

# In Search of Bug-free Software

**Yechiel M. Kimchi**

The Technion, CS Faculty

**VLSI – Verification, Logic Synthesis, Israel Ltd.**

August Penguin – Bug Free SW. 2017 © by Yechiel M. Kimchi

1

- Copyright © 2010–2017 Yechiel M. Kimchi

This presentation is an ongoing work that was first presented in writing on 2010[4]. Most of the detailed ideas that appear here were developed by many people and have appeared years ago. However, a few detailed ideas and the compilation of all of them into a coherent structure – especially the abridged list of coding rules and their rationale by the meta-rules – are original. The presentation in general, and in particular the original parts, are copyrighted under the terms of the GFDL v.1.3 as in <https://www.gnu.org/licenses/fdl-1.3.en.html> or later. Permission is granted to copy, distribute and/or modify this document under the terms of the GFDL with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.

August Penguin – Bug Free SW. 2017 © by Yechiel M. Kimchi

2

## The First Writing



Log in / create account

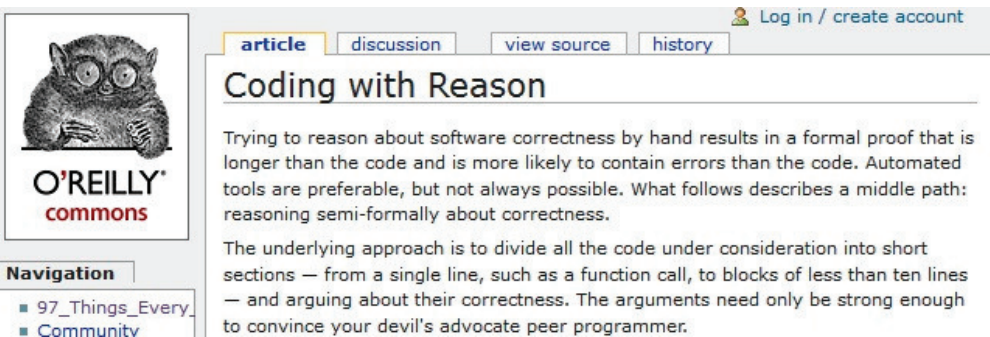
article discussion view source history

### 97 Things Every Programmer Should Know

Welcome to the home page for the *97 Things Every Programmer Should Know* project, pearls of wisdom for programmers collected from leading practitioners. You can read through the Contributions Appearing in the Book plus the Other Edited Contributions, browse Contributions in Progress, view the list of Contributors, and also learn How to Become a Contributor. If you would simply like to comment on a contribution, please also read How to Become a Contributor as some of it applies to you.

Navigation

[http://programmer.97things.oreilly.com/wiki/index.php/Coding\\_with\\_Reason](http://programmer.97things.oreilly.com/wiki/index.php/Coding_with_Reason)



Log in / create account

article discussion view source history

### Coding with Reason

Trying to reason about software correctness by hand results in a formal proof that is longer than the code and is more likely to contain errors than the code. Automated tools are preferable, but not always possible. What follows describes a middle path: reasoning semi-formally about correctness.

The underlying approach is to divide all the code under consideration into short sections — from a single line, such as a function call, to blocks of less than ten lines — and arguing about their correctness. The arguments need only be strong enough to convince your devil's advocate peer programmer.

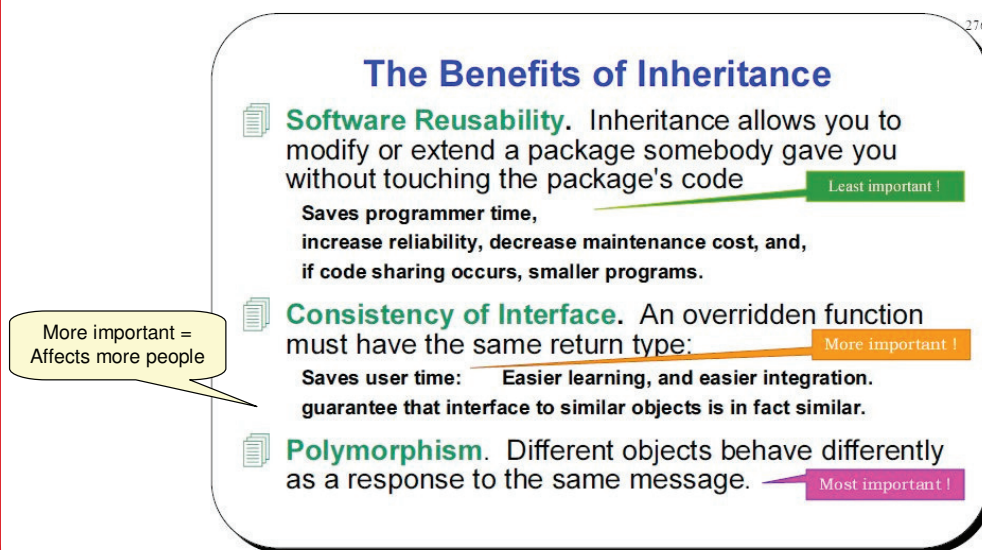
Navigation

- 97\_Things\_Every\_
- Community

August Penguin – Bug Free SW. 2017 © by Yechiel M. Kimchi

3

## But Earlier (the motivation)



276

### The Benefits of Inheritance

- Software Reusability.** Inheritance allows you to modify or extend a package somebody gave you without touching the package's code. Least important !
  - Saves programmer time,
  - increase reliability, decrease maintenance cost, and,
  - if code sharing occurs, smaller programs.
- Consistency of Interface.** An overridden function must have the same return type: More important !
  - Saves user time: Easier learning, and easier integration.
  - guarantee that interface to similar objects is in fact similar.
- Polymorphism.** Different objects behave differently as a response to the same message. Most important !

More important = Affects more people

Copyright 1995 - 2002

August Penguin – Bug Free SW. 2017 © by Yechiel M. Kimchi

4

## Before

- The program that worked only on Wednesdays
- Accidents do not happen – they are caused
- Reckless Parking
- Clog intersections
- The Prisoner's Dilemma
- Naivety (for changes by individuals)

**Writing quality code**  
is just a case of  
**Practicing Good Manners**

## My main interest is SW quality

- **From the theoretical point of view**
  - The abstract principles that guide the quality
- **The (counter-)examples will be practical**
  - Coding, but also Psychological, sociological, legal

**My claim: practice fails when it lacks theory**

## Why Software is So Bad? (cont.)

An Interview w. B. Stroustrup (2006) [5]

Q. “Why is most software so bad? ...”

BS: “... if software had been as bad as its reputation,  
most of us would have been dead by now.”

