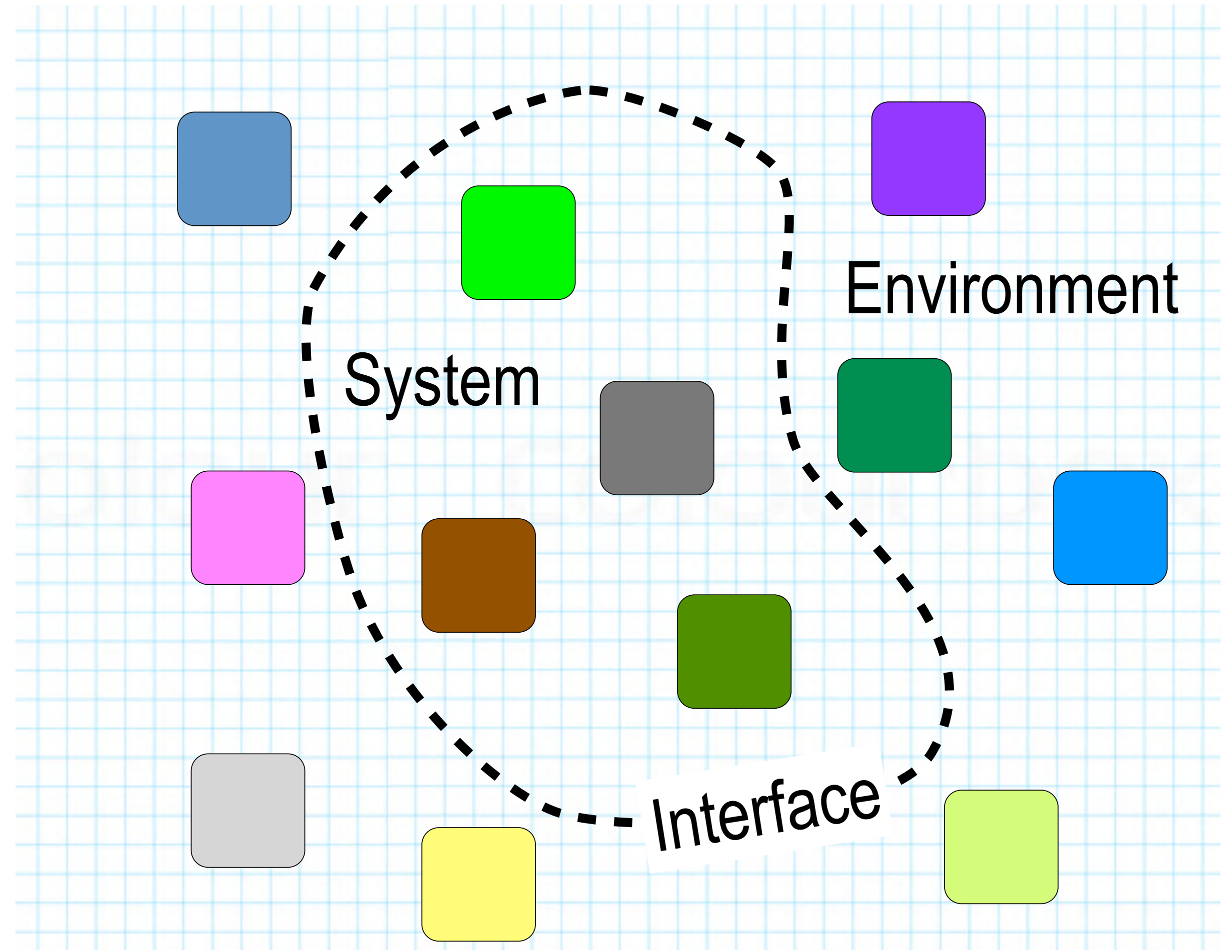
Why Is A Computer System ?

Prof. George Candea
School of Computer & Communication Sciences

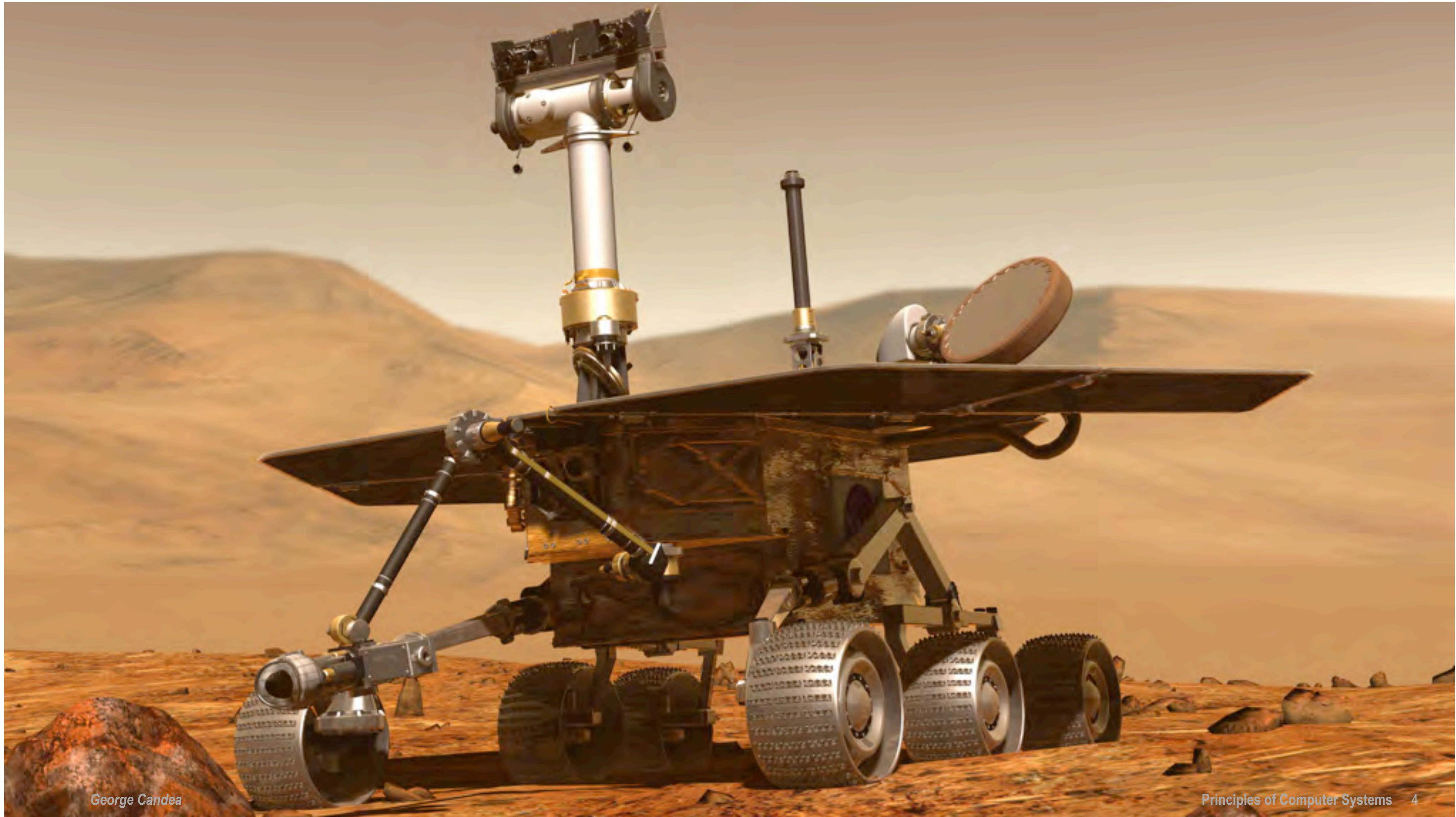
What is a Computer System ?

A group of interconnected components that exhibits an expected collective behavior observed at the interfaces with its environment.

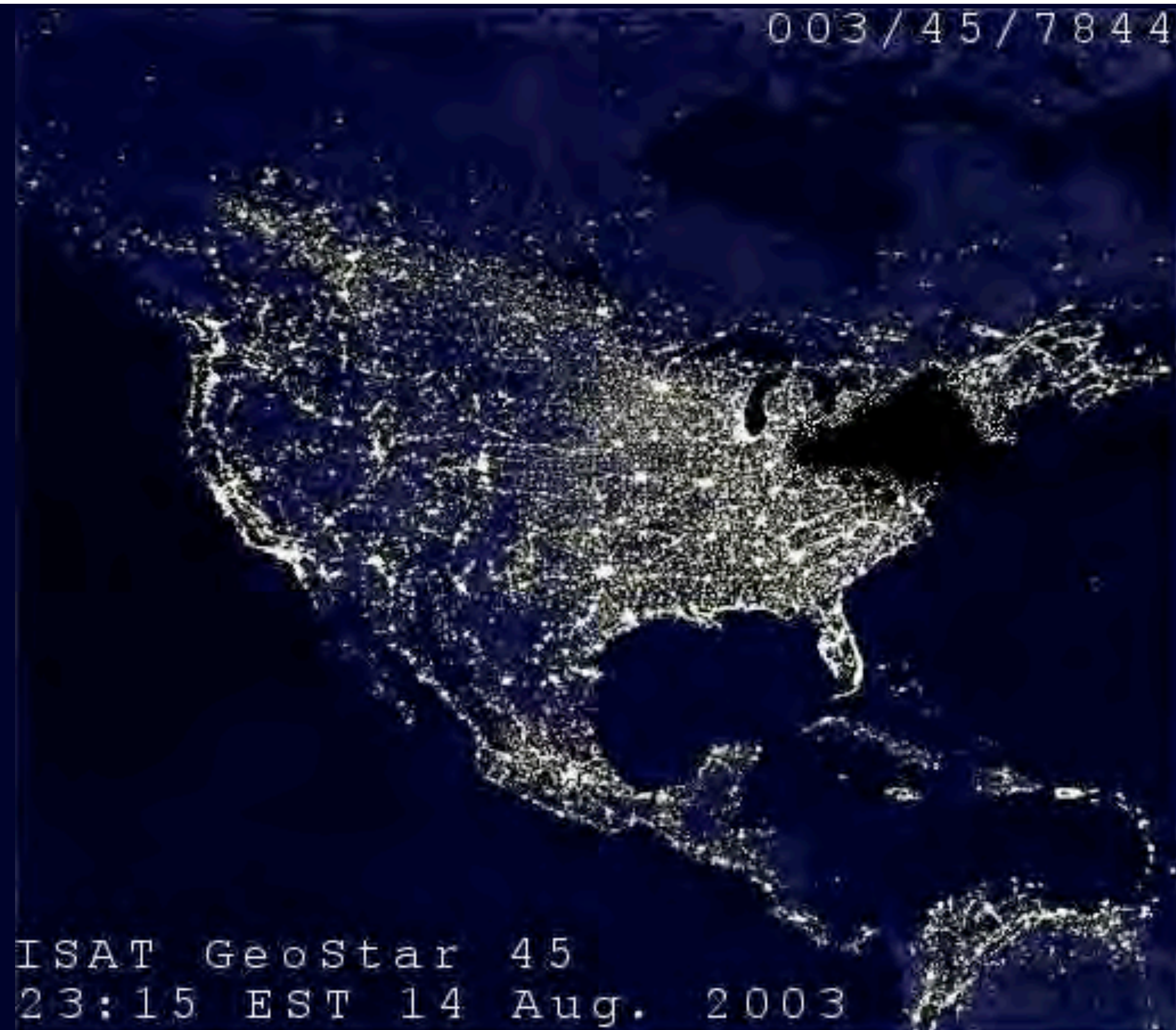


Examples of Systems

- Examples of systems
  - *a smartphone or tablet*
  - *the Internet*
  - *ticket reservation system*
  - *shared calendaring system*
  - *air traffic control*
  - *an e-commerce site*
  - *smart home*
  - *self-driving cars*

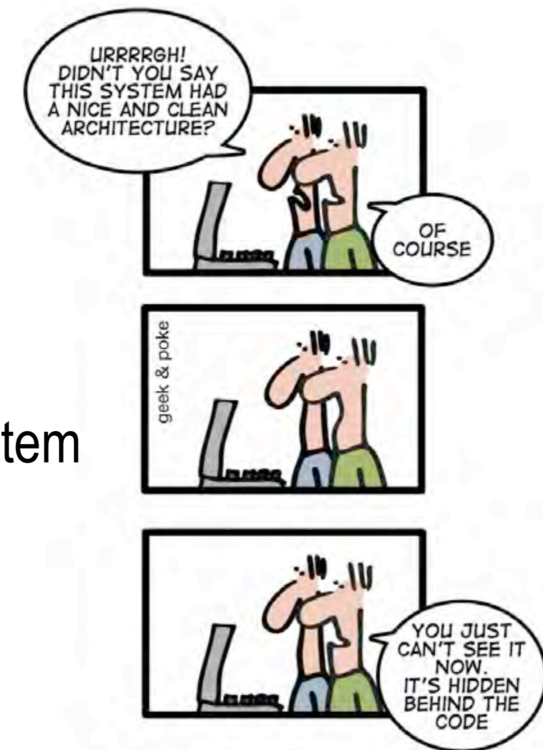


UPS6944
N27DUP
UPS6931
KLM1033
B737 PH-BXF
EHAM-EGLL
UPS6727
NO-REG
ARN8293
COA67
NO-REG
ARN8745
CGTBP
CFZJA
ARN8716
DAL1898
B757 N671DN
CGBKB
CGBKB
HSA798
HSA90
B737 C-FKHS
HSA5002
B737 C-FZHS
AMT4625
B757 N551TZ
NHA210
NO-REG
SAA210
NO-REG
THA774
NES3525
N506XJ
HAL1
NO-REG
KLAX-PHNL
NHA433
NO-REG
HSA699
C-FHSO
HSA803
C-GZHS
UAL921
B767 N642UA
NHA319
NO-REG
COA97
NO-REG
EGLL-KIAH
UAL951
B767 N643UA
HSA197
C-FBWJ


003/45/7844
ISAT GeoStar 45
23:15 EST 14 Aug. 2003

What Is A Computer System ?

