Software Testing

Dr. Mark C. Paulk

Mark.Paulk@utdallas.edu, Mark.Paulk@ieee.org

UT Dallas, Jonsson School of Engineering and Computer Science

Verification & Validation

Verification **"Are we building the product right."**

- **The software should conform to its specification.**

Validation **"Are we building the right product."**

- **The software should do what the user really requires.**

Testing is verification.

The software is correct if it meets the requirements.

The Purpose of Testing

What is the purpose of testing?

To demonstrate that the software works?

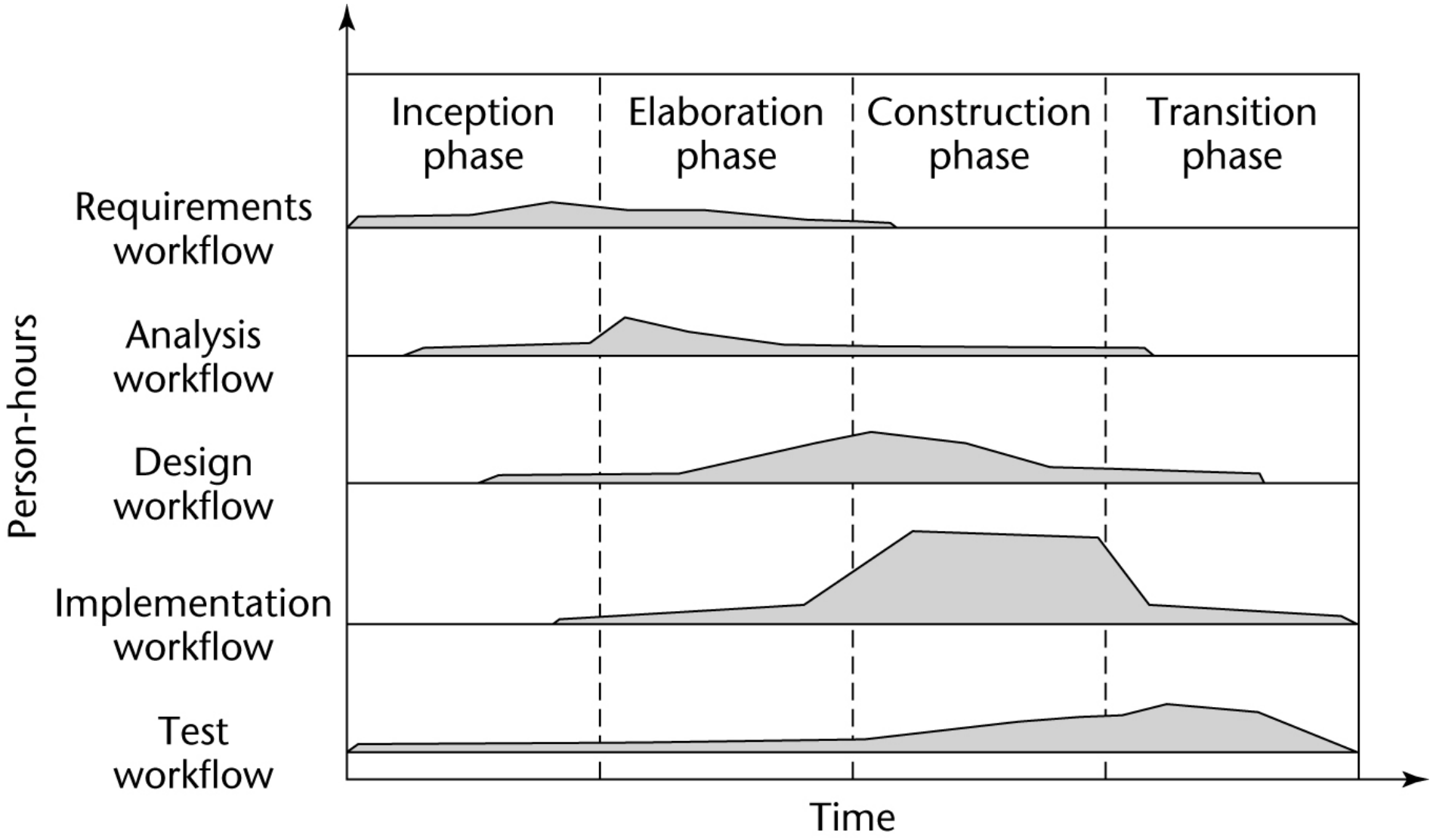
NO!

There is always the possibility of latent defects…

To find defects in the software?

YES!

UP Core Workflows and Phases



UP Test Workflow

The aim of the test workflow is to remove defects from the software effectively and efficiently.

