FUNCTIONAL TESTING

F. Tip and M. Weintraub

ACKNOWLEDGEMENTS

Thanks go to Andreas Zeller for allowing incorporation of his materials

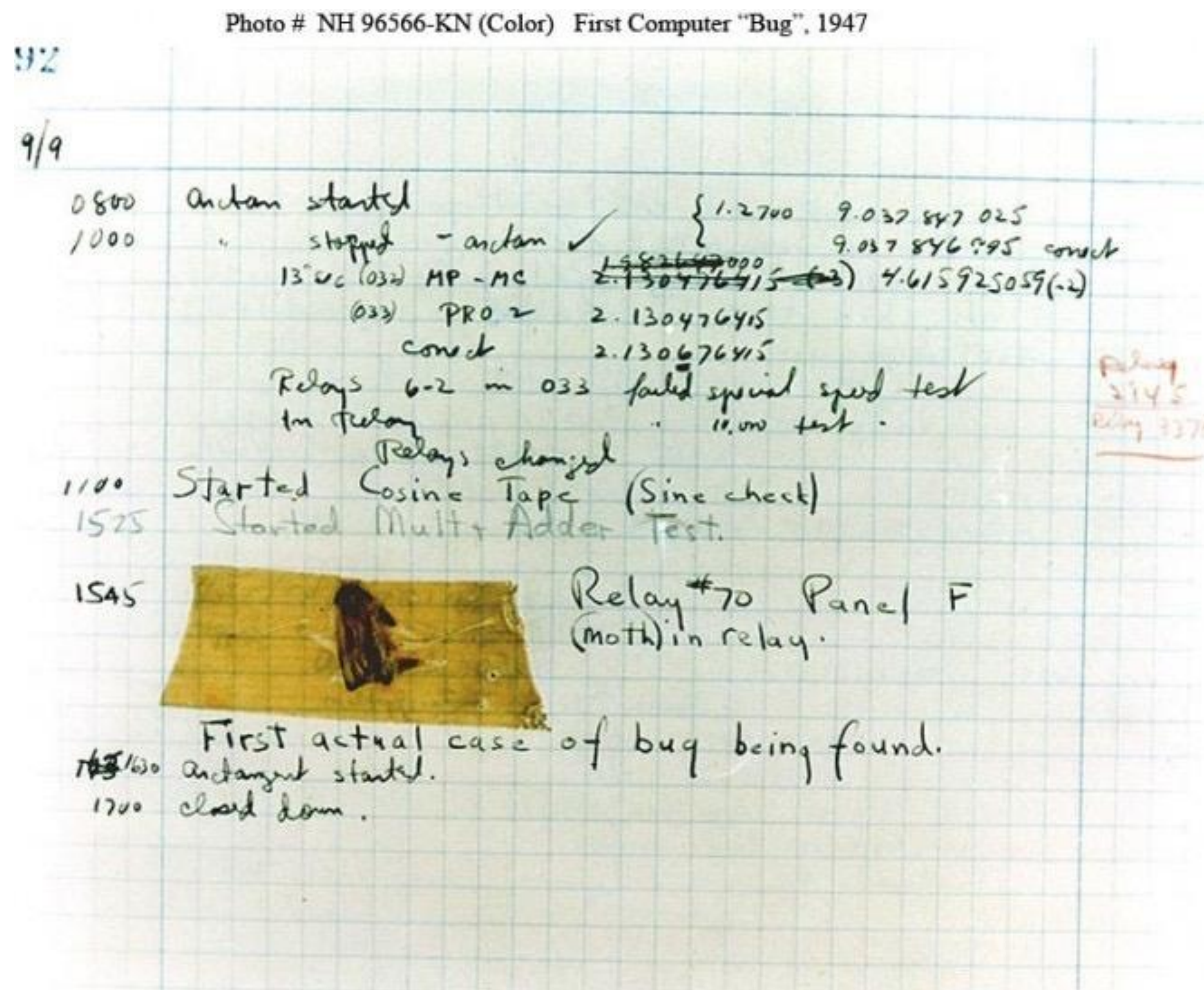
HOW TO TELL IF A SYSTEM MEETS EXPECTATIONS?

Two options:

1. **testing**: execute parts of the program and observe if unexpected behaviors occur

2. **formal verification**: exhaustively enumerate all states of the system, and try to prove that properties to be verified hold in each state.
   - Various techniques, e.g. model checking

THE FIRST COMPUTER BUG (1947)



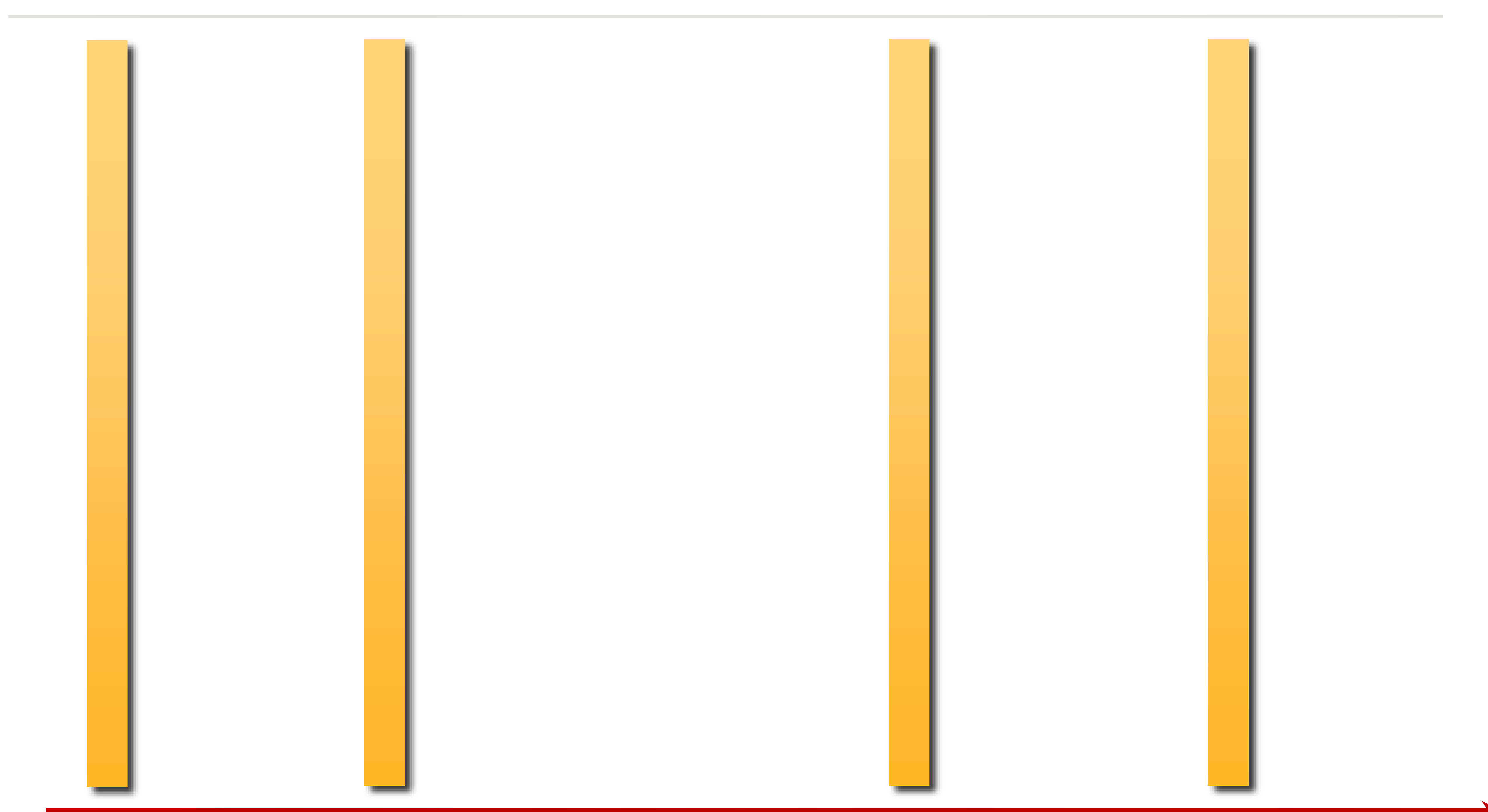
The First "Computer Bug". Moth found trapped between points at Relay # 70, Panel F, of the Mark II Aiken Relay Calculator while it was being tested at Harvard University, 9 September 1947.

The operators affixed the moth to the computer log, with the entry: "First actual case of bug being found". They put out the word that they had "debugged" the machine, thus introducing the term "debugging a comp...uter program".

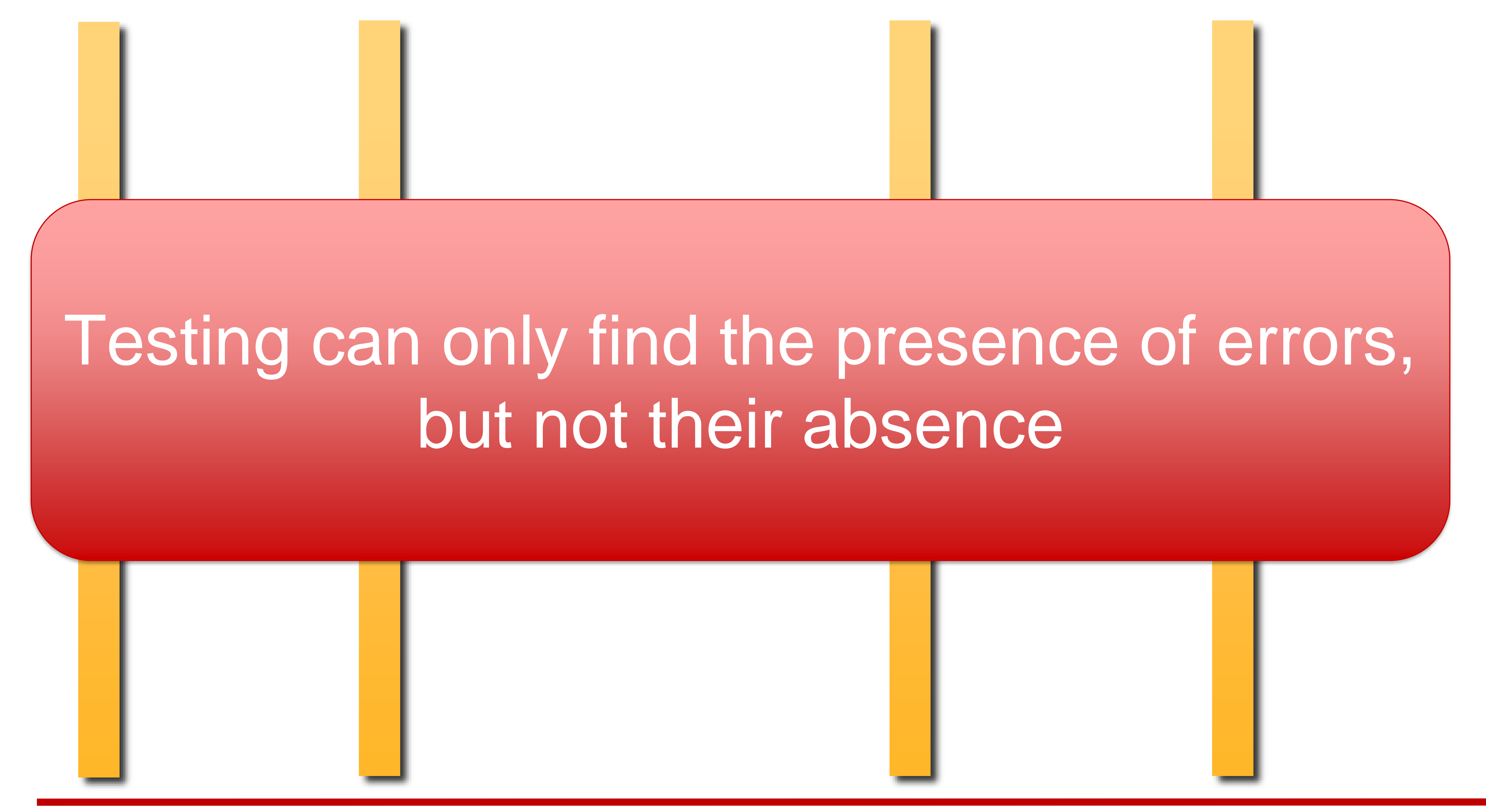
In 1988, the log, with the moth still taped by the entry, was in the Naval Surface Warfare Center Computer Museum at Dahlgren, Virginia. The log is now housed at the Smithsonian Institution’s National Museum of American History, who have corrected the date from 1945 to 1947. Courtesy of the Naval Surface Warfare Center, Dahlgren, VA., 1988. NHHC Photograph Collection, NH 96566-KN (Color).

From https://www.facebook.com/navalhistory/photos/a.77106563343.78834.76845133343/10153057920928344/

WHAT TO TEST?