Q. “How can we fix the mess we are in?”

BS: [a full page] “In theory, ...: educate our software developers  
better, ... Reward correct, solid, and safe systems.  
Punish sloppiness. In reality, that’s essentially impossible.  
People want new fancy gadgets right now and reward  
people who deliver them cheaply, buggy, and first. ...”

August Penguin – Bug Free SW. 2017 © by Yechiel M. Kimchi

7

## General Purpose SW is Buggy

**What about safety-critical systems?**  
I’ll concentrate on them only

**I care about the SW tool itself**  
**The code, including design**

I care about the process only as long as it directly affects the code itself

Concentrating on

**Correct**, **Robust**, and ~~**Efficient**~~

August Penguin – Bug Free SW. 2017 © by Yechiel M. Kimchi

8

## How to Review the Coding Process?

### I am reluctant to read M-LOC

So I have focused my attention on well known  
**Coding Standard documents**

#### **Coding standards** [from Wikipedia: **Coding conventions**]

Where coding conventions have been specifically designed to produce high-quality code, and have then been formally adopted, they then become coding standards. Specific styles, irrespective of whether they are commonly adopted, do not automatically produce good quality code. It is only if they are designed to produce good quality code that they actually result in good quality code being produced, i.e., they must be very logical in every aspect of their design - every aspect justified and resulting in quality code being produced.

August Penguin – Bug Free SW. 2017 © by Yechiel M. Kimchi

9

## How to Review the Coding Process?

### I have reviewed

- MISRA-C (Motor Industry Software Reliability Association)
- JSF AV C++ Coding Standards (F-35)  
(Joint Strike Fighter Air Vehicle)
- Linux kernel coding style
- Google C++ Style Guide
- GNU Coding Standards

August Penguin – Bug Free SW. 2017 © by Yechiel M. Kimchi

10

## How to Review the Coding Process?

### I have reviewed

- MISRA-C (Motor Industry Software Reliability Association)
- JSF AV C++ Coding Standards (F-35)  
(Joint Strike Fighter Air Vehicle)
- Linux kernel coding style
- Google C++ Style Guide
- GNU Coding Standards

Let's go for the ideal

The Desired Code

What should those be compared with  
in order to find what they miss?

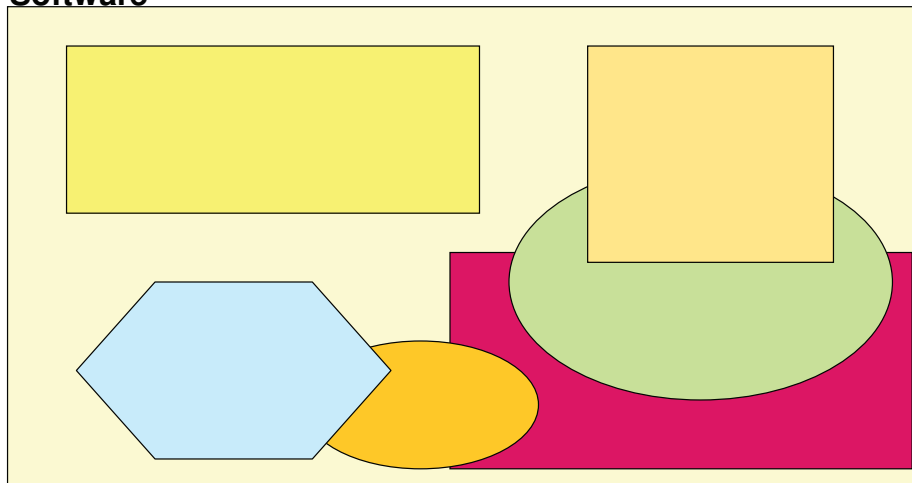
August Penguin – Bug Free SW. 2017 © by Yechiel M. Kimchi

11

## An External View of Software

### The Structure of Software

Software



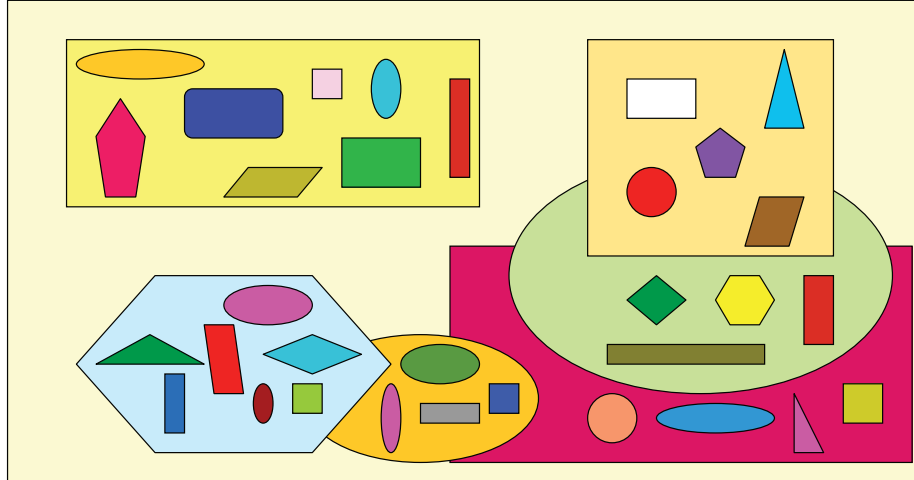
August Penguin – Bug Free SW. 2017 © by Yechiel M. Kimchi

