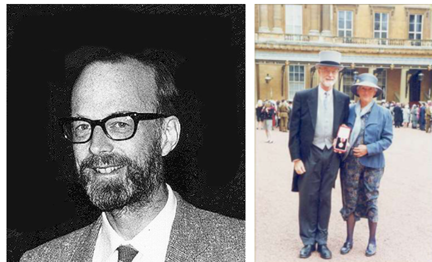
ANTONY HOARE

Made enduring contributions to programming language design and definition

A.M. TURING AWARD 1980

C. A. R. Hoare



Emeritus Professor of Computing at the University of Oxford and is now a senior researcher at Microsoft Research in Cambridge, England.
He received the 1980 ACM Turing Award for "his fundamental contributions to the definition and design of programming languages." Knighted by the Queen of England in 2000.

When Brunel's ship the SS Great Britain was launched into the River Thames, it made such a splash that several spectators on the opposite bank were drowned. Nowadays, engineers reduce the force of entry into the water by rope tethers which are designed to break at carefully calculated intervals.

When the first computer came into operation in the Mathematish Centrum in Amsterdam, one of the first tasks was to calculate the appropriate intervals and breaking strains of these tethers. In order to ensure the correctness of the program which did the calculations, the programmers were invited to watch the launching from the first row of the ceremonial viewing stand set up on the opposite bank. They accepted and they survived.

... [1.5 pages omitted]

I therefore suggest that we should explore an additional method, which promises to increase the reliability of programs. The same method has assisted the reliability of designs in other branches of engineering, namely the use of mathematics to calculate the parameters and check, the soundness of a design before passing it for construction and installation.

C. A. R. Hoare, *New Scientist*, 18 September 1986.

Correct Programming

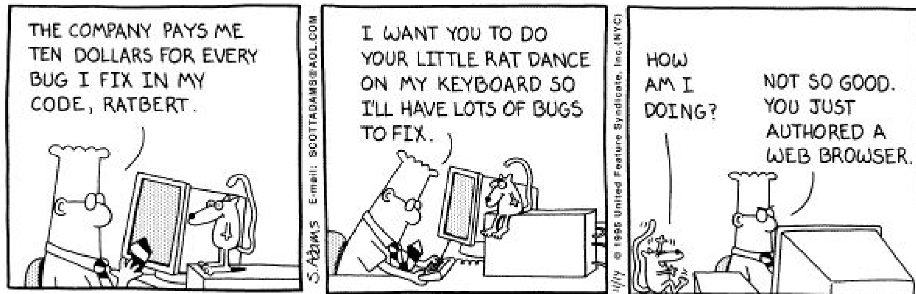
Most people can learn to write simple programs. Few people can learn to write correct programs.

In September 1999, NASA's Mars Climate Orbiter crashed into Mars instead of going into orbit. Preliminary findings indicated that one team of programmers used English units while the other used metric units for a key spacecraft operation. Total mission cost of $327.6 million.

Programming Without Thought

If I let my fingers wander idly over the keys of a typewriter it might happen that my screed made an intelligible sentence. If an army of monkeys were strumming on typewriters they might write all the books in the British Museum.

A. S. Eddington. *The Nature of the Physical World: The Gifford Lectures, 1927*. New York: Macmillan, 1929, page 72.



Copyright © 1995 United Feature Syndicate, Inc.
Redistribution in whole or in part prohibited

From an empirical study at Microsoft:

> *If the developers are mature enough to understand the code base and write useful assertions, it is highly likely that they understand the code base, which will lead to a lower fault density. We feel there is an urgent need in educating students about the utility of software assertions.*

Levels of Correctness

1. legal according to syntax rules
2. ... fine print
3. good style
4. makes sense
5. no runtime errors
6. works on some data (tested)
7. formal correctness with respect to some specification
8. *is* correct

Correctness

The opening levels of correctness are more familiar since they are obvious parts of the development cycle.
It is the later levels which are more subtle and difficult. One reason that state of software in the world is disappointing is that programmers tend to pay less attention to the later levels because

1. they are unwilling to put in the effort,
2. they are pressed for time, or
3. not enough emphasis is placed on correctness.

For this reason it is important to examine correctness.

