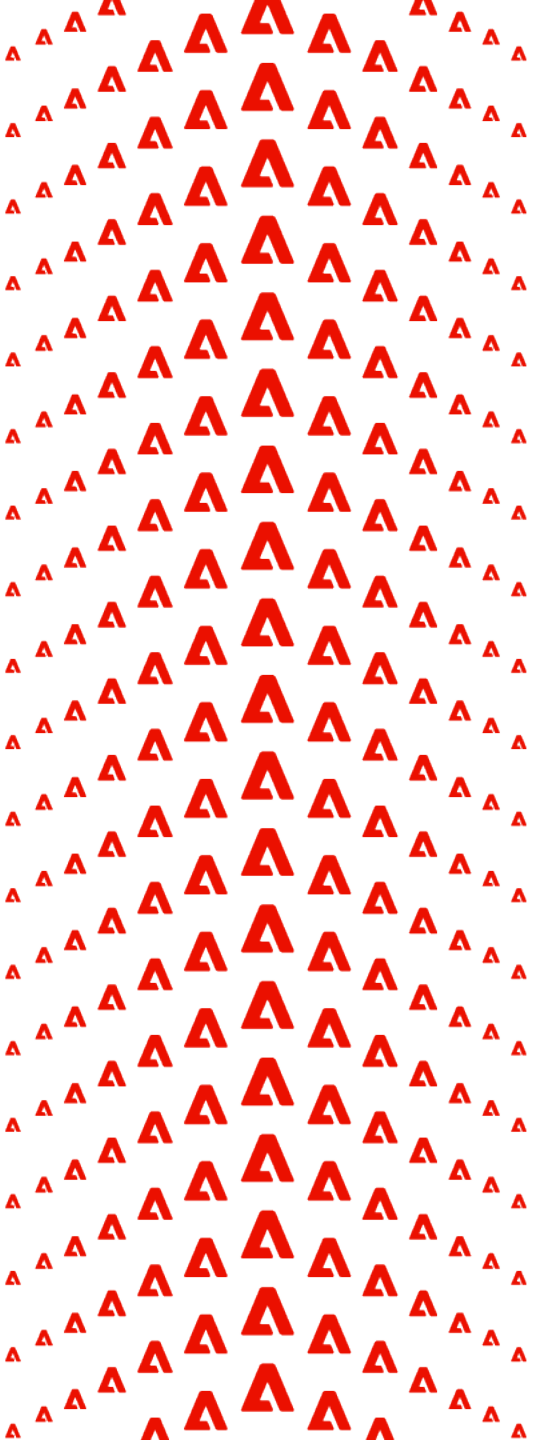
Warning: std::find() is broken!

Sean Parent | Sr. Principal Scientist, STLab

Artwork by **Momomi Sato** / Japan

“Understanding why software fails is important, but the real challenge is understanding why software works.”

– Alexander Stepanov



Robert W. Floyd

ASSIGNING MEANINGS TO PROGRAMS[1]

Introduction. This paper attempts to provide an adequate basis for formal definitions of the meanings of programs in appropriately defined programming languages, in such a way that a rigorous standard is established for proofs about computer programs, including proofs of correctness, equivalence, and termination. The basis of our approach is the notion of an interpretation of a program: that is, an association of a proposition with each connection in the flow of control through a program, where the proposition is asserted to hold whenever that connection is taken. To prevent an interpretation from being chosen arbitrarily, a condition is imposed on each command of the program. This condition guarantees that whenever a command is reached by way of a connection whose associated proposition is then true, it will be left (if at all) by a connection whose associated proposition will be true at that time. Then by induction on the number of commands executed, one sees that if a program is entered by a connection whose associated proposition is then true, it will be left (if at all) by a connection whose associated proposition will be true at that time. By this means, we may prove certain properties of programs, particularly properties of the form: "If the initial values of the program variables satisfy the relation R_1, the final values on completion will satisfy the relation R_2." Proofs of termination are dealt with by showing that each step of a program decreases some entity which cannot decrease indefinitely.

These modes of proof of correctness and termination are not original; they are based on ideas of Perlis and Gorn, and may have made their earliest appearance in an unpublished paper by Gorn. The establishment of formal standards for proofs about programs in languages which admit assignments, transfer of control, etc., and the proposal that the semantics of a programming language may be defined independently of all processors for that language, by establishing standards of rigor for proofs about

[1] This work was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (SD-146).

1967

Robert W. Floyd

ASSIGNING MEANINGS TO PROGRAMS[1]

an interpretation from being chosen arbitrarily, a condition is imposed on each command of the program. This condition guarantees that whenever a command is reached by way of a connection whose associated proposition is then true, it will be left (if at all) by a connection whose associated proposition will be true at that time. Then by induction on the number of commands executed, one sees that if a program is entered by a connection whose associated proposition is then true, it will be left (if at all) by a connection whose associated proposition will be true at that time. By this means, we may prove certain properties of programs, particularly properties of the form: "If the initial values of the program variables satisfy the relation R_1, the final values on completion will satisfy the relation R_2." Proofs of termination are dealt with by showing that each step of a program decreases some entity which cannot decrease indefinitely.

These modes of proof of correctness and termination are not original; they are based on ideas of Perlis and Gorn, and may have made their earliest appearance in an unpublished paper by Gorn. The establishment of formal standards for proofs about programs in languages which admit assignments, transfer of control, etc., and the proposal that the semantics of a programming language may be defined independently of all processors for that language, by establishing standards of rigor for proofs about

[1] This work was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (SD-146).

1967

An Axiomatic Basis for Computer Programming

C. A. R. Hoare
The Queen's University of Belfast, Northern Ireland*

In this paper an attempt is made to explore the logical foundations of computer programming by use of techniques which were first applied in the study of geometry and have later been extended to other branches of mathematics. This involves the elucidation of sets of axioms and rules of inference which can be used in proofs of the properties of computer programs. Examples are given of such axioms and rules, and a formal proof of a simple theorem is displayed. Finally, it is argued that important advantages, both theoretical and practical, may follow from a pursuance of these topics.

KEY WORDS AND PHRASES: axiomatic method, theory of programming, proofs of programs, formal language definition, programming language design, machine-independent programming, program documentation
CR CATEGORY: 4.0, 4.21, 4.22, 5.20, 5.21, 5.23, 5.24

1. Introduction

Computer programming is an exact science in that all the properties of a program and all the consequences of executing it in any given environment can, in principle, be found out from the text of the program itself by means of purely deductive reasoning. Deductive reasoning involves the application of valid rules of inference to sets of valid axioms. It is therefore desirable and interesting to elucidate the axioms and rules of inference which underlie our reasoning about computer programs. The exact choice of axioms will to some extent depend on the choice of programming language. For illustrative purposes, this paper is confined to a very simple language, which is effectively a subset of all current procedure-oriented languages.

2. Computer Arithmetic

The first requirement in valid reasoning about a program is to know the properties of the elementary operations which it invokes, for example, addition and multiplication of integers. Unfortunately, in several respects computer arithmetic is not the same as the arithmetic familiar to mathematicians, and it is necessary to exercise some care in selecting an appropriate set of axioms. For example, the axioms displayed in Table I are rather a small selection of axioms relevant to integers. From this incomplete set of axioms it is possible to deduce such simple theorems as:

$$x = x + y \times 0$$

$$y \leqslant r \supset r + y \times q = (r - y) + y \times (1 + q)$$

The proof of the second of these is:

A5 $(r - y) + y \times (1 + q)$

$= (r - y) + (y \times 1 + y \times q)$

A9 $= (r - y) + (y + y \times q)$

A3 $= ((r - y) + y) + y \times q$

A6 $= r + y \times q$ provided $y \leqslant r$

The axioms A1 to A9 are, of course, true of the traditional infinite set of integers in mathematics. However, they are also true of the finite sets of "integers" which are manipulated by computers provided that they are confined to *nonnegative* numbers. Their truth is independent of the size of the set; furthermore, it is largely independent of the choice of technique applied in the event of "overflow"; for example:

(1) Strict interpretation: the result of an overflowing operation does not exist; when overflow occurs, the offending program never completes its operation. Note that in this case, the equalities of A1 to A9 are strict, in the sense that both sides exist or fail to exist together.

(2) Firm boundary: the result of an overflowing operation is taken as the maximum value represented.

(3) Modulo arithmetic: the result of an overflowing operation is computed modulo the size of the set of integers represented.

These three techniques are illustrated in Table II by addition and multiplication tables for a trivially small model in which 0, 1, 2, and 3 are the only integers represented.

It is interesting to note that the different systems satisfying axioms A1 to A9 may be rigorously distinguished from each other by choosing a particular one of a set of mutually exclusive supplementary axioms. For example, infinite arithmetic satisfies the axiom:

$A10_I \quad \neg\exists x \forall y \qquad (y \leqslant x)$,

where all finite arithmetics satisfy:

$A10_F \quad \forall x \qquad (x \leqslant \max)$

where "max" denotes the largest integer represented.

Similarly, the three treatments of overflow may be distinguished by a choice of one of the following axioms relating to the value of max + 1:

$A11_S$	$\neg\exists x \quad (x = \max + 1)$	(strict interpretation)
$A11_B$	$\max + 1 = \max$	(firm boundary)
$A11_M$	$\max + 1 = 0$	(modulo arithmetic)

Having selected one of these axioms, it is possible to use it in deducing the properties of programs; however,

* Department of Computer Science

1969

An Axiomatic Basis for Computer Programming

C. A. R. Hoare
The Queen's University of Belfast, Northern Ireland*

of purely deductive reasoning. Deductive reasoning involves the application of valid rules of inference to sets of valid axioms. It is therefore desirable and interesting to elucidate the axioms and rules of inference which underlie our reasoning about computer programs. The exact choice of axioms will to some extent depend on the choice of programming language. For illustrative purposes, this paper is confined to a very simple language, which is effectively a subset of all current procedure-oriented languages.

2. Computer Arithmetic

The first requirement in valid reasoning about a program is to know the properties of the elementary operations which it invokes, for example, addition and multiplication of integers. Unfortunately, in several respects computer arithmetic is not the same as the arithmetic familiar to mathematicians, and it is necessary to exercise some care in selecting an appropriate set of axioms. For example, the axioms displayed in Table I are rather a small selection of axioms relevant to integers. From this incomplete set

* Department of Computer Science

It is interesting to note that the different systems satisfying axioms A1 to A9 may be rigorously distinguished from each other by choosing a particular one of a set of mutually exclusive supplementary axioms. For example, infinite arithmetic satisfies the axiom:

$A10_I \quad \neg\exists x \forall y \qquad (y \leqslant x),$

where all finite arithmetics satisfy:

$A10_F \quad \forall x \qquad (x \leqslant \max)$

where "max" denotes the largest integer represented.

Similarly, the three treatments of overflow may be distinguished by a choice of one of the following axioms relating to the value of max + 1:

$A11_S$	$\neg\exists x \quad (x = \max + 1)$	(strict interpretation)
$A11_B$	$\max + 1 = \max$	(firm boundary)
$A11_M$	$\max + 1 = 0$	(modulo arithmetic)

Having selected one of these axioms, it is possible to use it in deducing the properties of programs; however,

1969

Reasoning About Code: Hoare Logic

Reasoning About Code: Hoare Logic

- Hoare logic, also known as Floyd-Hoare logic, describes computation statements as a Hoare triple

Reasoning About Code: Hoare Logic

- Hoare logic, also known as Floyd-Hoare logic, describes computation statements as a Hoare triple

$$P\{Q\}R\,.$$

Reasoning About Code: Hoare Logic

- Hoare logic, also known as Floyd-Hoare logic, describes computation statements as a Hoare triple

$$P\{Q\}R\,.$$

- Where P is a precondition, Q is an operation, and R is the postcondition.