Testing is the responsibility of

- **developers and maintainers (unit testing)**
- **independent testing group (aka QA in many organizations) (integration and system testing)**
- **independent verification & validation group (acceptance testing)**

"Testing" encompasses

- **dynamic testing of the code**
- **static peer reviews of requirements, design, code, etc.**
- **formal specification and proofs of correctness**

Subtle Distinctions

A person makes a mistake

Which becomes a fault (bug, defect)

Resulting at execution in a failure

Which is wrong with some degree of error

From the IEEE definition of fault tolerance…

Test and Test Case

Test (*IEEE 24765: 2010*)

- **an activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component.**

Test case (*IEEE 29119-1:2013*)

- **set of test case preconditions, inputs (including actions, where applicable), and expected results**
  - **Examples: t_1: <x=0, y=1, result=3>, t_1: <0, 1, (3)>, t_1: <x=0, y=1>, t_1: <0, 1>**

Test Set and Test Procedure

Test set ***(IEEE 29119-1:2013)***

- **set of one or more test cases with a common constraint on their execution**
  - **Example: T = {t_1: <2, 2>, t_2: <0, 1>}**

Test procedure ***(IEEE 29119-1:2013)***

- **sequence of test cases in execution order, and any associated actions that may be required to set up the initial preconditions and any wrap up activities post execution**
  - **Note 1 to entry: Test procedures include detailed instructions for how to run a set of one or more test cases selected to be run consecutively, including set up of common preconditions, and providing input and evaluating the actual result for each included test case.**

Debugging

The process of determining the cause of failures (defects) and removing the defects.

If debugging is the process of removing bugs, then programming must be the process of putting them in.

- ***Dijkstra's Observation***

Programming – the art of debugging a blank sheet of paper.

- ***Robert Seacord***

Who Should Do Testing?

Programming is constructive.

Testing is destructive.

- **a successful test finds a fault**

Implication: programmers should not test their own code artifacts.

- **the programmer may do unit testing**
- **an independent testing group may then do integration and system testing**
- **if custom software development, the customer may hire an IV&V contractor to do acceptance testing**

Unit vs Integration Testing
(ISO/IEC/IEEE 24765)

Unit testing

- **testing of individual routines and modules by the developer or an independent tester**
- **a test of individual programs or modules in order to ensure that there are no analysis or programming errors**

Integration testing

- **testing in which software components, hardware components, or both are combined and tested to evaluate the interaction among them**

System vs Acceptance Testing
(ISO/IEC/IEEE 24765)

System testing

- **testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements**
  - **aka product testing**

Acceptance testing

- **testing conducted to determine whether a system satisfies its acceptance criteria and to enable the customer to determine whether to accept the system**
- **formal testing conducted to enable a user, customer, or other authorized entity to determine whether to accept a system or component.**

Regression Testing
(ISO/IEC/IEEE 24765)

Selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements.

Testing required to determine that a change to a system component has not adversely affected functionality, reliability or performance and has not introduced additional defects.

Categorizing Testing

Unit testing – usually white box

Integration testing – sometimes white box, sometimes black box

System testing – typically black box

Acceptance testing – typically black box

Regression testing – includes both black box and white box

Alpha testing – black box

- **done as part of an early release within the company for employees to provide early feedback**
- **Microsoft uses the phrase "eating your own dog food" to describe alpha testing**

Beta testing – black box

- **done as part of an early release to selected customers for them to provide early feedback before the official release**

Many other kinds of testing, e.g., endurance, interoperability, load, portability, risk-based, security, stress, …

Test-Driven Development

"Test, then code" means a failed test case is an entry criterion for writing code.

- **aka test-first programming**

1) **write an automated test**
2) **write enough code to pass the test**
3) **refactor to improve readability and remove duplication**
4) **repeat**

The goal of TDD is clean code that works.

- **Ron Jeffries**

Martin's Three Laws of TDD

You may not write production code unless you've first written a failing unit test.

You may not write more of a unit test than is sufficient to fail.

You may not write more production code than is sufficient to make the failing unit test pass.

Following these laws perfectly doesn't always make sense… The goal isn't perfect adherence to the laws — it's to decrease the interval between writing tests and production code to a matter of minutes.

R.C. Martin, "Professionalism and Test-Driven Development," IEEE Software, May/June 2007.

