



**HOW TO WRITE
BETTER CODE**

ISE 114

SUNY AT STONY BROOK

“AS SOON AS WE STARTED PROGRAMMING, WE FOUND TO OUR SURPRISE THAT IT WASN'T AS EASY TO GET PROGRAMS RIGHT AS WE HAD THOUGHT. DEBUGGING HAD TO BE DISCOVERED. I CAN REMEMBER THE EXACT INSTANT WHEN I REALIZED THAT A LARGE PART OF MY LIFE FROM THEN ON WAS GOING TO BE SPENT IN FINDING MISTAKES IN MY OWN PROGRAMS.”

— MAURICE WILKES, DESIGNER OF EDSAC, ON PROGRAMMING, 1949

“IF BUILDERS BUILT BUILDINGS THE WAY PROGRAMMERS WRITE PROGRAMS, THEN THE FIRST WOODPECKER THAT CAME ALONG WOULD DESTROY CIVILIZATION.”

— WEINBERG'S SECOND LAW.

THE MARINER I SPACE PROBE WAS LAUNCHED FROM CAPE CANAVERAL ON 28 JULY 1962 TOWARDS VENUS. AFTER 13 MINUTES' FLIGHT A BOOSTER ENGINE WOULD GIVE ACCELERATION UP TO 25,820 MPH; AFTER 44 MINUTES 9,800 SOLAR CELLS WOULD UNFOLD; AFTER 80 DAYS A COMPUTER WOULD CALCULATE THE FINAL COURSE CORRECTIONS AND AFTER 100 DAYS THE CRAFT WOULD CIRCLE THE UNKNOWN PLANET, SCANNING THE MYSTERIOUS CLOUD IN WHICH IT IS BATHED.

HOWEVER, WITH AN EFFICIENCY THAT IS TRULY HEARTENING, MARINER I PLUNGED INTO THE ATLANTIC OCEAN ONLY FOUR MINUTES AFTER TAKEOFF.

INQUIRIES LATER REVEALED THAT A MINUS SIGN HAD BEEN OMITTED FROM THE INSTRUCTIONS FED INTO THE COMPUTER. "IT WAS HUMAN ERROR", A LAUNCH SPOKESMAN SAID.

THIS MINUS SIGN COST £4,280,000.

— STEPHEN PILE, "THE BOOK OF HEROIC FAILURES"

OUTLINE OF TOPICS

- **TEST-FIRST DEVELOPMENT**
- **DEBUGGING STRATEGIES**
- **FORMAL METHODS**

“IN EVERY NON-TRIVIAL PROGRAM THERE IS AT LEAST ONE BUG.”
— UNIX FORTUNE



TESTING

WHY SHOULD I TEST?

- ACCORDING TO THE U.S. NATIONAL INSTITUTE OF STANDARDS & TECHNOLOGY, SOFTWARE BUGS COST THE U.S. ECONOMY ABOUT \$59.5 BILLION/YEAR
- THE LONGER BUGS REMAIN IN A PROGRAM, THE MORE COSTLY IT IS TO REMOVE THEM
- IN INDUSTRY, TESTING/QA CONSUMES 50% OF DEVELOPMENT TIME/EFFORT
- GOOD STRATEGIES FOR TESTING AND DEBUGGING WILL REDUCE YOUR OVERALL WORKLOAD AND IMPROVE YOUR PROGRAMMING EFFICIENCY

BASIC TERMINOLOGY

- **BUG — AN ERROR IN A PROGRAM**
- **TESTING — THE PROCESS OF ANALYZING AND EXECUTING A PROGRAM TO DETERMINE WHETHER IT HAS BUGS**
- **DEBUGGING — THE PROCESS OF LOCATING A BUG AND REMOVING IT**
- **GOOD STRATEGIES FOR TESTING AND DEBUGGING WILL REDUCE YOUR OVERALL WORKLOAD AND IMPROVE YOUR PROGRAMMING EFFICIENCY**

TYPES OF BUGS

- **“BOHR-BUGS”**
 - EASY TO REPRODUCE
 - THEREFORE, EASY TO FIX
- **“HEISENBUGS”**
 - DIFFICULT TO REPRODUCE
 - EX. RACE CONDITIONS
 - HARDER TO LOCATE AND FIX

