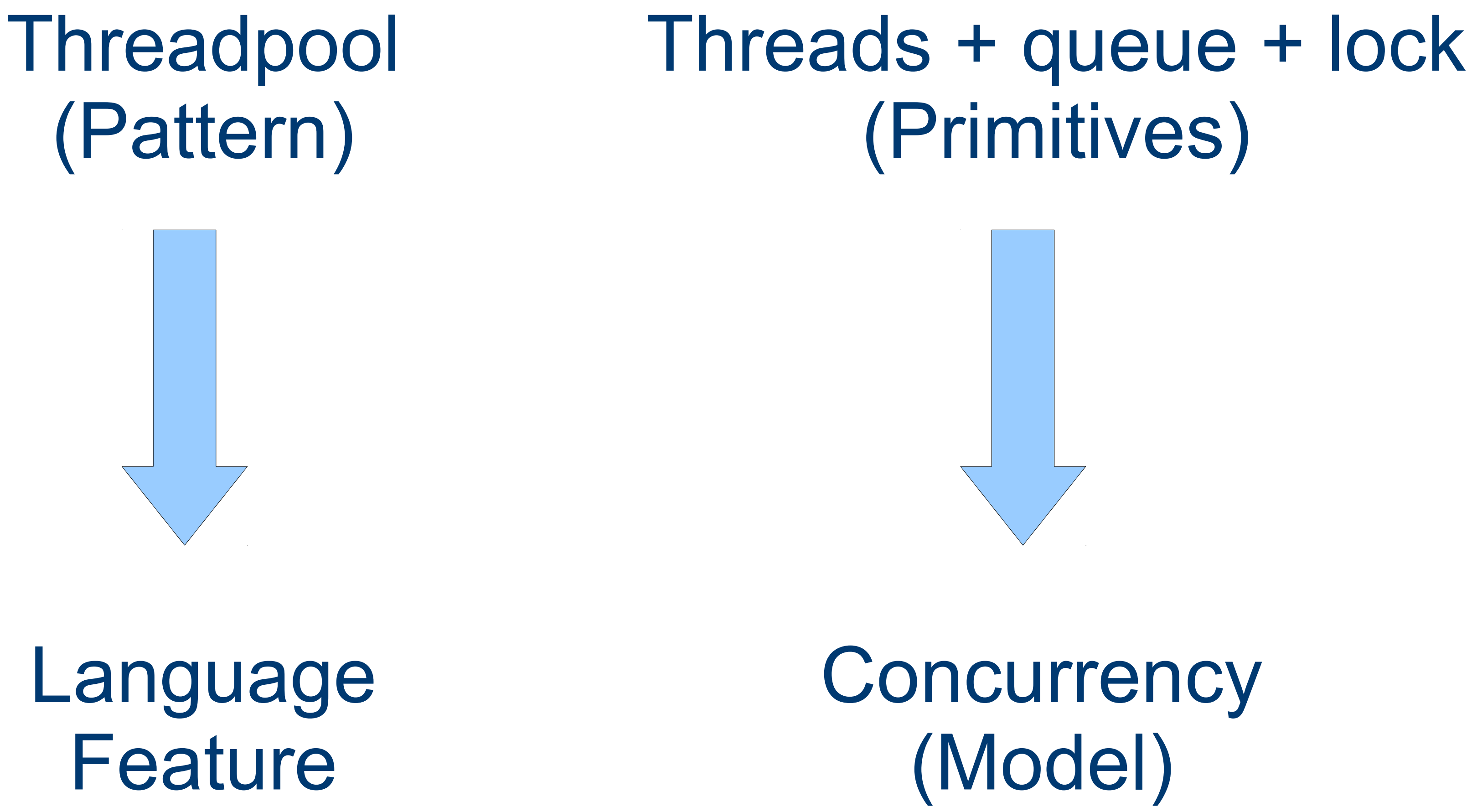
Reactor

**Thread pool**

Thread-specific storage

*These you will see on comp.lang.python*

## Patterns and Primitives



Threadpool
(Pattern)

Language
Feature

Threads + queue + lock
(Primitives)

Concurrency
(Model)

