Building Correct Programs

The Art of Computer Programming

Pierre Senellart

PSL UNIVERSITÉ PARIS

15 December 2025

Outline

Building Correct Programs

- Writing a program that runs is not enough
- Writing a program that seems to work is not enough
- We want programs that are:
  - correct
  - understandable
  - maintainable
- Correctness is a disciplined process

Sources of Bugs

- Ambiguous or incorrect understanding of the problem
- Implicit assumptions
- Boundary cases
- Incorrect reasoning about loops or data structures
- Incorrect interaction between components

The Four Pillars of Correctness

- Specifications: what the program should do
- Reasoning: why the program does it
- Testing: checking concrete behaviors
- Debugging: finding and fixing violations

Outline

From Problem to Program

- Writing a correct program is a multi-step process
- Each step answers a different question

From Problem to Program

- Writing a correct program is a multi-step process
- Each step answers a different question

Problem statement	What needs to be solved? (informal, in natural language)

From Problem to Program

- Writing a correct program is a multi-step process
- Each step answers a different question

Problem statement	What needs to be solved? (informal, in natural language)
Specification	What the solution must do, independently of how

From Problem to Program

- Writing a correct program is a multi-step process
- Each step answers a different question

Problem statement	What needs to be solved? (informal, in natural language)
Specification	What the solution must do, independently of how
Algorithm	A high-level method to meet the specification

From Problem to Program

- Writing a correct program is a multi-step process
- Each step answers a different question

Problem statement	What needs to be solved? (informal, in natural language)
Specification	What the solution must do, independently of how
Algorithm	A high-level method to meet the specification
Pseudo-code	A precise, language-agnostic description of the algorithm

From Problem to Program

- Writing a correct program is a multi-step process
- Each step answers a different question

Problem statement	What needs to be solved? (informal, in natural language)
Specification	What the solution must do, independently of how
Algorithm	A high-level method to meet the specification
Pseudo-code	A precise, language-agnostic description of the algorithm
Implementation	Concrete code in a programming language

From Problem to Program

- Writing a correct program is a multi-step process
- Each step answers a different question

Problem statement	What needs to be solved? (informal, in natural language)
Specification	What the solution must do, independently of how
Algorithm	A high-level method to meet the specification
Pseudo-code	A precise, language-agnostic description of the algorithm
Implementation	Concrete code in a programming language

- Correctness:
  - Algorithm is correct with respect to the specification
  - Implementation is correct with respect to the algorithm
- Bugs arise when these levels are confused or skipped

Levels of Specification (1/2)

- A specification can be expressed at several levels of formality
- Each level has different goals, costs, and tool support

Levels of Specification (1/2)

- A specification can be expressed at several levels of formality
- Each level has different goals, costs, and tool support

1. Human-readable description
   - Natural language (accompanying an algorithm or a program, in a README)
   - Explains intent, intuition, and assumptions
   - Accessible but ambiguous

Levels of Specification (1/2)

- A specification can be expressed at several levels of formality
- Each level has different goals, costs, and tool support

1. Human-readable description
   - Natural language (accompanying an algorithm or a program, in a README)
   - Explains intent, intuition, and assumptions
   - Accessible but ambiguous
2. Unstructured comments
   - Informal comments in the code
   - Useful for maintainers, close to relevant portions of the code
   - Not machine-checked

Levels of Specification (1/2)

- A specification can be expressed at several levels of formality
- Each level has different goals, costs, and tool support

1. Human-readable description
   - Natural language (accompanying an algorithm or a program, in a README)
   - Explains intent, intuition, and assumptions
   - Accessible but ambiguous
2. Unstructured comments
   - Informal comments in the code
   - Useful for maintainers, close to relevant portions of the code
   - Not machine-checked
3. Structured comments
   - Documented interfaces: preconditions, postconditions, invariants
   - Parsed by documentation tools (Doxygen, Sphinx)
   - Still informal, but disciplined

Levels of Specification (2/2)

4. Assertions and contracts
   - Formal annotations attached to code
   - Checked at compile time or at runtime
   - Examples:
     - Assertions
     - Experimental/proposed versions of C++: `[[expects]]`, `[[ensures]]` and contracts

Levels of Specification (2/2)

4. Assertions and contracts
   - Formal annotations attached to code
   - Checked at compile time or at runtime
   - Examples:
     - Assertions
     - Experimental/proposed versions of C++: `[[expects]]`, `[[ensures]]` and contracts
5. Formal specifications
   - Mathematical semantics (logic, automata, temporal properties)
   - Used by model checkers and verification tools

Levels of Specification (2/2)

