Principles of Computer Systems

Prof. George Candea
School of Computer & Communication Sciences

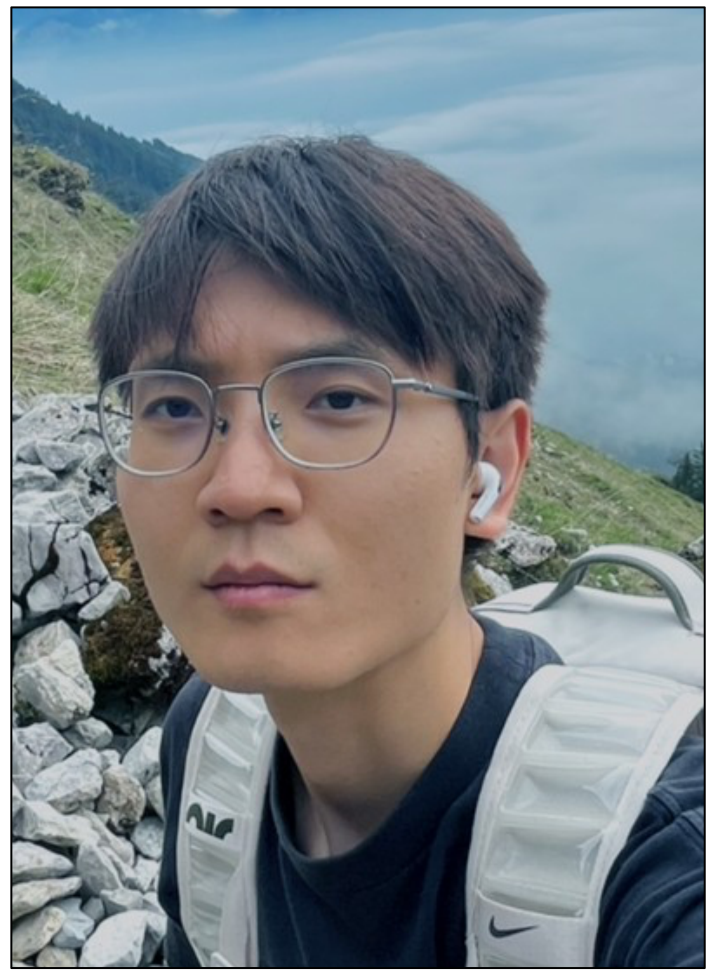
Administrivia

Your POCS Team


Katerina Argyraki
Instructor


George Candea
Instructor


Can Cebeci
TA


Jiacheng Ma
TA

Syllabus

Case studies

Syllabus and schedule subject to change

Week	Topic	Who
1	Introduction Modules & Interfaces	George
2	Naming & Indirection	Katerina
3	Case study: Memory virtualization	George
4	Layering (organization of modules)	Katerina
5	Case study: Machine virtualization	Ed Bugnion
6	Case study: Internet design	Katerina
7	Client/server organization	George
8	Locality & Caching	Sanidhya Kashyap
9	Redundancy	George
10	Case study: Transactions	George
11	Case studies in HW/SW co-design	Thomas Bourgeat
12	Lazy / speculative execution	Butler Lampson
13	Hints for Computer System Design	George & Katerina
14	Exam	

Fundamentals

Guest lecturers

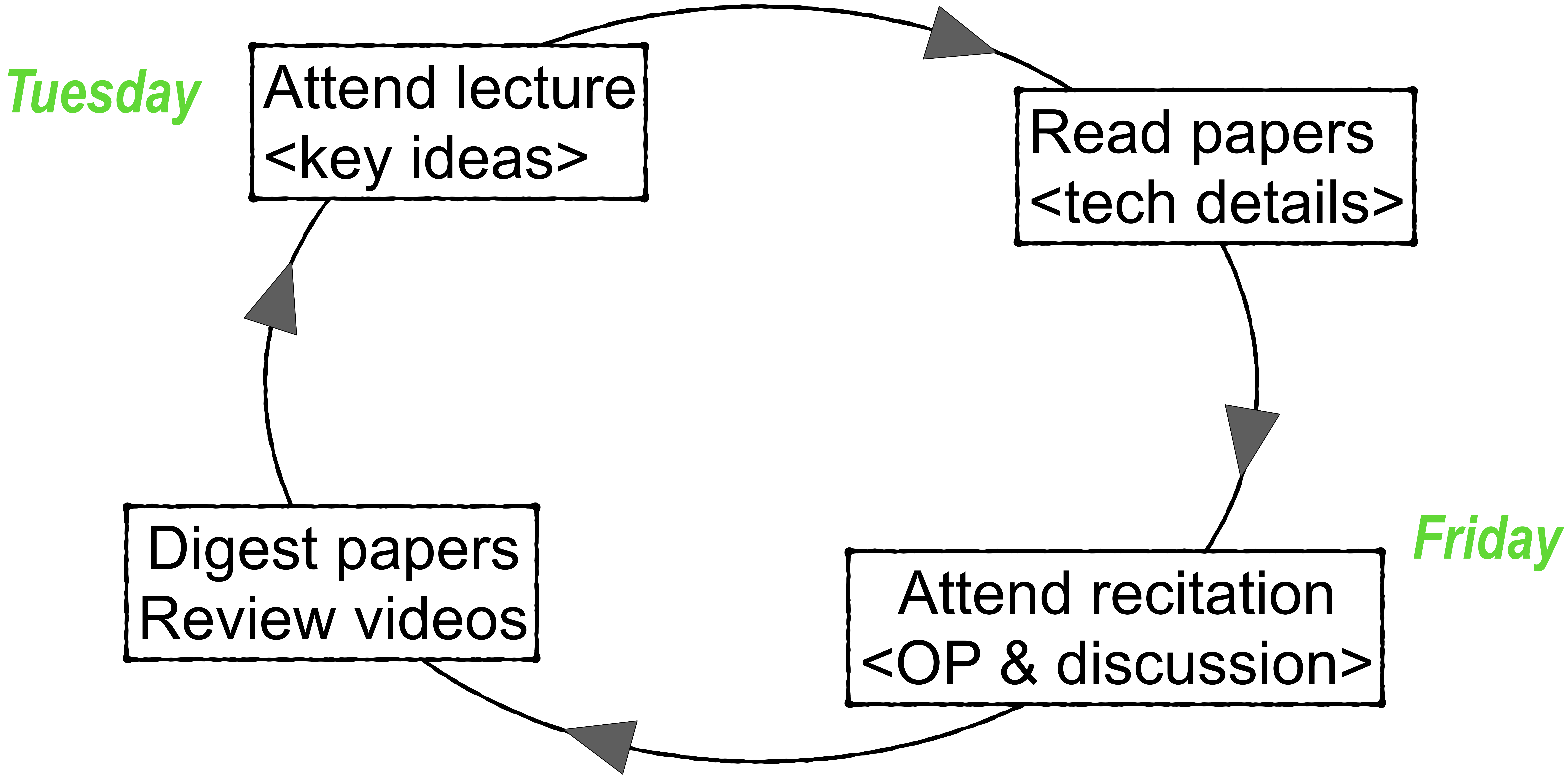
Prerequisites

Take prerequisites very seriously

- Good knowledge of
  - *Operating systems (e.g., via CS-323)*
  - *Networks (e.g., via COM-407)*
  - *Computer architecture (e.g., via CS-470)*
  - *Databases (e.g., via CS-422)*
- Read the Exokernel, GNS, and Chord papers
  - *if you don't "get it" then POCS might not be right for you at this time*
- You cannot "just wing" POCS

Typical week in POCS

Don't fall behind



Grading

- OPs (one-paragraphs) = 50%
  - *demonstrate that you understood the papers*
  - *learn to identify system challenges and the techniques used to solve them*
  - *learn to express your ideas concisely*
  - *individual work done in class, closed book*
  - *graded on a curve*
- Exam = 50%
  - *in-class during the last week, closed book*
  - *read a system description, short paper, etc.*
  - *... then answer individual questions similar to OPs*
  - *graded on a curve*

Resources

- Moodle
  - *slides*
  - *lecture recordings (best-effort)*
  - *Ed Discussion*
- Course website
  - *all static content*

Advice

- 8 credits = heavyweight course
  - *~17 hours/week*
- Do not take POCS if you don't have the background
  - *Really, this is no joke !*
- Do not fall behind
  - *pace is fast, if you lose one week, it's hard to recover*
- Ask classmates/TAs/instructors when you don't fully grasp something
  - *don't just "let it be", because it may come back to bite you later*
  - *Really, do not fall behind !*

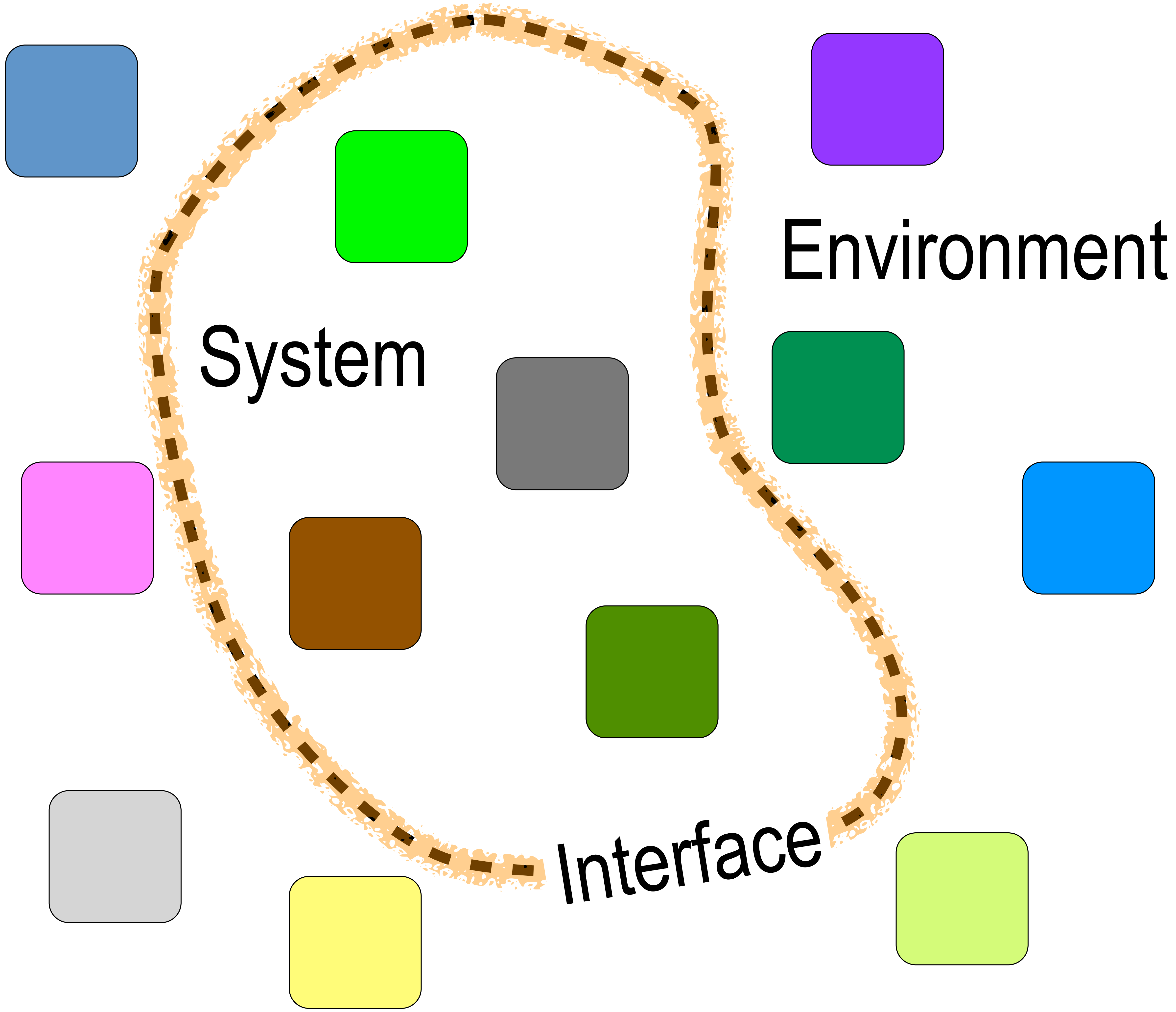
What does it mean to study the principles of comp sys design?

What does it mean to study principles of comp sys design ?

- What do we mean by a system ?
- What are the challenges in building and maintaining systems ?
- How do we address those challenges ?

What is a Computer System ?

Definition : A system is a group of interconnected components that exhibits an expected collective behavior observed at the interface with its environment.



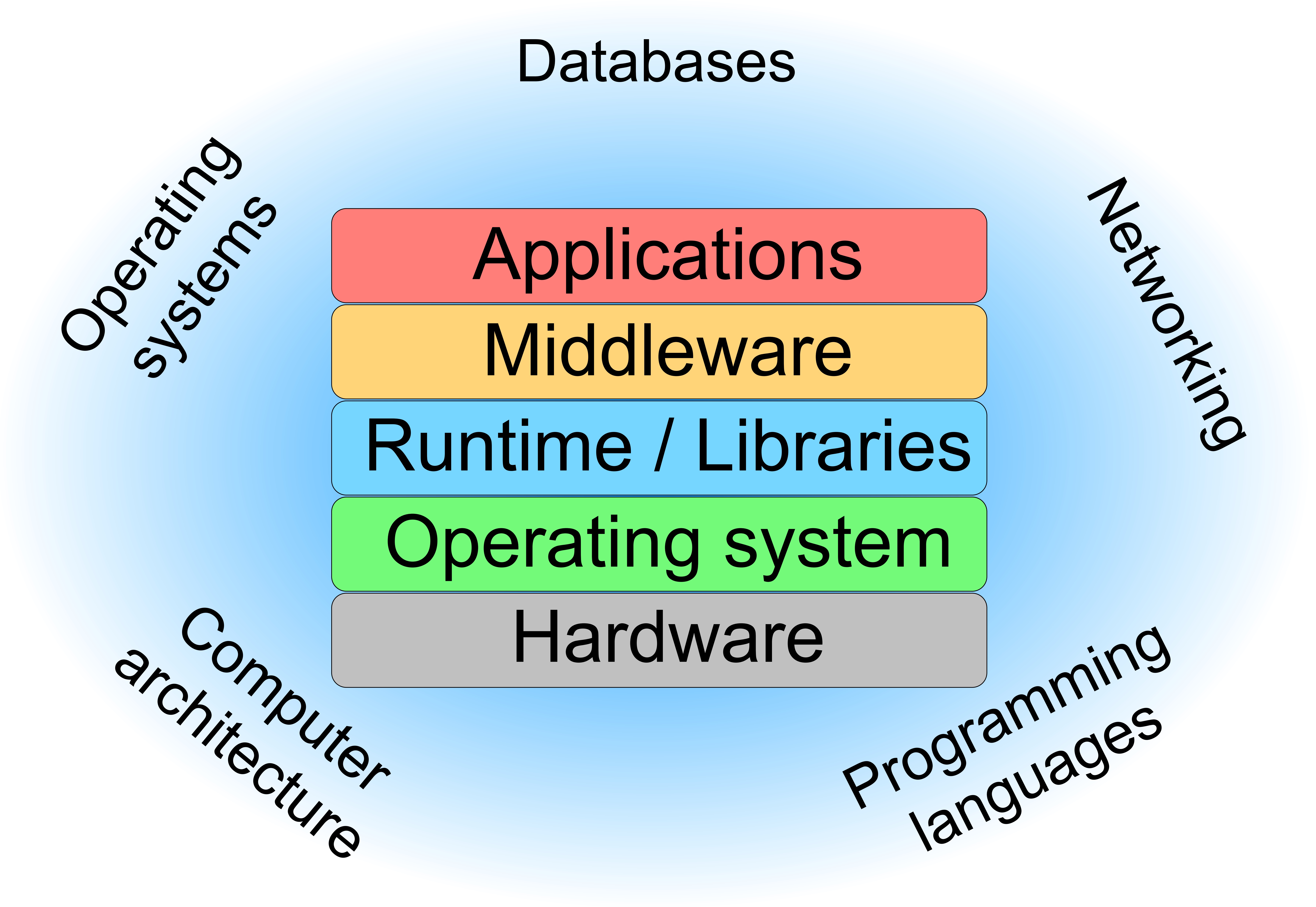
Examples of Systems

System	Environment	Interfaces
OS kernel = code	hw + applications + libraries + ...	syscall interface
Smartphone = hw + OS + libraries + apps	cell towers + GPS satellites + cloud svcs + users + ...	network protocols, touch screen, ...
Smart home controller = hw + OS + libs	HVAC devices + access-control devices + meteo station + inhabitants + ...	KNX protocol, HTTP
Amazon WS = hw + code	apps + Internet + Web browsers + credit card billing svcs + ...	x86, provisioning API, HTTP, ISO 8585, ...

Properties of Good Systems

- Safety
- Security
- Reliability
- Performance
- Manageability

Definition : A system is a group of interconnected components that exhibits an expected collective behavior observed at the interface with its environment.


Databases
Operating systems
Networking
Applications
Middleware
Runtime / Libraries
Operating system
Hardware
Computer architecture
Programming languages

" Systems Thinking "

- global all-encompassing vs. narrow focus on individual aspects
- study many prior systems to understand what made them succeed/fail
- using back-of-the-envelope calculations to quickly eliminate designs that wouldn't work

Today's lecture ...

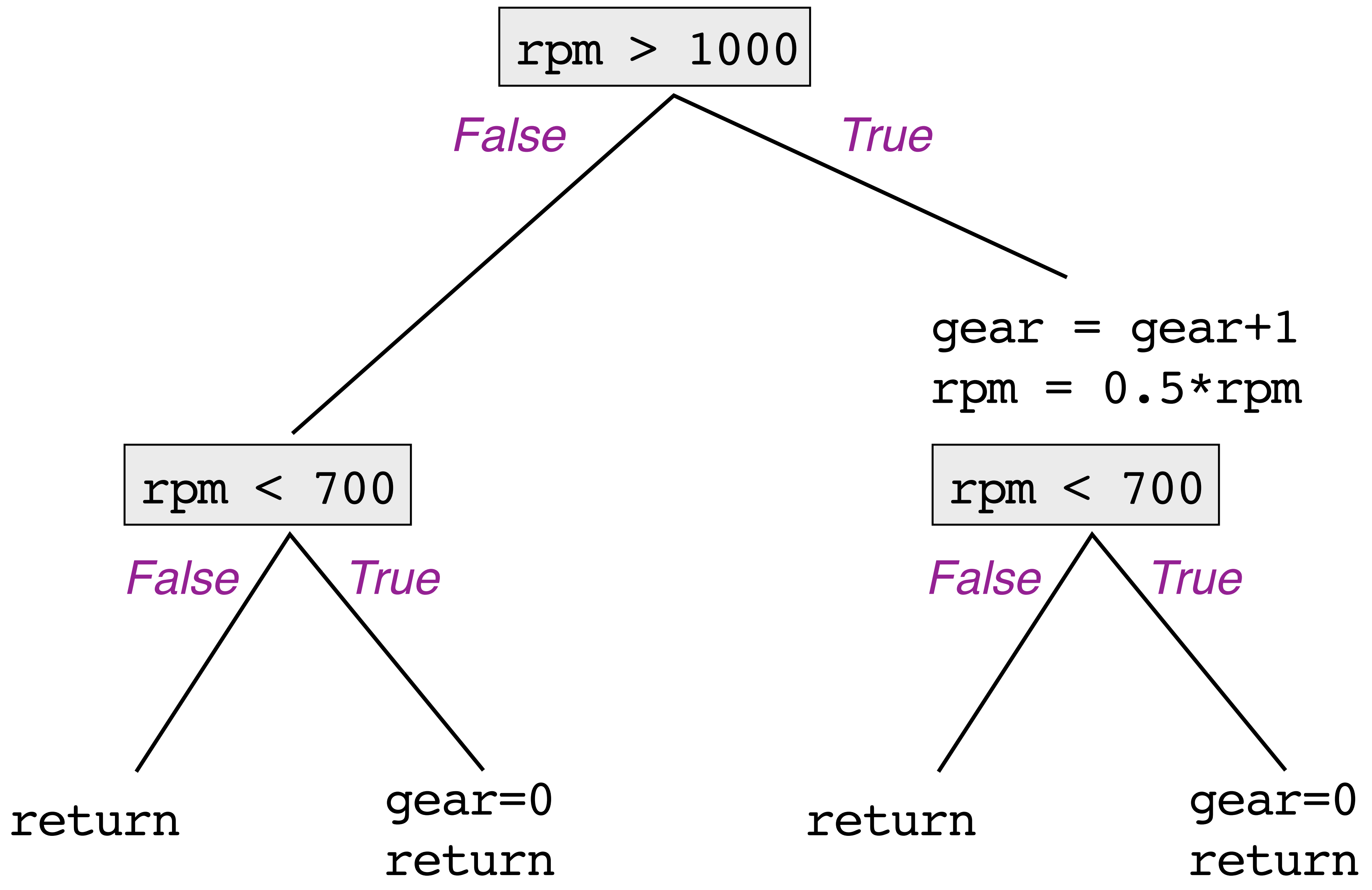
- Sources of complexity:
  - *lots of code, emergent behaviors, many interconnections, evolution, trade-offs*
- Use modularity to
  - *encapsulate elements into components & subsystems => fewer visible elements*
  - *control interactions and propagation of behaviors => fewer interconnections*
- Use abstraction to
  - *make emergent behavior predictable => less irregularity & fewer exceptions*
- Later on...
  - *patterns of using modularity and abstraction (layering, naming, client/server, etc.)*