Consequences of TDD (Beck 2003)

Design organically with running code providing feedback between decisions

Write our own tests because we can't wait for someone else to

Environment must provide rapid response to small changes

- **continuous integration**

Design must consist of many loosely coupled, highly cohesive components to make testing easy

Benefits of TDD

(Beck 2004, Maximilien 2003, Rasmusson 2003)

The unit tests actually get written

- **provides a rich set of regression tests for continuous integration**

More consistent test writing – apply good test generation techniques

- **rapid feedback to developers, remove defects more effectively and efficiently**

Prevents scope creep

Clarifies detailed interfaces and behavior

- **Improved understanding of requirements**

Loosely coupled, highly cohesive testable modules – better OO designs

Repeatable, automated verification

Better unit testing leads to a reduction in defect density in testing

The confidence to change things – continuous integration implies continual regression testing so we don't break things that work

Leads to a rhythm of development

- **test, code, refactor**

Test driven development (TDD) is more important than both Scrum and XP. (Kniberg 2015)

TDD Observations

TDD is black-box testing, not white-box testing.

TDD demonstrates functionality by example.

TDD ties directly to the confirmation aspect of the user story's card / conversation / confirmation.

Testing Tools

X-unit test framework

- **Kent Beck**

- **JUnit for Java**
  - **junit.org**
  - **JUnit tutorial: http://www.asjava.com/junit/junit-tutorials-and-example/**
- **CppUnit for C++**
- **httpUnit**

FIT (Framework for Integrated Tests)

- **Ward Cunningham**

FitNesse = FIT + Ward's Wiki technology

- **Robert Martin**
- **www.fitnesse.org**

Black-Box Testing

Functional testing (IEEE 24765)

- **testing that ignores the internal mechanism of a system or component and focuses solely on the outputs generated in response to selected inputs and execution conditions**
- **testing conducted to evaluate the compliance of a system or component with specified functional requirements**
  - **Synonymous with: specification-based, closed-box testing**

Specification-based testing (IEEE 29119-1:2013)

- **testing in which the principal test basis is the external inputs and outputs of the test item, commonly based on a specification, rather than its implementation in source code or executable software**

Black-Box Testing Measures

Have you covered all the requirements with at least one test case?

- **prerequisite: a requirements "specification" with clearly stated and labeled requirements**
- **how would "user stories" (from agile methods) fit with this prerequisite?**

Measure – percentage of requirements covered

Have you covered all combinations of (independent) requirements?

Other black-box testing measures:

- **(fraction) percentage of equivalence classes covered**
- **(fraction) percentage of boundary values covered**

Equivalence Class Partitioning

Subdivide the input domain into a relatively small number of subdomains (equivalence classes).

- **assumes program P exhibits similar behavior for every element within the class**
- **subdomains cover the entire input domain**
  - **both legal inputs and illegal (unexpected) inputs**
- **subdomains are disjoint**
  - **every input belongs to one and only one equivalence class**

Equivalence Classes for Data Types

Range (integers and floating point)

- **one or more classes with values inside the range**
  - **there may be multiple legal ranges**
  - **ranges may or may not be adjacent**
- **two or more classes with values outside the range**
  - **less than the legal values**
  - **greater than the legal values**

Strings

- **at least one class containing legal strings**
- **at least one class containing illegal strings**
- **the empty string ε is an (illegal) equivalence class**
- **legality is determined based on constraints on the length and other semantic features of the string**

Enumerations

- each value in a separate class
- may combine values if program behavior is the same

Arrays

- one class containing all legal arrays
  - may be additional legal equivalence classes depending on semantics, e.g., values must be in [-3,3]
- one class containing only the empty array
- one class containing arrays larger than the expected size

Compound data types (e.g., structures in C++)

- legal and illegal values for each component

Unidimensional Partitioning

Consider one input variable at a time

- **simple and scalable**

Multidimensional partitioning can become too large.

- **number of test cases is the number of levels for each factor multiplied by the number of level for every other factor**

Combining illegal inputs does not add much value.

- **Addressing one illegal input frequently masks subsequent illegal inputs.**
- **Combinations of errors can cause unexpected results, so it's a judgment call.**

Equivalence Class Example One Variable

Program P: integer X in [0, 100]

Equivalence classes

- **E_1: $X < 0$** **(-∞, 0)**
- **E_2: $0 \leq X \leq 100$** **[0, 100]**
- **E_3: $X > 100$** **(100, +∞)**

Test cases

- **$t_1 = -10$** **(-∞, 0)**
- **$t_2 = 50$** **[0, 100]**
- **$t_3 = 120$** **(100, +∞)**