Syntax

The Java compiler checks to see if the input file conforms to a precise set of syntax rules.

Adherence to syntax rules is necessary for the program to successfully communicate its actions to the computer and to any human reader.

Advanced Java programmer know more constructs enabling them to do write more complex programmer easier than novice programs. For example, `?:`, for-each loops, exception handling, interfaces, generics, wildcards, and so on.

The compiler may check that the program uses the constructs properly, but it does not teach the programmer how to use them.

Semantics

In addition to the syntax checks, the compiler checks lots of obvious and not so obvious rules.
For example, the programmer is free to choose any syntactically correct identifier to refer to objects in the program, but it is a rule that the identifier must be declared somewhere. So, if the programmer misspells a name, this will lead to a semantic error.

Style/Design

- good identifier names
- indent consistently, use white space judiciously
- use exception handling
- do not use gotos or multiple return
- avoid `import .*`
- localize declarations
- don’t repeat yourself
- avoid side effects: use functions, avoid non-local variables
- modularize

Localize Scope

- Declare variables right before you need them. (Localize scope.)
- Initialize the variable with the value you need.
- Don't reuse the variable with a different value or purpose.
- Mark the variable as `final`.

Runtime Errors

Some languages do not check for runtime errors. This leads to something worse than a runtime error—the absence of a runtime error. For example, by not checking array indexes or doing illegal operations on pointers, C and C++ allow the program to write over useful parts of memory. This is called a buffer overflow and is the cause of much malware.

This is not possible in Java, Ada (checks can be suppressed), C#, Python, Modula-3, Haskell, or any recently designed high-level language.

Runtime Errors

Java was designed so that compiler would detect at compile-time many things which would be runtime errors in other languages. No language can eliminate runtime errors altogether by improved compile-time checking. Some Java runtime errors (exceptions) represent a logical error (a mistake, a bug, by the programmer as opposed to, say, bad luck) in the program.

- `java.lang.ArithmeticException` (e.g., division by zero)
- `java.lang.NullPointerException` ("billion dollar mistake")
- `java.lang.ArrayIndexOutOfBounds`

Billion Dollar Mistake — presentation by Hoare in 2009.

Some programming languages have evolved more safeguards, e.g., option types.

Null

In Kotlin, the type system distinguishes between references that can hold null (nullable references) and those that cannot (non-null references).

```
// Regular initialization means non-null by default
var a: String = "abc"
a = null // compilation error

var b: String? = "abc" // can be set null
b = null // ok
```

```
var a: String = "abc"
var b: String? = "abc"

val l1 = a.length  // Save the length in an Int
val l2 = b.length // error: variable 'b' can be null
val l3 = if (b != null) b.length else -1
val l4 = b?.length  // Int? value might be null
```

The hair style of Elvis Presley was well-known

Elvis operator `?:` in Kotlin

```
var b: String? = "abc"
val l = b?.length ?: -1
```

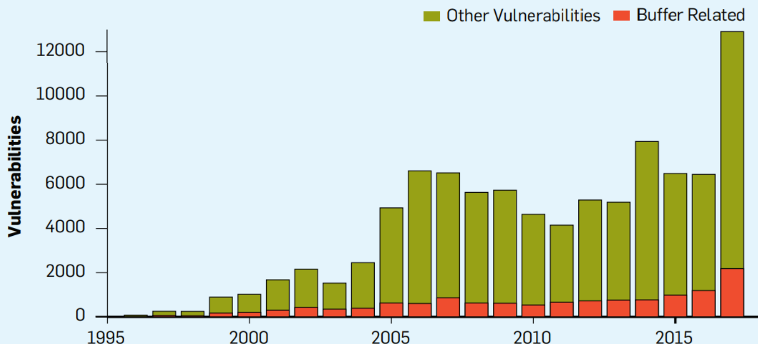
```
String b = "abc";
int l = b==null ? b.length : -1
```

Index Out of Bounds

Hoare in his 1981 Turing award lecture decried the lack of runtime range checking. The problem persists today.