5 Challenges in Comp Sys Design

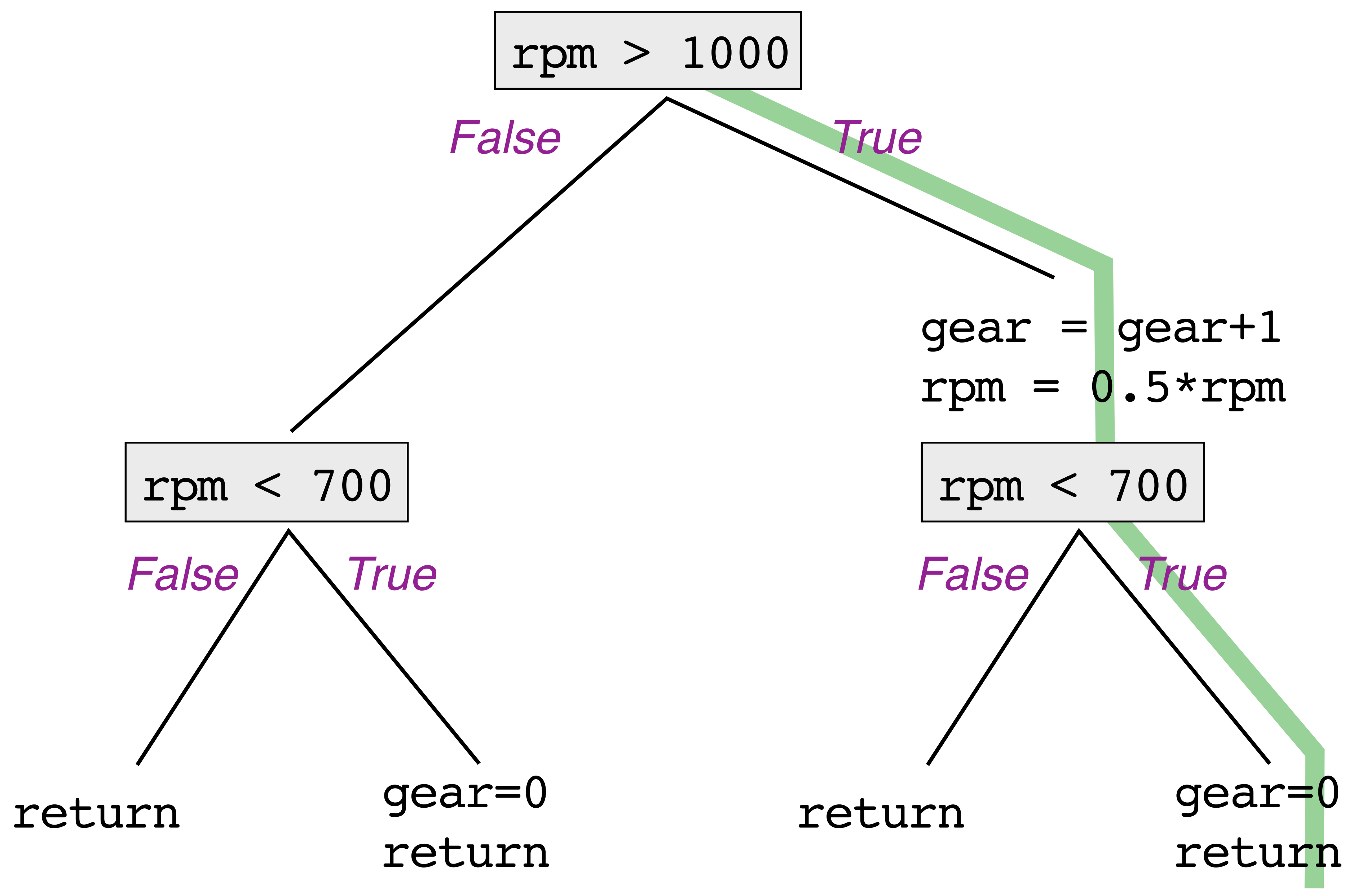
#1 — Software/firmware has lots of possible behaviors

rpm=1200

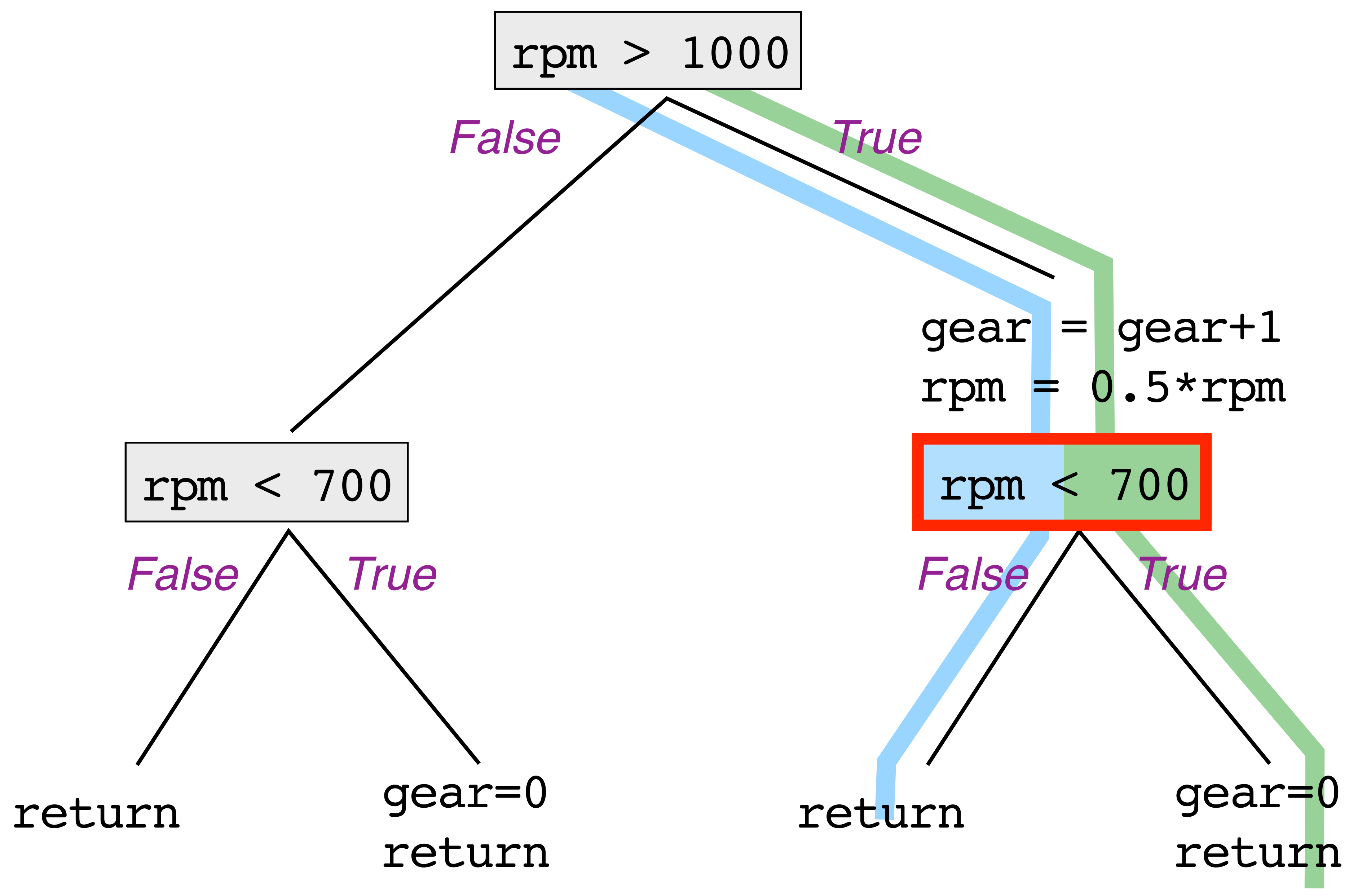
```
autoShift (int rpm)
   if (rpm > 1000)
      gear = gear+1
      rpm = 0.5*rpm
   if (rpm < 700)
      gear=0
   return
```



```
autoShift (int rpm)
   if (rpm > 1000)
      gear = gear+1
      rpm = 0.5*rpm
   if (rpm < 700)
      gear=0
   return
```

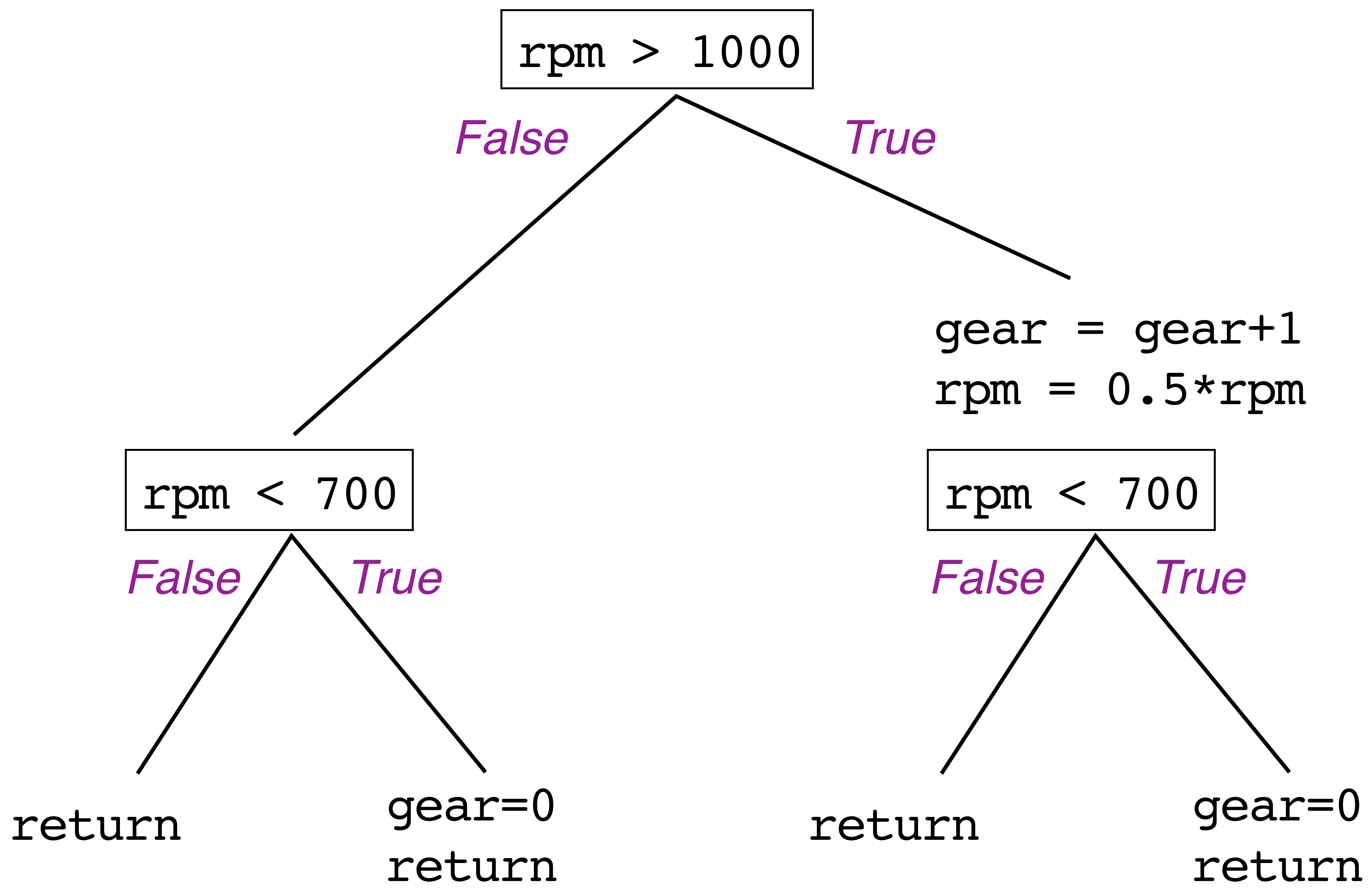


```
autoShift (int rpm)
   if (rpm > 1000)
      gear = gear+1
      rpm = 0.5*rpm
   if (rpm < 700)
      gear=0
   return
```

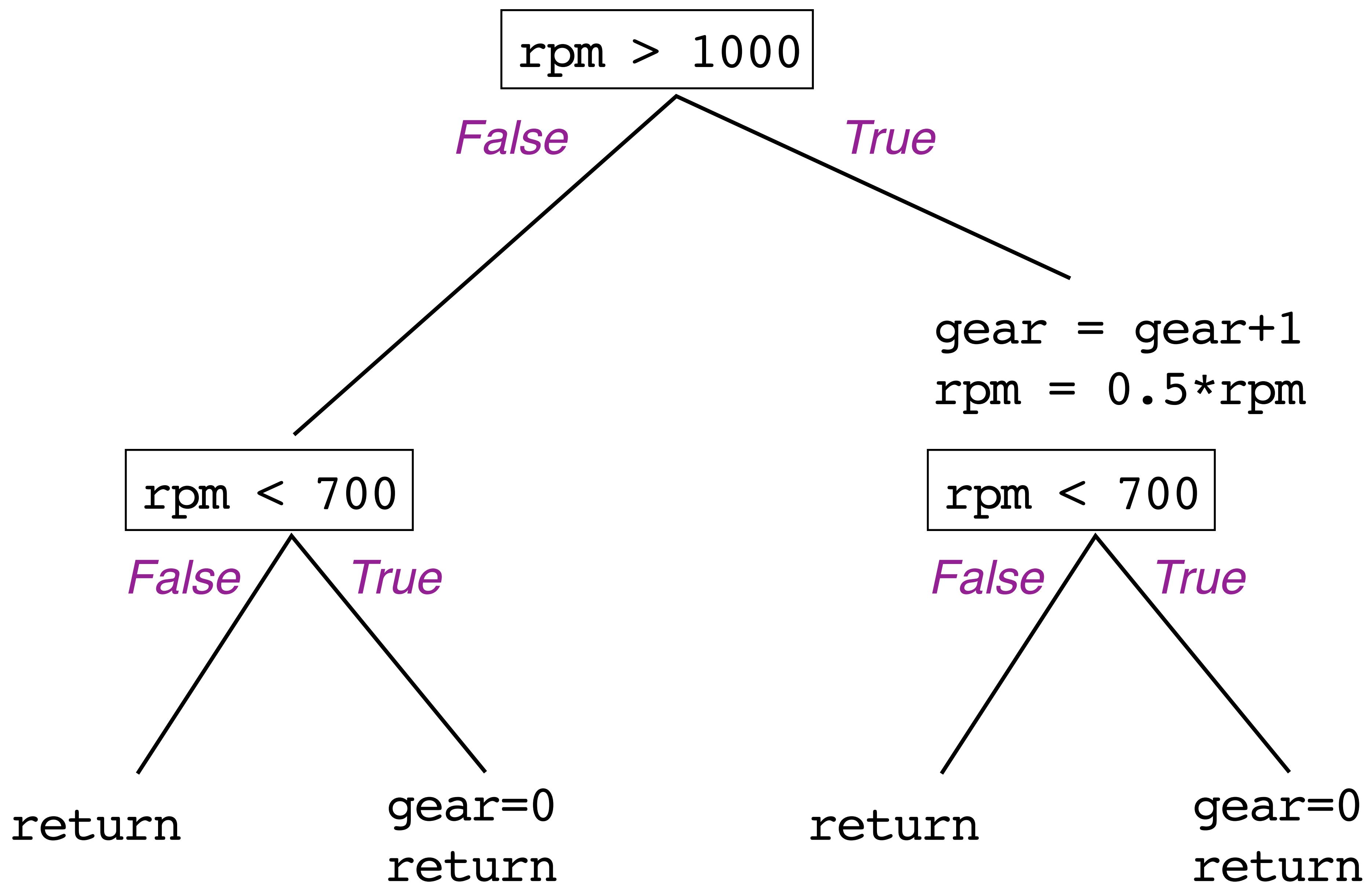


rpm ∈ {0,350,944,1200,1800}

```
autoShift (int rpm)
   if (rpm > 1000)
      gear = gear+1
      rpm = 0.5*rpm
   if (rpm < 700)
      gear=0
   return
```



```
autoShift (int rpm)
   if (rpm > 1000)
      gear = gear+1
      rpm = 0.5*rpm
   if (rpm < 700)
      gear=0
   return
```



paths ≃ 2^program size^

$\text{paths} \simeq 2^{\text{program size}}$

>5,000,000 lines of code[1] (LOC) ⇒ $\sim 2^{500,000}$ paths

Can we test $2^{500,000}$ paths?

[1] Black Duck Software, Inc. *Mozilla Firefox Code Analysis*, http://www.ohloh.net/p/firefox/analyses/latest

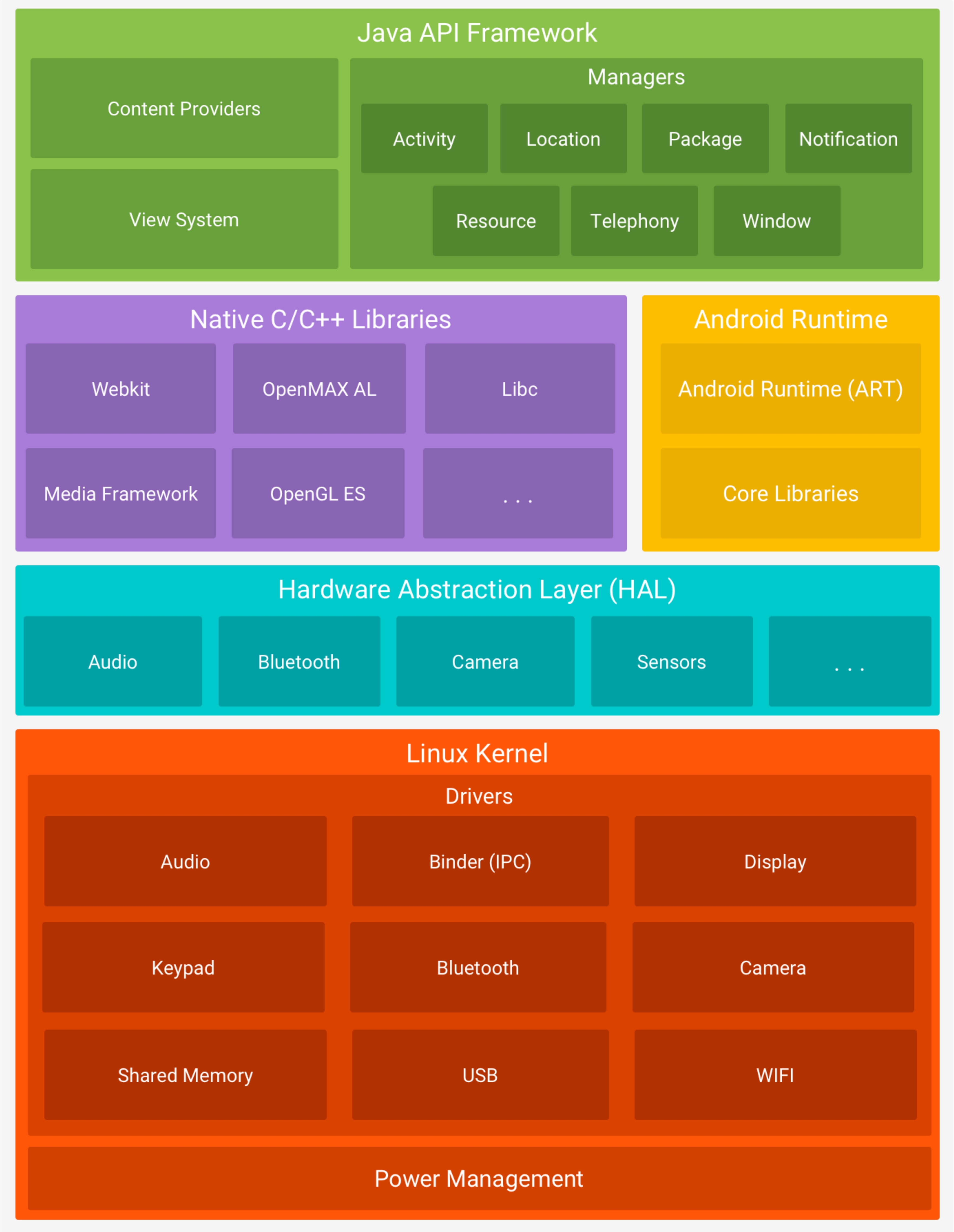
Some Code Sizes (in LOC)

- Boeing 787 avionics + online support ~several million LOC
- Chrome browser ~several million LOC
- entry-level electric vehicle (Chevy Volt) ~10 million LOC
- Android operating system ~a few tens of millions LOC
- the Large Hadron Collider ~50 million LOC
- all car software in a high-end car ~100 million LOC
- all Google services combined ~2 billion LOC

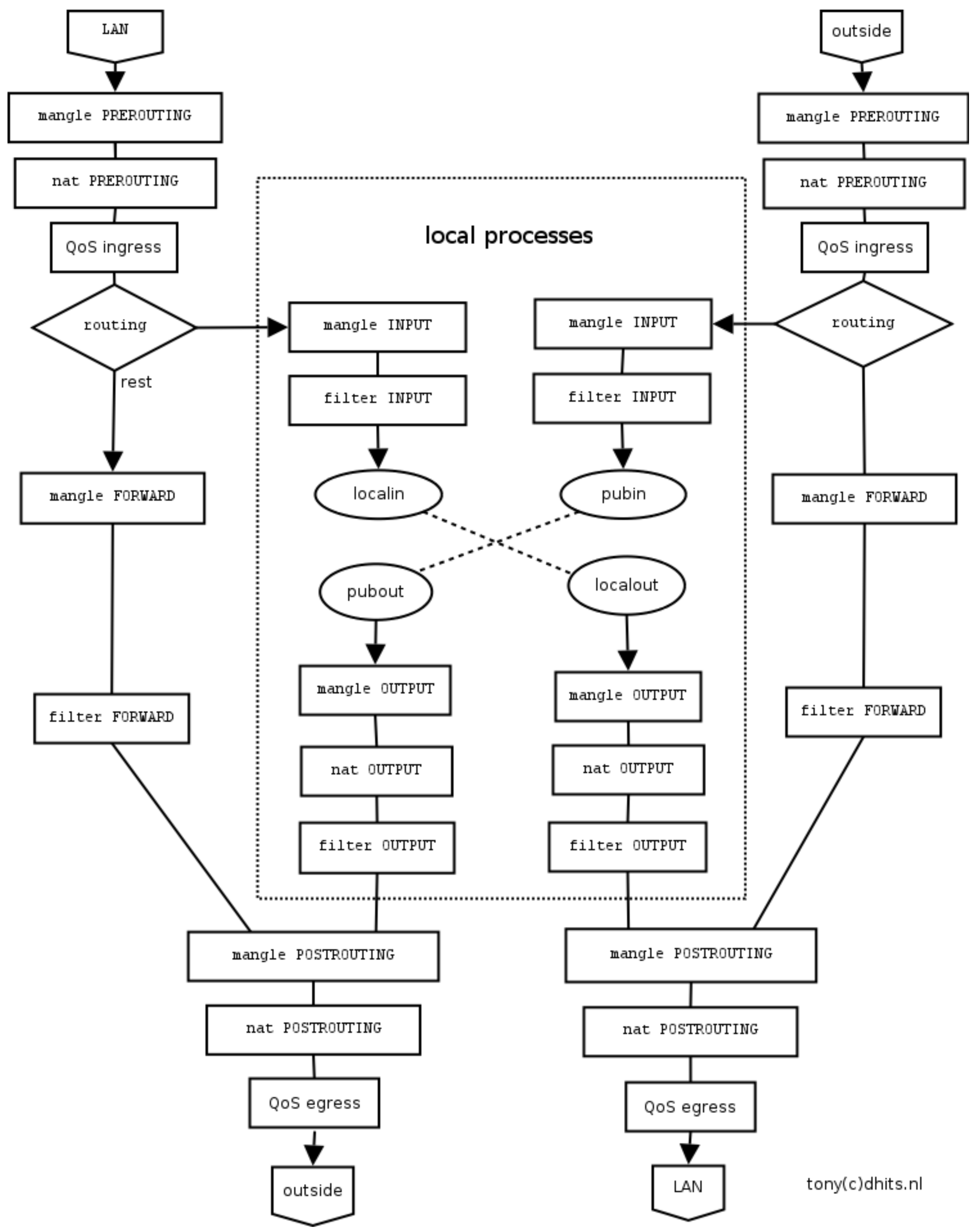
#2 — Combining many components leads to unforeseen interactions

Many Components

Android stack



https://developer.android.com/guide/platform



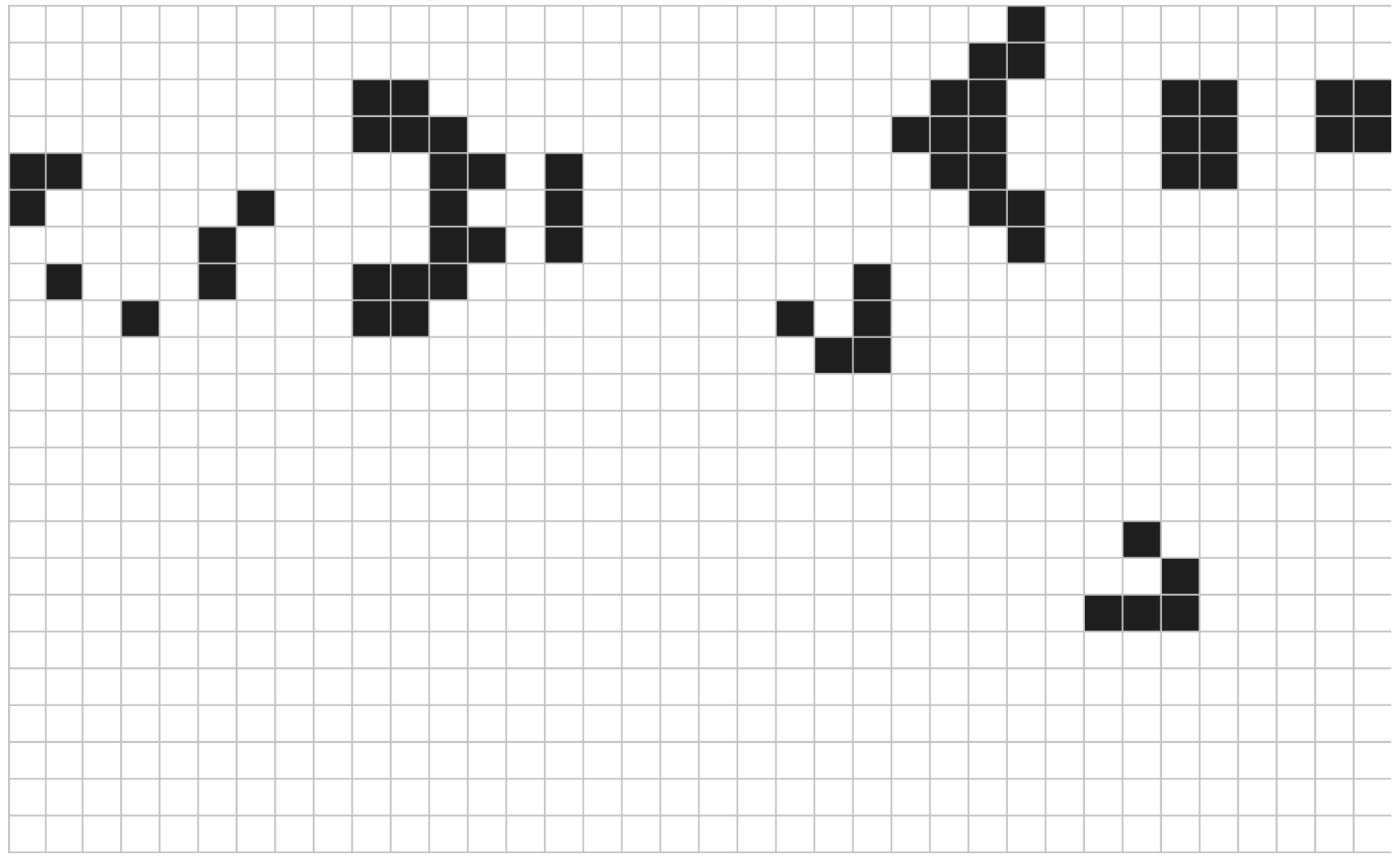
Netfilter FW/NAT

Emergent Behaviors

Behaviors that are not evident in the components, but appear when the components are combined.