Test set T = {<-10>, <50>, <120>}

Equivalence Class Example Two Variables

Program P

- **integer *a* in [99, 999]**
- **integer *b* in [1, 100]**

Equivalence classes for *a*

- **E_1: $a < 99$**
- **E_2: $99 \leq a \leq 999$**
- **E_3: $a > 999$**

Equivalence classes for *b*

- **E_4: $b < 1$**
- **E_5: $1 \leq b \leq 100$**
- **E_6: $b > 100$**

Test set for equivalence classes

a [99, 999], *b* [1, 100]	t_n: <a, b>
E_1: *a* < 99	t_1: <50,50>
E_2: 99 ≤ *a* ≤ 999	t_2: <500, 50>
E_3: *a* > 999	t_3: <2000,50>
E_4: *b* < 1	t_4: <300, 0>
E_5: 1 ≤ *b* ≤ 100	t_5: <300, 60>
E_6: *b* > 100	t_6: <300, 200>

Income Tax Example

Given a program to calculate income tax based on the following marginal tax rates.

Income	**Tax**
Income < $10K	**no tax**
$10K ≤ Income < $20K	**10%**
$20K ≤ Income < $30K	**12%**
$30K ≤ Income < $40K	**15%**
Income ≥ $40K	**20%**

An example of using marginal tax rates: on an income of $25K, you pay $0 on the first $10K, 10% for income from $10-20K, and 12% for income from $20-25K, i.e., 0 + 1K + 0.6K = $1.6K.

Income Tax Equivalence Examples

Income < $10K	no tax
$10K ≤ Income < $20K	10%
$20K ≤ Income < $30K	12%
$30K ≤ Income < $40K	15%
Income ≥ $40K	20%

What are the equivalence classes for this problem?

E_1: Income < 0	(-∞, 0)	IMPLICIT
E_2: 0 ≤ Income < 10,000	[0, 10,000)	
E_3: 10,000 ≤ Income < 20,000	[10,000, 20,000)	
E_4: 20,000 ≤ Income < 30,000	[20,000, 30,000)	
E_5: 30,000 ≤ Income < 40,000	[30,000, 40,000)	
E_6: 40,000 ≤ Income	[40,000, +∞)	

Either notation is acceptable. Note that (,) are the same as <,> and [,] are the same as ≤,≥ in standard mathematical notation.

Income Tax ECP Test Set

E_1: Income < 0	**-100**
E_2: 0 ≤ Income < 10,000	**5000**
E_3: 10,000 ≤ Income < 20,000	**15000**
E_4: 20,000 ≤ Income < 30,000	**25000**
E_5: 30,000 ≤ Income < 40,000	**35000**
E_6: 40,000 ≤ Income	**50000**

T = {-100, 5000, 15000, 25000, 35000, 50000}

Boundary Value Analysis

Programmers make mistakes in process values at or near the boundaries of equivalence classes.

The "requirement" is

if (x ≤ 0) then return f1; else return f2;

But the program executes

if (x < 0) then return f1; else return f2;

x = 0 lies at the boundary between the equivalence classes…

Three-Value and Two-Value BVA
(IEEE 29119:4 section 5.2.3)

Three-value boundary value analysis
- **three test cases derived from each boundary**
- **based on values on the boundary and an incremental distance on <u>each side</u> of the boundary**

"Incremental distance" is defined as the smallest <u>significant</u> value for the data type.
- **integer → ± 1**
- **floating point with granularity of 0.01 → ± 0.01**

Two-value boundary value analysis
- ***a value on the boundary (closed boundaries)***
  - ***no need to test a value inside the boundary if this works***
- ***a value an incremental distance <u>outside</u> the boundary***

Boundary Values Example
(One Variable, Two-Value BVA)

Program P: integer X in [0, 100]

- **E_1: $X < 0$** **(-∞, -1]** **→ -1,0**
- **E_2: $0 \leq X \leq 100$** **[0, 100]** **→ -1,0 100,101**
- **E_3: $X > 100$** **[101, +∞)** **→ 100,101**

Test set T_b = {-1, 0, 100, 101}

Two test cases fewer than the three-value BVA example.

Boundary Values Example
(Two Variables, Two-Value BVA)

Program P
- **integer *a* in [99, 999]**
- **integer *b* in [1, 100]**

Boundaries for *a*: 99, 999
Test cases for *a:* T = {98, 99, 999, 1000}

Boundaries for *b*: 1, 100
Test cases for *b:* T = {0, 1, 100, 101}

But the inputs come in pairs…

Boundary Value Test Set

Pick a Valid "Partner" Value

a [99, 999], b [1, 100]	**t_n: <a, b>**
E_1: a: 98	**t_1: <98,50>**
E_2: a: 99	**t_2: <99, 50>**
E_3: a: 999	**t_3: <999, 50>**
E_4: a: 1000	**t_4: <1000, 50>**
E_5: b: 0	**t_5: <300, 0>**
E_6: b: 1	**t_6: <300, 1>**
E_7: b: 100	**t_7: <300, 100>**
E_8: b: 101	**t_8: <300, 101>**

Two-Value BVA Income Tax Test Set

What is the income tax test set based on boundary value analysis?