12

## An External View (cont.)

### The Structure of Software

Software

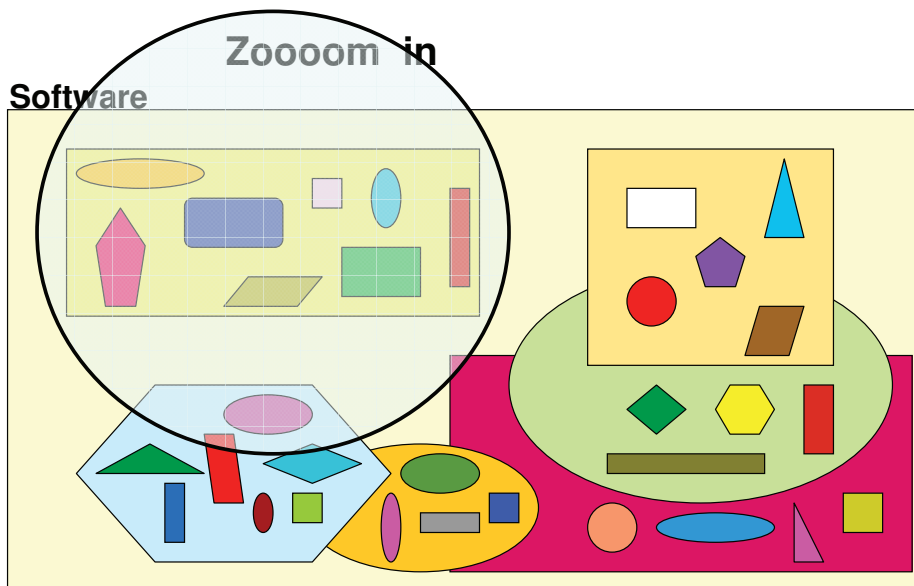


August Penguin – Bug Free SW. 2017 © by Yechiel M. Kimchi

13

## An External View (cont.)

Software

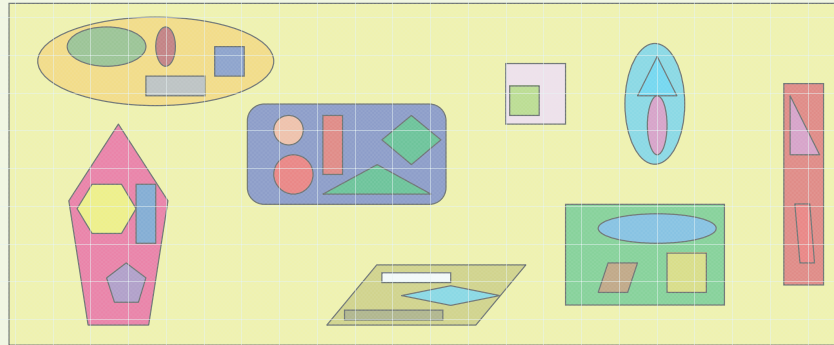


August Penguin – Bug Free SW. 2017 © by Yechiel M. Kimchi

14

## What is Software? (cont.)

Zooooom in  
Software

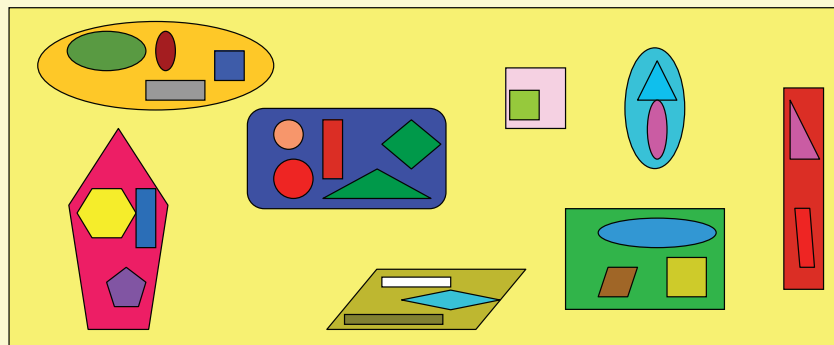


August Penguin – Bug Free SW. 2017 © by Yechiel M. Kimchi

15

## What is Software? (cont.)

Software Is fractal



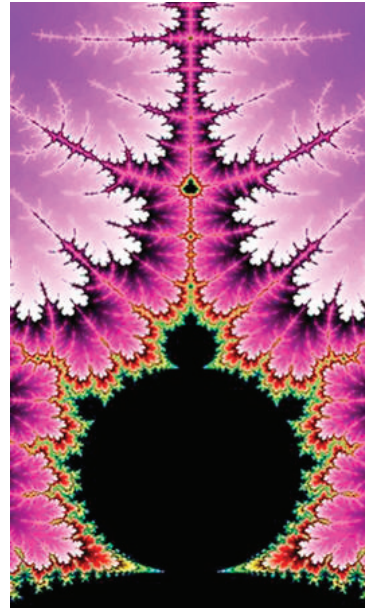
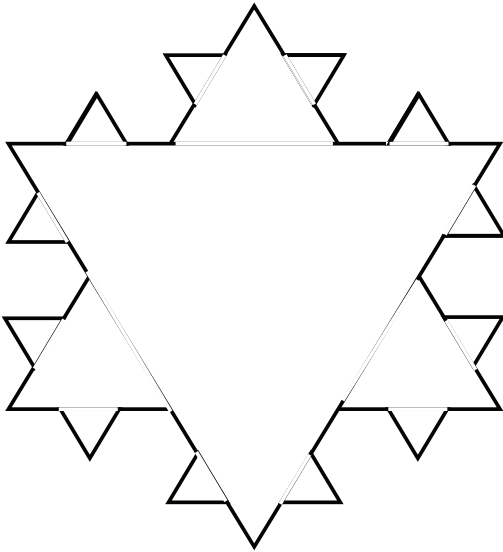
Finite – but unboundedly deep

August Penguin – Bug Free SW. 2017 © by Yechiel M. Kimchi

16



## Fractals (explanation)



August Penguin – Bug Free SW. 2017 © by Yechiel M. Kimchi

17

## Expectations from Quality Software

**As long as programmers write code:**

*They know the algorithms, but they err*

*So they have to test and they have to modify*

**Human cognition has limited capacity**

*Same thing with computer's memory*

*Software state-space is too big for both*

*HW too*

August Penguin – Bug Free SW. 2017 © by Yechiel M. Kimchi

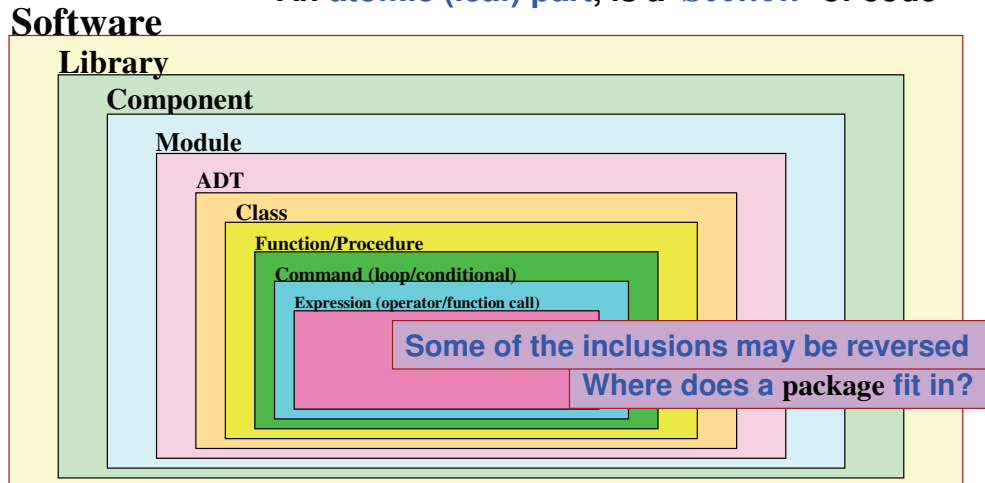
18

# Nomenclature in Quality Software

## Software is Fractal like

What are the recurring *Parts* ?

