



Principles of Computer Systems

Prof. George Candea

School of Computer & Communication Sciences

Administrivia

Your POCS Team



Can Cebeci
TA



George Candea
Instructor



Katerina Argyraki
Co-instructor



Jiacheng Ma
TA

Syllabus

Fundamentals

Modularity & Abstraction
Naming & Indirection
Layering
Virtualization
Client/Server Organization
Laziness & Speculation
Redundancy
Locality & Caching
Hardware/Software Co-Design

Case studies

Memory Virtualization
Machine Virtualization
Distributed Denial of Service
Modern DNS
Transactions

Technical Writing

Prerequisites

 Take prerequisites
very seriously

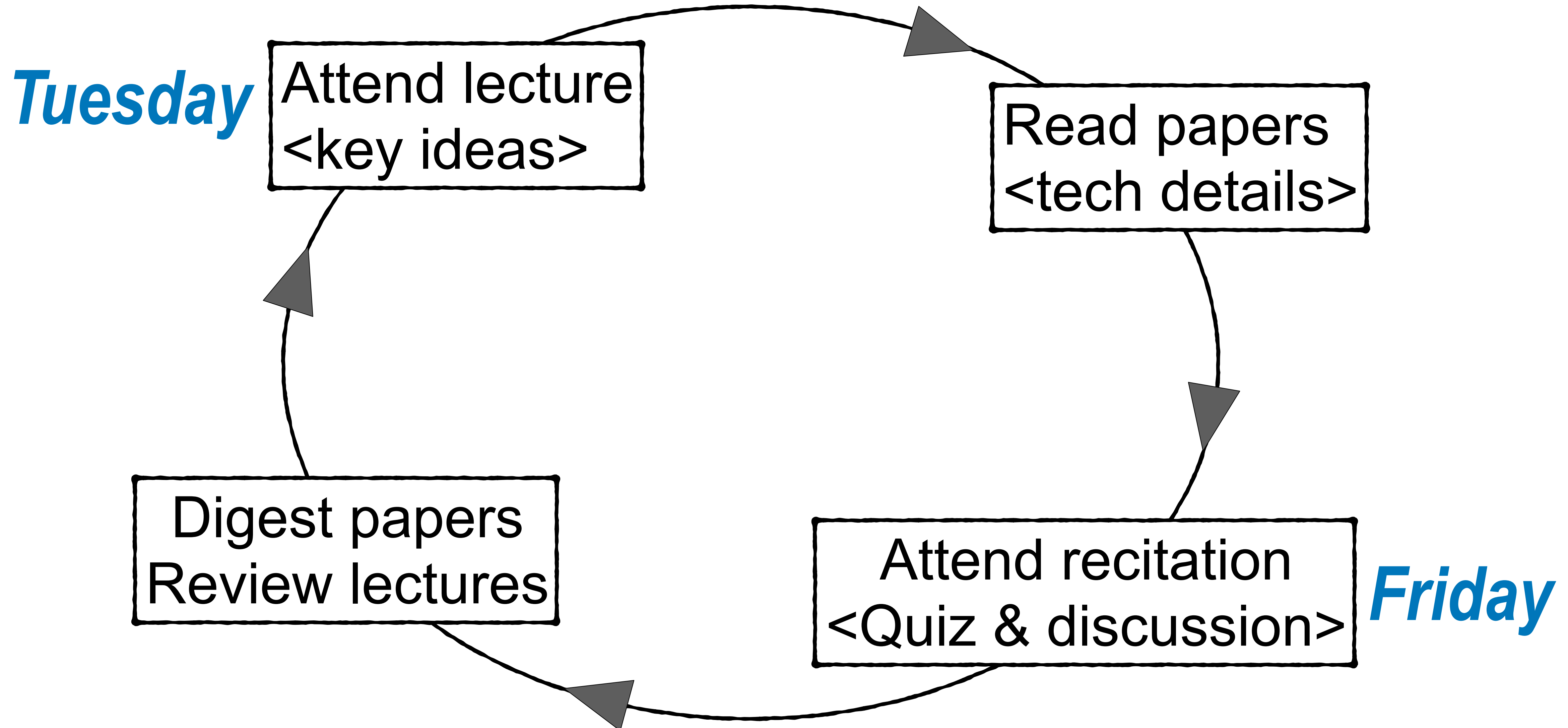
- Good knowledge of
 - *Operating systems (e.g., via CS-323)*
 - *Networks (e.g., via COM-407)*
 - *Computer architecture (e.g., via CS-470)*
 - *Databases (e.g., via CS-422)*
- Read the Exokernel, GNS, and Chord papers
 - *if you don't "get it" then POCS might not be right for you at this time*
- You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

Grading

- Quizzes = 50% of course grade
 - *demonstrate that you can identify system challenges and the techs to solve them*
 - *learn to express your ideas concisely*
 - *individual work done in class, closed book, pen & paper*
 - *graded on a curve*
 - *each quiz covers all material discussed in the course up to that point*
- Exam = 50% of course grade
 - *in-class during the last week, closed book, pen & paper*
 - *read a system description, short paper, etc.*
 - *... then answer individual questions similar to the quizzes*
 - *graded on a curve*

Typical week in POCS

 Don't fall behind



Resources

- <https://pocs.epfl.ch> → course README
- Moodle + Ed
- Lectures will not be recorded

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?**

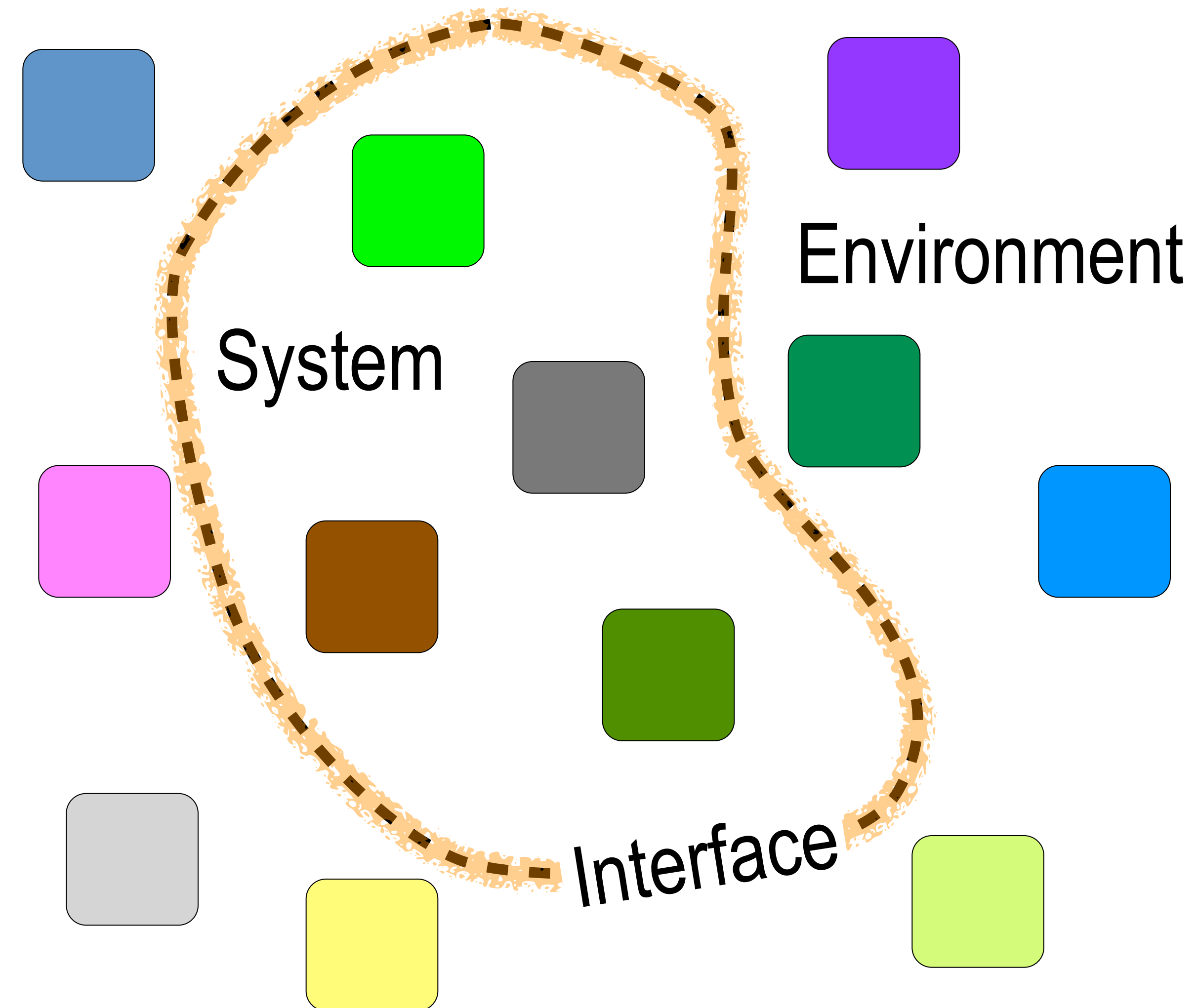


What does it mean to study principles of comp sys design ?

- 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.



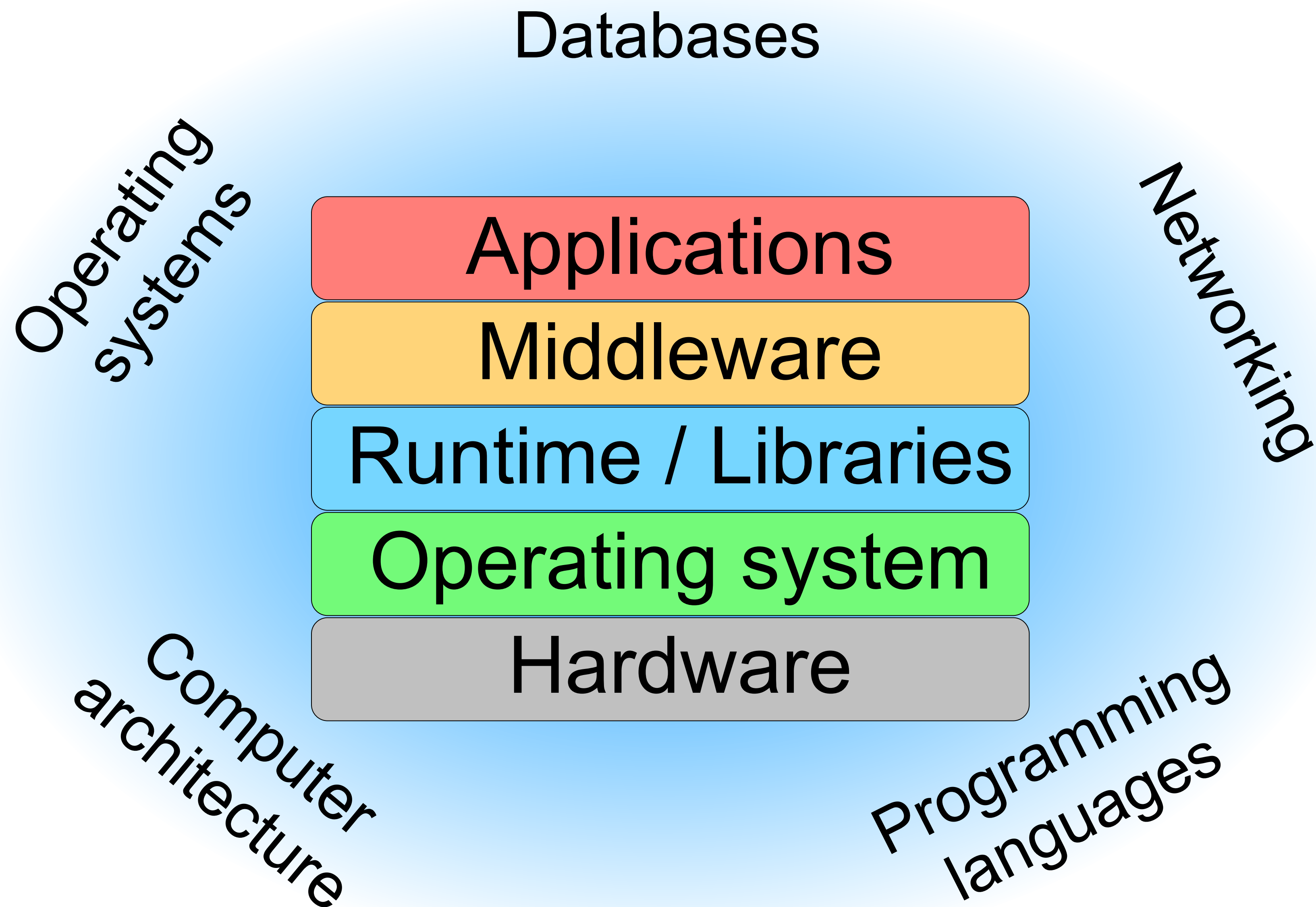
Examples of Systems

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

- Reliability
- Safety
- Security
- Performance
- Manageability

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



" 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

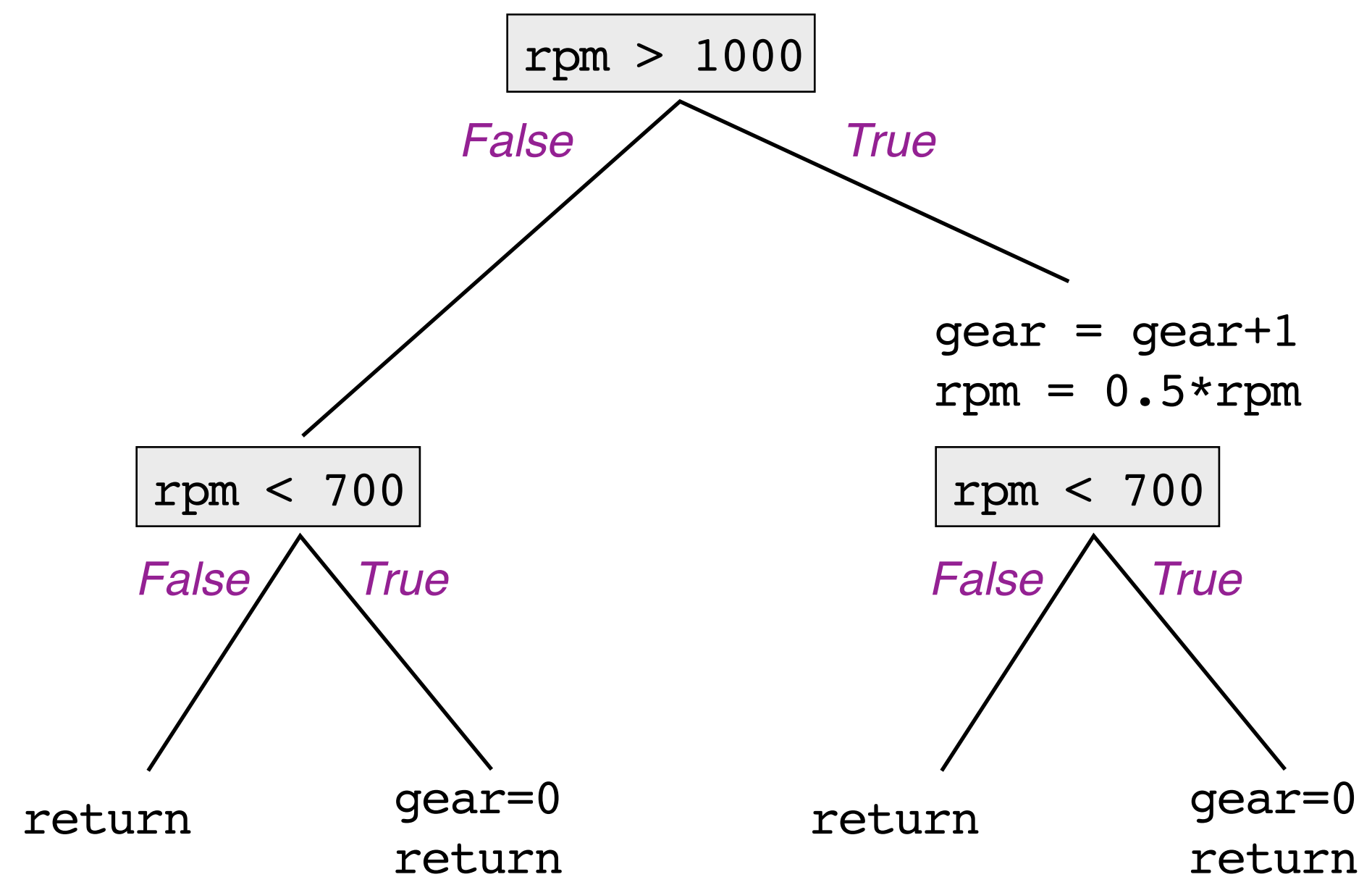
Today's lecture ...