Reasoning About Code: Hoare Logic

- Hoare logic, also known as Floyd-Hoare logic, describes computation statements as a Hoare triple

$$P\{Q\}R\,.$$

- Where P is a precondition, Q is an operation, and R is the postcondition.
- Statements are combined with rules for *assignment*, *consequence*, *composition*, and *iteration*.

Reasoning About Code: Hoare Logic

- Hoare logic, also known as Floyd-Hoare logic, describes computation statements as a Hoare triple

$$P\{Q\}R\,.$$

- Where P is a precondition, Q is an operation, and R is the postcondition.
- Statements are combined with rules for *assignment*, *consequence*, *composition*, and *iteration*.
- Given a sequence of statements and assuming an initial precondition, if we can show that the subsequent postconditions guarantee subsequent preconditions are satisfied, then the program is correct.

Properties of Addition for Integers ($\mathbb{Z}$)

$\forall a, b, c \in \mathbb{Z}$	$(a + b) + c = a + (b + c)$	(associative)
$\forall a, b \in \mathbb{Z}$	$a + b = b + a$	(cummutative)
$\exists 0 \ni \forall a, 0 \in \mathbb{Z}$	$a + 0 = a$	(additive identity)
$\forall a \in \mathbb{Z}, \exists (\text{-}a) \in \mathbb{Z} \ni$	$a + (\text{-}a) = 0$	(additive inverse)

$\mathbb{Z} \neq$ int

$\mathbb{Z} \neq$ int

- When signed integers overflow or underflow the result is undefined.

$\mathbb{Z} \neq$ int

- When signed integers overflow or underflow the result is undefined.
- In Hoare logic this could be expressed as an additional axiom:

$\mathbb{Z} \neq$ int

- When signed integers overflow or underflow the result is undefined.
- In Hoare logic this could be expressed as an additional axiom:

$$\neg \exists (x \in \text{int}) \ni (x > max)$$

$\mathbb{Z} \neq$ int

- When signed integers overflow or underflow the result is undefined.
- In Hoare logic this could be expressed as an additional axiom:

$$\neg \exists (x \in \text{int}) \ni (x > max)$$

- Leading to a Hoare-triple:

$\mathbb{Z} \neq$ int

- When signed integers overflow or underflow the result is undefined.
- In Hoare logic this could be expressed as an additional axiom:

$$\neg \exists (x \in \text{int}) \ni (x > max)$$

- Leading to a Hoare-triple:

$$(a + b \leq max)\{\text{int n = a + b; }\}(n \leq max)$$

$\mathbb{Z} \neq$ int

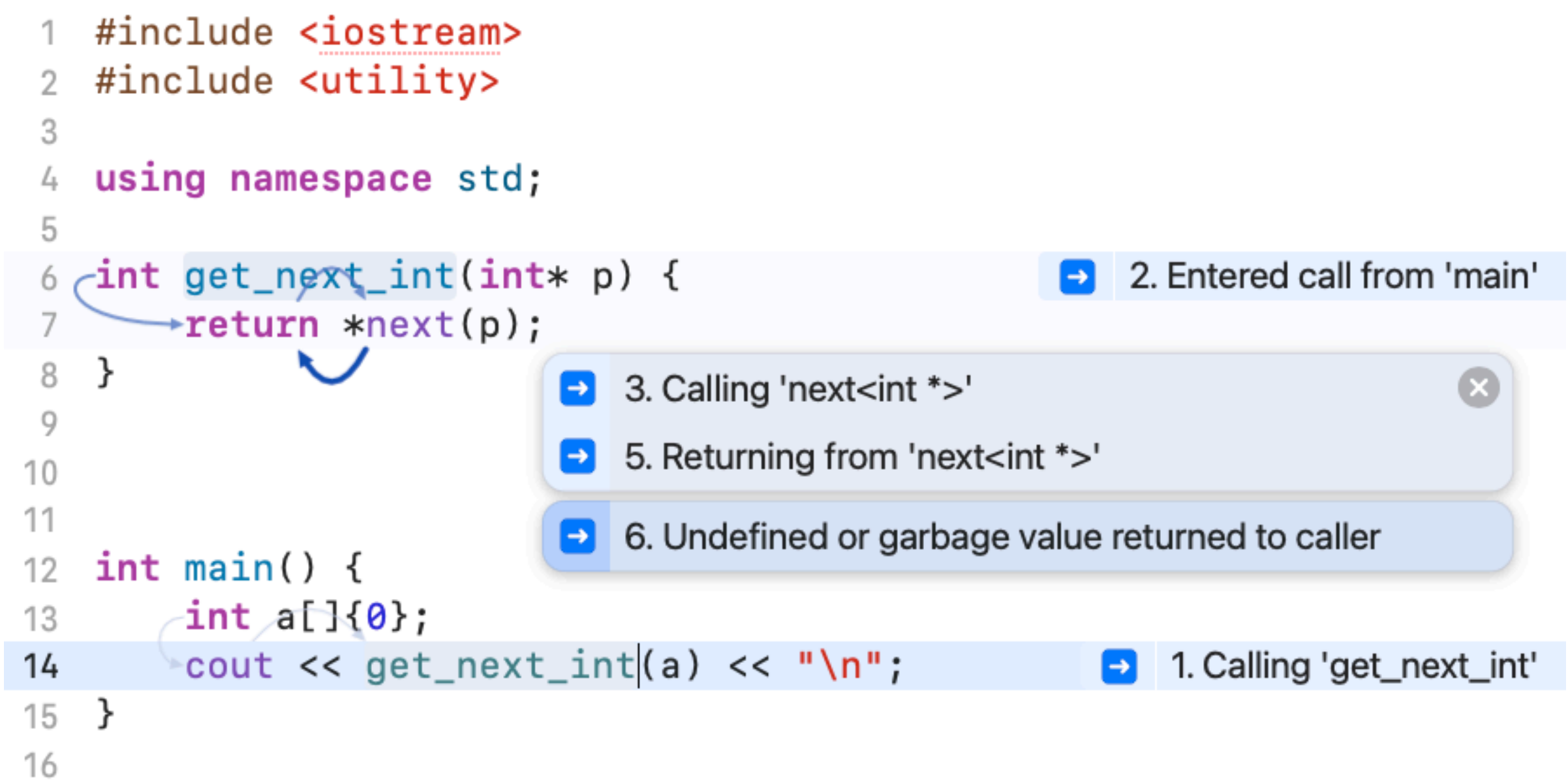
- When signed integers overflow or underflow the result is undefined.
- In Hoare logic this could be expressed as an additional axiom:

$$\neg \exists (x \in \text{int}) \ni (x > max)$$

- Leading to a Hoare-triple:

$$(a + b \leq max)\{\text{int n = a + b; }\}(n \leq max)$$

"Even the characterization of integer arithmetic is far from complete."
– C.A.R. Hoare, An Axiomatic Basis for Computer Programming



Applying "Design by Contract"

Bertrand Meyer

Interactive Software Engineering

As object-oriented techniques steadily gain ground in the world of software development, users and prospective users of these techniques are clamoring more and more loudly for a "methodology" of object-oriented software construction — or at least for some methodological guidelines. This article presents such guidelines, whose main goal is to help improve the reliability of software systems. *Reliability* is here defined as the combination of correctness and robustness or, more prosaically, as the absence of bugs.

Everyone developing software systems, or just using them, knows how pressing this question of reliability is in the current state of software engineering. Yet the rapidly growing literature on object-oriented analysis, design, and programming includes remarkably few contributions on how to make object-oriented software more reliable. This is surprising and regrettable, since at least three reasons justify devoting particular attention to reliability in the context of object-oriented development:

- The cornerstone of object-oriented technology is reuse. For reusable components, which may be used in thousands of different applications, the potential consequences of incorrect behavior are even more serious than for application-specific developments.
- Proponents of object-oriented methods make strong claims about their beneficial effect on software quality. Reliability is certainly a central component of any reasonable definition of quality as applied to software.
- The object-oriented approach, based on the theory of abstract data types, provides a particularly appropriate framework for discussing and enforcing reliability.

Reliability is even more important in object-oriented programming than elsewhere. This article shows how to reduce bugs by building software components on the basis of carefully designed contracts.

The pragmatic techniques presented in this article, while certainly not providing infallible ways to guarantee reliability, may help considerably toward this goal. They rely on the theory of *design by contract*, which underlies the design of the Eiffel analysis, design, and programming language[1] and of the supporting libraries, from which a number of examples will be drawn.

The contributions of the work reported below include

- a coherent set of *methodological principles* helping to produce correct and robust software;
- a systematic approach to the delicate problem of how to deal with abnormal cases, leading to a simple and powerful *exception-handling* mechanism; and

0018-9162/92/1000-0040$03.00 © 1992 IEEE

1992 (original 1986)

Applying "Design by Contract"

Bertrand Meyer

Interactive Software Engineering

Reliability is even more important in object-oriented programming than elsewhere. This article shows how to reduce bugs by building software components on the basis of carefully designed contracts.

- The cornerstone of object-oriented technology is reuse. For reusable components, which may be used in thousands of different applications, the potential consequences of incorrect behavior are even more serious than for application-specific developments.
- Proponents of object-oriented methods make strong claims about their beneficial effect on software quality. Reliability is certainly a central component of any reasonable definition of quality as applied to software.
- The object-oriented approach, based on the theory of abstract data types, provides a particularly appropriate framework for discussing and enforcing reliability.

The pragmatic techniques presented in this article, while certainly not providing infallible ways to guarantee reliability, may help considerably toward this goal. They rely on the theory of *design by contract*, which underlies the design of the Eiffel analysis, design, and programming language[1] and of the supporting libraries, from which a number of examples will be drawn.

The contributions of the work reported below include

- a coherent set of *methodological principles* helping to produce correct and robust software;
- a systematic approach to the delicate problem of how to deal with abnormal cases, leading to a simple and powerful *exception-handling* mechanism; and

0018-9162/92/1000-0040$03.00 © 1992 IEEE

1992 (original 1986)

Design by Contract

- Preconditions and postconditions are asserted in code, in the interface

```
set_second (s: INTEGER)
        -- Set the second from `s'.
    require
        valid_argument_for_second: 0 <= s and s <= 59
    ensure
        second_set: second = s
    end
```

Design by Contract

- *Class invariants* define postconditions for all (public) operations on a class

```
invariant
    second_valid: 0 <= second and second <= 59
```

Design by Contract

- *Class invariants* define postconditions for all (public) operations on a class

```
invariant
    second_valid: 0 <= second and second <= 59
```

- By extension, class invariants define a precondition for any operation taking an instance of the class as an argument

Limitations of Design by Contract

- Assertions must be expressible in code

Limitations of Design by Contract

- Assertions must be expressible in code
- Complexity of assertions is limited

Limitations of Design by Contract

- Assertions must be expressible in code
- Complexity of assertions is limited
  - A linear time assertion on a constant operation can transform code from $O(n)$ to $O(n \bullet m)$

Limitations of Design by Contract

- Assertions must be expressible in code
- Complexity of assertions is limited
  - A linear time assertion on a constant operation can transform code from $O(n)$ to $O(n \bullet m)$
- Cannot express universal quantifiers, ∀, or existential quantifiers, ∃