PROGRAMMING VS. TESTING

- **DIFFERENT GOALS:**

- **PROGRAMMING: “MAKE IT WORK”**

- **TESTING: “MAKE IT FAIL”**

- **TESTING IS DESIGNED TO UNCOVER FLAWS IN A PROGRAM BEFORE IT IS RELEASED**

- **TESTING IS A SYSTEMATIC ATTEMPT TO FIND FAULTS**

- **THIS IS NOT THE SAME AS ENSURING THAT FAULTS ARE NOT PRESENT!**

● **LUBARSKY’S LAW OF CYBERNETIC ENTOMOLOGY: THERE’S ALWAYS ANOTHER BUG.**

CREATING BETTER SOFTWARE

- FAULT AVOIDANCE TRIES TO PREVENT THE INSERTION OF DEFECTS INTO THE SYSTEM BEFORE IT IS RELEASED
- FAULT DETECTION CONDUCTS EXPERIMENTS TO IDENTIFY DEFECTS BEFORE SYSTEM RELEASE
- THIS IS WHERE TESTING FITS IN
- FAULT TOLERANCE DEALS WITH SYSTEM FAILURES BY RECOVERING FROM THEM AT RUNTIME
- E.G., EXCEPTION-HANDLING

DIFFERENT TYPES OF TESTS

- **UNIT TESTS — MAKE SURE A SINGLE ELEMENT (METHOD/OBJECT) OF THE SYSTEM WORKS CORRECTLY**
- **FUNCTIONAL TESTS — MAKE SURE THE SYSTEM OFFERS A GIVEN FEATURE**
- **PARALLEL TESTS — PROVE THAT THE NEW SYSTEM WORKS EXACTLY LIKE THE OLD SYSTEM**
- **STRESS TESTS — SIMULATE THE WORST POSSIBLE WORKLOAD**
- **MONKEY TESTS — CHECK SYSTEM RESPONSE IN THE FACE OF NONSENSICAL INPUT**

NEW DEVELOPMENT STYLE

- **THE OLD WAY**

- **EDIT**
- **COMPILE**
- **EXECUTE**
- **TEST**
- **FOCUS: WRITE CODE, THEN TEST TO SEE IF IT WORKS (AND FIX IT IF NECESSARY)**

- **THE NEW WAY**

- **DESIGN TESTS**
- **WRITE CODE**
- **COMPILE AND EXECUTE**
- **FOCUS: CREATE TESTS FIRST, THEN WRITE CODE TO PASS THE TESTS**

DETAILED TEST-FIRST DEVELOPMENT PROCESS

- 1. SELECT A BIT OF CODE TO IMPLEMENT**
- 2. WRITE A TEST BASED ON THE PROBLEM STATEMENT**
- 3. RUN THE TEST; YOUR CODE SHOULD FAIL AT FIRST**
- 4. WRITE SOME CODE TO PASS THE TEST**
- 5. RUN THE TEST AGAIN**
- 6. REPEAT (4) AND (5) UNTIL THE TEST IS PASSED**
- 7. MOVE ON TO THE NEXT PIECE OF FUNCTIONALITY**

THE ART OF TESTING

- IT'S MUCH EASIER TO PASS A TEST IF YOU ALREADY KNOW THE ANSWER!
- A TEST CASE HAS FOUR BASIC PARTS: A NAME, A SET OF INPUT, AN EXPECTED OUTPUT (THE ORACLE), AND THE ACTUAL OUTPUT (THE LOG)
 - IF THE LOG MATCHES THE ORACLE, THE TEST PASSES
- A TEST SUITE IS A COLLECTION OF TEST CASES