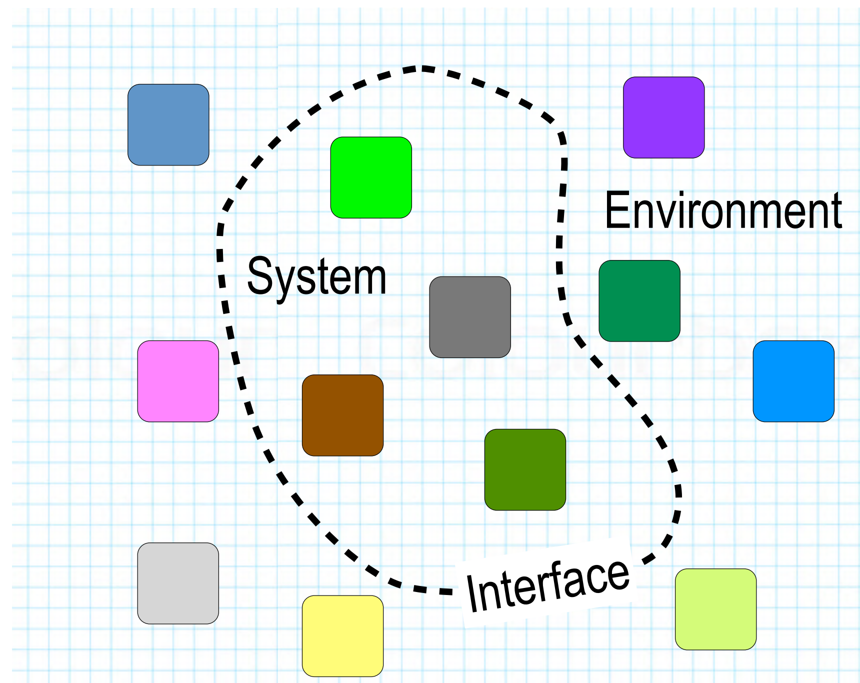
- Three parts
  - *system components*
  - *surrounding environment*
  - *interface*
- Litmus test of good system
  - *predict system behavior from component-level behaviors ?*



Properties of Computer Systems

Prof. George Candea
School of Computer & Communication Sciences

Computer Systems

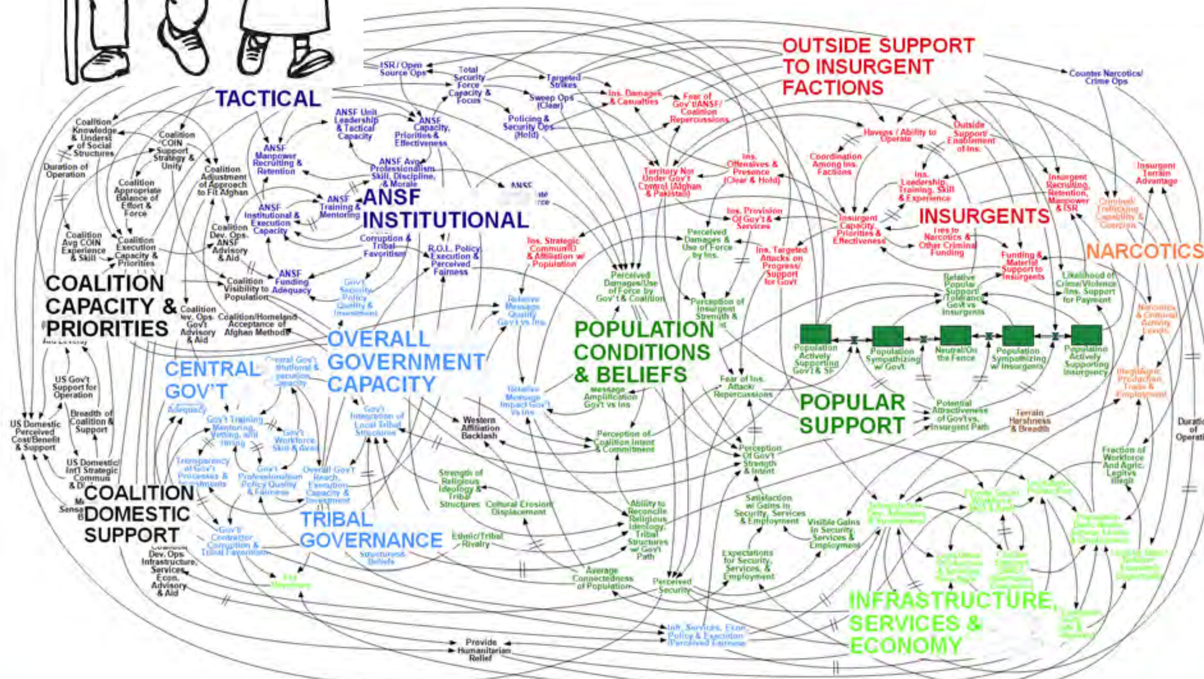


- A system is ...
  - *a group of interconnected components that exhibits an expected collective behavior observed at the interfaces with its environment*
- Good system design ⇒
  - *predict system behavior from component-level behaviors*
- Elemental components are often complex
  - *a component is itself a (sub)system*

Computer Systems

- A system is ...
  - *a group of interconnected components that exhibits an expected collective behavior observed at the interfaces with its environment*
- Good system design ⇒
  - *predict system behavior from component-level behaviors*
- Elemental components are often complex
  - *a component is itself a (sub)system*

Component vs. System



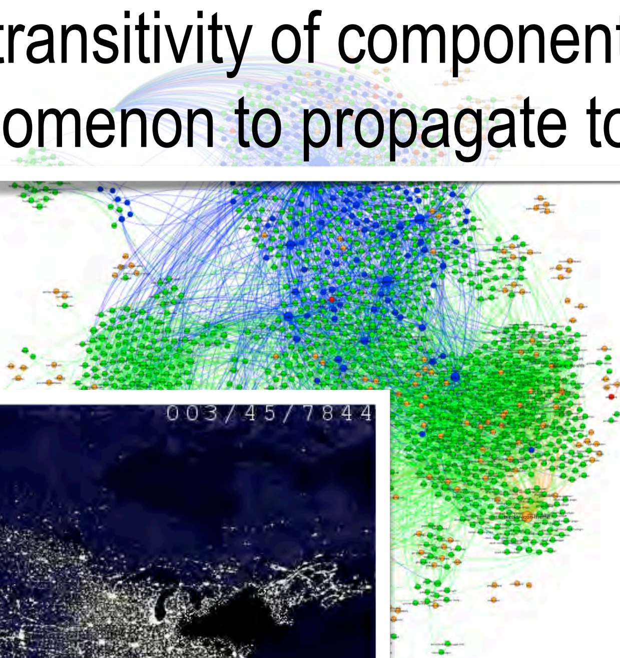
- Four distinguishing characteristics
  - *emergent properties*
  - *propagation of effects*
  - *incommensurate scaling*
  - *inevitable trade-offs*

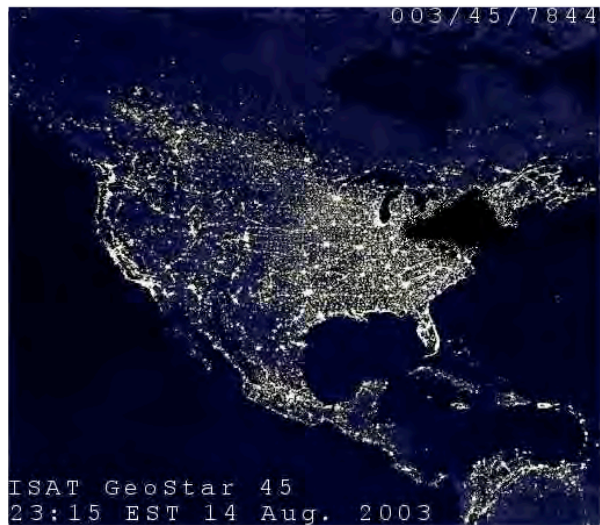
#1: Emergent Properties

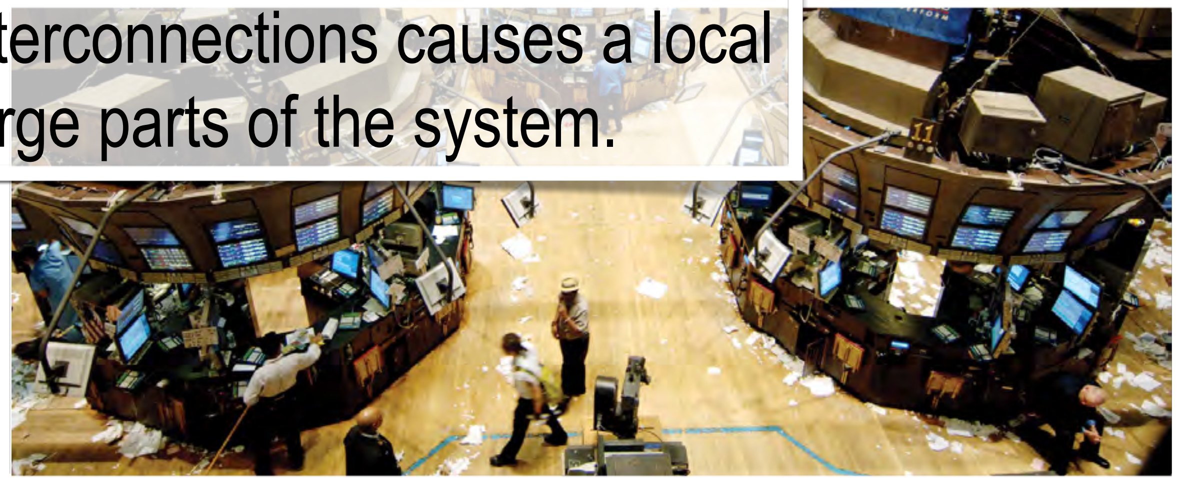
Properties that are not evident in the components, but appear when the components are combined.

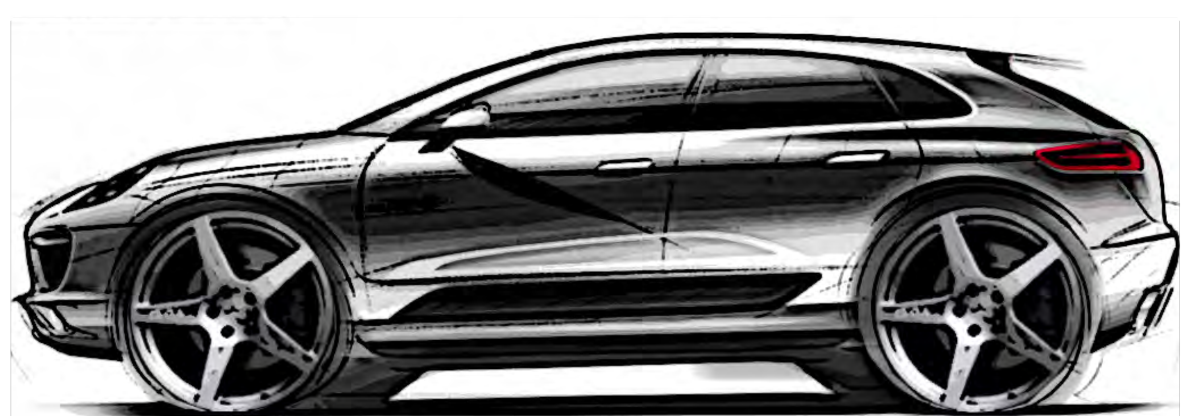
#2: Propagation of Effects

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



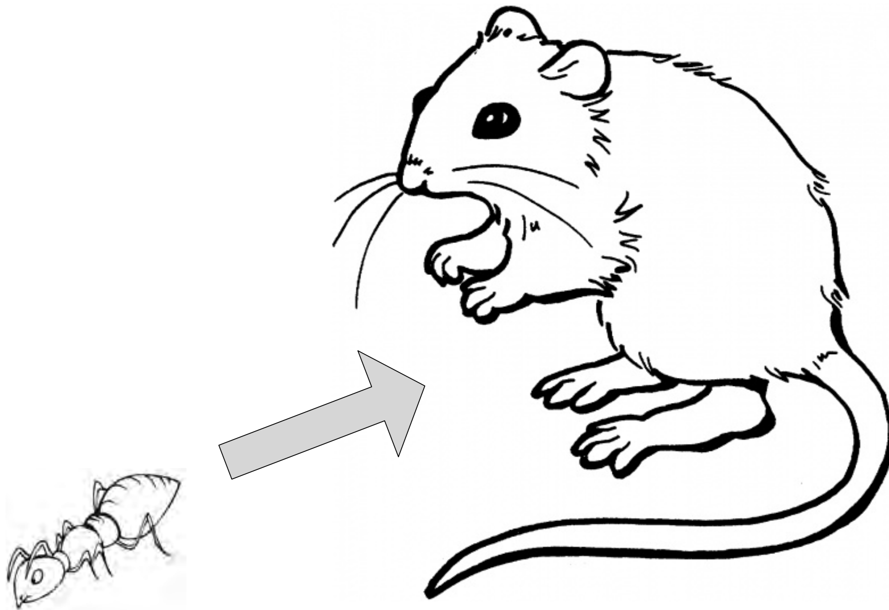




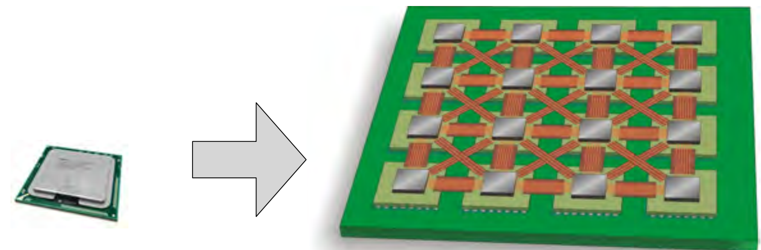


#3: Incommensurate Scaling

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

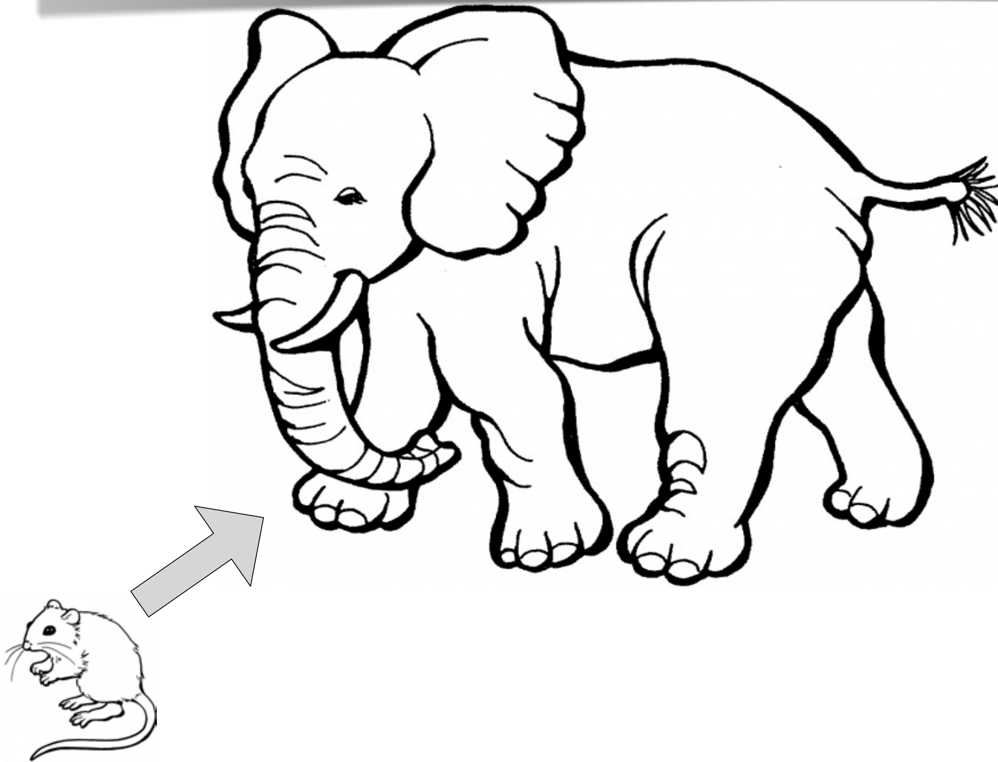


- Reason
  - *Scalability of each component is described by a function*
  - *The order of these functions is not the same for each component ⇒ as system grows, components scale disproportionally to each other*

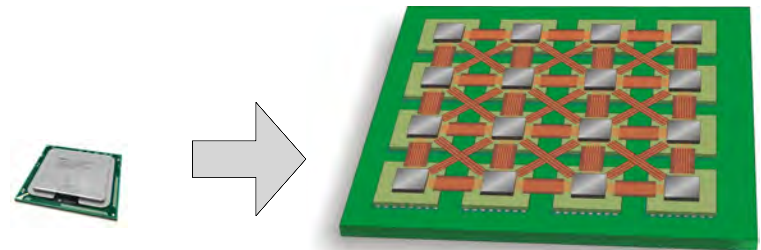


#3: Incommensurate Scaling

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



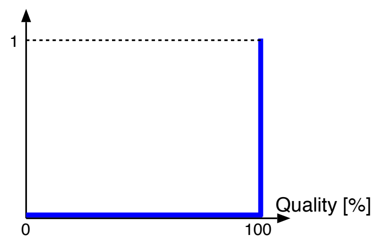
- Reason
  - *Scalability of each component is described by a function*
  - *The order of these functions is not the same for each component ⇒ as system grows, components scale disproportionally to each other*

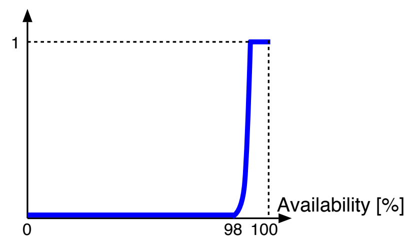


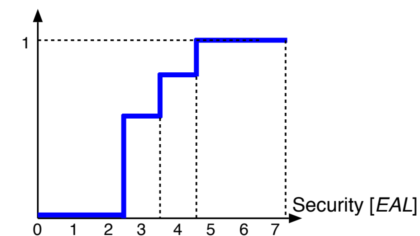
#4: Trade-Offs

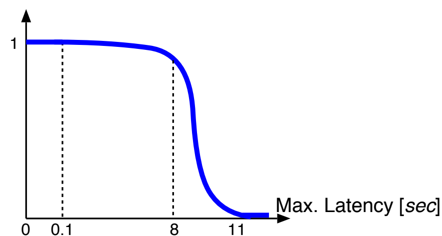
Designing a system consists of trading off properties against each other so as to maximize the system's overall utility.