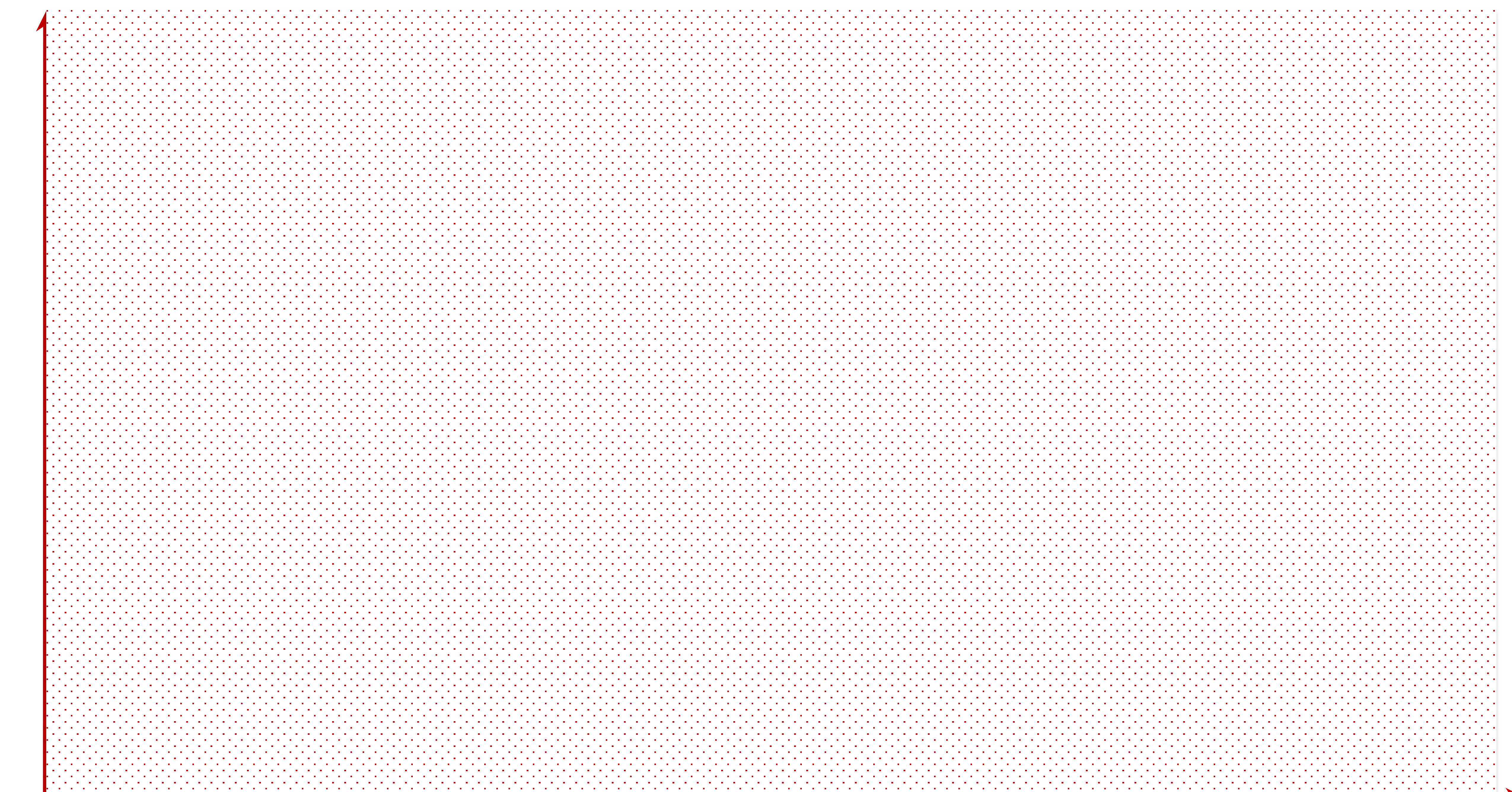
DIJKSTRA'S CURSE



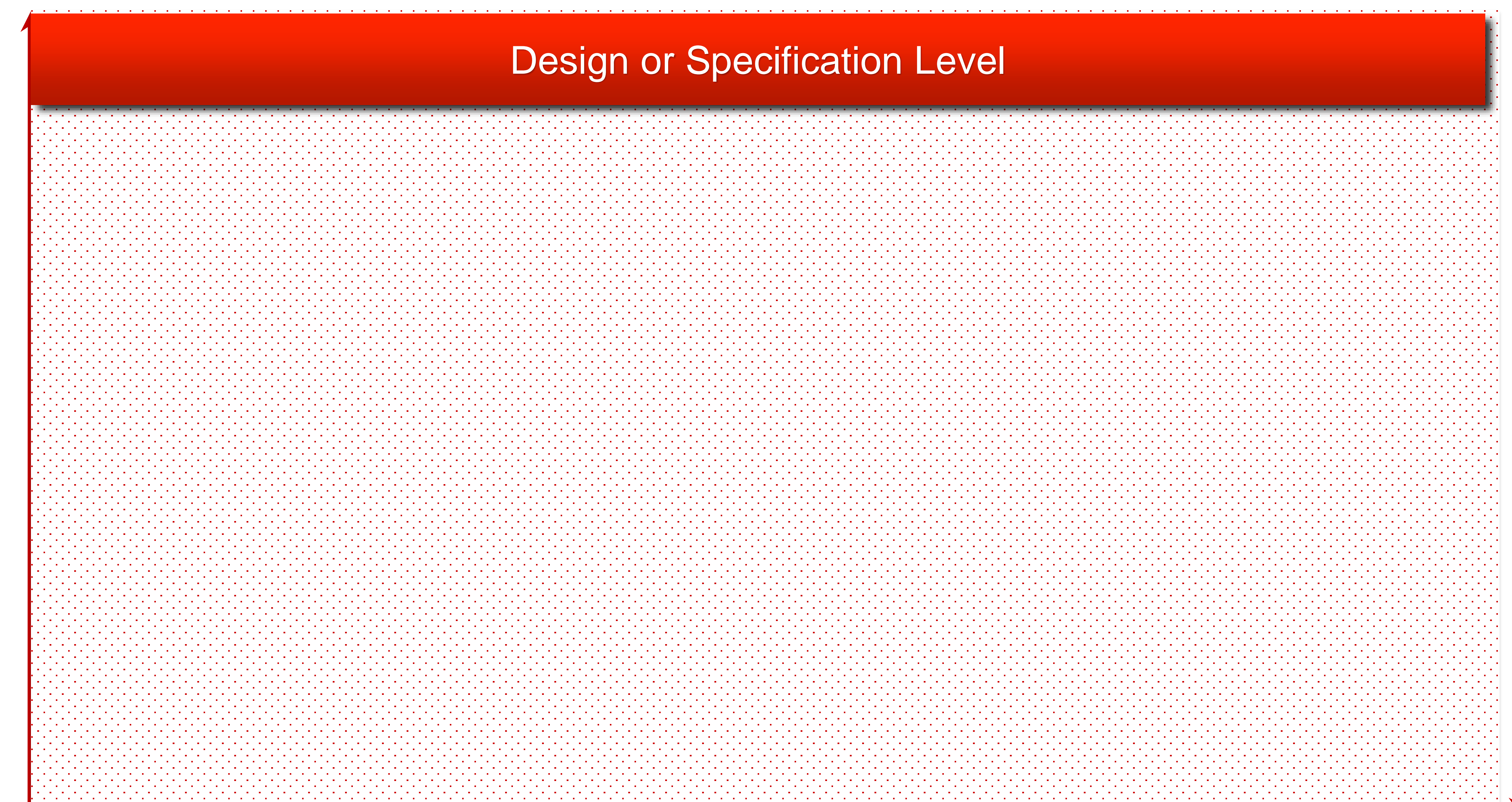
FORMAL VERIFICATION

Configurations

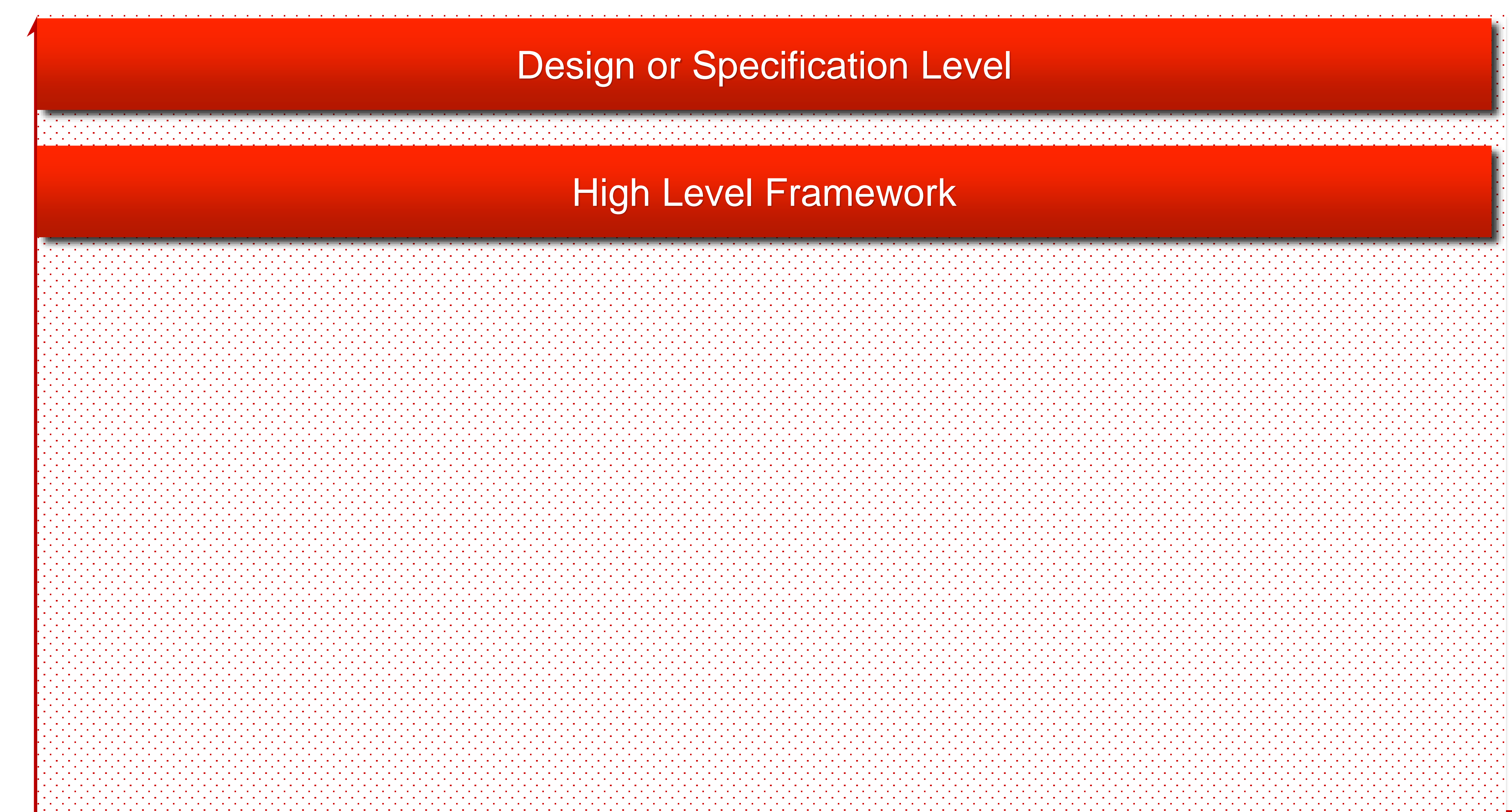
FORMAL VERIFICATION



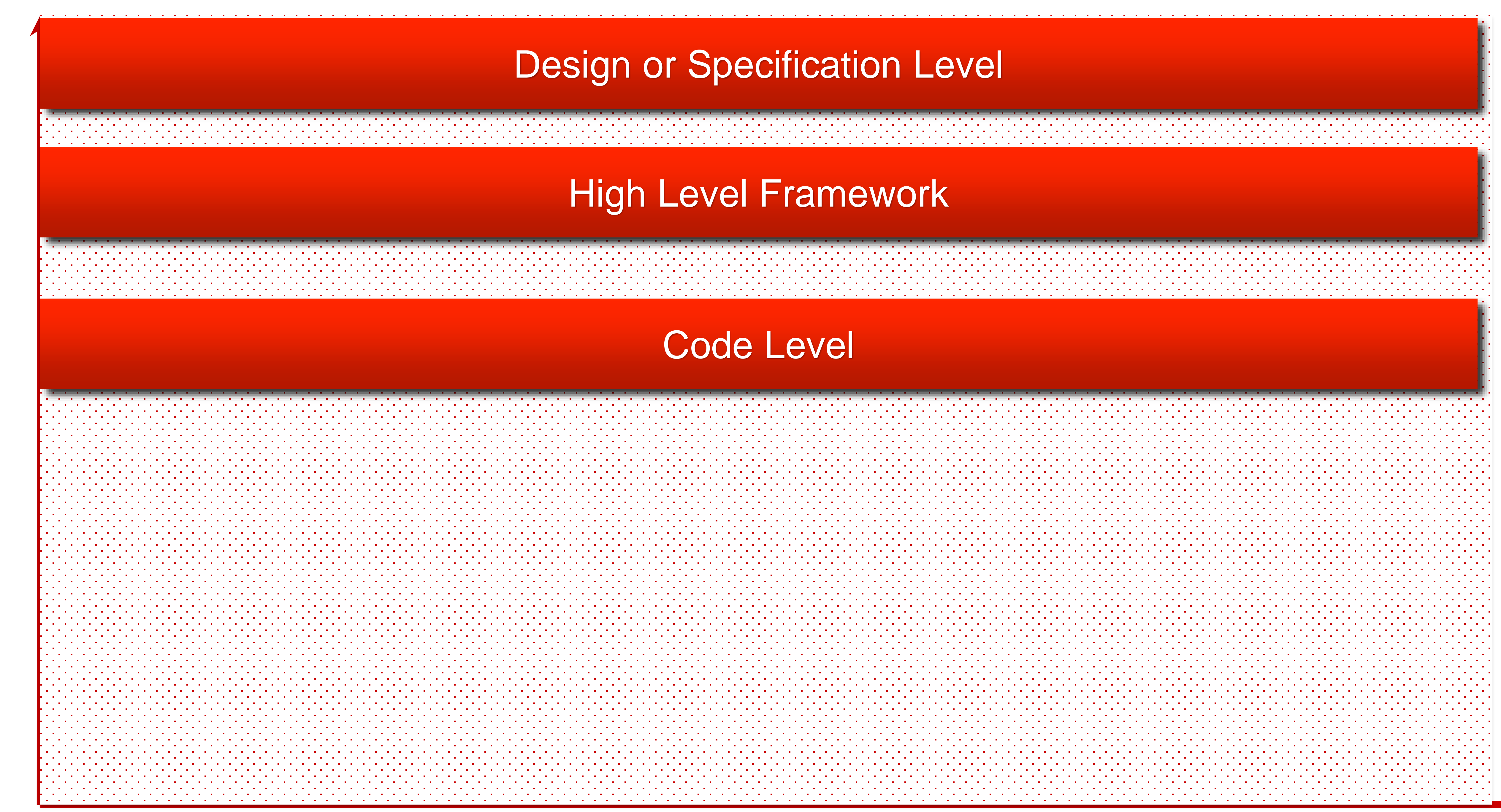
FORMAL VERIFICATION



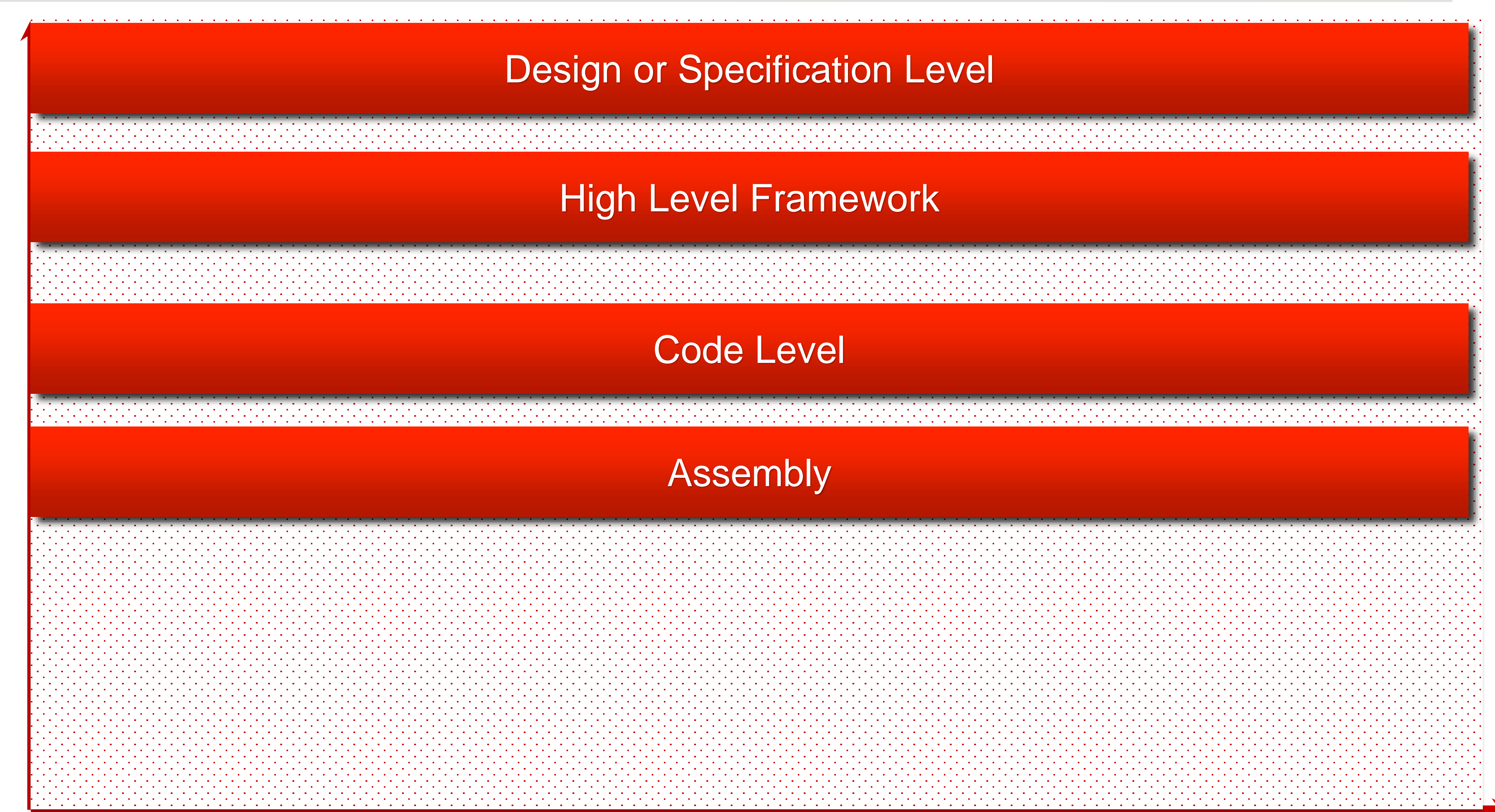
FORMAL VERIFICATION



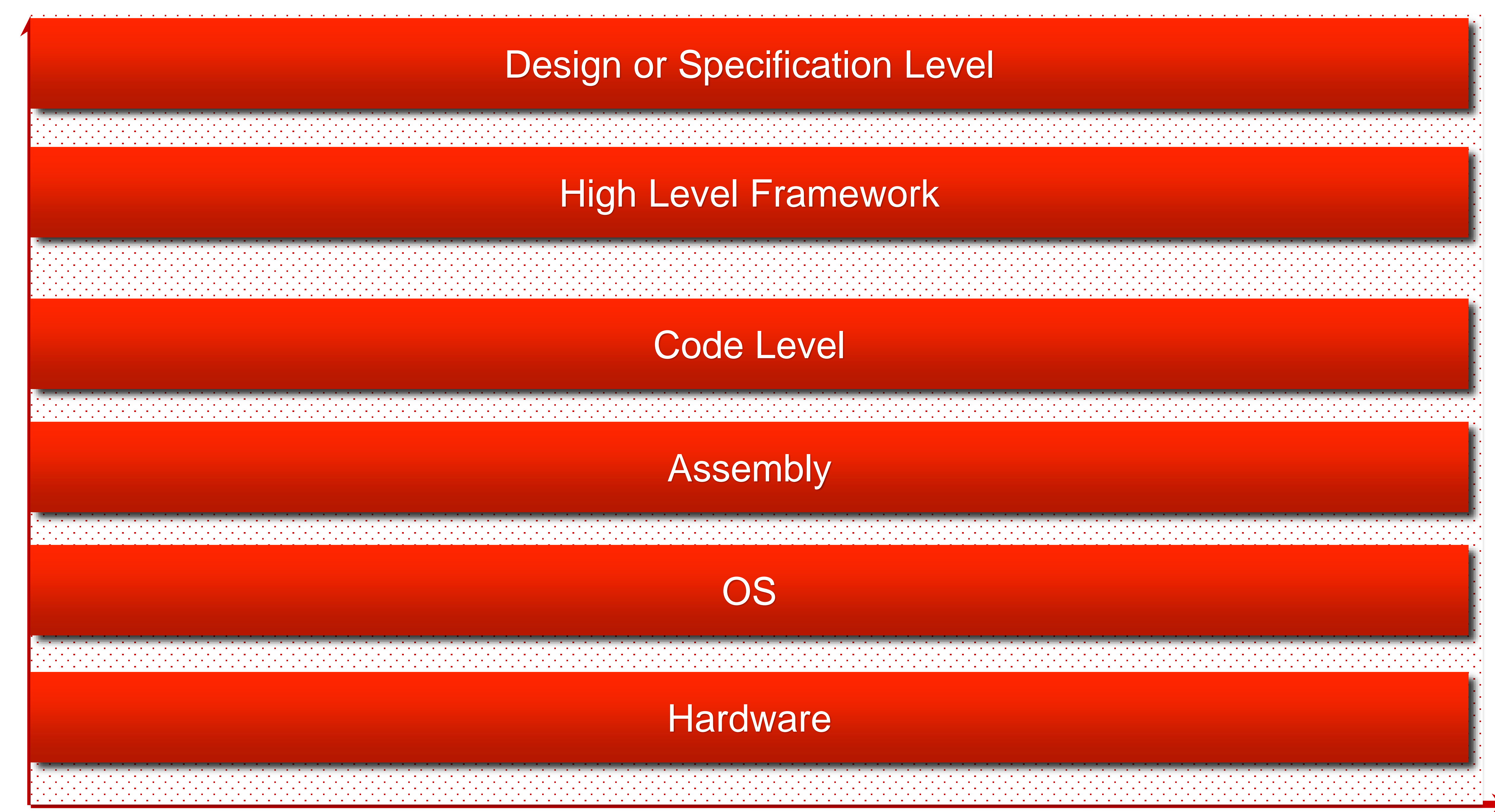
FORMAL VERIFICATION



FORMAL VERIFICATION



FORMAL VERIFICATION

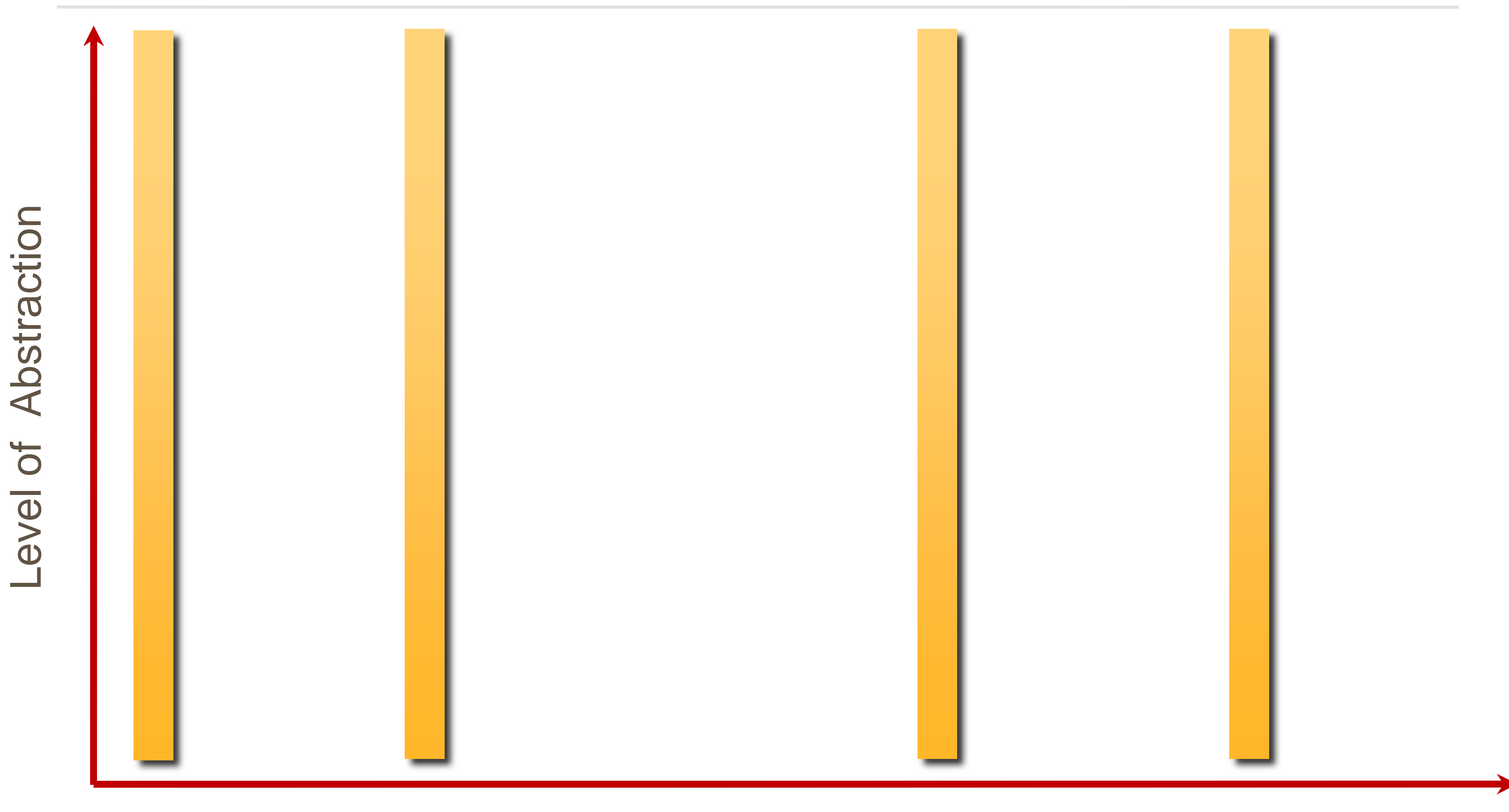


ZELLER'S COROLLARY

Level of Abstraction

Configurations

BACK TO TESTING: HOW TO COVER AS MUCH OF THE SPACE AS POSSIBLE?

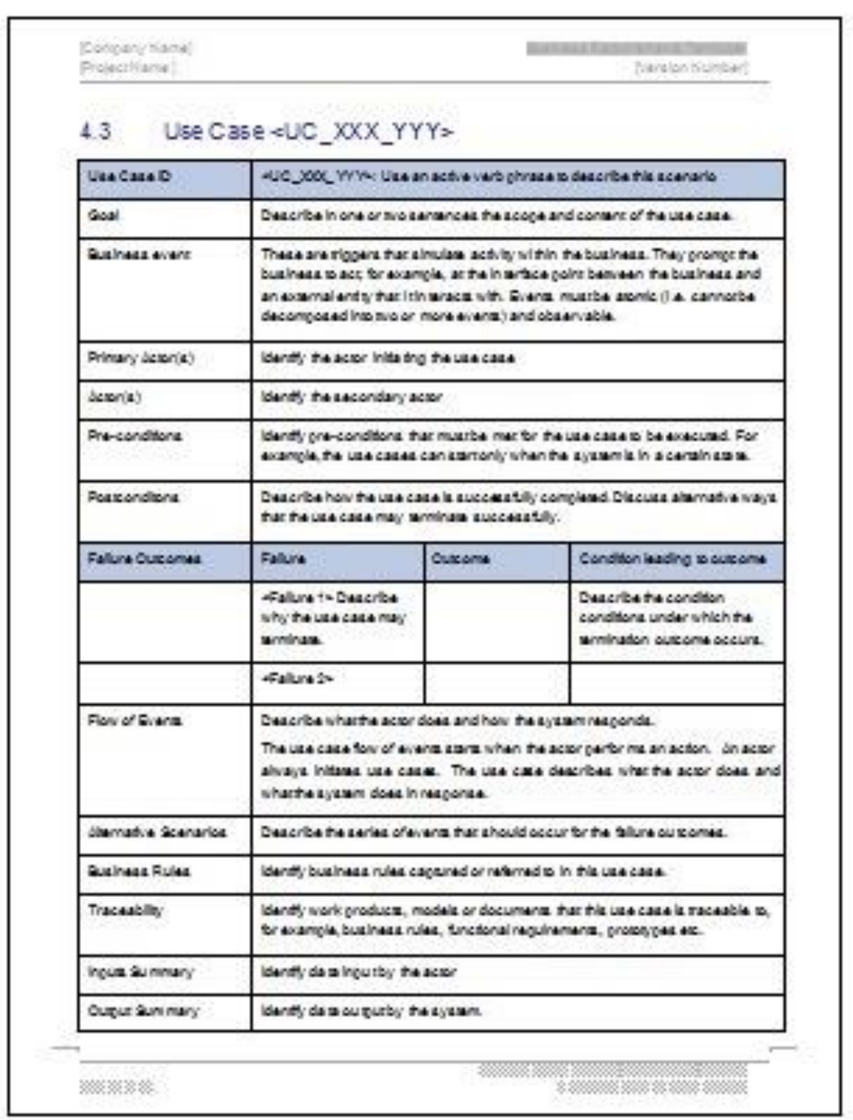


FUNCTIONAL TESTING – AKA BLACK BOX TESTING

WHITE BOX TESTING IS WHERE YOU TEST BASED ON KNOWING WHAT'S INSIDE THE MODULE

IF WE CANNOT KNOW THE CODE INSIDE, AGAINST WHAT DO WE WRITE TESTS?

IF WE CANNOT KNOW THE CODE INSIDE, AGAINST WHAT DO WE WRITE TESTS?



[Company Name]
[Project Name]

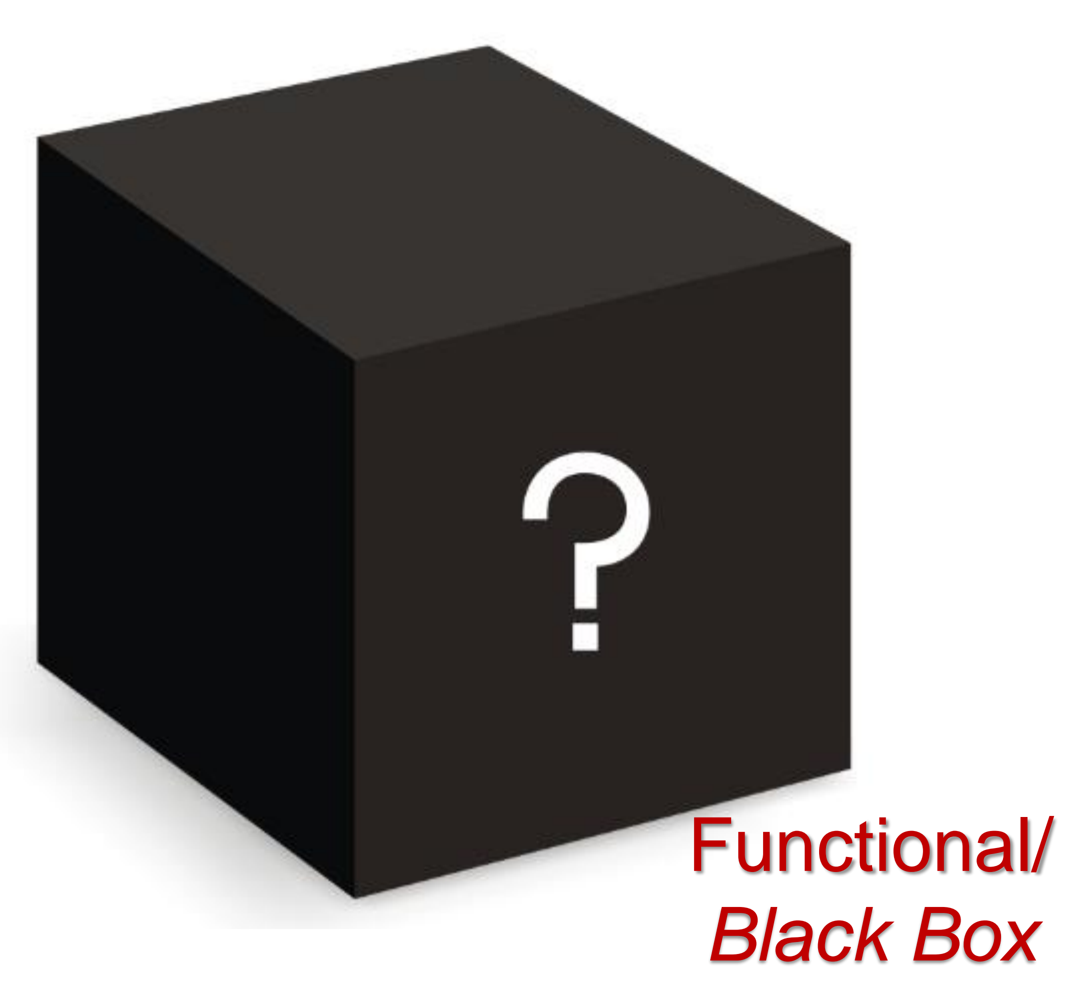
[Version Number]

4.3 Use Case <UC_XXX_YYY>

Use Case ID	<UC_XXX_YYY>: Use an active verb phrase to describe this scenario		
Goal	Describe in one or two sentences the scope and content of the use case.		
Business event	These are triggers that simulate activity within the business. They prompt the business to act, for example, at the interface point between the business and an external entity that it interacts with. Events must be atomic (i.e. cannot be decomposed into two or more events) and observable.		
Primary Actor(s)	Identify the actor initiating the use case		
Actor(s)	Identify the secondary actor		
Pre-conditions	Identify pre-conditions that must be met for the use case to be executed. For example, the use cases can start only when the system is in a certain state.		
Postconditions	Describe how the use case is successfully completed. Discuss alternative ways that the use case may terminate successfully.		
Failure Outcomes	Failure	Outcome	Condition leading to outcome
	<Failure 1> Describe why the use case may terminate.		Describe the condition conditions under which the termination outcome occurs.
	<Failure 2>		
Flow of Events	Describe what the actor does and how the system responds. The use case flow of events starts when the actor performs an action. An actor always initiates use cases. The use case describes what the actor does and what the system does in response.		
Alternative Scenarios	Describe the series of events that should occur for the failure outcomes.		
Business Rules	Identify business rules captured or referred to in this use case.		
Traceability	Identify work products, models or documents that this use case is traceable to, for example, business rules, functional requirements, prototypes etc.		
Input Summary	Identify data input by the actor		
Output Summary	Identify data output by the system.		