TEST SUITE EXAMPLE

NAME	INPUT	EXPECTED RESULT	ACTUAL RESULT
TESTZEROINPUT	0	0	0
TESTPOSITIVEINPUT	2	2	2
TESTNEGATIVEINPUT	-4	4	-4

TESTING STYLES

1. EXHAUSTIVE TESTING

- TEST ALL POSSIBLE INPUT VALUES

2. BLACKBOX TESTING

- CHOOSE INPUT BASED ON FUNCTIONAL SPECIFICATION
- USE EQUIVALENCE CLASSES TO REDUCE TESTING SCOPE

3. "GLASS BOX" TESTING

- CHOOSE TEST INPUTS BASED ON HOW THE UNDERLYING CODE IS IMPLEMENTED

TESTING GUIDELINES

- **TEST EACH METHOD AS IT IS COMPLETED**
 - **THIS MAY NOT ALWAYS BE FEASIBLE; SEVERAL METHODS MAY INTERACT IN SUCH A WAY THAT TESTING THEM TOGETHER IS MORE APPROPRIATE**
- **ALWAYS SPECIFY (DOCUMENT) WHAT THE METHOD IS SUPPOSED TO DO BEFORE THE METHOD HEADER**
 - **EXPLAIN WHY THE METHOD EXISTS**
 - **EXPLAIN EXACTLY WHAT IT IS SUPPOSED TO DO**

TESTING COUPLED METHODS

- IF METHOD A USES METHOD B, THERE ARE TWO APPROACHES:
 1. BOTTOM UP: TEST METHOD B FULLY, THEN TEST A
 2. TOP DOWN: TEST METHOD A AND USE A STUB FOR B
- A STUB IS A METHOD THAT STANDS IN FOR THE FINAL VERSION AND DOES LITTLE ACTUAL WORK
- IT USUALLY DOES SOMETHING TRIVIAL, LIKE PRINTING A MESSAGE OR RETURNING A FIXED VALUE

PRINCIPLES OF TESTING

1. TEST EARLY AND OFTEN (TEST AS YOU GO)
2. TEST ONLY ONE THING AT A TIME
3. TEST 0, 1, & MANY INPUTS
4. TEST NULL, BEGINNING, MIDDLE, & END OF RANGE OF INPUT
5. VERIFY THE DOCUMENTATION
6. DEVELOP TEST CASES THAT PROVIDE COMPLETE CODE COVERAGE

CODE COMMENTS

- **COMMENTS ARE NOTES TO THE READER OF YOUR PROGRAM (WHICH INCLUDES YOU AS THE AUTHOR)**
 - **“EMBEDDED DOCUMENTATION”**
 - **JAVA IGNORES COMMENTS WHEN PROCESSING CODE**
- **COMMENTS COME IN THREE FORMS:**
 - **EVERYTHING FROM // UNTIL THE END OF THE LINE**
 - **EVERYTHING BETWEEN /* AND */ TAGS**
 - **EVERYTHING BETWEEN /** AND */ TAGS (JAVADOC)**

GOOD COMMENTING STYLE

- **COMMENTS SHOULD EXPLAIN WHY, NOT HOW**
 - **“HOW” IS OBVIOUS FROM THE CODE ITSELF**
- **EVERY METHOD SHOULD HAVE A COMMENT BLOCK:**
 - **METHOD NAME AND PURPOSE**
 - **LIST OF PARAMETERS AND RETURN VALUE**
 - **IF POSSIBLE, SAMPLE CALLS WITH EXPECTED RESULTS**

JAVADOC

- JAVADOC IS A TOOL THAT AUTOMATICALLY CREATES HTML-FORMATTED DOCUMENTATION FROM COMMENTS IN YOUR SOURCE CODE
- THE ONLINE JAVA API USES THIS FORMAT
- TO USE JAVADOC, USE THE “JAVADOC” COMMAND:

```
javadoc *.java
```

- BY DEFAULT, JAVADOC ONLY SHOWS PUBLIC AND PROTECTED CLASSES AND FIELDS; TO SHOW PRIVATE DATA, ADD THE “-private” COMMAND-LINE OPTION

JAVADOC TAGS

- A JAVADOC COMMENT IS SURROUNDED BY `/**` AND `*/`
- ANY TEXT INSIDE THESE MARKERS IS COPIED AS-IS
- USE THESE TAGS TO EMPHASIZE SPECIAL INFO:
 - `@author` *author-information*
 - `@param` *parameter-name description*
 - `@return` *description-text*