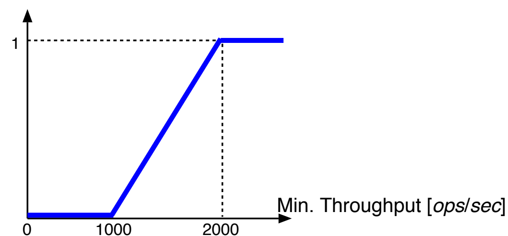
- *every system property has a utility curve*
- *a given design ties utility curves to each other*
- *configurations, implementations, and inputs each fix a point along the utility curve*

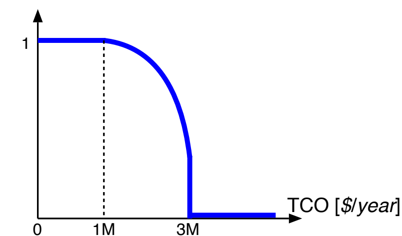












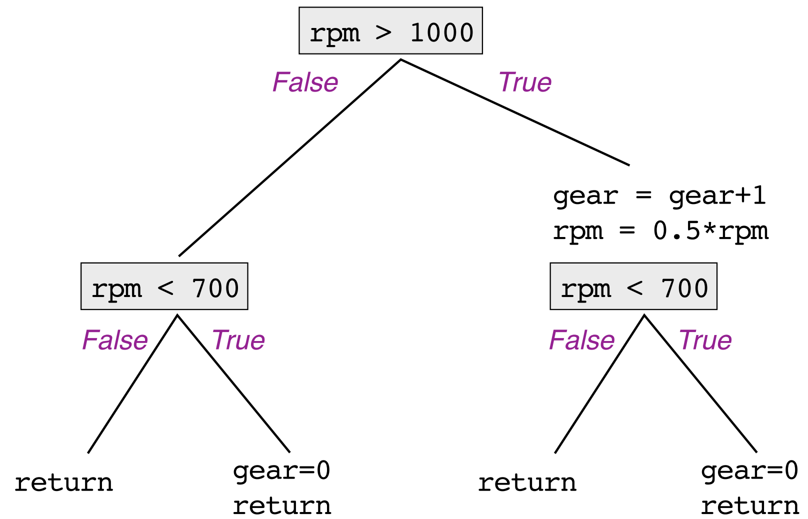
Properties of Computer Systems

- Four characteristics
  - *emergent properties*
  - *propagation of effects*
  - *incommensurate scaling*
  - *inevitable trade-offs*
- Synthesize a system from requirements
- Road ahead:
  - *representations, abstractions, and design ideas*
  - *combat the ad-hoc nature of system design*
  - *take a principled approach to the design process*

The Complexity Challenge

Prof. George Candea
School of Computer & Communication Sciences

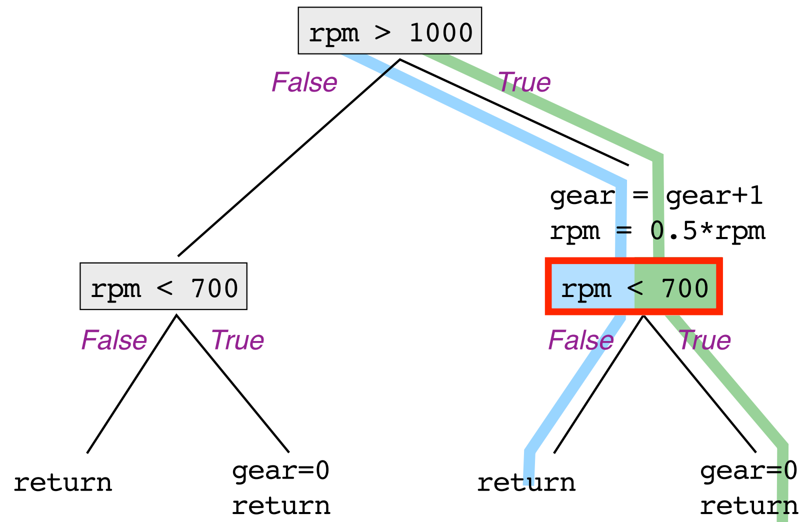
```
autoShift (int rpm)
  if (rpm > 1000)
     gear = gear+1
     rpm = 0.5*rpm
  if (rpm < 700)
     gear=0
  return
```



rpm=1800

rpm=1200

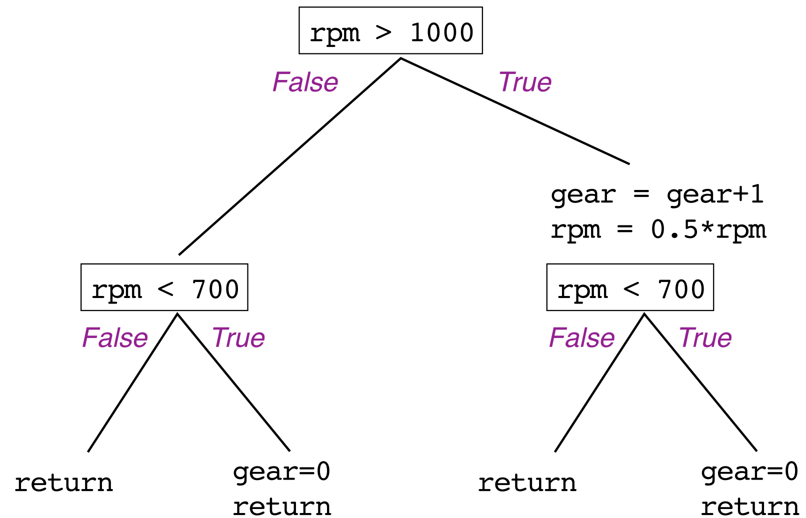
```
autoShift (int rpm)
   if (rpm > 1000)
      gear = gear+1
      rpm = 0.5*rpm
   if (rpm < 700)
      gear=0
   return
```



Software Testing

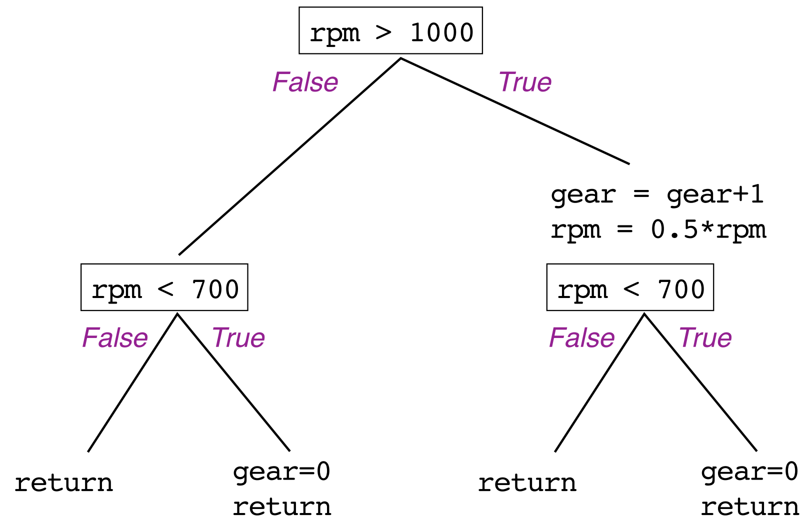
```
rpm ∈ {0,350,944,1200,1800}
```

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



Software Testing

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



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

Software Testing

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

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

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

- 30 picoseconds/test ⇒ $2^{499,968}$ years to finish
  - *planet Earth is $\sim 2^{32}$ years old*
- 1 bit/test ⇒ $2^{499,957}$ Terabytes to store the answer
  - *Universe contains $\sim 2^{266}$ atoms in total*

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

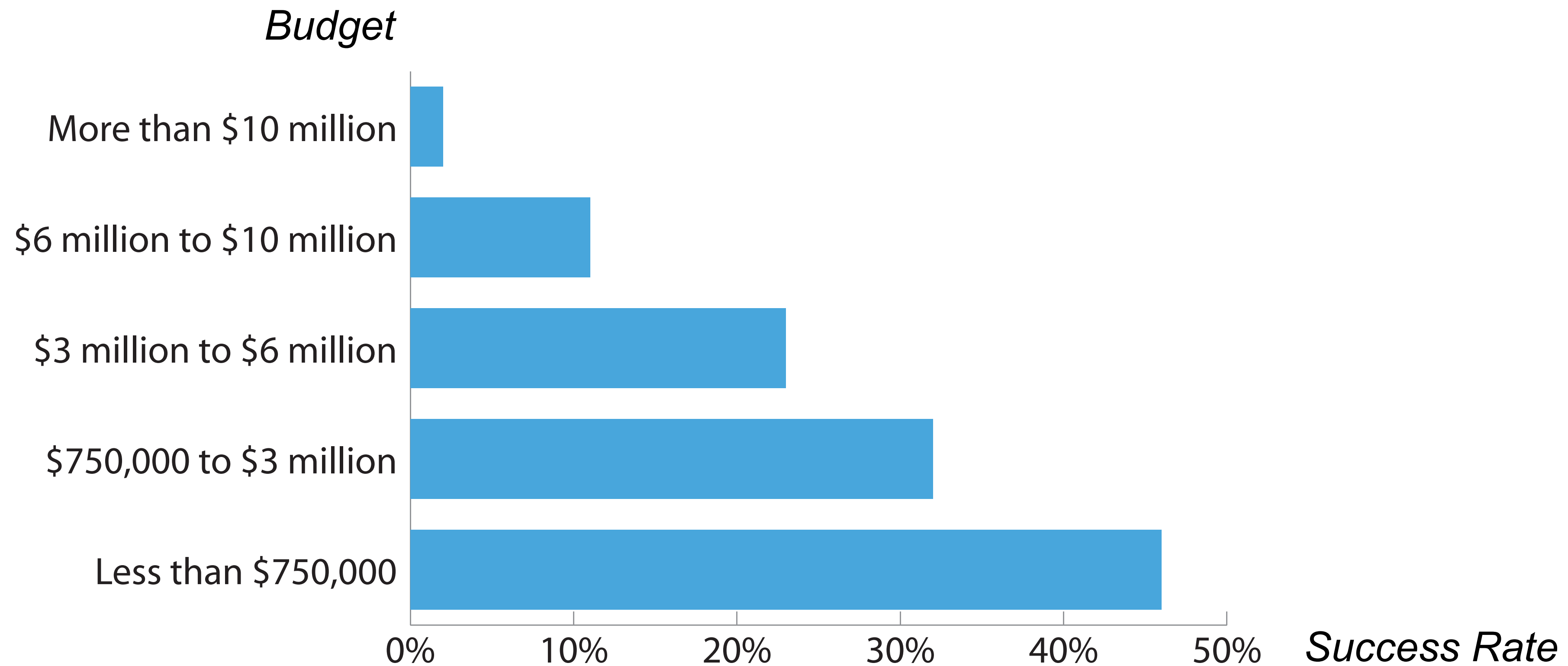
The Complexity Challenge

- Testing
  - *most basic form of software verification*
  - *naive approach does not offer a lot of confidence*
- Correct by construction ?
  - *mathematical view of software engineering*
- Proofs of correctness ?
  - *write code and prove it correct before running it*