- Functional
  - *ant colonies, blockchain, deadlock, livelock, ...*
- Non-functional
  - *reliability, security,...*

Conway's Game of Life



Emergent Behaviors in Computer Systems

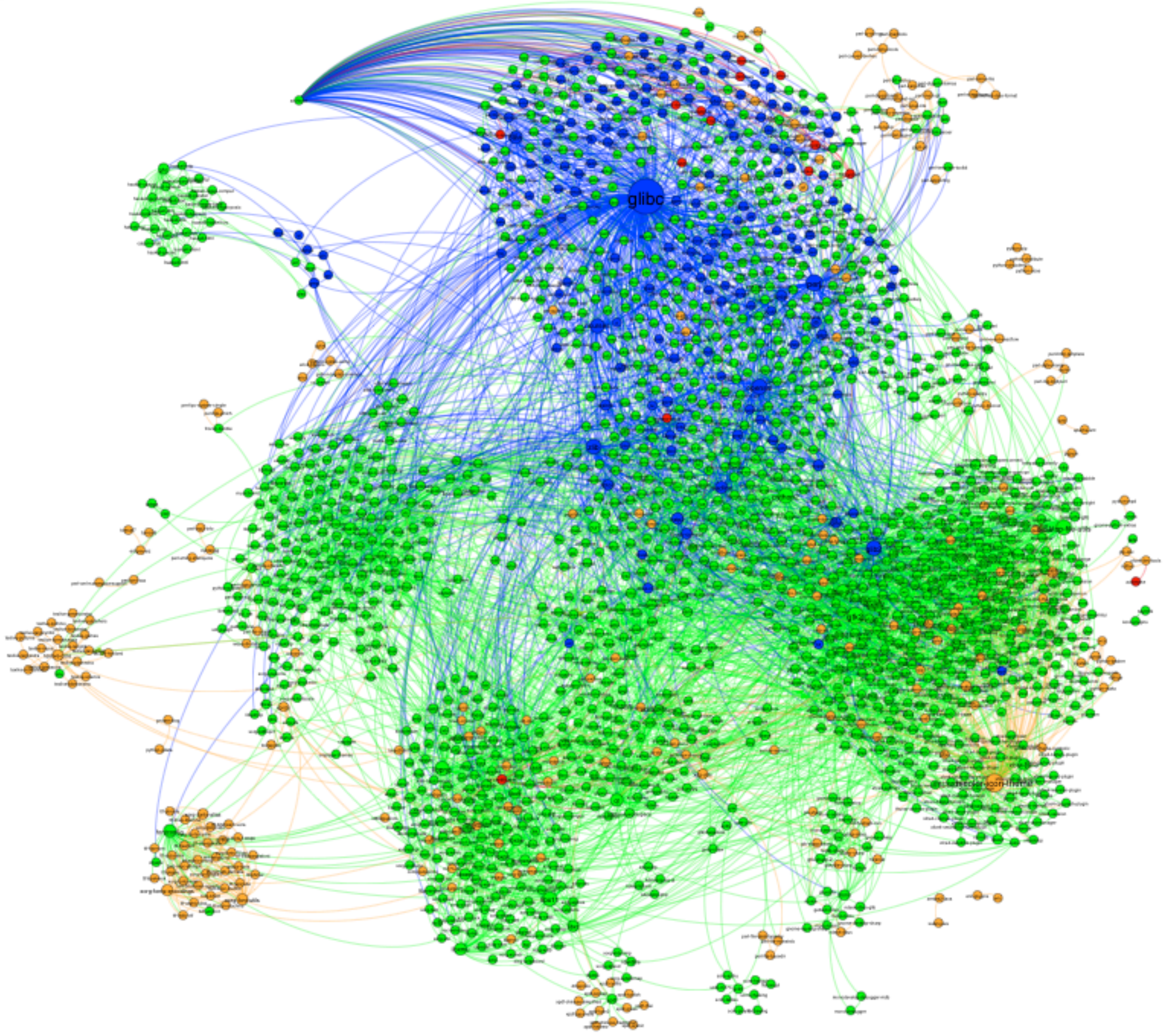
- Thrashing
- Unwanted synchronization
- Unwanted oscillation or periodicity
- Livelock/Deadlock
- Phase change
- ...

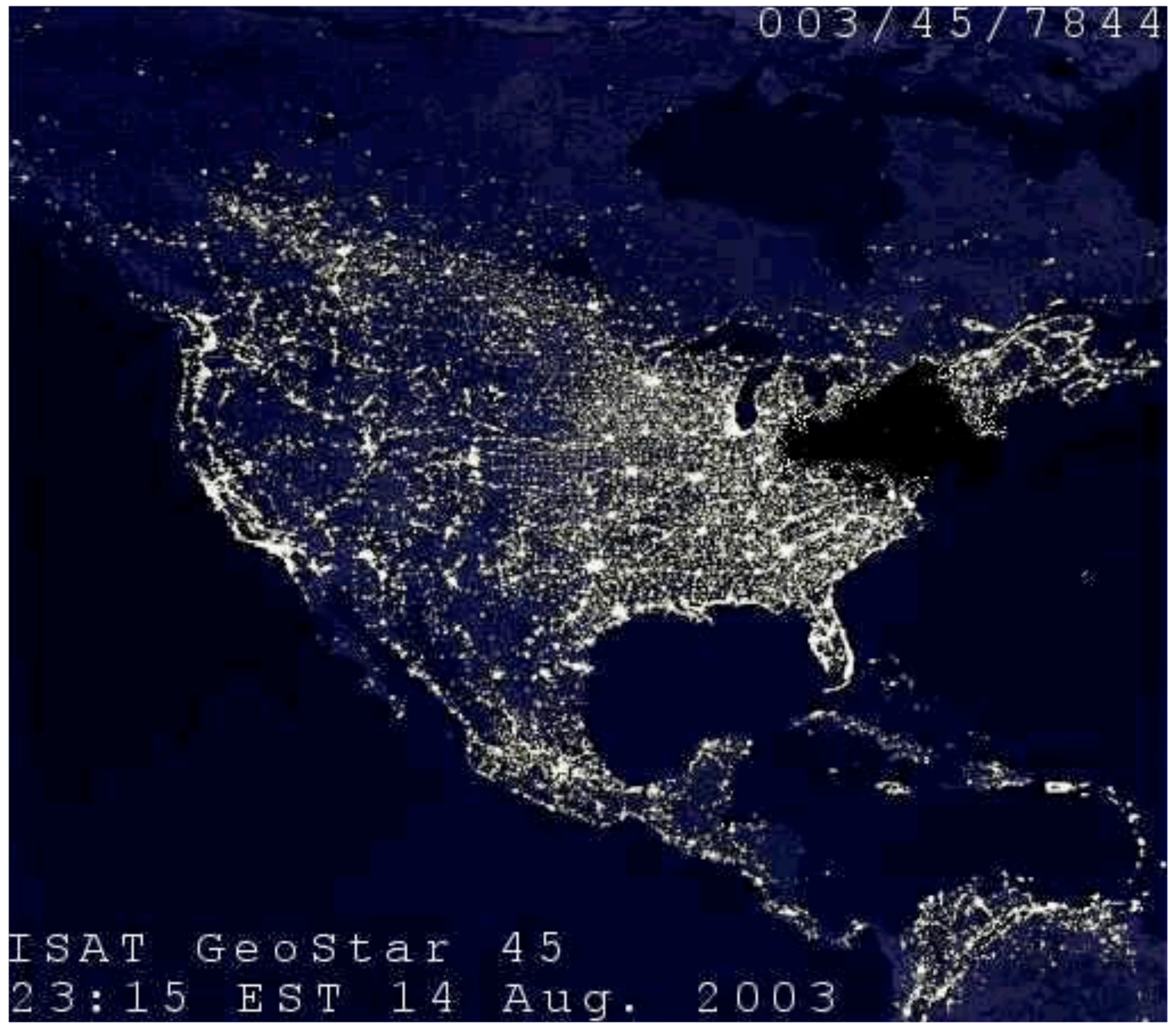
For more insights, see
J. Mogul, Emergent (Mis) behavior vs. Complex Software Systems,
ACM SIGOPS Operating Systems Review, October 2006

#3 — Propagation of Effects

Many Interconnections -> Propagation of Effects

The transitivity of component interconnections causes a local phenomenon to propagate to large parts of the system.

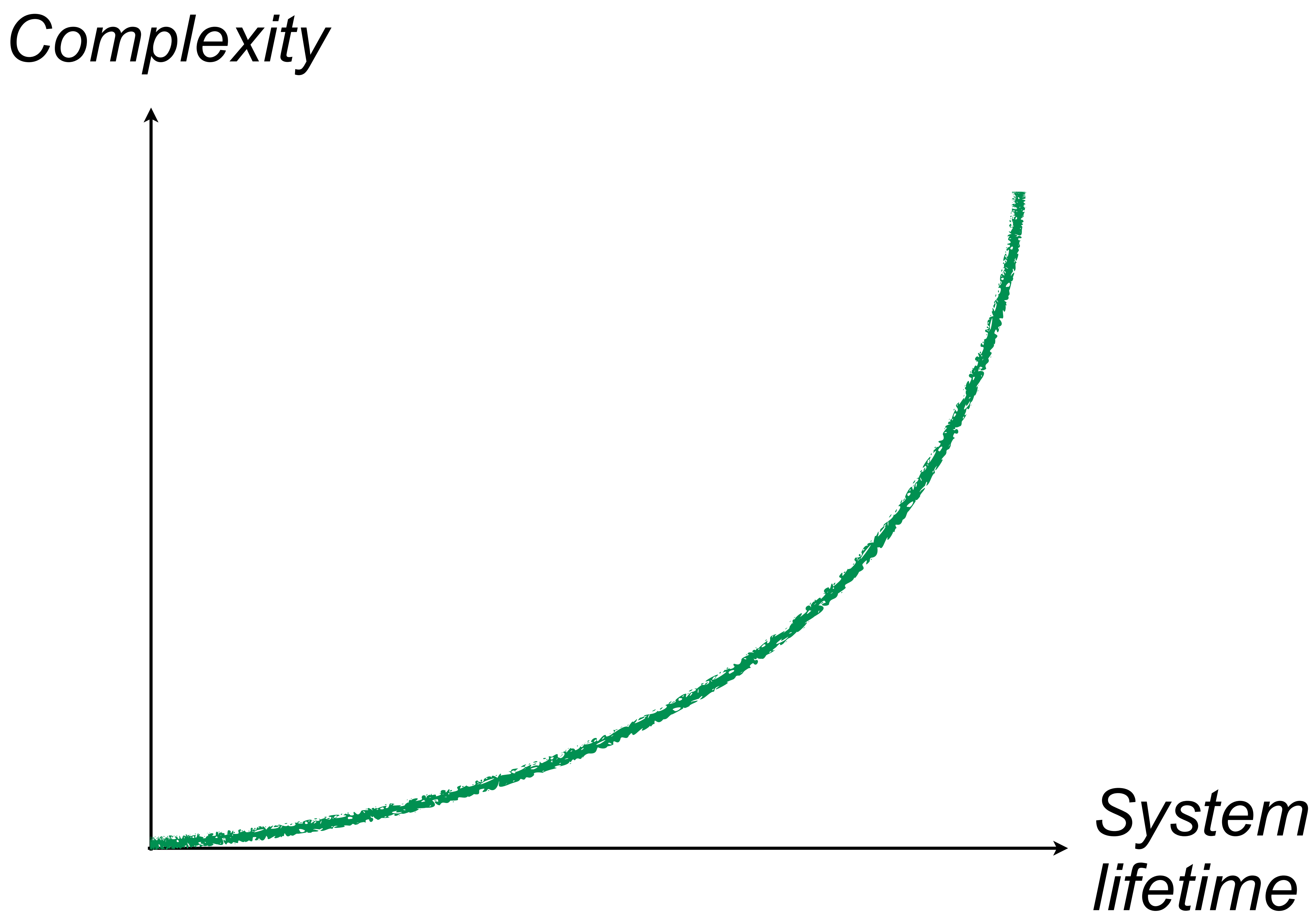




#4 — System evolution makes these challenges even harder*

* not always... e.g., redesign and refactoring

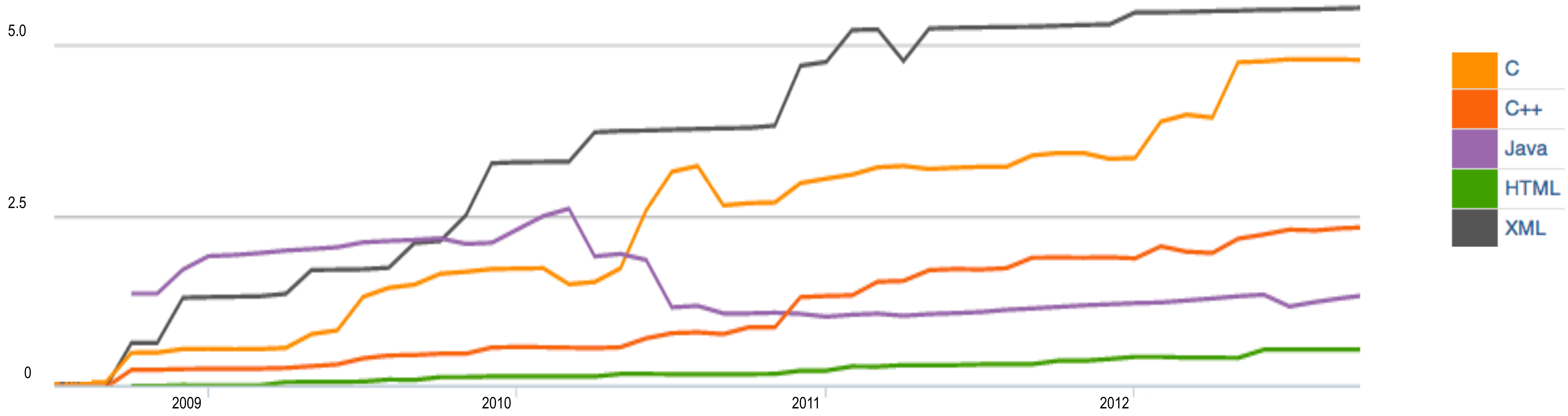
System Evolution ⇒ Complexity



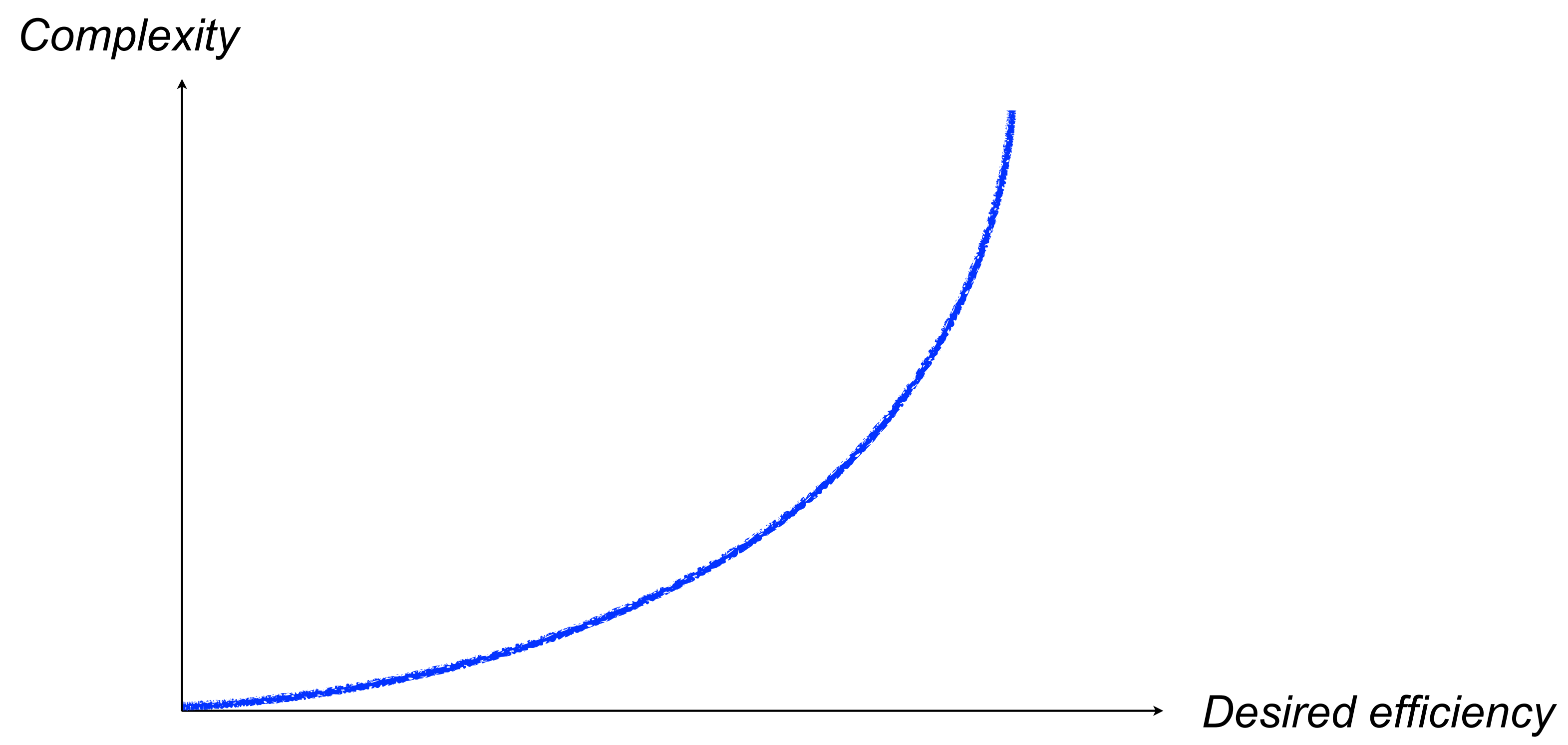
Legacy Systems = Complex Systems

- Evolution
  - *is a process of satisfying new requirements*
  - *successful systems evolve fast*

Android codebase size
(in millions of lines, over its first 4 years)