4. Assertions and contracts
   - Formal annotations attached to code
   - Checked at compile time or at runtime
   - Examples:
     - Assertions
     - Experimental/proposed versions of C++: `[[expects]]`, `[[ensures]]` and contracts
5. Formal specifications
   - Mathematical semantics (logic, automata, temporal properties)
   - Used by model checkers and verification tools

- Trade-off: More formality $\Rightarrow$ more precision, higher cost
- We will stop at structured comments + assertions

Preconditions and Postconditions

- Precondition: what must be true before calling a function
- Postcondition: what is guaranteed after it returns
- For a function to be correct, whenever the precondition is satisfied, the postcondition must hold upon return

Example: Binary Search (Specification)

Input: Sorted array T of length n; key x

Precondition: T is sorted in nondecreasing order

Postcondition:

- returns an index i such that $T[i] = x$, or
- reports that $x \notin T$

Invariant

- An invariant is a property that:
  - holds in the initial state
  - is preserved by every allowed state change
- Loop invariant:
  - the invariant is associated with a loop
  - each iteration is a state transition
- State (or object) invariant:
  - the invariant is associated with an object or system
  - each method call is a state transition
- Useful invariants help showing correctness (often, along with completion of the loop or termination of the program)

Example: Maximum of an Array

for $i \leftarrow 1$ **to** n **do**
  if $T[i] > m$ **then**
    $m \leftarrow T[i]$;

Example: Maximum of an Array

for $i \leftarrow 1$ **to** n **do**
  if $T[i] > m$ **then**
    $m \leftarrow T[i]$;

- Invariant: m is the maximum of elements $T[1..i-1]$

Running Example

- We study a small banking system
- Objects:
  - individual bank accounts
  - a manager coordinating transfers
- This example involves state, invariants, and interactions

Abstract Specification: Bank Account

- Each account has a balance $b \in \mathbb{R}$
- Supported operations:
  - deposit(x)
  - withdraw(x)
- Observable behavior only: no implementation details

Preconditions, Postconditions, Invariants

deposit(x):
- Pre: $x > 0$
- Post: $b_{\text{new}} = b_{\text{old}} + x$

withdraw(x):
- Pre: $x > 0 \wedge b \geqslant x$
- Post: $b_{\text{new}} = b_{\text{old}} - x$

Invariant: $b \geqslant 0$

Structured Documentation: Why Tools Matter

- Specifications must be:
  - written once
  - readable by humans
  - kept consistent with code
- Documentation generators help enforce structure

Doxygen (C/C++)

- Extracts documentation from structured comments
- Supports:
  - functions, classes, files
  - preconditions and invariants
- Widely used in C/C++ projects

C++

```
/**
 * @brief Withdraw money from the account
 * @pre amount > 0 and balance >= amount
 * @post balance is decreased by amount
 */
void withdraw(unsigned amount);
```

Sphinx (Python)

- Main documentation generator for Python
- Very similar features to Doxygen

Python

```
def withdraw(self, amount):
    """
    Withdraw money from the account.

    :param amount: positive amount
    :precondition: balance >= amount
    :postcondition: balance decreases by amount
    """
    ...
```

Assertions: Executable Contracts

- An assertion is a condition that must hold at a given point during execution
- If the condition is false, the program fails immediately
- Assertions are used to:
  - document assumptions (preconditions, invariants)
  - detect bugs early
  - localize errors close to their cause
- Assertions are not error handling:
  - they detect programmer errors
  - they should never fail in a correct program

Assertions in C, C++, and Python (1/2)

C

```c
#include <assert.h>

void withdraw(unsigned balance, unsigned amount) {
    assert(amount > 0);
    assert(balance >= amount);
    /* ... */
}
```

C++

```cpp
#include <cassert>

void withdraw(unsigned amount) {
    assert(amount > 0);        // runtime check
    /* ... */
}

static_assert(sizeof(int) >= 4,
              "int must be at least 32 bits");
```

Assertions in C, C++, and Python (2/2)

Python

```
def withdraw(balance, amount):
    assert amount > 0
    assert balance >= amount
    # ...
```

- `assert`: checked at runtime
- `static_assert`: checked at compile time
- Assertions can be disabled in optimized builds

Outline

Reasoning About Stateful Objects

- State evolves over time
- We reason about:
  - method effects
  - preservation of invariants
- Correctness means invariants are never broken

Invariant Method

To prove correctness of a loop:

- Initialization: invariant holds initially
- Preservation: invariant remains true
- Termination: invariant implies postcondition