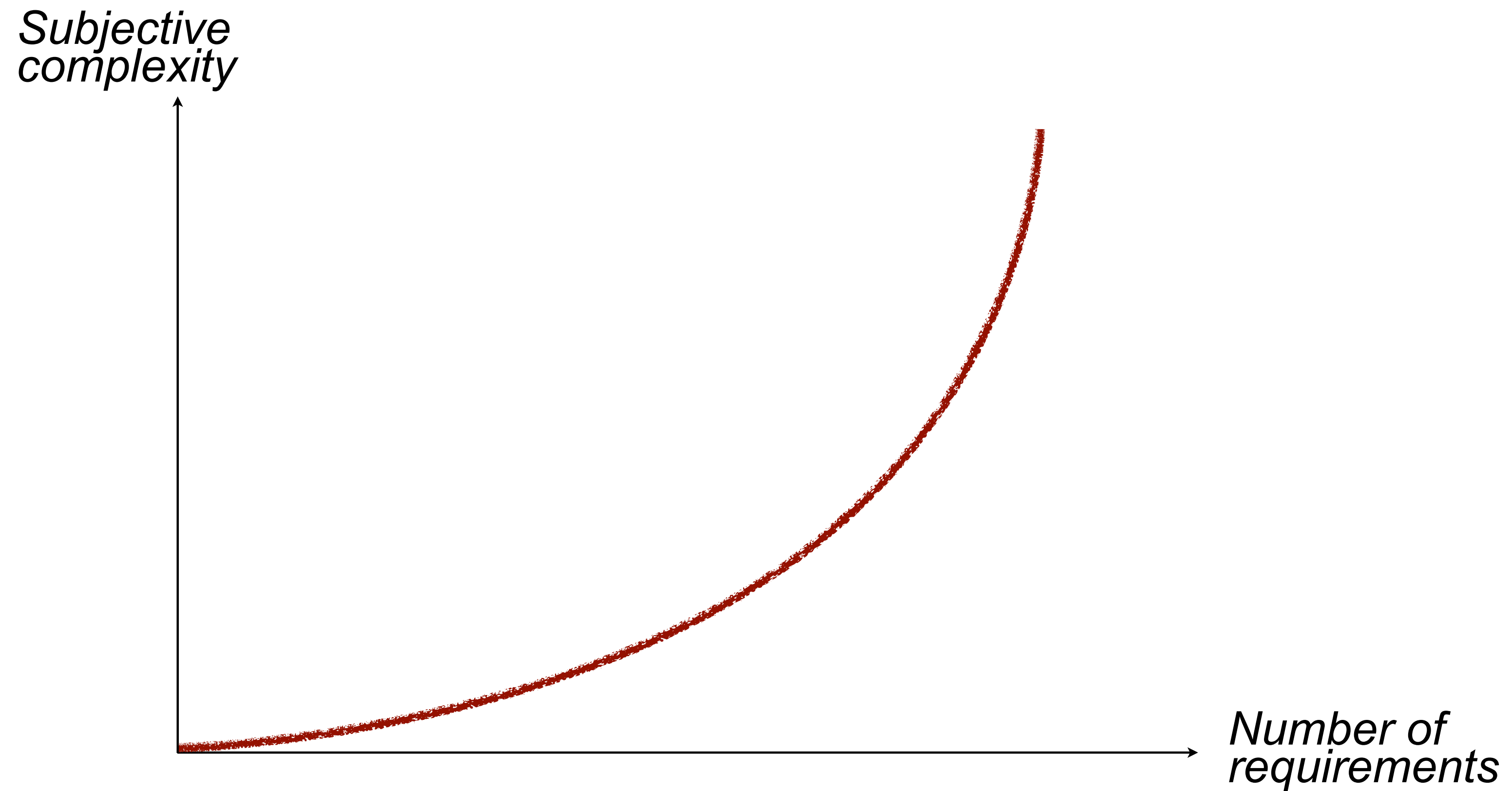
Sources of Complexity

Prof. George Candea
School of Computer & Communication Sciences

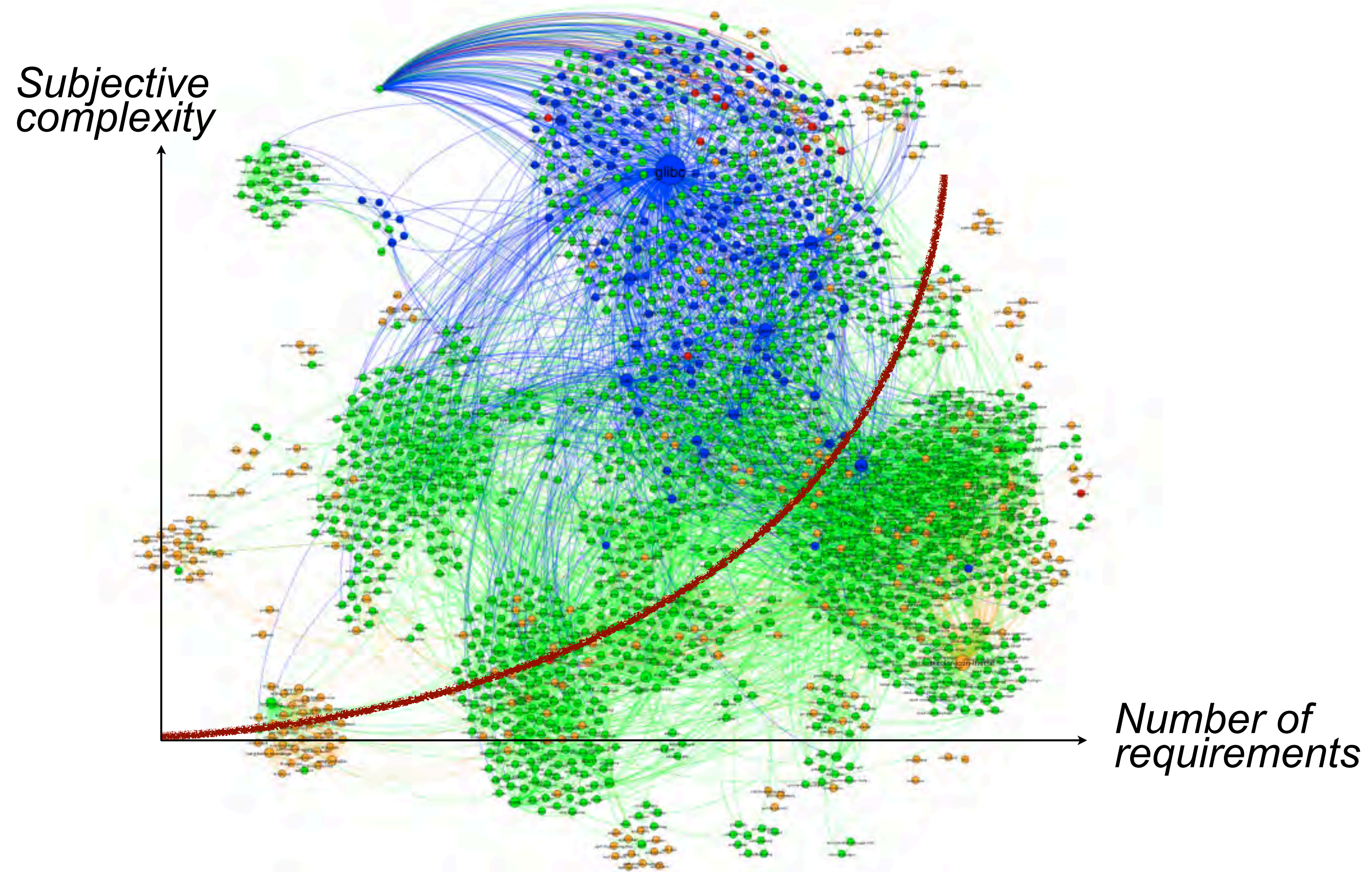
Software Projects Success Rate = *f* (Budget)



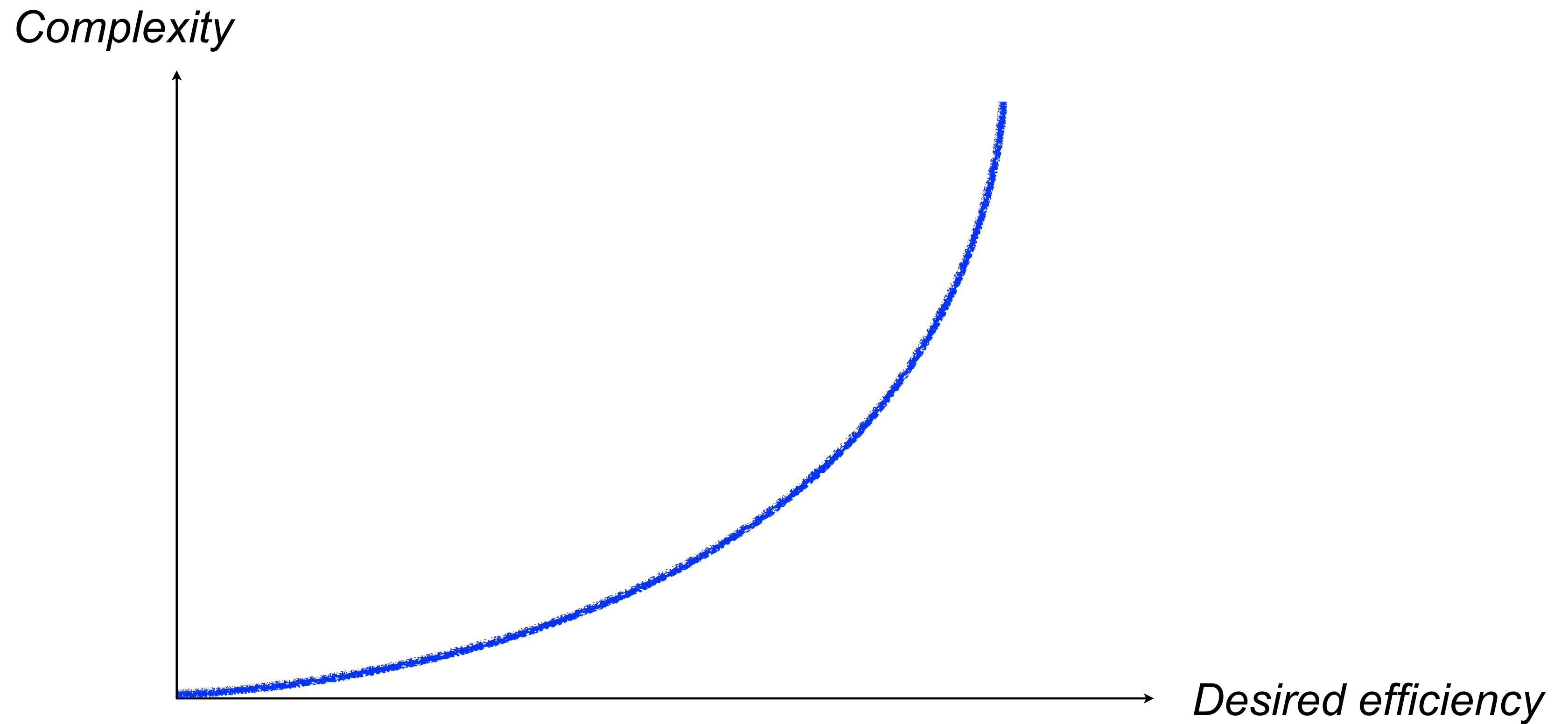
Many Requirements ⇒ Complexity



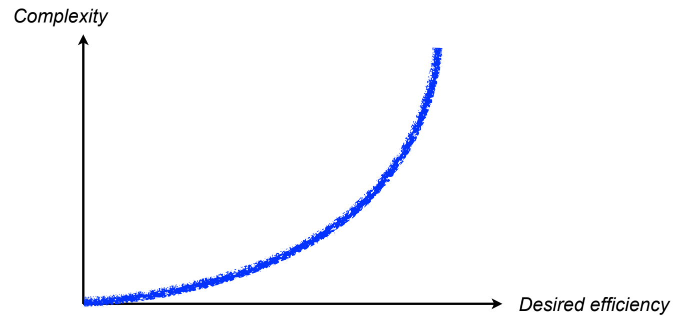
Many Requirements ⇒ Complexity



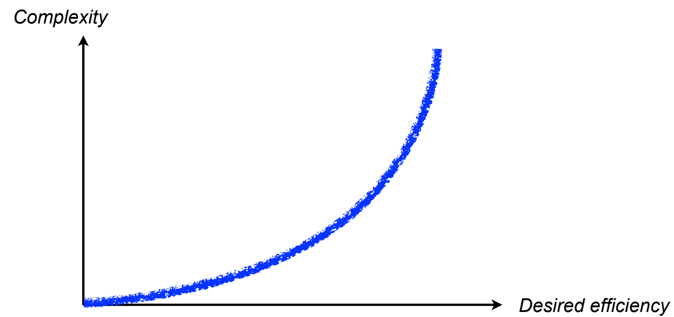
Quest for Efficiency ⇒ Complexity



Quest for Efficiency ⇒ Complexity

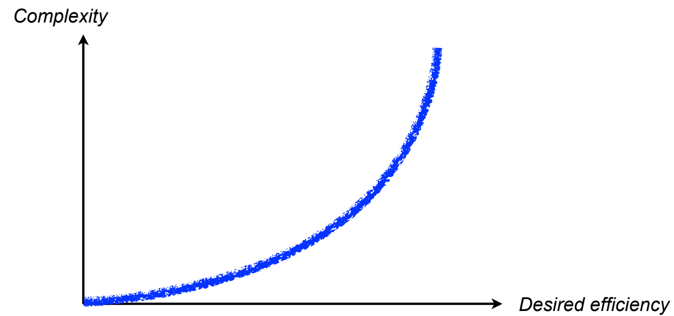


Quest for Efficiency ⇒ Complexity



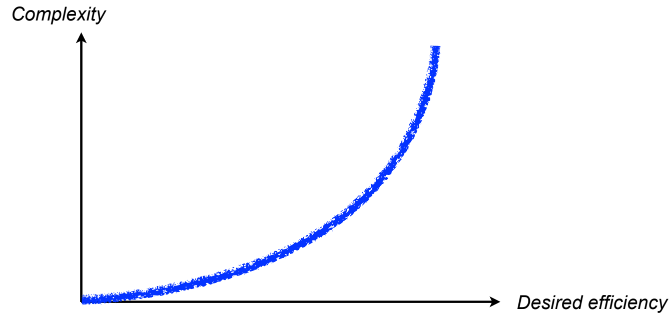
Buffer cache

Quest for Efficiency ⇒ Complexity



Writeback buffer cache

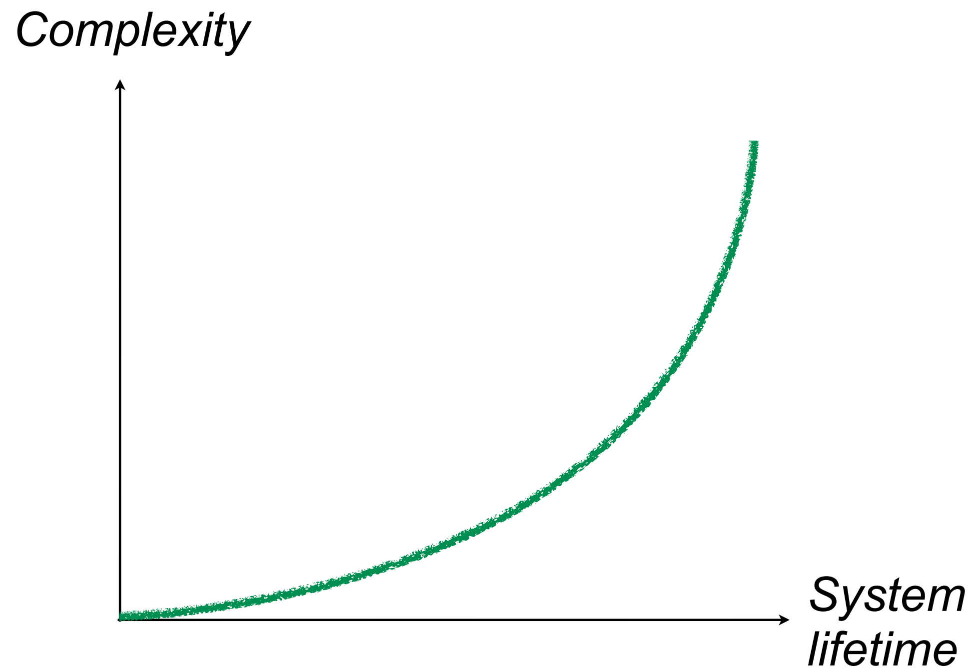
Quest for Efficiency ⇒ Complexity



fsync()

Writeback buffer cache

System Evolution ⇒ Complexity

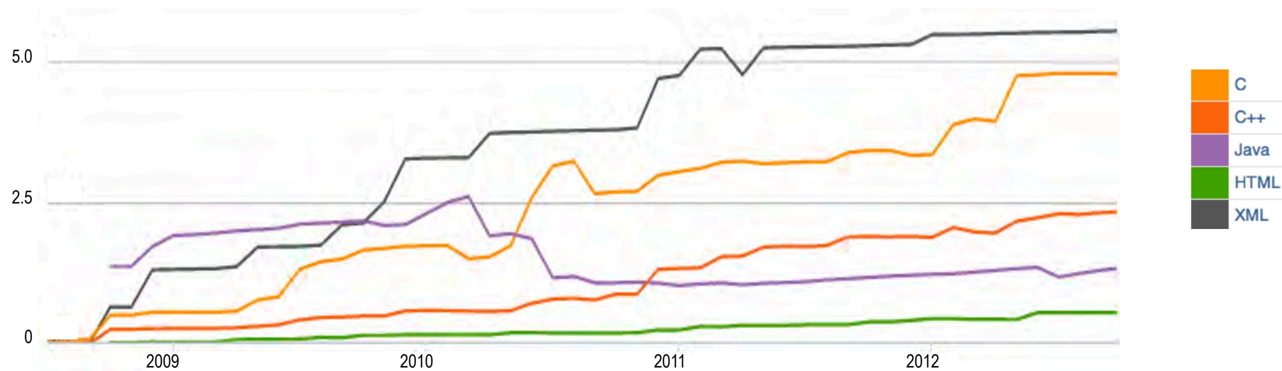


- Changes in requirements
  - *same system design, different environments*
  - *users want new things, but replacing the system is costly*
- Changes to address these = Evolution
  - *changes introduce complexity*

Legacy Systems = Complex Systems

Android codebase size

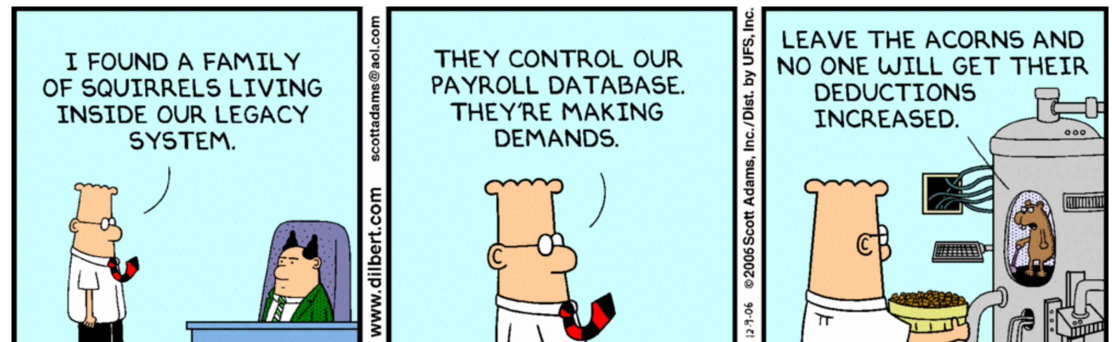
(in millions of lines, over its first 4 years)



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

Sources of Complexity

- Exponential path/state explosion
- Large number of requirements
- Quest for high efficiency
- System evolution



Symptoms of Complexity

Prof. George Candea
School of Computer & Communication Sciences

Building Bridges

A COMPUTER SCIENCE PERSPECTIVE OF BRIDGE DESIGN

What kinds of lessons does a classical engineering discipline like bridge design have for an emerging engineering discipline like computer systems design? Case-study editors Alfred Spector and David Gifford consider the insight and experience of bridge designer Gerard Fox to find out how strong the parallels are.