An **atomic (leaf)** part, is a *Section* of code



August Penguin – Bug Free SW. 2017 © by Yechiel M. Kimchi

19

# Expectations from Quality Software

## Meta-Meta-Rule:

*Software is a collection of parts  
that are governed by common requirements*

What is Common to these *Parts*?

Definition

*Each one of them gives an **abstract** service*

**But this is just the beginning**

August Penguin – Bug Free SW. 2017 © by Yechiel M. Kimchi

20

## Common Knowledge

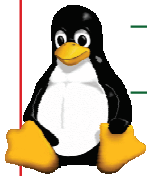
**In theory,**

Every rule has an exception  
– including this one

**In practice, they are not**

## A Spoiler

- I want to explain my intentions



- I'll use examples, related to subjects that I skip
- Both appear in Linux Kernel Coding Style

- Linux Kernel bans using Hungarian naming
  - If unsigned `u_val` is modified to double
  - Should we change the name to `d_val` ?
  - But how about price `p_val` ?

**JoelOnSoftware:** Making Wrong Code Look Wrong explains it

## A Spoiler (Cont.)



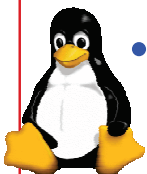
- Linux Kernel accepts **goto** in some cases
  - I totally agree with their criteria
  - But it's a slippery slope – be careful
- Here is an industrial example:

```
Status Class::set_status(int id, State state)
{
    Status status = OK;
    Container::const_iterator itr = cont_.find(id);
    if (itr == cont_.end()) {
        status = NOT_FOUND;
        goto bail;
    }
    status = itr->second->act(state);
bail:
    return status;
}
```

August Penguin – Bug Free SW. 2017 © by Yechiel M. Kimchi

23

## A Spoiler (Cont.)



- Linux Kernel accepts **goto** in some cases
  - I totally agree with their criteria
  - But it's a slippery slope – be careful
- Here is the code **without** **goto**:

```
Status Class::set_status(int id, State state)
{
    Container::const_iterator itr = cont_.find(id);
    if (itr != cont_.end()) {
        return itr->second->act(state);
    }
    return NOT_FOUND;
}
```

August Penguin – Bug Free SW. 2017 © by Yechiel M. Kimchi

24

## Expectations from Quality Software

What is Common to these *Parts*?

*Definitions*

Service = Interface + Implementation

Interface = Preconditions + Post Conditions  
– Invariants ☺

*A direct customer requirement*

Implementation = Correct

+ As Independent As Can Be<sup>(\*)</sup>

*An indirect customer requirement*

**(\*) Independent Commands/Expressions?**

**An Opportunity for Concurrency**

August Penguin – Bug Free SW. 2017 © by Yechiel M. Kimchi

25

## Requirements of a single *Part*

- **A *Part* should be dedicated to a single task**
  - **Easy to comprehend** (*Separation of concerns*)
- **A *Section* should be short and simple**
  - Most sections control other parts
  - **Easy to comprehend**
- **A *Part* should have clear boundaries**
  - **Easy to describe begin/end states**
  - **Easy to define pre/post conditions**
  - **Non *Sections*, have the basics of it for free**

August Penguin – Bug Free SW. 2017 © by Yechiel M. Kimchi

26

## Requirements of *Parts* – Independence

- *Parts* are as independent of one another as possible<sup>(\*)</sup>
  - Easy to comprehend
  - Easy to modify (fix or enhance)
  - Easy to reuse
  - Easy to test

(\*) For the four bullets above,  
we only have to know the few  
it directly depends on

## Requirements of *Parts* – Independence

- *Parts* are as independent of one another as possible

**High dependency**  
**does not make your program wrong –**  
**it just makes it harder to make it right**

## Expectations from Quality Software

### Meta-Rules: *Requirements from a service*

These four can be considered  
as **axioms** for quality SW

- **Independence** *(Implementation)*
- **Separation** *(Implementation)*
- **Controlled communication** *(Interface)*
- **Simplicity<sup>(\*)</sup>** *(Implementation)*

I ignore uniformity rules, which are mostly stylistic

## Simplicity<sup>(\*)</sup> A Word by a CS Icon

**“Simplicity and elegance are unpopular because they require hard work and discipline to achieve and education to be appreciated.”**

Edsger W. Dijkstra, 14 June 1989

## Explaining the Meta-Rules

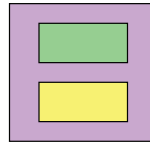
Independence, separation, simplicity,  
controlled communication

These four are  
not at all independent  
of one another

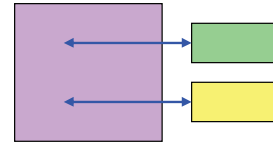
Independence  $\Rightarrow$  **Modularity** and downward  
separation  $\Rightarrow$  **Precondition** for independence

$\Rightarrow$  **Functions**

*What is yet another benefit?*



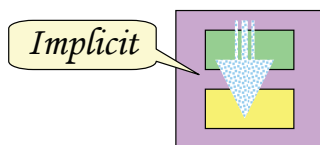
VS.



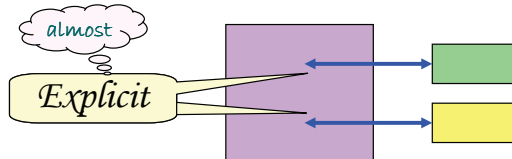
If a function contains two loops, it is almost impossible  
to test one of them separated from the other

## Explaining the Meta-Rules (cont.)

Interface  $\Rightarrow$  **Explicit** communication



VS.