Key Contributions of Design by Contract

- Simplifies formal methods by inverting the process to *top-down*

Key Contributions of Design by Contract

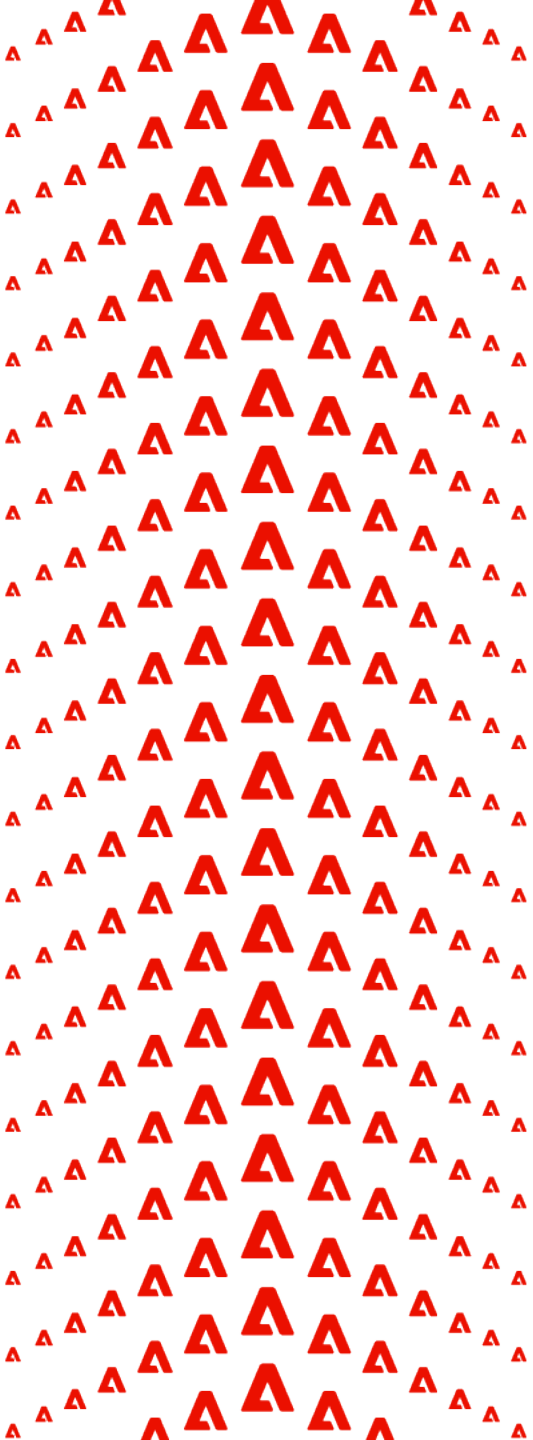
- Simplifies formal methods by inverting the process to *top-down*
  - Given a precondition, it is simpler to prove a function satisfies a postcondition than to derive a preconditions and postconditions from a composition of operations

Key Contributions of Design by Contract

- Simplifies formal methods by inverting the process to *top-down*
  - Given a precondition, it is simpler to prove a function satisfies a postcondition than to derive a preconditions and postconditions from a composition of operations
- Makes formal methods practical for every programmer

"With a sufficient number of users of an API, it does not matter what you promise in the contract: all observable behaviors of your system will be depended on by somebody."

– Hyrum Wright



Remove the first odd number (attempt 1):

```
vector a{0, 1, 2, 3, 4, 5};
```

Remove the first odd number (attempt 1):

```
vector a{0, 1, 2, 3, 4, 5};

auto p = remove_if(begin(a), end(a), [n = 0](int x) mutable {
    return (x & 1) && (n++ == 0);
});
```

Remove the first odd number (attempt 1):

```
vector a{0, 1, 2, 3, 4, 5};

auto p = remove_if(begin(a), end(a), [n = 0](int x) mutable {
    return (x & 1) && (n++ == 0);
});

a.erase(p, end(a));
display(a);
```

Remove the first odd number (attempt 1):

```
vector a{0, 1, 2, 3, 4, 5};

auto p = remove_if(begin(a), end(a), [n = 0](int x) mutable {
    return (x & 1) && (n++ == 0);
});

a.erase(p, end(a));
display(a);
```

{ 0, 2, 4, 5 }

Remove the first odd number (attempt 1):

```
vector a{0, 1, 2, 3, 4, 5};

auto p = remove_if(begin(a), end(a), [n = 0](int x) mutable {
    return (x & 1) && (n++ == 0);
});

a.erase(p, end(a));
display(a);
```

```
{ 0, 2, 4, 5 }
{ 0, 2, 3, 4, 5 }
```

Implementation of std::remove_if()

```
template <class F, class P>
auto remove_if(F f, F l, P pred) {
    f = find_if(f, l, pred); // <-- pred is passed by value

    if (f == l) return f;

    for (auto p = next(f); p != l; ++p) {
        if (!pred(*p)) *f++ = move(*p);
    }
    return f;
}
```

Remove the first odd number (attempt 2):

```
vector a{0, 1, 2, 3, 4, 5};
```

Remove the first odd number (attempt 2):

```
vector a{0, 1, 2, 3, 4, 5};

int n = 0;
auto p = remove_if(begin(a), end(a), [&n](int x) {
    return (x & 1) && (n++ == 0);
});
```

Remove the first odd number (attempt 2):

```
vector a{0, 1, 2, 3, 4, 5};

int n = 0;
auto p = remove_if(begin(a), end(a), [&n](int x) {
    return (x & 1) && (n++ == 0);
});

a.erase(p, end(a));
display(a);
```

Remove the first odd number (attempt 2):

```
vector a{0, 1, 2, 3, 4, 5};

int n = 0;
auto p = remove_if(begin(a), end(a), [&n](int x) {
    return (x & 1) && (n++ == 0);
});

a.erase(p, end(a));
display(a);
```

{ 0, 2, 3, 4, 5 }

Standard Requirement for Unary Predicate

- "Given a glvalue `u` of type (possibly `const`) T that designates the same object as `*first`, `pred(u)` shall be a valid expression that is equal to `pred(*first)`."

Standard Requirement for Unary Predicate

- "Given a glvalue `u` of type (possibly `const`) T that designates the same object as `*first`, `pred(u)` shall be a valid expression that is equal to `pred(*first)`."
- `pred()` is a required to be a *regular* function.

Standard Requirement for Unary Predicate

- "Given a glvalue `u` of type (possibly `const`) T that designates the same object as `*first`, `pred(u)` shall be a valid expression that is equal to `pred(*first)`."
- `pred()` is a required to be a *regular* function.
- But Hyram's Law…

Unnecessary Preconditions

- *Unnecessary* precondition is a precondition in the contract, not imposed by the implementation

Unnecessary Preconditions

- *Unnecessary* precondition is a precondition in the contract, not imposed by the implementation
- No precondition is expressed as True$\{Q\}R$

Unnecessary Preconditions

- *Unnecessary* precondition is a precondition in the contract, not imposed by the implementation
- No precondition is expressed as True$\{Q\}R$
- Not stating precondition assumes no-precondition

Unnecessary Preconditions

- *Unnecessary* precondition is a precondition in the contract, not imposed by the implementation
- No precondition is expressed as True$\{Q\}R$
- Not stating precondition assumes no-precondition
  - Except for those imposed by convension

Unnecessary Preconditions

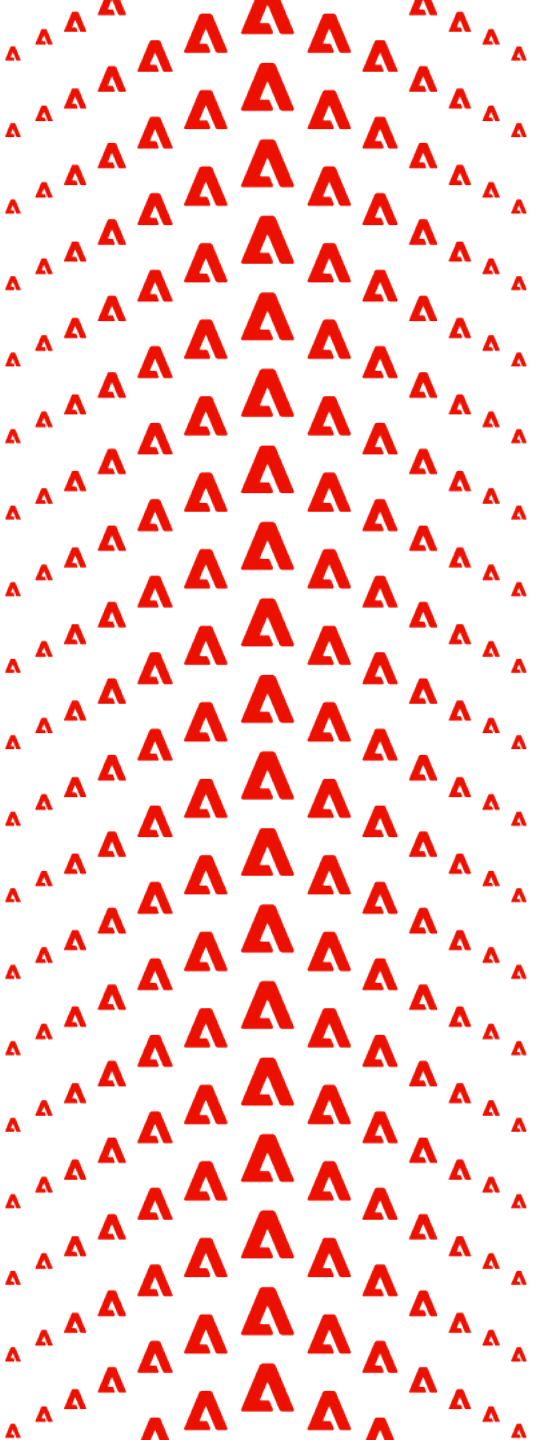
- *Unnecessary* precondition is a precondition in the contract, not imposed by the implementation
- No precondition is expressed as True$\{Q\}R$
- Not stating precondition assumes no-precondition
  - Except for those imposed by convension
- Are operations without preconditions or weakest preconditions “good”?

Unnecessary Preconditions

- *Unnecessary* precondition is a precondition in the contract, not imposed by the implementation
- No precondition is expressed as True$\{Q\}R$
- Not stating precondition assumes no-precondition
  - Except for those imposed by convension
- Are operations without preconditions or weakest preconditions “good”?
  - Are unnecessary preconditions “bad”?

“It’s complicated.”