- 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 => fewer surprises*
- Later on...
 - *patterns of using modularity and abstraction (layering, naming, client/server, etc.)*

5 Challenges in Comp Sys Design

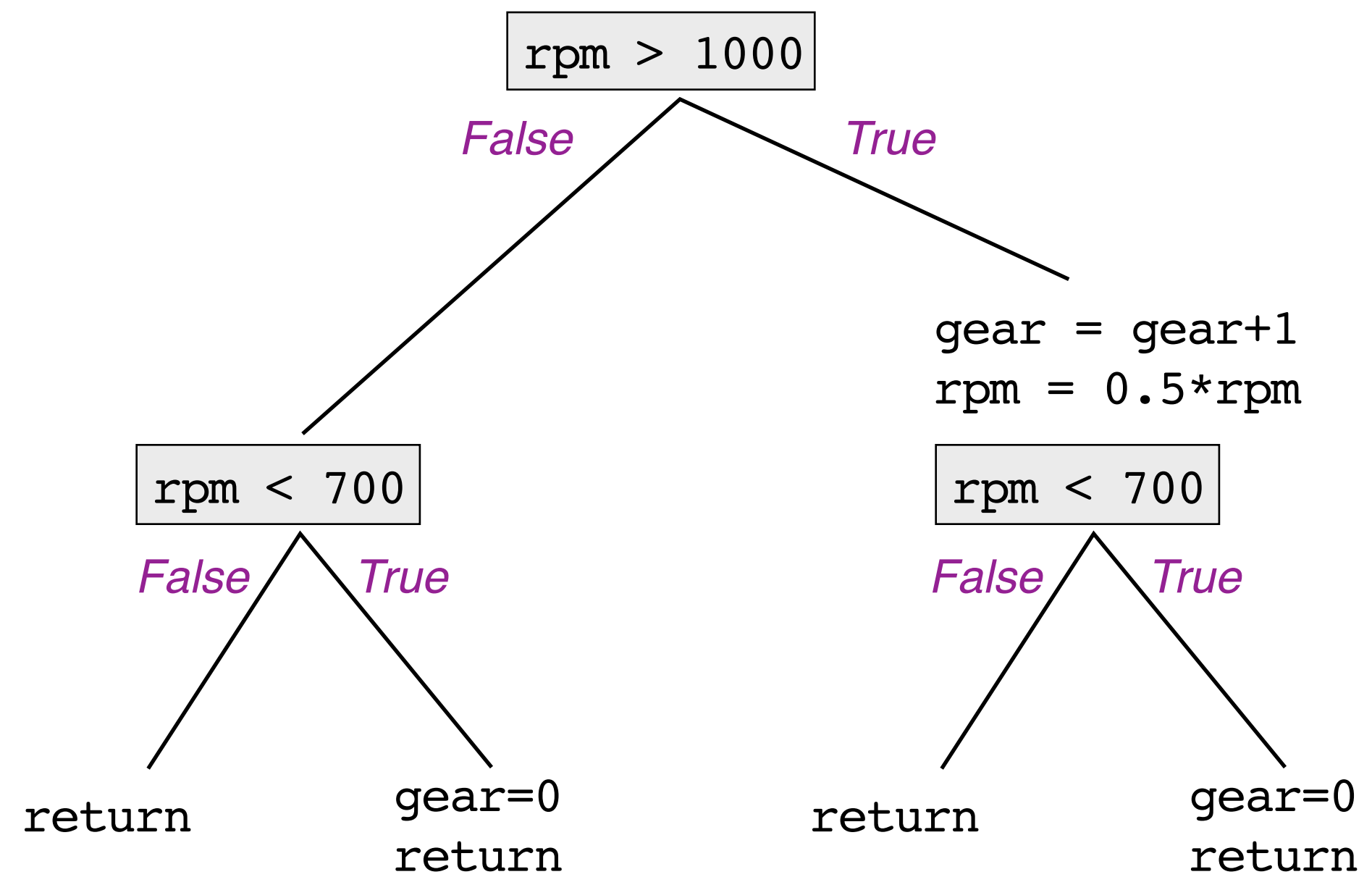
**#1 — Software/firmware
has lots of possible behaviors**


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

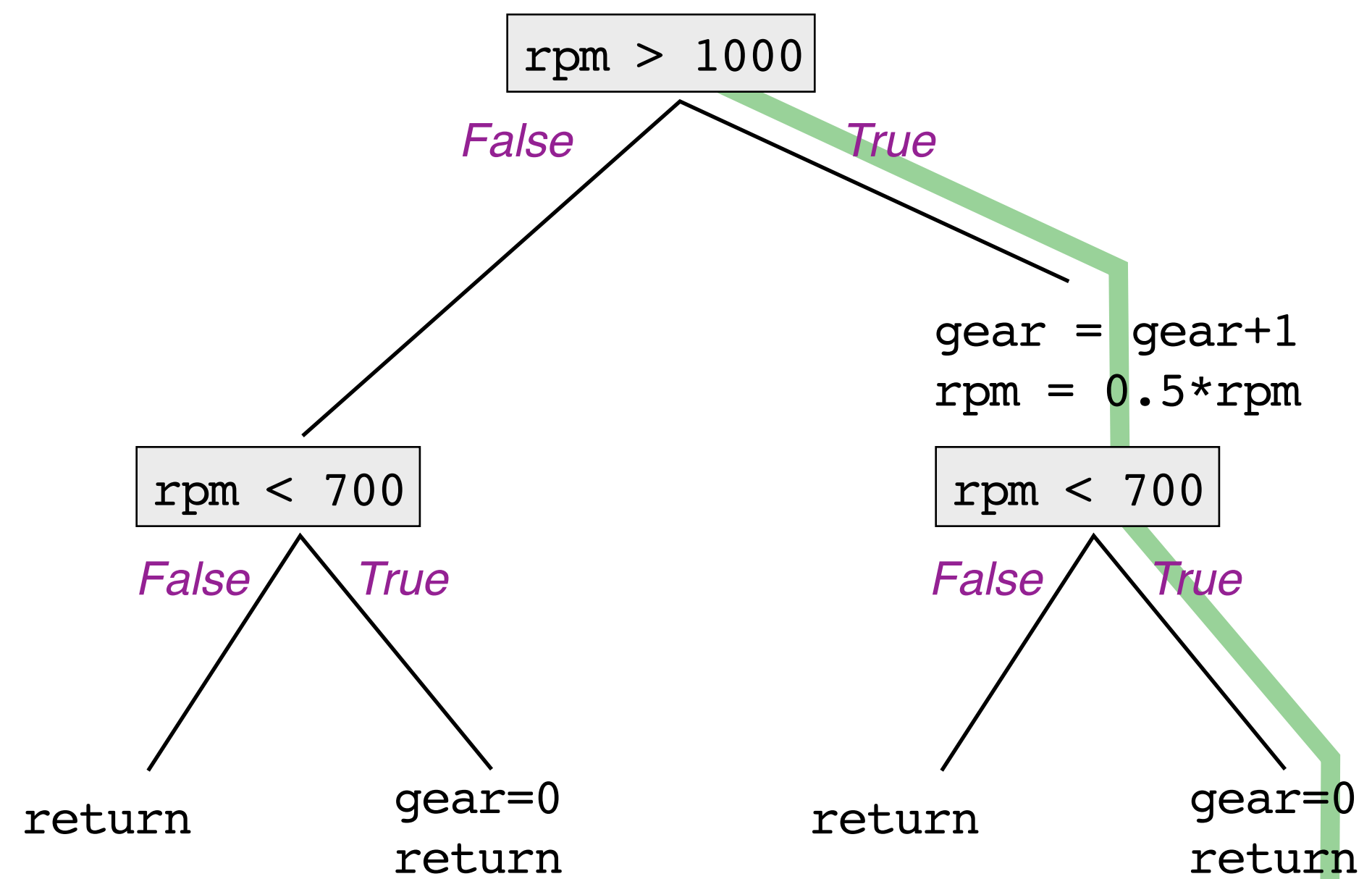


```
autoShift (int rpm)
  if (rpm > 1000)
    gear = gear+1
    rpm = 0.5*rpm
  if (rpm < 700)
    gear=0
  return
```

rpm=1200



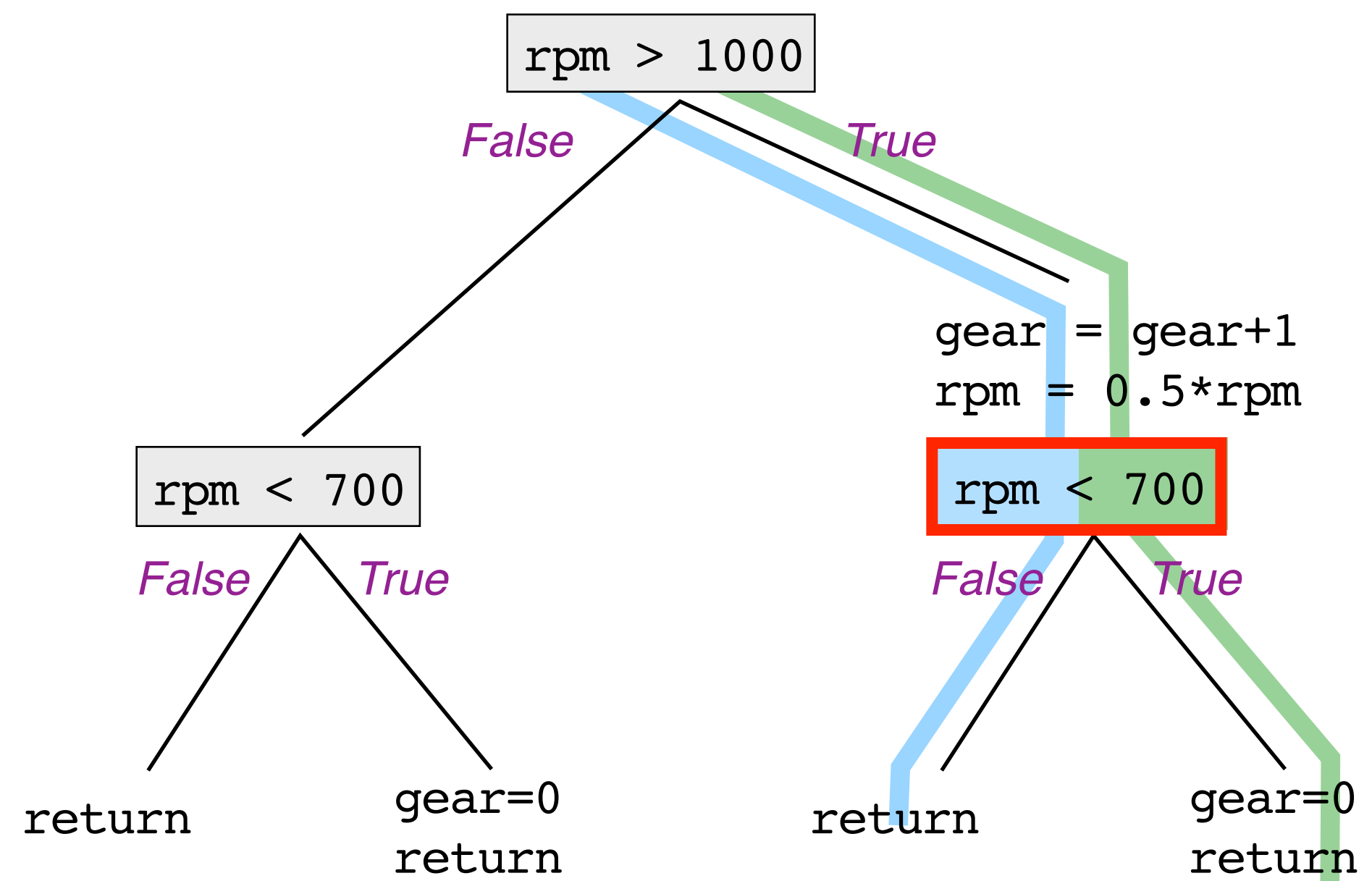
```
autoShift (int rpm)
  if (rpm > 1000)
    gear = gear+1
    rpm = 0.5*rpm
  if (rpm < 700)
    gear=0
  return
```




```

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

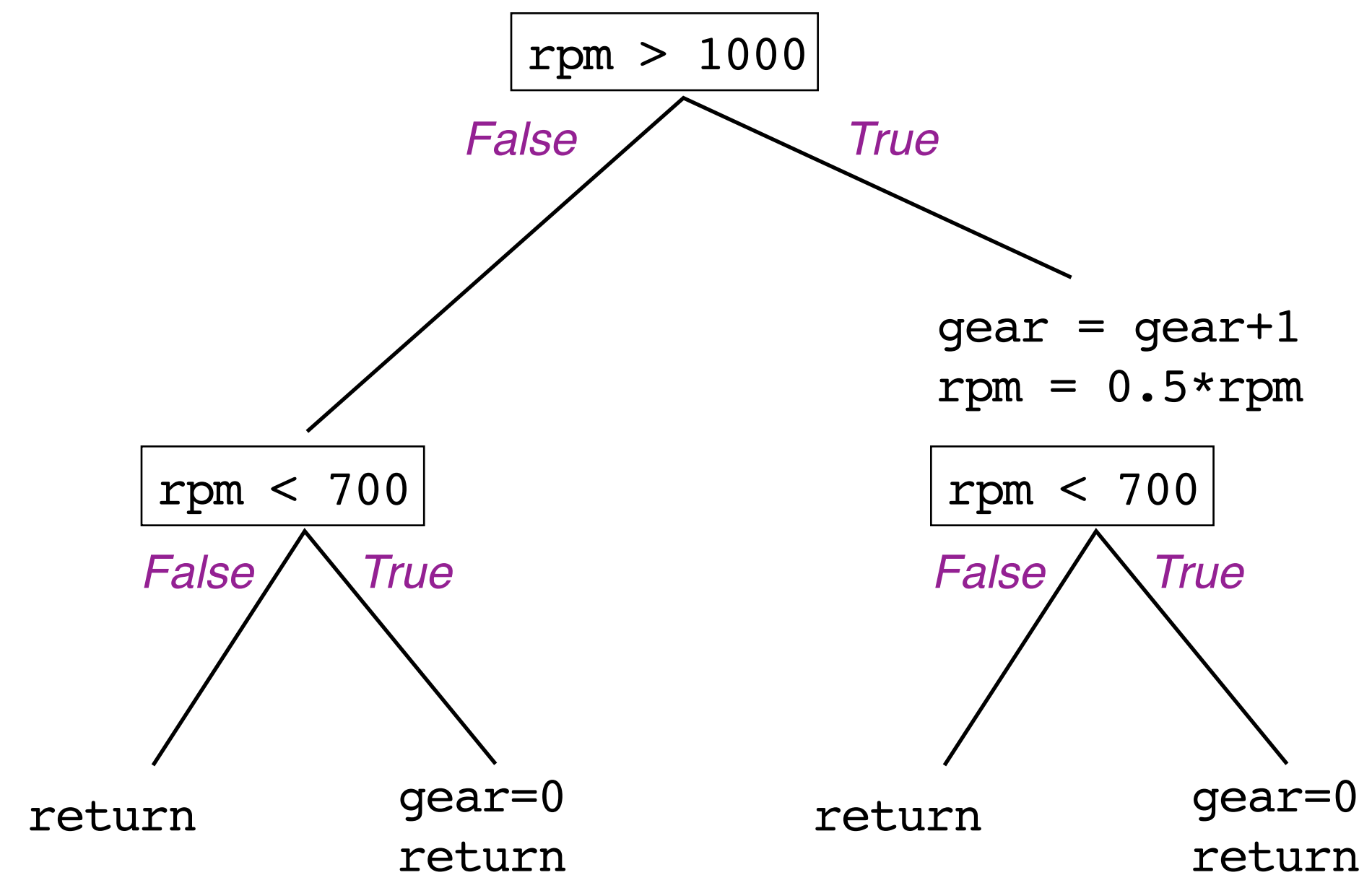
```



```

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

```



paths $\approx$ 2 program size

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



>5,000,000 lines of code¹ (LOC) $\Rightarrow \sim 2^{500,000}$ 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>

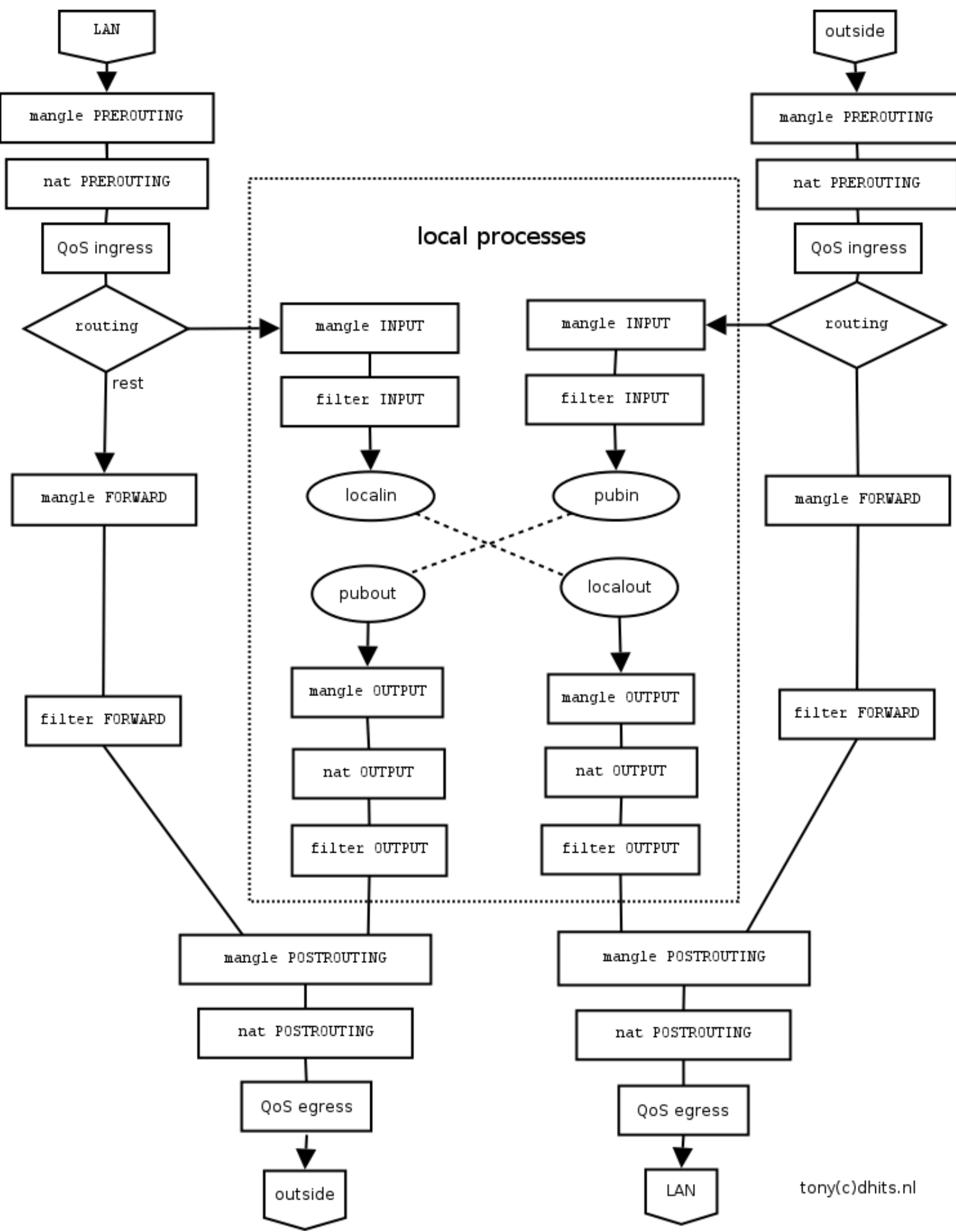
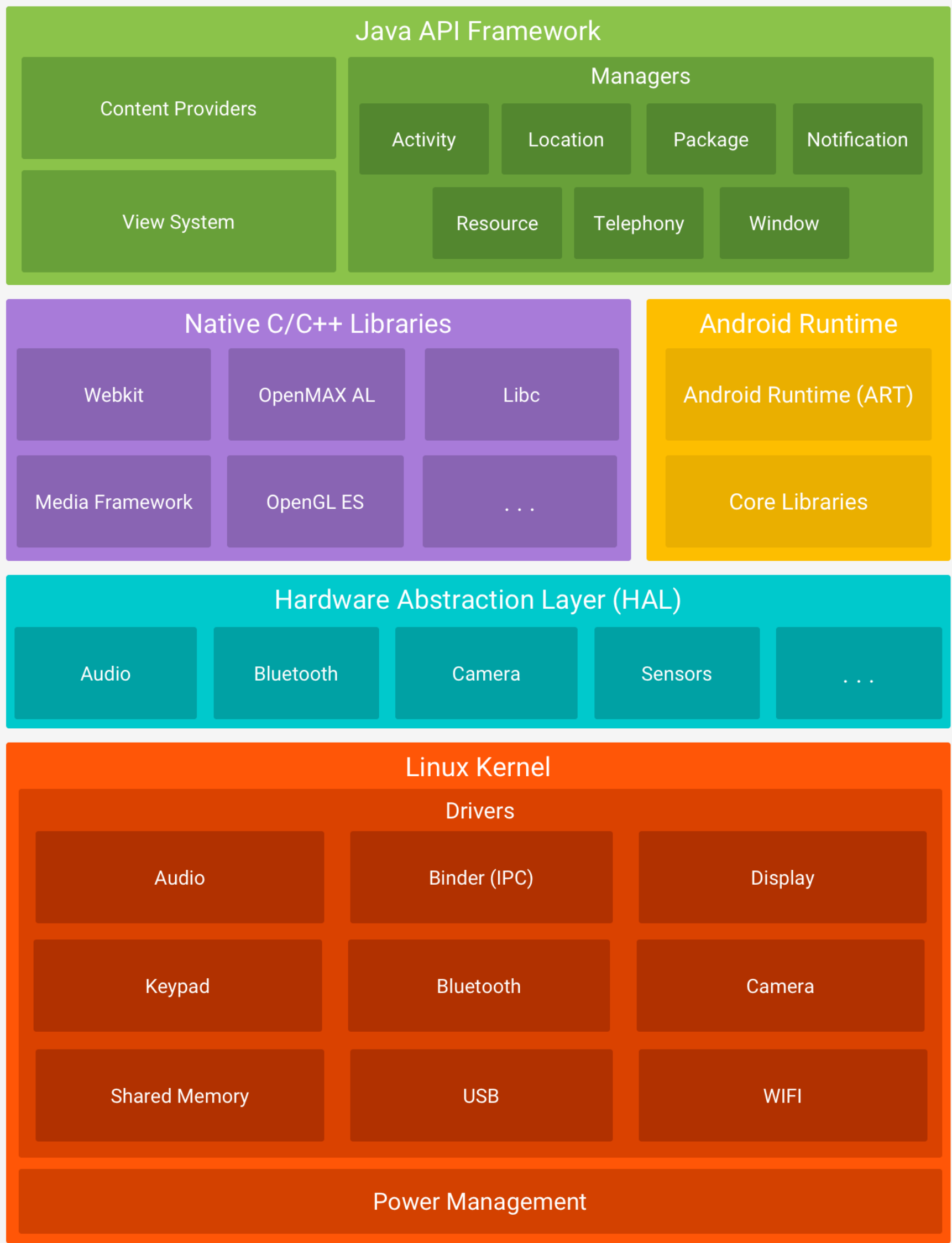
Some Code Sizes (in LOC)