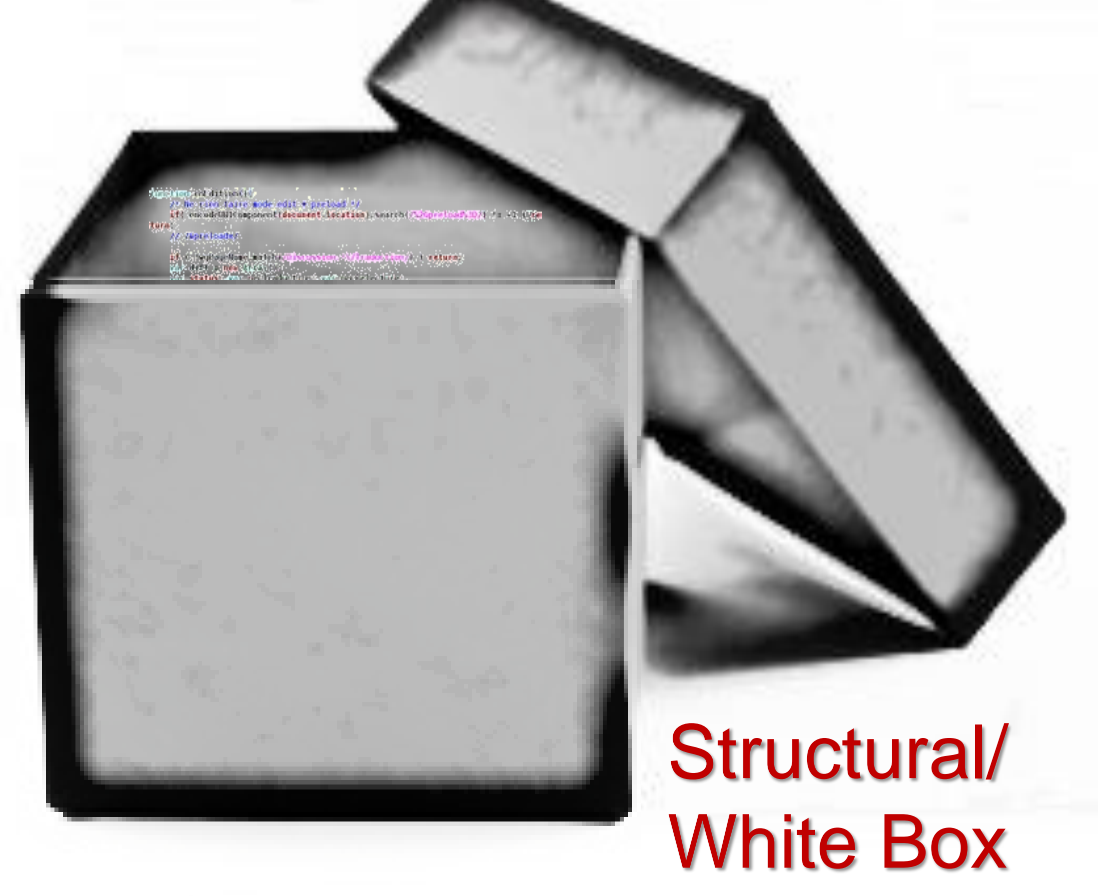
Specifications

TESTING TACTICS



Functional/
Black Box

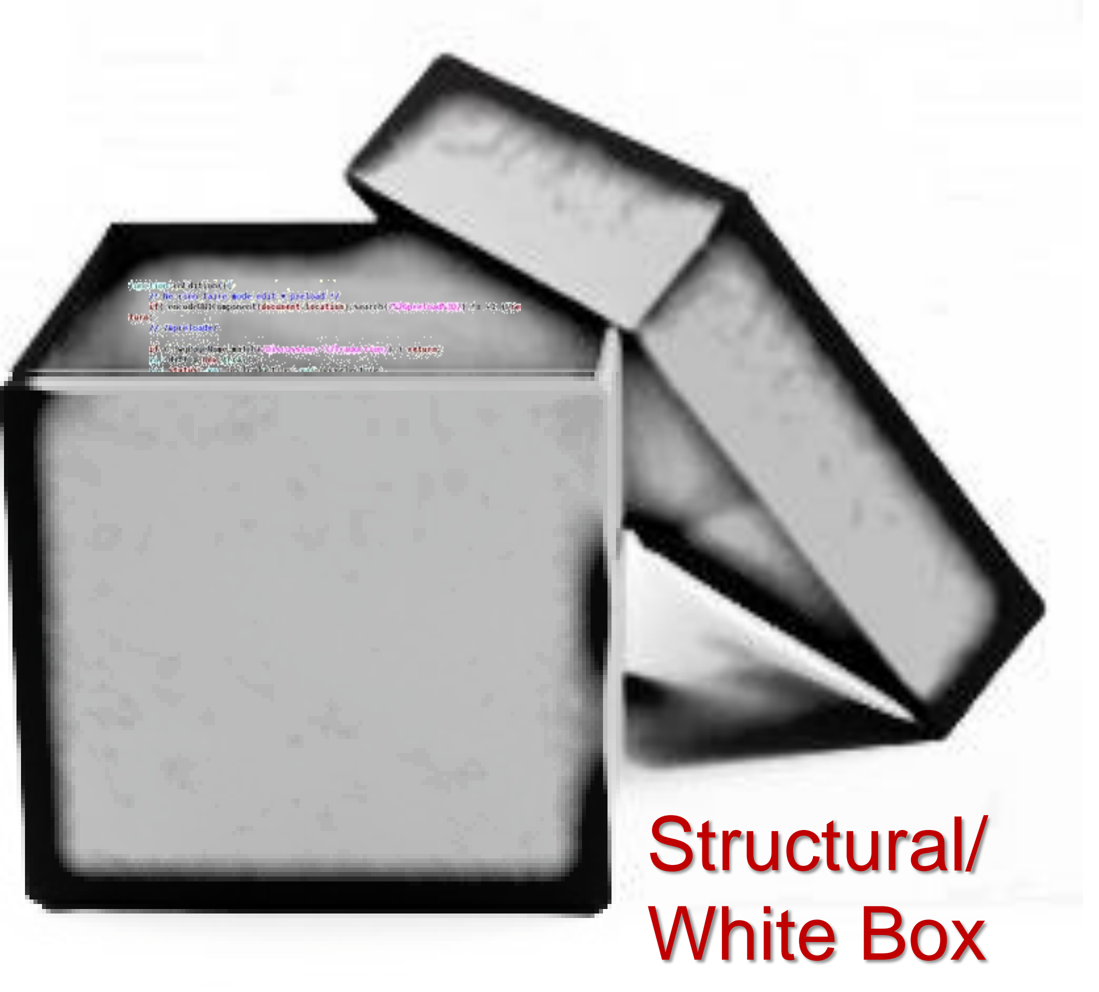
- Tests based on *spec*
- Test covers as much *specified* behavior as possible



Structural/
White Box

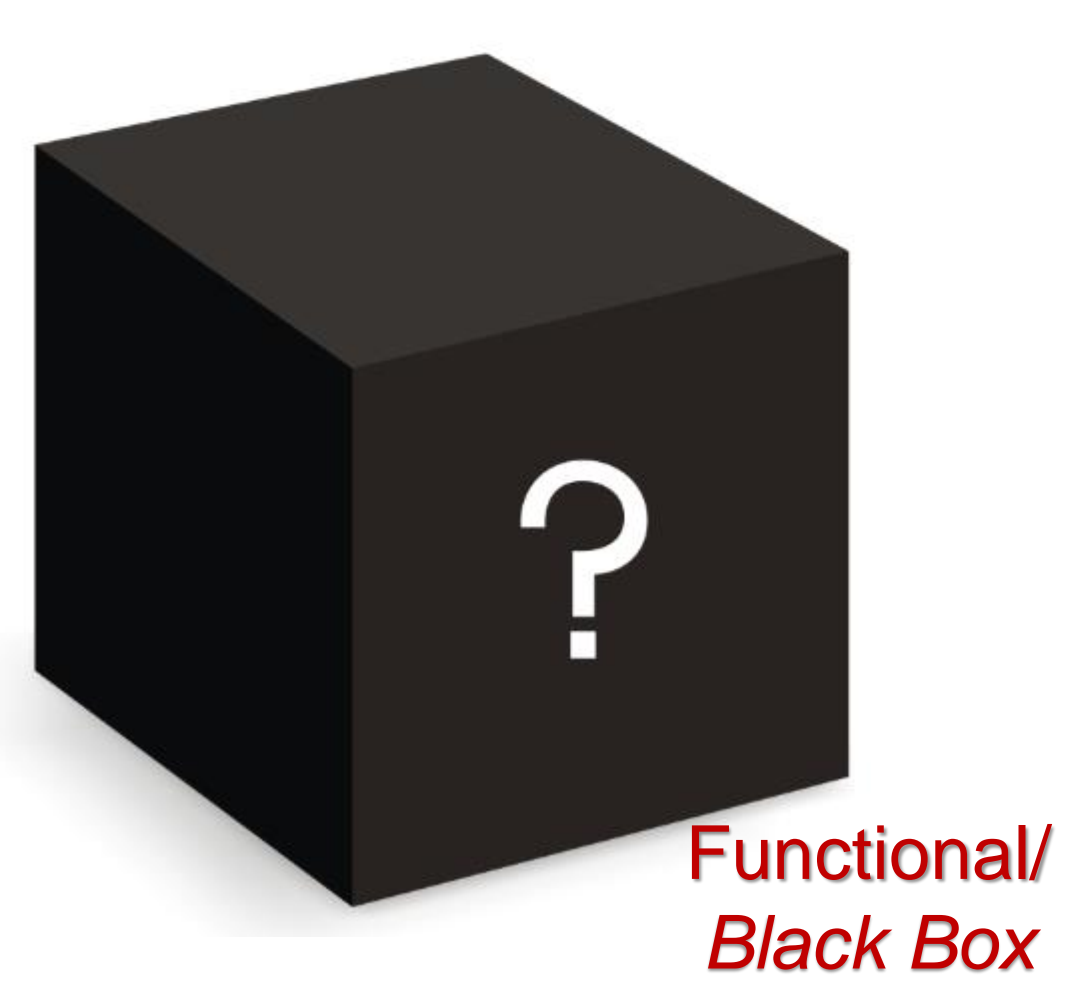
- Tests based on *code*
- Test covers as much *implemented* behavior as possible

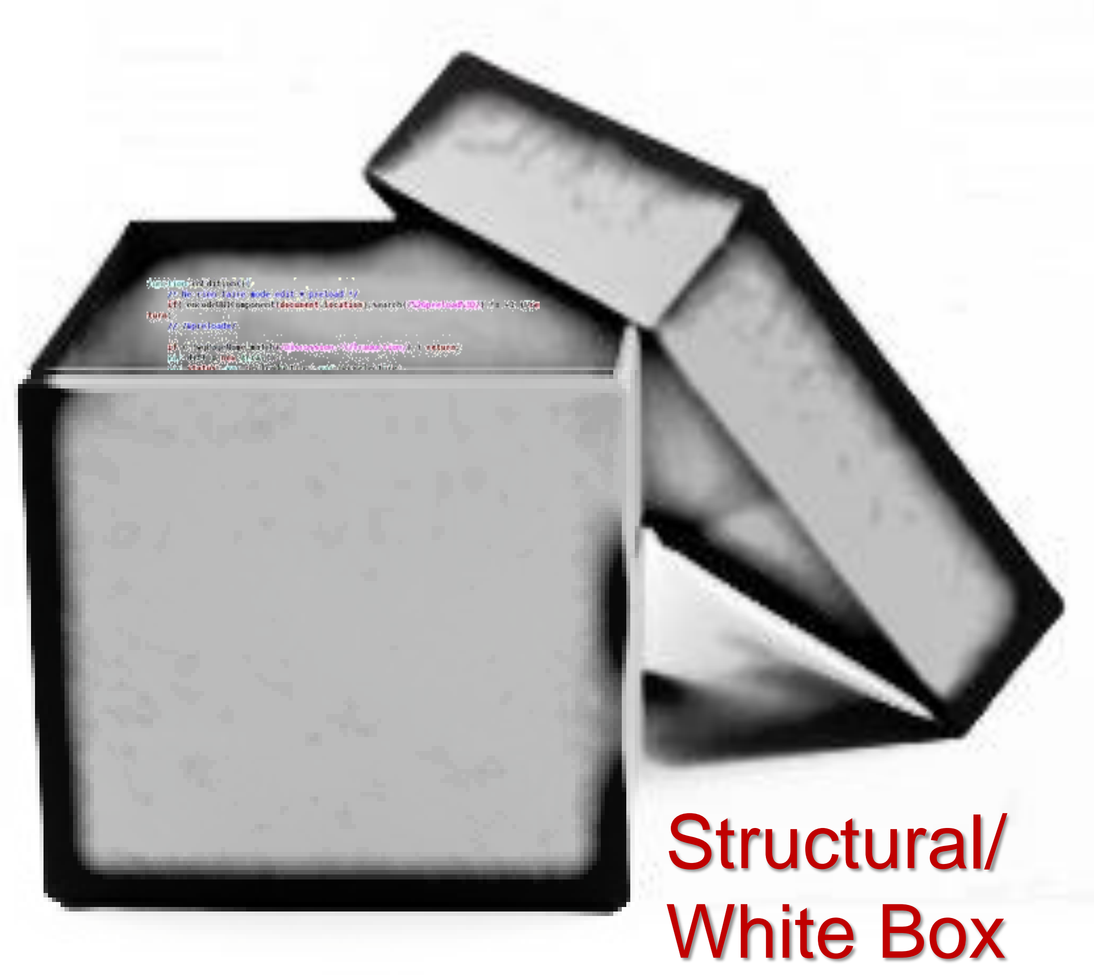
WHY DO FUNCTIONAL TESTING?



1. Program code not necessary
2. Early functional test design has benefits
   1. Reveals spec problems
   2. Assesses testability
   3. Gives additional explanation of spec
   4. May even serve as spec, as in XP

WHY DO FUNCTIONAL TESTING?