$\Rightarrow$  **Global data** is used judiciously

Simplicity  $\Rightarrow$  **External** by the above three

$\Rightarrow$  **Internal** is context dependent

An abstract rule: *No nested sections*

*Example: Nested loops are rare*



## Safety-Critical Standards Break the Rules

### MISRA-C (2004) has [Rule 8.7]:

- Requiring to minimize usage of global variables
  - “Whether objects are declared at the outermost or innermost block is largely a matter of style”
  - Namely, reducing the dependency among neighboring, or nested blocks is a matter of style [Oops?]

```
int main()
{ int sum = 0, i = 0;
  while (i < 100) {
    int num;
    scanf("%d", &num);
    sum += num;
    ++i;
  }
  printf("%d\n", sum);
  return 0;
}
```

First semester,  
end of 1<sup>st</sup> lecture

## OOPS (Cont.)

In “C Unlashed” (SAMS, 2000), p. 206, we see

```
int main(void)
{
  size_t len = 0;
  char buffer[1024] = {0};
  if (fgets(...) != NULL)
  {
    size_t len = strlen(buffer);
    printf(...);
  }
  return 0;
}
```

**Warning:** ‘len’ is assigned a value that is not used

In this case:  
Declare len  
inside if block

After a long discussion,  
on p. 208 it’s written:

“If you ever find a style that will, for any program, produce no warnings at all under all ANSI C compilers at their pickiest warning level, the world would be very glad to hear from you.”

## Misguided Coding Standards Guides

**JSF-AV C++ (2005) has (# is rule's no.):**

- **#1** Any one function (or method) will contain no more than 200 logical source lines of code (L-SLOCs).
  - **Rationale:** Long functions tend to be complex and therefore difficult to comprehend and test.
- **Fact:** A function with 200 lines of logical (actual) code, breaks all four meta-rules above.

## Misguided Coding Standards (Cont.)

**More from JSF-AV C++ (2005) :**

- **#3** All functions shall have a **cyclomatic complexity** number of 20 or less
  - **Rationale:** Limit function complexity.

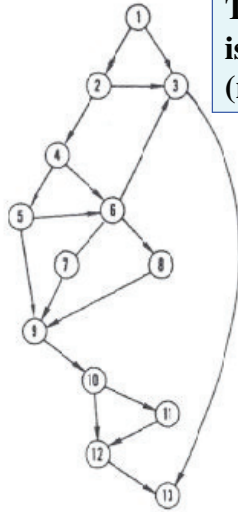
*Flow-chart  
complexity*

I am not very good at visualizing cyclomatic complexity,  
so I went to the original paper that had introduced this concept:

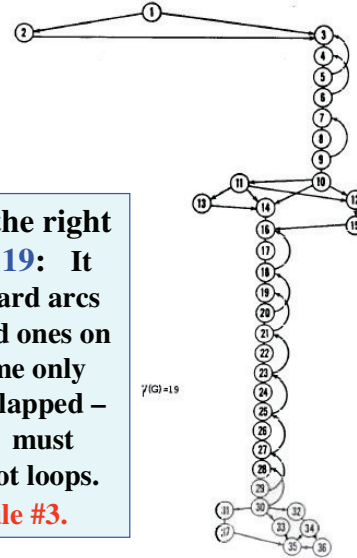
# Visualizing Cyclomatic Complexity

**McCabe, 1976**

The diagrams are from the original paper defining cyclomatic complexity.

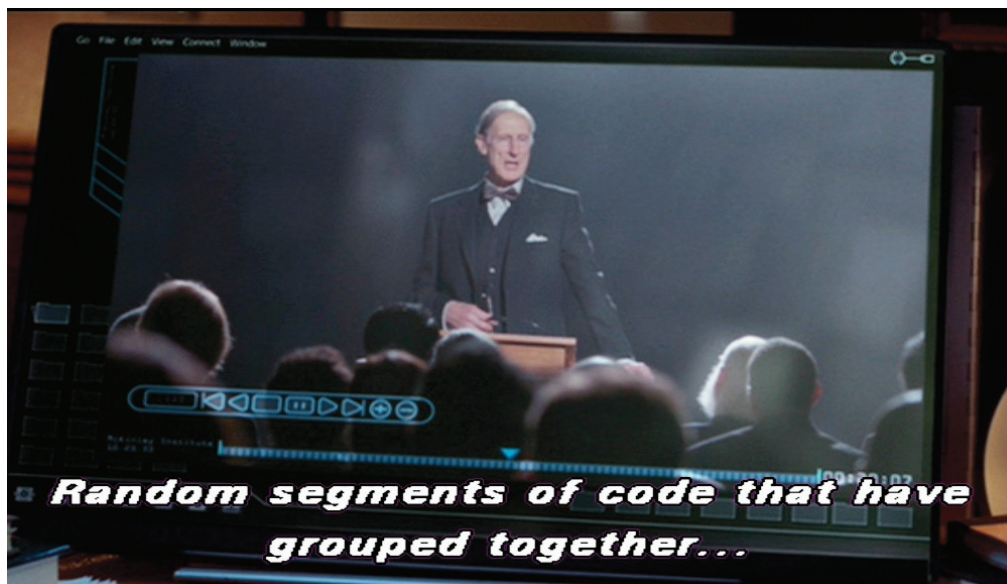


The one on the left is measured 8 (no backward arcs).



The one on the right is measured 19: It has 11 backward arcs (all the curved ones on the right), some only partially overlapped – meaning they must be gotos, not loops. Go back to rule #3.

## “I Robot” – the movie



## **“I Robot” – the movie**



August Penguin – Bug Free SW. 2017 © by Yechiel M. Kimchi

39

## **“I Robot” – the movie**



August Penguin – Bug Free SW. 2017 © by Yechiel M. Kimchi

40

## Unrelated to the above

**The Washington Times** HOME NEWS ▾ OPINION ▾ SPORTS ▾

By Bill Gertz - - Wednesday, March 30, 2016 [Print](#)

The U.S. military's frontline [redacted] that are delaying operational deployment, according to the Pentagon's senior weapons tester.

**J. Michael Glimore**, director of operational test and evaluation within the Office of the Secretary of Defense, told a House hearing last week that the F-35 — which is being built in three different versions for the Air Force, Navy and Marine Corps — is “at a critical time.”

“There are shortfalls in electronic warfare, electronic attack, shortfalls in the performance of distributed aperture system and other issues that are classified,” Mr. Glimore said March 23. “With regard to mission assistance, stealth aircraft are not visible to achieve success against the modern stressing mobile threats. We’re relying on our \$400 million investment in F-35 to provide mission systems [that] must work in some reasonable sense of that word.”

August Penguin – Bug Free SW. 2017 © by Yechiel M. Kimchi

41

## Before we translate theory to practice

Before we derive rules from meta-rules

There are computer-scientists that claim that  
**Programming is a Tool**

My reaction is:

- **You are right**, but the difference between us is
- You think of it as the **plumber's hammer** and **chisel**
  - Break the wall, fix the leak and cover.
- I think of it as the **sculpturer's hammer** and **chisel**
  - If you don't use it the right way, you'll break the marble.