- 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 — Emergent behaviors

Many Components

Android stack



Netfilter FW / NAT

tony(c)dhits.nl

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,...*

Undesirable Emergent Behaviors

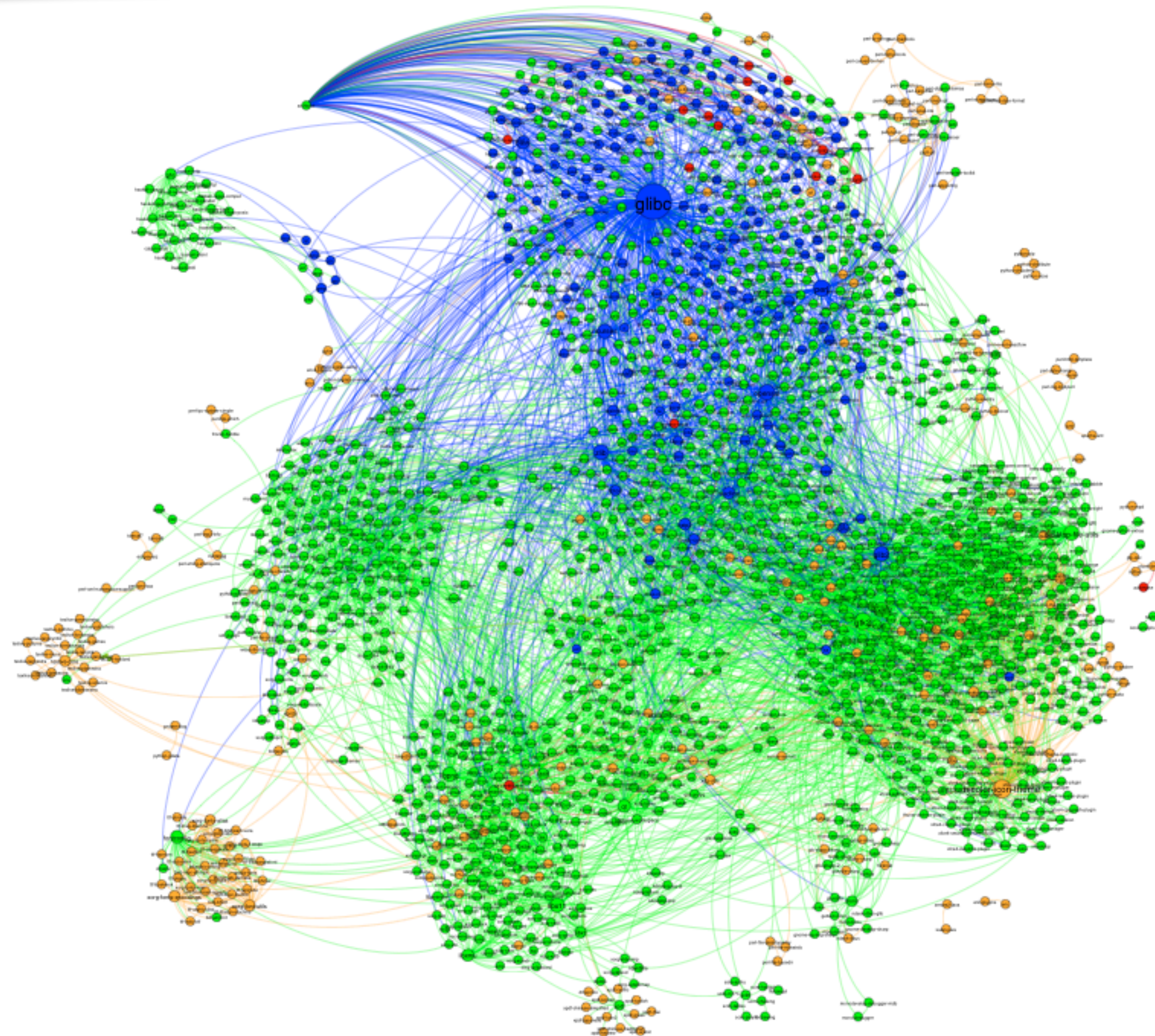
- Thrashing
- Unwanted synchronization
- Unwanted oscillation or periodicity
- Livelock/Deadlock
- Phase change
- ...

*For more examples and insights, see
J. Mogul, **Emergent (Mis) behavior vs. Complex Software Systems** (EuroSys 2006)*

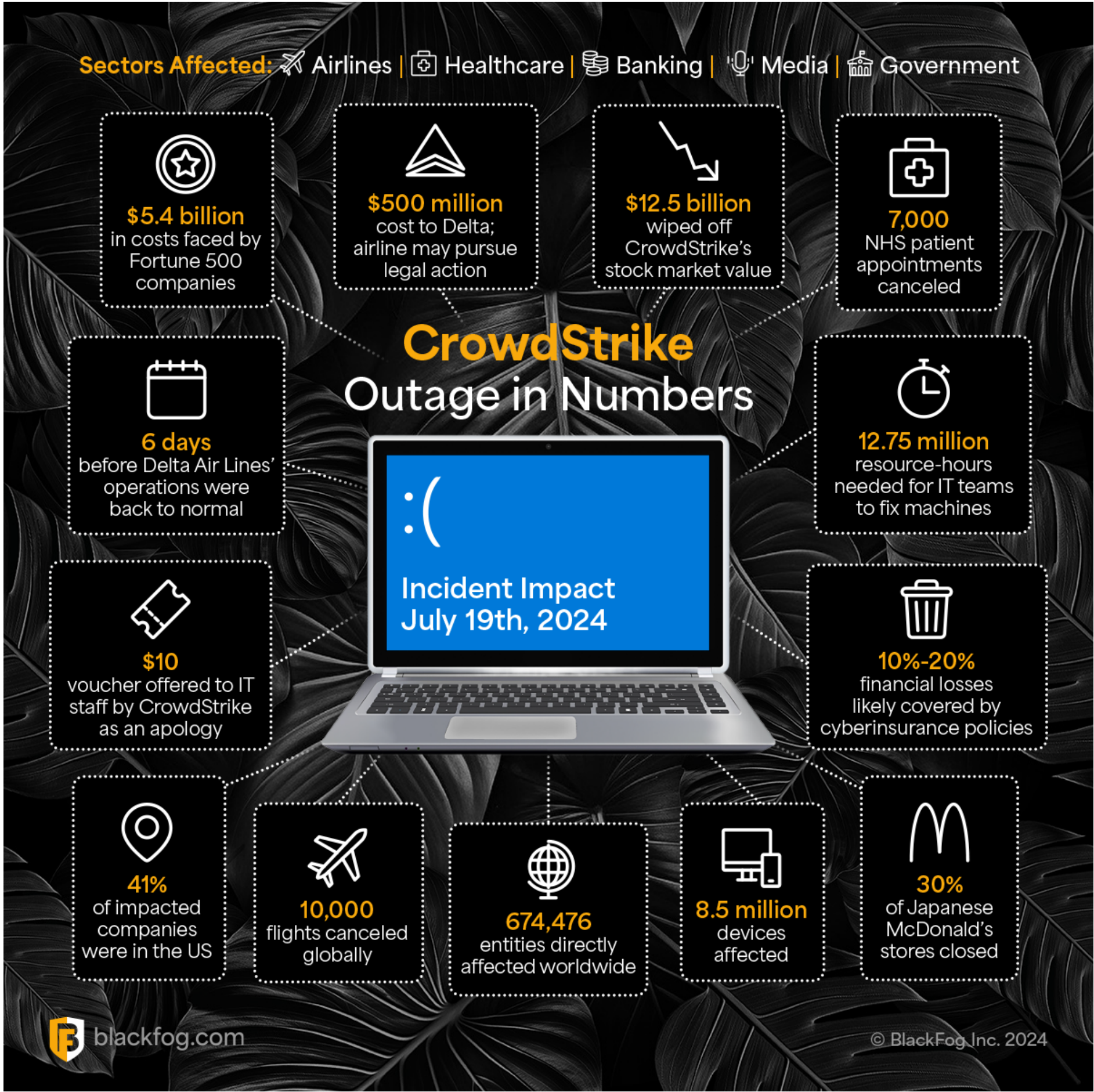
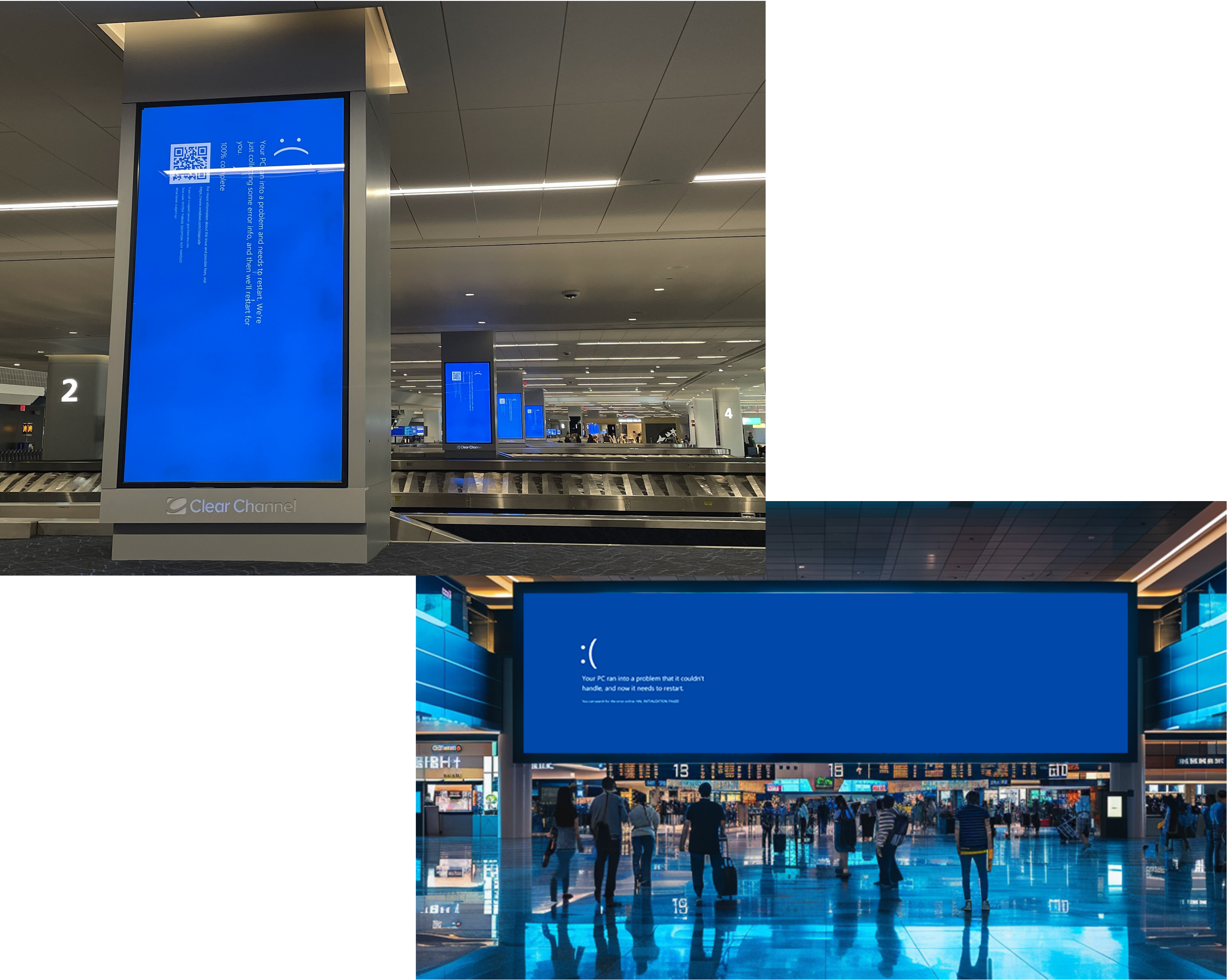
#3 — Propagation of Effects

Many Interconnections -> Propagation of Effects

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



Propagation of Effects: Crowdstrike Outage



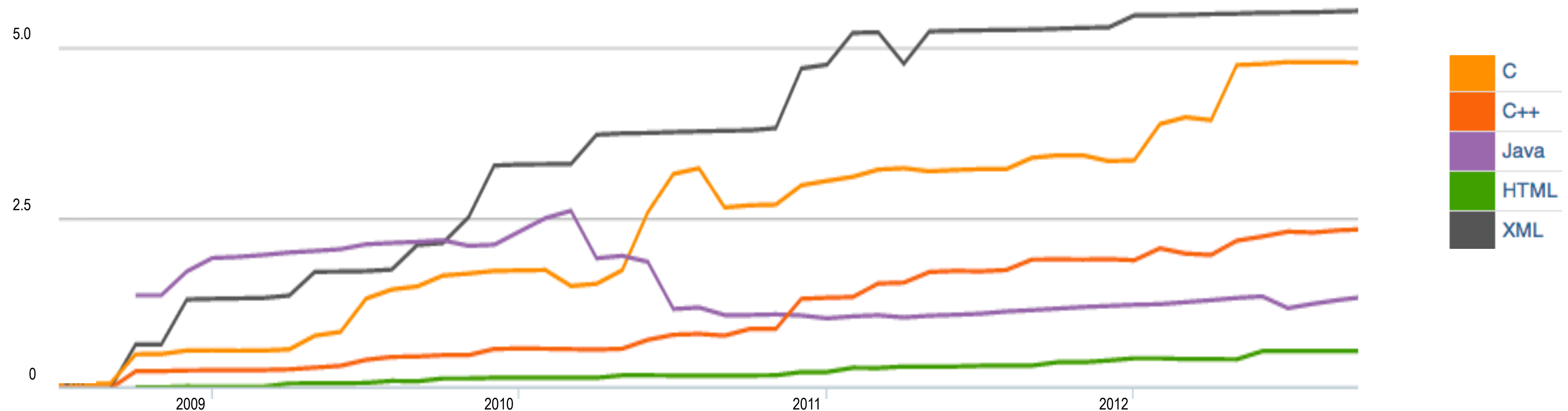
#4 — System evolution makes these challenges even harder*