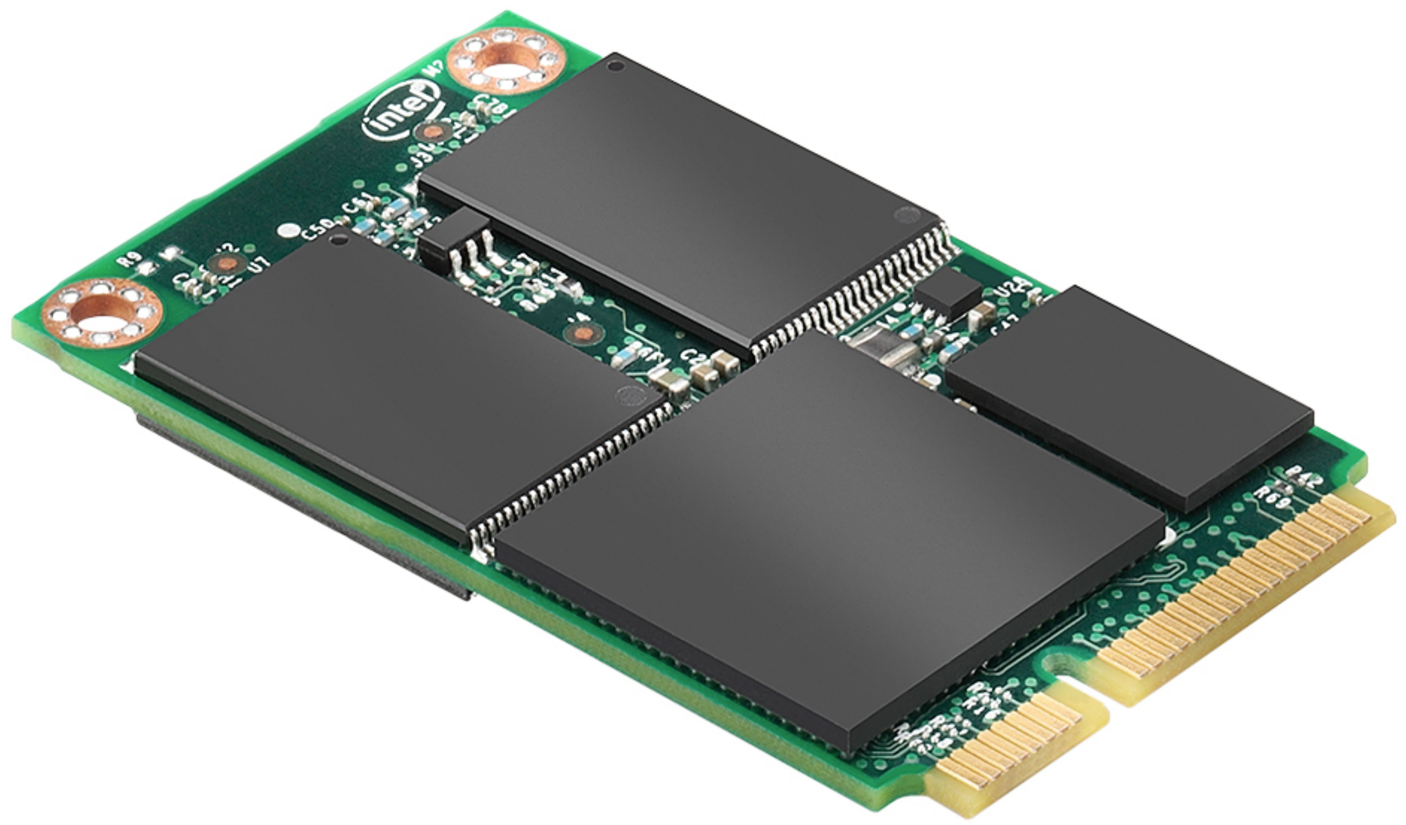


Quest for Efficiency ⇒ Complexity

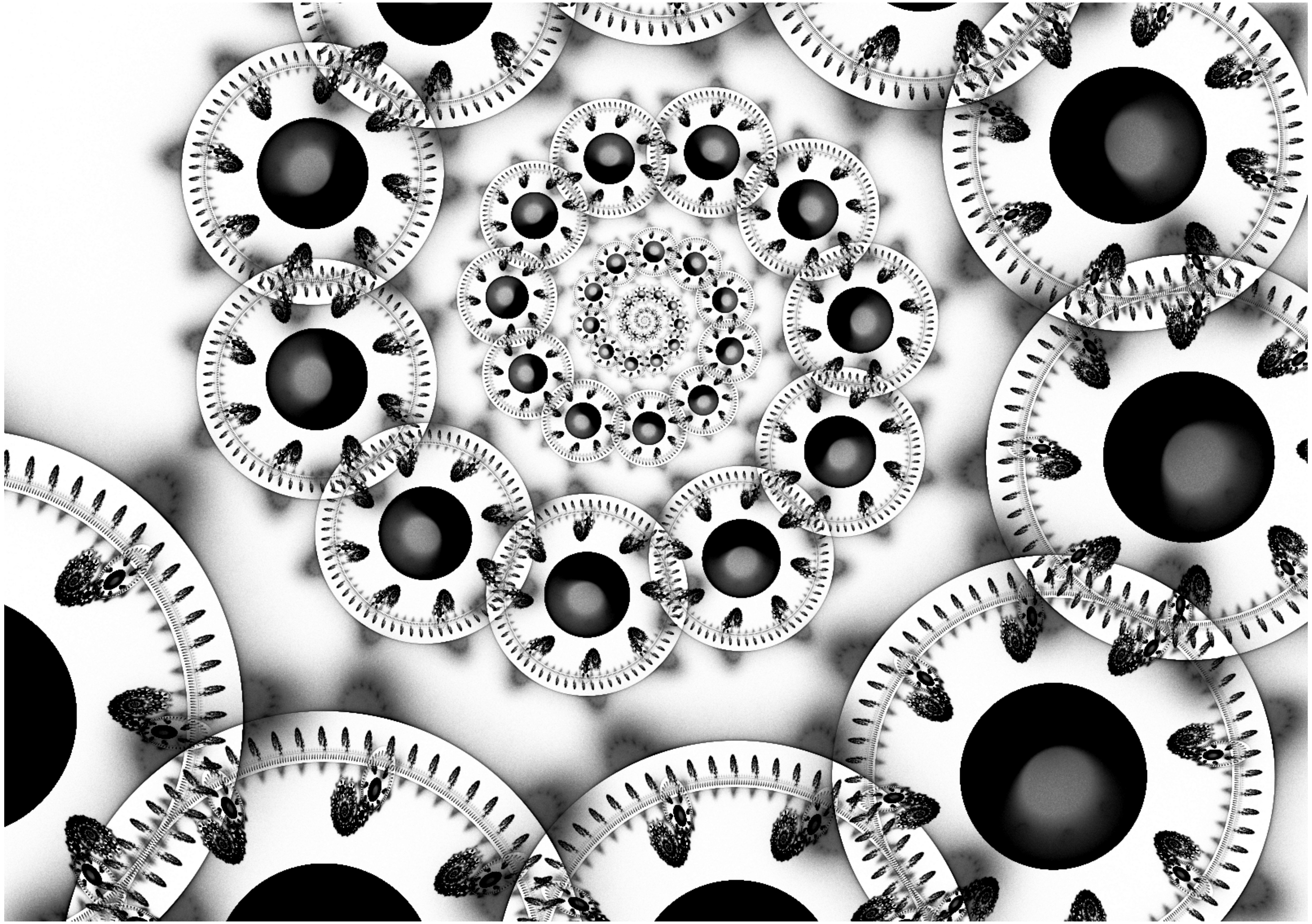


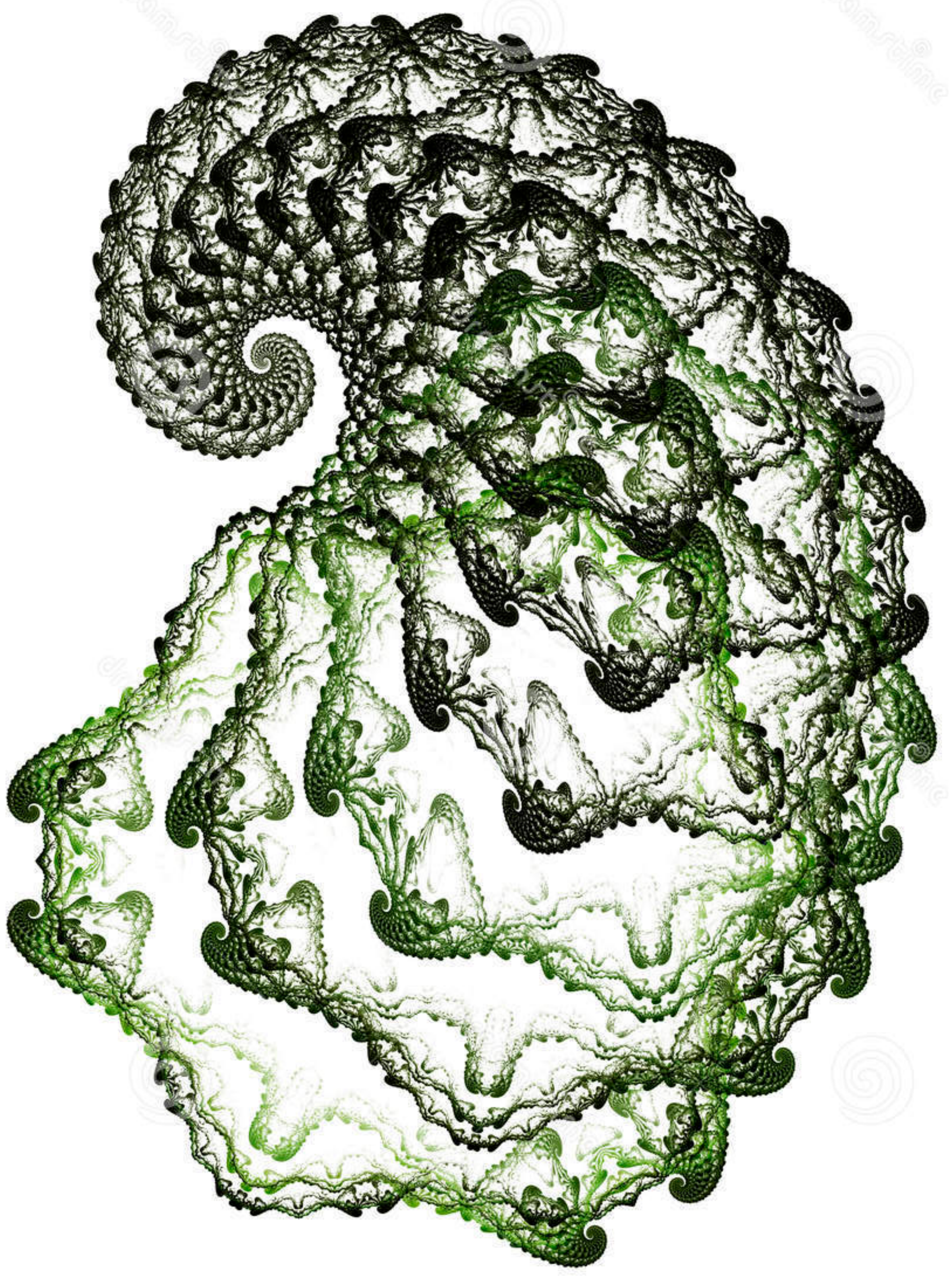
fsync()

Writeback buffer cache



Irregularity and Exceptions ⇒ Complexity





Images from http://dreamstime.com and http://www.wallpapers-backgrounds.net

System specifications

- IEEE 802.11 standard for wireless networking
  - *published in 1997 → 45 pages*
  - *1999 revision → 90 pages*
  - *2007 revision → 1,250 pages (incl. amendments)*
  - *2012 revision → 2,793 pages (incl. amendments)*
- HTTP protocol
  - *RFC 1945 (1996) HTTP/1.0 → 60 pages*
  - *RFC 2068 (1997) HTTP/1.1 → 160 pages*
  - *RFC 2616 (1999) HTTP/1.1 v.2 → 176 pages long*

Quantify irregularity

- "Kolmogorov complexity"
  - *computation resources needed to specify an object*
  - *minimal length of a description of the object*
- K(object) >= |object| => complex

  K(object) << |object| => simple

|*AAAAAAA ... AAAAB*| = 10^6+1

K(*AAAAAAA ... AAAAB*) =

|"1 million *A*s followed by 1 *B*"|

⇒ simple

|*ABDAGHDBBCAD...*| = 10^6+1

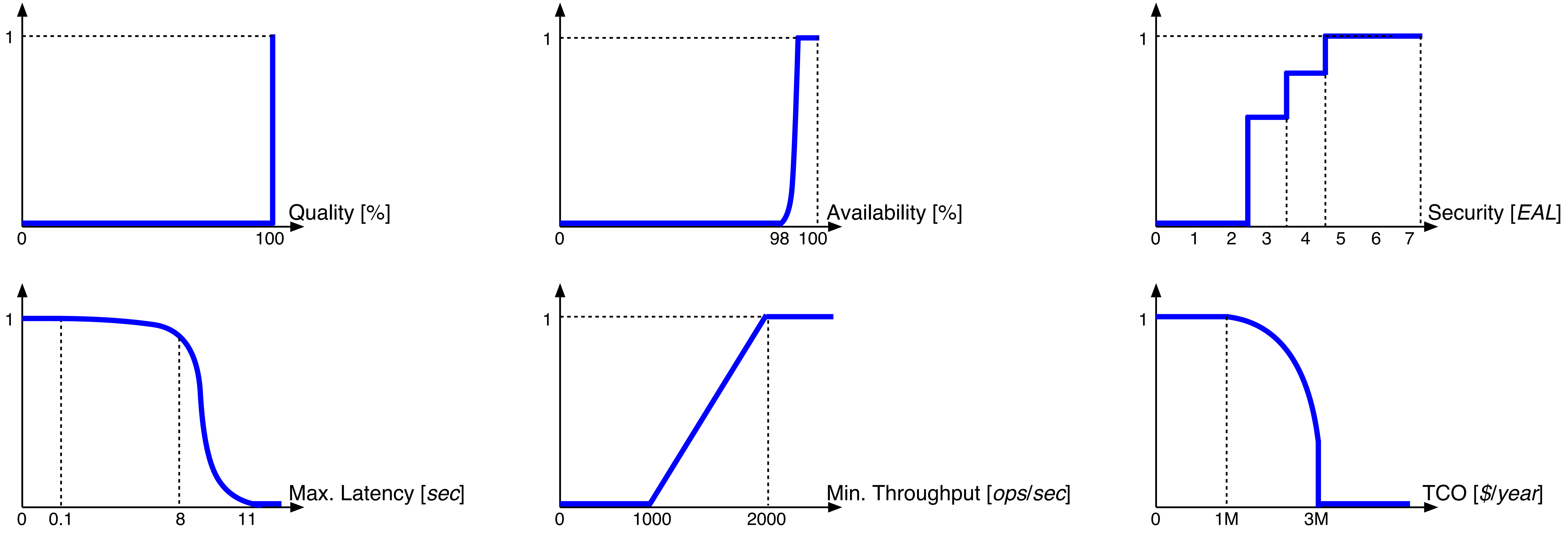
K(*ABDAGHDBBCAD...*) = 10^6+1

⇒ complex

#5 — System design is subservient to users, workloads, and technology

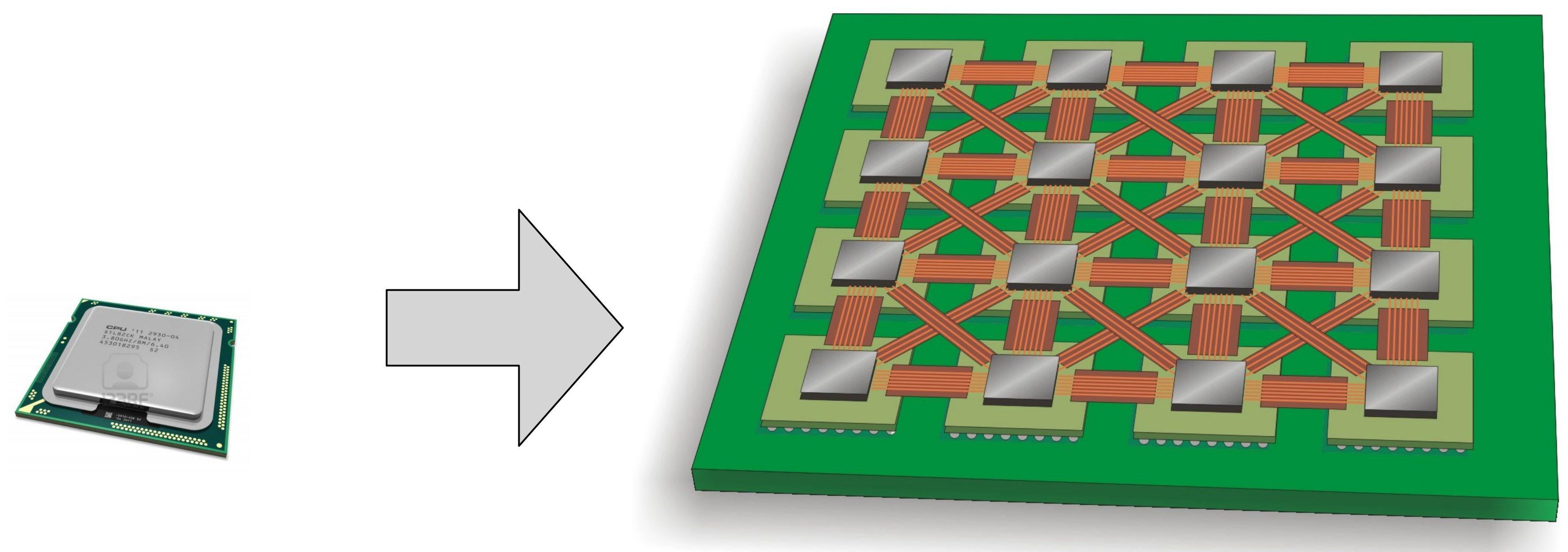
Inescapable Trade-Offs

Designing a system consists of trading off properties against each other so as to maximize the system's overall utility.



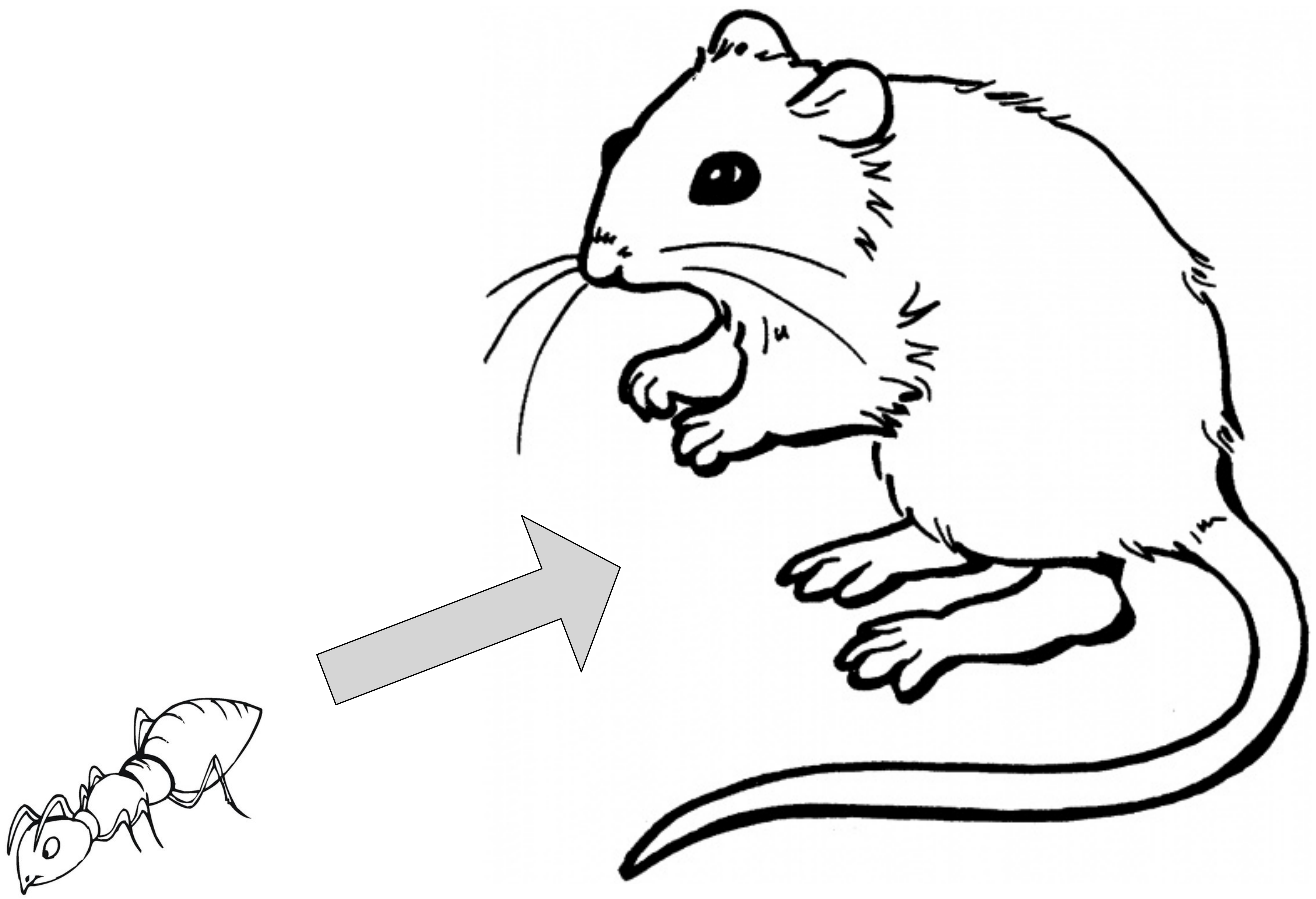
Incommensurate Scaling — chasing the bottleneck

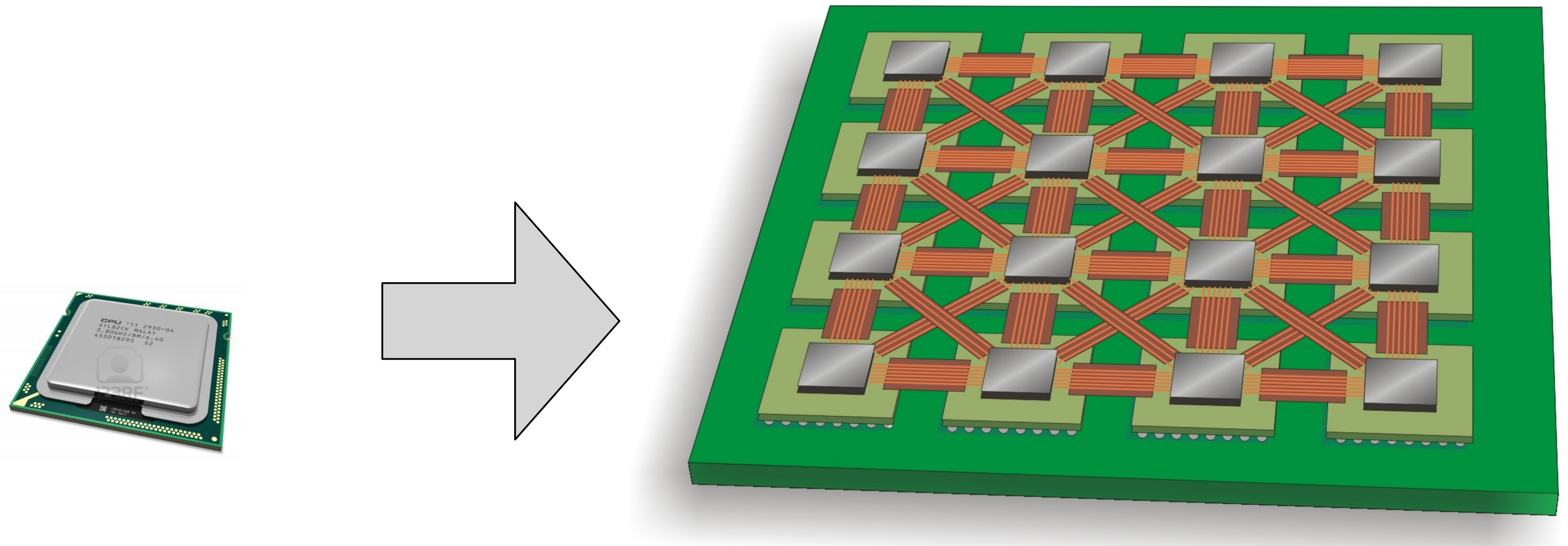
As a system increases in size or speed, different parts scale unequally, causing the system as a whole to stop working.