ALFRED SPECTOR and DAVID GIFFORD

AS **Gerry, let's begin with an overview of bridges.**
GF In the United States, most highway bridges are mandated by a government agency. The great majority are small bridges (with spans of less than 150 feet) and are part of the public highway system. There are fewer large bridges, having spans of 600 feet or more, that carry roads over bodies of water, gorges, or other large obstacles. There are also a small number of superlarge bridges with spans approaching a mile, like the Verrazzano Narrows Bridge in New York.

AS **What are the requirements for a bridge?**
GF There are several categories of requirements. For instance, there are *functionality* requirements: The lanes should be sufficiently wide, the bridge should have safe barriers to deflect cars back onto the roadway, and the lighting should be sufficient. There are *serviceability* requirements: We don't want the bridge to vibrate excessively and scare people, and we don't want large cracks in concrete bridges. Of course, there is the *ultimate strength* requirement: We don't want the bridge to fail. Then there is an *aesthetics* requirement: The bridge should be pleasing to the eye. There's also a *long-term maintainability* requirement, which involves corrosion protection of various elements. For example, cables tend to be very susceptible to stress corrosion, and therefore their protection is very important. Finally, there is the *cost-effectiveness* requirement: The finished product should meet all of the above requirements at the best possible cost.

© 1986 ACM 0001-0782/86/0400-0268 75¢

THE DESIGN PROCESS

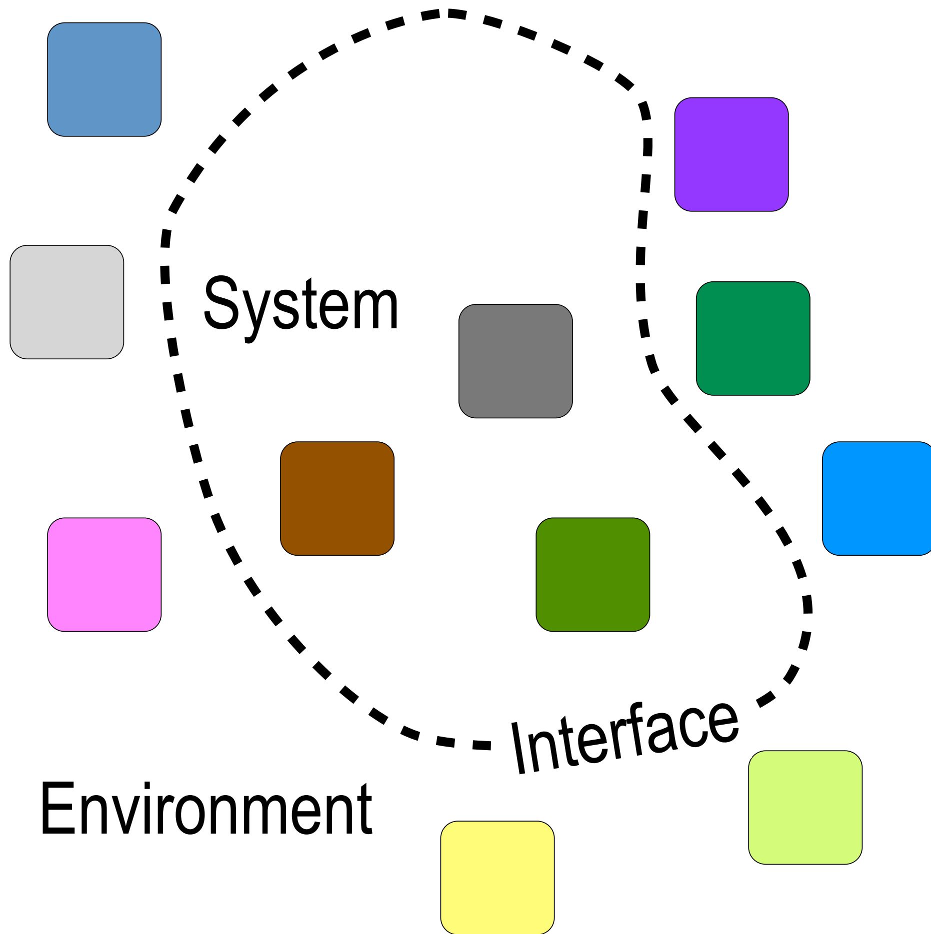
AS **What is the procedure for designing and constructing a bridge?**
GF It breaks down into three phases: the *preliminary design phase*, the *main design phase*, and the *construction phase*. For larger bridges, several alternative designs are usually considered during the preliminary design phase, whereas simple calculations or experience usually suffices in determining the appropriate design for small bridges. There are a lot more factors to take into account with a large bridge: aesthetics, method of construction, cost of materials, etc. The preliminary design report for a large bridge usually describes three or four alternative bridge types, estimates their costs, and provides a rendering of what the bridge will look like. Usually, the designer recommends one of the alternatives to the client. There would also usually be hearings to get the public's reaction.

DG **Do you estimate both the initial cost and the life-cycle cost for each of the alternatives?**
GF Life-cycle costing is not in wide use for bridges, although I think it should be. For example, consider the life-cycle cost of a bridge's deck, the portion of the bridge that comprises the riding surface. One alternative is to design an orthotropic steel deck, which can support traffic and also help to carry the weight of the bridge itself. The alternative is a concrete slab deck, which costs a lot less initially, but does not last nearly as long as a steel deck. Since the initial cost is the primary thing that clients look at today, most new bridges in this country are being built with concrete decks. At the same time, many

Communications of the ACM *April 1986 Volume 29 Number 4*

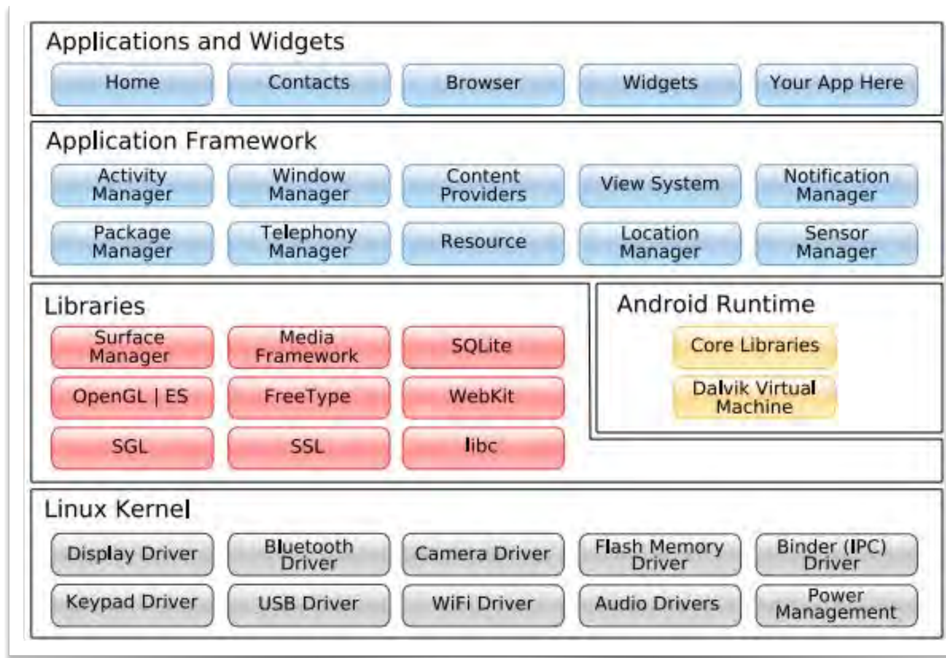
- Observation
  - *bridges are normally on-time, on-budget, and don't fall*
  - *software projects rarely ship on-time, are often over-budget, and rarely work exactly as specified*
- Blueprints for bridges must be approved...
  - *for structural integrity, earthquake and flood safety, etc.*
- Foundations are inspected
  - *you don't just build a pillar and test if it stands*
- Fundamental difference:
  - *laws of physics vs. laws of human intellect ...*

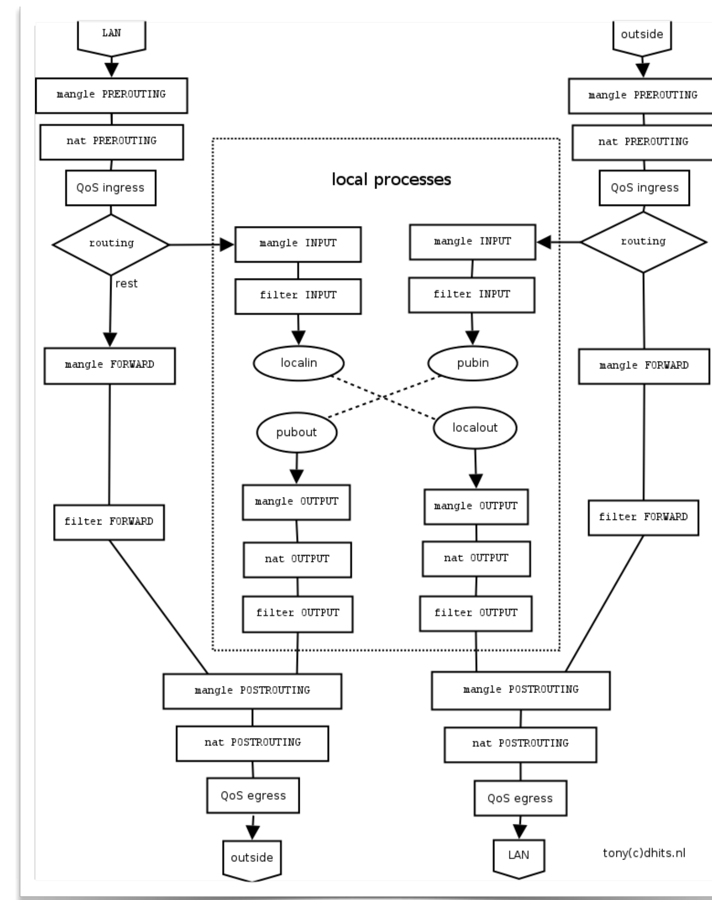
Four Symptoms of Complexity



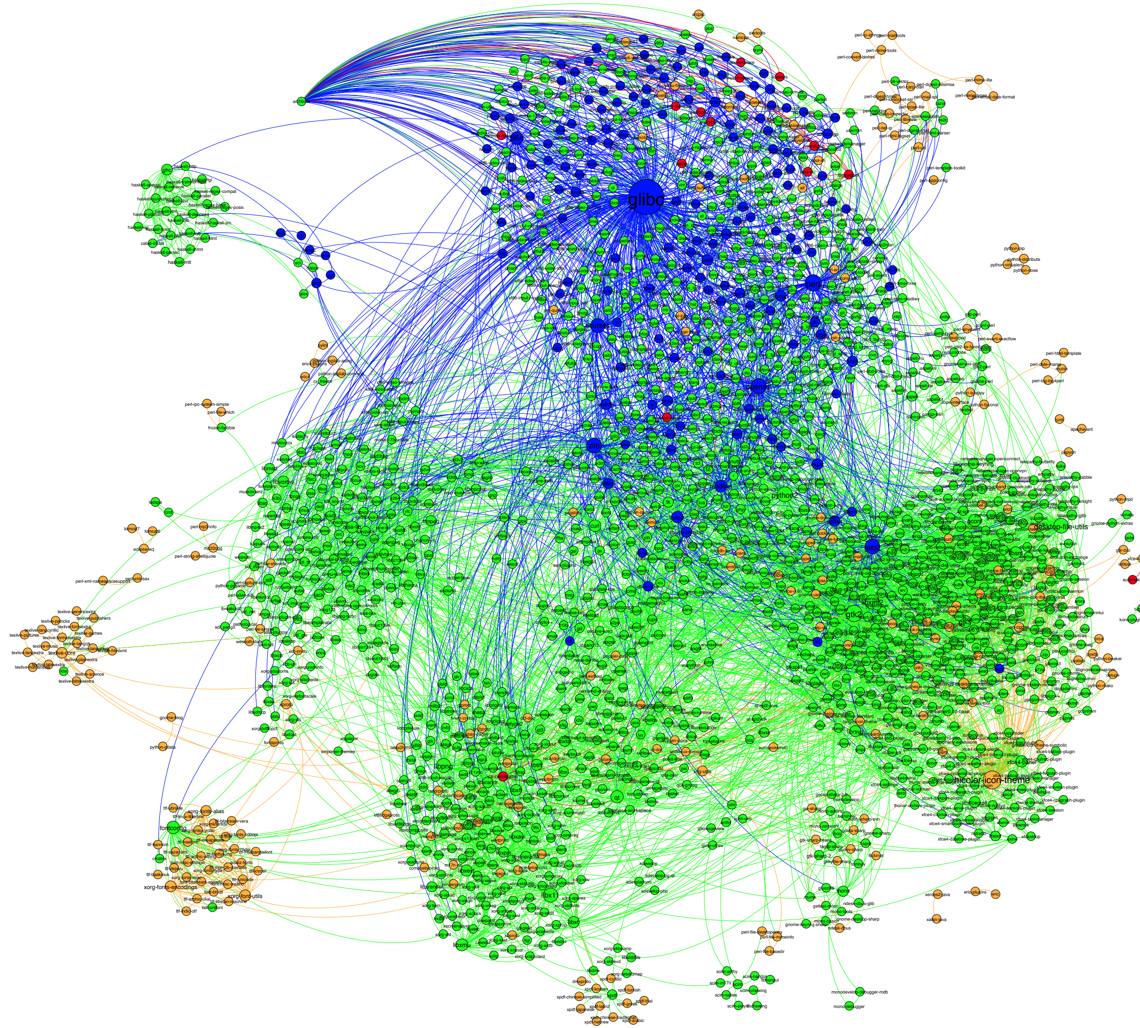
- Large number of components
- Large number of interconnections
- Many irregularities and exceptions
- High “Kolmogorov complexity”