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

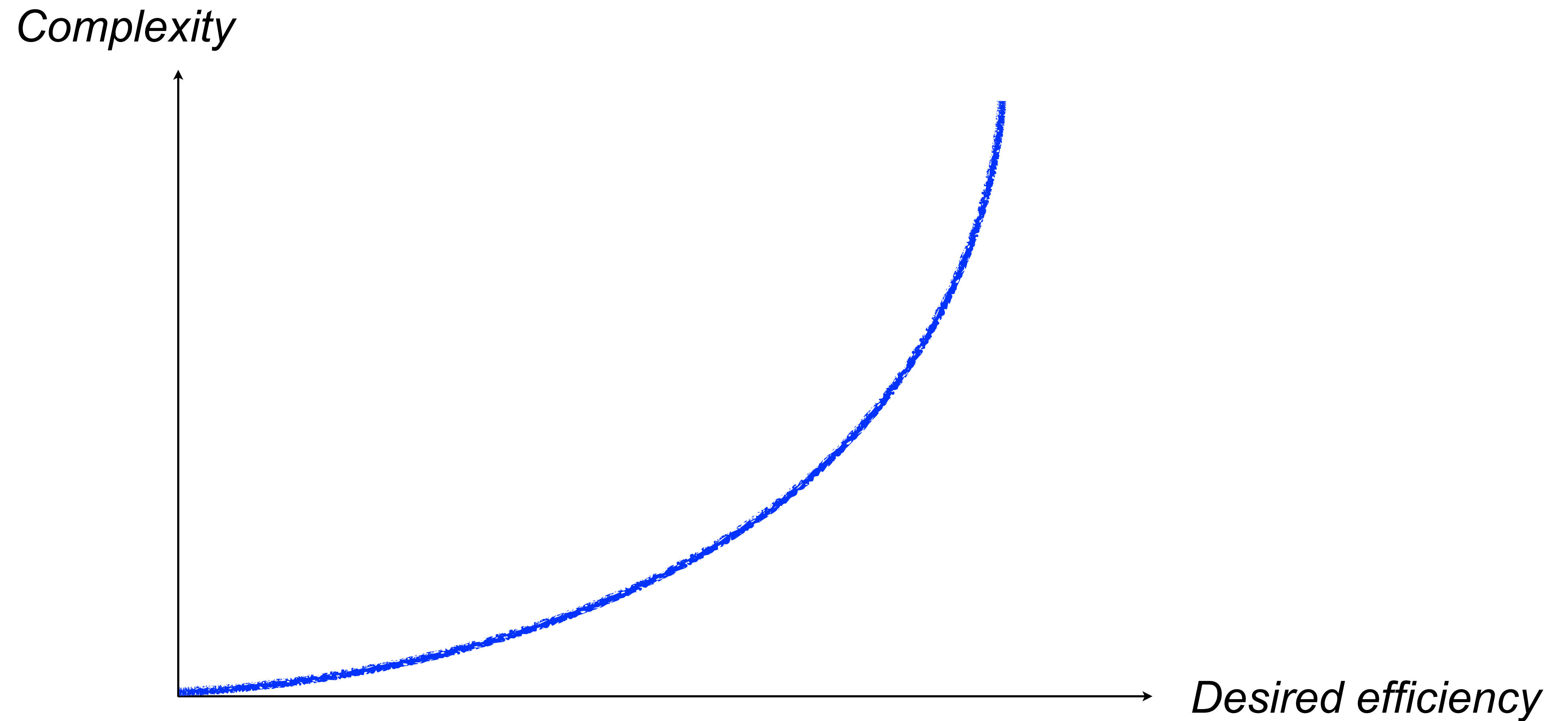
Legacy Systems = Complex Systems

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

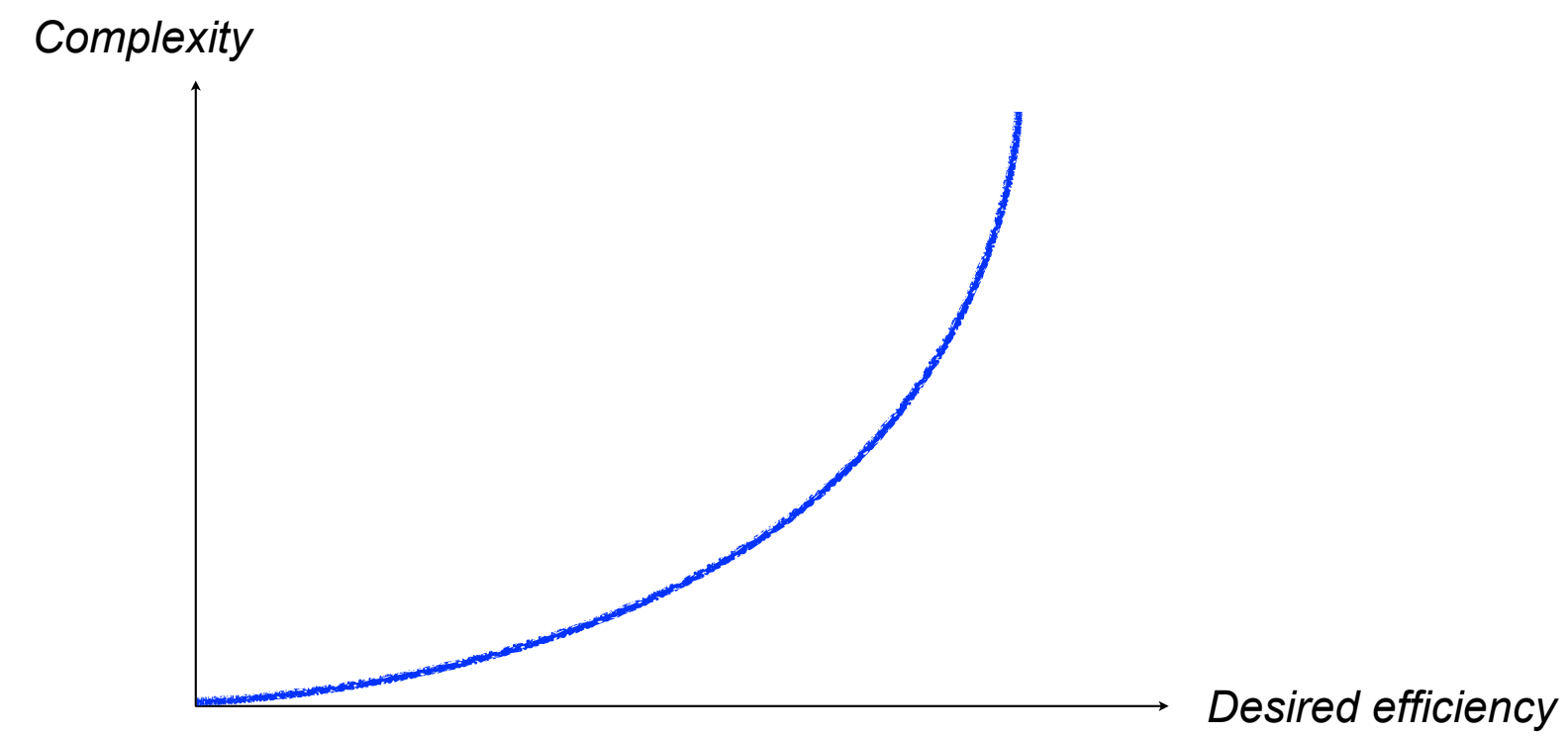
Android codebase size
(in millions of lines, over its first 4 years)



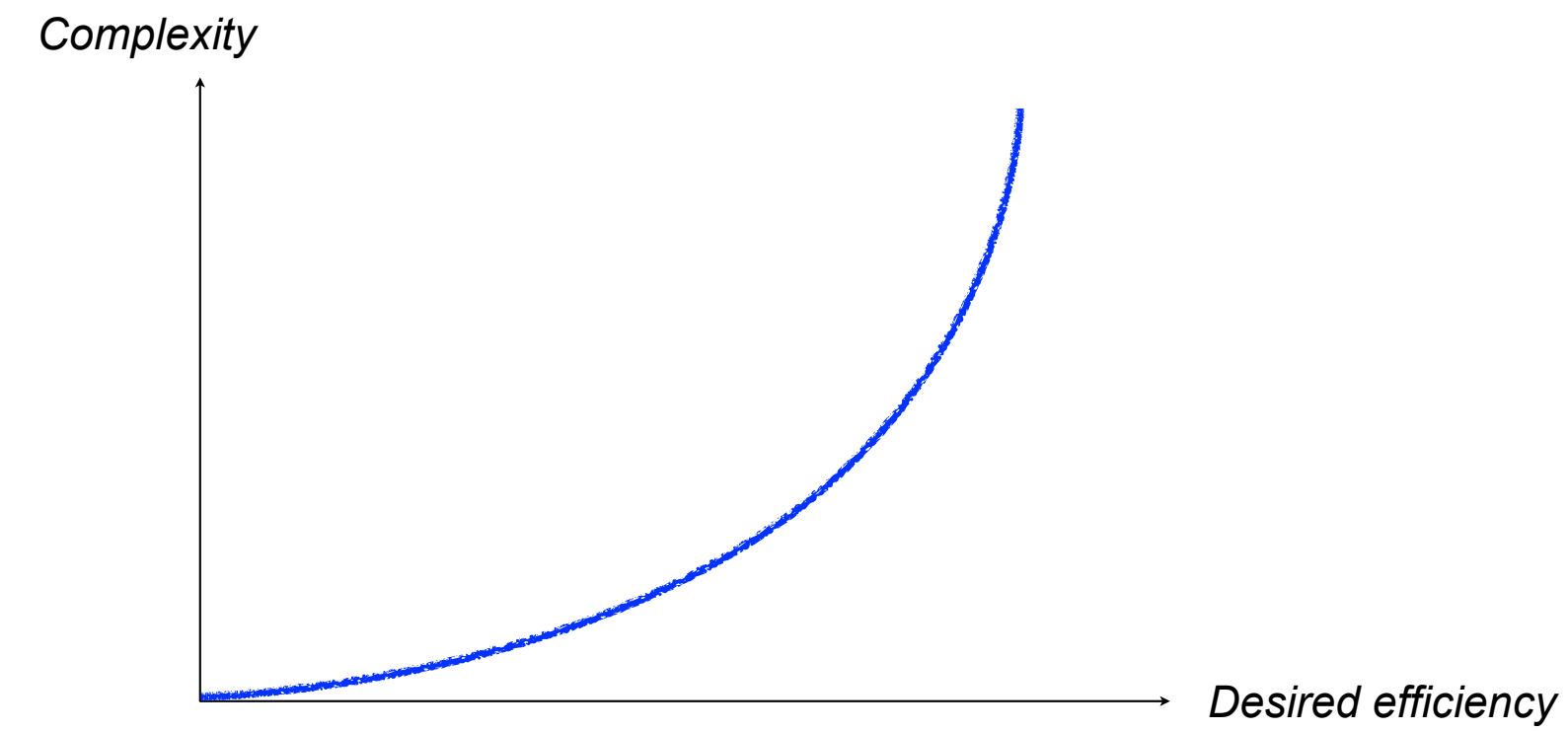
Quest for Efficiency $\Rightarrow$ Complexity



Quest for Efficiency $\Rightarrow$ Complexity



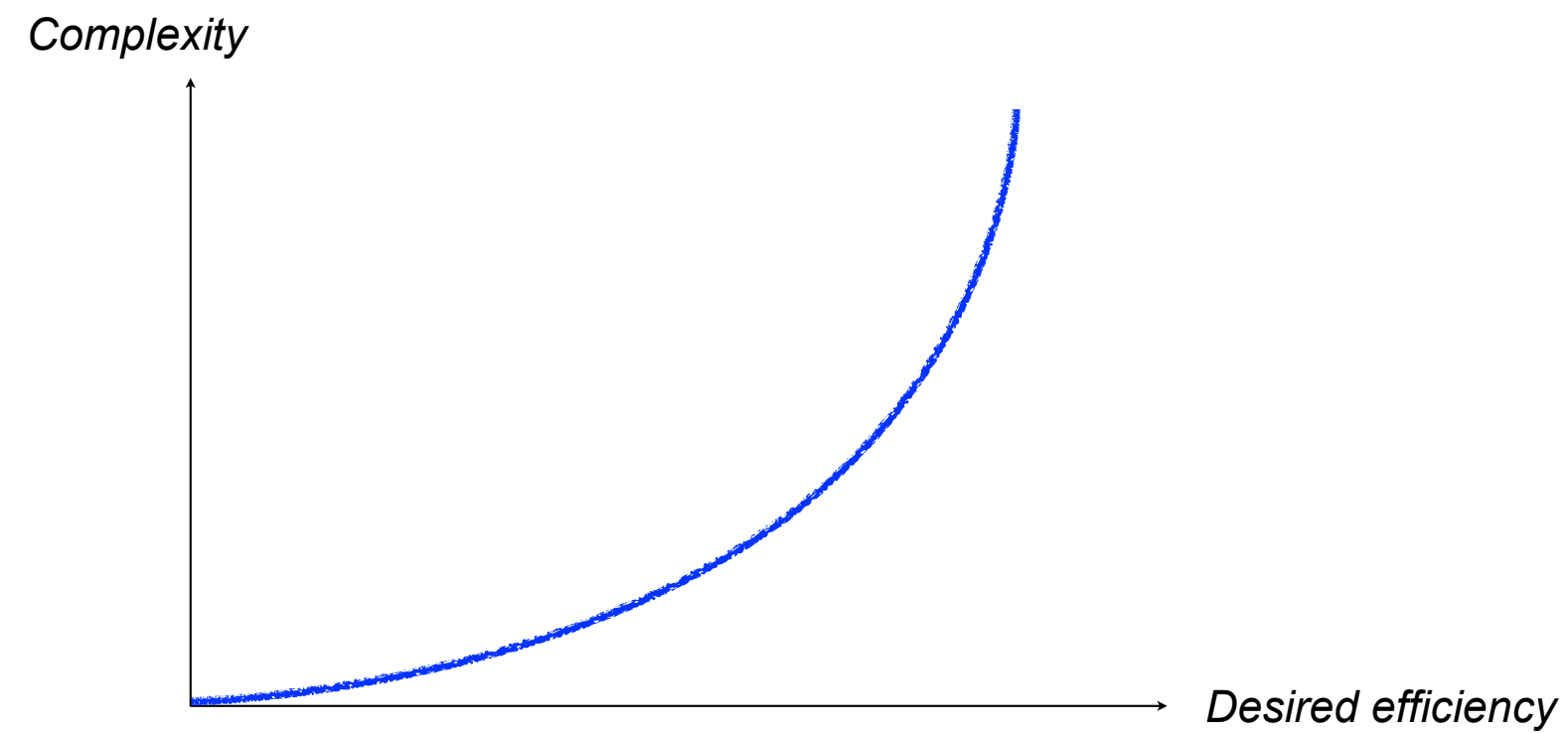
Quest for Efficiency $\Rightarrow$ Complexity



Buffer cache



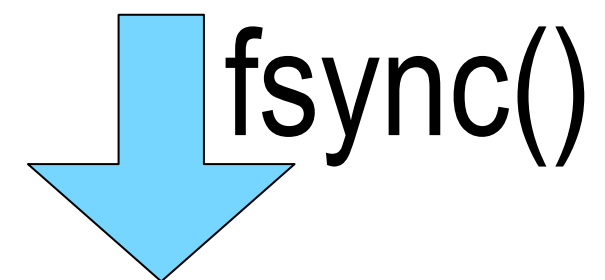
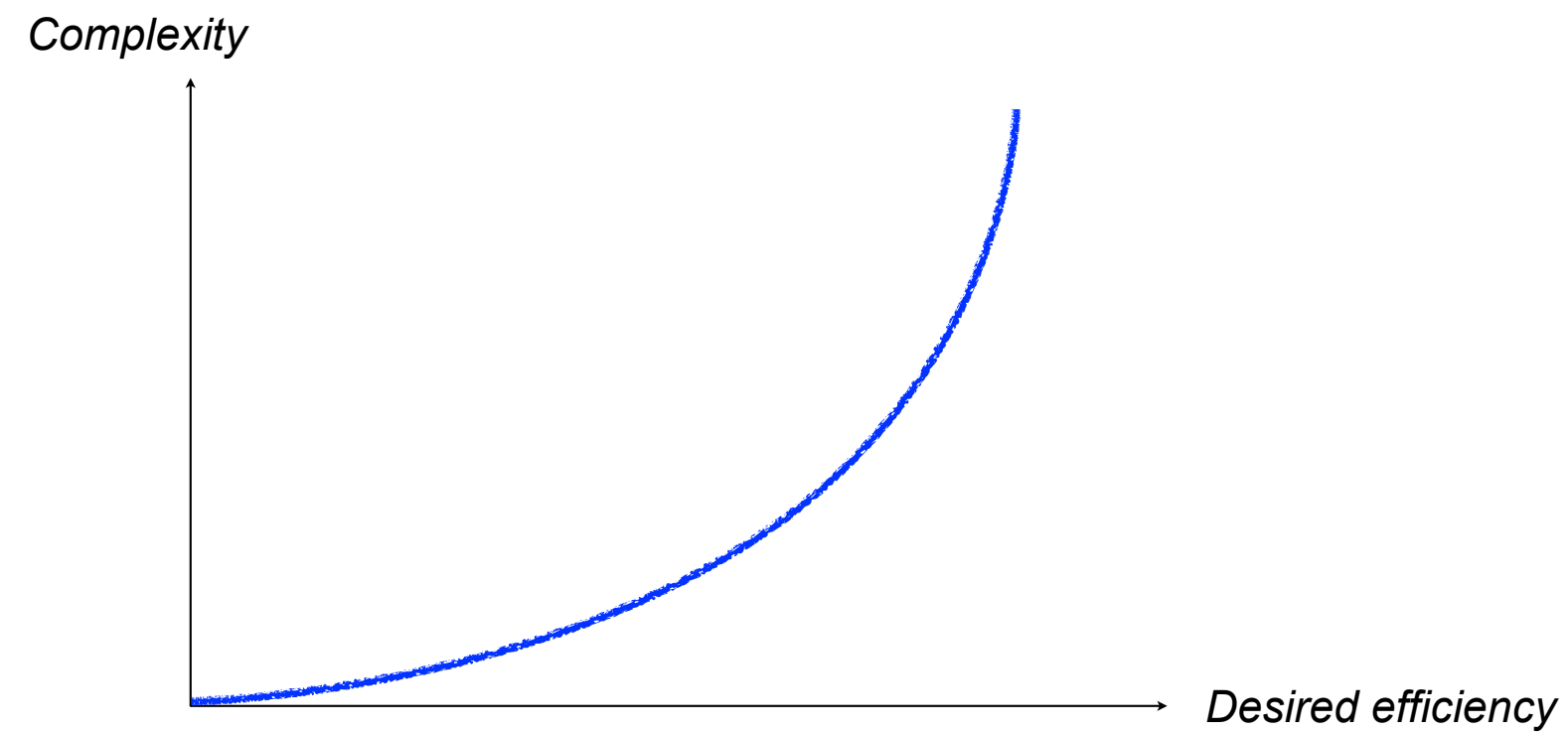
Quest for Efficiency $\Rightarrow$ Complexity