- Best for *missing logic* defects
- Common problem:
  Some program logic was simply forgotten
  Structural testing would not focus on code that is not there

- Applies at all granularity levels
  - unit tests
  - integration tests
  - system tests
  - regression tests

RANDOM TESTING

- Pick possible inputs uniformly
- Avoids designer bias

A real problem: The test designer can make the same logical mistakes and bad assumptions as the program designer (especially if they are the same person)

- But treats all inputs as equally valuable

PUNKTE
0

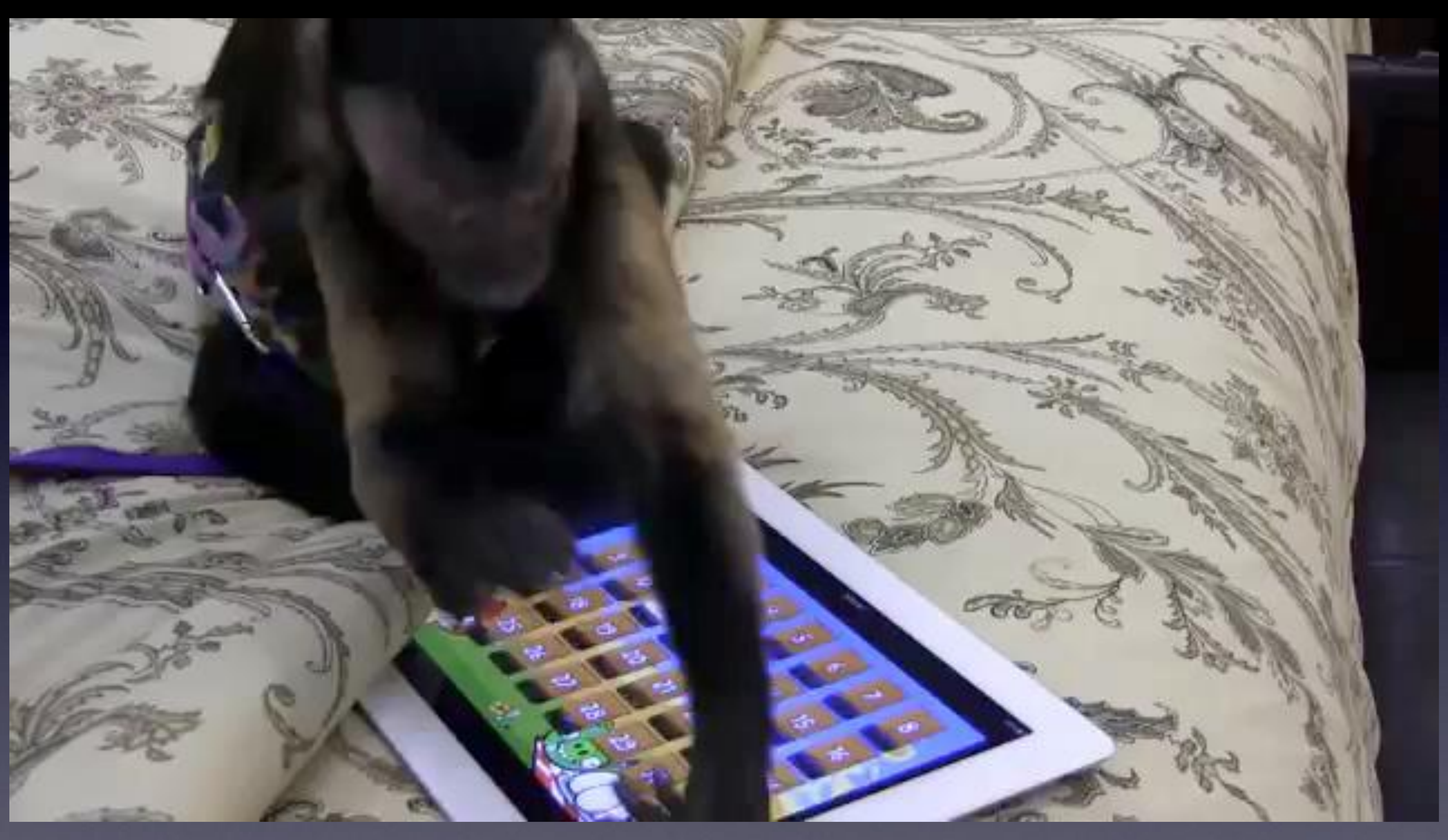
Angle
Force

PUNKTE
0

INFINITE MONKEY THEOREM

INFINITE MONKEY THEOREM

If you put enough monkeys in front of typewriters and give them enough time, you eventually will get Shakespeare



Youtube

Angle
2^{32} = 4.294.967.296
different values
Force
2^{32} = 4.294.967.296
different values

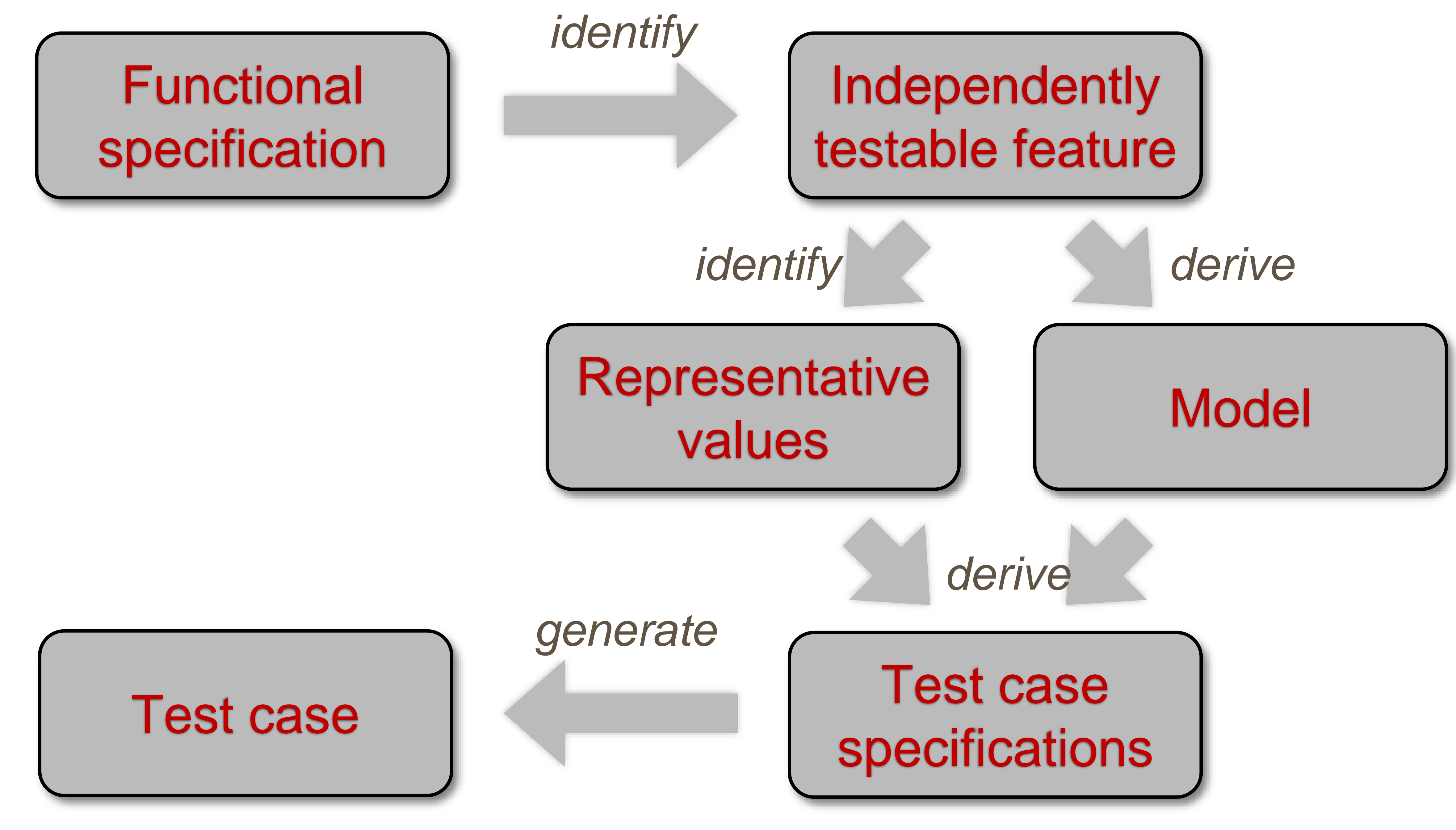
18,446,744,073,709,551,616 COMBINATIONS

$$total\ number\ of\ trials = 2^{32} * 2^{32} = 2^{64}$$
$$= 18{,}446{,}744{,}073{,}709{,}551{,}616$$

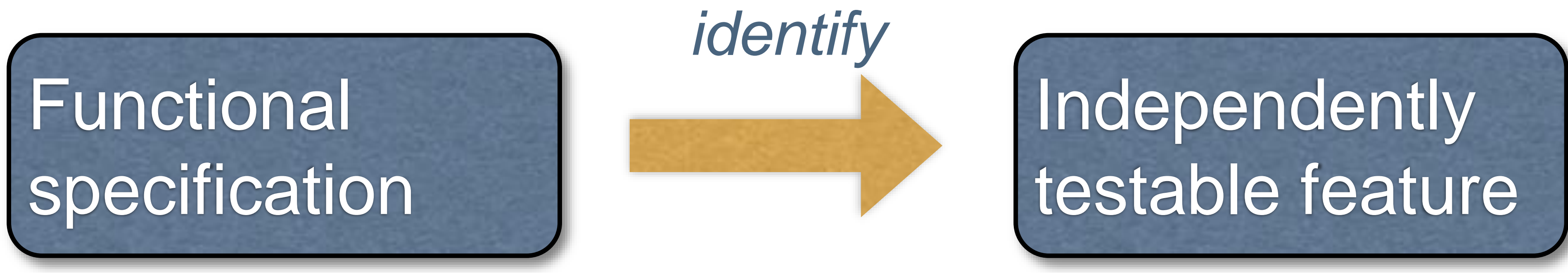
THE ALTERNATIVE: COMPUTER SCIENCE APPROACHES

Computer scientists are smart, and they can systematically test and analyze programs.

SYSTEMATIC FUNCTIONAL TESTING



TESTABLE FEATURES



- Decompose system into *independently testable features* (ITF)
- An ITF need not correspond to units or subsystems of the software
- For system testing, ITFs are exposed through user interfaces or APIs

WHAT ARE THE INDEPENDENTLY TESTABLE FEATURES?

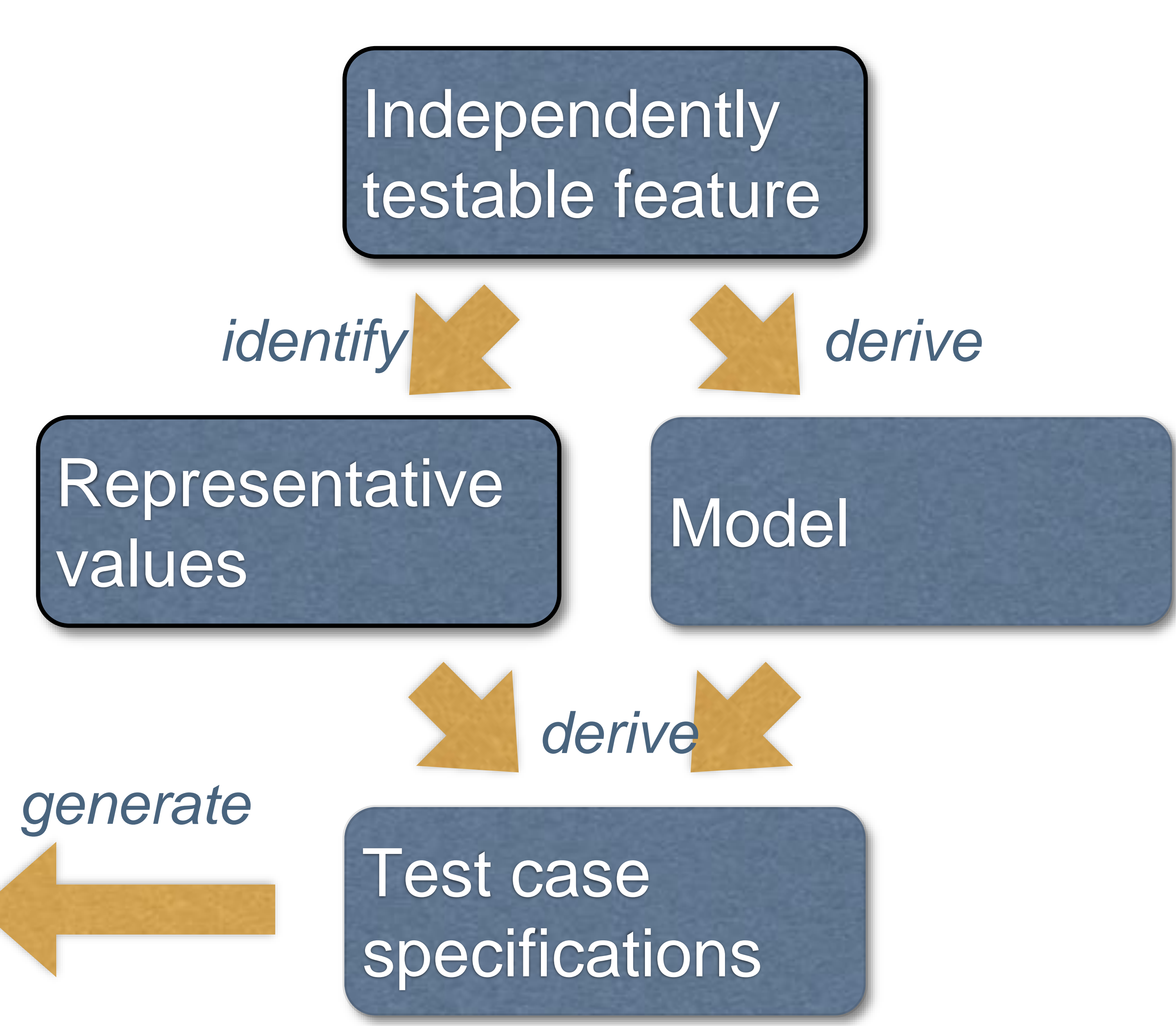
```
class Roots {
    // Solve ax2 + bx + c = 0
    public roots(double a, double b, double c)
    { … }

    // Result: values for x
    double root_one, root_two;
}
```

EVERY FUNCTION IS AN INDEPENDENTLY TESTABLE FEATURE

- Consider a multi-function calculator
- What are the independently testable features?

REPRESENTATIVE VALUES

- Try to select inputs that are *especially valuable*
- Usually by choosing *representatives* of *equivalence classes* that are apt to fail often or not at all



LIKE FINDING NEEDLES IN A HAYSTACK

To find bugs systematically, we need to find out *what makes certain inputs or behaviors special*

SYSTEMATIC PARTITION TESTING

■ Failure (valuable test case)

□ No failure

Failures are sparse in some regions of possible inputs ...

... but dense in other

The space of possible input values (the haystack)

If we systematically test some cases from each part, we will include the dense parts

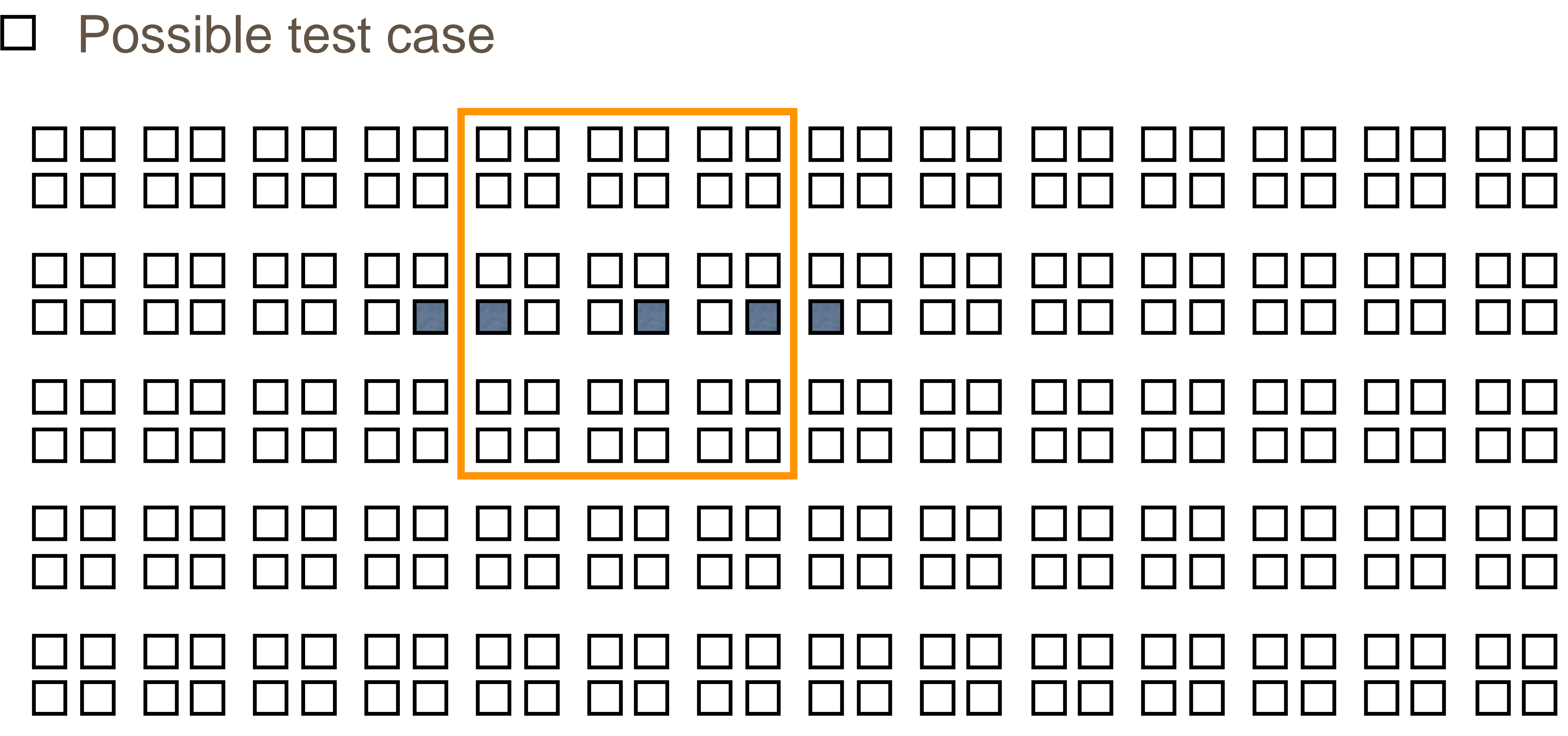
Functional testing is one way of drawing lines to isolate regions with likely failures