August Penguin – Bug Free SW. 2017 © by Yechiel M. Kimchi

42

## Practical Rules – Independence

Independence  $\Rightarrow$  **No** goto *(and much more)*

**Did you know?** *There are three versions of goto*

- **Control:** The well known goto command
  - Allows two sections to mix together
- **Value:** Global variables
  - Allows several sections to share a value
  - Using a value created by an unknown section
- **Type:** Using ptr/ref casting<sup>(\*)</sup>
  - It's not a conversion, it is an assumption

**Which one is worst?**

(\*) Thanks to Marshall Cline, owner of C++ FAQs

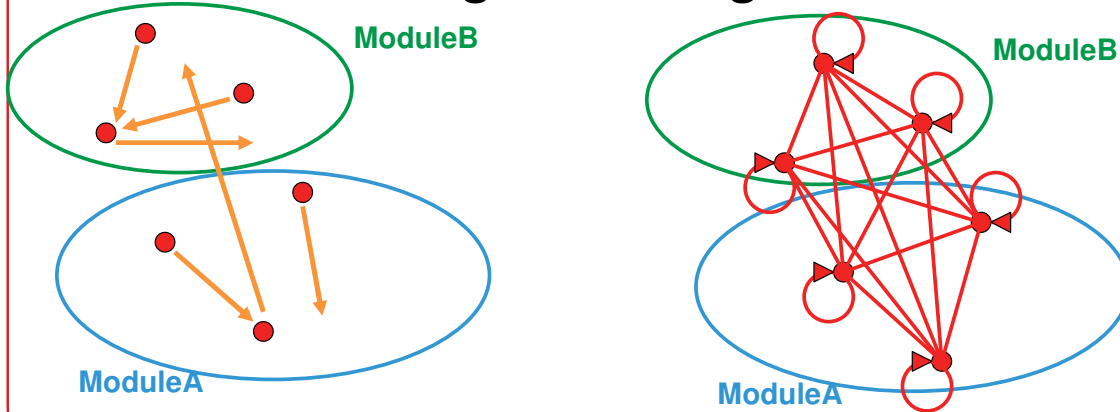
August Penguin – Bug Free SW. 2017 © by Yechiel M. Kimchi

43

## Practical Rules – Independence

**What's Worse?**

**a cross-module goto or a global variable**



August Penguin – Bug Free SW. 2017 © by Yechiel M. Kimchi

44



## Practical Rules – Separation

### Separation

**Two common techniques  
for separation are **hiding**  
& **hiding implementation****

Separation is the principal technique  
for achieving **Independence**

## Practical Rules – Separation

Separation ➡ Functions *(hiding implementation)*

**The Biggest **Misconception**  
About Functions** (◇)

(◇) Except interface functions

## Practical Rules – Separation

Separation ➡ Functions *(hiding implementation)*

~~The Purpose of Functions is  
to Eliminate Code Duplication~~

## Practical Rules – Separation

Separation ➡ Functions *(hiding implementation)*

The Purpose of Functions  
is to make the Code  
Easier to Understand

- By naming a piece of code (saving comments)
- By hiding its implementation (high level code)
- By making pre/post-conditions explicit
  - Also allowing (partial) isolation for testing
- By making the hosting code/function shorter



## Practical Rules – Separation

### Separation

What about performance?

- Functions are for easy understanding
- Separate different concerns (aka SRP)
- The evil of code-duplication (\*)
- Encapsulation (using functions/classes/modules)
- No getters (Tell, don't ask)

(\*) The lesson of Ariane5

## Principles for Interface

- Interface design is very delicate
  - Both separates and connects entities (contradictory) (\*)
  - Modifications are painful (expansions are OK)

(\*) JoelOnSoftware: Leaky Abstractions

Dependency  
inversion

- Should be easy to use (best: intuitive)
  - Helps achieving designed operations
  - Prevents misusing it (error or malice)

Strong  
typing

## Practical Rules – Interface

### Controlled communication (Interface)

- Minimal and complete (S. Meyers Eff. C++ 2<sup>nd</sup>)
  - Minimize **width** (#functions / #parameters)
  - Minimality and completeness are context dependent
- Minimize number of users (**area** =  $||^*|U| = \sum_{(i \in I)} u_i$ )
- Make pre/post-conditions explicit
- Interface should preserve invariants
  - No setters

What about performance?

## Principles for Simplicity

*simplicity* is the most intangible characteristic

- **Subjective** (A novice vs. an expert)
- **Subjective** (habits and taste)
- Should be based on *culture* and *idioms*
  - Defined by **experts**, not by majority
  - Culture and idioms are not stable (The singleton case)

## Misguided Simplicity

*“Write your code in a form that can be maintained by the less experienced member of the project”*

**Such practices hinder progress:**

- Mainly because novices will never learn by example
- Some implementations are either sophisticated or bad

## Simplicity is Difficult

*“The present letter is a very long one,  
simply because I had no leisure to make it shorter.”*

**Blaise Pascal**

**Edsger W. Dijkstra:**

*“... simplicity and elegance are unpopular  
because they require hard work and discipline to achieve  
and education to be appreciated.”*

## Practical Rules – Simplicity

### Simplicity

- Short functions and single task (aka SRP)
- Shallow nesting – low (cyclomatic) complexity
- Minimize function's side-effect
  - Avoid global variables
- Visible side-effects - via interface
- *“Comment only what the code cannot say”*

What about performance?

## Coding Standards Guides

I argue that coding standards documents:

- Miss most of the aforementioned coding rules
- Do not distinguish between essence and style.

Indeed, they are **more** about low-level style  
– e.g., **uniformity** and language **don'ts** + mini-rules.  
Those are very important in practice,  
but they do not replace the general rules.

## Coding Standards Guides

**Of all the rules above MISRA-C (2004) has:**

- (adv) Restrictions on pointer casting
- No goto/continue (break is restricted)
- Avoid using unnecessary global data.

All of them are **goto** related.

## Coding Standards Guides

**Of the rules above JSF-AV C++ (2005) has:**

- Class interface should be complete and minimal
- Const member functions are better
- (adv) usage of invariants
- No goto/continue (break is restricted)
- (adv) avoiding global variables
- Restricts down-casting (and casting in general)

Last three of the six items are **goto** related

## How many general rules are missing?

Why are so many rules missing?

And I have shown just about half of what I have

OTOH

## Coding Standards Guides

### **MISRA-C (2004) has:**

- 14.7 (req) A function shall have a single point of exit at the end of the function.
- 14.9 (req) An **if** (**<expr>**) construct shall be followed by a compound statement.
- 14.10 (req) All **if ... else if** constructs shall be terminated with an **else** clause.