– Kate Gregory



Signed vs Unsigned Integral Types

- Signed integral types in C++ have more preconditions than unsigned types

Signed vs Unsigned Integral Types

- Signed integral types in C++ have more preconditions than unsigned types
  - It is undefined behavior to overflow or underflow a signed type

Signed vs Unsigned Integral Types

- Signed integral types in C++ have more preconditions than unsigned types
  - It is undefined behavior to overflow or underflow a signed type
  - unsigned types are mod(2^b)

Signed vs Unsigned Integral Types

- Signed integral types in C++ have more preconditions than unsigned types
  - It is undefined behavior to overflow or underflow a signed type
  - unsigned types are mod(2^b)
- Unsigned type arithmetic still has preconditions

Signed vs Unsigned Integral Types

- Signed integral types in C++ have more preconditions than unsigned types
  - It is undefined behavior to overflow or underflow a signed type
  - unsigned types are mod(2^b)
- Unsigned type arithmetic still has preconditions

```
{
    unsigned a;
    unsigned b = a + 1u; // undefined behavior
}
```

Signed vs Unsigned Integral Types

- Signed integral types in C++ have more preconditions than unsigned types
  - It is undefined behavior to overflow or underflow a signed type
  - unsigned types are mod(2^b)
- Unsigned type arithmetic still has preconditions

```
{
    unsigned a;
    unsigned b = a + 1u; // undefined behavior
}
```

- A precondition of reading a variable is that it has been initialized

Bresenham Line Algorithm

```
template <class F>
void bresenham_line(int dx, int dy, F out) {
    assert((0 <= dy) && (dy <= dx) && (dx <= (INT_MAX - dy)));

    for (int x = 0, y = 0, a = dy / 2; x != dx; ++x) {
        out(x, y);
        a += dy;
        if (dx <= a) {
            ++y;
            a -= dx;
        }
    }
}
```

Bresenham Line Algorithm

```
bresenham_line(10, 6, [](auto, auto y) {
    cout << string(y, ' ') << "*\n";
});
```

Bresenham Line Algorithm

```
bresenham_line(10, 6, [](auto, auto y) {
    cout << string(y, ' ') << "*\n";
});
```

```
*
*
 *
  *
  *
   *
   *
    *
     *
      *
      *
```

Bresenham Line Algorithm

```
template <class F>
void bresenham_line(int dx, int dy, F out) {
    assert((0 <= dy) && (dy <= dx) && (dx <= (INT_MAX - dy)));

    for (int x = 0, y = 0, a = dy / 2; x != dx; ++x) {
        out(x, y);
        a += dy;
        if (dx <= a) {
            ++y;
            a -= dx;
        }
    }
}
```

Bresenham Line Algorithm

```
a += dy;
if (dx <= a) {

    a -= dx;
}
```

Bresenham Line Algorithm

```
a += dy;
if (dx <= a) {

    a -= dx;
}
```

$$a + dy \rightarrow a \pmod{dx}$$

Bresenham Line Algorithm

- Unsigned integer arithmetic in C++ is mod 2^n = mod $(max + 1)$

Bresenham Line Algorithm

- Unsigned integer arithmetic in C++ is mod 2^n = mod $(max + 1)$
- Project the slop to $dx' = max + 1$

Bresenham Line Algorithm

- Unsigned integer arithmetic in C++ is mod 2^n = mod $(max + 1)$
- Project the slop to $dx' = max + 1$

$$\frac{dy}{dx} = \frac{dy'}{(max + 1)}$$

$$dy' = \frac{(max + 1)dy}{dx}$$

Fast Bresenham Line Algorithm - exploiting unsigned

```
template <class F>
void fast_bresenham_line(unsigned dx, unsigned dy, F out) {
    assert(dy < dx);

    dy = (UINT_MAX + 1.0) * dy / dx;

    for (unsigned x = 0, y = 0, a = dy / 2; x != dx; ++x) {
        out(x, y);
        a += dy;      // add ebx, r12d
        y += a < dy; // addc r15d, 0
    }
}
```

Fast Bresenham Line Algorithm

```
fast_bresenham_line(10, 6, [](auto, auto y) {
    cout << string(y, ' ') << "*\n";
});
```

Fast Bresenham Line Algorithm

```
fast_bresenham_line(10, 6, [](auto, auto y) {
    cout << string(y, ' ') << "*\n";
});
```

```
*
*
 *
  *
  *
   *
   *
    *
     *
      *
      *
```

Unsigned vs Signed Integers

- It is easier to detect overflow with modular arithmetic

Unsigned vs Signed Integers

- It is easier to detect overflow with modular arithmetic
- Modular arithmetic properties can be exploited

Unsigned vs Signed Integers

- It is easier to detect overflow with modular arithmetic
- Modular arithmetic properties can be exploited

- Usually math is modeling a subset of $\mathbb{Z}$ or $\mathbb{N}$

Unsigned vs Signed Integers

- It is easier to detect overflow with modular arithmetic
- Modular arithmetic properties can be exploited

- Usually math is modeling a subset of $\mathbb{Z}$ or $\mathbb{N}$
  - Signed math provides more opportunities for analyzers to detect possible overflow

Unsigned vs Signed Integers

- It is easier to detect overflow with modular arithmetic
- Modular arithmetic properties can be exploited

- Usually math is modeling a subset of $\mathbb{Z}$ or $\mathbb{N}$
  - Signed math provides more opportunities for analyzers to detect possible overflow
  - Signed math provides more opportunities for compilers to optimize assuming no overflow

Undefined Behavior Can Catch Defects

```
template <class F>
void bresenham_line(int dx, int dy, F out) {
    for (int x = 0, y = 0, a = dy / 2; x != dx; ++x) {
        out(x, y);
        a += dy;
        if (!(a < dx)) {
            ++y;
            a -= dx;
        }
    }
}
```

⚠ Signed integer overflow: 1073741823 + 2147483647 cannot be represented in type 'int'

Unsigned vs Signed Integers

- Type attributes are a possible solution:

Unsigned vs Signed Integers

- Type attributes are a possible solution:

```
unsigned [[ limits(0, 59) ]] seconds{0};
```

Unnecessary Preconditions

- Provide flexibility of implementation

Unnecessary Preconditions

- Provide flexibility of implementation
- Can ascribe meaning and intent to an operation

Unnecessary Preconditions

- Provide flexibility of implementation
- Can ascribe meaning and intent to an operation
- Simplify requirements and reasoning about code

Unnecessary Preconditions

- Provide flexibility of implementation
- Can ascribe meaning and intent to an operation
- Simplify requirements and reasoning about code

- Limit clever uses that exploit defined behavior

Unnecessary Preconditions

- Provide flexibility of implementation
- Can ascribe meaning and intent to an operation
- Simplify requirements and reasoning about code

- Limit clever uses that exploit defined behavior
- Allow for variance in behavior between implementations

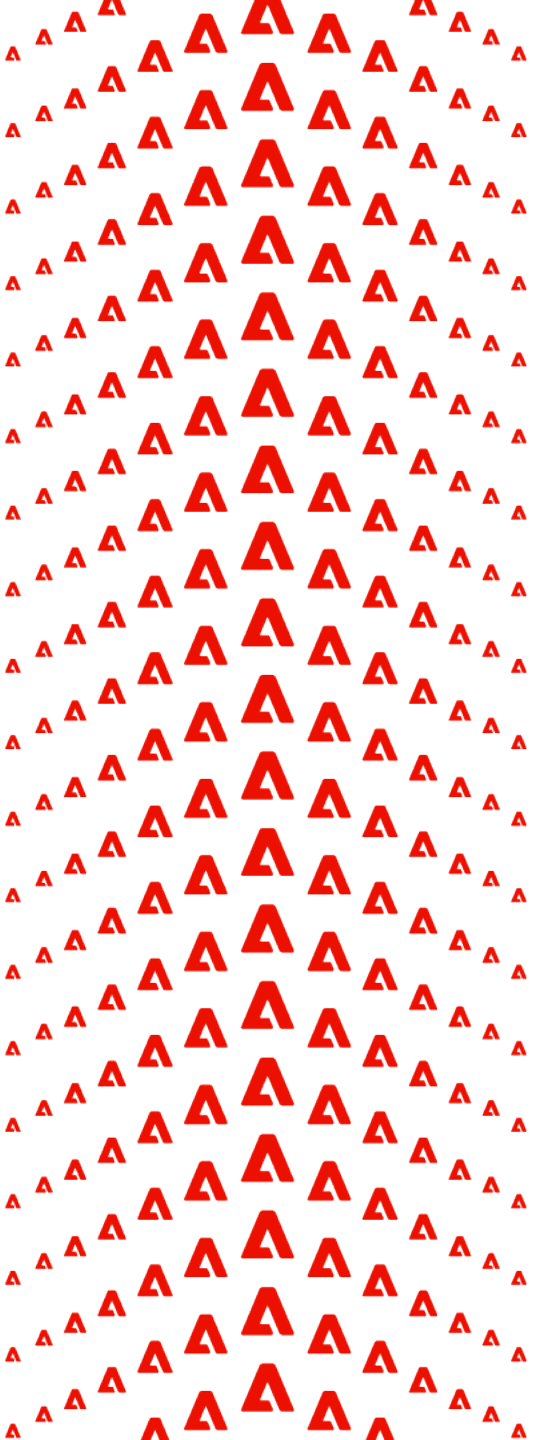
Unnecessary Preconditions

- Provide flexibility of implementation
- Can ascribe meaning and intent to an operation
- Simplify requirements and reasoning about code

- Limit clever uses that exploit defined behavior
- Allow for variance in behavior between implementations
- Open an opportunity for Hyram's law

“God created the natural numbers. All else is the work of man.”

– Leopold Kronecker



Generic Programming*

David R. Musser†
Rensselaer Polytechnic Institute
Computer Science Department
Amos Eaton Hall
Troy, New York 12180

Alexander A. Stepanov
Hewlett–Packard Laboratories
Software Technology Laboratory
Post Office Box 10490
Palo Alto, California 94303–0969

Abstract

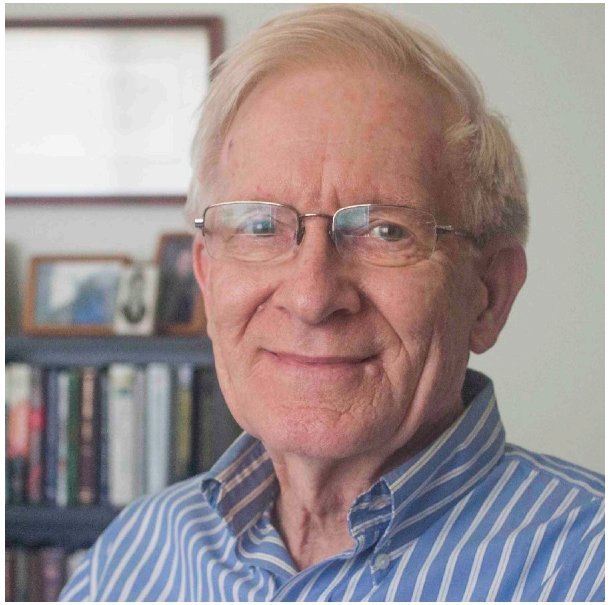
Generic programming centers around the idea of abstracting from concrete, efficient algorithms to obtain generic algorithms that can be combined with different data representations to produce a wide variety of useful software. For example, a class of generic sorting algorithms can be defined which work with finite sequences but which can be instantiated in different ways to produce algorithms working on arrays or linked lists.

Four kinds of abstraction—data, algorithmic, structural, and representational—are discussed, with examples of their use in building an Ada library of software components. The main topic discussed is generic algorithms and an approach to their formal specification and verification, with illustration in terms of a partitioning algorithm such as is used in the quicksort algorithm. It is argued that generically programmed software component libraries offer important advantages for achieving software productivity and reliability.

*This paper was presented at the First International Joint Conference of ISSAC-88 and AAECC-6, Rome, Italy, July 4-8, 1988. (ISSAC stands for International Symposium on Symbolic and Algebraic Computation and AAECC for Applied Algebra, Algebraic Algorithms, and Error Correcting Codes). It was published in *Lecture Notes in Computer Science* 358, Springer-Verlag, 1989, pp. 13-25.

†The first author's work was sponsored in part through a subcontract from Computational Logic, Inc., which was sponsored in turn by the Defense Advanced Research Projects Agency, ARPA order 9151. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, the U.S. Government, or Computational Logic., Inc.