## “Just” use threads

# Threading is not a model

*Threading is a primitive, along with locks, transactional memory, etc.*

## What are the concurrency models?

1. Communicating Sequential Processes (CSP)

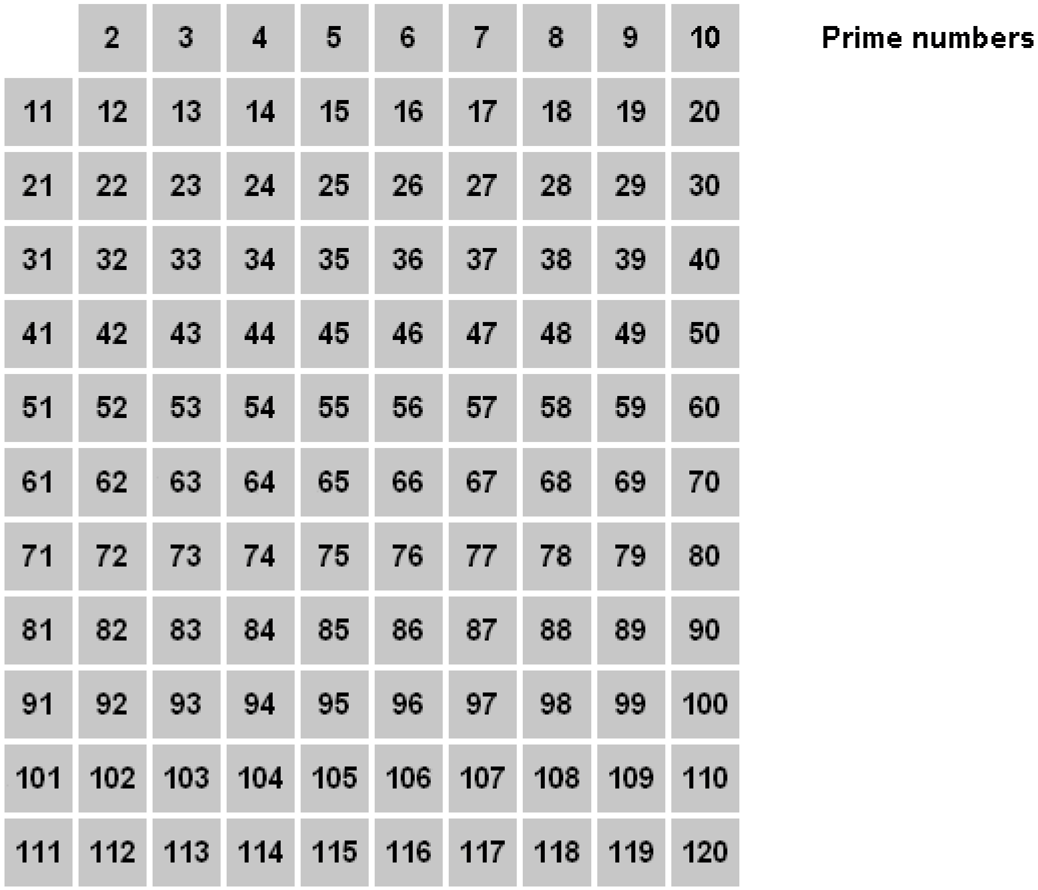
2. Actors

*The difference is only in 'what' is concurrent*

# CSP Model

- Based on CSP by C.A.R. Hoare.
- An actual model for processes
- All code is written single threaded
- Communication via channels.

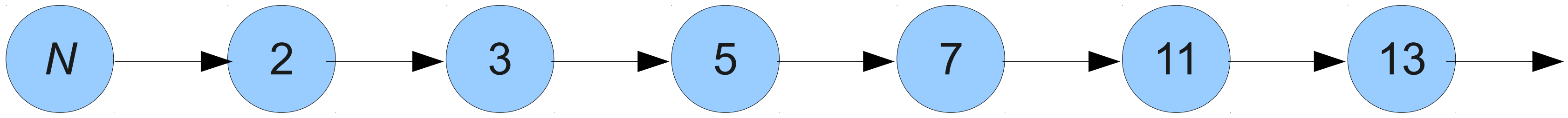
# Sieve of Eratosthenes



|  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|
|  | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
| 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 |
| 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 50 |
| 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 60 |
| 61 | 62 | 63 | 64 | 65 | 66 | 67 | 68 | 69 | 70 |
| 71 | 72 | 73 | 74 | 75 | 76 | 77 | 78 | 79 | 80 |
| 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 | 89 | 90 |
| 91 | 92 | 93 | 94 | 95 | 96 | 97 | 98 | 99 | 100 |
| 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 |
| 111 | 112 | 113 | 114 | 115 | 116 | 117 | 118 | 119 | 120 |