Symptom #1: Many Components



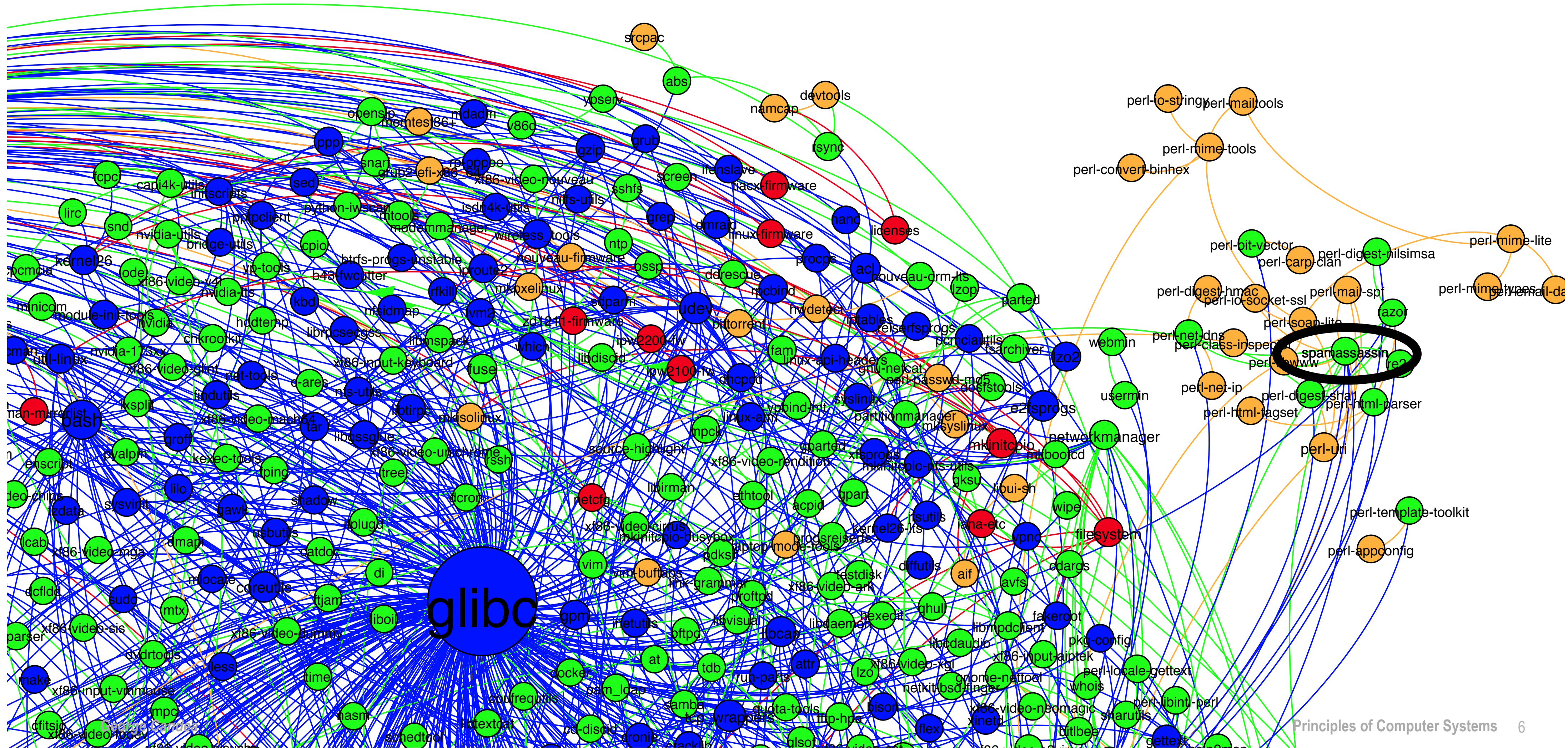


Symptom #2: Many Interconnections

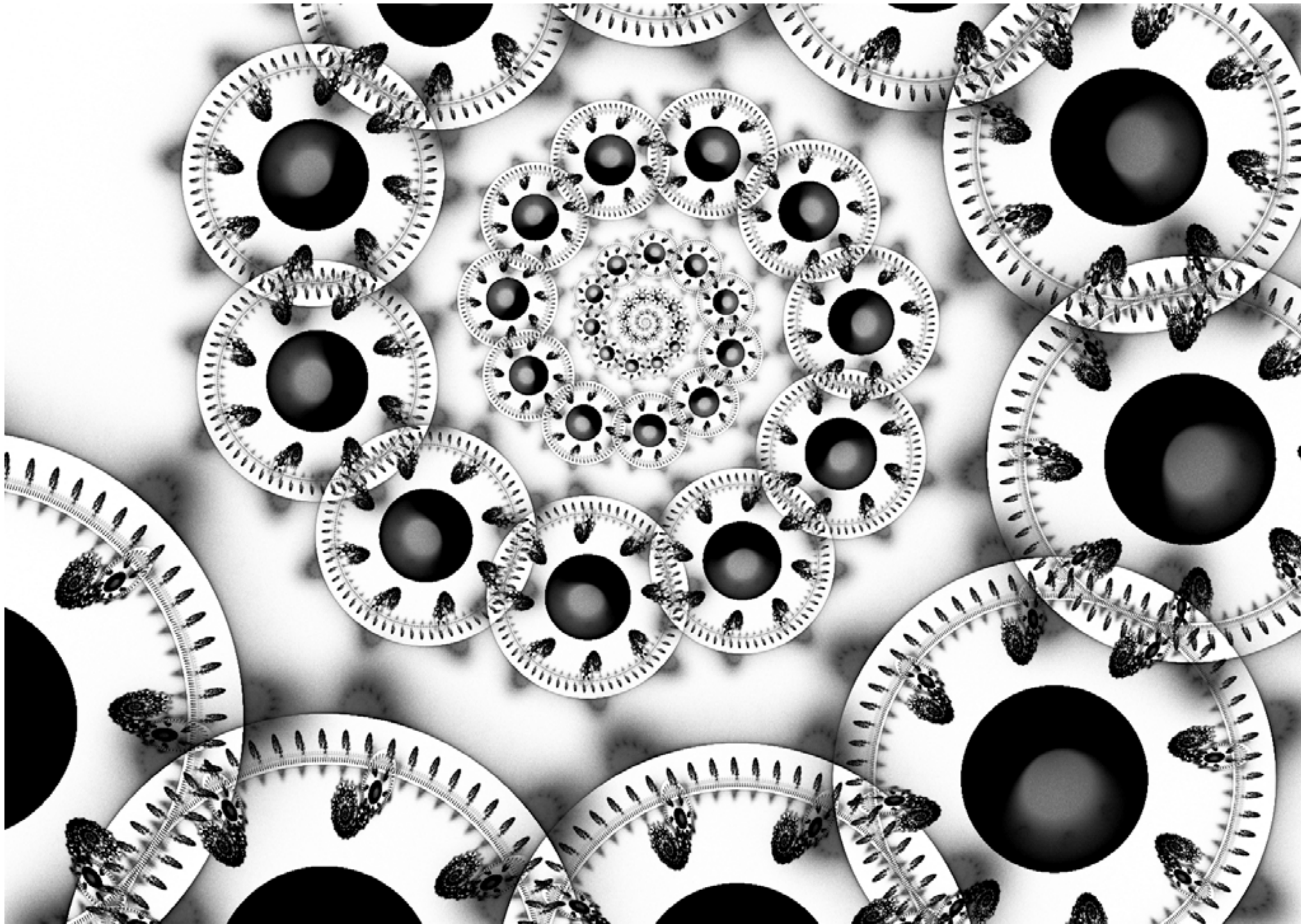


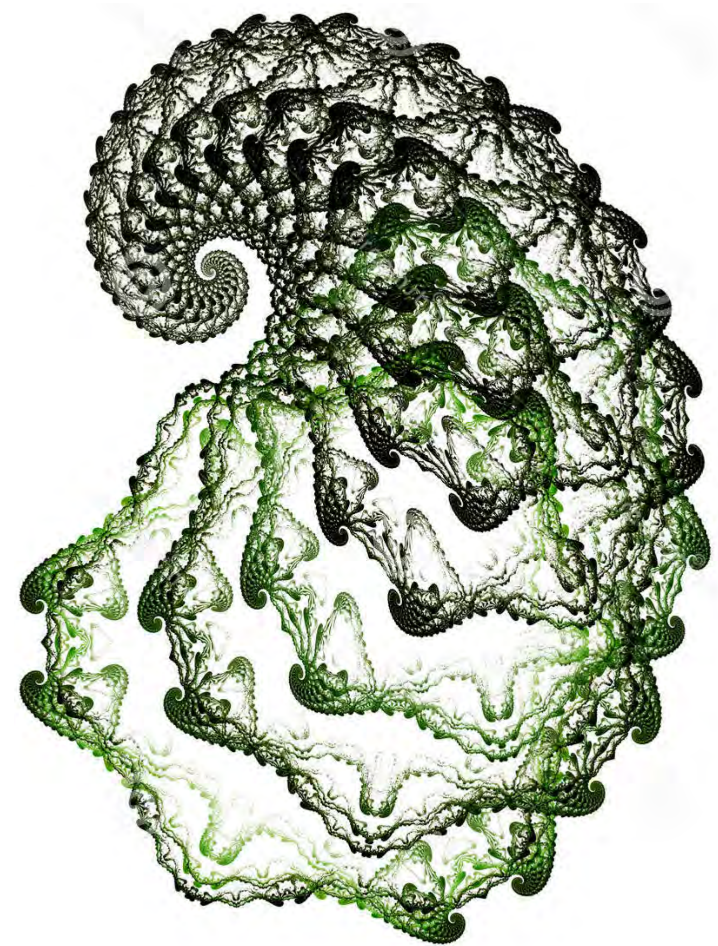
http://www.toofishes.net/blog/arch-package-visualization/

Symptom #2: Many Interconnections



Symptom #3: Irregularity and Exceptions





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

Symptom #4: High “Kolmogorov Complexity”

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

$\mathbf{K}$(*AAAAAAA ... AAAAB*) =
|“1 million *A*s followed by 1 *B*”|

⇒ simple

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

$\mathbf{K}$(*ABDAGHDBBCAD...*) = 10^6+1

⇒ complex

- Kolmogorov complexity
  - *computation resources needed to specify an object*
  - *minimal length of a description of the object*
- $\mathbf{K}$(object) >= |object| => complex
  $\mathbf{K}$(object) << |object| => simple

Symptoms of Complexity

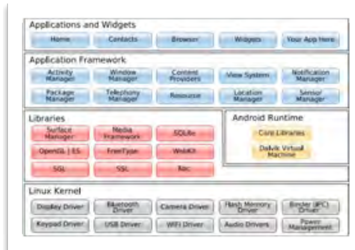
ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

- Four symptoms of complexity
  - *large number of components*
  - *large number of interconnections*
  - *many irregularities and exceptions*
  - *high "Kolmogorov complexity"*

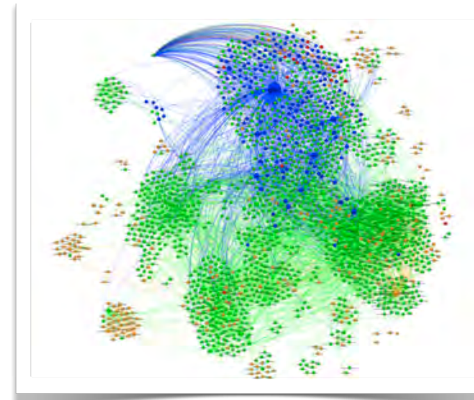
Modularity

Prof. George Candea
School of Computer & Communication Sciences

Symptoms of Complexity (Recap)



- Large number of components



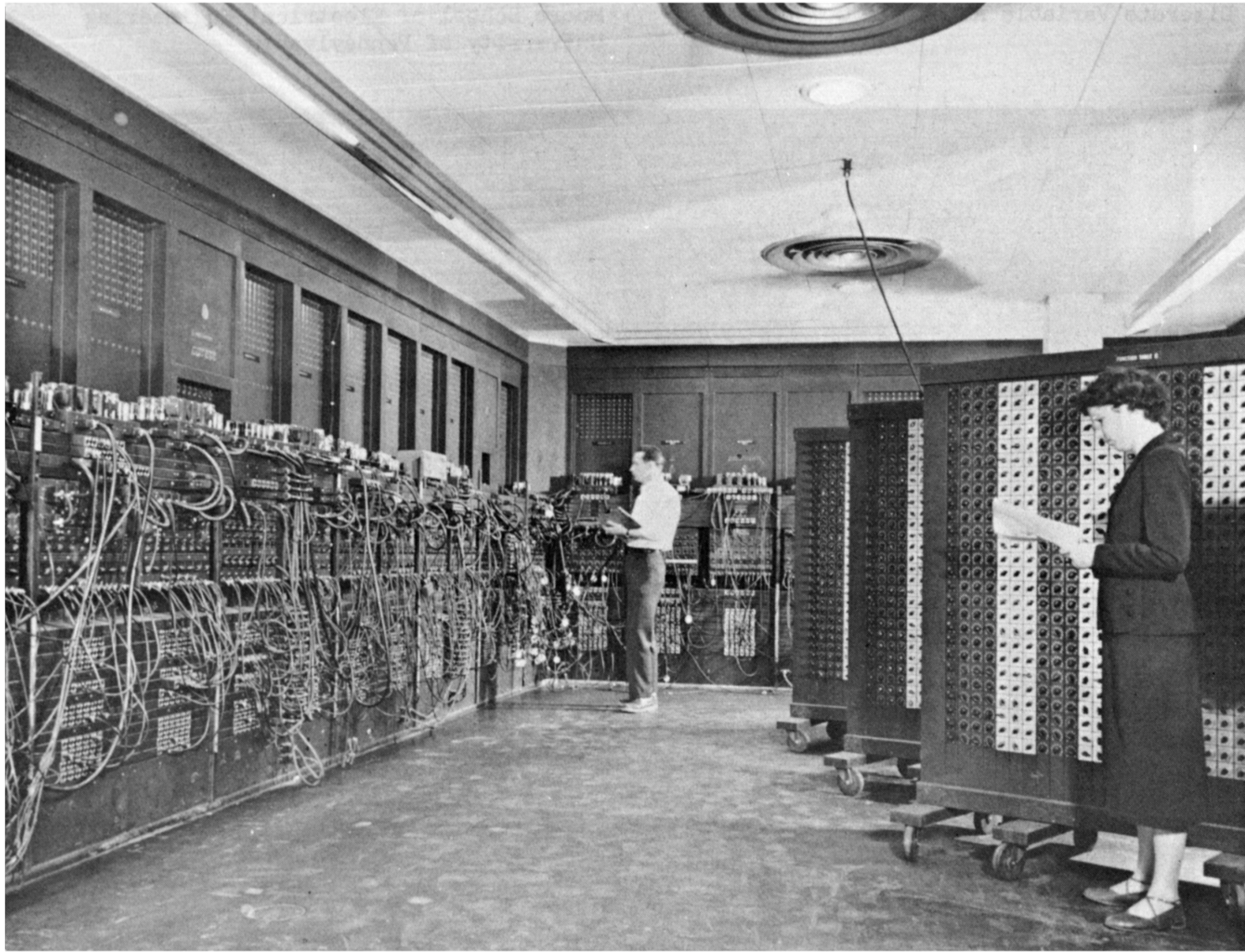
- Large number of interconnections
  - ⇒ *small disturbance can have chaotic consequences*

- Irregularities and exceptions

$\mathbf{K}$(object) >= |object| => complex

$\mathbf{K}$(object) << |object| => simple

- High Kolmogorov complexity
  - ⇒ *large # of brains required to understand the whole*