EQUIVALENCE PARTITIONING

Input condition	Equivalence classes
range	one valid, two invalid (larger and smaller)
specific value	one valid, two invalid (larger and smaller)
member of a set	one valid, one invalid
boolean	one valid, one invalid

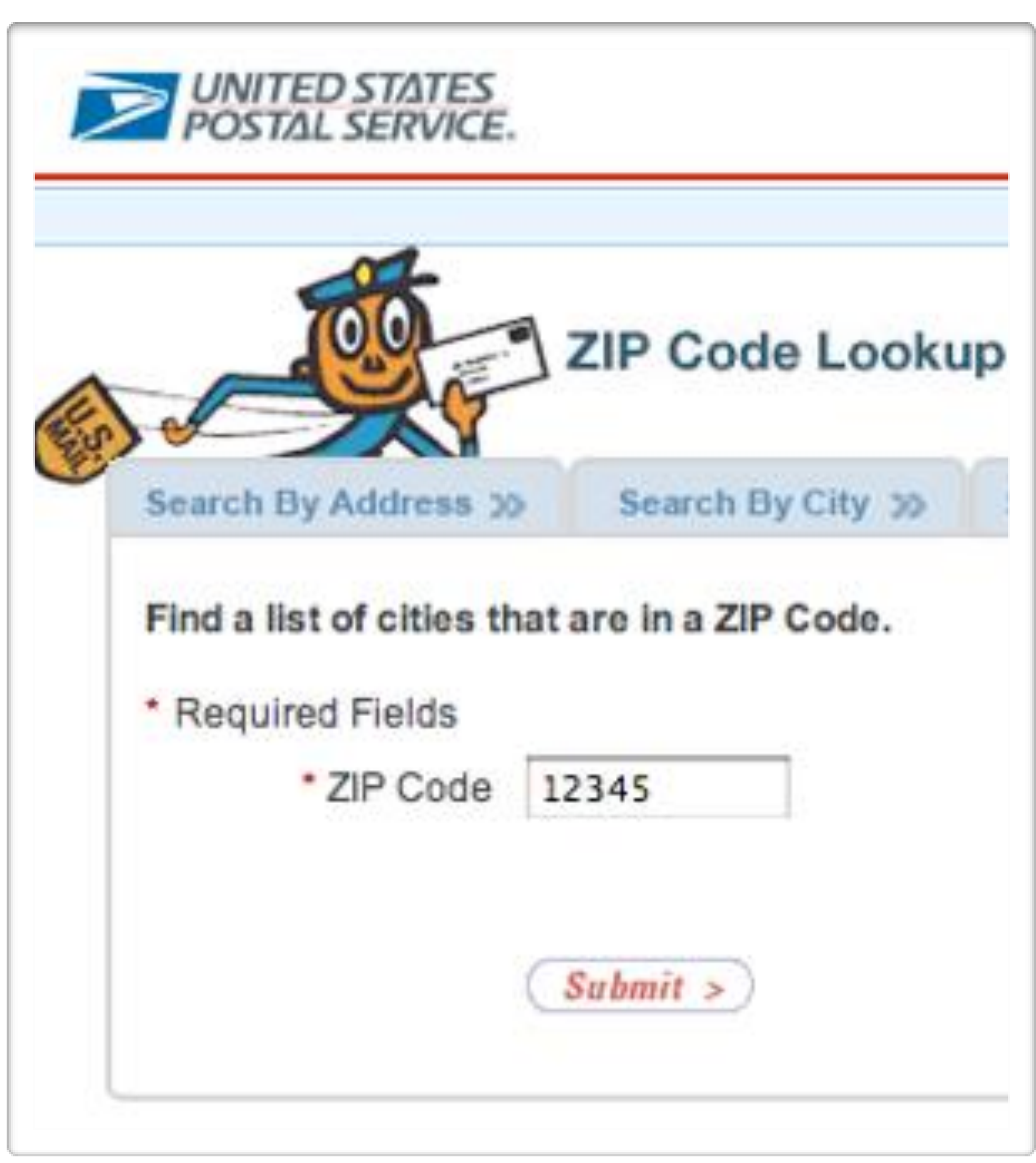
Defining equivalence classes comes from input conditions in the spec. Each input condition induces an equivalence class – valid and invalid inputs.

BOUNDARY ANALYSIS – FINDING ERROR AT THE EDGES



Test
- at *lower range* (valid and invalid)
- at *center*
- at *higher range* (valid and invalid)

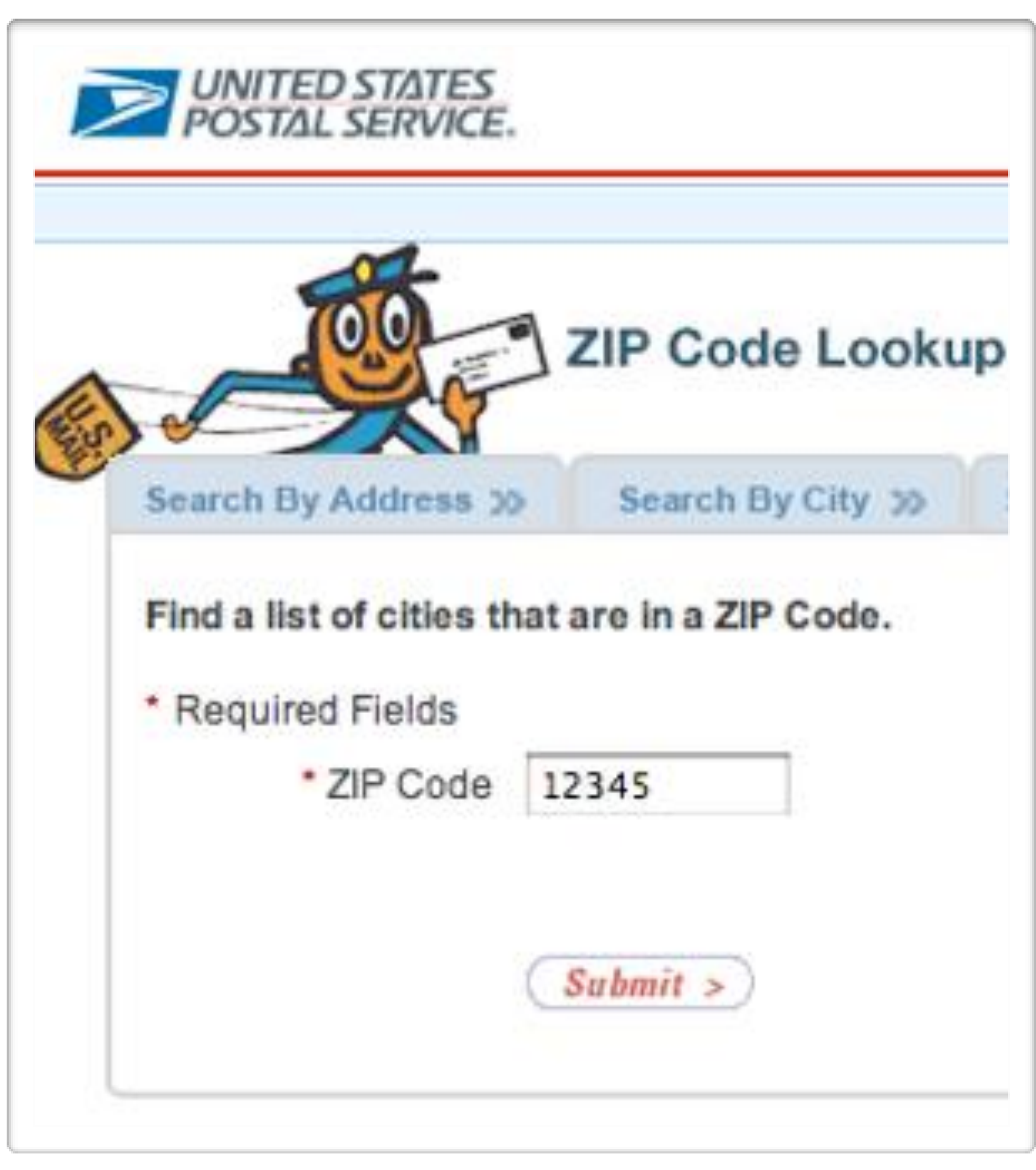
EXAMPLE: ZIP CODE



- Input: 5-digit ZIP code
- Output: list of cities

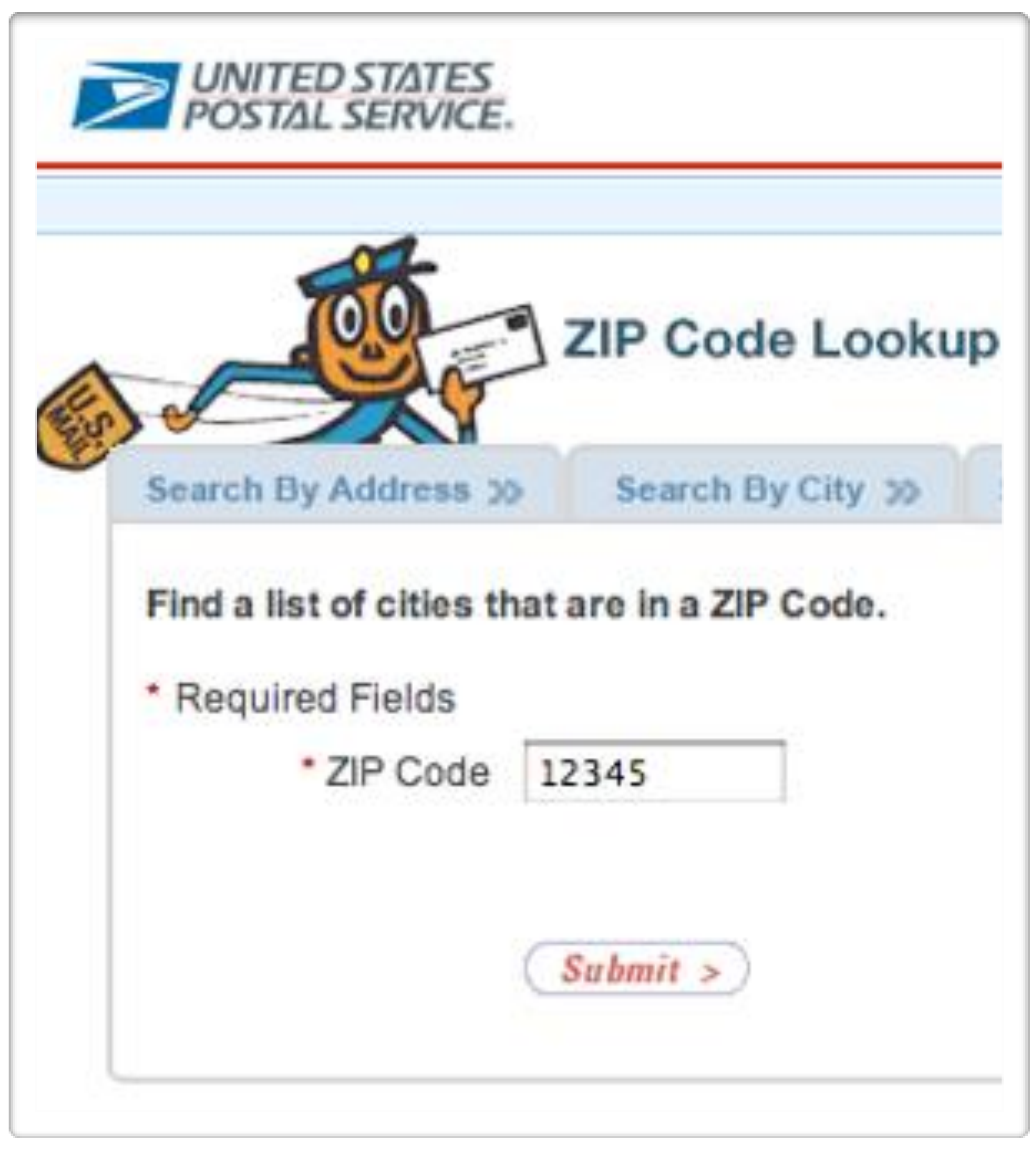
What are representative values to test?

VALID ZIP CODES



1. With 0 cities as output
   (0 is boundary value)
2. With 1 city as output
3. With many cities as output

INVALID ZIP CODES



4. Empty input
5. 1–4 characters
   (4 is boundary value)
6. 6 characters
   (6 is boundary value)
7. Very long input
8. No digits
9. Non-character data

“SPECIAL” ZIP CODES

1. How about a ZIP code that reads

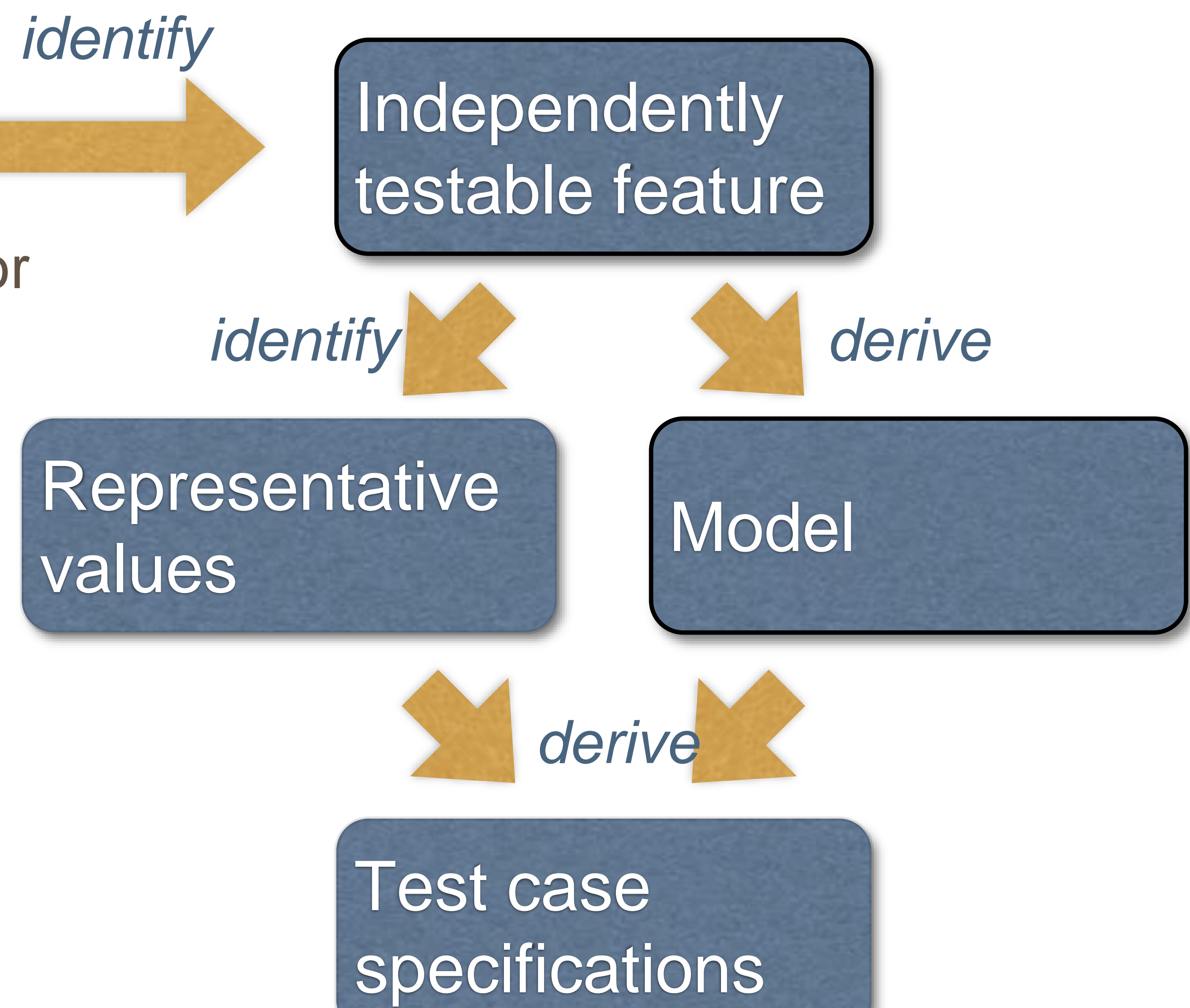
   ```
   12345‘; DROP TABLE orders; SELECT * FROM
   zipcodes WHERE ‘zip’ = ‘
   ```

2. A ZIP code with 65536 characters…

This is security testing

OR, YOU CAN USE MODELS TO DEFINE TESTS

- Use a formal model that specifies software behavior
- Models typically come as
  - finite state machines and
  - decision structures



FINITE STATE MACHINE FOR PRODUCT MAINTENANCE

Requirements

Maintenance: The *Maintenance* function records the history of items undergoing maintenance.