E_1: Income < 0	(-∞, -1]
E_2: 0 ≤ Income < 10,000	[0, 9,999]
E_3: 10,000 ≤ Income < 20,000	[10,000, 19,999]
E_4: 20,000 ≤ Income < 30,000	[20,000, 29,999]
E_5: 30,000 ≤ Income < 40,000	[30,000, 39,999]
E_6: 40,000 ≤ Income	[40,000, +∞)

T_b = { -1, 0,
9,999, 10,000,
19,999, 20,000,
29,999, 30,000,
39,999, 40,000 }

Incremental Distance

What is the "incremental distance"?

What is 10% for $9,999? $10,000? $10,001?

- **marginal tax rate**
- **$0 $0 $0.10**
- **Are outputs rounded or truncated by your programming language?**
- **Should just inside/outside be 9990, 10000, 10010?**

If the answer for an "off-by-one" calculation is the same as the correct calculation, is that a "good" test case?

Boundaries at "Infinity"

You could add a large positive income to test what happens when an input exceeds the "practical" maximum income that we might expect.

- **You should always talk to the customer about this kind of boundary.**

You could add an income near the max (or min) value that will fit into a variable.

  - **INT_MAX, INT_MIN, DBL_MAX, DBL_MIN in C and C++**

- **This boundary is set by the technology, not the customer or the problem.**
- **What happens when an integer value overflows? A floating point value?**

White-Box Testing

Structural testing (IEEE 24765)

- **testing that takes into account the internal mechanism of a system or component.**
  - **aka structure-based, white-box, glass-box testing**
  - **NOTE Types include branch testing, path testing, statement testing.**

Structure-based testing (IEEE 29119)

- **dynamic testing in which the tests are derived from an examination of the structure of the test item**

White-Box Testing Measures

Have you covered every statement in the program with at least one test case?

- **don't need to include syntactical markers, e.g., else, {, }, end**

Have you covered both branches of every decision in the program with at least one test case?

- **decisions include loops, ifs, switches**

Have you covered both branches of every condition in the program with at least one test case?

- **multiple conditions for compound predicates: AND, OR, …**

Measure – (fraction) percentage of statements, decisions, conditions covered

Test Coverage Example P1

```
1)   // Program P1
2)   integer x, y;
3)   input (x, y);
4)   if (x > 0 && y > 0)
5)   {
6)      y = y / x;
7)   }
8)   else
9)   {
10)     y = x ** 2;
11)  }
12)  output (x, y);
13)  end;
```

What is the statement coverage for T={t_1 = (1, 2)}?

- **do not count syntactical markers, including comments, {, }, else, end**

The decision coverage?

The condition coverage?

```
1)    // Program P1
2)    integer x, y;
3)    input (x, y);
4)    if (x > 0 && y > 0)
5)    {
6)       y = y / x;
7)    }
8)    else
9)    {
10)      y = x ** 2;
11)   }
12)   output (x, y);
13)   end;
```

Statement coverage for $T=\{t_1 = (1, 2)\}$?

- **Domain: {2, 3, 4, 6, 10, 12}**
- **t_1 traverses 2, 3, 4, 6, 12**
- **Statement Coverage = 5 / 6**

Decision coverage for T?

- **Decision: 4) if (x > 0 && y > 0)**
- **t_1 traverses the true branch**
- **Decision Coverage = 0 / 1**

Condition coverage for T?

- **Conditions: 4) x > 0; 4) y > 0**
- **t_1 traverses the true branch of the two conditions**
- **Condition Coverage = 0 / 2**

```
1)    // Program P1
2)    integer x, y;
3)    input (x, y);
4)    if (x > 0 && y > 0)
5)    {
6)       y = y / x;
7)    }
8)    else
9)    {
10)      y = x ** 2;
11)   }
12)   output (x, y);
13)   end;
```

Statement coverage: add t_2 = <-1, 2> to create T^A

- **t_2 traverses 2, 3, 4, 10, 12**
- **t_2 traverses line 10**
- **C_S = 6 / 6**

Decision coverage:

- **t_2 traverses the false branch of the decision at line 4**
- **t_1 and t_2 cover the decision at line 4**
- **C_D = 1 / 1**

```
1)    // Program P1
2)    integer x, y;
3)    input (x, y);
4)    if (x > 0 && y > 0)
5)    {
6)       y = y / x;
7)    }
8)    else
9)    {
10)      y = x ** 2;
11)   }
12)   output (x, y);
13)   end;
```