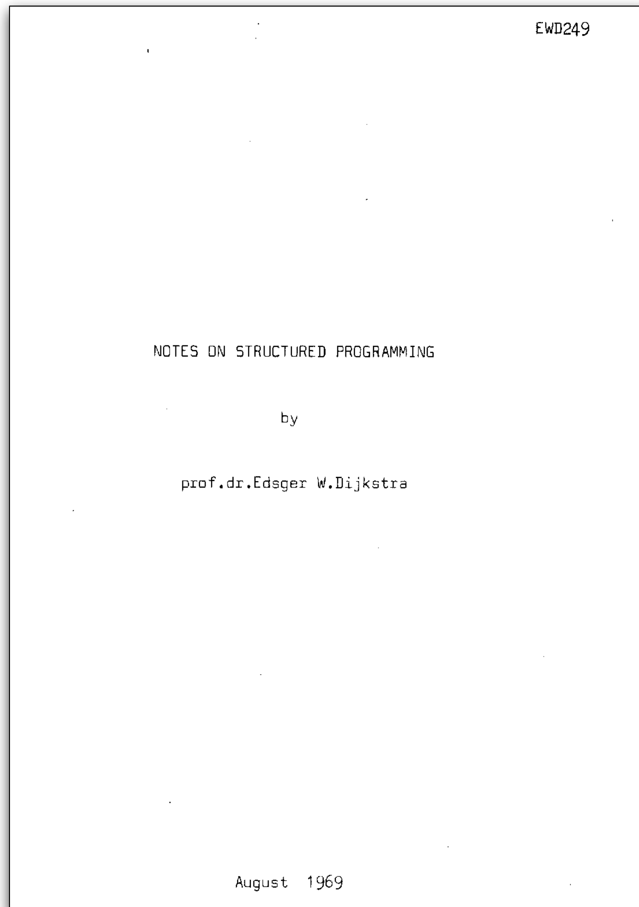
```
MAIN0001* PROGRAM TO SOLVE THE QUADRATIC EQUATION
MAIN0002        READ 10,A,B,C $
MAIN0003        DISC = B*B-4*A*C $
MAIN0004        IF (DISC) NEGA,ZERO,POSI $
MAIN0005   NEGA R = 0.0 - 0.5 * B/A $
MAIN0006        AI = 0.5 * SQRTF(0.0-DISC)/A $
MAIN0007        PRINT 11,R,AI $
MAIN0008        GO TO 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

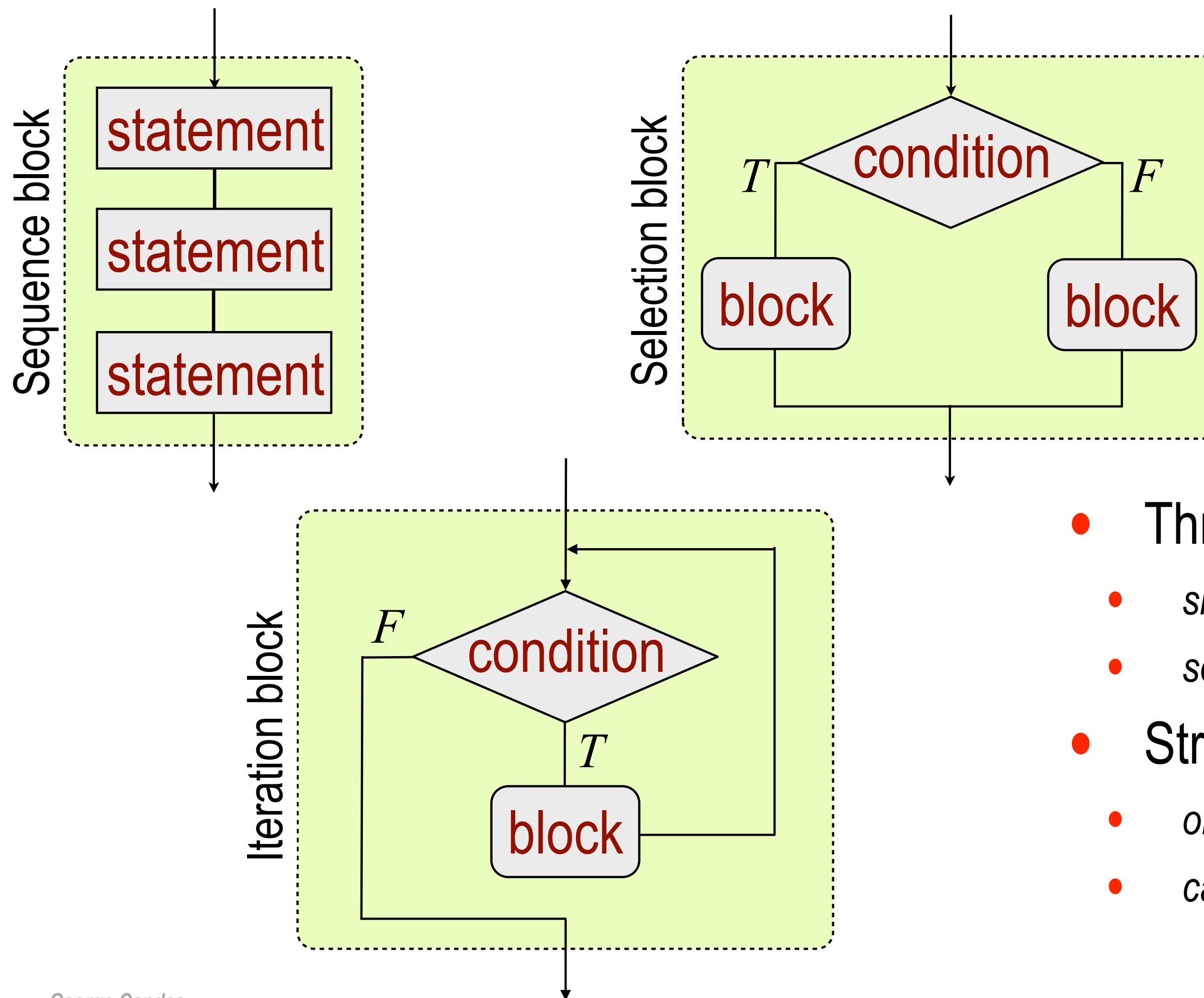
by

prof.dr.Edsger W.Dijkstra

August 1969

- 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*

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*

fs/debugfs/file.c

```
static int fops_u8_open(struct inode *inode, struct file *file)
{
	__simple_attr_check_format("%llu\n", 0ull);
	return simple_attr_open(inode, file, debugfs_u8_get, debugfs_u8_set, "%llu\n");
}
static const struct file_operations fops_u8 = {
	.owner   = THIS_MODULE,
	.open    = fops_u8_open,
	.release = simple_attr_release,
	.read    = simple_attr_read,
	.write   = simple_attr_write,
	.llseek  = generic_file_llseek,
};
static int fops_u8_ro_open(struct inode *inode, struct file *file)
{
	__simple_attr_check_format("%llu\n", 0ull);
	return simple_attr_open(inode, file, debugfs_u8_get, NULL, "%llu\n");
}
static const struct file_operations fops_u8_ro = {
	.owner   = THIS_MODULE,
	.open    = fops_u8_ro_open,
	.release = simple_attr_release,
	.read    = simple_attr_read,
	.write   = simple_attr_write,
	.llseek  = generic_file_llseek,
};
static int fops_u8_wo_open(struct inode *inode, struct file *file)
{
	__simple_attr_check_format("%llu\n", 0ull);
	return simple_attr_open(inode, file, NULL, debugfs_u8_set, "%llu\n");
}
static const struct file_operations fops_u8_wo = {
	.owner   = THIS_MODULE,
	.open    = fops_u8_wo_open,
	.release = simple_attr_release,
	.read    = simple_attr_read,
	.write   = simple_attr_write,
	.llseek  = generic_file_llseek,
};
```

fs/debugfs/file.c

```
static int fops_u8_open(struct inode *inode, struct file *file)
{
        __simple_attr_check_format("%llu\n", 0ull);
        return simple_attr_open(inode, file, debugfs_u8_get, debugfs_u8_set, "%llu\n");
}
static const struct file_operations fops_u8 = {
        .owner   = THIS_MODULE,
        .open    = fops_u8_open,
```

```
#define DEFINE_SIMPLE_ATTRIBUTE(__fops, __get, __set, __fmt)          \
static int __fops ## _open(struct inode *inode, struct file *file)    \
{                                                                     \
        __simple_attr_check_format(__fmt, 0ull);                      \
        return simple_attr_open(inode, file, __get, __set, __fmt);    \
}                                                                     \
static const struct file_operations __fops = {                        \
        .owner   = THIS_MODULE,                                       \
        .open    = __fops ## _open,                                   \
        .release = simple_attr_release,                               \
        .read    = simple_attr_read,                                  \
        .write   = simple_attr_write,                                 \
        .llseek  = generic_file_llseek,                               \
};
```

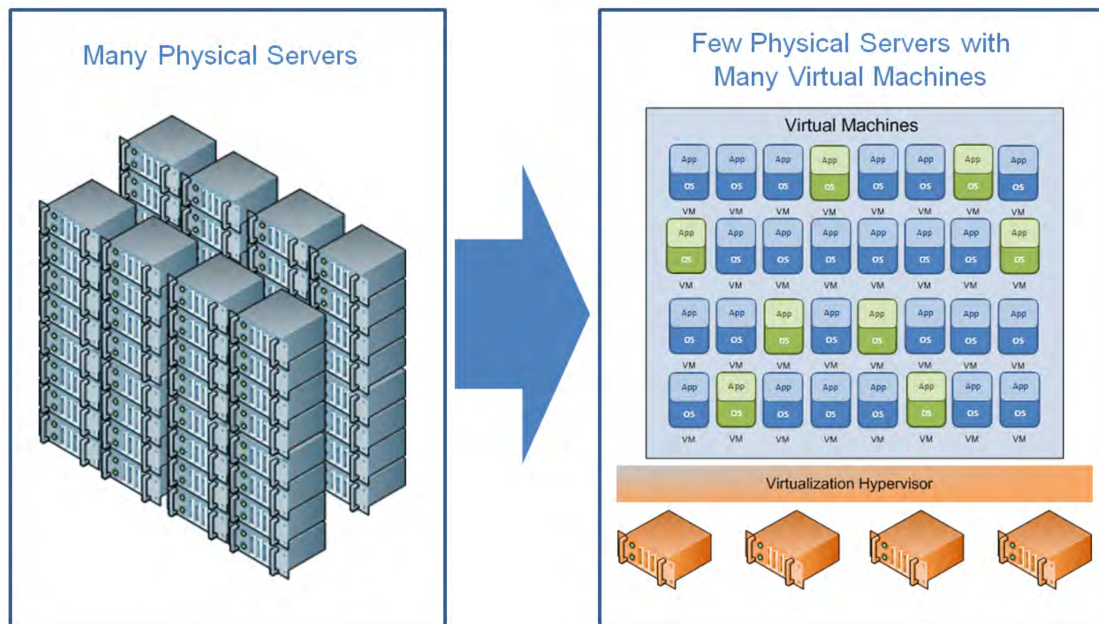
```
static const struct file_operations fops_u8_wo = {
        .owner   = THIS_MODULE,
        .open    = fops_u8_wo_open,
        .release = simple_attr_release,
        .read    = simple_attr_read,
        .write   = simple_attr_write,
        .llseek  = generic_file_llseek,
};
```

```
static int fops_u8_open(struct inode *inode, struct file *file)
{
	__simple_attr_check_format("%llu\n", 0ull);
	return simple_attr_open(inode, file, debugfs_u8_get, debugfs_u8_set, "%llu\n");
}
static const struct file_operations fops_u8 = {
	.owner   = THIS_MODULE,
	.open    = fops_u8_open,
	.release = simple_attr_release,
	.read    = simple_attr_read,
	.write   = simple_attr_write,
	.llseek  = generic_file_llseek,
};
static int fops_u8_ro_open(struct inode *inode, struct file *file)
{
	__simple_attr_check_format("%llu\n", 0ull);
	return simple_attr_open(inode, file, debugfs_u8_get, NULL, "%llu\n");
}
static const struct file_operations fops_u8_ro = {
	.owner   = THIS_MODULE,
	.open    = fops_u8_ro_open,
	.release = simple_attr_release,
	.read    = simple_attr_read,
	.write   = simple_attr_write,
	.llseek  = generic_file_llseek,
};
static int fops_u8_wo_open(struct inode *inode, struct file *file)
{
	__simple_attr_check_format("%llu\n", 0ull);
	return simple_attr_open(inode, file, NULL, debugfs_u8_set, "%llu\n");
}
static const struct file_operations fops_u8_wo = {
	.owner   = THIS_MODULE,
	.open    = fops_u8_wo_open,
	.release = simple_attr_release,
	.read    = simple_attr_read,
	.write   = simple_attr_write,
	.llseek  = generic_file_llseek,
};
```