Incommensurate Scaling — chasing the bottleneck

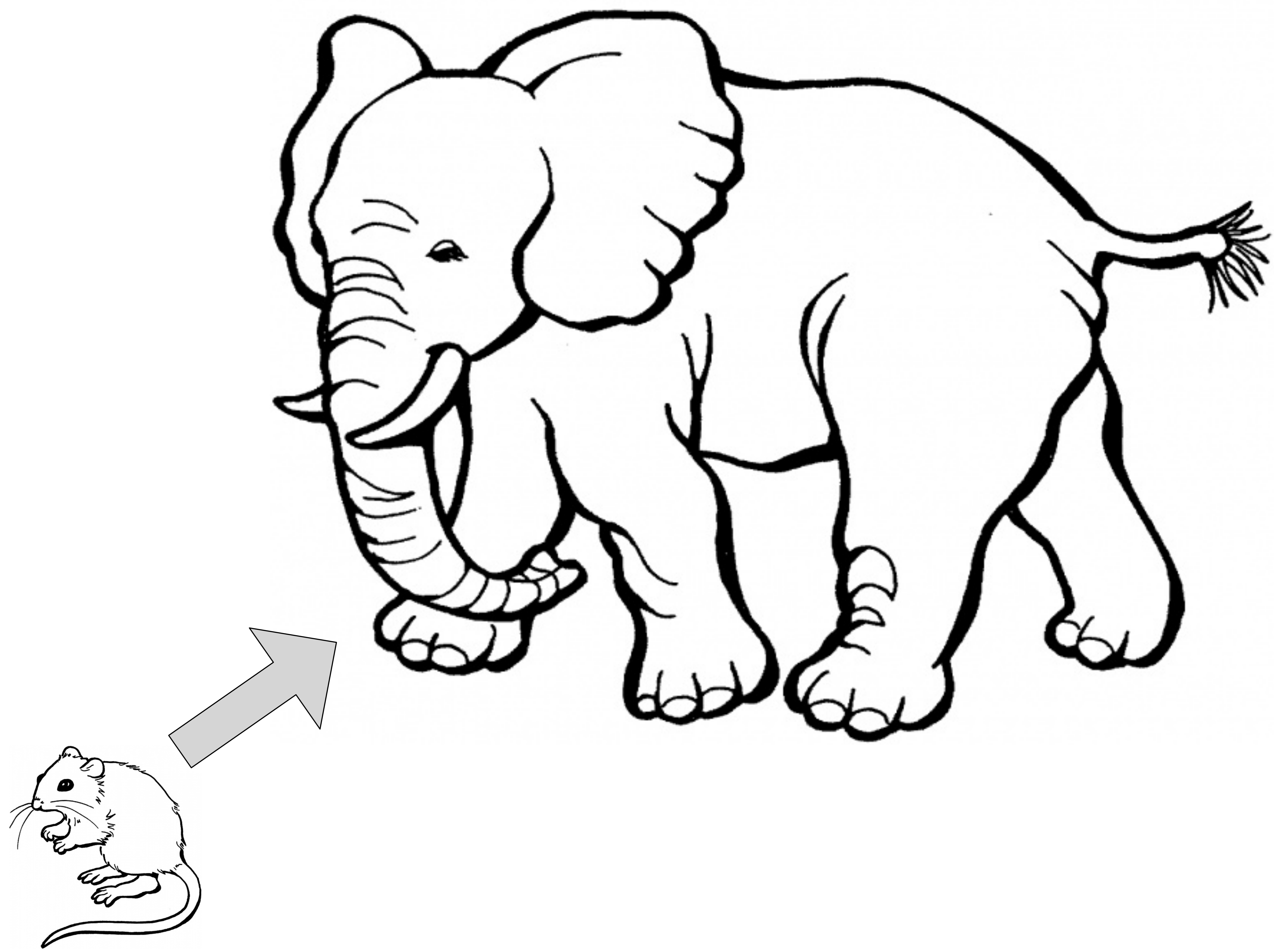
As a system increases in size or speed, different parts scale unequally, causing the system as a whole to stop working.



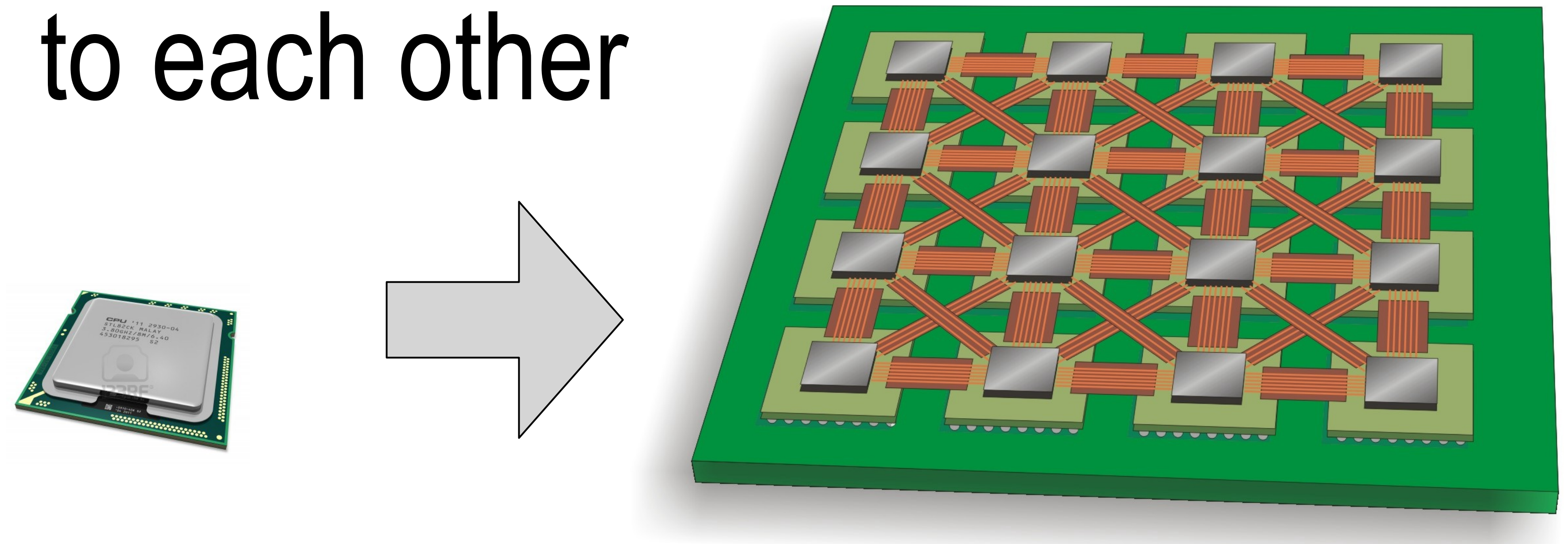


Incommensurate Scaling — chasing the bottleneck

As a system increases in size or speed, different parts scale unequally, causing the system as a whole to stop working.



- Reason:
  - Scalability of each component is described by a function
  - The order of these functions is not the same for each component => as system grows, components scale disproportionally to each other



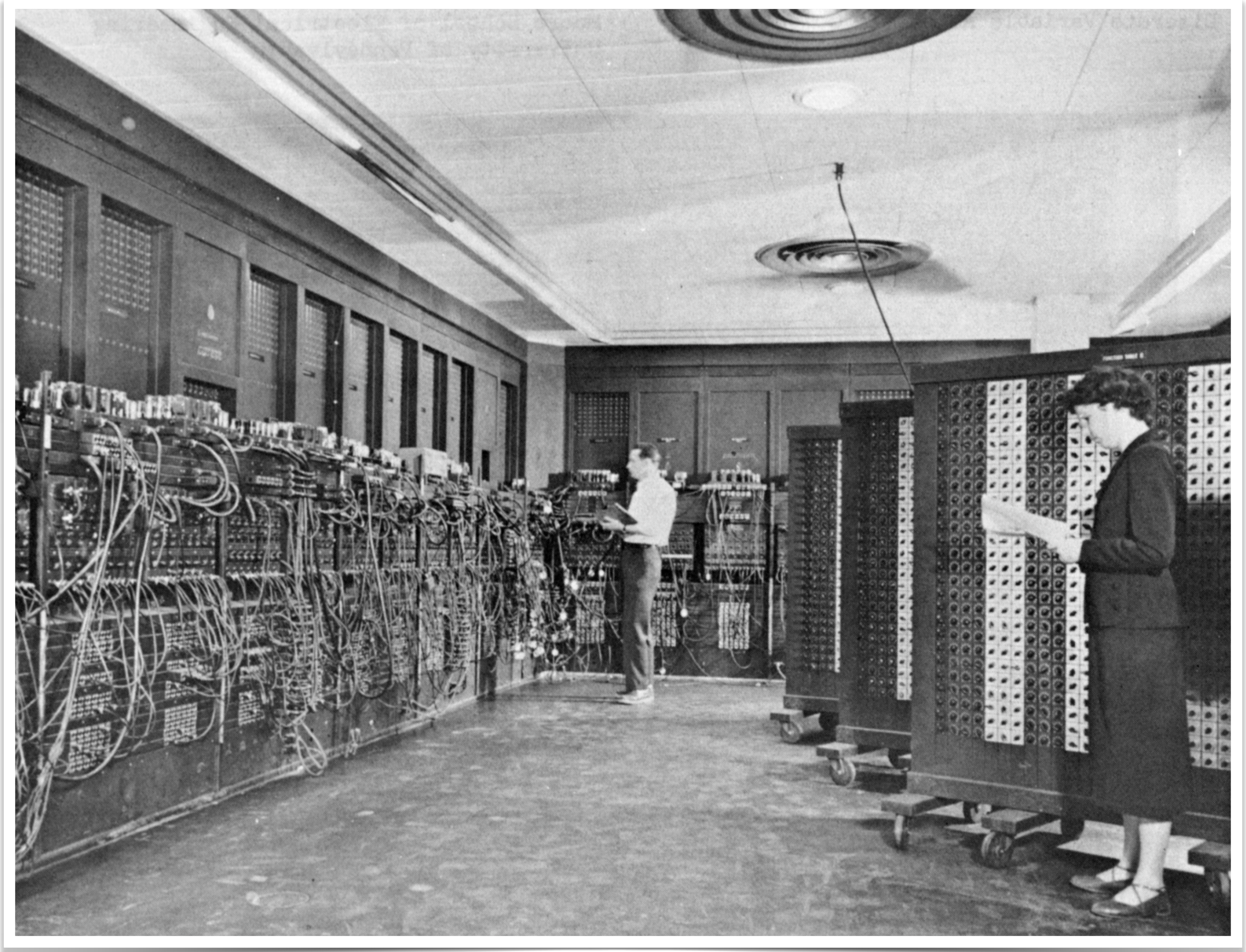
Challenges of going from components to systems

- # of behaviors of software ~ $2^{\text{code size}}$
- many components => emergent behaviors => unpredictable
- many interconnections => propagation of effects => unpredictable
- system evolution introduces exceptions and irregularity
- system is the result of trade-offs
  - *driven by its users, workloads, and technology*
  - *incommensurate scaling*

Use Modularity to Control Interactions and Propagation

Definition

- We have limited capacity to remember and disentangle details
  - *cannot reason about many things at a time => need to compartmentalize*
  - *Modularity = put things in "boxes" (components or subsystems) and treat as a unit*
- Module
  - *distinct, self-contained unit that provides a specific service or function*
  - *can be easily plugged / unplugged into different systems*
  - *often encapsulates its own state*
- Examples
  - *classes in OOP, folders in file systems, separation of src code into multiple src files, ...*



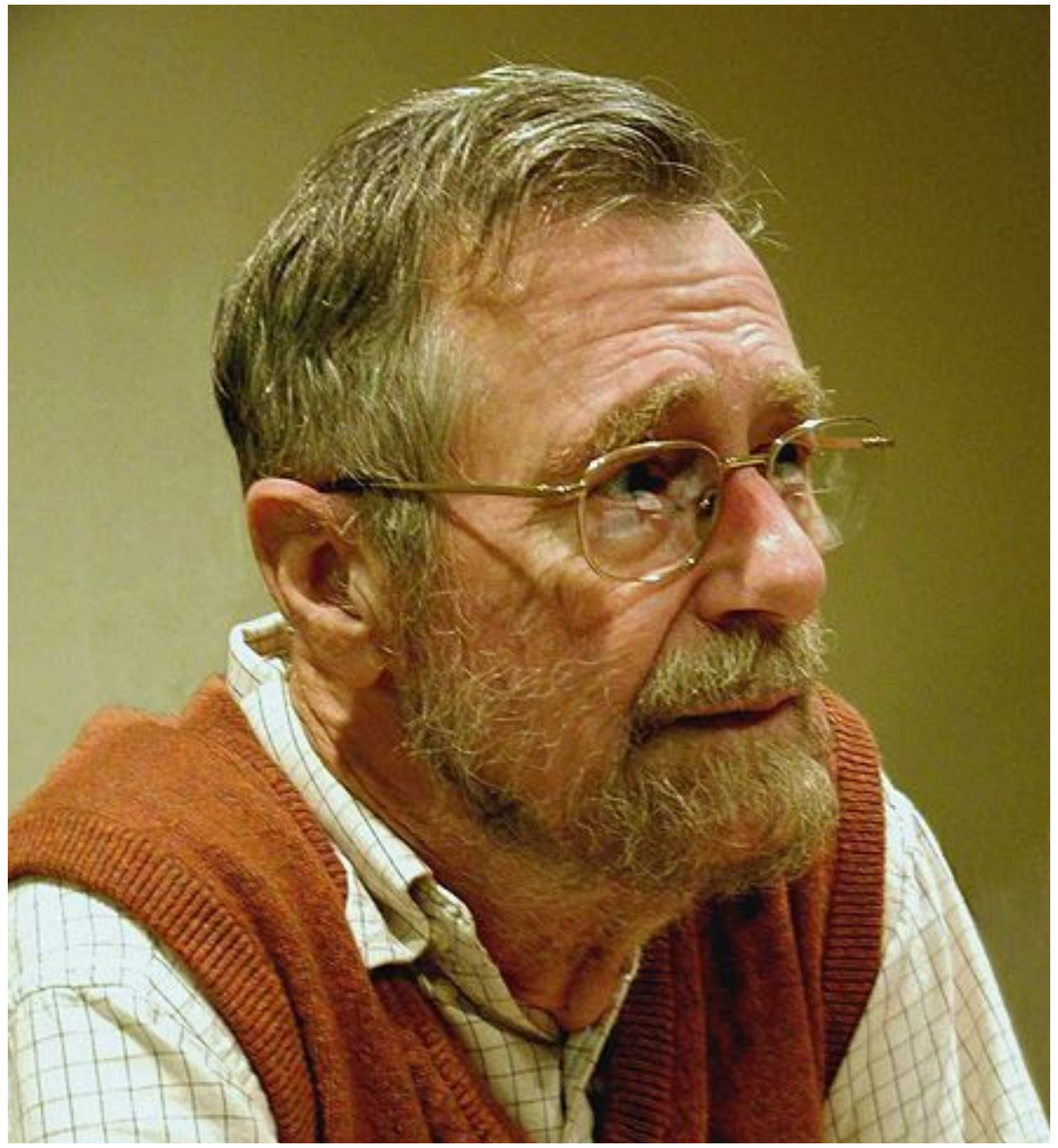
```
MAIN0001* PROGRAM TO SOLVE THE QUADRATIC EQUATION
MAIN0002        READ 10,A,B,C $
MAIN0003        DISC = B*B-4*A*C $
MAIN0004        IF (DISC) NEGA,ZERO,POSI $
MAIN0005   NEGA R = 0.0 - 0.5 * B/A $
MAIN0006        AI = 0.5 * SQRTF(0.0-DISC)/A $
MAIN0007        PRINT 11,R,AI $
MAIN0008        GO TO FINISH $
MAIN0009   ZERO R = 0.0 - 0.5 * B/A $
MAIN0010        PRINT 21,R $
MAIN0011        GO TO FINISH $
MAIN0012   POSI SD = SQRTF(DISC) $
MAIN0013        R1 = 0.5*(SD-B)/A $
MAIN0014        R2 = 0.5*(0.0-(B+SD))/A $
MAIN0015        PRINT 31,R2,R1 $
MAIN0016 FINISH STOP $
MAIN0017     10 FORMAT( 3F12.5 ) $
MAIN0018     11 FORMAT( 19H TWO COMPLEX ROOTS:, F12.5,14H PLUS OR MINUS,
MAIN0019        F12.5, 2H I ) $
MAIN0020     21 FORMAT( 15H ONE REAL ROOT:, F12.5 ) $
MAIN0021     31 FORMAT( 16H TWO REAL ROOTS:, F12.5, 5H AND , F12.5 ) $
MAIN0022        END $
```

```
MAIN0001* PROGRAM TO SOLVE THE QUADRATIC EQUATION
MAIN0002        READ 10,A,B,C $
MAIN0003        DISC = B*B-4*A*C $
MAIN0004        IF (DISC) NEGA,ZERO,POSI $
MAIN0005   NEGA R = 0.0 - 0.5 * B/A $
MAIN0006        AI = 0.5 * SQRTF(0.0-DISC)/A $
MAIN0007        PRINT 11,R,AI $
MAIN0008        GO TO FINISH $
MAIN0009   ZERO R = 0.0 - 0.5 * B/A $
MAIN0010        PRINT 21,R $
MAIN0011        GO TO FINISH $
MAIN0012   POSI SD = SQRTF(DISC) $
MAIN0013        R1 = 0.5*(SD-B)/A $
MAIN0014        R2 = 0.5*(0.0-(B+SD))/A $
MAIN0015        PRINT 31,R2,R1 $
MAIN0016 FINISH STOP $
MAIN0017     10 FORMAT( 3F12.5 ) $
MAIN0018     11 FORMAT( 19H TWO COMPLEX ROOTS:, F12.5,14H PLUS OR MINUS,
MAIN0019        F12.5, 2H I ) $
MAIN0020     21 FORMAT( 15H ONE REAL ROOT:, F12.5 ) $
MAIN0021     31 FORMAT( 16H TWO REAL ROOTS:, F12.5, 5H AND , F12.5 ) $
MAIN0022        END $
```

```
MAIN0001* PROGRAM TO SOLVE THE QUADRATIC EQUATION
MAIN0002        READ 10,A,B,C $
MAIN0003        DISC = B*B-4*A*C $
MAIN0004        IF (DISC) NEGA,ZERO,POSI $
MAIN0005   NEGA R = 0.0 - 0.5 * B/A $
MAIN0006        AI = 0.5 * SQRTF(0.0-DISC)/A $
MAIN0007        PRINT 11,R,AI $
MAIN0008        GO TO FINISH $
MAIN0009   ZERO R = 0.0 - 0.5 * B/A $
MAIN0010        PRINT 21,R $
MAIN0011        GO TO FINISH $
MAIN0012   POSI SD = SQRTF(DISC) $
MAIN0013        R1 = 0.5*(SD-B)/A $
MAIN0014        R2 = 0.5*(0.0-(B+SD))/A $
MAIN0015        PRINT 31,R2,R1 $
MAIN0016 FINISH STOP $
MAIN0017     10 FORMAT( 3F12.5 ) $
MAIN0018     11 FORMAT( 19H TWO COMPLEX ROOTS:, F12.5,14H PLUS OR MINUS,
MAIN0019        F12.5, 2H I ) $
MAIN0020     21 FORMAT( 15H ONE REAL ROOT:, F12.5 ) $
MAIN0021     31 FORMAT( 16H TWO REAL ROOTS:, F12.5, 5H AND , F12.5 ) $
MAIN0022        END $
```

```
MAIN0001* PROGRAM TO SOLVE THE QUADRATIC EQUATION
MAIN0002          READ 10,A,B,C $
MAIN0003          DISC = B*B-4*A*C $
MAIN0004          IF (DISC) NEGA,ZERO,POSI $
MAIN0005    NEGA R = 0.0 - 0.5 * B/A $
MAIN0006          AI = 0.5 * SQRTF(0.0-DISC)/A $
MAIN0007          PRINT 11,R,AI $
MAIN0008          GO TO FINISH $
MAIN0009    ZERO R = 0.0 - 0.5 * B/A $
MAIN0010          PRINT 21,R $
MAIN0011          GO TO FINISH $
MAIN0012    POSI SD = SQRTF(DISC) $
MAIN0013          R1 = 0.5*(SD-B)/A $
MAIN0014          R2 = 0.5*(0.0-(B+SD))/A $
MAIN0015          PRINT 31,R2,R1 $
MAIN0016 FINISH STOP $
MAIN0017      10 FORMAT( 3F12.5 ) $
MAIN0018      11 FORMAT( 19H TWO COMPLEX ROOTS:, F12.5,14H PLUS OR MINUS,
MAIN0019          F12.5, 2H I ) $
MAIN0020      21 FORMAT( 15H ONE REAL ROOT:, F12.5 ) $
MAIN0021      31 FORMAT( 16H TWO REAL ROOTS:, F12.5, 5H AND , F12.5 ) $
MAIN0022          END $
```

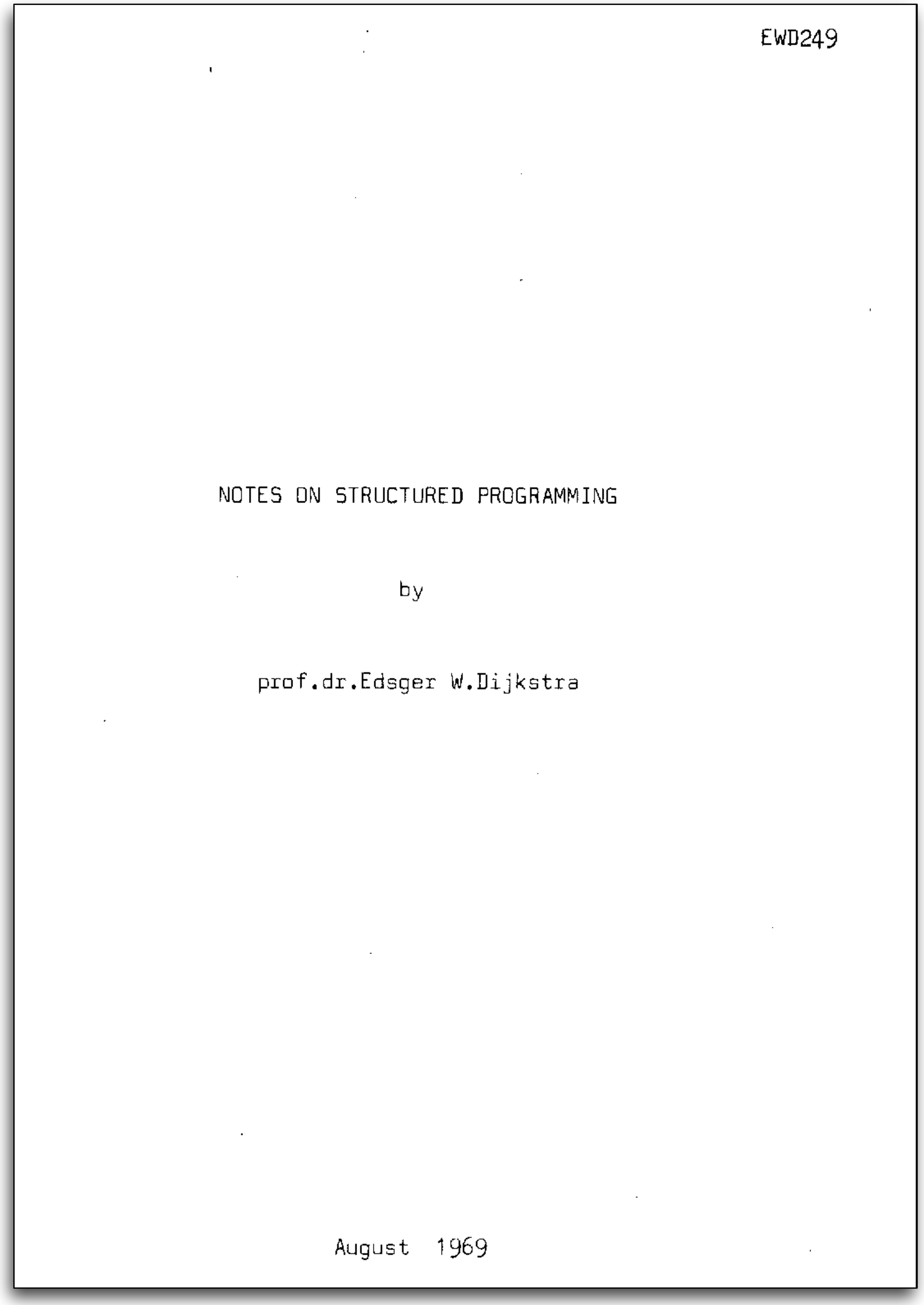
Structured Programming



Edsger Dijkstra

The competent programmer is fully aware of the strictly limited size of his own skull and therefore approaches the programming task in full humility.