See next slide

### **JSF-AV C++ (2005) has :**

- AV Rule 113 (MISRA Rule 82, Revised)  
Functions will have a single exit point.
- AV Rule 192 (MISRA Rule 60, Revised)  
All if, else if constructs will contain either a final else clause or a comment indicating why a final else clause is not necessary.

## Do we really want single exit?

```
bool IsPrime(int n)
{
    if (n < 0) n = -n;
    if (n < 2) return FALSE;
    if (n == 2) return TRUE;
    if (n % 2 == 0) return FALSE;

    int bound = Round2Whole(sqrt(n) + 1);
    for (int div = 3; div <= bound; div += 2) {
        if (n % div == 0) return FALSE;
    }
    return TRUE;
} /* End - IsPrime() - */
```

What is the cost of making this function following the rules ?

## Yet, they have

### MISRA-C (2004):

- 17.6 (req) The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.

### JSF-AV C++ (2005):

- #111 A function shall not return a pointer or reference to a non-static local object

See next slide

## Coding Standards Guides

(\*)The first day I've got the new, 3<sup>rd</sup> edition, of Stan Lippman's *C++ Primer*, I found three related errors: an automatic variable returned by reference.

Stan's response to my e-mail was not just apologetic – he couldn't understand how that error eluded both his review as well as the technical reviewers.

**Do you think that a rule such as the above  
could have helped them?**

- Such rules belong to **learning**
- Most are checked by **lint-like tools**

Coding standard is about **conscious activity**  
not about unintentional errors

## Some Missing Rules

**Linux Kernel's contains 8 of the 26 Rules.**

**Some of the missing ones are:**

- Minimize global objects (not just variables)
- Minimize scope (not just variables)
- Minimize side-effects (not just functions)
- Minimize interface (interface is minimal)
- Minimize surprise (explicit pre/post-conditions and invariants)



## A general suggestion

**Separate Coding Style Guides to three parts:**

- 1. Uniformity rules – related to perception only**
- 2. Knowledge rules – pitfalls of the language etc.**
- 3. Design + Programming rules – language indep.**

## Example: What's Wrong 1

### A Simple Industrial Example

```
PlumberStatus
Tap::open_tap(const string& tap_name)
{
    LockSys<Mutex> LL(tap_lock_);
    TapMap::const_iterator it =
                                taps_.find(tap_name);
    if (it == taps_.end()) {
        return PLUMB_TAP_NOT_FOUND;
    }
    it->second->operate(true);
    return PLUMB_OK;
}
```

## Example : What's Wrong 2

### What's the difference?

PlumberStatus

```
Tap::close_tap(const string& tap_name)
{
    LockSys<Mutex> LL(tap_lock_);
    TapMap::const_iterator it =
                                taps_.find(tap_name);
    if (it == taps_.end()) {
        return PLUMB_TAP_NOT_FOUND;
    }
    it->second->operate(false);
    return PLUMB_OK;
}
```

August Penguin – Bug Free SW. 2017 © by Yechiel M. Kimchi

67

## Examples and Observations

### What's The Problem?

- Is it **code duplication**?
  - Let's see:
    - **After extracting out the common parts we get**

```
PlumberStatus
Tap::open_tap(const string& tap_name)
{
    LockSys<Mutex> LL(tap_lock_);
    if (!tap_found(tap_name)) {
        return PLUMB_TAP_NOT_FOUND;
    }
    it->second->operate(true);
    return PLUMB_OK;
}
```

August Penguin – Bug Free SW. 2017 © by Yechiel M. Kimchi

68

## Code Duplication is just the Symptom

The real problem:  
Each one of them has two tasks

- Delegation (of a function call) after checking
- Wrapping: Transforming `boolean value => name`

## Single Task Implementation - Delegation

```
PlumberStatus
Tap::operate_tap(const string& name, bool open)
{
    LockSys<Mutex> LL(tap_lock_);
    TapMap::const_iterator it =
                                taps_.find(tap_name);
    if (it == taps_.end()) {
        return PLUMB_TAP_NOT_FOUND;
    }
    it->second->operate(open);
    return PLUMB_OK;
}
```

## Single Task Implementation – Wrappers

With appropriate design, these may be made  
non-member non-friend functions

```
inline PlumberStatus  
Tap::open_tap(const string& tap_name)  
{ return operate_tap(tap_name, true);}
```

Both functions are `inlined`, so they consume  
neither executable space nor executable time

```
inline PlumberStatus  
Tap::close_tap(const string& tap_name)  
{ return operate_tap(tap_name, false);}
```

## The Original has a Third Problem

### It enforces awkward usage

```
if (activation_required) {  
    open_tap(name);  
} else {  
    close_tap(name);  
}
```

### Instead of

```
operate_tap(name, activation_required);
```

## The SW Process

- When a project/task has more than two developers, there must be some management – therefore, a process
- However, a process addresses the group
  - But, code-development is solitary
    - Or, by pairs (XP)

## Approaches to Software Quality (cont.)

### An Important Observation

- Processes mainly aim at **collaboration** level, from a team – up to a corporation.
- The basics of software development is done at the **personal** level, individually.
- Therefore, a software development group, no matter its size, resembles a **team of chess players** – not a football team.

## SW Obstacles

## Deadlines: Quality is Last

- Relies on “bugs are inevitable” perception by the public.
- Delaying features seems more appropriate

## Next Quarter's Bottom-line

### As unpleasant as it is

- Politicians look forward to next elections
  - Most of the time more than a year ahead
- CEOs look forward to next quarter
  - Most of the time less than two months

## Fighting Bugs (any resemblance to reality is imaginary)

Imagine two SW-engineers that get to share an assignment for twelve weeks. The first one finishes his part in ten weeks – he gets a (+) for quick coding. Then he tests his part and finds, say, 40 flaws, and he fixes them in six weeks. He then gets another (+) for quickly fixing many bugs.

The second is slower, and finishes coding in fourteen weeks – he gets a (–) for slow coding. While testing he finds three nasty bugs, and it takes him two weeks to fix them. He then gets

## Process vs. Knowledge

Here is a mere speculation: Why companies are ready to spend so much money on processes? Several orders of magnitude when compared to what they spend on improving their staff's qualifications? Start-ups excluded. If you follow the money, a simple answer pops-up: When engineers leave the company to another, they take their knowledge with them – but they cannot take the process with them.