> *In any respectable branch of engineering, failure to observe such elementary precautions would have long been against the law.*

Index Out of Bounds; Old But Peristent



Dror Feitson, *Tony's Law*, ACM, Feb 2019, volume 62, number 2, page 29

Random Fact 6.1 An Early Internet Worm

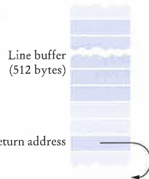
In November 1988, Robert Morris, a student at Cornell University, launched a so-called virus program that infected about 6,000 computers connected to the Internet across the United States. Tens of thousands of computer users were unable to read their e-mail or otherwise use their computers. All major universities and many high-tech companies were affected. (The Internet was much smaller then than it is now.)

The particular kind of virus used in this attack is called a worm. The virus program crawled from one computer on the Internet to the next. The worm would attempt to connect to `finger`, a program in the UNIX operating system for finding information on a user who has an account on a particular computer on the network. Like many programs in UNIX, `finger` was written in the C language. In C, as in C++, arrays have a fixed size. To store the user name to be looked up (say, walters@cs.sjsu.edu), the `finger` program allocated an array of 512 characters, under the assumption that nobody would ever provide such a long input. Unfortunately, C, like C++, does not check that an array index is less than the length of the array. If you write into an array using an index that is too large, you simply overwrite memory locations that belong to some other objects. In some versions of the `finger` program, the programmer had been lazy and had not checked whether the array holding the input characters was large enough to hold the input. So the worm program purposefully filled the 512-character array with 536 bytes. The excess 24 bytes would overwrite a return address, which the attacker knew was stored just after the line buffer. When that function was finished, it didn't return to its caller but to code supplied by the worm (see Figure 4). That code ran under the same super-user privileges as `finger`, allowing the worm to gain entry into the remote system. Had the programmer who wrote finger been more conscientious, this particular attack would not be possible. In C++, as in C, all programmers must be very careful not to overrun array boundaries.

One may well speculate what would possess the virus author to spend many weeks to plan the antisocial act of breaking into thousands of computers and disabling them. It appears that the break-in was fully intended by the author, but the disabling of the computers was a bug, caused by continuous reinfection. Morris was sentenced to 3 years probation, 400 hours of community service, and fined $10,000.

In recent years, computer attacks have intensified and the motives have become more sinister. Instead of disabling computers, viruses often steal financial data or use the attacked computers for sending spam e-mail. Sadly, many of these attacks continue to be possible because of poorly written programs that are susceptible to buffer overrun errors.

1 Before the attack



2 After the attack

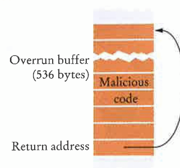


Figure 4
A "Buffer Overrun" Attack

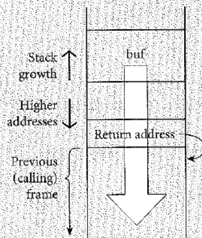
Runtime Guarantees Better Than Nothing

DESIGN & IMPLEMENTATION Scott

Stack smashing

The lack of bounds checking on array subscripts and pointer arithmetic is a major source of bugs and security problems in C. Many of the most infamous Internet viruses have propagated by means of *stack smashing*, a particularly nasty form of *buffer overflow attack*. Consider a (very naive) routine designed to read a number from an input stream:

```
int get_acct_num(FILE *s) {
    char buf[100];
    char *p = buf;
    do {
        /* read from stream s: */
        *p = getc(s);
    } while (*p++ != '\n');
    *p = '\0';
    /* convert ascii to int: */
    return atoi(buf);
}
```



If the stream provides more than 100 characters without a newline ('\n'), those characters will overwrite memory beyond the confines of buf, as shown by the large white arrow in the figure. A careful attacker may be able to invent a string whose bits include both a sequence of valid machine instructions and a replacement value for the subroutine's return address. When the routine attempts to return, it will jump into the attacker's instructions instead.

Stack smashing can be prevented by manually checking array bounds in C, or by configuring the hardware to prevent the execution of instructions in the stack (see the sidebar on page ©179). It would never have been a problem in the first place, however, if C had been designed for automatic bounds checks.

The whole economic boom in cybersecurity seems largely to be a consequence of poor engineering. We have allowed ourselves to become dependent on an infrastructure with the characteristics of a medieval firetrap—a maze of twisty little streets and passages bordered by buildings highly vulnerable to arson.