Writeback buffer cache



Quest for Efficiency $\Rightarrow$ Complexity



Writeback buffer cache



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*
- $K(\text{object}) \geq |\text{object}| \Rightarrow \text{complex}$
 $K(\text{object}) \ll |\text{object}| \Rightarrow \text{simple}$

$$|AAAAAAAA \dots AAAAB| = 10^6 + 1$$

$$K(AAAAAAAAA \dots AAAAB) =$$

|“1 million As followed by 1 B”|

$\Rightarrow$ simple

$$|ABDAGHDBBCAD\dots| = 10^6 + 1$$

$$K(ABDAGHDBBCAD\dots) = 10^6 + 1$$

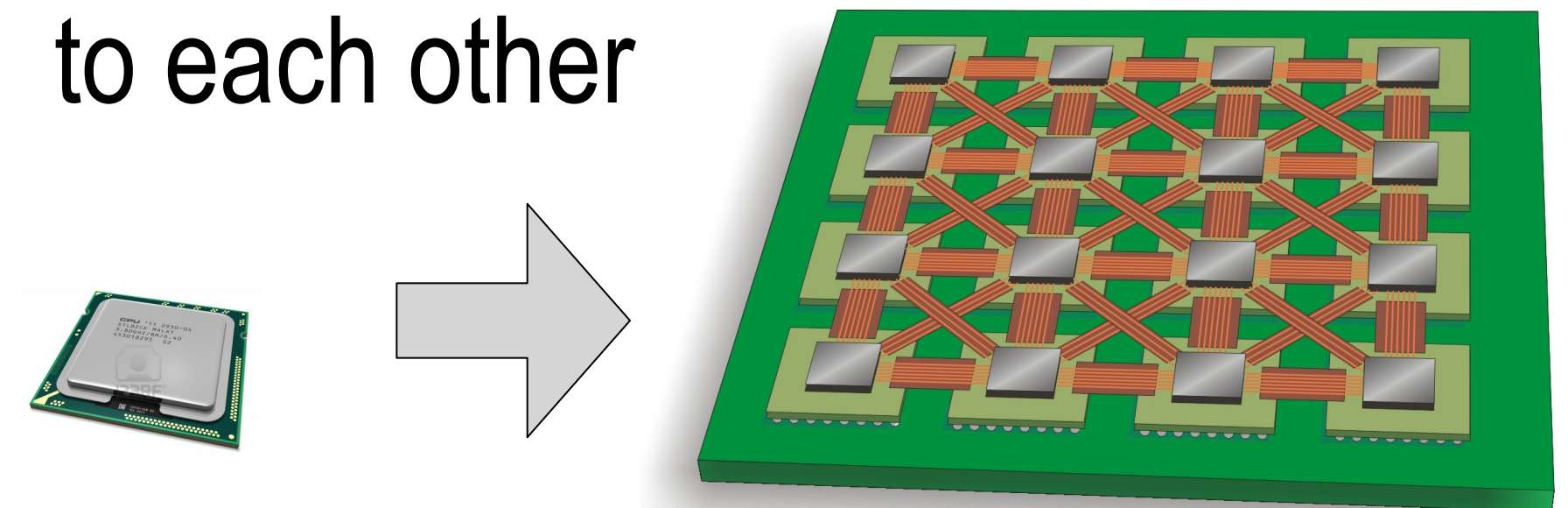
$\Rightarrow$ complex

#5 — Incommensurate scaling

Incommensurate Scaling — chasing the bottleneck

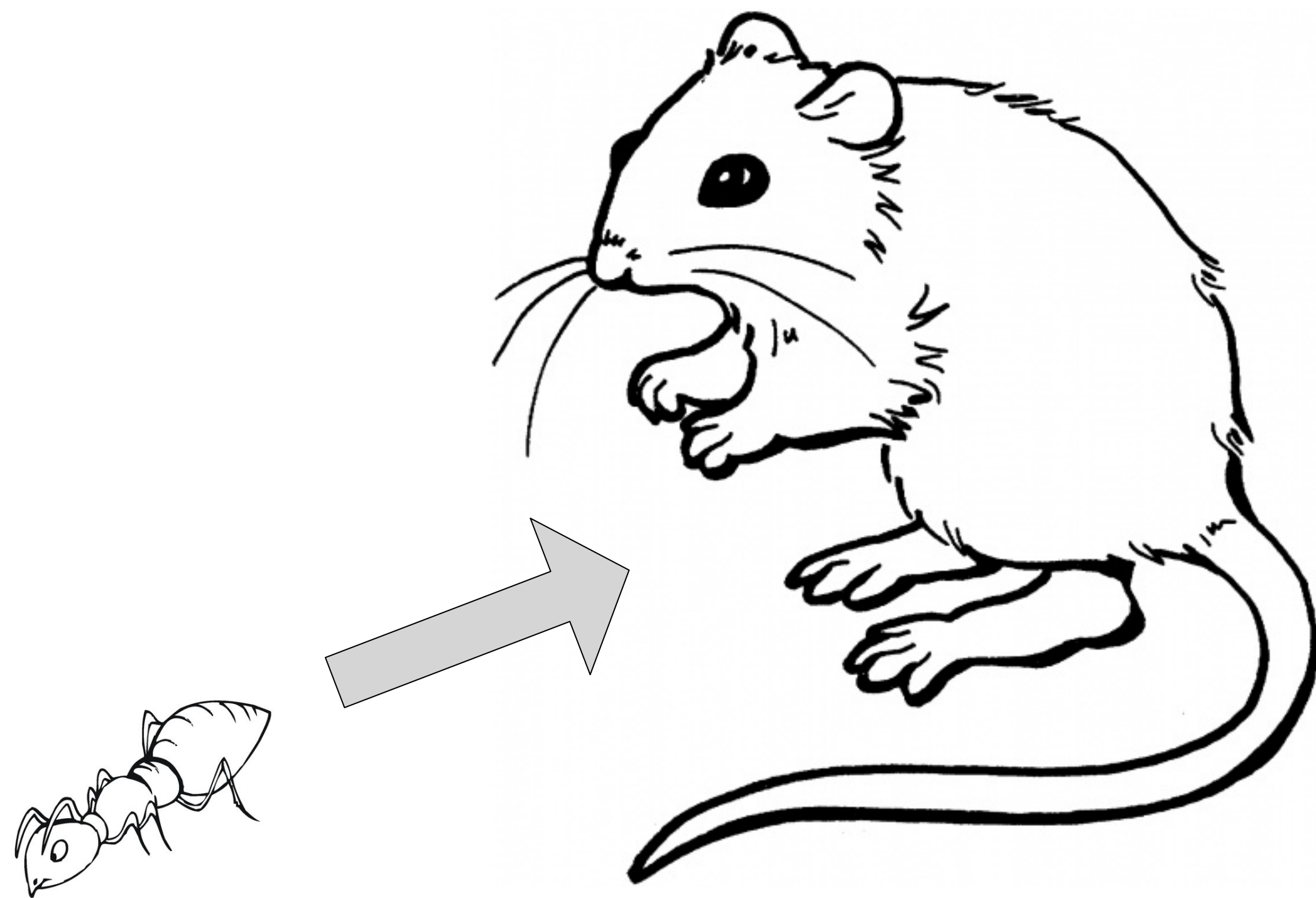
As a system increases in size or speed, different parts scale unequally, causing the system as a whole to stop working.

- 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 disproportionately to each other

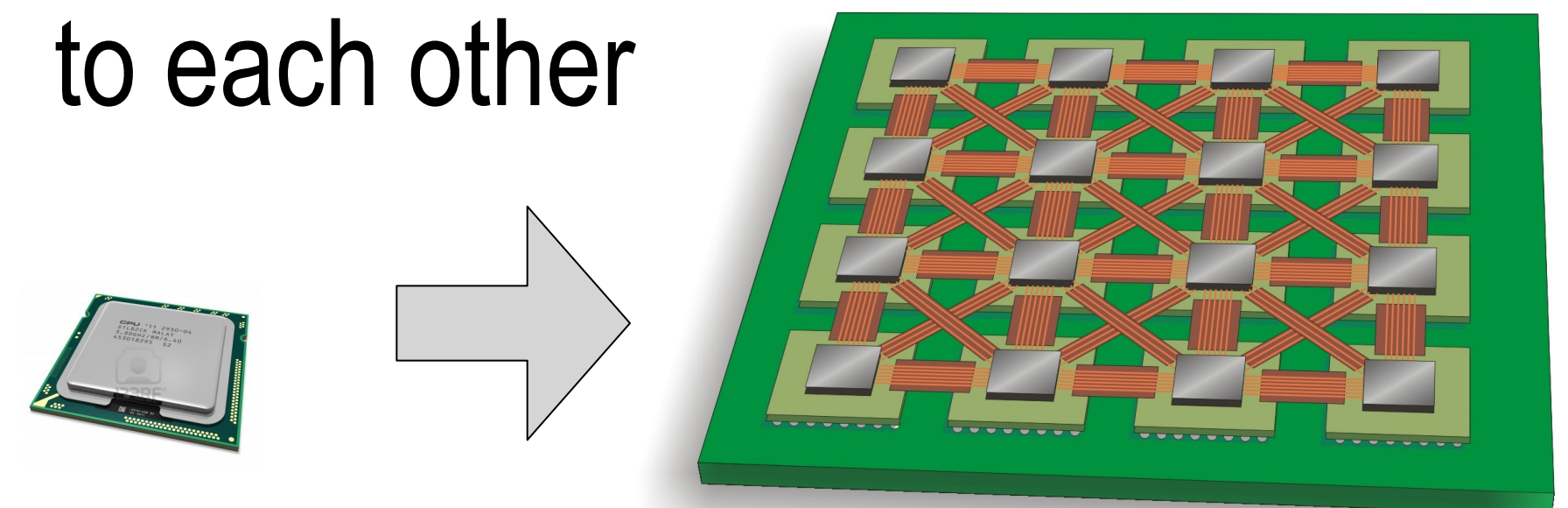


Incommensurate Scaling — chasing the bottleneck

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

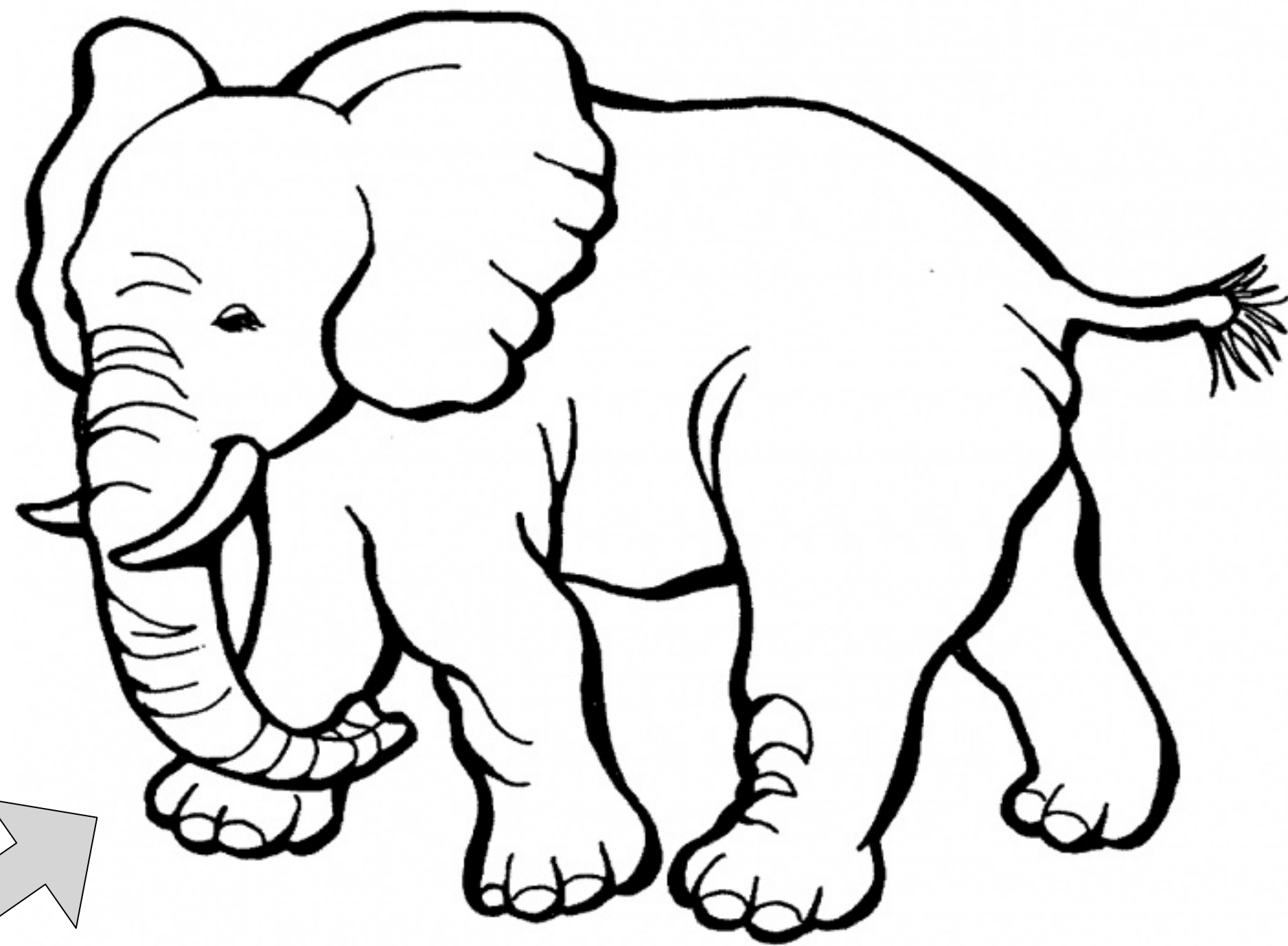


- Reason:
- Scalability of each component is described by a function
- The order of these functions is not the same for each component => as system grows, components scale disproportionately to each other

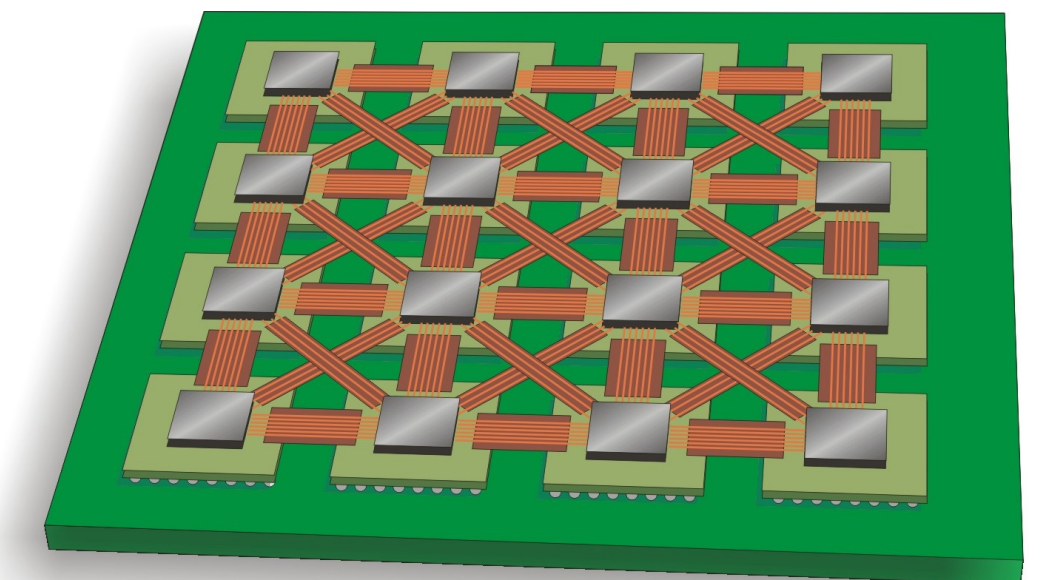
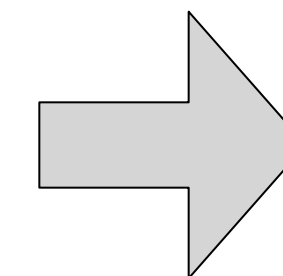


Incommensurate Scaling — chasing the bottleneck

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 disproportionately to each other



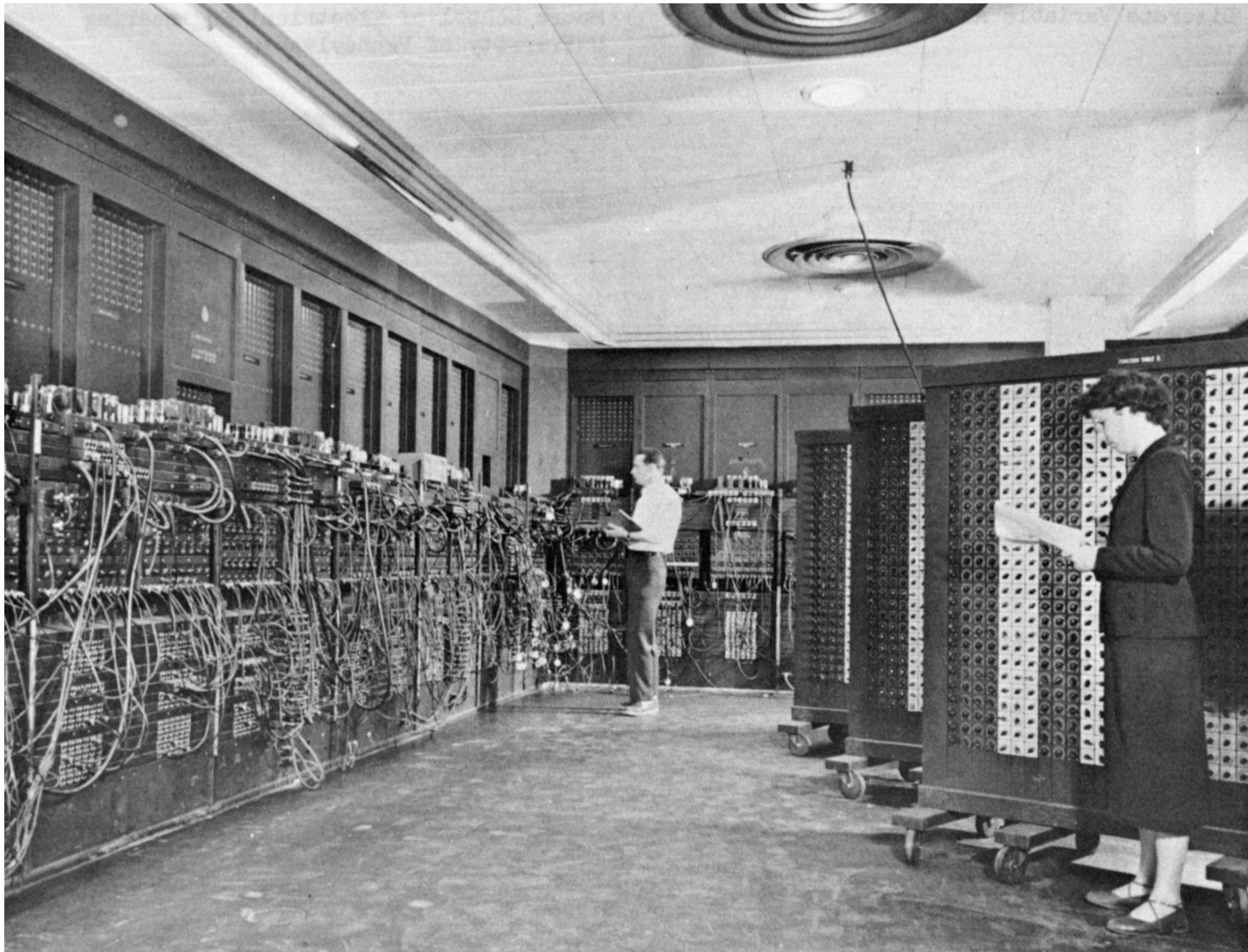
Challenges of going from components to systems