Condition coverage:

- **Conditions: 4) x > 0; 4) y > 0**
- **t_1, t_2 traverse the true and false branches of x>0**
- **t_1, t_2 traverse only the true branch of y>0**

T^A has C_C = 1/2

- **add t_3 = <1,-2> which traverses the false branch of y>0 to create T^B**

T^B has C_C = 2/2

```
1)    // Program P1
2)    integer x, y;
3)    input (x, y);
4)    if (x > 0 && y > 0)
5)    {
6)        y = y / x;
7)    }
8)    else
9)    {
10)       y = x ** 2;
11)   }
12)   output (x, y);
13)   end;
```

Could we have generated a smaller (minimal) test set to have statement, decision, and condition coverage?

T^B = {t_1=<1,2>, t_2=<-1, 2>, t_3=<1,-2>}

Consider

T^C = {t_1=<1,2>, t_2=<-1, -2>}

t_1: 2, 3, 4(tt=t), 6, 12
t_2: 2, 3, 4(ff=f), 10, 12

C_S = 6/6 = 100%
C_D = 1/1 = 100%
C_C = 2/2 = 100%

```
1)      integer A, B, C;
2)      input (A, B);
3)      if (A<-10 or A>10 or B<0 or B>5)
4)           output ("Boundary condition failure on inputs.");
5)      else // valid input
6)      {
7)           C = A * B;
8)           if (A < 0)
9)           {
10)               C = C + A + B;
11)               if (B > 2)
12)                    C = C + 3;
13)               C = C * C;
14)          } // end if (A<0)
15)          else
16)          {
17)               C = C - A - B;

18)               if (B <= 1 or C == 0)
19)                    C = B * C;
20)               else
21)                    C = B / C;
22)               C = C + 2;
23)          } // end else !(A<0)
24)          output (A, B, C);
25)     } // end else valid input
26)     return;
27)     end;
```

Example P2

What is the statement, decision, and condition coverage for P2, given
T = {t_1 = <-5, 2>,
t_2 = <3, 1>,
t_3 = <9, 3>}

```
1)      integer A, B, C;
2)      input (A, B);
3)      if (A<-10 or A>10 or B<0 or B>5)
4)           output ("Boundary condition failure on inputs.");
5)      else // valid input
6)      {
7)           C = A * B;
8)           if (A < 0)
9)           {
10)               C = C + A + B;
11)               if (B > 2)
12)                    C = C + 3;
13)               C = C * C;
14)          } // end if (A<0)
15)          else
16)          {
17)               C = C - A - B;
18)               if (B <= 1 or C == 0)
19)                    C = B * C;
20)               else
21)                    C = B / C;
22)               C = C + 2;
23)          } // end else !(A<0)
24)          output (A, B, C);
25)     } // end else valid input
26)     return;
27)     end;
```

What is the domain for statement coverage of P2?

Note: do not include syntactical markers such as comments, {, }, else, begin, end.

Excluding syntactical markers, there are 17 statements in P3:

D_S = {1, 2, 3, 4, 7, 8, 10, 11, 12, 13, 17, 18, 19, 21, 22, 24, 26}

$|D_S| = 17$

$t_n = \langle A, B \rangle$ $t_1 = \langle -5, 2 \rangle$ $t_2 = \langle 3, 1 \rangle$ $t_3 = \langle 9, 3 \rangle$

```
1)  integer A, B, C;
2)  input (A, B);
3)  if (A<-10 or A>10 or B<0 or B>5)
4)        output ("Boundary condition failure on inputs.");
5)  else // valid input
6)  {
7)        C = A * B;
8)        if (A < 0)
9)        {
10)             C = C + A + B;
11)             if (B > 2)
12)                  C = C + 3;
13)             C = C * C;
14)       } // end if (A<0)
15)       else
16)       {
17)             C = C - A - B;
18)             if (B <= 1 or C == 0)
19)                  C = B * C;
20)             else
21)                  C = B / C;
22)             C = C + 2;
23)       } // end else !(A<0)
24)       output (A, B, C);
25) } // end else valid input
26) return;
27) end;
```

1 2 3 FFFFF=F

1 2 3 FFFF=F

1 2 3 FFFFF =F

7 C=-10 8 T

7 C=3 8 F

7 C=27 8 F

10 C=-13 11 F 13

17 C=-1 18 TF=T 19

17 C=15 18 FF=F 21 22

22

24 26

24 26

24 26

$D_5 = \{1, 2, 3, (4), 7, 8, 10, 11, (12), 13, 17, 18, 19, 21, 22, 24, 26\}$

What is the statement coverage for T?