If the product is covered by warranty or maintenance contract, maintenance can be requested either by calling the maintenance toll free number, or through the Web site, or by bringing the item to a designated maintenance station.

If the maintenance is requested by phone or Web site and the customer is a US or EU resident, the item is picked up at the customer site, otherwise, the customer shall ship the item with an express courier.

If the maintenance contract number provided by the customer is not valid, the item follows the procedure for items not covered by warranty.

If the product is not covered by warranty or maintenance contract, maintenance can be requested only by bringing the item to a maintenance station. The maintenance station informs the customer of the estimated costs for repair. Maintenance starts only when the customer accepts the estimate. If the customer does not accept the estimate, the product is returned to the customer.

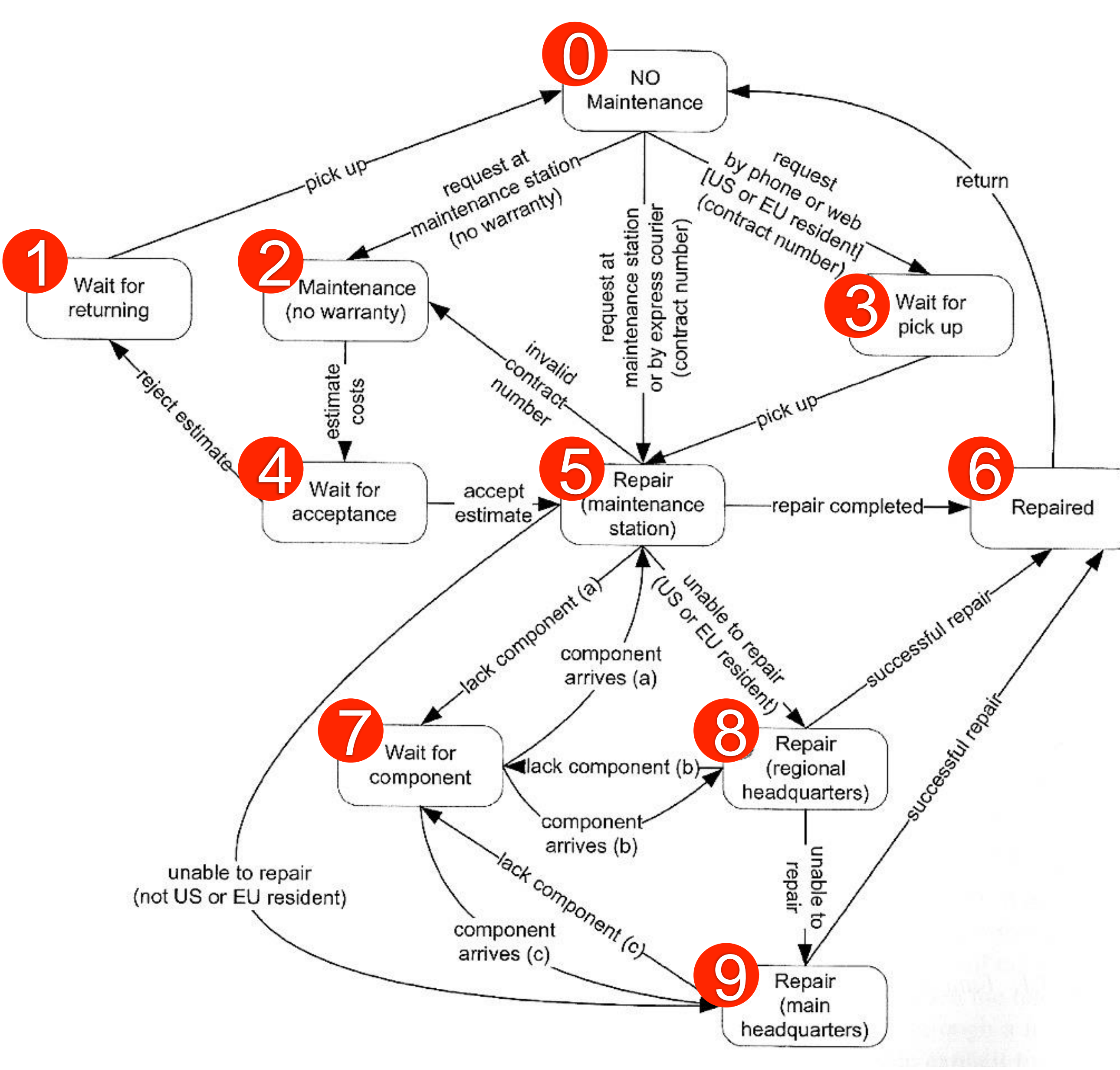
Small problems can be repaired directly at the maintenance station. If the maintenance station cannot solve the problem, the product is sent to the maintenance regional headquarters (if in US or EU) or to the maintenance main headquarters (otherwise).

If the maintenance regional headquarters cannot solve the problem, the product is sent to the maintenance main headquarters.

Maintenance is suspended if some components are not available.

Once repaired, the product is returned to the customer.

Representation



COVERAGE CRITERIA

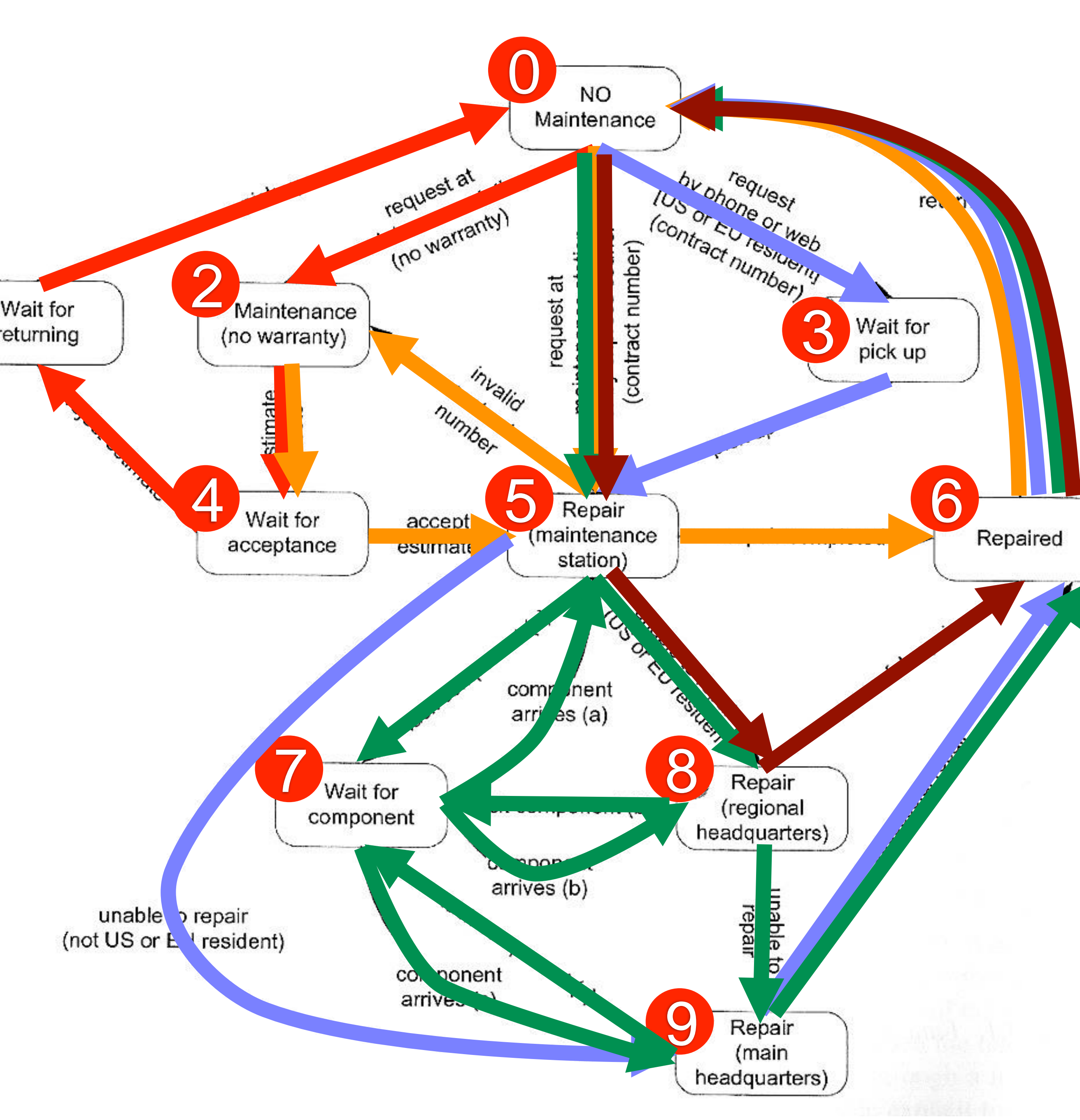
1. **Path coverage**: Tests cover every path
   - *Not feasible in practice*
     - *Cycles create infinite paths*
     - *Acyclic graphs can still have an exponential number of paths*
2. **State coverage**: Every node is executed
   - *A minimum testing criterion*
3. **Transition coverage**: Every edge is executed
   - *Typically, a good coverage criterion to aim for*

TRANSITION COVERAGE

Each test case covers a set of transitions

Here, there are five needed to cover each transition once

one color = one test case



STATE-BASED TESTING

- Protocols (e.g., network communication)
- GUIs (sequences of interactions)
- Objects (methods and states)

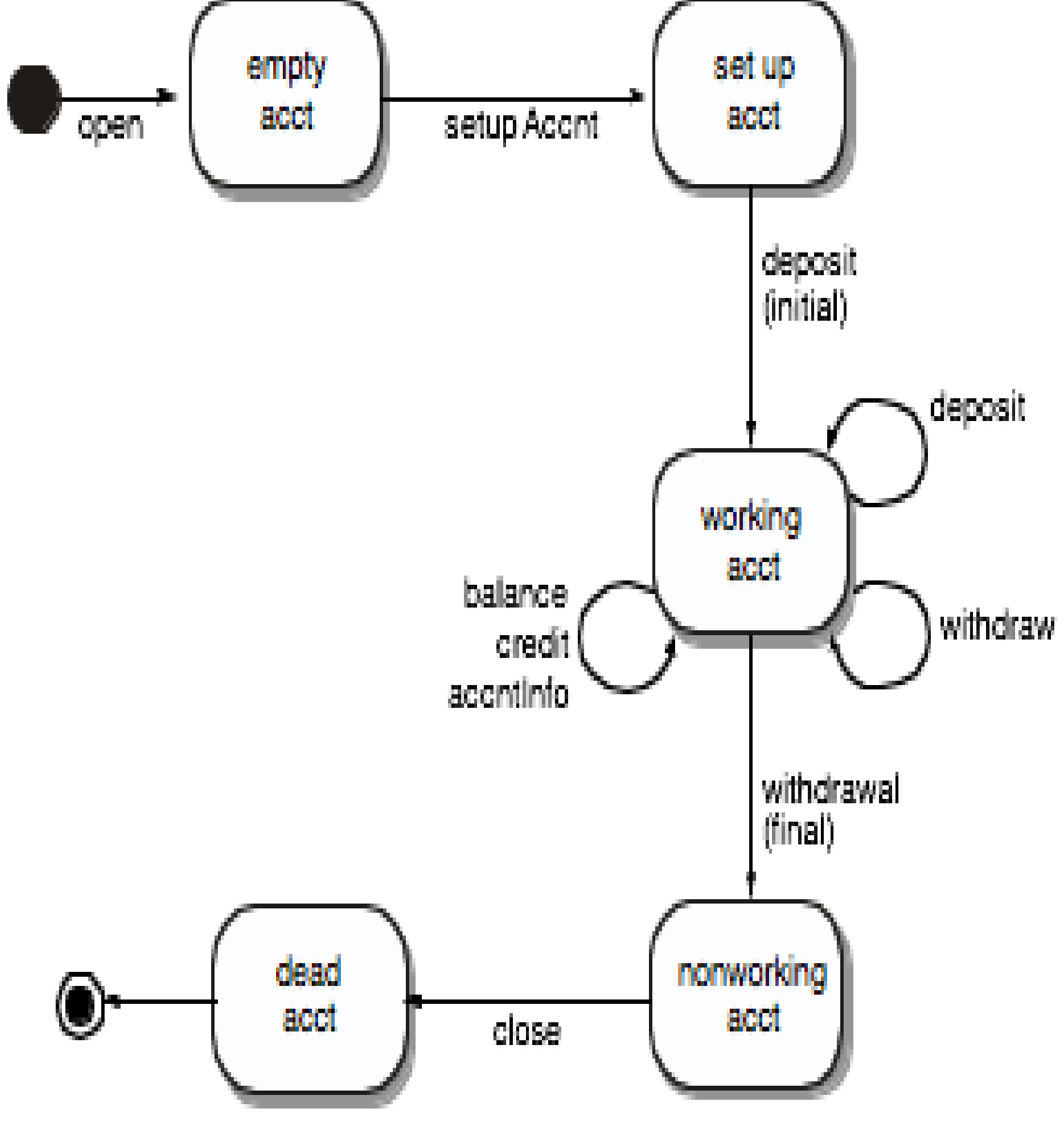


Figure 14.3 State diagram for Account class (adapted from [KIR94])