1989



Generic Programming*

David R. Musser†
Rensselaer Polytechnic Institute
Computer Science Department
Amos Eaton Hall
Troy, New York 12180

Alexander A. Stepanov
Hewlett–Packard Laboratories
Software Technology Laboratory
Post Office Box 10490
Palo Alto, California 94303–0969

Abstract

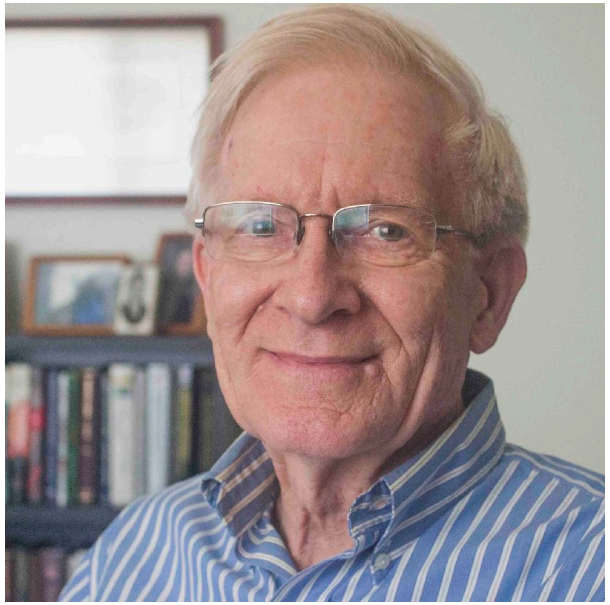
Generic programming centers around the idea of abstracting from concrete, efficient algorithms to obtain generic algorithms that can be combined with different data representations to produce a wide variety of useful software. For example, a class of generic sorting algorithms can be defined which work with finite sequences but which can be instantiated in different ways to produce algorithms working on arrays or linked lists.

Four kinds of abstraction—data, algorithmic, structural, and representational—are discussed, with examples of their use in building an Ada library of software components. The main topic discussed is generic algorithms and an approach to their formal specification and verification, with illustration in terms of a partitioning algorithm such as is used in the quicksort algorithm. It is argued that generically programmed software component libraries offer important advantages for achieving software productivity and reliability.

*This paper was presented at the First International Joint Conference of ISSAC-88 and AAECC-6, Rome, Italy, July 4-8, 1988. (ISSAC stands for International Symposium on Symbolic and Algebraic Computation and AAECC for Applied Algebra, Algebraic Algorithms, and Error Correcting Codes). It was published in *Lecture Notes in Computer Science* 358, Springer-Verlag, 1989, pp. 13-25.

†The first author's work was sponsored in part through a subcontract from Computational Logic, Inc., which was sponsored in turn by the Defense Advanced Research Projects Agency, ARPA order 9151. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, the U.S. Government, or Computational Logic., Inc.

1989



Generic Programming*

David R. Musser†
Rensselaer Polytechnic Institute
Computer Science Department
Amos Eaton Hall
Troy, New York 12180

Alexander A. Stepanov
Hewlett–Packard Laboratories
Software Technology Laboratory
Post Office Box 10490
Palo Alto, California 94303–0969

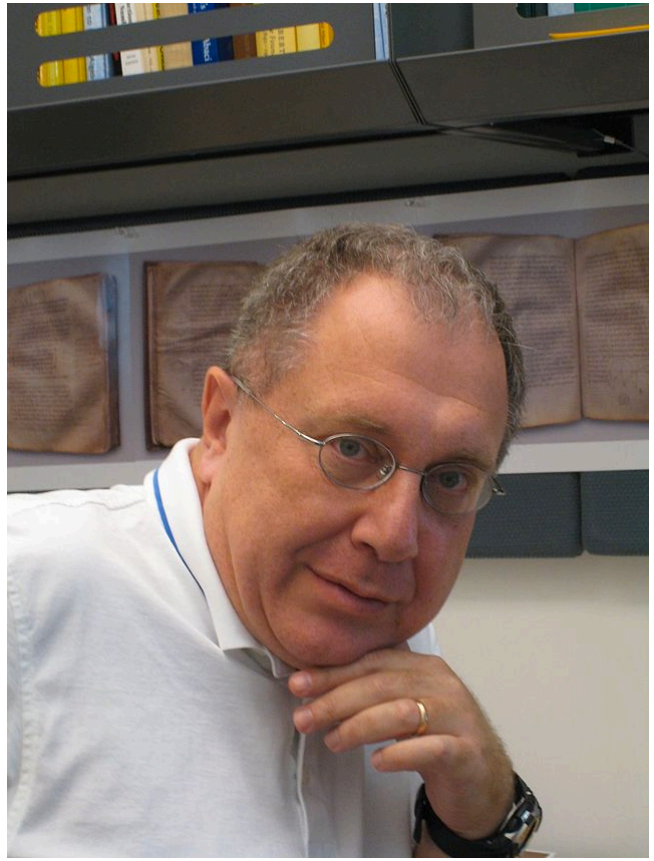
Abstract

Generic programming centers around the idea of abstracting from concrete, efficient algorithms to obtain generic algorithms that can be combined with different data representations to produce a wide variety of useful software. For example, a class of generic sorting algorithms can be defined which work with finite sequences but which can be instantiated in different ways to produce algorithms working on arrays or linked lists.

Four kinds of abstraction—data, algorithmic, structural, and representational—are discussed, with examples of their use in building an Ada library of software components. The main topic discussed is generic algorithms and an approach to their formal specification and verification, with illustration in terms of a partitioning algorithm such as is used in the quicksort algorithm. It is argued that generically programmed software component libraries offer important advantages for achieving software productivity and reliability.

*This paper was presented at the First International Joint Conference of ISSAC-88 and AAECC-6, Rome, Italy, July 4-8, 1988. (ISSAC stands for International Symposium on Symbolic and Algebraic Computation and AAECC for Applied Algebra, Algebraic Algorithms, and Error Correcting Codes). It was published in *Lecture Notes in Computer Science* 358, Springer-Verlag, 1989, pp. 13-25.

†The first author's work was sponsored in part through a subcontract from Computational Logic, Inc., which was sponsored in turn by the Defense Advanced Research Projects Agency, ARPA order 9151. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, the U.S. Government, or Computational Logic., Inc.



1989

Fundamentals of Generic Programming

James C. Dehnert and Alexander Stepanov

Silicon Graphics, Inc.
dehnertj@acm.org, stepanov@attlabs.att.com

Keywords: Generic programming, operator semantics, concept, regular type.

Abstract. Generic programming depends on the decomposition of programs into components which may be developed separately and combined arbitrarily, subject only to well-defined interfaces. Among the interfaces of interest, indeed the most pervasively and unconsciously used, are the fundamental operators common to all C++ built-in types, as extended to user-defined types, e.g. copy constructors, assignment, and equality. We investigate the relations which must hold among these operators to preserve consistency with their semantics for the built-in types and with the expectations of programmers. We can produce an axiomatization of these operators which yields the required consistency with built-in types, matches the intuitive expectations of programmers, and also reflects our underlying mathematical expectations.

1992

Fundamentals of Generic Programming

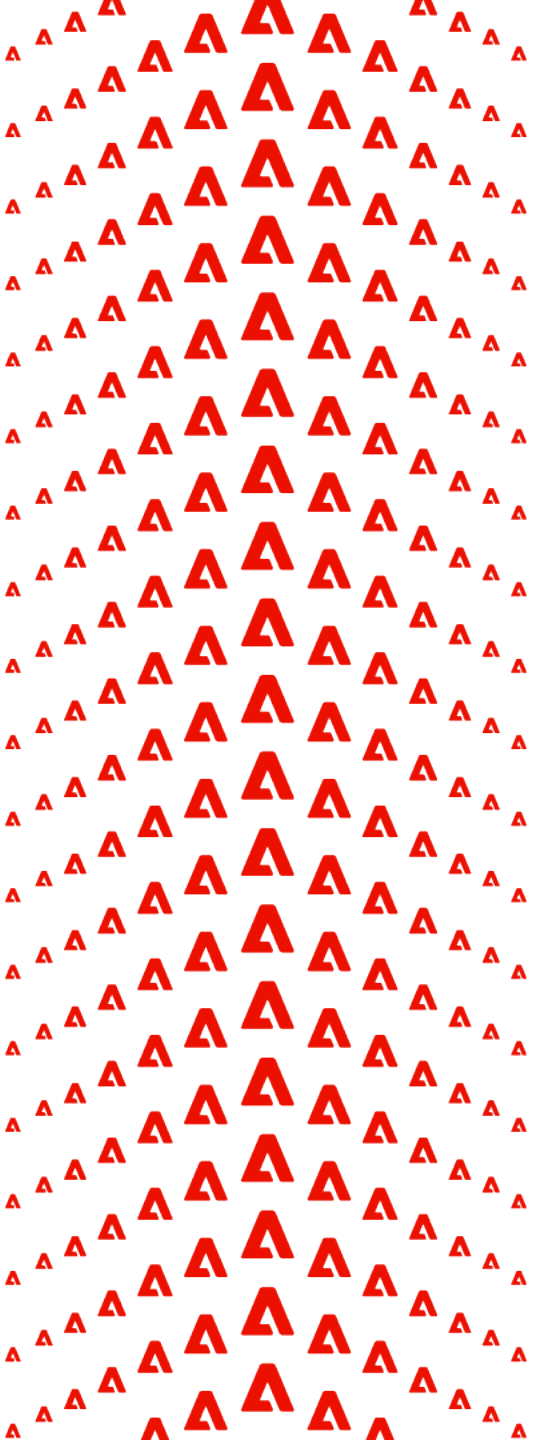
James C. Dehnert and Alexander Stepanov

Silicon Graphics, Inc.
dehnertj@acm.org, stepanov@attlabs.att.com

1992

"We call the set of axioms satisfied by a data type and a set of operations on it a concept."

– Fundamentals of Generic Programming



Concepts

- Associate semantics & complexity with syntax

Concepts

- Associate semantics & complexity with syntax
- Defines a component that will work for any type satisfying the *requirements*

Concepts

- Associate semantics & complexity with syntax
- Defines a component that will work for any type satisfying the *requirements*
- Assign meaning to an unbounded set of operations

Concepts

- Associate semantics & complexity with syntax
- Defines a component that will work for any type satisfying the *requirements*
- Assign meaning to an unbounded set of operations
- An argument is *required* to satisfy a concept

Concepts

- Associate semantics & complexity with syntax
- Defines a component that will work for any type satisfying the *requirements*
- Assign meaning to an unbounded set of operations
- An argument is *required* to satisfy a concept
- A data type or operation may *guarantee* it is able to satisfy a concept

Requirements vs Guarantees

- A *requirement* applies to the arguments of a type or operation consisting of required:

Requirements vs Guarantees

- A *requirement* applies to the arguments of a type or operation consisting of required:
  - Valid expressions

Requirements vs Guarantees

- A *requirement* applies to the arguments of a type or operation consisting of required:
  - Valid expressions
  - Preconditions

Requirements vs Guarantees

- A *requirement* applies to the arguments of a type or operation consisting of required:
  - Valid expressions
  - Preconditions
  - Semantics

Requirements vs Guarantees

- A *requirement* applies to the arguments of a type or operation consisting of required:
  - Valid expressions
  - Preconditions
  - Semantics
- A *guarantee* applies to an instance of an object:

Requirements vs Guarantees

- A *requirement* applies to the arguments of a type or operation consisting of required:
  - Valid expressions
  - Preconditions
  - Semantics
- A *guarantee* applies to an instance of an object:
  - Asserting such an instance satisfies a requirement (or models a concept)

Requirements

- `distance(f, l)` requires:

Requirements

- `distance(f, l)` requires:
  - `f` and `l` satisfy *InputIterators*

Requirements

- `distance(f, l)` requires:
  - `f` and `l` satisfy *InputIterators*
    - preincrement, `++i`, postincrement, `(void)i++`, and postincrement and dereference, `*i++`

Requirements

- `distance(f, l)` requires:
  - `f` and `l` satisfy *InputIterators*
    - preincrement, `++i`, postincrement, `(void)i++`, and postincrement and dereference, `*i++`
      - precondition: `i != l`

Requirements

- `distance(f, l)` requires:
  - `f` and `l` satisfy *InputIterators*
    - preincrement, `++i`, postincrement, `(void)i++`, and postincrement and dereference, `*i++`
      - precondition: `i != l`
    - `f` and `l` satisfy *TrivialIterator*