- # of behaviors of software $\sim 2^{\text{code size}}$
- many components $\Rightarrow$ emergent behaviors $\Rightarrow$ unpredictable
- many interconnections $\Rightarrow$ propagation of effects $\Rightarrow$ unpredictable
- system evolution introduces exceptions and irregularity
- incommensurate scaling forces major re-designs

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 $

```

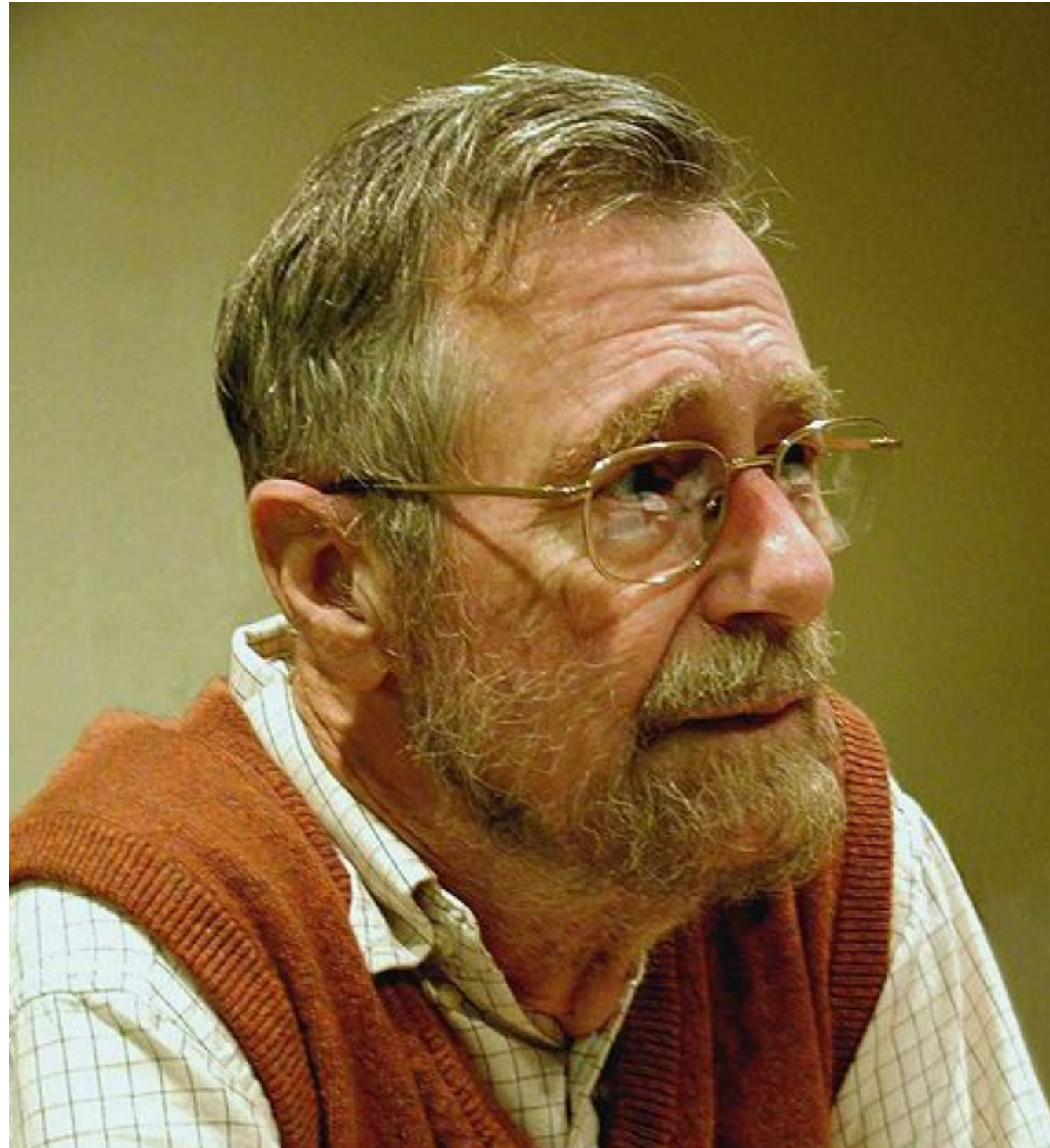