t_1 covers statements 1, 2, 3 (false), 7, 8 (true), 10, 11 (false), 13, 24, 26

t_2 covers statements 1, 2, 3 (false), 7, 8 (false), 17, 18 (true), 19, 22, 24, 26

- **7) C=3, 17) C=-1 → 18) -1!=0**

t_3 covers statements 1, 2, 3 (false), 7, 8 (false), 17, 18 (false), 21, 22, 24, 26

- **7) C=27, 17) C=15 → 18) 15!=0**

Coverage is

- **{~~1, 2, 3,~~ 4, ~~7, 8, 10, 11,~~ 12, ~~13, 17, 18, 19, 21, 22, 24, 26~~} = 15 / 17 = 88%**
- **did not cover statements 4 and 12**

```
1)          integer A, B, C;
2)          input (A, B);
3)          if (A<-10 or A>10 or B<0 or
B>5)
4)               output ("Boundary
condition failure on inputs.");
5)          else // valid input
6)          {
7)               C = A * B;
8)               if (A < 0)
9)               {
10)                   C = C + A + B;
11)                   if (B > 2)
12)                        C = C + 3;
13)                   C = C * C;
14)              } // end if (A<0)
15)              else
16)              {
17)                   C = C - A - B;
18)                   if (B <= 1 or C == 0)
19)                        C = B * C;
20)                   else
21)                        C = B / C;
22)                   C = C + 2;
23)              } // end else !(A<0)
24)              output (A, B, C);
25)         } // end else valid input
26)         return;
27)         end;
```

What test cases should you add to T to provide 100% statement coverage?

$T = \{t_1 = <-5, 2>, t_2 = <3, 1>, t_3 = <9, 3>\}$

Line 4 → decision at 3 true

- A<-10,A>10,B<0,B>5 – pick one

• $t_4 = <12, 3>$

Line 12

→ decision at 3 false

- A in [-10,10], B in [0,5]

→ decision at 8 true

- A<0

→ decision at 11 true

- B>2

• $t_5 = <-1, 3>$

Add test cases t_4 and t_5 to create T_A

Decision Coverage

```
1)        integer A, B, C;
2)        input (A, B);
3)        if (A<-10 or A>10 or B<0 or
B>5)
4)             output ("Boundary
condition failure on inputs.");
5)        else // valid input
6)        {
7)             C = A * B;
8)             if (A < 0)
9)             {
10)                 C = C + A + B;
11)                 if (B > 2)
12)                      C = C + 3;
13)                 C = C * C;
14)            } // end if (A<0)
15)            else
16)            {
17)                 C = C - A - B;
18)                 if (B <= 1 or C == 0)
19)                      C = B * C;
20)                 else
21)                      C = B / C;
22)                 C = C + 2;
23)            } // end else !(A<0)
24)            output (A, B, C);
25)       } // end else valid input
26)       return;
27)       end;
```

What is the domain for decision coverage of P2?

3) if (A<0 or A>10 or B<0 or B>5)

8) if (A<0)

11) if (B>2)

18) if (B<=1 or C==0)

4 decisions, $|D_D| = 4$

What is the decision coverage for T?

	t_1	t_2	t_3
decision @3	false	false	false
decision @8	true	false	false
decision @11	false	--	--
decision @18	--	true	false

Only the decisions at line 8 and 18 are covered (both true and false branches taken): 2 / 4 = 50%

Condition Coverage

```
1)        integer A, B, C;
2)        input (A, B);
3)        if (A<-10 or A>10 or B<0 or B>5)
4)             output ("Boundary condition
failure on inputs.");
5)        else // valid input
6)        {
7)             C = A * B;
8)             if (A < 0)
9)             {
10)                 C = C + A + B;
11)                 if (B > 2)
12)                      C = C + 3;
13)                 C = C * C;
14)            } // end if (A<0)
15)            else
16)            {
17)                 C = C - A - B;
18)                 if (B <= 1 or C == 0)
19)                      C = B * C;
20)                 else
21)                      C = B / C;
22)                 C = C + 2;
23)            } // end else !(A<0)
24)            output (A, B, C);
25)       } // end else valid input
26)       return;
27)       end;
```

What is the domain for condition coverage of P3?