Structured Programming


EWD249

NOTES ON STRUCTURED PROGRAMMING

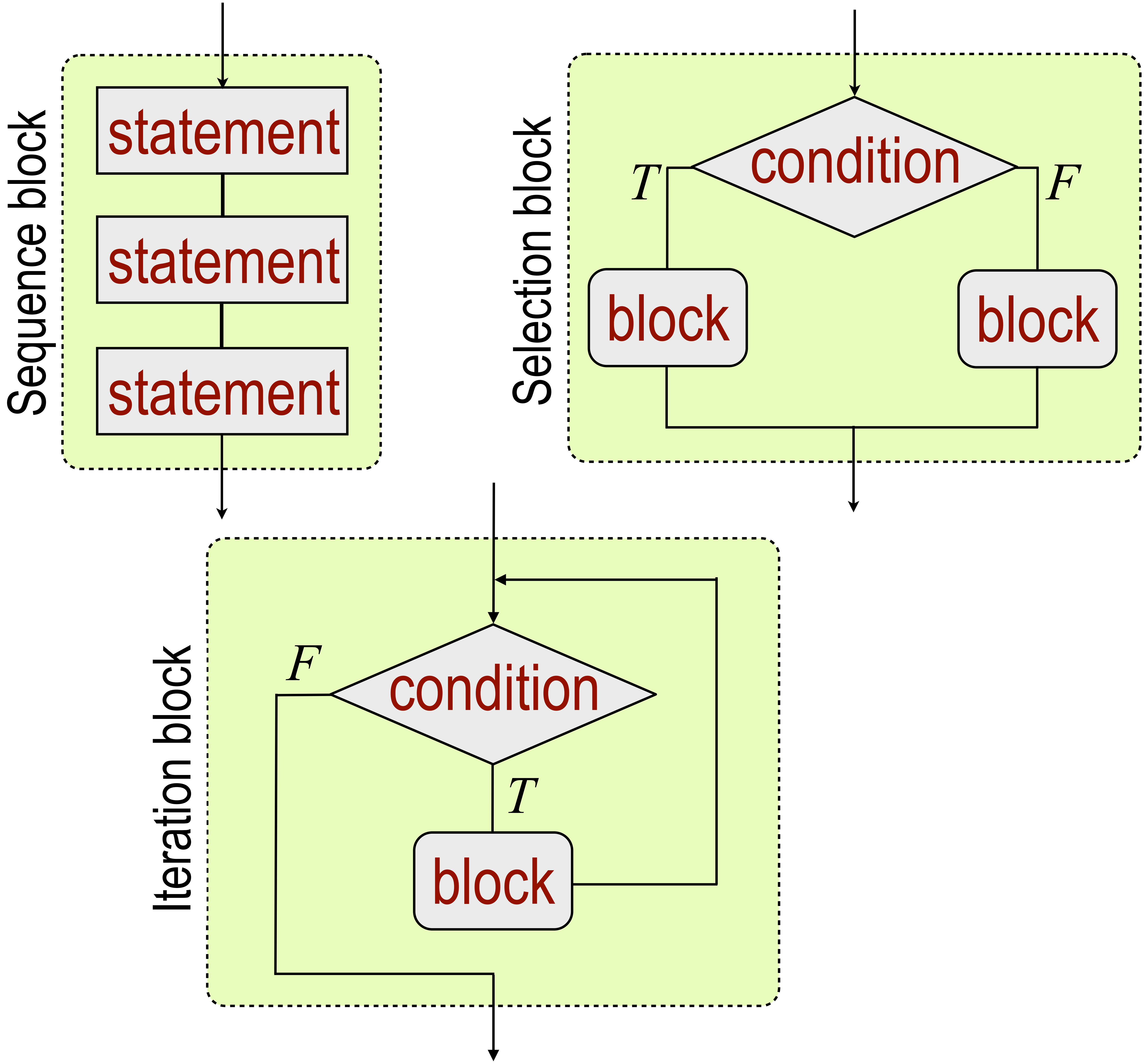
by

prof.dr.Edsger W.Dijkstra

August 1969

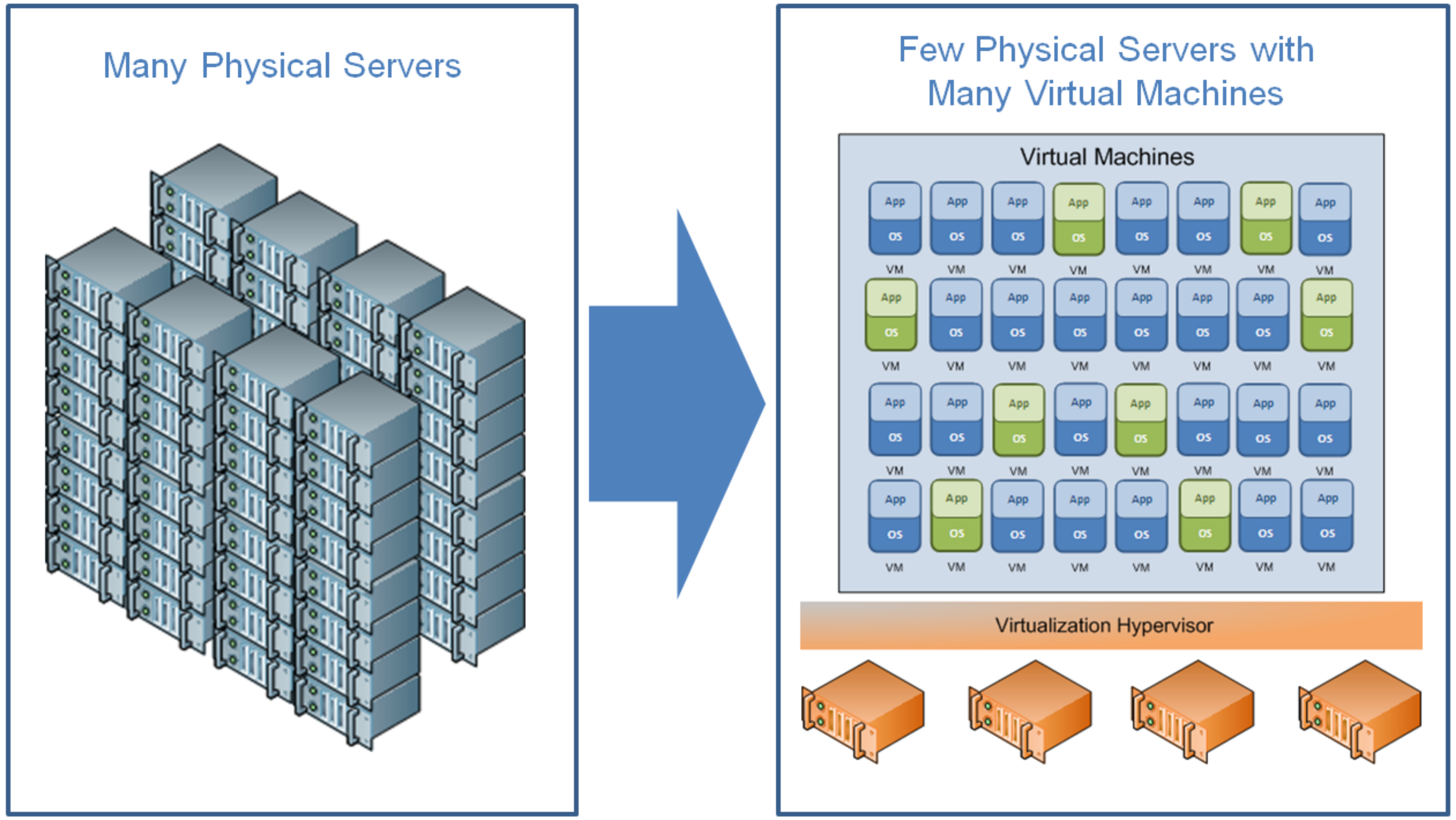
Structured Programming

- Three basic constructs
  - *single-entry / single-exit control constructs*
  - *sequence, selection, iteration*
- Structured program
  - *ordered, disciplined, doesn't jump around unpredictably*
  - *can read easily and reason about ⇒ higher quality*



Modularity Through Virtualization

- Virtual machines

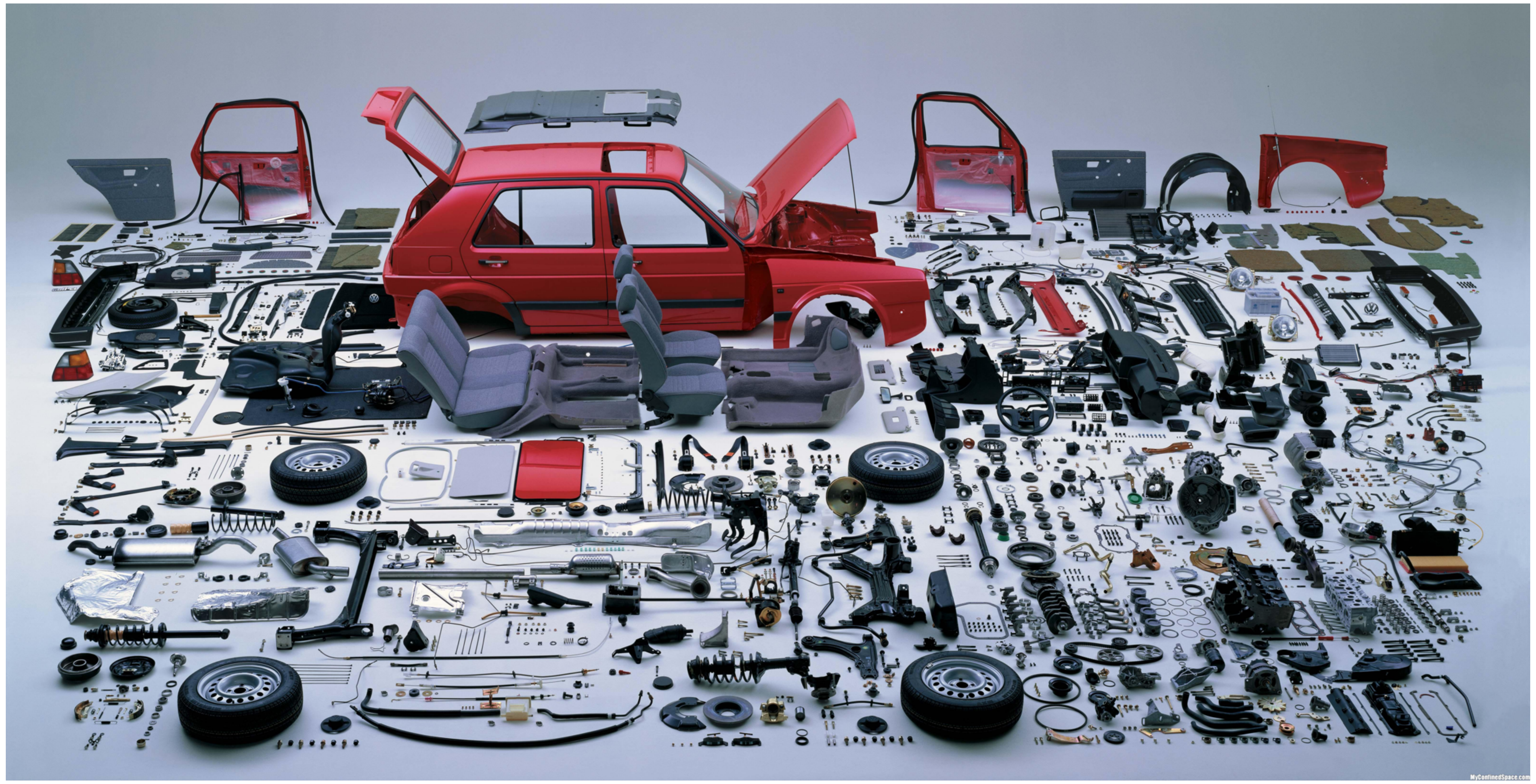


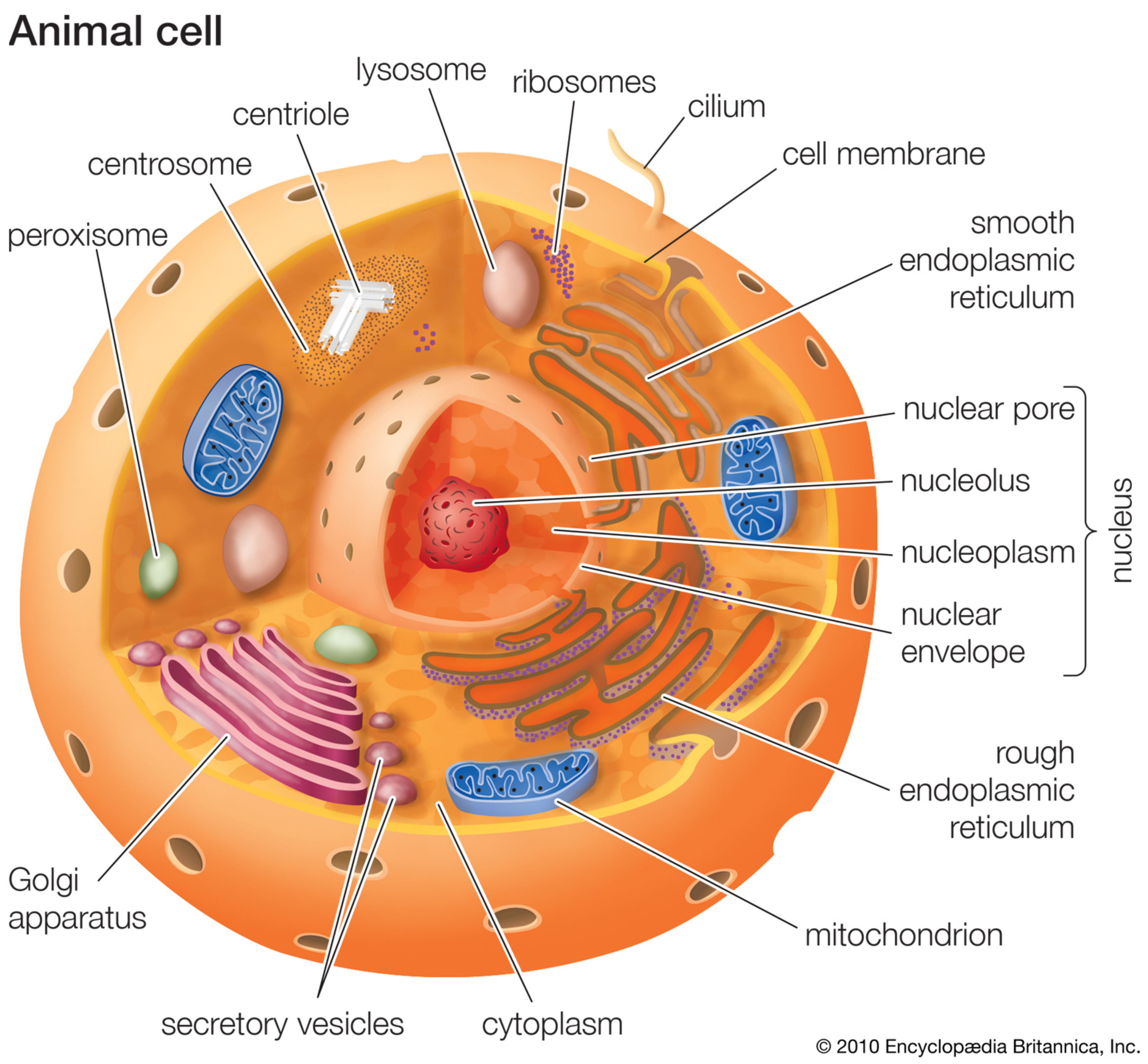
- Containers
  - *Docker, LXC, Podman, ...*
  - *Zones (Solaris)*
  - *Virtual private servers (OpenVZ)*
  - *Partitions, virtual environments*
  - *Virtual kernels (DragonFly BSD)*
  - *Jails (FreeBSD jail, chroot)*

What is Modularity ?

- Isolate behavior into “boxes”
- Controlled entry/exit points
  - *replace components without affecting rest of system*
- Criterion
  - *Interdependence within modules + independence across modules*

For more insights, see
C. Y. Baldwin and K. B. Clark, Design Rules: The power of modularity, The MIT Press, 2000




Animal cell
lysosome
ribosomes
centriole
cilium
centrosome
cell membrane
peroxisome
smooth endoplasmic reticulum
nuclear pore
nucleolus
nucleoplasm
nuclear envelope
nucleus
rough endoplasmic reticulum
Golgi apparatus
mitochondrion
secretory vesicles
cytoplasm
© 2010 Encyclopædia Britannica, Inc.