To a disturbing extent the kinds of underlying flaws exploited by attackers have not changed very much. . . . One of the most widespread vulnerabilities found recently, the so-called Heartbleed flaw in OpenSSL, was apparently overlooked . . . for more than two years. What was the flaw? Failure to apply adequate bounds-checking to a memory buffer.

Carl Landwehr, CACM, 2015, pages 24–25

Testing

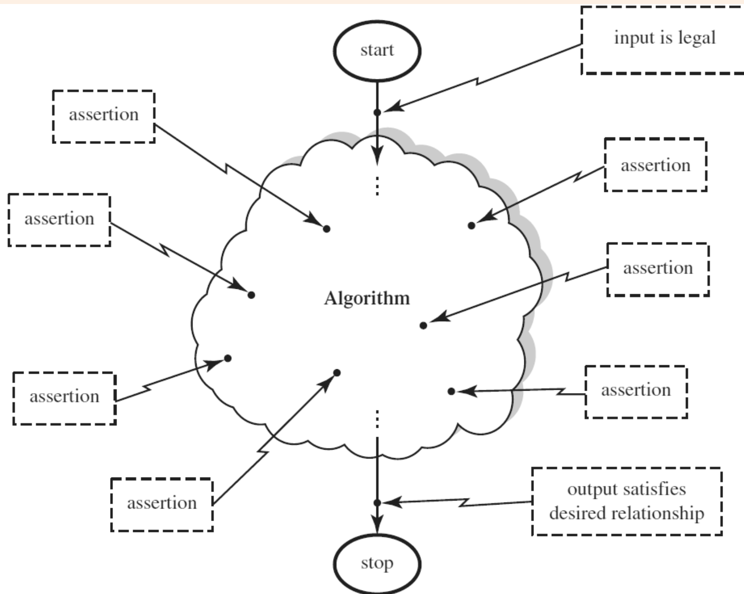
- code walk-throughs, code inspection
- unit testing
- integration testing
- bottom-up testing, drivers
- top-down testing, stubs
- regression testing
- black-box testing
- white-/clear-/glass- box testing
- statement and path coverage

Assertions

Assertion. An *assertion* is a boolean-valued expression relating the program variables, e.g., `x>0`, or `x+y>0`.
Precondition. A *precondition* is an assertion that must be true, *if* the following statement or block of code in the program is to work correctly. To make use of code correctly, one must insure that the preconditions are met.
Postcondition. A *postcondition* is an assertion that is true, after a statement or block of code has been executed, *if* the preconditions are met. The author of the code promises that the postconditions will be achieved.

Figure 5.5

Annotating an algorithm with invariants.



Assert statement

It looks like this: keyword `assert`, follow by a boolean-valued expression, followed by an optional string message.
It works like this: if the Java option "enable assertions" is set and the condition is false, the normal execution of the program is halted (an exception is raised) and failure reported. The source-code line number and stack trace confront the programmer with a specific problem in the code.

```
java -ea Main  - -enableassertions
java -da Main  - -disableassertions
java -eas Main - System assertions as well
java -das Main
```

Can specify package or class name as well.

Assert statement

The significance is: (1) the use of assertions allows the program to explain and document how the problem is solved. The program code alone does not capture the solution. Good comments are necessary to explain the solution, but they are (nearly always) incomplete, imprecise, and cannot be trusted. Furthermore, (2) the use of assertions facilitates testing and debugging by revealing misconceptions earlier during the execution of the program rather than at the end. Even if the programs runs to completion with the wrong answer, assertions eliminate portions of the program where the bug may be lurking.
A mathematical framework of statically proving (without testing) the programs meets its specification can evolved out of a complete systems of assertions (see Ada SPARK, programming by contract, etc).