3) A<-10
3) A>10
3) B<0
3) B>5
8) A<0
11) B>2
18) B<=1
18) C==0

8 conditions in P3 (on lines 3, 8, 11, 18).
$| D_C | = 8$

What is the condition coverage for T?

	t_1	t_2	t_3
3) A<-10	false	false	false
3) A>10	false	false	false
3) B<0	false	false	false
3) B>5	false	false	false
8) A<0	true	false	false
11) B>2	false	--	--
18) B<=1	--	true	false
18) C==0	--	false	false

- 3) A<-10 is not covered (no true)
- 3) A>10 is not covered (no true)
- 3) B<0 is not covered (no true)
- 3) B>5 is not covered (no true)
- 8) A<0 is covered
- 11) B>2 is not covered (no true)
- 18) B<=1 is covered
- 18) C==0 is not covered (no true)

Condition coverage is 2 / 8 = 25%

What test cases should you add to T to provide 100% condition coverage?

- **Many correct answers that satisfy the criteria…**

Try T^A = {t_1 = <-5, 2>, t_2 = <3, 1>, t_3 = <9, 3>,
t_4:<12, 3>, t_5:<-1, 3>, t_6:<-11, -5>, t_7:<0, -1>, t_8:<0, 6>, t_9:<0, 0>}

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9
3) A<-10	false	false	false	false	false	true	false	false	false
3) A>10	false	false	false	true	false	false	false	false	false
3) B<0	false	false	false	false	false	false	true	false	false
3) B>5	false	false	false	false	false	false	false	true	false
8) A<0	true	false	false	--	true	--	--	--	false
11) B>2	false	--	--	--	true	--	--	--	--
18) B<=1	--	true	false	--	--	--	--	--	true
18) C==0	--	false	false	--	--	--	--	--	true

For t_9, 3) A in [-10,10], B in [0,5], 8) A≥0, 18) C==0

- **7) C=A*B, 17) C=C-A-B → try <0,0>**

Program P3

```
1) integer A, B, C;
2)    input (A, B);
3)    while (A≥0)
4)    {
5)       C = A + B;
6)       if (B<0)
7)          C = 0;
8)       input (A, B);
9)    }
10)   print A,B,C;
11)   end;
```

Example P3

What is the statement, decision, and condition coverage for P3, given
T = {t_1 = <0, 0>,
t_2 = <1, 1>,
t_3 = <-2, 2>}

$t_1 = \langle 0, 0 \rangle$ $t_2 = \langle 1, 1 \rangle$ $t_3 = \langle -2, 2 \rangle$

Program P3

```
1) integer A, B, C;
2)     input (A, B);
3)     while (A≥0)
4)     {
5)         C = A + B;
6)         if (B<0)
7)             C = 0;
8)         input (A, B);
9)     }
10)    print A,B,C;
11)    end;
```

1
2 A=0,B=0
3 t
5
6 f
8 A=1, B=1

3 t
5
6 f
8 A=-2,B=2

3 f
10

$D_S = \{ 1, 2, 3, 4, 5, 6, 7, 8, 10 \}$
$|D_S| = 8$ $C_S = 7/8$

$D_D = \{ 3, 6 \}$
$|D_D| = 2$ $C_D = 1/2$

$D_C = \{ 3) A \geq 0,\ 6) B < 0 \}$
$|D_C| = 2$ $C_C = 1/2$

Issues to Ignore for Now

We are not considering a number of issues when we discuss control flow coverage yet...

- **dead code**
- **infeasible decisions and conditions**
- **short circuit evaluation**
- **coupled conditions (non-singular predicates)**

Summing Up Testing

Test set vs test case

Test driven development (TDD)

Black-box testing (specification-based)

- **Requirements coverage**
- **Equivalence class partitioning**
- **Two-value boundary value analysis**

White-box testing (structural)

- **Statement coverage**
- **Decision coverage**
- **Condition coverage**

Questions?

Dr. Mark C. Paulk
University of Texas at Dallas
ECSS 3.610, EC31
800 W. Campbell Road
Richardson, TX 75080-3021

Mark.Paulk@utdallas.edu
Mark.Paulk@ieee.org

https://personal.utdallas.edu/~mcp130030/

TDD References

K. Beck, Test-Driven Development, 2003.

K. Beck and C. Andres, Extreme Programming Explained: Embrace Change, 2nd Edition, 2004.

H. Kniberg, Scrum and XP from the Trenches, 2nd Edition, 2015.

C. Larman, Applying UML and Patterns, Third Edition, 2005.

R.C. Martin, "Professionalism and Test-Driven Development," IEEE Software, May/June 2007.

E.M. Maximilien and L. Williams, "Assessing Test-Driven Development at IBM," ICSE, 2003.

J. Rasmusson, "Introducing XP Into Greenfield Projects: Lessons Learned," IEEE Software, May/June 2003.