Requirements

- `distance(f, l)` requires:
  - `f` and `l` satisfy *InputIterators*
    - preincrement, `++i`, postincrement, `(void)i++`, and postincrement and dereference, `*i++`
      - precondition: `i != l`
    - `f` and `l` satisfy *TrivialIterator*
      - `f` and `l` satisfy *Assignable*, *EqualityComparable*, *DefaultConstructible*

Requirements

- `distance(f, l)` requires:
  - `f` and `l` satisfy *InputIterators*
    - preincrement, `++i`, postincrement, `(void)i++`, and postincrement and dereference, `*i++`
      - precondition: `i != l`
    - `f` and `l` satisfy *TrivialIterator*
      - `f` and `l` satisfy *Assignable*, *EqualityComparable*, *DefaultConstructible*
        - *EqualityComparable* precondition: arguments are in the domain of ==

Requirements

- `distance(f, l)` requires:
  - `f` and `l` satisfy *InputIterators*
    - preincrement, `++i`, postincrement, `(void)i++`, and postincrement and dereference, `*i++`
      - precondition: `i != l`
    - `f` and `l` satisfy *TrivialIterator*
      - `f` and `l` satisfy *Assignable*, *EqualityComparable*, *DefaultConstructible*
        - *EqualityComparable* precondition: arguments are in the domain of ==
  - precondition: `[f, l)` is a valid range

Requirements

- Naming the set of requirements is a significant simplification

Requirements

- Naming the set of requirements is a significant simplification
- The concept `std::input_iterator` encapsulates a complex set of syntactic and **semantic** requirements

Requirements

- Naming the set of requirements is a significant simplification
- The concept `std::input_iterator` encapsulates a complex set of syntactic and **semantic** requirements
- Only the syntactic requirements are enforced by the compiler but analyzers and sanitizers can validate some of the semantic requirements

Requirements

- Naming the set of requirements is a significant simplification
- The concept `std::input_iterator` encapsulates a complex set of syntactic and **semantic** requirements
- Only the syntactic requirements are enforced by the compiler but analyzers and sanitizers can validate some of the semantic requirements

- Concepts in the standard are requirements of the arguments to the library components

Requirements

- Naming the set of requirements is a significant simplification
- The concept `std::input_iterator` encapsulates a complex set of syntactic and **semantic** requirements
- Only the syntactic requirements are enforced by the compiler but analyzers and sanitizers can validate some of the semantic requirements

- Concepts in the standard are requirements of the arguments to the library components
  - Not requirements of the implementation of the standard types

Requirements

- Naming the set of requirements is a significant simplification
- The concept `std::input_iterator` encapsulates a complex set of syntactic and **semantic** requirements
- Only the syntactic requirements are enforced by the compiler but analyzers and sanitizers can validate some of the semantic requirements

- Concepts in the standard are requirements of the arguments to the library components
  - Not requirements of the implementation of the standard types
  - The standard types are often described as guaranteeing they satisfying those requirements

Concepts

- Named requirements, or concepts, are distilled from:

Concepts

- Named requirements, or concepts, are distilled from:
  - A set of related components (algorithms or containers)

Concepts

- Named requirements, or concepts, are distilled from:
  - A set of related components (algorithms or containers)
  - A set of common models

Concepts

- Named requirements, or concepts, are distilled from:
  - A set of related components (algorithms or containers)
  - A set of common models

- They create a simple way to match data types to components and know the result will work correctly

Concepts

- Named requirements, or concepts, are distilled from:
  - A set of related components (algorithms or containers)
  - A set of common models

- They create a simple way to match data types to components and know the result will work correctly
  - But for any specific component, some of the requirements may be *unnecessary*

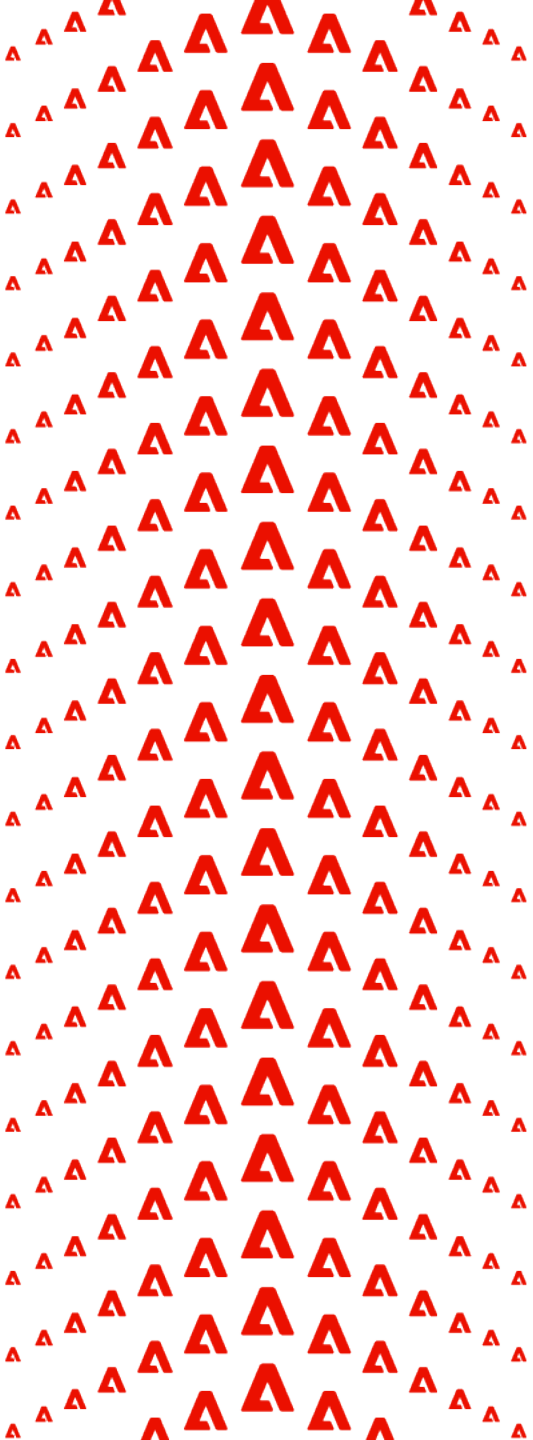
Concepts

- Named requirements, or concepts, are distilled from:
  - A set of related components (algorithms or containers)
  - A set of common models

- They create a simple way to match data types to components and know the result will work correctly
  - But for any specific component, some of the requirements may be *unnecessary*

- The purpose is not to specify the implementation but to specify the *meaning*

std::find(first, last, value)



Meaning of Equality

- Two objects are equal iff they represent the same entity (i.e., have the same value)

Meaning of Equality

- Two objects are equal iff they represent the same entity (i.e., have the same value)
- *Equality* is an equivalence relation

Meaning of Equality

- Two objects are equal iff they represent the same entity (i.e., have the same value)
- *Equality* is an equivalence relation

$$\forall a \qquad a = a \qquad \text{(reflexive)}$$

$$\forall a, b \qquad a = b \iff b = a \qquad \text{(symmetric)}$$

$$\forall a, b, c \qquad (a = b \land b = c) \implies a = c \qquad \text{(transitive)}$$

Meaning of Equality

- Two objects are equal iff they represent the same entity (i.e., have the same value)
- *Equality* is an equivalence relation

$$\forall a \qquad a = a \qquad \text{(reflexive)}$$

$$\forall a, b \qquad a = b \iff b = a \qquad \text{(symmetric)}$$

$$\forall a, b, c \qquad (a = b \wedge b = c) \implies a = c \qquad \text{(transitive)}$$

- Consistent with other operations on the type

Meaning of Equality

- Two objects are equal iff they represent the same entity (i.e., have the same value)
- *Equality* is an equivalence relation

$$\forall a \qquad a = a \qquad \text{(reflexive)}$$

$$\forall a, b \qquad a = b \iff b = a \qquad \text{(symmetric)}$$

$$\forall a, b, c \qquad (a = b \land b = c) \implies a = c \qquad \text{(transitive)}$$

- Consistent with other operations on the type

$$\forall a, b \qquad b \to a \implies a = b \qquad \text{(equivalence of copies)}$$

$$\forall a, b \qquad a \not< b \land b \not< a \iff a = b \qquad \text{(excluded middle)}$$

SGI STL std::find() documentation

```
template<class InputIterator, class EqualityComparable>
InputIterator find(InputIterator first, InputIterator last,
                   const EqualityComparable& value);
```

Requirements on types

- EqualityComparable is a model of EqualityComparable.
- InputIterator is a model of InputIterator.
- Equality is defined between objects of type EqualityComparable and objects of InputIterator's value type.

Preconditions

- `[first, last)` is a valid range.

Complexity

Linear: at most `last - first` comparisons for equality.

SGI STL EqualityComparable documentation

Expression semantics

Name	Expression	Precondition
Equality	`x == y`	`x` and `y` are in the domain of `==`

Invariants

Identity	`&x == &y` implies `x == y`
Reflexivity	`x == x`
Symmetry	`x == y` implies `y == x`
Transitivity	`x == y` and `y == z` implies `x == z`

C++20 std::find() specification (25.6.5)

```
template<class InputIterator, class T>
  constexpr InputIterator find(InputIterator first, InputIterator last,
                               const T& value);
```

Let *E* be:

- `*i == value` for `find`;

Returns: The first iterator `i` in the range [`first, last`) for which *E* is `true`. Returns `last` if no such iterator is found.

C++20 Cpp17EqualityComparable requirements

Table 27: *Cpp17EqualityComparable* requirements [tab:cpp17.equalitycomparable]

Expression	Return type	Requirement
`a == b`	convertible to `bool`	== is an equivalence relation, that is, it has the following properties: — For all a, a == a. — If a == b, then b == a. — If a == b and b == c, then a == c.

NaN refresher - a value which is not *equality comparable*

- `nan("")` is typically generated by 0.0/0.0

NaN refresher - a value which is not *equality comparable*

- `nan("")` is typically generated by `0.0/0.0`
- `nan("") == nan("")` is `false` (irreflexive)

NaN refresher - a value which is not *equality comparable*

- `nan("")` is typically generated by `0.0/0.0`
- `nan("") == nan("")` is `false` (irreflexive)
- `nan("")` does not satisfy the requirements of EqualityComparable or Cpp17EqualityComparable

Find Without Equality

```
double a[]{ 0.8, 7.0, nan(""), 3.0, 2.4 };
```

Find Without Equality

```
double a[]{ 0.8, 7.0, nan(""), 3.0, 2.4 };

auto p = find(begin(a), end(a), nan(""));
```

Find Without Equality

```
double a[]{ 0.8, 7.0, nan(""), 3.0, 2.4 };

auto p = find(begin(a), end(a), nan(""));

if (p == end(a)) {
    cout << "not-found\n";
} else {
    cout << "found: " << *p << "\n";
}
```

Find Without Equality

```
double a[]{ 0.8, 7.0, nan(""), 3.0, 2.4 };

auto p = find(begin(a), end(a), nan(""));

if (p == end(a)) {
    cout << "not-found\n";
} else {
    cout << "found: " << *p << "\n";
}
```

not-found

Find Without Equality

```
double a[]{ 0.8, 7.0, nan(""), 3.0, 2.4 };
```

Find Without Equality

```
double a[]{ 0.8, 7.0, nan(""), 3.0, 2.4 };

auto p = find(begin(a), end(a), 3.0);
```

Find Without Equality

```
double a[]{ 0.8, 7.0, nan(""), 3.0, 2.4 };

auto p = find(begin(a), end(a), 3.0);

if (p == end(a)) {
    cout << "not-found\n";
} else {
    cout << "found: " << *p << "\n";
}
```

Find Without Equality

```
double a[]{ 0.8, 7.0, nan(""), 3.0, 2.4 };

auto p = find(begin(a), end(a), 3.0);

if (p == end(a)) {
    cout << "not-found\n";
} else {
    cout << "found: " << *p << "\n";
}
```

found: 3

Problem Statement

Given a sequence of fractions $\frac{a_0}{b_0}, \frac{a_1}{b_1}, \ldots \frac{a_n}{b_n}$, find the first fraction such that $a_p = xb_p$.