How to Find a Loop Invariant

- Ask: what is already true at iteration i?
- Express progress: what part of the input is processed?
- Ignore control variables, focus on meaning

Binary Search: Pseudo-Code

Input: sorted array $T[0\,..\,n-1]$, key x

```
function BinarySearch(T, x) low ← 0;
high ← n − 1;
// Loop invariant: ∀i < low, T[i] < x and ∀i > high, T[i] > x
while low ⩽ high do
    m ← ⌊(low + high)/2⌋;
    if T[m] = x then
        return m;
    else
        if T[m] < x then
            low ← m + 1;
        else
            high ← m − 1;
return not found;
```

Binary Search: Loop Invariant

Assume an array $T[0\,..\,n-1]$ sorted in increasing order, and a key x.

Algorithm variables:

$$low,\ high \quad \text{with } 0 \leqslant low \leqslant high + 1 \leqslant n$$

Loop invariant:

$$\forall i < low,\ T[i] < x \quad \text{and} \quad \forall i > high,\ T[i] > x$$

Binary Search: Loop Invariant

Assume an array $T[0\,..\,n-1]$ sorted in increasing order, and a key x.

Algorithm variables:

$$low,\ high \quad \text{with } 0 \leqslant low \leqslant high + 1 \leqslant n$$

Loop invariant:

$$\forall i < low,\ T[i] < x \quad \text{and} \quad \forall i > high,\ T[i] > x$$

Meaning:

- If x occurs in T, then it is in the interval

$$[low,\ high]$$

- All elements outside this interval are already known to be irrelevant

Binary Search: Loop Invariant

Assume an array $T[0\,..\,n-1]$ sorted in increasing order, and a key x.

Algorithm variables:

$$low,\ high \quad \text{with } 0 \leqslant low \leqslant high + 1 \leqslant n$$

Loop invariant:

$$\forall i < low,\ T[i] < x \quad \text{and} \quad \forall i > high,\ T[i] > x$$

Meaning:

- If x occurs in T, then it is in the interval

$$[low,\ high]$$

- All elements outside this interval are already known to be irrelevant

Correctness argument:

- Initialization: holds for $low = 0$, $high = n - 1$
- Preservation: each iteration shrinks the interval
- Termination: when $low > high$, x is not in T

Python Implementation of Bank Account

Python

```
class BankAccount:
    def __init__(self, balance):
        assert balance >= 0
        self._balance = balance

    def deposit(self, amount):
        assert amount > 0
        self._balance += amount

    def withdraw(self, amount):
        assert amount > 0 and self._balance >= amount
        self._balance -= amount

    def getBalance(self):
        return self._balance
```

Local Reasoning on Running Example

- Constructor establishes invariant ($b \geqslant 0$)
- Each method preserves it
- Encapsulation ensures no external violation

From Local to Global Correctness

- Programs manipulate multiple objects
- Correctness properties span objects
- We introduce a global invariant:
  - Let $\mathcal{A}$ be the set of all accounts
  - Define total balance:

$$S = \sum_{a \in \mathcal{A}} \text{balance}(a)$$

  - Invariant: transfers preserve S

Python: Account Manager

Python

```
from BankAccount import BankAccount

class AccountManager:
    def __init__(self):
        self._accounts = {}

    def add_account(self, name, balance):
        self._accounts[name] = BankAccount(balance)

    def transfer(self, src, dst, amount):
        self._accounts[src].withdraw(amount)
        self._accounts[dst].deposit(amount)

    def getBalance(self, name):
        return self._accounts[name].getBalance()
```

Reasoning About Transfer

- Withdrawal decreases one balance
- Deposit increases another by same amount
- Therefore total balance unchanged
- Only true if preconditions for using `transfer` are met:
  - Both `src` and `dst` are names of valid accounts
  - The amount is positive
  - The balance of the source account is $\geqslant$ the amount

Outline

Why Testing is Necessary

- Reasoning can be incomplete
- Testing checks specific executions
- Tests encode expected behavior
- They operationalize the specification
- They do not replace reasoning

Unit Tests vs Feature Tests

Unit tests: individual functions

Feature tests: user-visible behavior

Regression tests: prevent reintroducing bugs

Test-Driven Development

- Write failing test
- Implement minimal code so that test passes
- TDD is a workflow, not a proof

pytest (Python)

- Lightweight testing framework
- Tests are simple functions with assertions
- Excellent for unit and feature tests

Python

```python
from AccountManager import AccountManager

def test_transfer_preserves_total():
    m = AccountManager()
    m.add_account("A", 10)
    m.add_account("B", 5)
    m.transfer("A", "B", 3)
    assert m.getBalance("A") + m.getBalance("B") == 15
```

Boost.Test (C++)

- Part of Boost libraries
- Supports unit tests for C++

C++

```
BOOST_AUTO_TEST_CASE(withdraw_test) {
  BankAccount a(10);
  a.withdraw(3);
  BOOST_CHECK_EQUAL(a.getBalance(), 7);
}
```

What Tests Miss

- Untested paths
- Unexpected interactions
- Undefined behavior

Outline

What Is Debugging?