JAVADOC EXAMPLE

```
/**
 * Raises an integer value to a user-specified power
 *
 * @author Michael S. Tashbook
 * @param base The number to be multiplied
 * @param power The number of times to multiply the base
 * @returns An integer equal to base^power
 */
public int raiseToPower (int base, int power)
{ ... }
```

DEBUGGING

DEBUGGING

- FINDING THE PRECISE LINE OF CODE WHERE THE ERROR FIRST OCCURRED IS HALF THE BATTLE
- TRACKING DOWN A BUG:
 - PLACE PRINT STATEMENTS AT THE BEGINNING AND END OF THE METHOD
 - GLEAN INFO FROM EACH TEST CASE
 - CHECK THE VALUES OF VARIABLES USING PRINT STATEMENTS (LATER WE'LL USE DEBUGGING TOOLS LIKE THOSE IN ECLIPSE)

AVOIDING PRINT OVERLOAD

- IT'S TEMPTING TO PRINT EVERYTHING WHEN DEBUGGING
 - THIS PRODUCES A FLOOD OF (GENERALLY USELESS) INFORMATION TO SIFT THROUGH
- TRY TO PRINT ONLY WHEN A TEST CASE FAILS
- BEYOND THAT, USE A "HALVING" (BINARY SEARCH) APPROACH TO PRINTING:
 - PRINT AT THE START, END, AND MIDDLE OF THE CODE
 - REPEATEDLY CUT THE RANGE OF LINES IN HALF

ASSERTIONS

- **USE AN `assert` STATEMENT TO GUARANTEE THAT A CERTAIN CONDITION IS TRUE**
- **IF THE CONDITION IS TRUE, NOTHING WILL HAPPEN**
- **IF IT ISN'T, JAVA WILL TERMINATE YOUR PROGRAM WITH AN ERROR MESSAGE**

- **USAGE:**

```
assert boolean-expression : output-if-false ;
```

- **EX. `assert x % 2 == 1 : "x is " + x ;`**



FORMAL METHODS

FORMAL METHODS

- IN SOFTWARE DEVELOPMENT, FORMAL METHODS PROVIDE TECHNIQUES FOR DESCRIBING A SOFTWARE ARTIFACT (E.G. SPECIFICATIONS, DESIGNS, SOURCE CODE) SUCH THAT FORMAL PROOFS ARE POSSIBLE, IN PRINCIPLE, ABOUT PROPERTIES OF THE ARTIFACT SO EXPRESSED.
- MATHEMATICALLY PROVES CODE WILL WORK
- THIS MAY BE OVERALL CORRECTNESS, OR IT MAY JUST GUARANTEE THAT CERTAIN PROPERTIES ARE TRUE

VALIDATION VS. VERIFICATION

- **VALIDATION: “DID WE BUILD THE RIGHT THING?”**
 - **DETERMINES WHETHER THE PRODUCT MATCHES WHAT THE CUSTOMER REALLY WANTED**
- **VERIFICATION: “DID WE BUILD THE THING RIGHT?”**
 - **DETERMINES WHETHER THE PRODUCT IMPLEMENTS THE SPECIFICATION CORRECTLY**
- **IT IS ENTIRELY POSSIBLE TO HAVE A VERIFIED PRODUCT THAT IS INVALID (DOESN'T DO WHAT THE CUSTOMER WANTS)**

PRE- AND POSTCONDITIONS

// PRECONDITION

PROGRAM CODE

// POSTCONDITION

- **A PRECONDITION (P) IS THE CONDITION THAT IS TRUE BEFORE THE CODE EXECUTES.**
- **A POSTCONDITION (Q) IS THE CONDITION THAT IS TRUE IF THE PRECONDITION IS TRUE AND THE CODE EXECUTES COMPLETELY.**

EXAMPLES

```
// PRE: x < 0
```

```
y = x * -2;
```

```
// POST: (x < 0) ∧ (y > 0) ∧ (y = -2x)
```

```
// PRE: x ≥ 0
```

```
x = x % 5;
```

```
// POST: (x' ≥ 0) ∧ (x' ≤ 4)
```

THE ASSIGNMENT RULE

- THE ASSIGNMENT RULE REASONS BACKWARD FROM A GOAL TO THE REQUIRED STARTING CONDITIONS.