Problem Statement

Given a sequence of fractions $\frac{a_0}{b_0}, \frac{a_1}{b_1}, \ldots \frac{a_n}{b_n}$, find the first fraction such that $a_p = xb_p$.

```
double a[]{4.0/5.0, 7.0/1.0, 0.0/0.0, 9.0/3.0, 12.0/5.0};
```

Problem Statement

Given a sequence of fractions $\frac{a_0}{b_0}, \frac{a_1}{b_1}, \ldots \frac{a_n}{b_n}$, find the first fraction such that $a_p = xb_p$.

```
double a[]{4.0/5.0, 7.0/1.0, 0.0/0.0, 9.0/3.0, 12.0/5.0};

double x = 3.0;
auto p = find(begin(a), end(a), x);
```

A Subtle Change…

- If `std::find()` required Cpp17EqualityComparable, the following code would be undefined behavior:

A Subtle Change…

- If `std::find()` required Cpp17EqualityComparable, the following code would be undefined behavior:

```
double a[] { 0.8, 7.0, 42.3, 3.0, 2.4 };

auto p = find(begin(a), end(a), 3.0);
```

A Subtle Change…

- If `std::find()` required Cpp17EqualityComparable, the following code would be undefined behavior:

```
double a[] { 0.8, 7.0, 42.3, 3.0, 2.4 };

auto p = find(begin(a), end(a), 3.0);
```

- The above code is well defined with the SGI definition of EqualityComparable

SGI STL EqualityComparable documentation

Expression semantics

Name	Expression	Precondition
Equality	`x == y`	`x` and `y` are in the domain of `==`

Invariants

Identity	`&x == &y` implies `x == y`
Reflexivity	`x == x`
Symmetry	`x == y` implies `y == x`
Transitivity	`x == y` and `y == z` implies `x == z`

SGI STL EqualityComparable documentation

Precondition
`x` and `y` are in the domain of `==`

C++20 Cpp17EqualityComparable requirements

Table 27: *Cpp17EqualityComparable* requirements [tab:cpp17.equalitycomparable]

Expression	Return type	Requirement
`a == b`	convertible to `bool`	== is an equivalence relation, that is, it has the following properties: — For all a, a == a. — If a == b, then b == a. — If a == b and b == c, then a == c.

The term *domain of the operation* is used in the ordinary mathematical sense to denote the set of values over which an operation is (required to be) defined. This set can change over time. Each component may place additional requirements on the domain of an operation. These requirements can be inferred from the uses that a component makes of the operation and are generally constrained to those values accessible through the operation's arguments.

Domain of the Operation

- The domain of an operation is *not* the types of the arguments

Domain of the Operation

- The domain of an operation is *not* the types of the arguments
- For a type, T, to satisfy a requirement, P:

Domain of the Operation

- The domain of an operation is *not* the types of the arguments
- For a type, T, to satisfy a requirement, P:

$$\forall a_0 \ldots a_n \; P(a_0 \ldots a_n)$$

Domain of the Operation

- The domain of an operation is *not* the types of the arguments
- For a type, T, to satisfy a requirement, P:

$$\forall a_0 \ldots a_n \; P(a_0 \ldots a_n)$$

- T must guarantee that

Domain of the Operation

- The domain of an operation is *not* the types of the arguments
- For a type, T, to satisfy a requirement, P:

$$\forall a_0 \ldots a_n \; P(a_0 \ldots a_n)$$

- T must guarantee that

$$\exists a_0 \ldots a_n \in T \ni P(a_0 \ldots a_n)$$

Domain of the Operation

- The domain of an operation is *not* the types of the arguments
- For a type, T, to satisfy a requirement, P:

$$\forall a_0 \ldots a_n \; P(a_0 \ldots a_n)$$

- T must guarantee that

$$\exists a_0 \ldots a_n \in T \ni P(a_0 \ldots a_n)$$

- `double` and `float` satisfy EqualityComparable

Domain of the Operation

- The domain of an operation is *not* the types of the arguments
- For a type, T, to satisfy a requirement, P:

$$\forall a_0 \ldots a_n \; P(a_0 \ldots a_n)$$

- T must guarantee that

$$\exists a_0 \ldots a_n \in T \ni P(a_0 \ldots a_n)$$

- `double` and `float` satisfy EqualityComparable
  - So long as `nan` is not in the set being compared

Domain of the Operation

- The domain of an operation is *not* the types of the arguments
- For a type, T, to satisfy a requirement, P:

$$\forall a_0 \ldots a_n \; P(a_0 \ldots a_n)$$

- T must guarantee that

$$\exists a_0 \ldots a_n \in T \ni P(a_0 \ldots a_n)$$

- `double` and `float` satisfy EqualityComparable
  - So long as `nan` is not in the set being compared
  - The absence of `nan` in the sequence for `find()` is a precondition

std::find() is broken in C++20

- `std::find()` doesn't require that there exist any equality comparable values in `T`

std::find() is broken in C++20

- `std::find()` doesn't require that there exist any equality comparable values in `T`
- `std::find()` doesn't guarantee that it finds `value`, even if `value` exists in the sequence

std::find() is broken in C++20

- `std::find()` doesn't require that there exist any equality comparable values in `T`
- `std::find()` doesn't guarantee that it finds `value`, even if `value` exists in the sequence
- The meaning of `std::find()` is reduced to *works-as-implemented*

std::find() is broken in C++20

- `std::find()` doesn't require that there exist any equality comparable values in `T`
- `std::find()` doesn't guarantee that it finds `value`, even if `value` exists in the sequence
- The meaning of `std::find()` is reduced to *works-as-implemented*
- Fortunately, it is trivial to show that iff `operator==()` models *EqualityComparable*

std::find() is broken in C++20

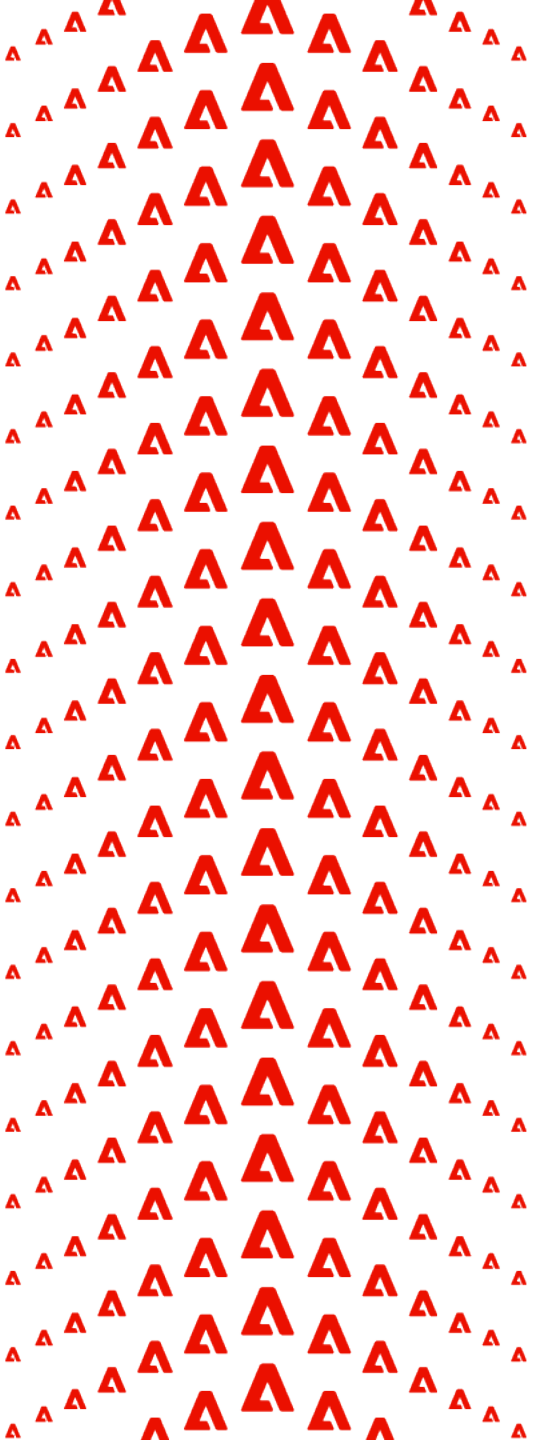
- `std::find()` doesn't require that there exist any equality comparable values in `T`
- `std::find()` doesn't guarantee that it finds `value`, even if `value` exists in the sequence
- The meaning of `std::find()` is reduced to *works-as-implemented*
- Fortunately, it is trivial to show that iff `operator==()` models *EqualityComparable*
  - And all values in the sequence and the value being sought are in the *domain of* ==

std::find() is broken in C++20

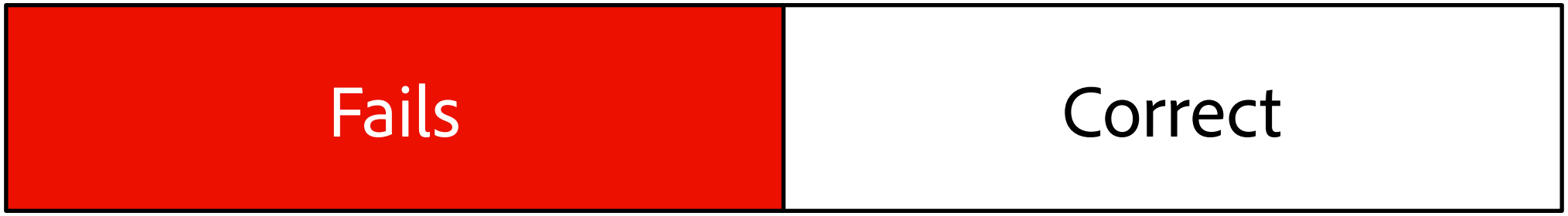
- `std::find()` doesn't require that there exist any equality comparable values in `T`
- `std::find()` doesn't guarantee that it finds `value`, even if `value` exists in the sequence
- The meaning of `std::find()` is reduced to *works-as-implemented*
- Fortunately, it is trivial to show that iff `operator==()` models *EqualityComparable*
  - And all values in the sequence and the value being sought are in the *domain of* ==
  - Then `std::find()` will *find*

“Understanding why software fails is important, but the real challenge is understanding why software works.”