**Prime numbers**

# Sieve of Eratosthenes

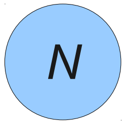


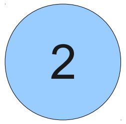
# CSP – Stackless – Primes

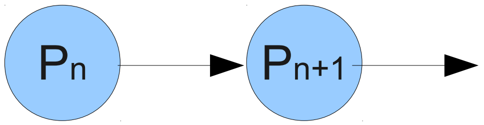
```
import stackless

def generate(ch):
    for i in range(2, 1000):
        ch.send(i)

def pfilter(chin, chout, p):
    for i in chin:
        if i % p != 0:
            chout.send(i)

def primes(chin):
    while 1:
        prime = chin.receive()
        print prime
        chout = stackless.channel()
        stackless.tasklet(pfilter)(chin, chout, prime)
        chin = chout

c = stackless.channel()
stackless.tasklet(generate)(c)
stackless.tasklet(primes)(c)
stackless.run()
```

# CSP – Stackless – Primes

```
import stackless

def generate(ch):
    for i in range(2, 1000):
        ch.send(i)

def pfilter(chin, chout, p):
    for i in chin:
        if i % p != 0:
            chout.send(i)

def primes(chin):
    while 1:
        prime = chin.receive()
        print prime
        chout = stackless.channel()
        stackless.tasklet(pfilter)(chin, chout, prime)
        chin = chout

c = stackless.channel()
stackless.tasklet(generate)(c)
stackless.tasklet(primes)(c)
stackless.run()
```

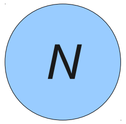


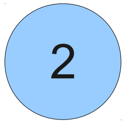


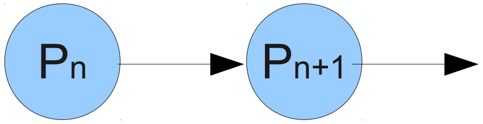


# CSP – Go – Primes

```
func generate(ch chan int) {
   for i := 2; ; i++ { ch <- i } // Send 'i' to channel 'ch'.
}

func filter(in, out chan int, prime int) {
   for {
      i := <-in  // Receive 'i' from 'in'.
      if i % prime != 0 { out <- i } // Send 'i' to 'out'.
   }
}

func main() {
   ch := make(chan int)  // Create a new channel.
   go generate(ch)  // Start generate() as a goroutine.
   for {
      prime := <-ch
      fmt.Println(prime)
      ch1 := make(chan int)
      go filter(ch, ch1, prime)
      ch = ch1
   }
}
```







# CSP

An implementation could use:

- Threads
- Locks
- Transactional Memory

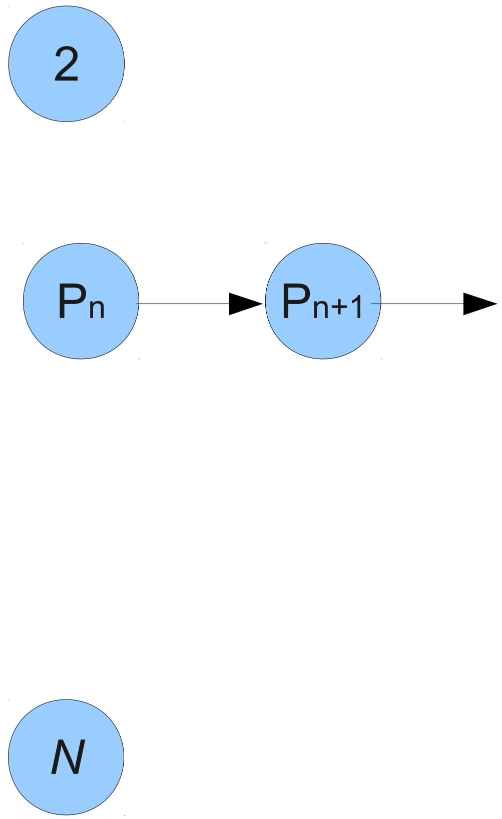
## Actor Model

- Objects are concurrent
- Objects send, and respond to messages
- All code is written single threaded

*Note that the 'channels' are implicit*

# Actors – IO – Primes

```
Filter := Object clone
Filter init := method(p,
  self prime := p
  self next := nil
  self
)

Filter number := method(n,
  r := n % prime;
  if (r != 0,
    if (self next == nil,
      n println;
      next = self clone init(n)
    )
    next @number(n); yield
  )
)

Filter init(2)
for (i, 2, 1000,
  Filter number(i); yield
)
```



# The path

A short story, a book, design patterns, and Djikstra

# Further Reading

http://golang.org

http://www.iolanguage.com/

http://www.stackless.com/

## Things not mentioned

- Futures
- Deterministic vs Non-Deterministic
- REST, MapReduce and other share-nothing architectures

# My Goal

Every time you use a concurrency pattern you remember the lack of affordances, and it proves sufficiently irritating.

*The short story, the book, and design patterns.*