- GIVEN:

// P P IS THE PRECONDITION

x = E; X IS A VARIABLE, E IS AN EXPRESSION

// Q Q IS THE POSTCONDITION

- DERIVE P BY REPLACING ALL OCCURRENCES OF X IN Q WITH E.

- NOTE: X MIGHT BE REPRESENTED AS x' IN Q.

The Assignment Rule (example)

What is the precondition of the following assignment statement?

// $P = ?$

$a = b + 5;$

// $Q = \{ (a > 0) \wedge (b > 0) \wedge (a = b + 5) \}$

ANSWER Using the assignment rule, take Q and replace a with $b + 5$ to get P :

$P = \{ (b + 5 > 0) \wedge (b > 0) \wedge (b + 5 = b + 5) \}$

$= \{ (b > -5) \wedge (b > 0) \wedge \text{TRUE} \}$

$= \{ (b > -5) \wedge (b > 0) \}$

$= \{ (b > 0) \}$

ANOTHER EXAMPLE

- WHAT IS THE PRECONDITION OF THE FOLLOWING ASSIGNMENT STATEMENT?

// P = ?

A = B - A;

// Q = { (A' > 0) ∧ (B > 0) ∧ (A' = B - A) }

- NOTE HERE THAT A' DENOTES THE NEW VALUE OF A AFTER THE STATEMENT IS EXECUTED

SOLUTION TO EXAMPLE

$$\begin{aligned} P &= \{ (B - A > 0) \wedge (B > 0) \wedge (B - A = B - A) \} \\ &= \{ (B > A) \wedge (B > 0) \wedge \text{TRUE} \} \\ &= \{ (B > A) \wedge (B > 0) \} \end{aligned}$$

PROOF EXAMPLE

- PROVE THAT THIS ASSIGNMENT STATEMENT IS CORRECT:

$$// P = \{ (M < 0) \wedge (N > 0) \}$$

$$M = N - M;$$

$$// Q = \{ (M' > N) \wedge (N > 0) \}$$

- ANSWER: START WITH Q AND DERIVE P:

$$\begin{aligned} Q &= \{ (N - M > N) \wedge (N > 0) \} \\ &= \{ (-M > 0) \wedge (N > 0) \} \\ &= \{ (M < 0) \wedge (N > 0) \} = P \end{aligned}$$

EXTENDED SEQUENCES

- DERIVE A PRECONDITION TO THE FOLLOWING SEQUENCE OF STATEMENTS

// P = ?

TEMP = A;

A = B;

B = TEMP;

// Q = { (B' = A) \wedge (A' = B) }

EXTENDED SEQUENCES

- THE PROBLEM CAN BE RESTATED AS FOLLOWS:

```
// P = ?  
TEMP = A;  
// P' = ?  
A = B;  
// P'' = ?  
B = TEMP;  
// Q = { (B' = A) ∧ (A' = B) }
```

SOLUTION

- USE THE ASSIGNMENT RULE 3 TIMES TO GET:

$$\begin{aligned}P'' &= \{ (\text{TEMP} = \text{A}) \wedge (\text{A}' = \text{B}) \} \\P' &= \{ (\text{TEMP} = \text{A}) \wedge (\text{B} = \text{B}) \} \\&= \{ (\text{TEMP} = \text{A}) \wedge \text{TRUE} \} \\&= \{ (\text{TEMP} = \text{A}) \} \\P &= \{ (\text{A} = \text{A}) \} \\&= \{ \text{TRUE} \} \\&= \{ \} \end{aligned}$$

- $\{ \}$ REPRESENTS THE UNIVERSAL CONDITION, WHICH IS ALWAYS TRUE. THUS, THERE ARE NO PRECONDITIONS.

LOOP INVARIANTS

- AN INVARIANT IS A CONDITION THAT IS TRUE BEFORE AND AFTER A STATEMENT EXECUTES.
- A LOOP INVARIANT I IS TRUE IN FOUR LOCATIONS:

```
// I
WHILE (C)
{
    // I
    LOOP BODY
    // I
}
// I
```