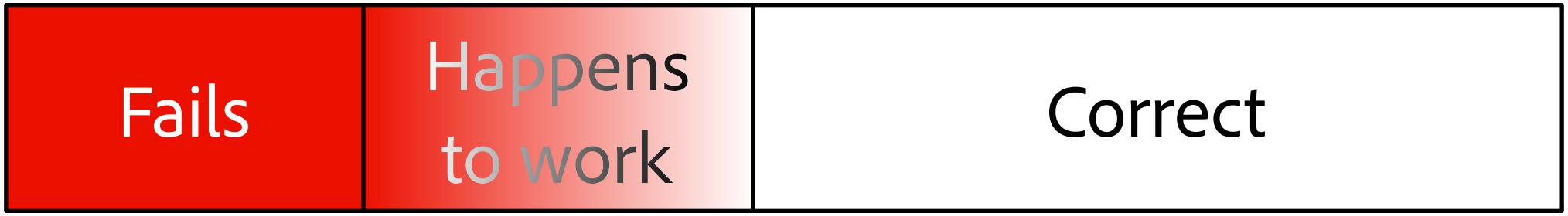
– Alexander Stepanov



Weakening Requirements

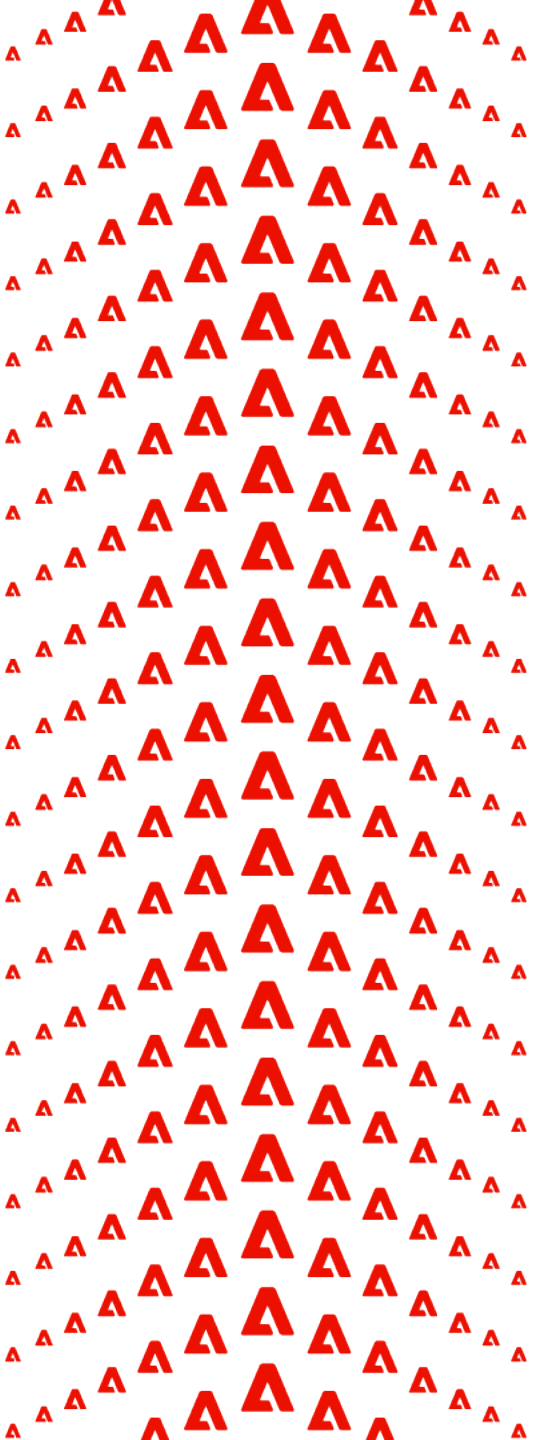


Weakening Requirements



"I work with very good programmers and I see a ton of happens-to-work and very little actually correct."

– Titus Winters



What You Can Do

- After trial-&-error, Stack Overflow, and Googling maybe you have code that happens-to-work

What You Can Do

- After trial-&-error, Stack Overflow, and Googling maybe you have code that happens-to-work
- Take the time to read the specification

What You Can Do

- After trial-&-error, Stack Overflow, and Googling maybe you have code that happens-to-work
- Take the time to read the specification
  - Lookup anything you aren't clear about

What You Can Do

- After trial-&-error, Stack Overflow, and Googling maybe you have code that happens-to-work
- Take the time to read the specification
  - Lookup anything you aren't clear about
  - Ask specific questions of experts

What You Can Do

- After trial-&-error, Stack Overflow, and Googling maybe you have code that happens-to-work
- Take the time to read the specification
  - Lookup anything you aren't clear about
  - Ask specific questions of experts
- Clever uses require additional validation

What You Can Do

- After trial-&-error, Stack Overflow, and Googling maybe you have code that happens-to-work
- Take the time to read the specification
  - Lookup anything you aren't clear about
  - Ask specific questions of experts
- Clever uses require additional validation
- Think about the meaning, the semantics, of your code

What You Can Do

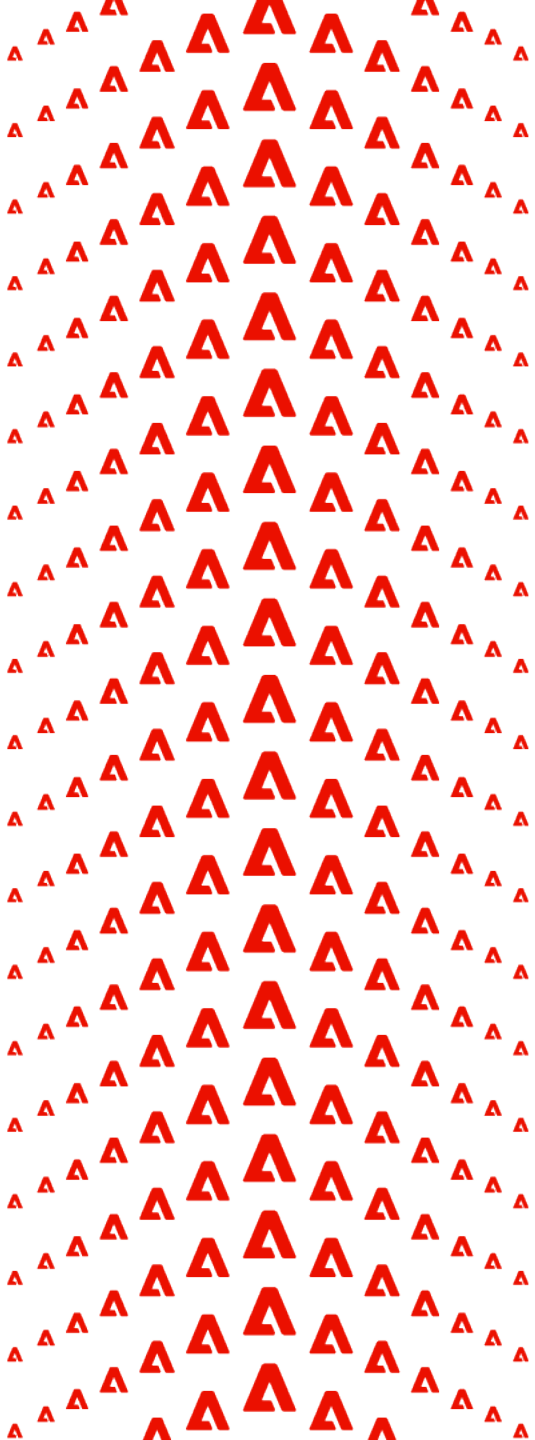
- After trial-&-error, Stack Overflow, and Googling maybe you have code that happens-to-work
- Take the time to read the specification
  - Lookup anything you aren't clear about
  - Ask specific questions of experts
- Clever uses require additional validation
- Think about the meaning, the semantics, of your code
  - Ensure your use reflects the implied semantics

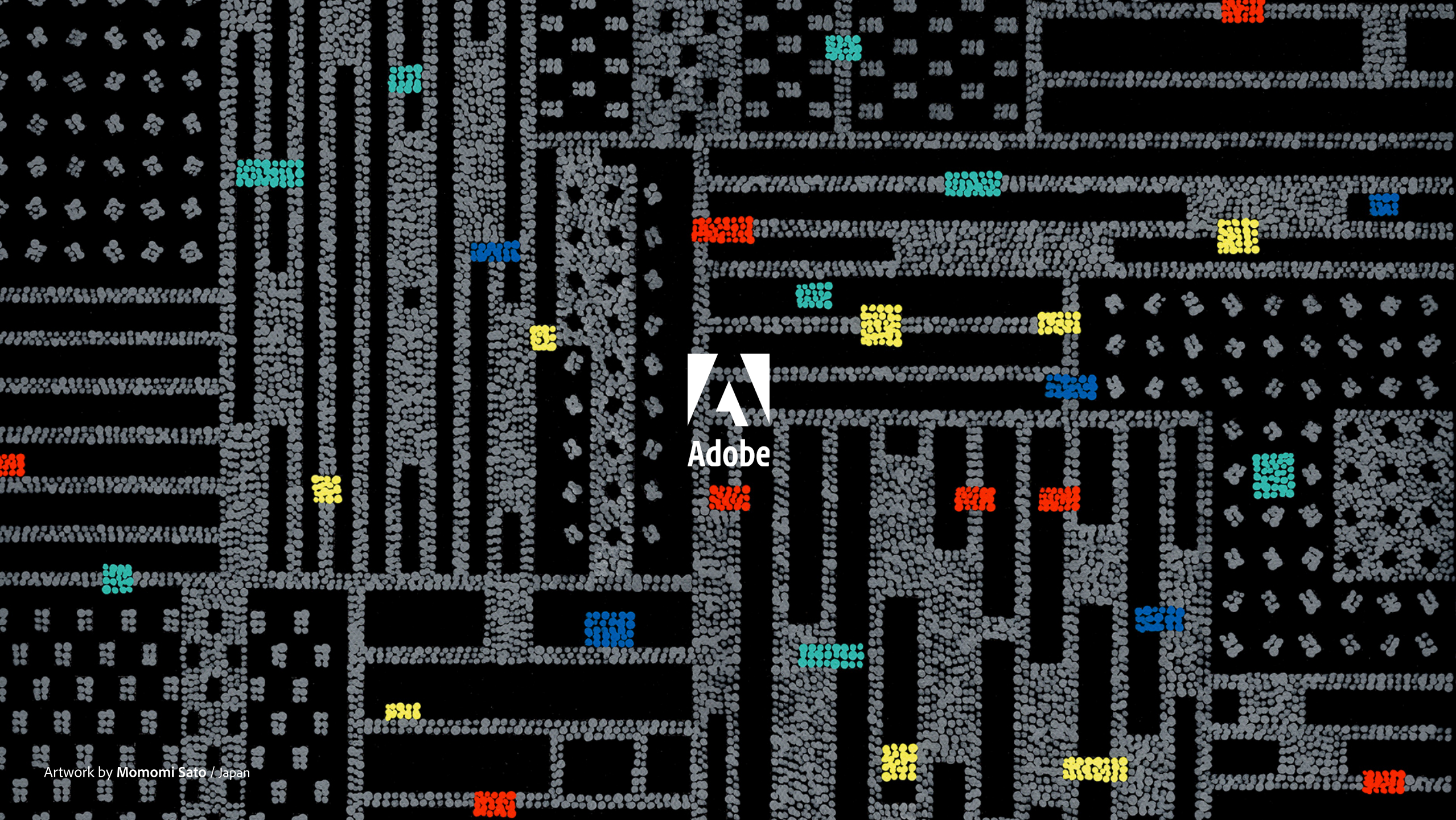
What You Can Do

- After trial-&-error, Stack Overflow, and Googling maybe you have code that happens-to-work
- Take the time to read the specification
  - Lookup anything you aren't clear about
  - Ask specific questions of experts
- Clever uses require additional validation
- Think about the meaning, the semantics, of your code
  - Ensure your use reflects the implied semantics
  - Ensure your names reflect the semantics of what they represent

“The gap between code that fails and code that is correct is vast. Within it lies all the code that happens-to-work. Strive to write correct code and you will write better code.”

– Me, This Talk




Adobe
Artwork by Momomi Sato / Japan

About the artist

Momomi Sato

Tokyo-based artist Momomi Sato meticulously applies paint using toothpicks to create fanciful, pointillistic works of animals, patterns, and other colorful subjects. With a style that ranges from abstract to kawaii, Sato's paintings are as charming as they are beautiful. For this piece, a train ride prompted an exploration of systems that influence daily life. As she stared intently at the pattern on the seats, the lines and shapes seemed to move and draw Sato into another dimension. She recreated the sensation by hand with acrylic paint on canvas.