```
#define DEFINE_SIMPLE_ATTRIBUTE(__fops, __get, __set, __fmt)                \
static int __fops ## _open(struct inode *inode, struct file *file)     \
{                                                                       \
	__simple_attr_check_format(__fmt, 0ull);                            \
	return simple_attr_open(inode, file, __get, __set, __fmt);          \
}                                                                       \
static const struct file_operations __fops = {                          \
	.owner   = THIS_MODULE,                                             \
	.open    = __fops ## _open,                                         \
	.release = simple_attr_release,                                     \
	.read    = simple_attr_read,                                        \
	.write   = simple_attr_write,                                       \
	.llseek  = generic_file_llseek,                                     \
};
```

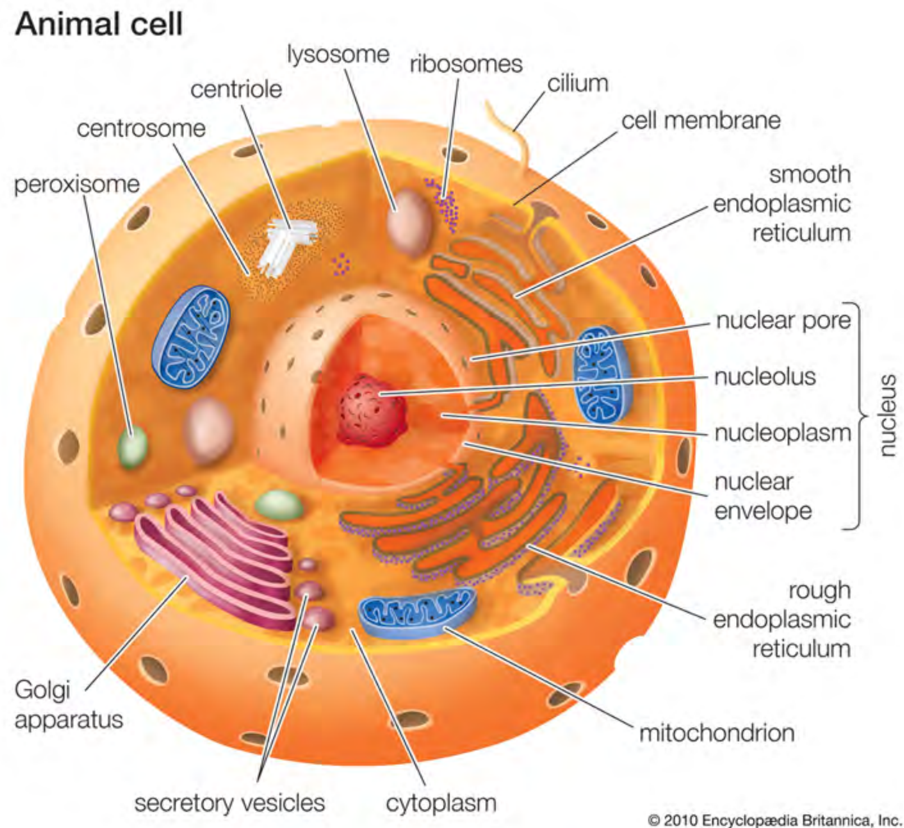
```
DEFINE_SIMPLE_ATTRIBUTE(fops_u8, debugfs_u8_get, debugfs_u8_set, "%llu\n");
DEFINE_SIMPLE_ATTRIBUTE(fops_u8_ro, debugfs_u8_get, NULL, "%llu\n");
DEFINE_SIMPLE_ATTRIBUTE(fops_u8_wo, NULL, debugfs_u8_set, "%llu\n");
```

Modularity Through Virtualization



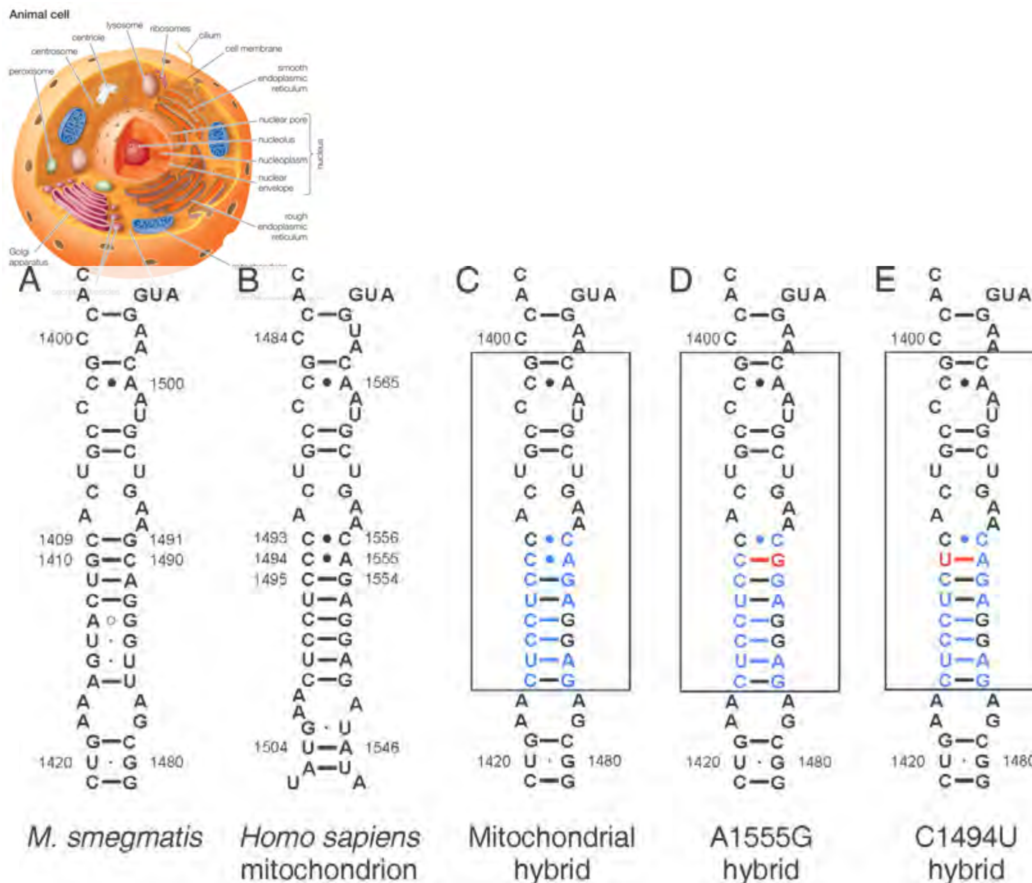
- Server consolidation
  - *Reduce operating costs*
  - *Reduce management costs*
- Cloud computing
  - *On-demand VM provisioning*
  - *VM migration*
  - *VM replication*

Modularity Outside Computer Science



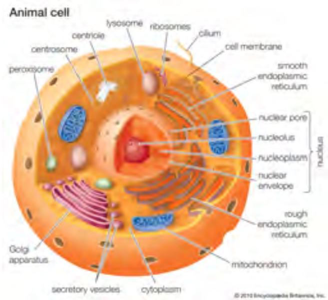
- Cell = module used to build organisms
- Gene = unit (module) of evolution
- Cognitive activity (may be) modularized
- Division of labor = modularization
- Spare parts = modules
- IKEA ... modular furniture

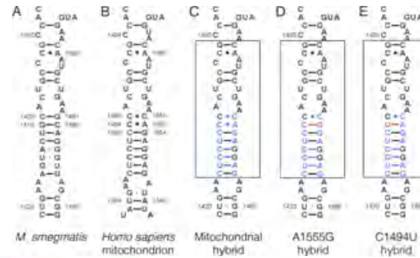
Modularity Outside Computer Science

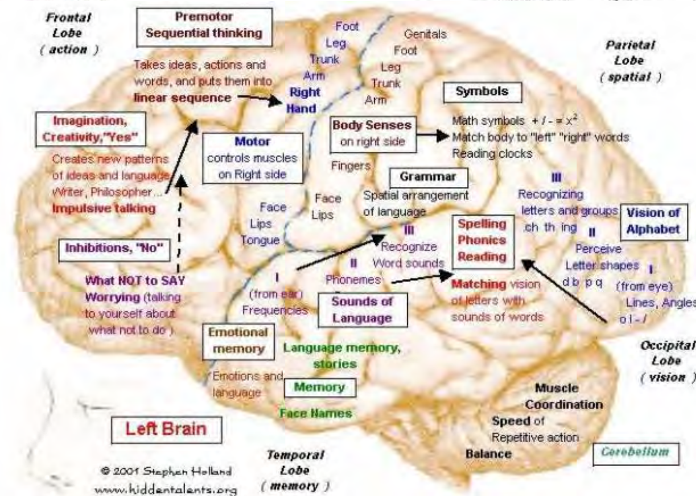


- Cell = module used to build organisms
- Gene = unit (module) of evolution
- Cognitive activity (may be) modularized
- Division of labor = modularization
- Spare parts = modules
- IKEA ... modular furniture

Modularity Outside Computer Science

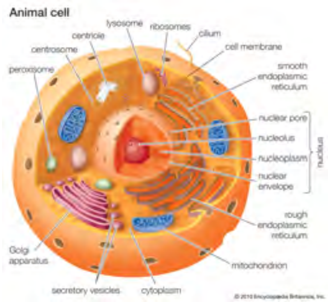


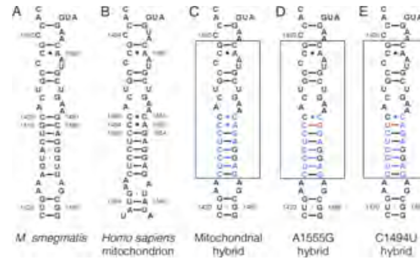


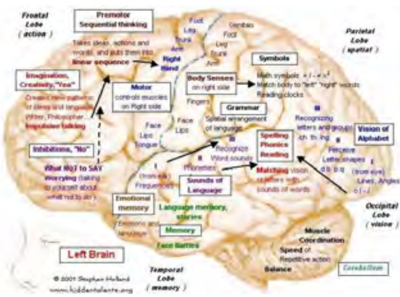


- Cell = module used to build organisms
- Gene = unit (module) of evolution
- Cognitive activity (may be) modularized
- Division of labor = modularization
- Spare parts = modules
- IKEA ... modular furniture

Modularity Outside Computer Science

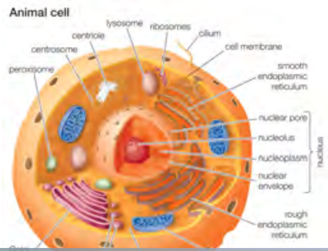


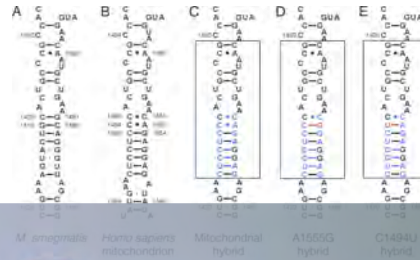




- Cell = module used to build organisms
- Gene = unit (module) of evolution
- Cognitive activity (may be) modularized
- Division of labor = modularization
- Spare parts = modules
- IKEA ... modular furniture

Modularity Outside Computer Science





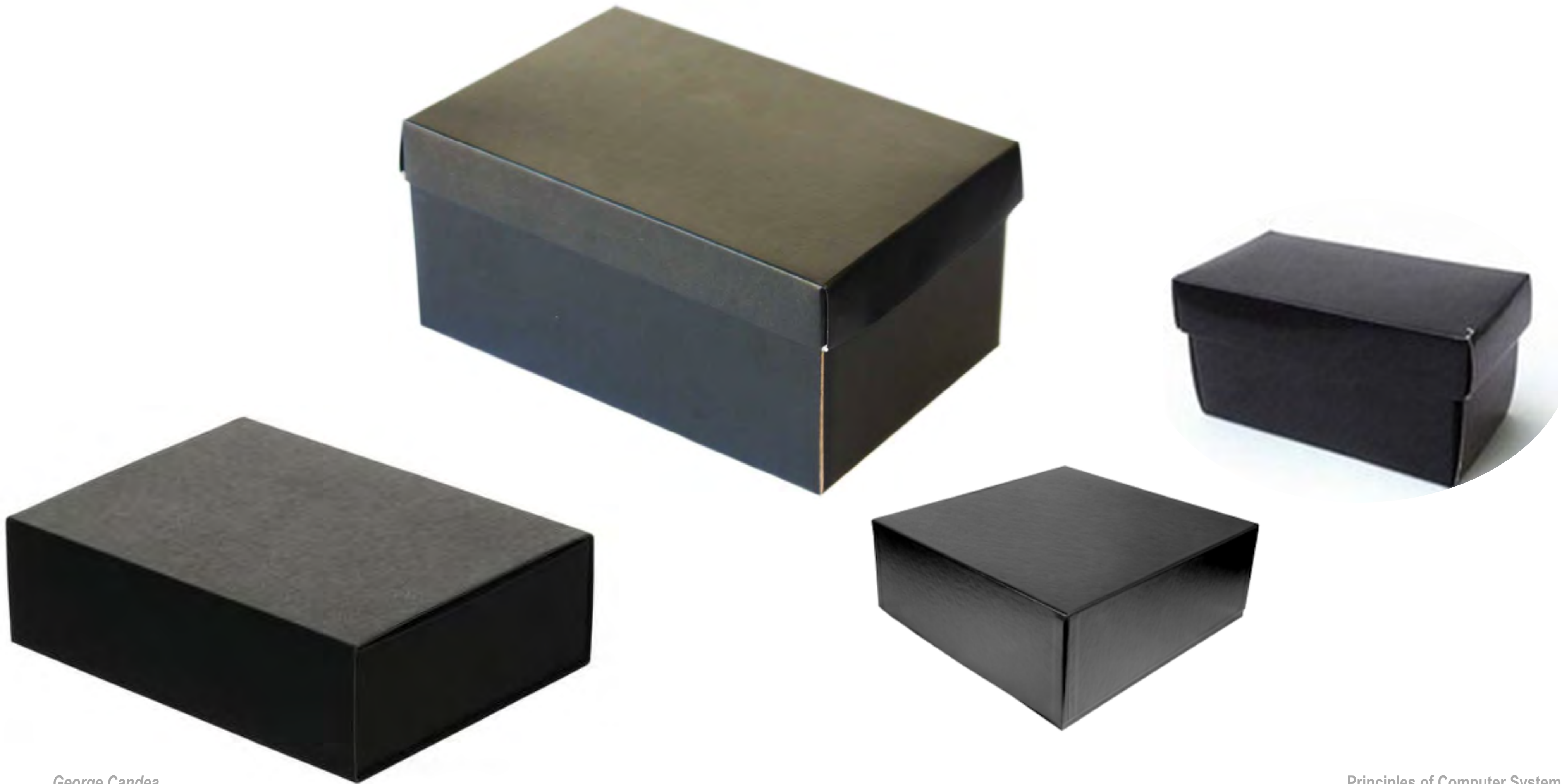
- Cell = module used to build organisms
- Gene = unit (module) of evolution
- Cognitive activity (may be) modularized
- Division of labor = modularization
- Spare parts = modules
- IKEA ... modular furniture

Abstraction

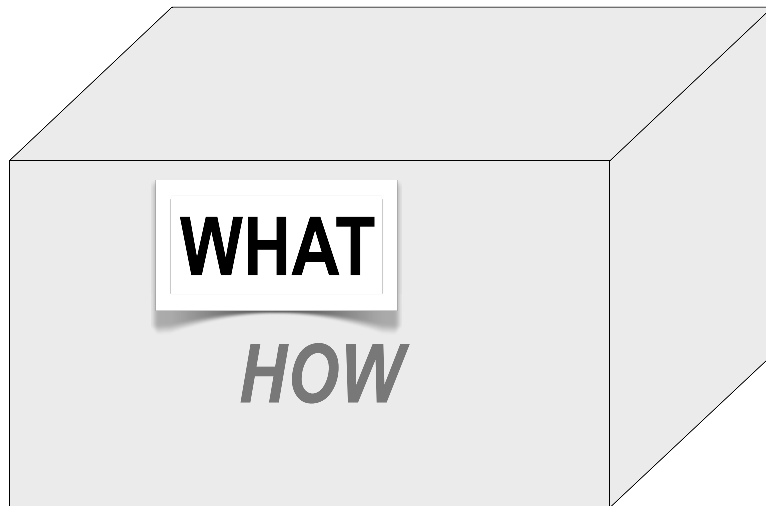
Prof. George Candea
School of Computer & Communication Sciences

Modularity (Recap)

Modularity (Recap)

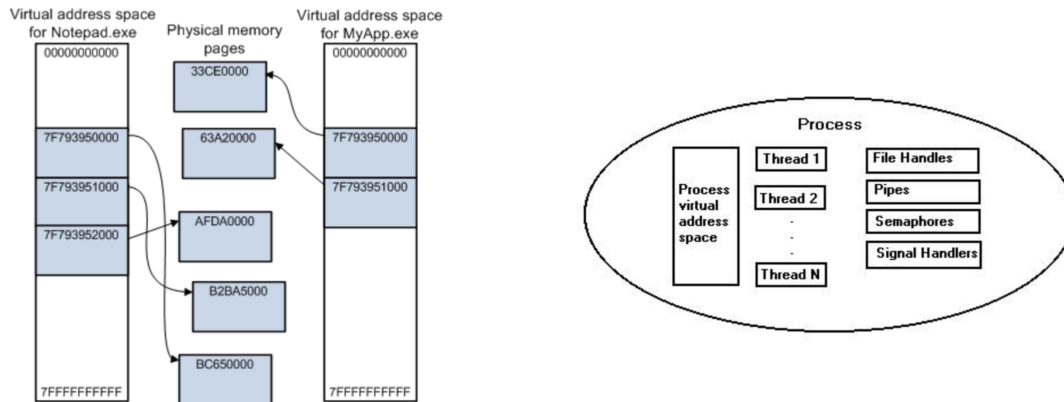


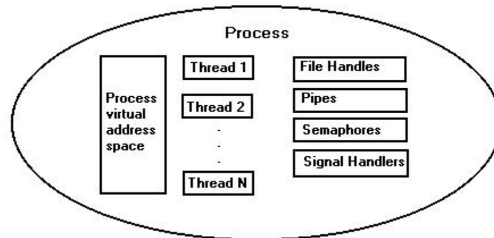
Abstraction

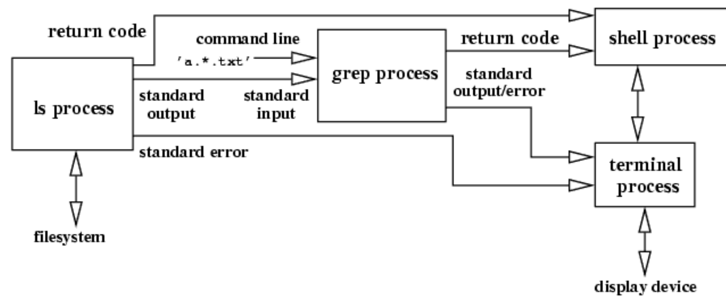


- Abstraction
  - *specifies "what" a component/subsystem does*
  - *together with modularity, it separates "what" from "how"*

Examples of Abstractions in Operating Systems







- Virtual address space
- Process
- Pipe
- Filesystem

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
- Duck typing

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
- Duck typing

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
```

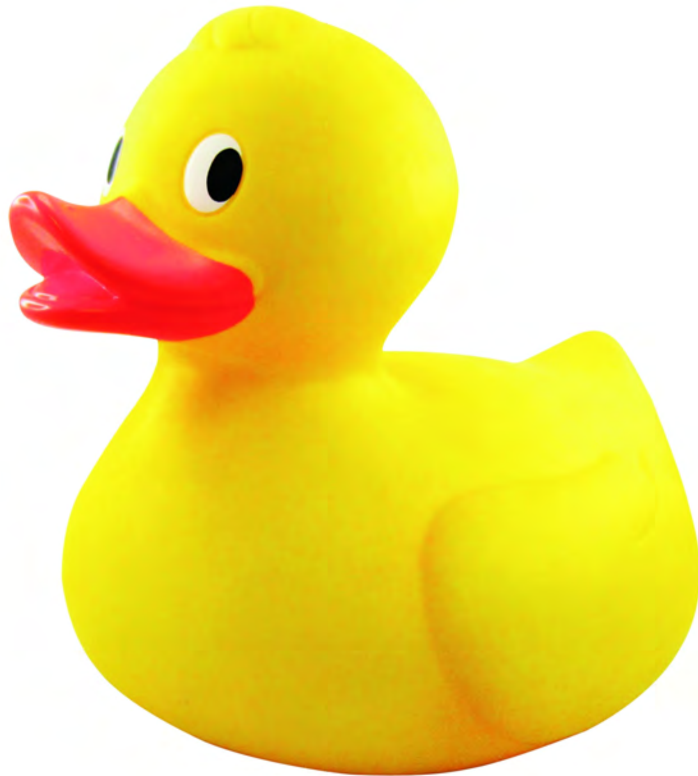
- Routines
  - *function, procedure, thread, etc.*
- Lambda functions
  - *a.k.a. anonymous functions*
- Abstract data types
- Objects
- Duck typing

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
- Duck typing

Examples of Abstractions in Programming Languages



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