```

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 $

```

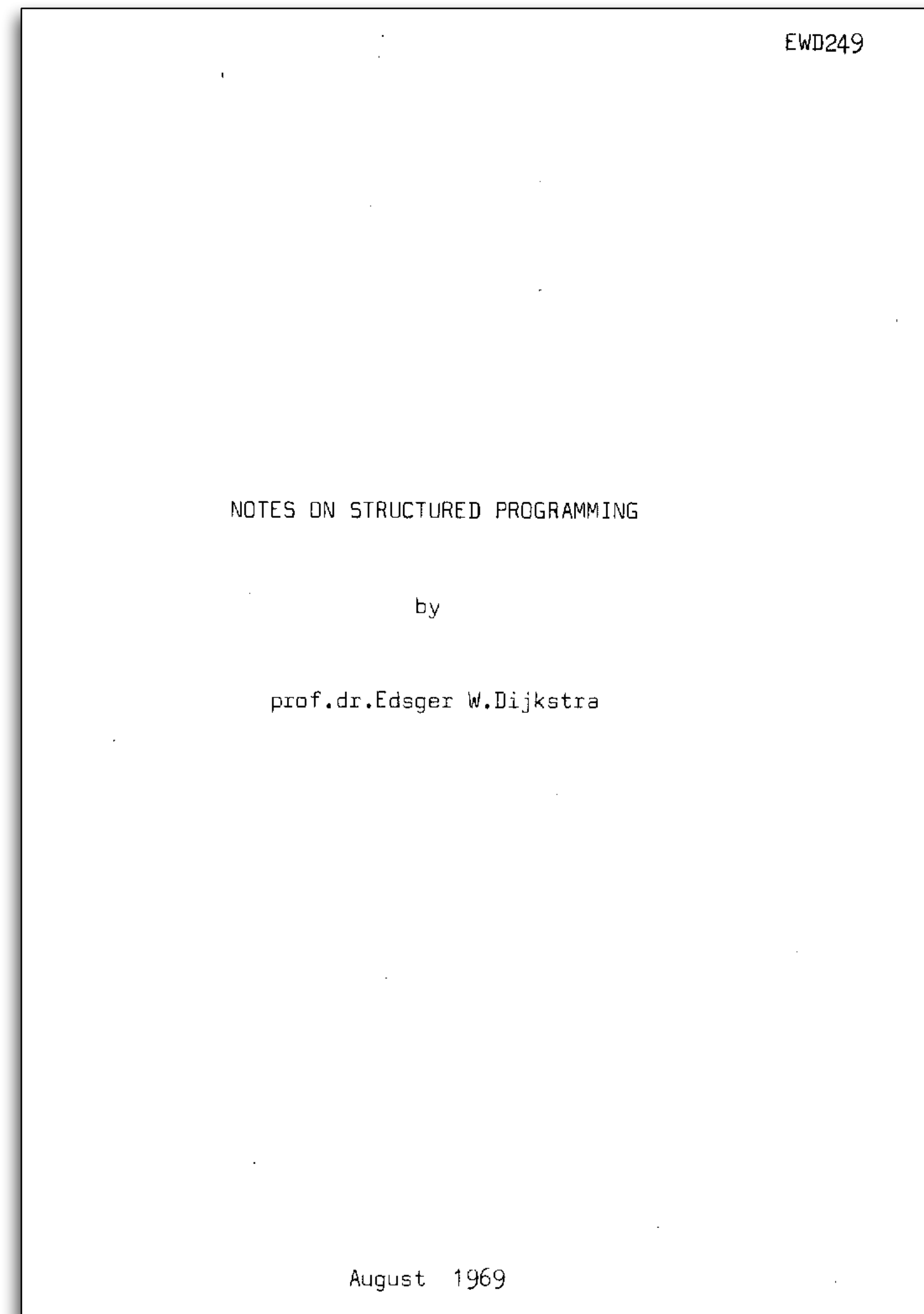

Structured Programming



Edsger Dijkstra

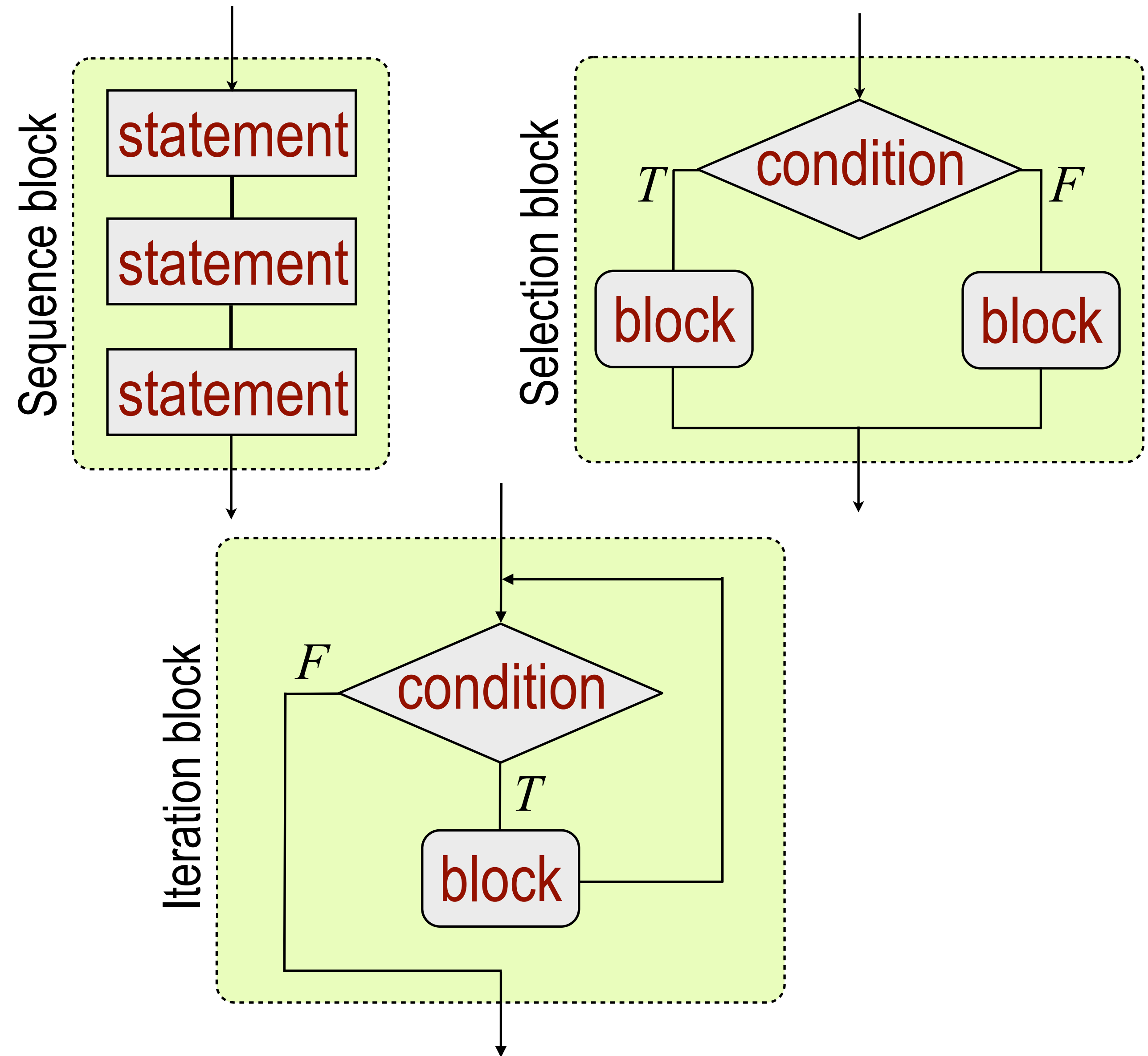
The competent programmer is fully aware of the strictly limited size of his own skull and therefore approaches the programming task in full humility.

Structured Programming



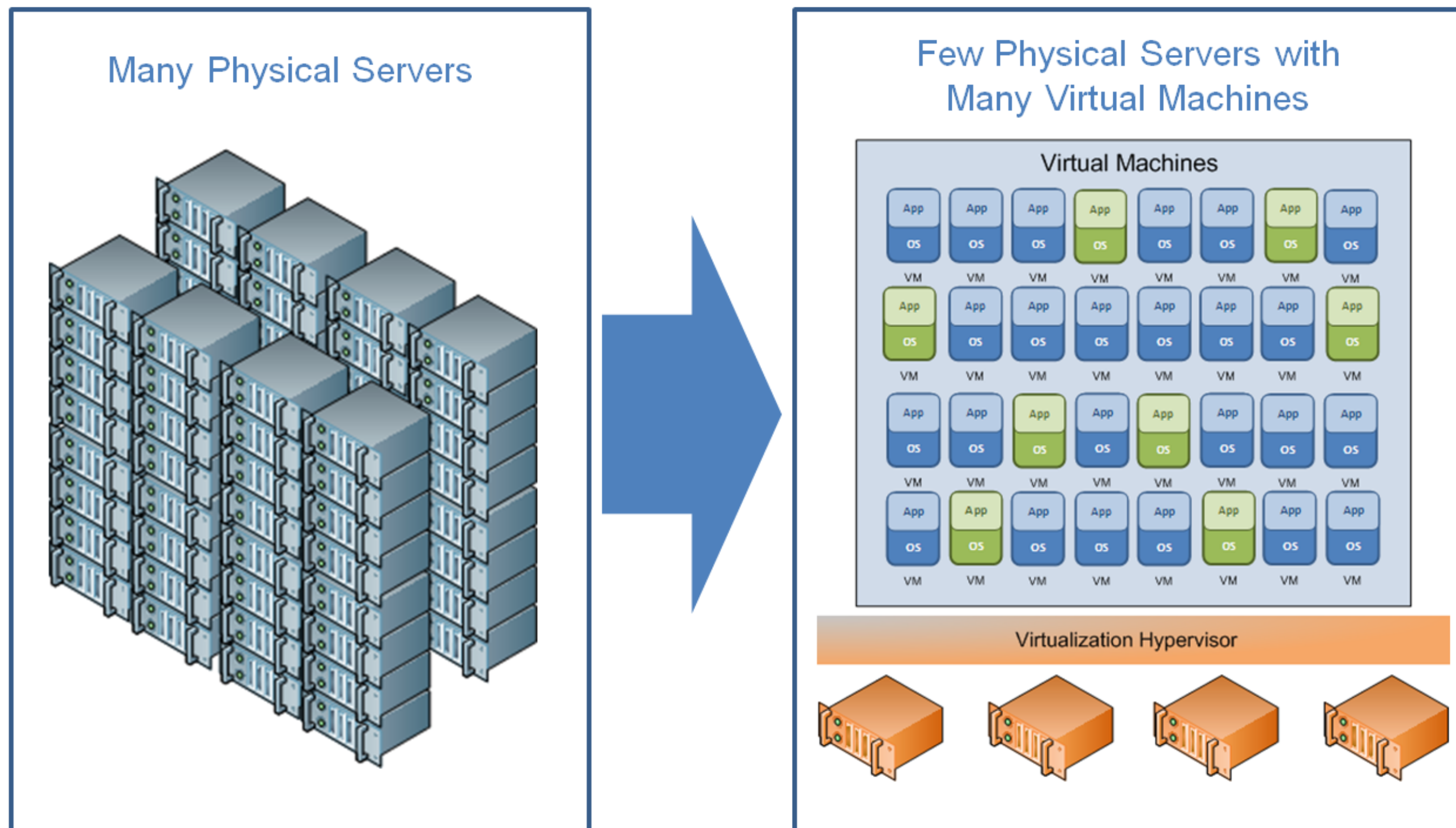
Structured Programming

- 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 $\Rightarrow$ higher quality*



Modularity Through Virtualization

- Virtual machines



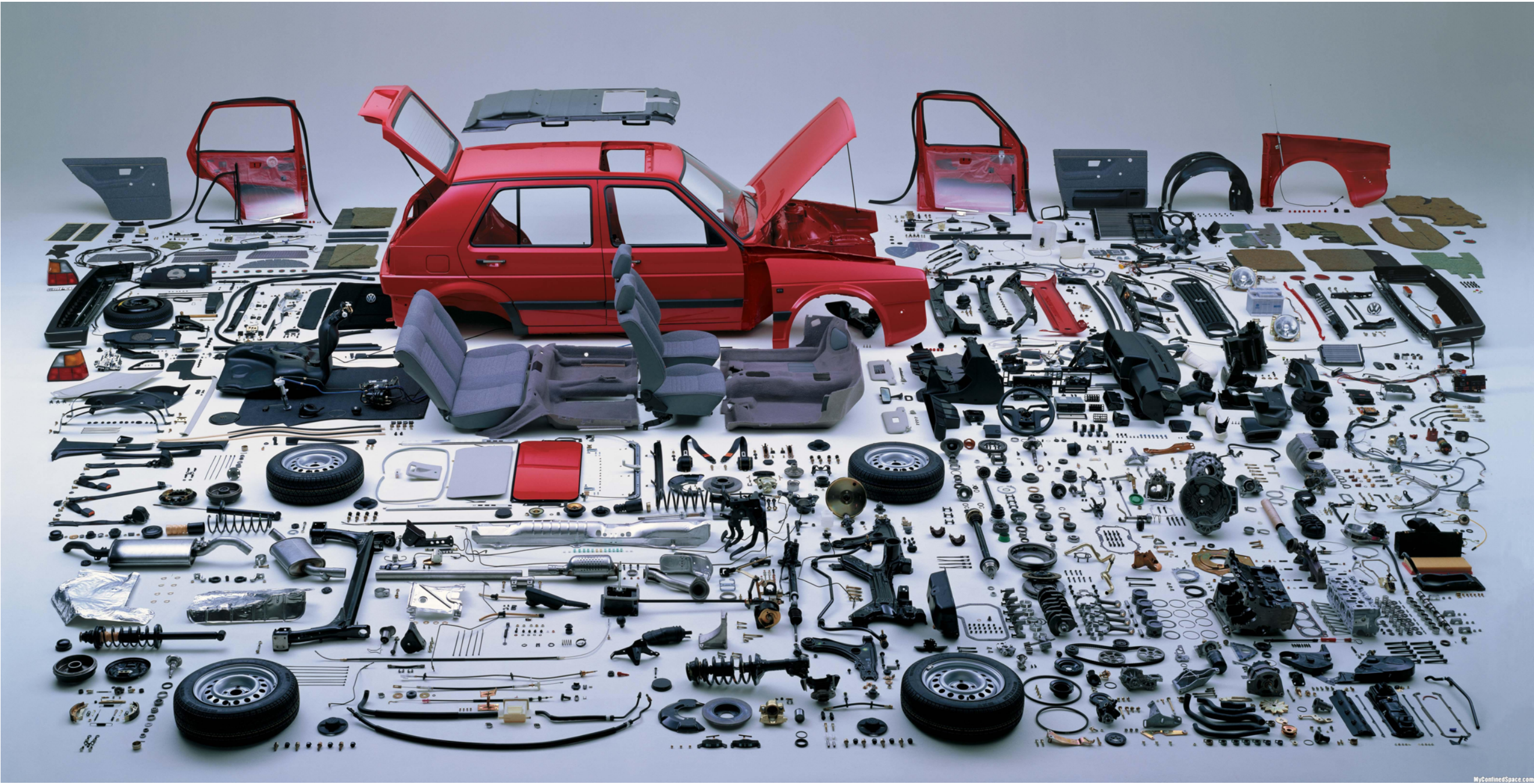
- Containers

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

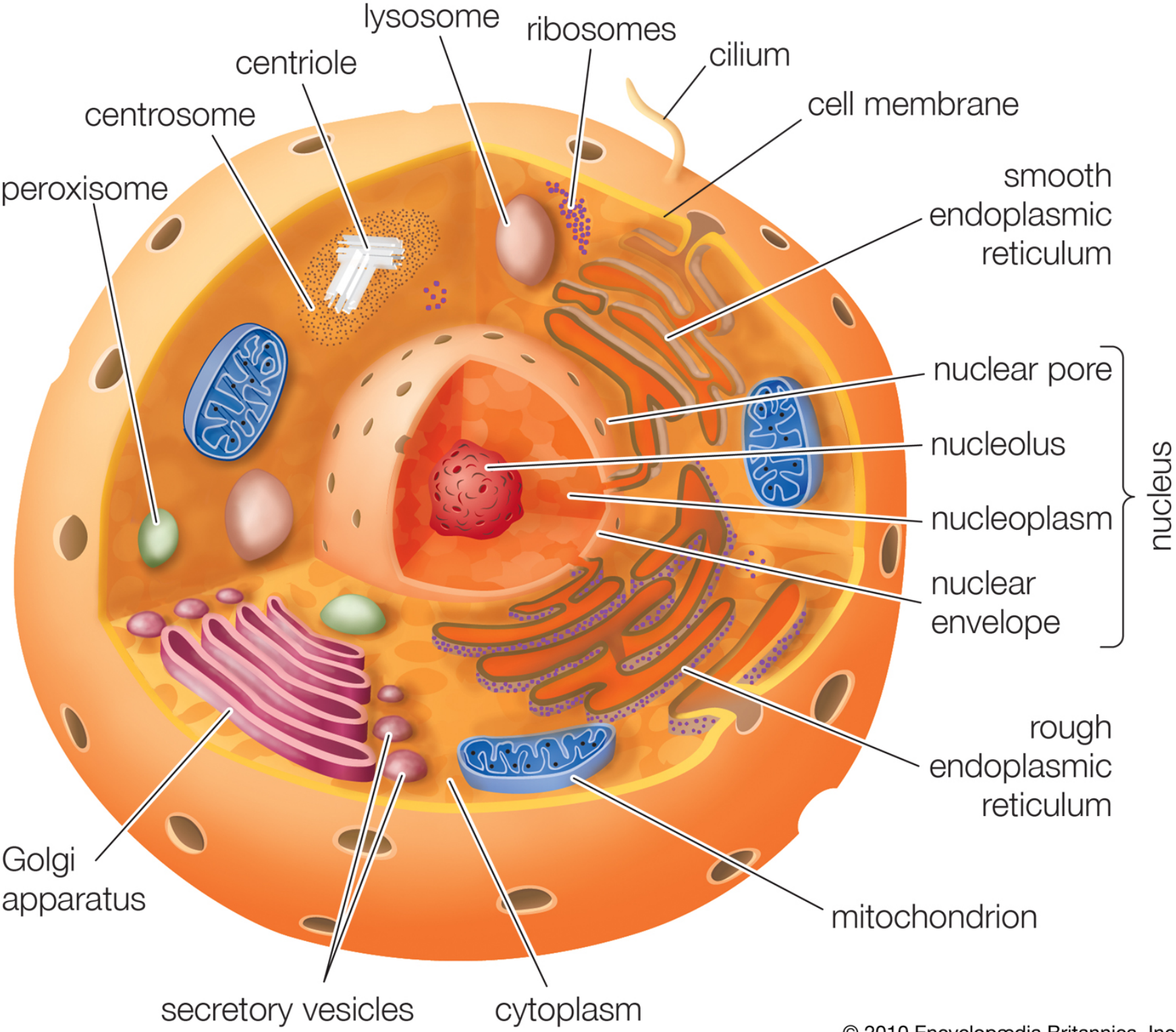
What is Modularity ?

- Isolate behavior into “boxes”
- Controlled entry/exit points
 - *replace components without affecting rest of system*
- Criterion for defining module boundaries
 - *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



© 2010 Encyclopædia Britannica, Inc.

Use Abstraction to Simplify and Regularize Behavior

Modularity

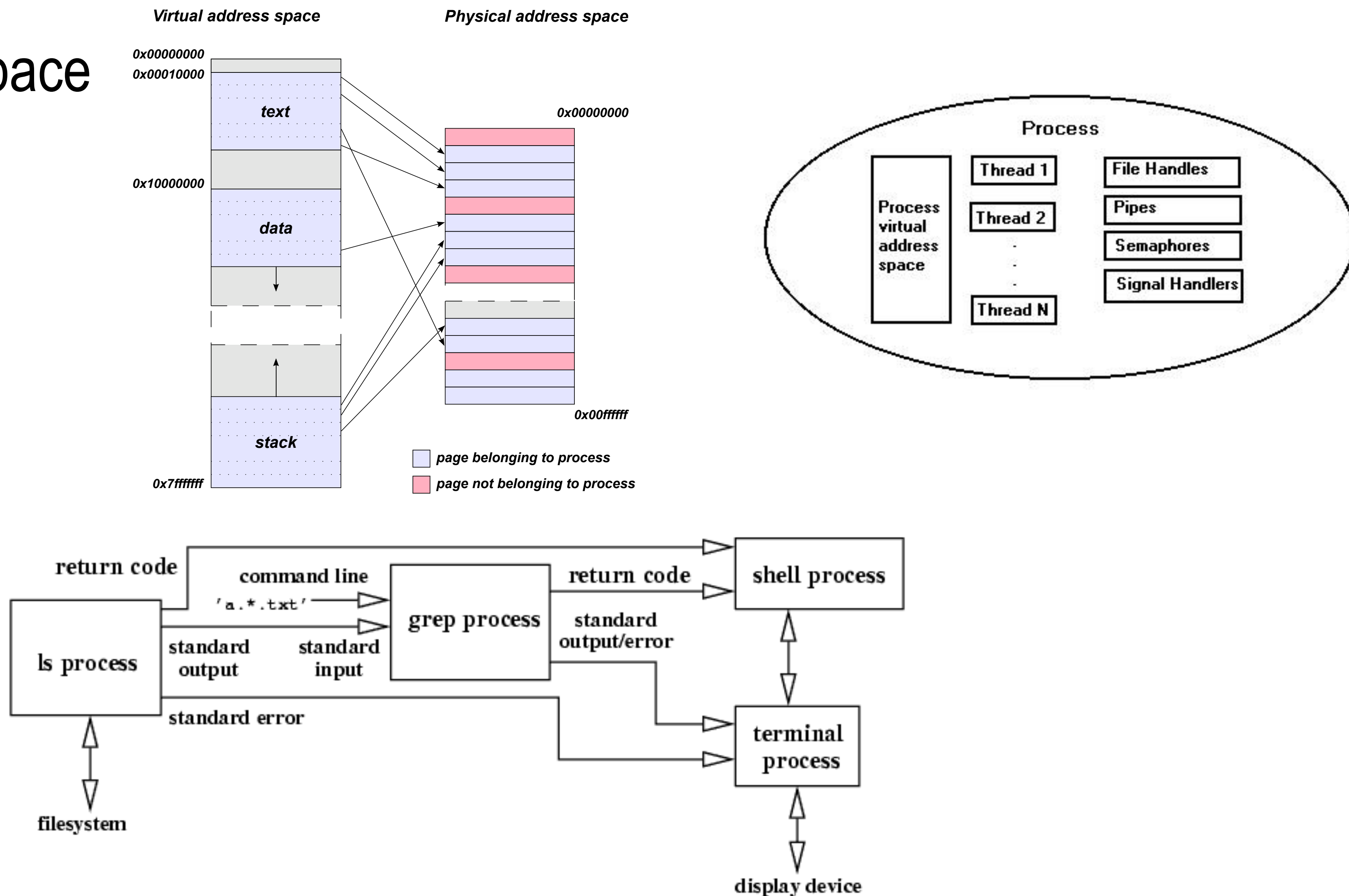


Modularity



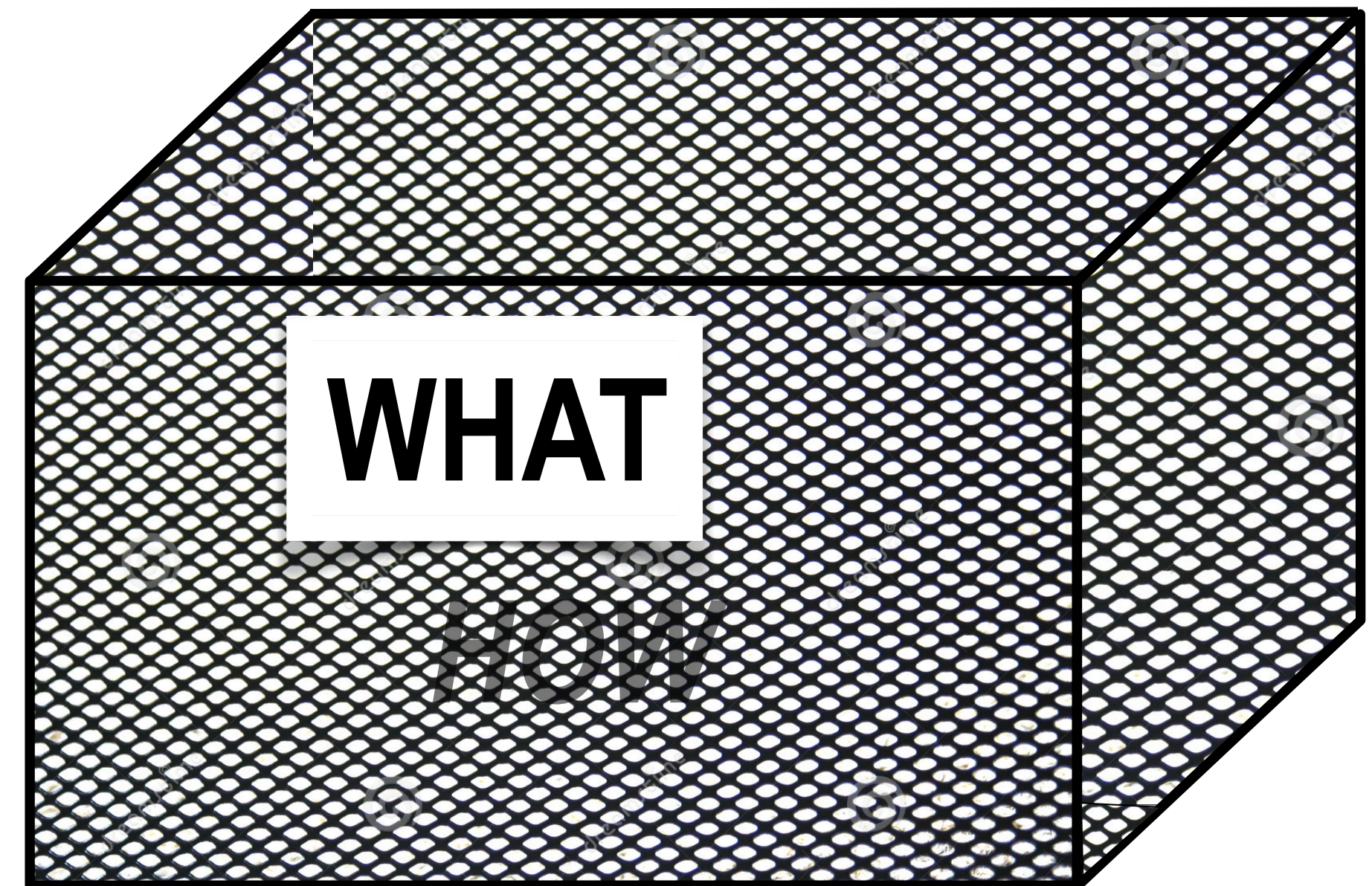
Examples of Abstractions in Operating Systems

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



Abstraction = Interface + Modularity

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



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

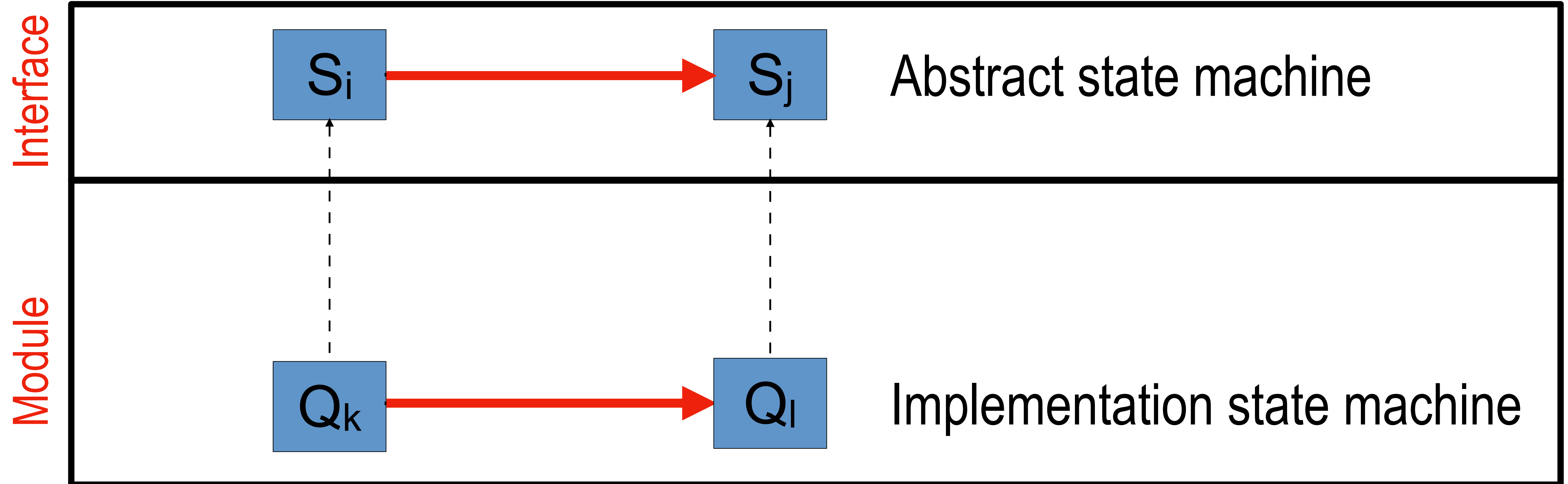
`IllegalBlockingModeException` - if this socket has an associated channel, and the channel is in non-blocking mode

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

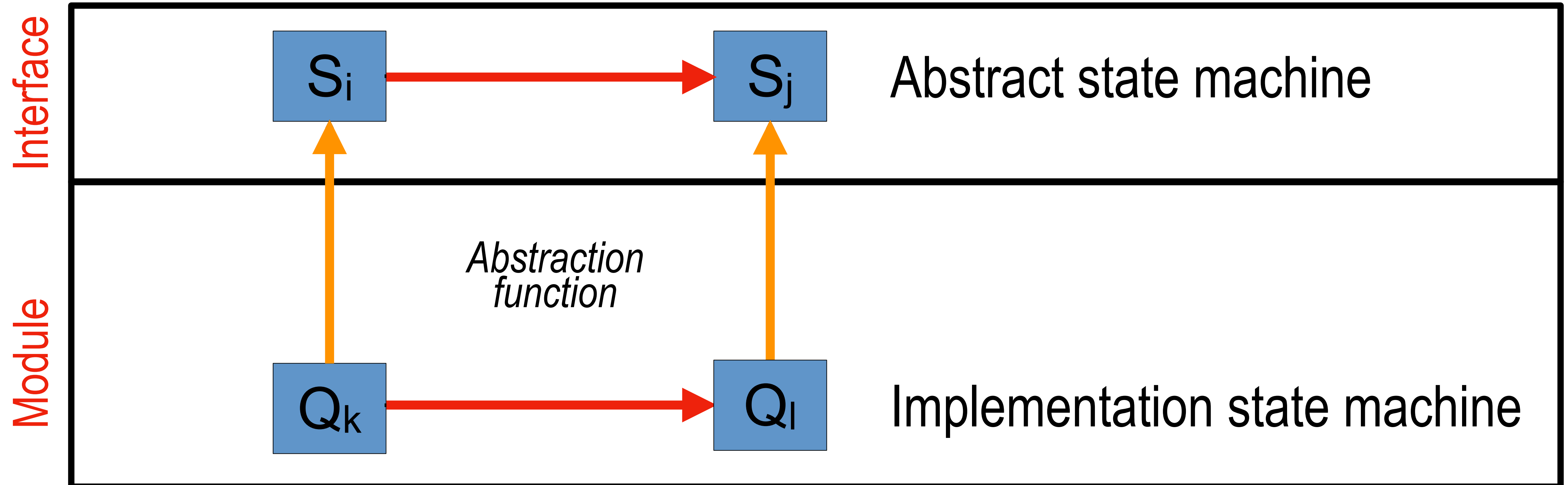
Since:

1.4