DECISION TABLES

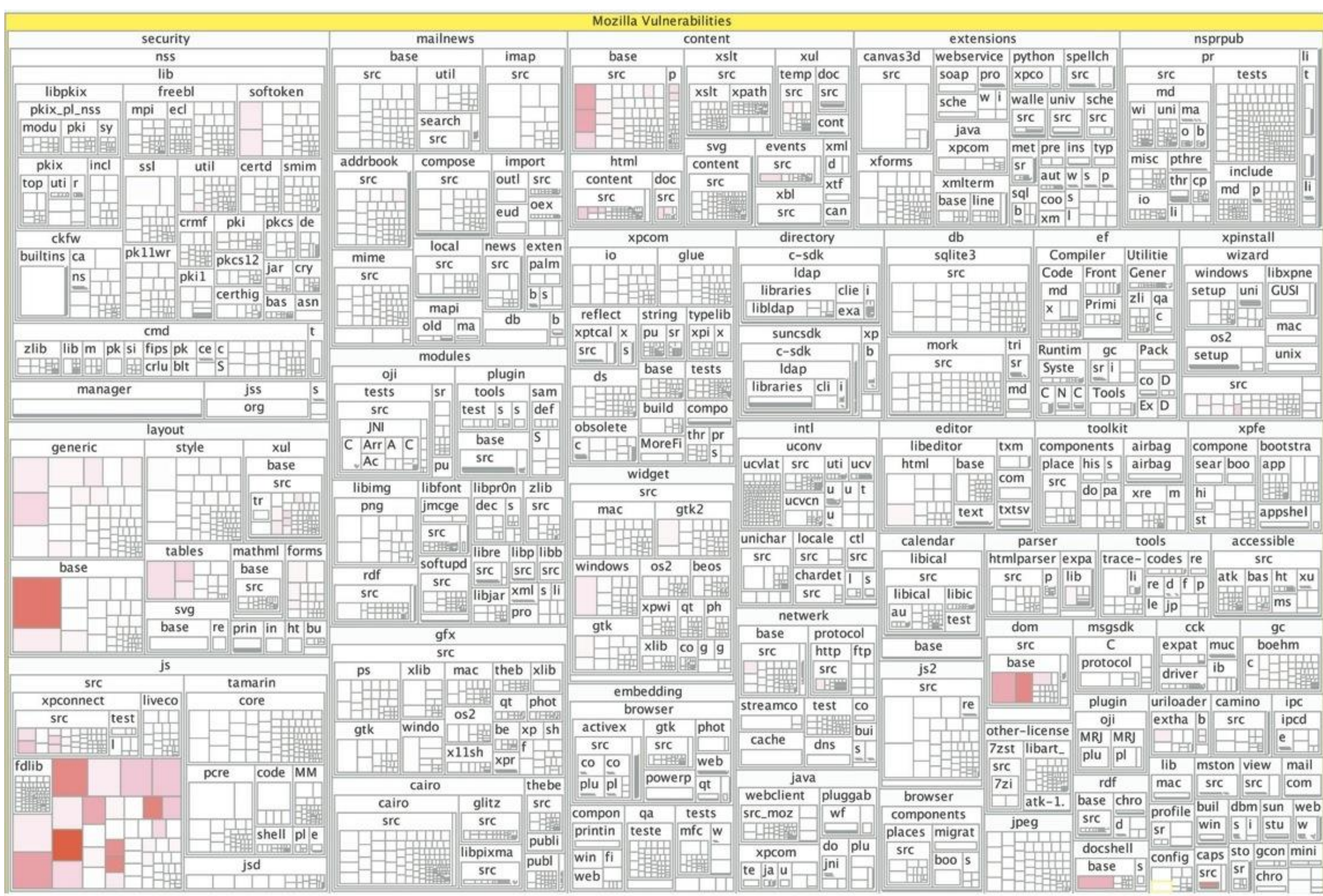
- Some specifications define decision tables, decision trees, or flow charts. We can define tests from these structures.

Type of Purchaser	Educational Purchaser		Individual Purchaser					
Education account	T	T	F	F	F	F	F	F
Current purchase > Threshold 1	–	–	F	F	T	T	–	–
Current purchase > Threshold 2	–	–	–	–	F	F	T	T
Special price < scheduled price	F	T	F	T	–	–	–	–
Special price < Tier 1	–	–	–	–	F	T	–	–
Special price < Tier 2	–	–	–	–	–	–	F	T
Outcome	*Edu discount*	*Special price*	*No discount*	*Special price*	*Tier 1 discount*	*Special price*	*Tier 2 discount*	*Special Price*

CONDITION COVERAGE

- **Basic Criterion**: each condition should be evaluated once using each possible setting
  - "Don't care" entries (–) can take arbitrary values
- **Compound Criterion**: Evaluate every possible combination of values for the conditions
- **Decision Coverage**: the expression should be evaluated once so it results in each possible outcome
- **Modified Condition/Decision Coverage (MC/DC)**
  - Each decision takes every possible outcome
  - Each condition in a decision takes every possible outcome
  - Each condition in a decision is shown to independently affect the outcome of the decision.
  - used in safety-critical avionics software
  - details in Pezze + Young, "Software Testing and Analysis", Chapter 14

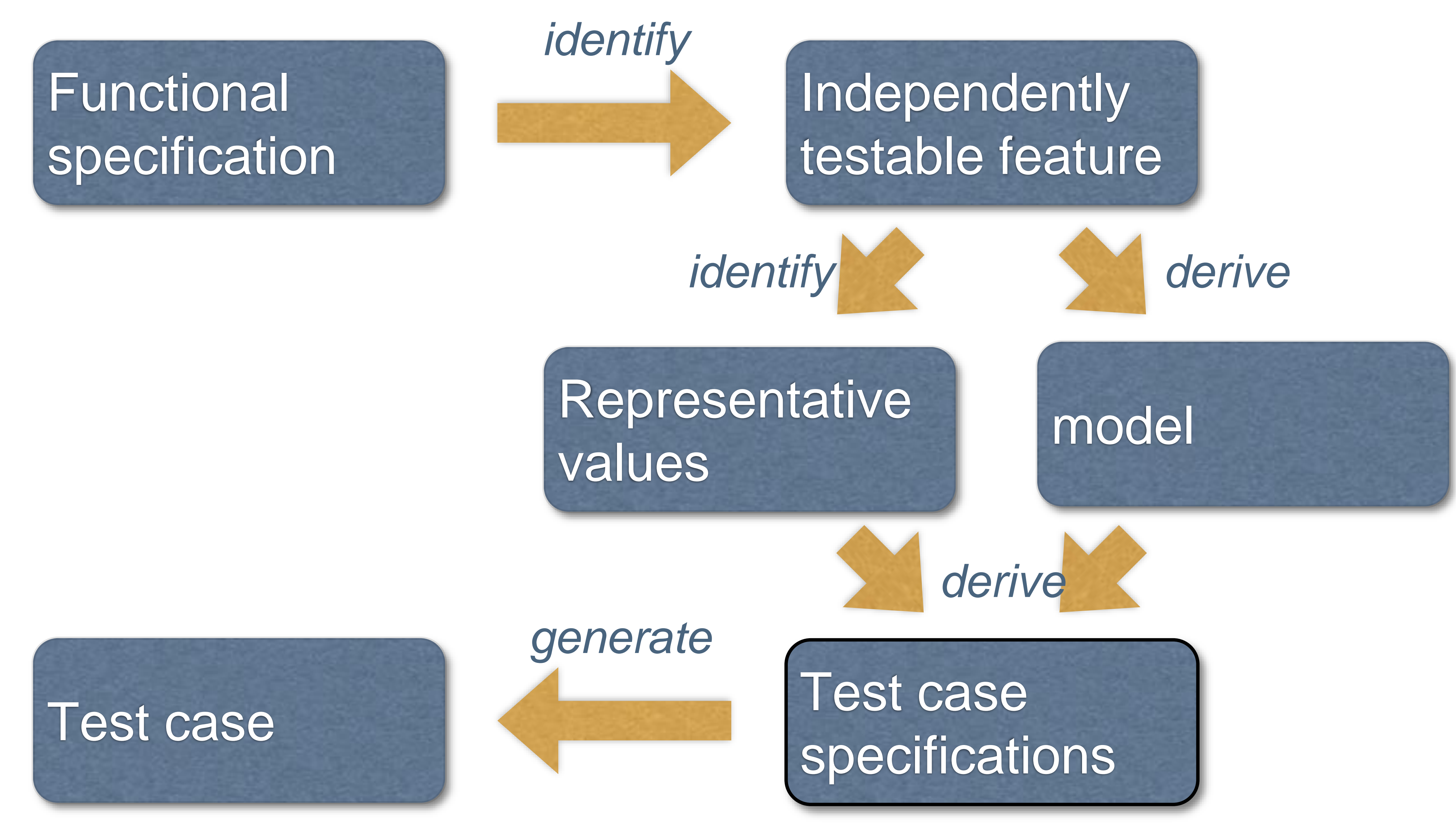
LEARNING FROM THE PAST



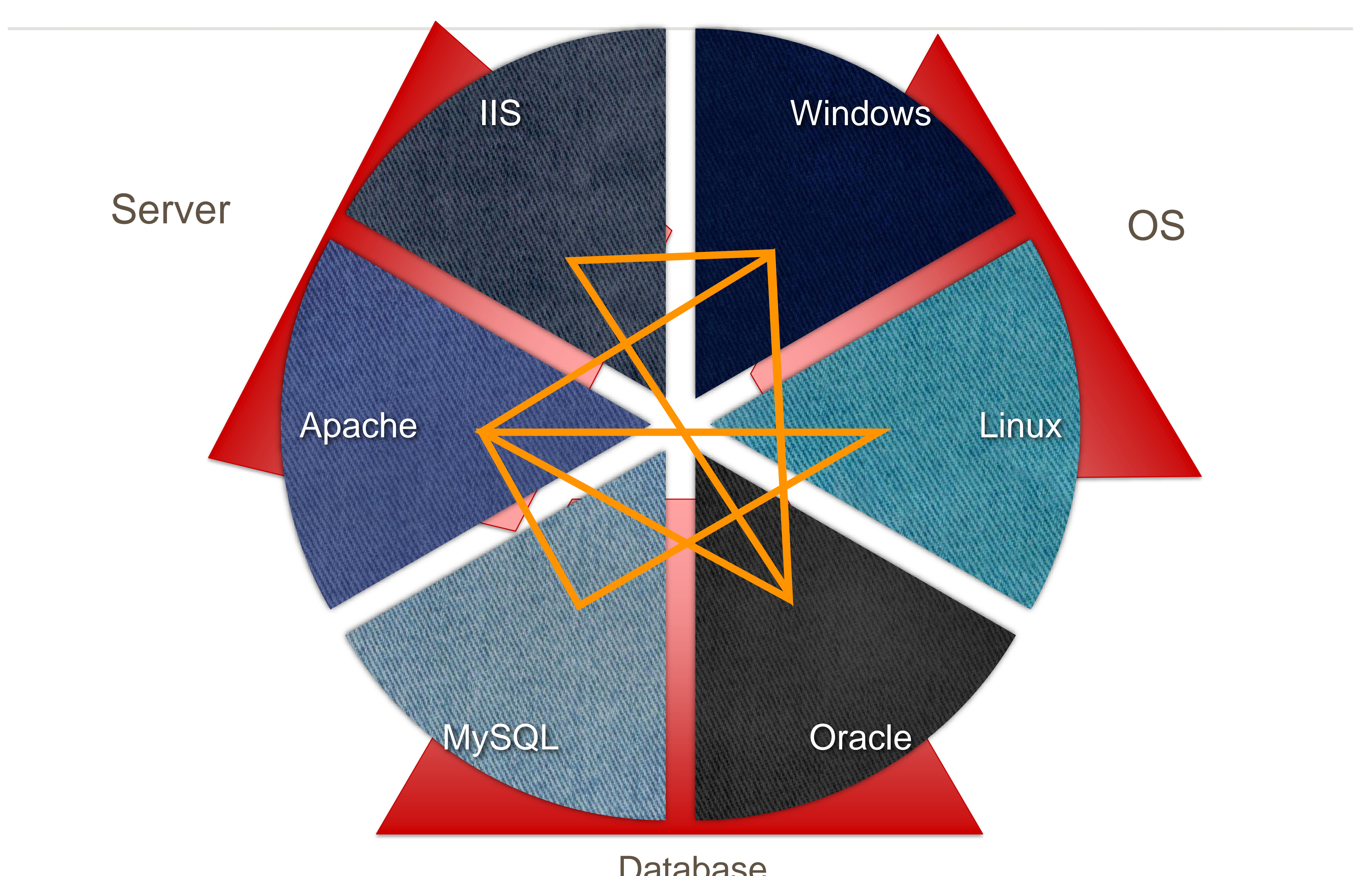
PARETO'S LAW

Approximately 80% of defects come from 20% of modules

DERIVING TEST SPEC'S



COMBINATORIAL TESTING



COMBINATORIAL TESTING

1. Eliminate invalid combinations
   - ✦ IIS only runs on Windows, for example
2. Cover all pairs of combinations
   such as MySQL on Windows and Linux
3. Combinations typically generated automatically
   and – hopefully – tested automatically, too

PAIRWISE TESTING MEANS TO COVER EVERY SINGLE PAIR OF CONFIGURATIONS

RUNNING A TEST

A test case…

1. *sets up an environment for the test*
2. *tests* the unit
3. *tears down* the environment again

Tests are organized into suites

TESTING A URL CLASS

JUNIT EXAMPLE

```
package junitexample;

public class Calculator {
    int add(int value1, int value2) {
        return value1 + value2;
    }
    int subtract(int value1, int value2) {
        return value1 - value2;
    }
    int multiply(int value1, int value2) {
        return value1 * value2;
    }
    int divide(int value1, int value2) {
        return value1 / value2;
    }
}
```

JUNIT, PART DEUX

```
package junitexample;

import junit.framework.TestCase;

public class CalculatorTest extends TestCase {
    private Calculator calc;
    public CalculatorTest(String s){
        super(s);
    }
    // called before each test
    protected void setUp() throws Exception {
        super.setUp();
        calc = new Calculator();
    }
    // called after each test
    protected void tearDown() throws Exception {
        super.tearDown();
    }
    …
```

```
…
    // test for the add() method
    public final void testAdd() {
        assertEquals(calc.add(20, 30), 50);
    }

    // test for the subtract() method
    public final void testSub() {
        assertEquals(calc.subtract(20, 10), 10);
    }

    // test for the multiply() method
    public final void testMult() {
        assertEquals(calc.multiply(9, 11), 99);
    }

    // test for the divide() method
    public final void testDiv() {
        assertEquals(calc.divide(18, 2), 9);
    }
}
```

JUNIT INTEGRATION IN ECLIPSE

Java - JUnitExample/src/jun

Packag | JUnit

Finished after 0.005 seconds

Runs: Errors: Failures:

- junitexample.CalculatorTest [Ru
  - testAdd (0.000 s)
  - testDiv (0.000 s)
  - testSub (0.000 s)
  - testMult (0.000 s)

Failure Trace

motivating-examples.tex | paper.tex | background.tex | code-form

```
21          super.tearDown();
22      }
23
24      // test for the add() method
25      public final void testAdd() {
26          assertEquals(calc.add(20, 30), 50);
27      }
28
29      // test for the subtract() method
30      public final void testSub() {
31          assertEquals(calc.subtract(20, 10), 10);
32      }
33
34      // test for the multiply() method
35      public final void testMult() {
36          assertEquals(calc.multiply(9, 11), 99);
37      }
38
39      // test for the divide() method
40      public final void testDiv() {
41          assertEquals(calc.divide(18, 2), 9);
42      }
```

TEST-DRIVEN DEVELOPMENT

- writing tests before you implement functionality involves extra effort, but…
- … it forces you to think about the problem you are trying to solve more concretely
  - and formulate a solution more quickly
- …and you will regain the time spent on unit tests by catching problems early
  - and reduce time spent later on debugging

RECOMMENDATIONS FOR WRITING GOOD TESTS

- write tests that cover a partition of the input space, and that cover specific features
- achieve good code coverage
- create an automated, fast running test suite, and use it all the time
- have tests that cover your system's tests at different levels of functionality
- set up your tests so that, when a failure occurs, it pinpoints the issue so that it does not require much further debugging

EXTRA

TESTING ENVIRONMENTS ARE OFTEN COMPLEX

- Millions of configurations
- Testing on dozens of different machines
- All needed to find and reproduce problems

DEFECT SEVERITY

- An assessment of a defect's impact
- Can be a major source of contention between dev and test

Critical **Show stopper.** The functionality cannot be delivered unless that defect is cleared. It does not have a workaround.

Major Major flaw in functionality but it still can be released. There is a workaround; but it is not obvious and is difficult.

Minor Affects minor functionality or non-critical data. There is an easy workaround.

Trivial Does not affect functionality or data. It does not even need a workaround. It does not impact productivity or efficiency. It is merely an inconvenience.