Examples

- `Input.java` – input validation
- `Mult.java` – loop invariant, code verification
- `GeoPoint.java` – class invariant

Example

```
if (constraint[i][j]=='S') {
} else if (constraint[i][j]=='D') {
}

if (constraint[i][j]=='S') {
   // Manatee i and j must be the same species
} else if (constraint[i][j]=='D') {
   // Manatee i and j must be the different species
}
```

Example

```
if (constraint[i][j]=='S') {
} else if (constraint[i][j]=='D') {
}

if (constraint[i][j]=='S') {
    // Manatee i and j must be the same species
} else if (constraint[i][j]=='D') {
    // Manatee i and j must be the different species
}
```

Example

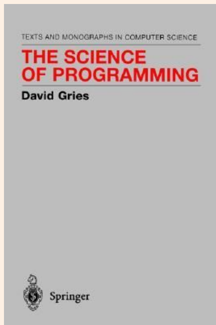
```
if (constraint[i][j]=='S') {
   // Manatee i and j must be the same species
} else if (constraint[i][j]=='D') {
   // Manatee i and j must be the different species
} else {
   // Manatee i and j are not constrained
}

if (constraint[i][j]=='S') {
   // Manatee i and j must be the same species
} else if (constraint[i][j]=='D') {
   // Manatee i and j must be the different species
} else {
   // Manatee i and j are not constraint-ed
   assert constraint[i][j]=='\u0000';
}
```

Class Invariant

A property that remains true about an object after every method call on the object.
E.e., “Account balance is always non-negative.”

Loop Invariant

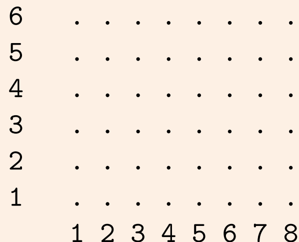
It is not possible to understand a program completely by mentally executing it.
The number of states is too large to comprehend.
Well-constructed programs have meaningful relationships among the program variables, and even though the values of the variables change the relationships do not.
It is through these unchanging relationships that it is possible to understand and to write correct programs.
Loop invariant. A *loop invariant* is an assertion relating the values of the program variables which is true before and after every execution of the body of a loop.
Invariants are a bridge from the mutable world of the machine to the timeless truths of mathematics.
Invariants are a bridge from dynamic systems which are hard to understand to state statements which are more amenable to precise analysis.



Gries:1981:SP

The following game illustrates the power of an invariant to understand dynamic systems.

Red and Blue take alternating turns, with Red going first. Red takes a turn by drawing a red line segment, either horizontal or vertical, connecting any two adjacent points on the grid that are not yet connected by any line segment. Blue takes a turn by doing the same thing, except that the line segment drawn is blue. Red's goal is to form a closed curve (i.e., a sequence of (four or more) distinct line segments starting at some point and returning to that point) comprised entirely of red line segments. Blue's goal is to prevent Red from doing so. The game ends when either Red has formed a closed curve or there are no more line segments to draw.



```java
public static int sum(int a[]) {
   int s = 0;
   for (int i = 0; i < a.length; i++) {
     // s is the sum of the first i array elements
     // s == a[0] + .. + a[i-1]
     s = s + a[i];
   }
   return s;
}
```

```java
public static int quotient(int n, int d) {
    int q = 0, r = n;
    assert n=q*d + r;
    while (r >= d) {
        r = r - d;
        q = q + 1;
        assert n=q*d + r;
    }
    return q;
}
```

```java
public static int power(int x, int n) {
    int p = 1, i = 0;
    assert p==Math.pow(x,i);
    while (i < n) {
        p = p * x;
        i = i + 1;
        assert p==Math.pow(x,i);
    }
    return p;
}
```

```
/*
   Convert a string of digits into a base 'base' number.  Raise an
   unchecked ArithmeticException if the result does not fit in a long.
*/
private static long decode (final String n, final int base) {
   assert base>0;
   long x = 0;
   for (final char c: n.toCharArray()) {
      x = multiplyExact (x,base);
      final int index = DIGITS.indexOf(c);
      assert 0<=index: String.format ("not a digit '%c'", c);
      assert index<base:
         String.format ("not a base %d digit '%c'", base, c);
      x = addExact (x, index+1L);
      assert x>0L;
   }
   assert x>=0L;
   return x;
}
```

Assert every blasted line in the function and then–if the stars align–you will find the bug on the next run of the code, because you are checking every bit of the function line by line.

George V. Nevill-Neil,“Kode Vicious Getting Off the Mad Path: Debuggers and assertions,” *CACM*, April 2022, volumn 85, number 4, page 26.