Implementation "simulates" the Abstraction



Implementation "simulates" the Abstraction



How To Modularize & Abstract?

- Abstraction vs. modularity
 - *these two are often considered the same, but they are 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++*

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/>

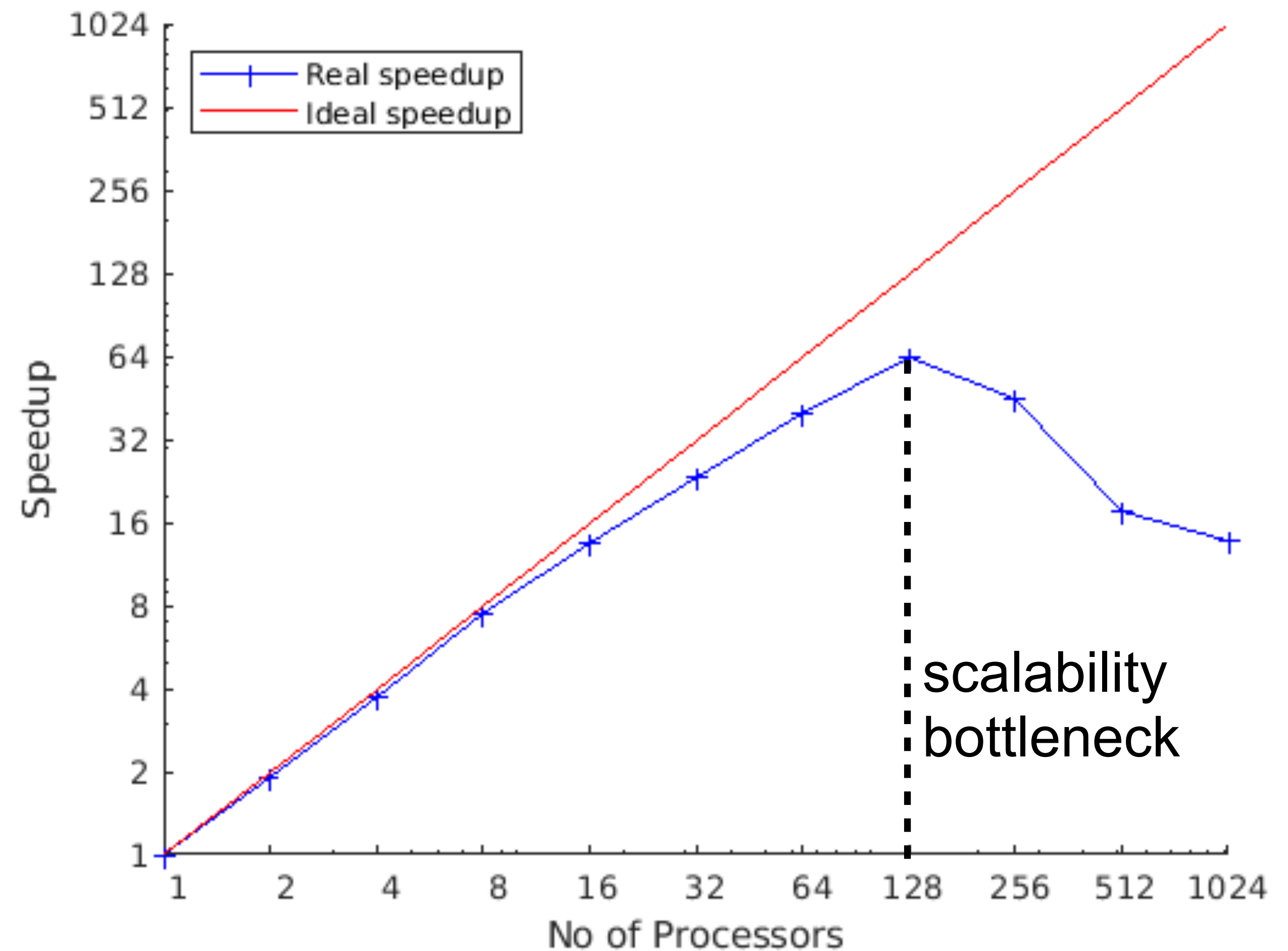
The Scalable Commutativity Rule

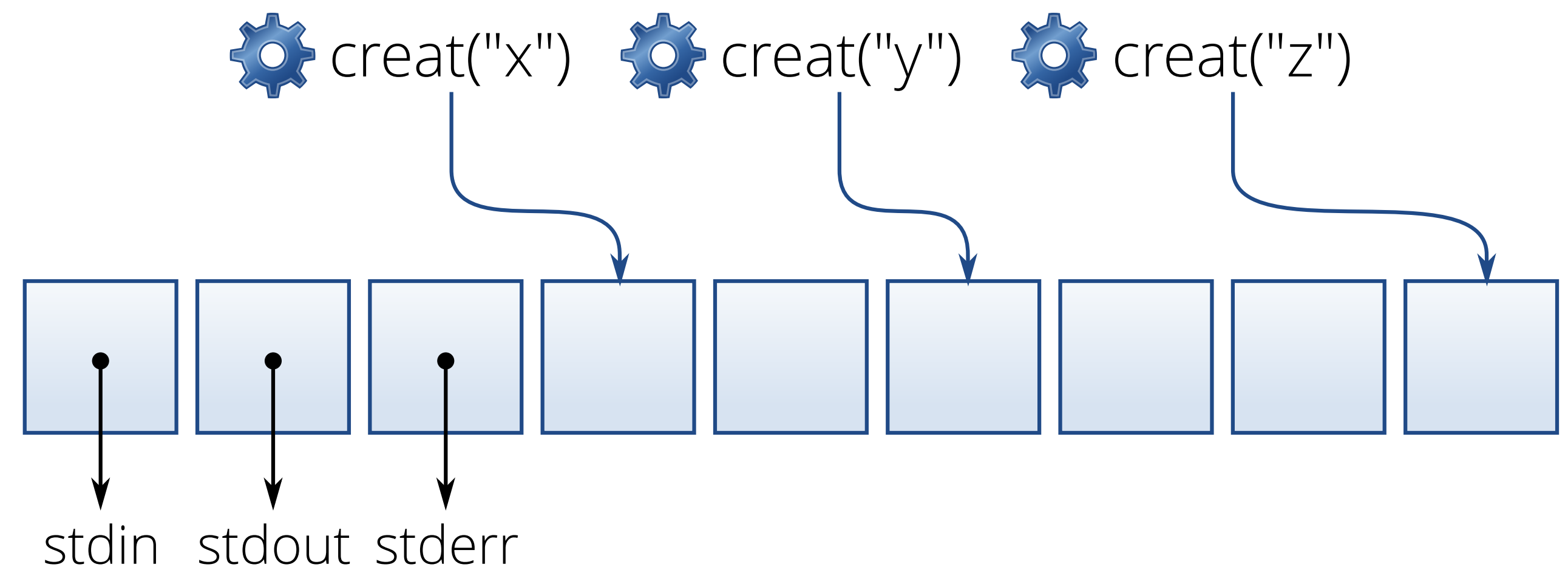
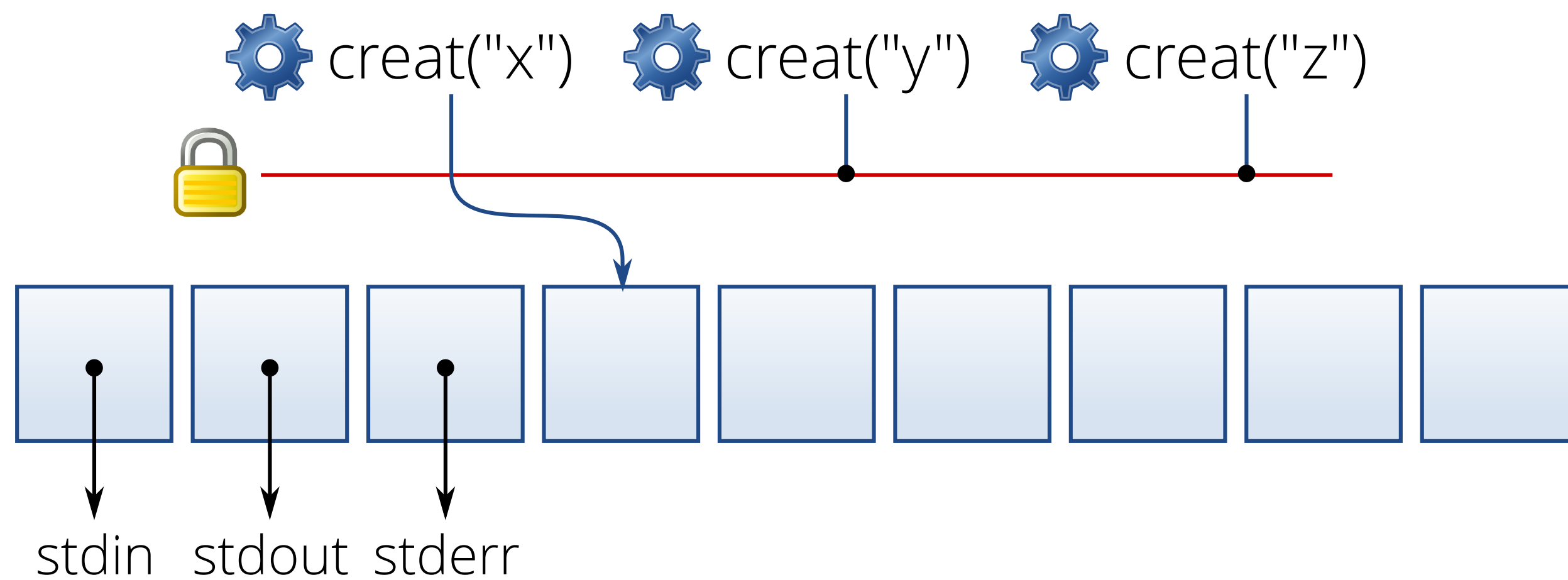
T. Clements et al, *The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors*, SOSP 2013

What is scalability ?

Ability to perform additional work given greater hardware resources

Good scalability => ability grows linearly with hw resources

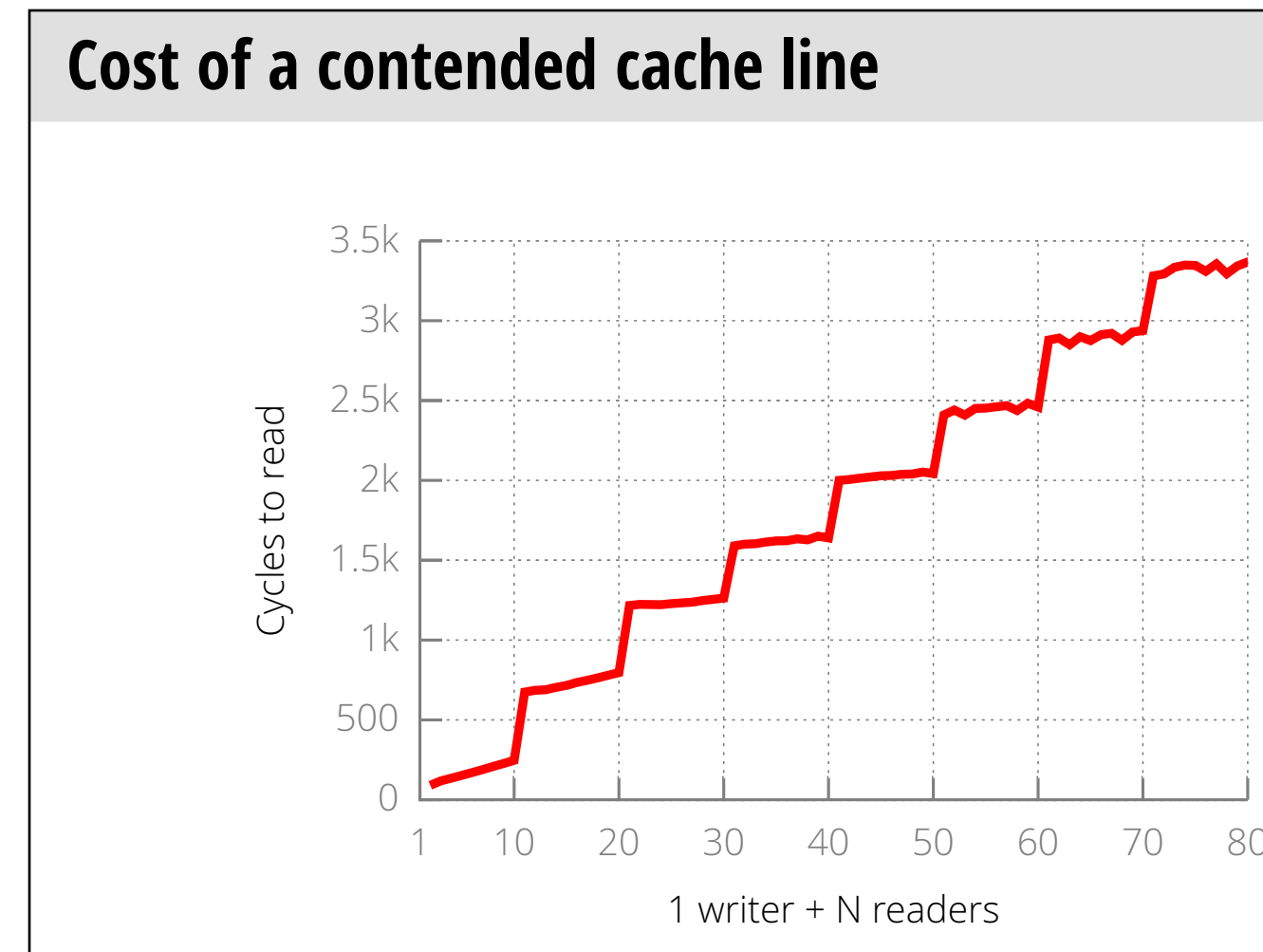




SC Rule: Whenever interface operations commute, they can be implemented in a way that scales.

Intuition behind rule (in the multicore context)

		Core X		
		W	R	-
Core Y	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



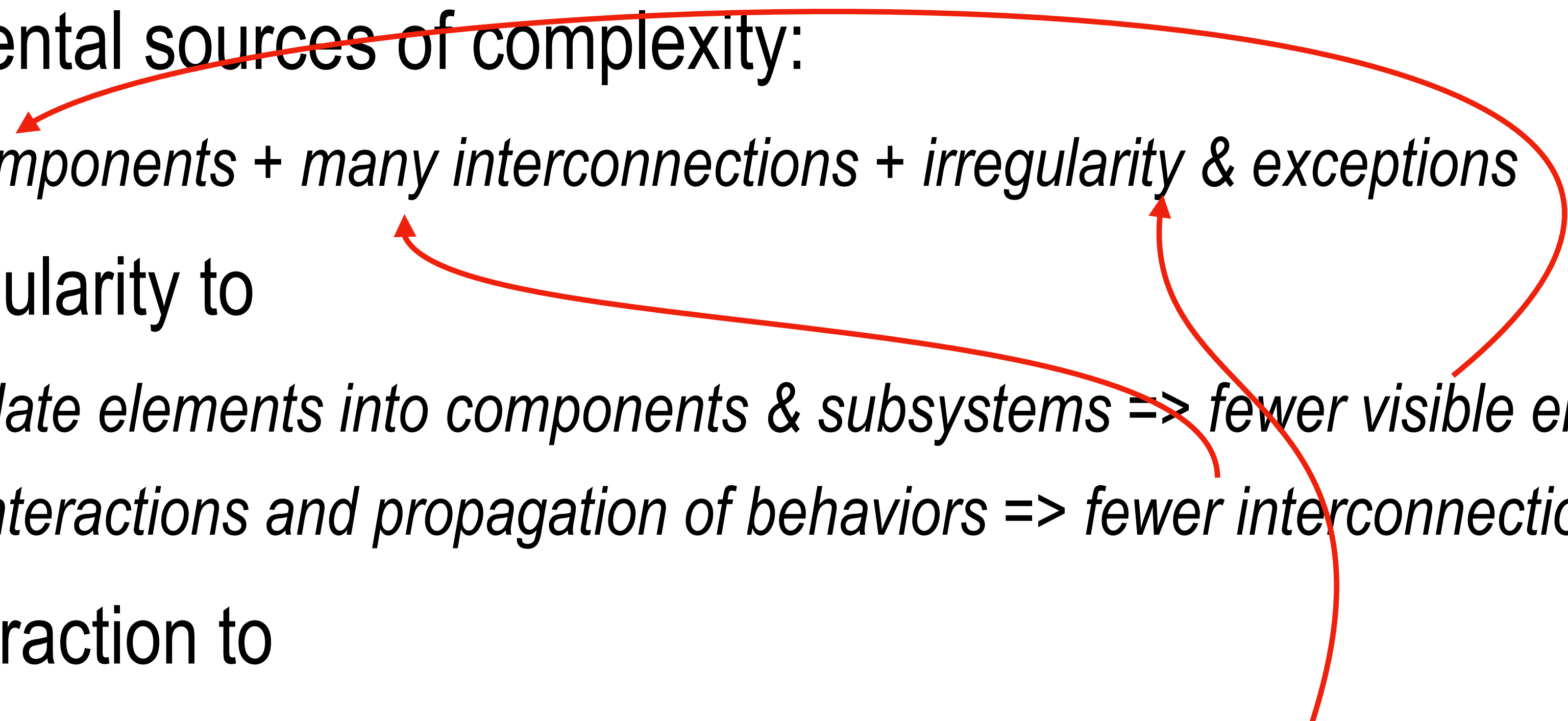
improve POSIX scalability

- 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, ...)*
- 
- Three red arrows originate from the 'Fundamental sources of complexity' list item. One arrow points from 'many components' to 'encapsulate elements into components & subsystems'. Another arrow points from 'many interconnections' to 'control interactions and propagation of behaviors'. A third arrow points from 'irregularity & exceptions' to 'make emergent behavior predictable'.

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 applications 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 (e.g., [1, 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]).