- Debugging is the process of understanding and fixing incorrect behavior
- It starts when:
  - the program crashes
  - a test fails
  - an invariant is violated
- Debugging is not:
  - random trial-and-error
  - adding prints until it works

Why Debugging Matters

- Bugs are inevitable in non-trivial programs
- Debugging:
  - reveals incorrect assumptions
  - improves program understanding
  - strengthens specifications and invariants
- Good debugging:
  - saves time in the long run
  - prevents similar bugs in the future

How to Debug

- Debugging is a scientific method
- Typical workflow:
  1. Reproduce the problem reliably
  2. Formulate hypotheses about the cause
  3. Observe execution (debugger, tracing)
  4. Eliminate possibilities
  5. Fix the root cause, not the symptom
- Key idea: invariants guide the investigation

Poor Man's Debugging: Tracing

- Print intermediate values (with `print`, `printf`, `std::cout`, etc.)
- Useful for quick inspection
- Limited scalability

Assertions as Debugging Aids

- Assertions check invariants dynamically
- They fail close to the cause
- Useful during development and testing

Debuggers: Core Concepts

- A debugger lets you observe and control program execution
- Core capabilities are largely the same across languages and tools
- We illustrate them using `pdb` (Python) and `gdb` (C/C++)

- Breakpoints: stop execution at chosen locations
- Stepping: execute the program one instruction at a time
- Inspection: examine variables and program state
- Control flow: continue, pause, or abort execution

pdb (Python Debugger)

- Built into Python
- Supports:
  - breakpoints
  - step-by-step execution
  - variable inspection

Python

```
import pdb; pdb.set_trace()
```

gdb (C/C++)

- Standard debugger for C and C++
- Inspect stack, memory, variables
- Essential for low-level debugging
- Programs must be compiled with debug information:

`gcc -g ...` or `g++ -g ...`

Common Debugger Commands: pdb vs gdb

Action	pdb	gdb
Set breakpoint	`break f`	`break f`
Continue execution	`continue`	`continue`
Step into	`step`	`step`
Step over	`next`	`next`
Inspect variable	`p x`	`print x`
Show call stack	`where`	`backtrace`
Quit debugger	`quit`	`quit`

- Same ideas, different syntax
- Skills transfer directly between debuggers

Debugging in Practice: VS Code

- Unified interface for:
  - pdb
  - gdb / lldb
- Breakpoints, watch variables, call stack
- Encourages disciplined debugging

Other Debugging Tools (Linux)

- Some bugs require observing interactions outside the program
- Linux provides powerful tools for this purpose

- Valgrind
  - Detects memory errors (invalid access, leaks)
  - Essential for C and C++ programs
- strace
  - Traces system calls
  - Shows interactions with the operating system
- ltrace
  - Traces library function calls
  - Useful to understand dynamic linking behavior

Other Debugging Tools: Typical Use Cases

- Valgrind
  - Example bug: program crashes "randomly" due to an out-of-bounds write or use-after-free
  - Valgrind detects memory errors even when gdb shows nothing obvious
- strace
  - Example bug: program fails because a file cannot be opened
  - `strace` reveals the failing `open()` or `read()` system call
- ltrace
  - Example bug: unexpected behavior due to incorrect library usage
  - `ltrace` shows which shared library functions are actually called

Key idea: some bugs only appear when observing memory, system calls, or libraries.

Beyond Linux

- Similar tools exist on other systems:
  - macOS: `Instruments`, `dtruss`, `lldb`
  - Windows: Visual Studio Debugger, Application Verifier
- Memory checkers, tracers, and profilers exist everywhere
- Key idea:
  - bugs can occur at different abstraction layers
  - choose tools appropriate to the layer

Outline

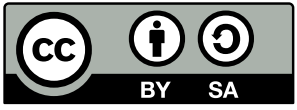
Putting It All Together

- Specifications define intent
- Reasoning ensures correctness
- Testing validates behavior
- Debugging repairs violations

Final Message

- Correctness is engineered, not accidental
- Tools support discipline, they do not replace thinking
- Write programs you can explain and defend

Licensing



Bibliography I