Use Abstraction to Simplify and Regularize Behavior

Modularity

Modularity

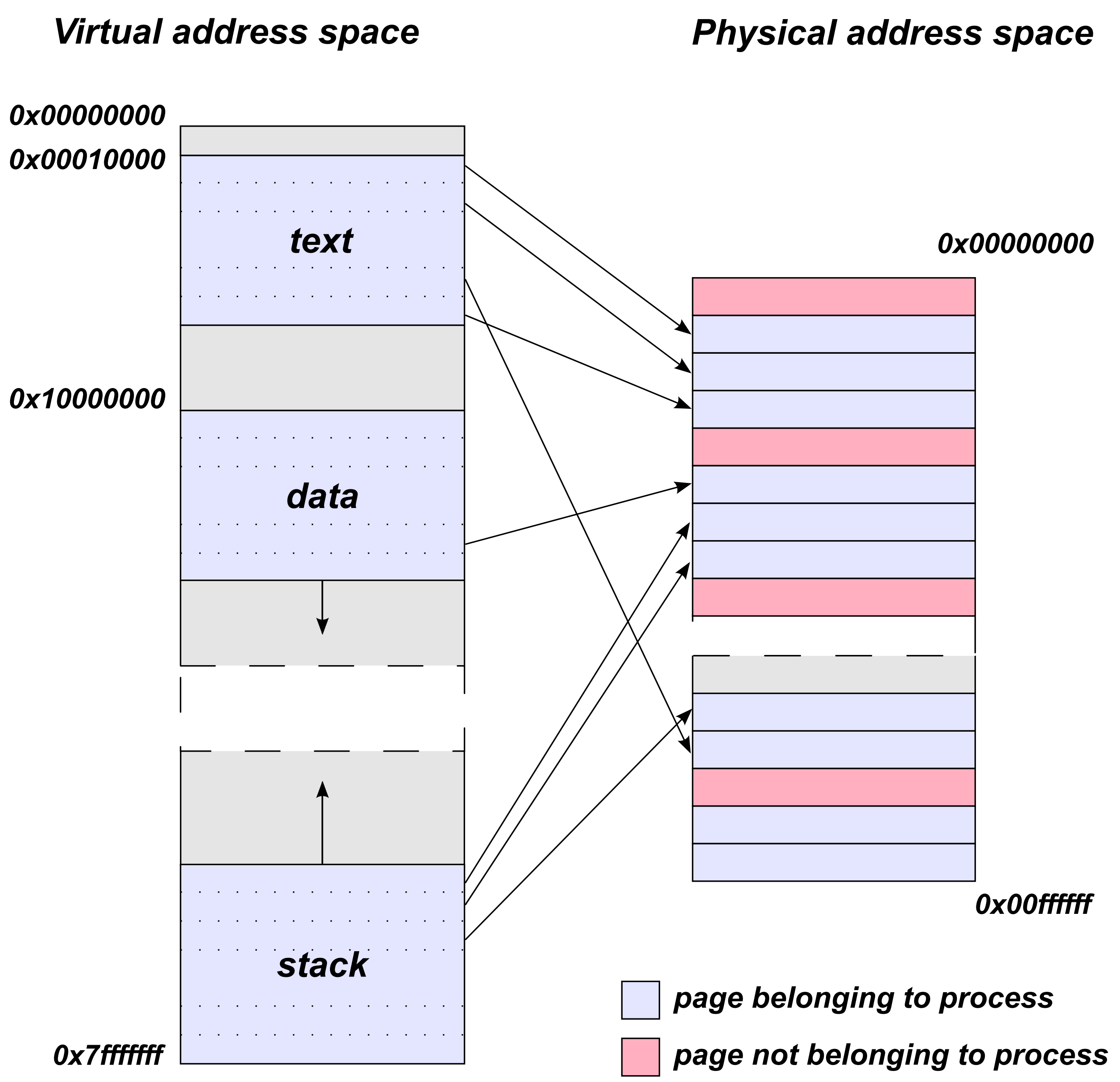


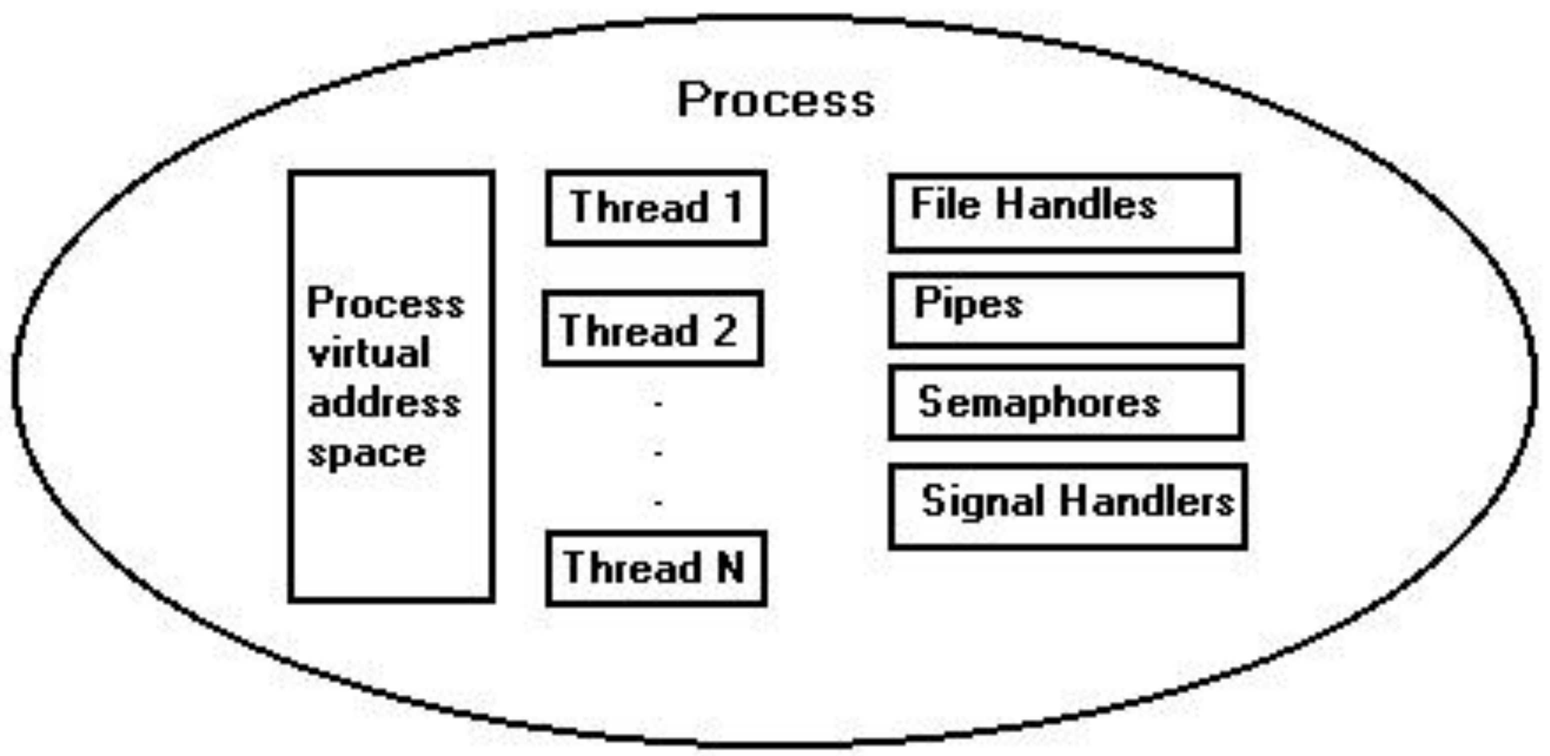

Examples of Abstractions in Operating Systems

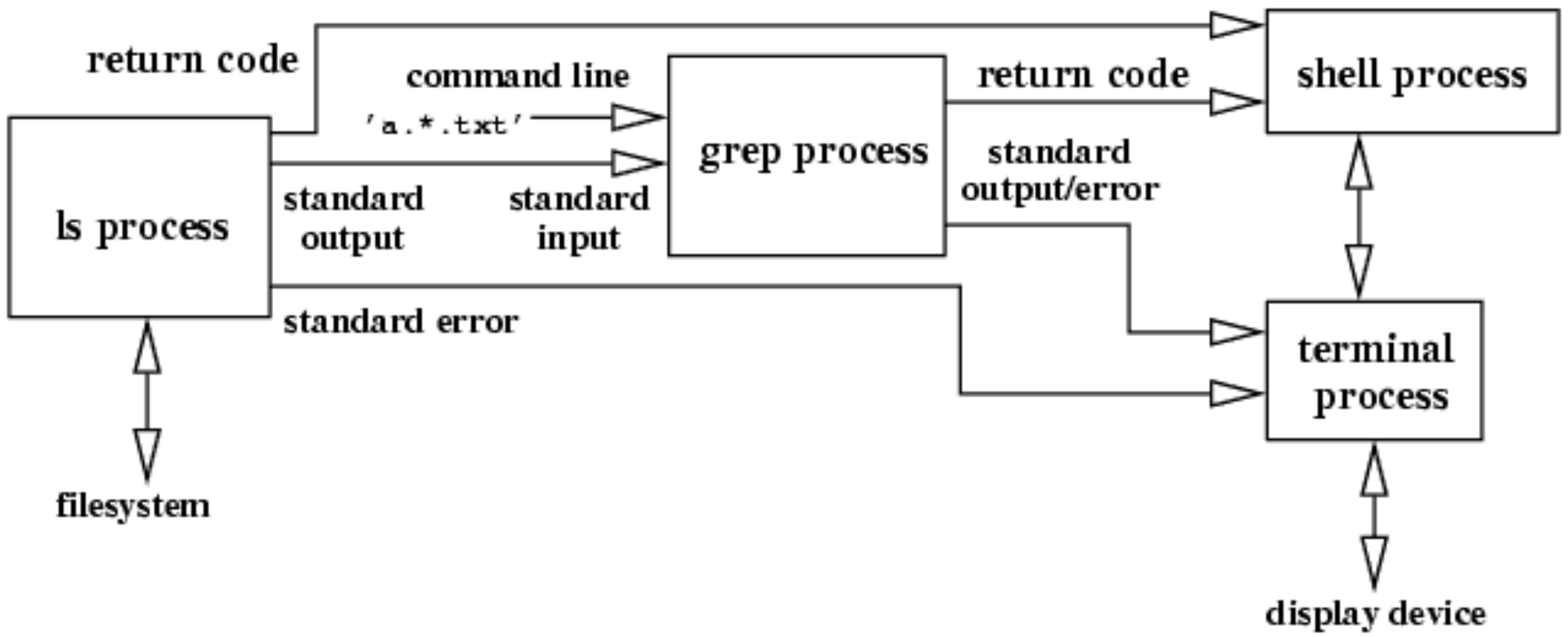
- Virtual address space
- Process
- Pipe
- Filesystem
- ...

Examples of Abstractions in Operating Systems

- Virtual address space
- Process
- Pipe
- Filesystem
- ...

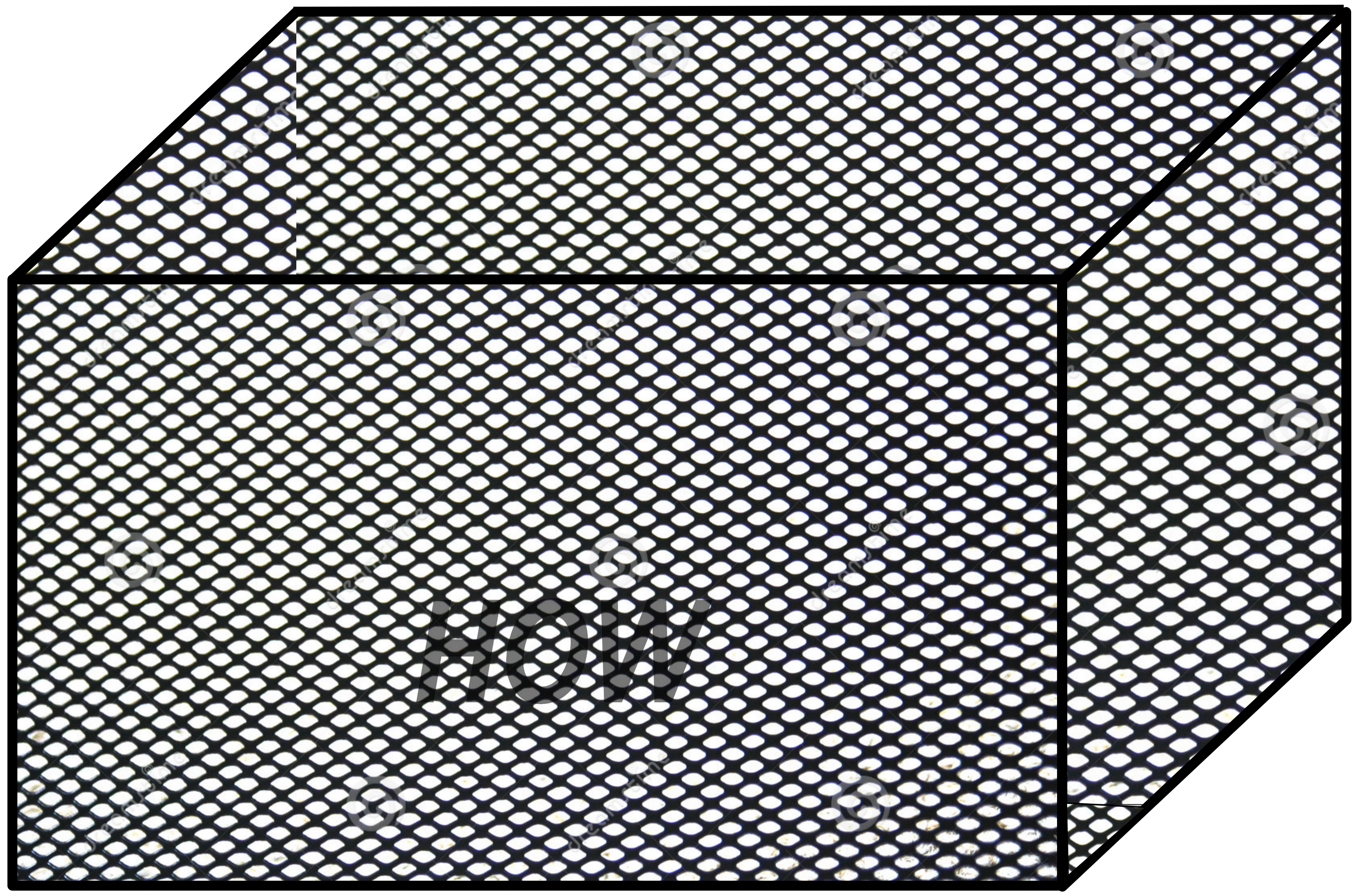






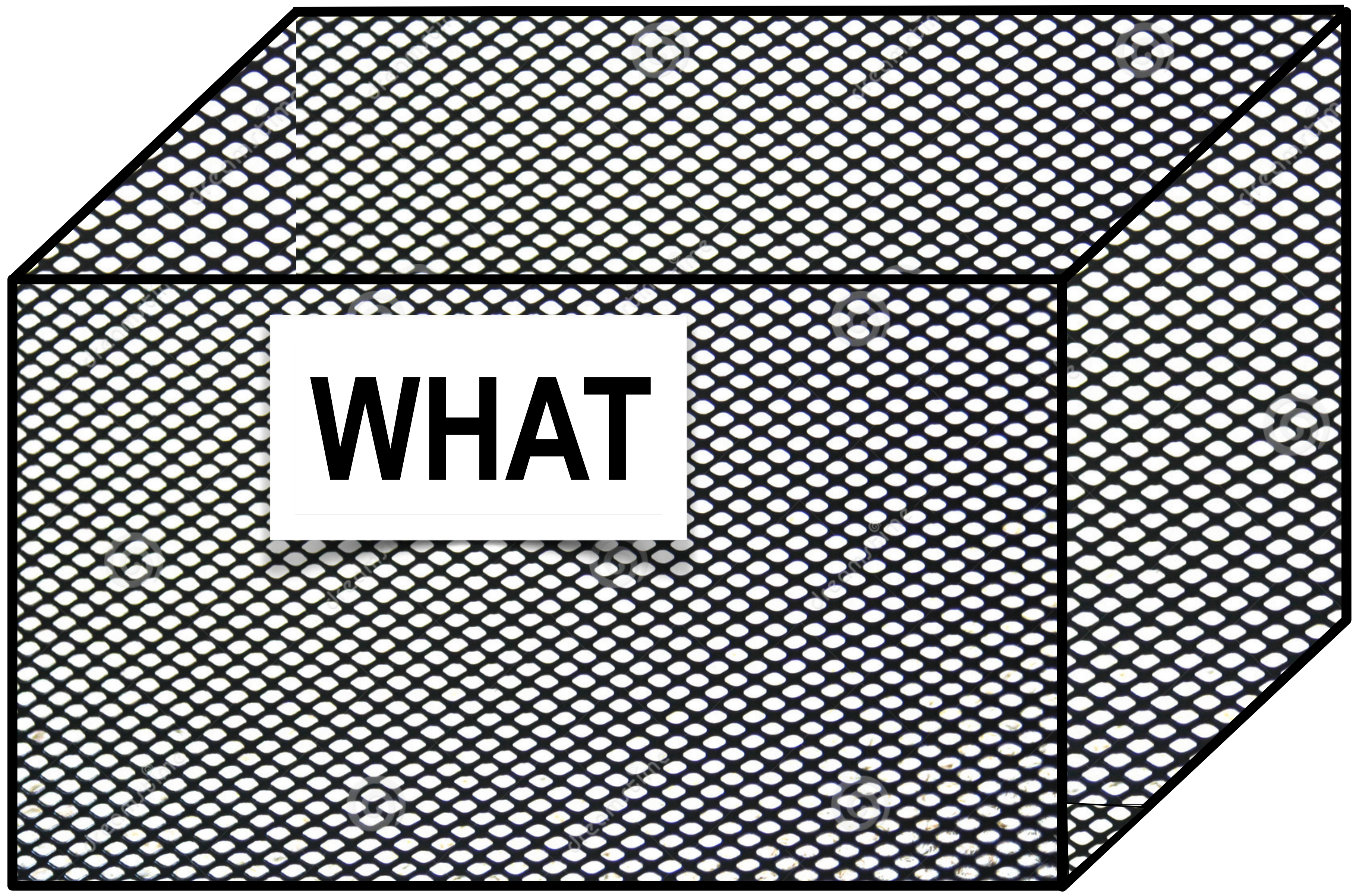
Abstraction = Interface + Modularity

- Specifies “what” a component/subsystem does
- Together with modularity,
  it separates “what” from “how”
  => abstraction



Abstraction = Interface + Modularity

- Specifies “what” a component/subsystem does
- Together with modularity,
  it separates “what” from “how”
  => abstraction



Examples of Abstractions in Programming Languages

```
(defun queue-get (queue &optional (default nil))
  "Get an element from QUEUE"
  (check-type queue queue)
  (let ((get (queue-get-ptr queue))
        (put (queue-put-ptr queue)))
    (if (= get put)
        default
        (prog1
          (svref (queue-elements queue) get)
          (setf (queue-get-ptr queue) (queue-next queue get))))))

(defun queue-put (queue element)
  "Store ELEMENT in the QUEUE and return T on success or NIL on failure."
  (check-type queue queue)
  (let* ((get (queue-get-ptr queue))
         (put (queue-put-ptr queue))
         (next (queue-next queue put)))
    (unless (= get next)
      (setf (svref (queue-elements queue) put) element)
      (setf (queue-put-ptr queue) next)
      t)))
```

- Routines
  - *function, procedure, thread, etc.*
- Lambda functions
  - *a.k.a. anonymous functions*
- Abstract data types
- Objects in OOP

Examples of Abstractions in Programming Languages

```
(defun myreverse (list)
  (let ((reverse-acc #'
         (lambda (func list acc)
           (let ((head (car list))
                 (rest (cdr list)))
             (cond (head
                    (cond ((atom head)
                           (funcall func func rest (cons head acc)))
                          (t
                           (funcall func func rest
                                    (cons
                                     (funcall func func head nil)
                                     acc)))))
                   (t
                    acc))))))
    (funcall reverse-acc reverse-acc list nil)))
```

```
object CurryTest extends Application {
    def filter(xs: List[Int], p: Int => Boolean): List[Int]
        if (xs.isEmpty) xs
        else if (p(xs.head)) xs.head :: filter(xs.tail, p)
        else filter(xs.tail, p)

    def modN(n: Int)(x: Int) = ((x % n) == 0)

    val nums = List(1, 2, 3, 4, 5, 6, 7, 8)
    println(filter(nums, modN(2)))
    println(filter(nums, modN(3)))
}
```

- Routines
  - *function, procedure, thread, etc.*
- Lambda functions
  - *a.k.a. anonymous functions*
- Abstract data types
- Objects in OOP

Examples of Abstractions in Programming Languages

```
class Stack:
    def __init__(self):
        self._list = []

    def put(self, item):
        self._list.append(item)

    def remove(self):
        if len(self._list) > 0:
            self._list.pop()

    def item(self):
        if len(self._list) > 0:
            return self._list[-1]

    def empty(self):
        return len(self._list) == 0
```

- Routines
  - *function, procedure, thread, etc.*
- Lambda functions
  - *a.k.a. anonymous functions*
- Abstract data types
- Objects in OOP

Examples of Abstractions in Programming Languages

```
class CefMainDelegate : public content::ContentMainDelegate {
 public:
  explicit CefMainDelegate(CefRefPtr<CefApp> application);
  virtual ~CefMainDelegate();

  virtual bool BasicStartupComplete(int* exit_code);
  virtual void PreSandboxStartup();
  virtual int RunProcess(
      const std::string& process_type,
      const content::MainFunctionParams& main_function_params);
  virtual void ProcessExiting(const std::string& process_type);
  virtual content::ContentBrowserClient* CreateContentBrowserClient();
  virtual content::ContentRendererClient*
      CreateContentRendererClient();

  void ShutdownBrowser();

  CefContentBrowserClient* browser_client() { return browser_client_.get(); }
  CefContentClient* content_client() { return &content_client_; }

 private:
  void InitializeResourceBundle();

  scoped_ptr<content::BrowserMainRunner> browser_runner_;
  scoped_ptr<base::Thread> ui_thread_;

  scoped_ptr<CefContentBrowserClient> browser_client_;
  scoped_ptr<CefContentRendererClient> renderer_client_;
  CefContentClient content_client_;
};
```

- Routines
  - *function, procedure, thread, etc.*
- Lambda functions
  - *a.k.a. anonymous functions*
- Abstract data types
- Objects in OOP

How To Modularize & Abstract?

- Abstraction + modularity often considered the same (but is not)
- Modularize along natural (effective) boundaries
  - *Few interactions between modules*
  - *Few propagations of effects*
- Must be able to interact with module without knowing internal details
- Beware of ability to truly encapsulate
  - *E.g., use of hw protection for address spaces vs. objects in C++*

Abstraction = Module + Interface

Interface = contract between
a module and the rest

bind

```
public void bind(SocketAddress bindpoint)
          throws IOException
```

Binds the socket to a local address.

If the address is `null`, then the system will pick up an ephemeral port and a valid local address to bind the socket.

Parameters:

`bindpoint` - the `SocketAddress` to bind to

Throws:

`IOException` - if the bind operation fails, or if the socket is already bound.

`IllegalArgumentException` - if bindpoint is a SocketAddress subclass not supported by this socket

Since:

1.4

See Also:

`isBound()`

connect

```
public void connect(SocketAddress endpoint)
             throws IOException
```

Connects this socket to the server.

Parameters:

`endpoint` - the `SocketAddress`

Throws:

`IOException` - if an error occurs during the connection

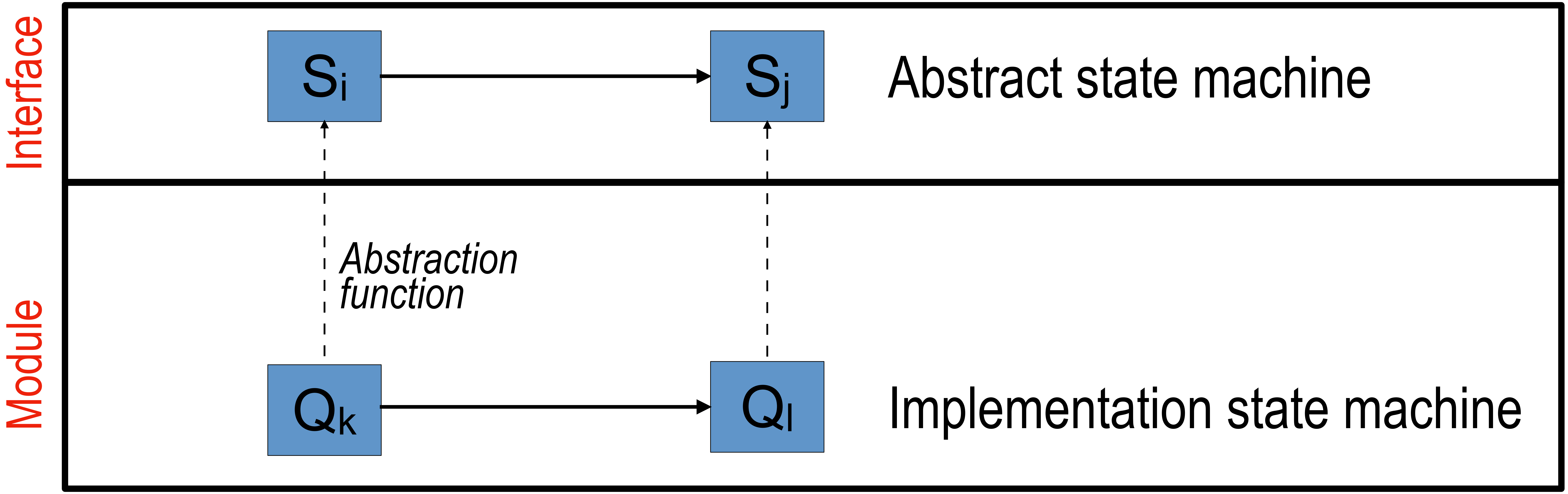
`IllegalBlockingModeException` - if this socket has an associated channel, and the channel is in non-blocking mode

`IllegalArgumentException` - if endpoint is null or is a SocketAddress subclass not supported by this socket

Since:

1.4

Implementation "simulates" the Abstraction



Properties of Abstractions

Well known properties of good abstractions

- Information hiding
- Completeness
- Consistency
- Separation of concerns
- Generality & Reusability
- Extensibility
- Single responsibility & Orthogonality
- Composability
- Efficiency