## The “Conspiracy” Argument

### Mark Minasi “The Software Conspiracy”

- **Would you accept buggy hardware?**
  - **No one does** (remember the Pentium bug)
- **Will the judicial system help?**
  - Greedy lawyers blur distinction between error and negligence in medicine, and cause bad *defensive medicine*.
- **Will regulation help?**
  - Bell was divided to seven companies (process took 1974 – 1984)
  - Microsoft case (1998 – 2004) failed, after a recusal of the original judge



# *Thank You*

## Q & A

**yechiel.kimchi@gmail.com**

August Penguin – Bug Free SW. 2017 © by Yechiel M. Kimchi

81

## Sources

- [1] Charles C. Mann “Why software is so bad?” MIT Technology Review, 2002 <http://www.technologyreview.com/featuredstory/401594/why-software-is-so-bad/>
- [2] Robert N. Charette “Why Software Fails” IEEE Spectrum 2005 <http://spectrum.ieee.org/computing/software/why-software-fails/>
- [3] Mark Minasi, “The Software Conspiracy”, McGraw-Hill, 1999
- [4] Y. Kimchi, “Coding with Reason”, in “97 Things Every Programmer Should Know”, ed. K.Henney, O’Reily 2010

August Penguin – Bug Free SW. 2017 © by Yechiel M. Kimchi

82

## Sources (cont.)

- [5] Beth Layman “An Interview w. Jerry Weinberg”  
Software Quality Professional, v.3 no.4, 2001 ASQ  
<http://www.stickyminds.com/interview/software-engineering-state-practice-interview-jerry-weinberg>
- [6] J. Pontin, “The problem with Programming: Interview w. B. Stroustrup”, MIT Technology Review, 2006  
<http://www.technologyreview.com/news/406923/the-problem-with-programming>
- [7] Misra-C: <http://www.misra.org.uk/>  
Retrieved July 30, 2015

## Sources (cont.)

- [8] <http://caxapa.ru/thumbs/468328/misra-c-2004.pdf>  
Retrieved January 20, 2017
- [9] <http://www.stroustrup.com/JSF-AV-rules.pdf>  
Retrieved July 30, 2015
- [10] S. Summit, (Retrieved, July 30, 2015)  
[http://www.eskimo.com/~scs/readings/software\\_elegance.html](http://www.eskimo.com/~scs/readings/software_elegance.html)
- [11] Dijkstra E. W.: Letters to the editor: goto statement considered harmful. Comm. ACM, V.11:3, 147-148 (1968)

## Sources (cont.)

- [12] Wulf W., Shaw M.: Global variable considered harmful. ACM SIGPLAN Notices, V. 8:2, 28-34 (1973)
- [13] S. Saariste: Resist the temptation of the singleton pattern. In "97 Things Every Programmer Should Know", ed. K. Henney, O'Reilly 2010
- [14] B. Klemens, MATH You Can't Use, Patents, Copyright, and Software. Brookings Institution Press, Washington, D.C. 2006
- [15] <https://www.kernel.org/doc/Documentation/CodingStyle> retrieved January 20, 2016

## Sources (cont.)

- [16] Cline M.: C++ FAQs (owner): private communication, 2011.
- [17] Meyers S.: Effective C++. Addison Wesley 2<sup>nd</sup> Ed. (1998), 3<sup>rd</sup> Ed. (2005)
- [18] Meyer B.: Object Oriented Software Construction. Prentice Hall PTR 2nd Ed. (1997)
- [19] Sommerville I.: Software Engineering. Pearson Education Inc. 9th Ed. (2011)
- [20] Jongerius J.: Bug-Free C, Retrieved September 5, 2017  
<http://www.duckware.com/bugfreec/index.html>

## Is Software So Bad?

The most amazing achievement of the computer software industry is its continuing cancellation of the steady and staggering gains made by the computer hardware industry.

— **Henry Petroski** (Historian of Technology)

## Why Software is So Bad?

- “Why software is so bad?” (2002) [1]
- “Why Software Fails” (2005) [2]
- M. Minasi: “The Software Conspiracy” (1999) [3]
- An Interview w. Jerry Weinberg (2001) [4]

Q. “What ... major milestones of software engineering discipline in the last three decades?”

JW: “Well, I don’t think there have been any.”

Q. “Really?” JW: [explaining]

Q. “... what about ... testing ...?”

JW: “... it has just made them sloppier developers; they are just more encouraged to throw stuff over the wall to testing.”

## Some Quality Software Characteristics

- **Correct** - Meets functional specifications.
- **Useful** - Meets customer expectations.
- **Robust** - **External:** Resistant to user/environment errors.  
**Internal:** Easy to modify/enhance.
- **Friendly** - Easy to learn/use (human engineering).
- **Efficient** - Where required.

## The Basic Assumptions

Quoting Bjarne Stroustrup (the father of C++):

***“Computer science must be at the center of software systems development.”***

Communications of the acm, January 2010

***“... [C]orrectness, efficiency, and comprehensibility are closely related. Getting them right requires essentially the same tools.”*** (and same mindset [YMK])

IEEE Computer, January 2012

## Computer Science and SW Development

**Computer science must** be at the center of software systems development. If it is not, we must rely on individual experience and rules of thumb, ending up with less capable, less reliable systems, developed and maintained at unnecessarily high cost. We need changes in education to allow for improvements of industrial practice.

**Bjarne Stroustrup**, *communications of the acm*, January 2010

## Software Development for Infrastructure

It isn't enough to be disciplined in our specification of data structures and interfaces: we must also simplify our code logic. **Complicated control structures** are as dangerous to efficiency and correctness **as are complicated data structures**.

[C]orrectness, efficiency, and comprehensibility are closely related. Getting them right requires essentially the same tools.

**Bjarne Stroustrup**, *IEEE Computer*, January 2012

## What about Performance (efficiency)?

**Did I forget them? *None in the least***

*First, it's a practical requirement, not abstract; like correctness, and robustness*

*Second*

**I will show that most rules improve efficiency**

*Efficiency comes from roughly three sources:*

- **Algorithms** *(has nothing to do with the code)*
- **Implementing algorithms<sup>(#)</sup>** *(depends on code)*
- **HW related code tweaks** *(may break code structure)*

**(#) Only this one (2<sup>nd</sup>) depends on coding rules**

## Practical Rules – Independence

**Landmarks for `goto` and global variables:**

- Dijkstra E. W.: Letters to the editor: `goto` statement considered harmful. Comm. ACM (1968)
- Wulf W., Shaw M.: Global variable considered harmful. ACM SIGPLAN Notices (1973)
  - They don't say that global variables are worse than **gotos**
- S. Saariste: Resist the temptation of the singleton pattern. In "97 Things Every Programmer Should Know" (2010)
  - Nowadays, the **singleton** pattern is considered an **anti-pattern**.