Leaky Abstractions

> All non-trivial abstractions, to some degree, are leaky.
>
> *(Joel Spolsky)*

https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/

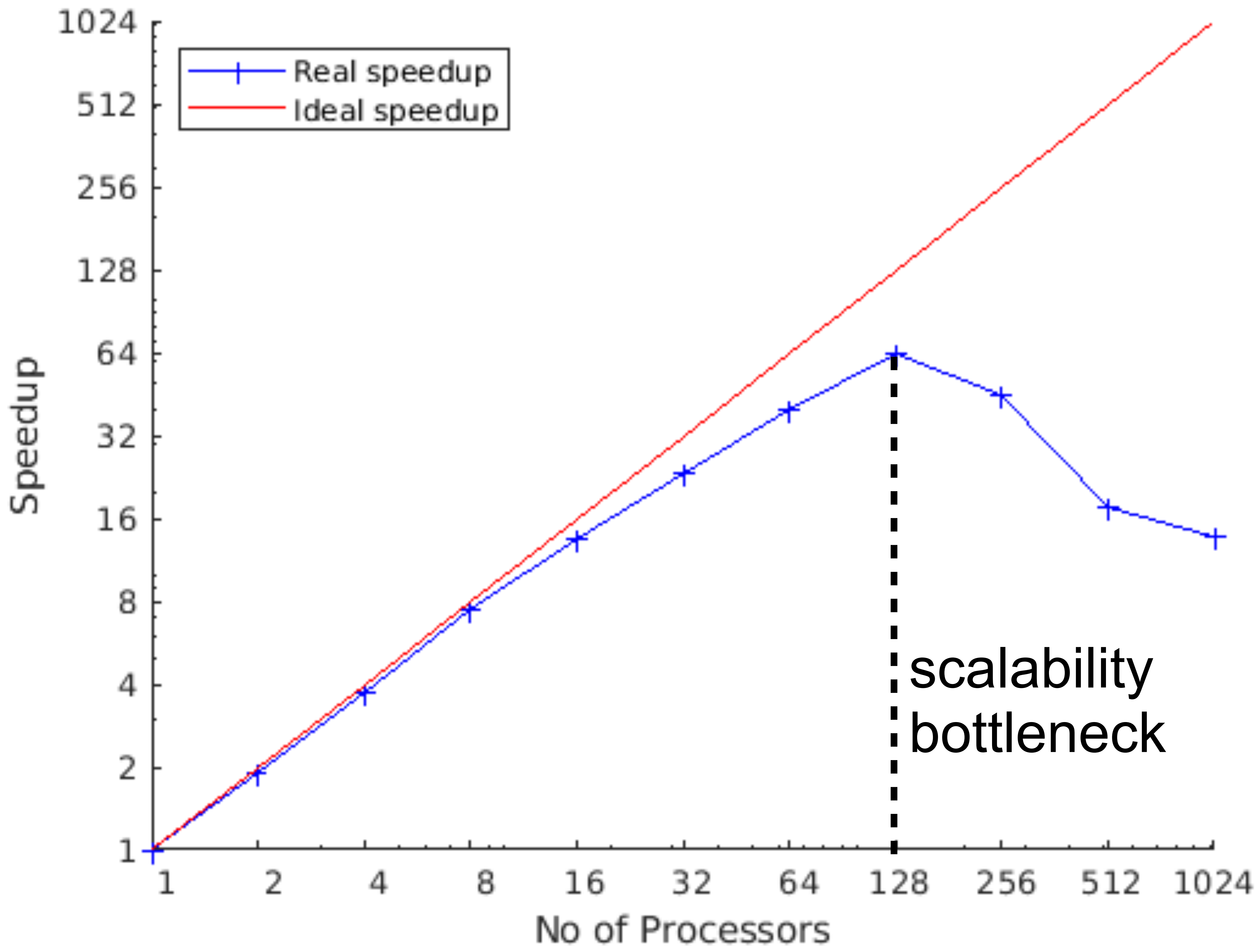
The Scalable Commutativity Rule

T. Clements et al, *The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors*, SOSP 2013

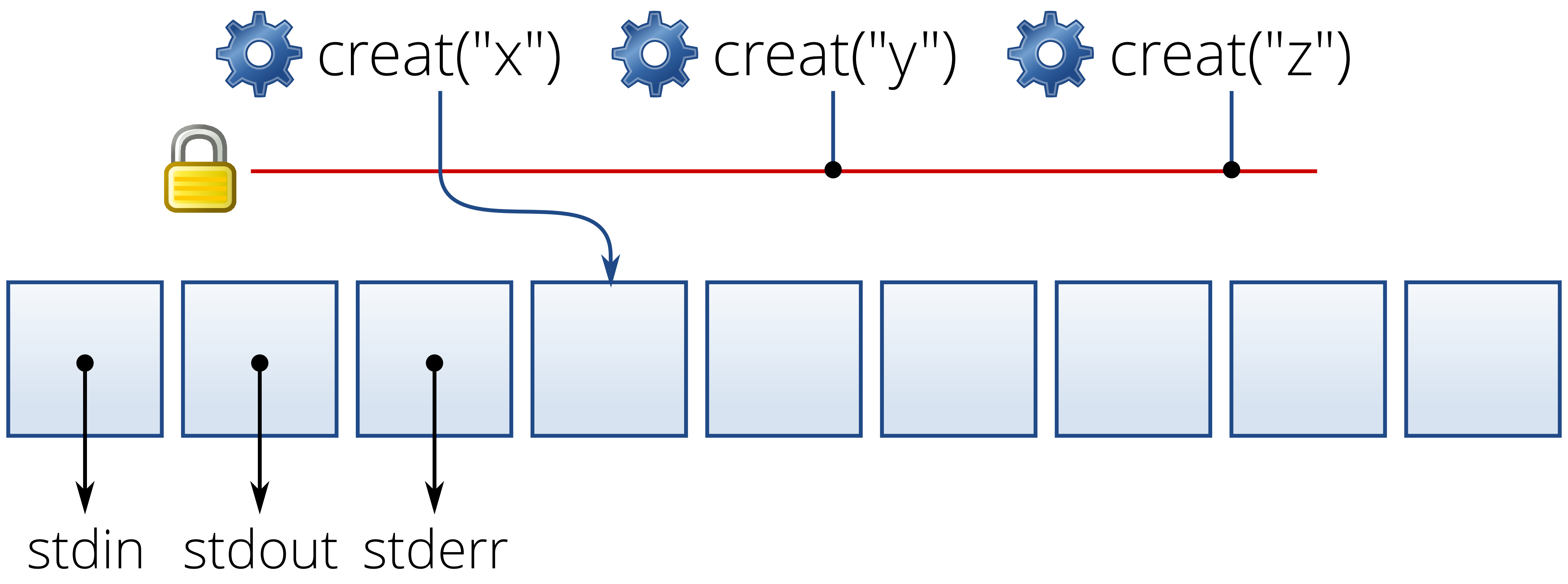
What is scalability ?

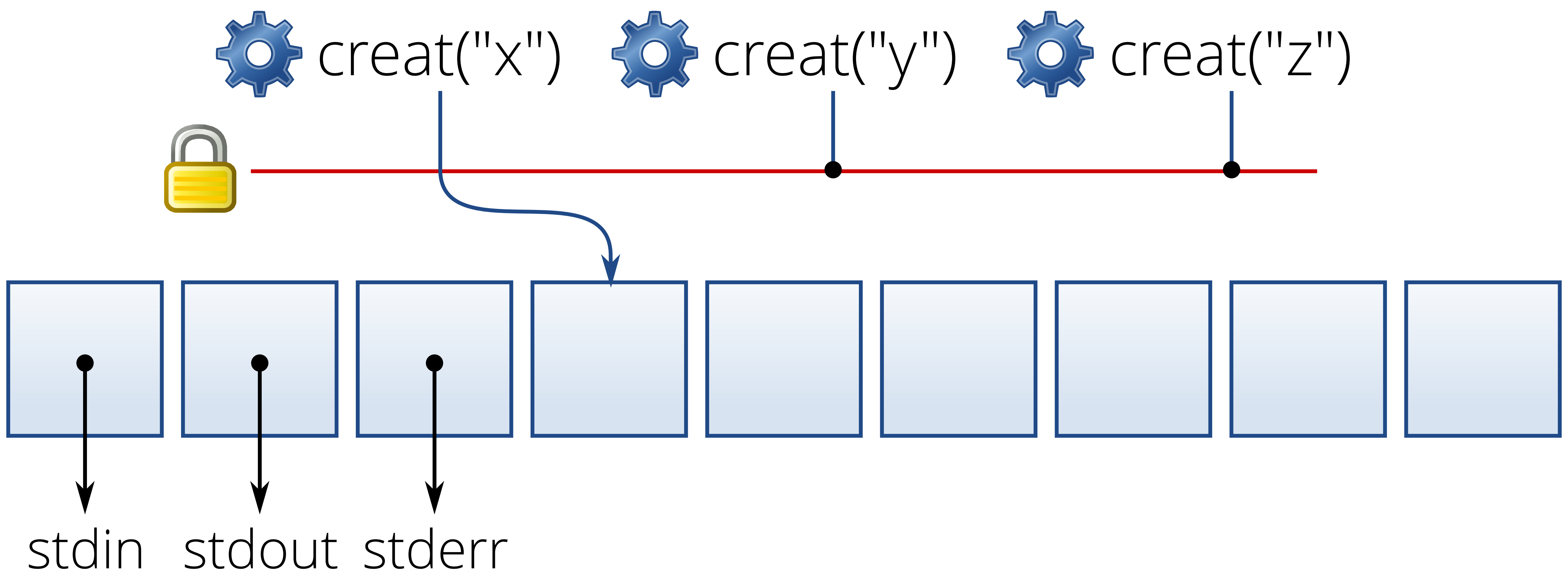
Ability to perform additional work given greater hardware resources

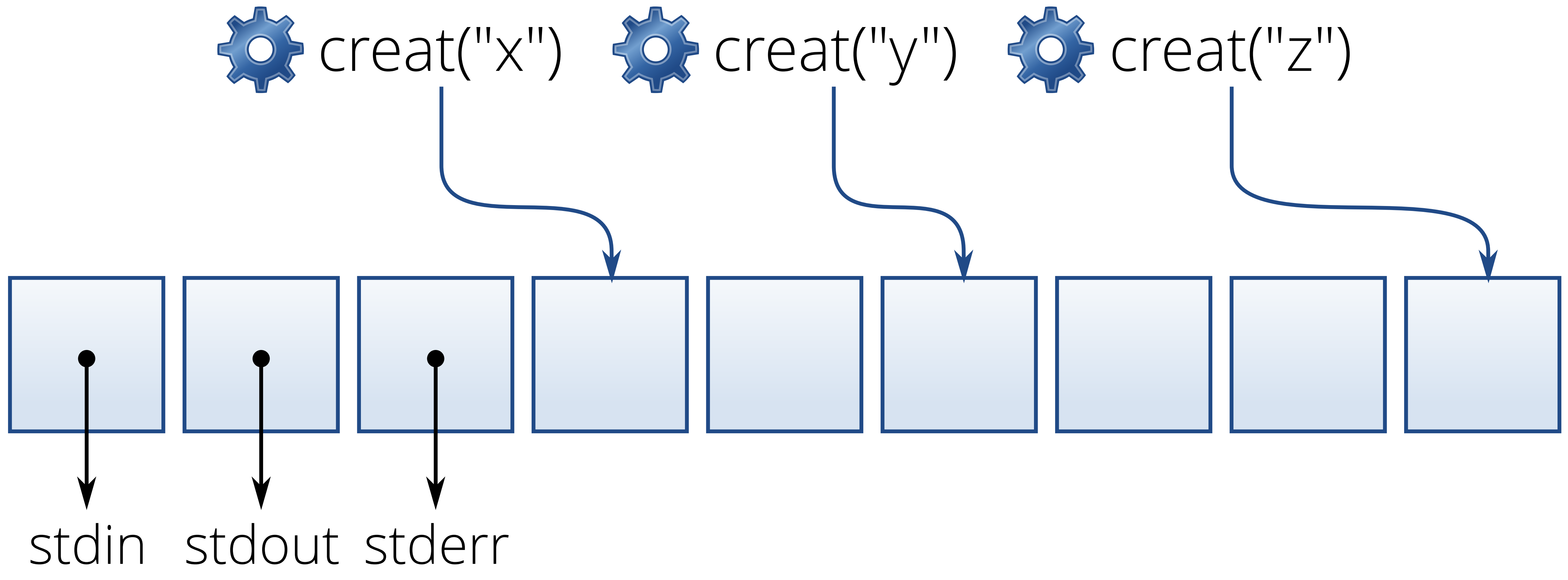
Good scalability => ability grows linearly with hw resources

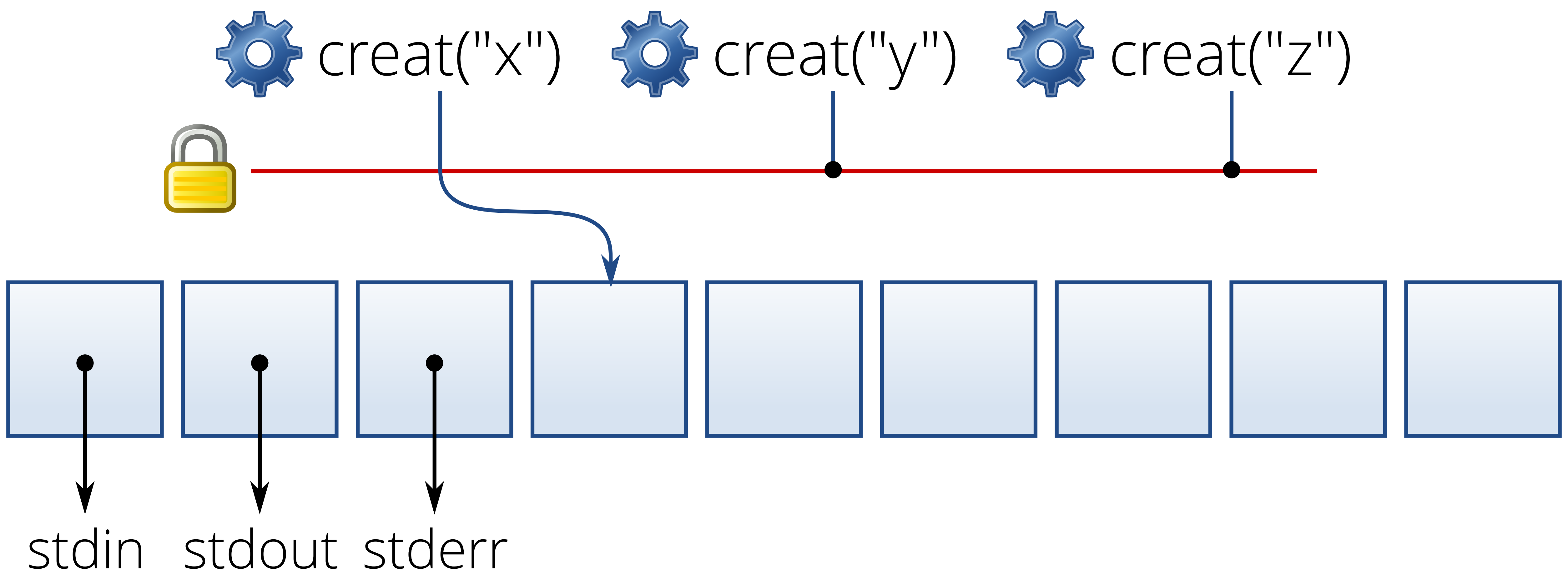


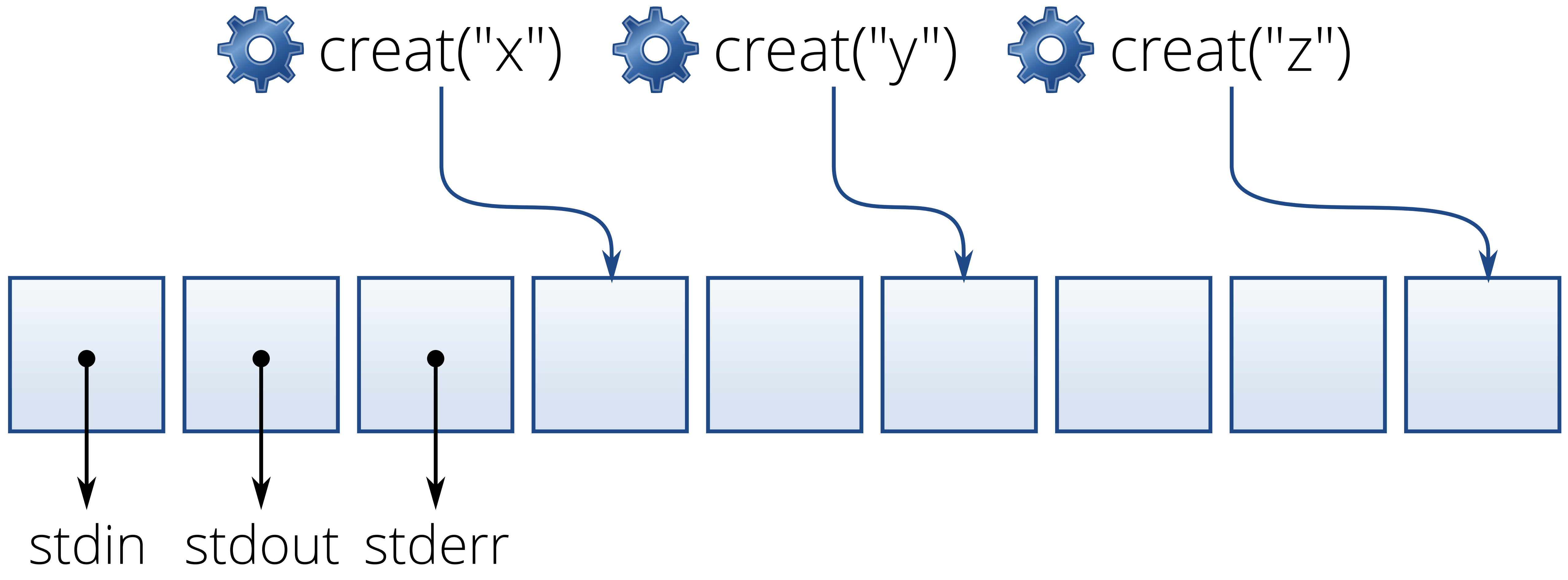
https://hpc-wiki.info/mediawiki/hpc_images/7/71/Cg_speedup.png


creat("x")
creat("y")
creat("z")
stdin
stdout
stderr


creat("x")
creat("y")
creat("z")
stdin
stdout
stderr


creat("x")
creat("y")
creat("z")
stdin
stdout
stderr

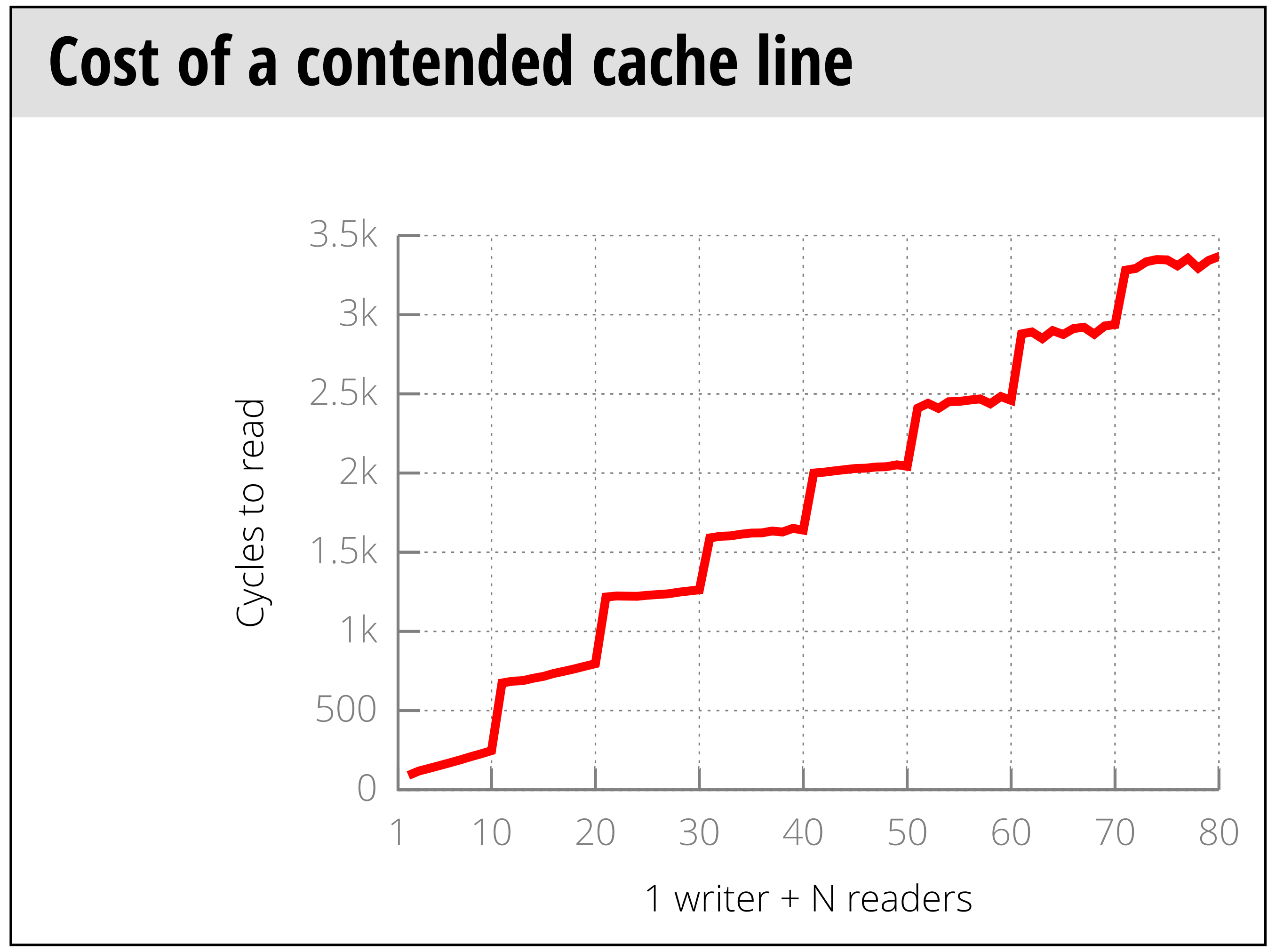




<u>SC Rule</u>: Whenever interface operations commute, they can be implemented in a way that scales.

Intuition behind rule (in the multicore context)

Core Y \ Core X	W	R	-
W	✗	✗	✓
R	✗	✓	✓
-	✓	✓	-



Two or more operations are scalable if they are conflict-free.

Operations commute

⇒ results independent of order

⇒ communication is unnecessary

⇒ without communication, no conflicts

Examples

	Commutes	Scalable implementation exists
P1: creat P1: creat	✗	
P1: creat("/tmp/x") P2: creat("/etc/y")	✓	✓ (Linux)
P1: creat("/x") P2: creat("/y")	✓	✓
P1: creat("x", O_EXCL) P2: creat("x", O_EXCL)		
Same CWD	✗	
Different CWD	✓	✓

Commutativity Rule

⇓

improve POSIX scalability

- Lowest FD versus any FD
- stat versus xstat
- Unordered sockets
- Delayed munmap
- fork+exec versus posix_spawn

Scalable Commutativity Rule

Whenever interface operations commute,
they can be implemented in a way that scales.

Recap

- Fundamental sources of complexity:
  - *many components + many interconnections + irregularity & exceptions*
- Use modularity to
  - *encapsulate elements into components & subsystems => fewer visible elements*
  - *control interactions and propagation of behaviors => fewer interconnections*
- Use abstraction to
  - *make emergent behavior predictable => less irregularity & fewer exceptions*
- Later on...
  - *patterns of using modularity and abstraction (layering, naming, client/server, ...)*

Exokernel: An Operating System Architecture for Application-Level Resource Management

Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr.
M.I.T. Laboratory for Computer Science
Cambridge, MA 02139, U.S.A
{engler, kaashoek, james}@lcs.mit.edu

Abstract

Traditional operating systems limit the performance, flexibility, and functionality of applications by fixing the interface and implementation of operating system abstractions such as interprocess communication and virtual memory. The *exokernel* operating system architecture addresses this problem by providing application-level management of physical resources. In the exokernel architecture, a small kernel securely exports all hardware resources through a low-level interface to untrusted library operating systems. Library operating systems use this interface to implement system objects and policies. This separation of resource protection from management allows application-specific customization of traditional operating system abstractions by extending, specializing, or even replacing libraries.

We have implemented a prototype exokernel operating system. Measurements show that most primitive kernel operations (such as exception handling and protected control transfer) are ten to 100 times faster than in Ultrix, a mature monolithic UNIX operating system. In addition, we demonstrate that an exokernel allows applica-
tions to control machine resources in ways not possible in traditional

inappropriate for three main reasons: it denies applications the advantages of domain-specific optimizations, it discourages changes to the implementations of existing abstractions, and it restricts the flexibility of application builders, since new abstractions can only be added by awkward emulation on top of existing ones (if they can be added at all).

We believe these problems can be solved through *application-level* (*i.e.,* untrusted) resource management. To this end, we have designed a new operating system architecture, *exokernel*, in which traditional operating system abstractions, such as virtual memory (VM) and interprocess communication (IPC), are implemented entirely at application level by untrusted software. In this architecture, a minimal kernel—which we call an *exokernel*—securely multiplexes available hardware resources. Library operating systems, working above the exokernel interface, implement higher-level abstractions. Application writers select libraries or implement their own. New implementations of library operating systems are incorporated by simply relinking application executables.

Substantial evidence exists that applications can benefit greatly from having more control over how machine resources are used